Contributions to handwriting recognition using deep neural networks and quantum computing

Bogdan-Ionut Cirstea

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2018-ENST-0059

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité Signal et Images

présentée et soutenue publiquement par

Bogdan-Ionuţ CÎRSTEA

le 17 décembre 2018

Contributions à la reconnaissance de l'écriture manuscrite en utilisant des réseaux de neurones profonds et le calcul quantique

Directrice de thèse :
Laurence LIKFORMAN-SULEM

Jury

Thierry PAQUET, Professeur, Université de Rouen	Rapporteur
Christian VIARD-GAUDIN, Professeur, Université de Nantes	Rapporteur
Nicole VINCENT, Professeur, Université Paris Descartes	Examinateur
Sandrine COURCINOUS, Expert, Direction Générale de l'Armement (DGA)	Examinateur
Laurence LIKFORMAN-SULEM, Maître de conférence, HDR Télécom ParisTech	Directrice de thèse

T H È S E

TELECOM ParisTech
école de l'Institut Mines-Télécom - membre de ParisTech
46 rue Barrault 75013 Paris - (+33) 1 45 81 77 77 - www.telecom-paristech.fr

Acknowledgments

I would like to start by thanking my PhD advisor, Laurence Likforman-Sulem. Laurence was always there to provide help and moral support, especially in the most difficult moments of this thesis project. This thesis would not have been completed without her support.

I am very grateful to Télécom Paristech and the DGA (Direction Générale de l'Armement), who co-financed this thesis through a DGA-MRIS scholarship and, particularly, to Sandrine Courcinous, who accepted to review this thesis, and to Emmanuèle Grosicki, who initiated this project.

I would also like to thank NVIDIA Corporation for the donation of the Tesla K40 GPU which was used to perform this research.

Many thanks to the jury, who accepted to review this work, which is not necessarily in the comfort zone of most handwriting recognition researchers.

Finally, I would like to thank my love, Sabina, my friends and my family for their continuous and unwavering support.

Résumé

Au cours des dernières années, l'apprentissage profond, le domaine d'étude des réseaux de neurones artificiels à couches multiples, a eu un impact important sur de nombreux domaines de l'intelligence artificielle, y compris la reconnaissance de l'écriture manuscrite. Le calcul quantique, en revanche, malgré une longue histoire, n'a été étudié que très récemment pour les applications d'apprentissage automatique. Dans cette thèse, nous fournissons plusieurs contributions des domaines de l'apprentissage profond et du calcul quantique à la reconnaissance de l'écriture manuscrite.

Nous commençons par intégrer certaines des techniques d'apprentissage profond les plus récentes (comme dropout, batch normalization et différentes fonctions d'activation) dans les réseaux de neurones à convolution et obtenons des meilleures performances sur le fameux jeu de données MNIST. Nous proposons ensuite des réseaux TSTN (Tied Spatial Transformer Networks), une variante des réseaux STN (Spatial Transformer Networks) avec poids partagés, ainsi que différentes variantes d'entraînement du TSTN. Nous présentons des performances améliorées sur une variante déformée du jeu de données MNIST. Dans un autre travail, nous comparons les performances des réseaux récurrents de neurones Associative Long Short-Term Memory (ALSTM), une architecture récemment introduite, par rapport aux réseaux récurrents de neurones Long Short-Term Memory (LSTM), sur le jeu de données de reconnaissance d'écriture arabe IFN-ENIT. Enfin, nous proposons une architecture de réseau de neurones que nous appelons réseau hybride classique-quantique, capable d'intégrer et de tirer parti de l'informatique quantique. Alors que nos simulations sont effectuées à l'aide du calcul classique (sur GPU), nos résultats sur le jeu de données Fashion-MNIST suggèrent que des améliorations exponentielles en complexité computationnelle pourraient être réalisables, en particulier pour les réseaux de neurones récurrents utilisés pour la classification de séquence.

Abstract

During the last past years, deep learning, the field of study of artificial neural networks with multiple layers, has had a strong impact on many fields of artificial intelligence, including handwriting recognition. Quantum computation, on the other hand, despite having a history spanning decades, has only very recently been studied for machine learning applications (arguably, for several years only). In this thesis, we provide several contributions from the fields of deep learning and quantum computation to handwriting recognition.

We begin by integrating some of the more recent deep learning techniques (such as dropout, batch normalization and different activation functions) into convolutional neural networks and show improved performance on the well-known MNIST dataset. We then propose Tied Spatial Transformer Networks (TSTNs), a variant of Spatial Transformer Networks (STNs) with shared weights, as well as different training variants of the TSTN. We show improved performance on a distorted variant of the MNIST dataset. In another work, we compare the performance of Associative Long Short-Term Memory (ALSTM), a recently introduced recurrent neural network (RNN) architecture, against Long Short-Term Memory (LSTM), on the Arabic handwriting recognition IFN-ENIT dataset. Finally, we propose a neural network architecture, which we denote as a hybrid classical-quantum neural network, which can integrate and take advantage of quantum computing. While our simulations are performed using classical computation (on a GPU), our results on the Fashion-MNIST dataset suggest that exponential improvements in computational requirements might be achievable, especially for recurrent neural networks trained for sequence classification.

Table of Contents

List of Figures

List of Tables

Introduction

During the last past years, deep learning, the field of study of artificial neural networks with multiple layers, has had a strong impact on many fields of artificial intelligence, including handwriting recognition. Quantum computation, on the other hand, despite having a history spanning decades, has only very recently been studied for machine learning applications (arguably, for several years only). An important reason for why quantum machine learning is in its infancy stage is the lack of large scale, practical quantum computers.

In this thesis, we provide several contributions from the fields of deep learning and quantum computation to handwriting recognition.

In Chapter 1, we will shortly introduce the most relevant methods and techniques from deep learning (and, more generally, machine learning) for the work described in this thesis.

In Chapter 2, we will discuss the relationship between deep learning and handwriting recognition from two different, but closely-related perspectives. First, we will present the historical role of handwriting recognition tasks and datasets in the development and the rise to prominence of neural networks. We will then discuss the state of the art in handwriting recognition, focusing on neural network architectures.

In Chapter 3, we present our work integrating some of the more recent (particularly at the time the work was performed) deep learning techniques (such as dropout, batch normalization and different activation functions) into convolutional neural networks and show improved performance on the well-known MNIST dataset.

In Chapter 4, we describe Tied Spatial Transformer Networks (TSTNs), a variant of the previously introduced Spatial Transformer Networks (STNs) with shared weights. We also describe different training variants of the Tied Spatial Transformer Networks and show how we obtain improved performance on a distorted variant of the MNIST dataset.

In Chapter 5, we compare the performance of Associative Long Short-Term Memory (ALSTM), a recently introduced recurrent neural network (RNN) archi-

tecture, against Long Short-Term Memory (LSTM), on the Arabic handwriting recognition IFN-ENIT dataset.

Finally, in Chapter 6, we shortly describe the basic principles of quantum computation we make use of and propose a neural network architecture which can integrate and take advantage of quantum computing. While our simulations are performed using classical computation (on a GPU), our results on the Fashion-MNIST dataset suggest that exponential improvements in computational requirements might be achievable, especially for recurrent neural networks trained for sequence classification.

We conclude with a discussion about the ultimate potential of neural networks to solve handwriting recognition and how quantum computation might aid in this effort, as a step towards human-level AI.

Chapter 1

Introduction to deep learning

Contents

In this chapter, we provide a brief introduction to the main deep learning techniques we have used in this thesis. A much more thorough introduction to deep

learning is provided by [87]. For briefer reviews, see [131], [168]. We will first provide a short introduction to machine learning (of which deep learning is a subdomain), followed by a presentation of the main computational primitives used in this thesis to create deep learning systems (practically all the most significant deep learning systems are composed of two basic types of operations: linear and nonlinear). We then introduce the main types of neural network architectures: multilayer perceptrons, convolutional neural networks and recurrent neural networks. These can be used to address different types of problems and can be combined creatively. We then introduce the main performance measures used to benchmark handwriting recognition systems: the label error rate (whose complement is the accuracy), character error rate (CER) and word error rate (WER). In the following section we present gradient-based optimization, introducing loss functions, gradient descent (with backpropagation and backpropagation through time) and some state of the art optimization algorithms (like ADAM) and regularization methods (like dropout).

1.1 Short introduction to machine learning

In this section we very briefly introduce the main concepts in the field of machine learning, of which deep learning is a subfield. This section is inspired by the presentation in [11].

The machine learning field and particularly neural networks, the main subject of this thesis, use an approach denoted as *data-driven*. This approach can be intuitively described in the following manner. Suppose we want to recognize the handwriting in a particular image. Unlike in most of computer science, we don't know how to explicitly write an algorithm which can achieve this. Even when a researcher is able to recognize the handwriting in a particular image, he is unable to consciously access *how* his brain performs the recognition, so as to directly translate the process into a programmable algorithm. Instead, we will first *train* an algorithm on a dataset. We will also use separate datasets to *validate* our approach and test what performance we could expect in the real world. The validation dataset can be used to calibrate the algorithm. Ideally, the test dataset is only touched once, to benchmark the final performance of the algorithm.

Three different learning paradigms are the best-known and most relevant in machine learning. In the case of *supervised learning*, which will be by far the most relevant for this thesis, the datasets are labeled and the algorithm is expected to learn the mapping $f : \text{input} \rightarrow \text{label}$. Another paradigm is unsupervised learning,

where only inputs are provided (no labels) and the algorithm is expected to learn to generate data similar to the inputs in the training set. Finally, reinforcement learning uses a process similar to how animals are trained: the algorithm is expected to learn how to act in a certain environment so as to maximize a cumulative reward.

Handwriting recognition fits in the supervised learning paradigm. In this task, an array of pixels representing an image is provided as an input and the algorithm has to learn to assign it a sequence of labels (in the most general case). The handwriting recognition pipeline can be formalized in the following manner:

1. **Input**: a set of N images, each labeled with a sequence of labels; each label belongs to an alphabet of K symbols. This data is denoted as the *training set*.

2. **Learning**: We use the training set to learn the mapping between images of handwriting and the corresponding label sequence. This step is denoted as *learning a model*.

3. **Evaluation**: The quality of the model is evaluated by asking it to predict the sequences of labels for a new set of images (the test set) never seen before and comparing the predictions to the true label sequences (ground truth). The intuition behind this procedure is that the performance on the test set should be indicative of the performance in the real world (after deployment).

The validation set is used to perform what is called **hyperparameter tuning**. Hyperparameters are parameters whose values are usually chosen using a different procedure than that used for the trainable parameters. For neural networks, for example, some examples of hyperparameters are the number of layers, the number of units (neurons), the learning rate of the stochastic gradient procedure (see Section 1.5). In the case of neural networks, while trainable parameters are usually modified using stochastic gradient descent, the hyperparameters are usually chosen by trying out different values on the validation set and settling on the value which works best. It is essential not to use the test set for tweaking the hyperparameters, so as not to risk **overfitting** to the test set. Overfitting to the test set intuitively means that the model obtains much better performance than what would actually be observed during deployment, for real-world data. Using the test set to tune hyperparameters can be interpreted as effectively including it into the training set, so the performance reported on the test set would be too optimistic compared to what can be expected when the model is deployed. On the other hand, if the test set is only used once, at the very end, it can be expected to provide a useful measure of how well the model

will generalize to real data. For these reasons, we should only evaluate on the test set once, at the very end.

The validation set, which can be used for tweaking the hyperparameters, can be obtained by splitting the training set into a slightly smaller training set, and using the rest of the data as the validation set. Intuitively, the validation set is used like a fake test set to tune the hyperparameters. For many datasets, and particularly those most used for benchmarking, the splits into different training and validation sets are already provided. This can aid with easier comparisons of the performance obtained by different algorithms.

1.2 Computational primitives

In their most general form, neural networks can be interpreted as computational graphs composed of primitive operations. The computational graphs allow for a richer set of primitive operations than those we described here, but we will restrict ourselves to the most commonly used and most successful operations.

1.2.1 Matrix vector multiplication

Matrix vector multiplication is probably the most widely-used deep learning computational primitive. It is a type of linear operation (no nonlinear effect) and is used as a component in all the most successful neural network architectures, including those used in this thesis: multilayer perceptrons, convolutional neural networks and recurrent neural networks..

In the most common setting, the vector $x \in \mathbb{R}^n$ represents information previously processed by the neural network and / or unprocessed (input) information. The matrix $W \in \mathbb{R}^{m*n}$ is dense and all of its entries are modifiable (trainable) - see Subsection 1.5. The result of this operation is the vector $y \in \mathbb{R}^m$:

$$y = W * x \tag{1.1}$$

Usually, a vector of biases $b \in \mathbb{R}^m$ is added to the matrix vector multiplication result, so that the previous equation becomes:

$$y = W * x + b \tag{1.2}$$

We can obtain the same result by appending the value 1 to the end of the column vector x and the column b to the matrix W, so that we can keep Eq. 1.1.

In the next subsections we will introduce various primitives which can be interpreted as matrix vector multiplications, where the matrices are factorized in different manners.

1.2.1.1 Element-wise multiplication

Element-wise multiplication can be interpreted as matrix vector multiplication where the matrix is diagonal. Here again, the vector $x \in \mathbb{R}^n$ represents information previously processed by the neural network and / or unprocessed (input) information, while the diagonal matrix $D \in \mathbb{R}^{n*n}$ contains the trainable parameters. By reshaping the diagonal matrix to the vector $d \in \mathbb{R}^n$:

$$y = D * x = d \bullet x \tag{1.3}$$

Biases can be added analogously to how this was performed in the previous subsection.

Element-wise multiplication is notably used in state of the art Long Short-Term Memory (LSTM - see Section 1.3.3.2) recurrent neural networks. The convolution operation, a computational primitive we describe in next, can also be factorized as Discrete Fourier Transforms and element-wise multiplications, but we won't go into more details here (see [9]).

1.2.1.2 Convolution

The convolution operation is widely used in state of the art neural networks architectures, particularly Convolutional Neural Networks (CNNs). We will only describe discrete convolution applied to functions with finite support (vectors in the 1D case and images in the 2D case), as these are the most relevant for neural networks. Discrete convolution can be reformulated as matrix vector multiplication [8].

We will consider two vectors f and g with indices in $\{0..., N-1\}$. The result of their 1D convolution can be written as:

$$o[n] = f[n] \circledast g[n] = \sum_{u=0}^{u=N-1} f[n-u] * g[u] \tag{1.4}$$

where $\circledast$ denotes the convolution operator.

This can be extended to 2D arrays as follows:

$$o[m,n] = f[m,n] \circledast g[m,n] = \sum_{u=0}^{u=M-1} \sum_{v=0}^{v=N-1} f[m-u, n-v] * g[u,v] \tag{1.5}$$

We will describe the intuition behind the convolution operation when we discuss Convolutional Neural Networks in Section 1.3.2.

1.2.1.3 Factorized matrix vector multiplication

The previously introduced matrix vector multiplication is the most popular linear operation in neural networks, but it can be quite expensive in computational time and memory. For a vector of size N and a matrix of size $N \times N$, the computational and memory complexity are $O(N^2)$. In this subsection we briefly present several methods which factorize the matrix in the matrix vector multiplication, in order to reduce the computational and / or memory complexity. We introduce a similar approach, which factorizes the matrix (and uses quantum computation) in Chapter 6. For this reason, we focus here on some of the methods most similar and relevant to our own.

[147] replaced the W matrix in fully-connected layers with the matrix product $A * C * D * C$, with A and D diagonal matrices, C the discrete cosine transform and C^{-1} the inverse discrete cosine transform, reducing the computational complexity of a to $O(N * \log(N))$ and the number of trainable parameters to $O(N)$, while maintaining comparable statistical performance for the task of object recognition on the ImageNet dataset.

[38] proposed a similar factorization, with $O(N*\log(N))$ computational complexity and $O(N)$ trainable parameters hidden-to-hidden transform of a recurrent neural network (see Section 1.3.3.1). The resulting transform is the product of multiple unitary matrices, some of which represent the Discrete Fourier Transform and the Inverse Discrete Fourier Transform. The RNN parameterization obtained state of the art results at the time of the proposal on several long-term dependency tasks. Our proposal in chapter 6 also decomposes the matrix W into a product of multiple unitary matrices, some of which represent Fourier transforms, but potentially reduces the computational and memory complexities even further, due to the use of quantum computation.

[121] introduced a Kronecker parameterization of the matrix implementing the hidden-to-hidden transform of a recurrent neural network (see Section 1.3.3.1), showing that, at least for certain tasks, the number of parameters in the hidden-to-

hidden part of the RNN can be drastically reduced, from $O(N^2)$ to $O(N * \log(N))$ and the computational complexity is reduced. In chapter 6, we propose a neural network architecture which makes use of quantum computation and can also dramatically reduce the number of trainable parameters, but also the computational complexity (to $O(\log(N)^2)$), while maintaining comparable performance, using a similar Kronecker matrix factorization.

1.2.2 Nonlinear activation functions

All previously described computational primitives are linear. Nonlinear operations are also necessary, otherwise a machine learning system containing only linear operations would not be expressive enough, no matter how many linear operations were composed. Intuitively, no matter how many linear operations are composed, the entire system is no more powerful than a simple linear regression. On the other hand, even the composition with a single nonlinear operation makes neural networks universal approximators of continuous functions [73].

In neural networks, nonlinearity is introduced using the concept of an *activation function*, which is applied element-wise to the input.

Historically, the most popular activation function used to be the sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1.6}$$

Another activation function with a long history is the tanh function:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{1.7}$$

Currently, one of the most successful activation functions is the Rectified linear unit (ReLU) [86]:

$$\mathrm{ReLU}(x) = \begin{cases} x, & \text{if } x > 0. \\ 0, & \text{otherwise.} \end{cases} \tag{1.8}$$

The interest of ReLU activation functions is that they provide less vanishing gradients because they saturate less (as compared to the logistic function)

Leaky rectified linear units (LReLUs) [140] have been found to either match or surpass ReLUs in performance by some authors [140] [200]:

$$\text{LReLU}(x) = \begin{cases} x, & \text{if } x > 0. \\ ax, & \text{otherwise.} \end{cases} \tag{1.9}$$

a is the scaling factor and is fixed. Parametrized Rectified Linear Units (PReLU) [105] is another rectified activation function, with the same equation as for LReLU. The difference, though, is that a is a trainable parameter (through gradient-based optimization). A different a can be chosen for each neuron or the a values can be 'tied' so that several neurons (see the next section for an introduction to the concept of artificial neuron) share a same value. This helps reduce the number of trainable parameters and, thus, can prevent overfitting.

The softmax nonlinearity is commonly used in the setting of supervised learning with discrete labels (classes). In this setting, it is most often combined with the cross-entropy loss function (see Subsection 1.5.1.1). Given K distinct classes, the softmax takes as input a vector x of K real values (this vector can be obtained from a vector of inputs or previously processed values through e.g. matrix vector multiplication). The softmax then processes these K values to provide the probabilities corresponding to the K classes. For each class $k \in 1, .., K$, its corresponding probability is:

$$p(k) = \frac{e^{x_k}}{\sum_{j=1}^{K} e^{x_j}} \tag{1.10}$$

1.3 Artificial neural network architectures

Artificial neural networks (ANNs) are the main topic of this thesis. As their name suggests, ANNs represent machine learning architectures (very loosely) inspired by their biological counterparts.

The basic computational unit in ANNs is the artificial neuron. The first celebrated model of an artificial neuron was introduced in 1943 by [144]. We will denote the output of a neuron's computation by the term *activation*. In biology, neurons are connected by synapses with varying strengths (weights) and each neuron has a threshold (whose electrical activity, which could be roughly interpreted as the activation in our description, has to be greater than for the neuron to fire). Artificial neurons have corresponding (and simplified) weights w connecting them to other neurons and bias b (the simplified equivalent of the biological threshold). To connect this model to the machine learning framework, the weights w and bias b are trainable parameters (whose values are optimized during the learning process).

The most basic widely-encountered modern artificial neurons (as they appear

especially in multilayer perceptrons, which will be described in the following subsection) compute a dot product, followed by an element-wise nonlinearity (such as those described in Section 1.2.2). Given a set of inputs x, the activation h of a neuron connected to the inputs using weights w and bias b is:

$$h = f(w * x + b) \tag{1.11}$$

where f is the element-wise nonlinearity.

This is a highly simplified model of biological neurons, which display many more features e.g. spiking, dendrites, etc. [3].

To distinguish the artificial neurons from their much more complex biological counterparts, they are often denoted by the term *units*. In this thesis, we will mostly use the term units and whenever we use the term neurons, artificial neurons are implied.

The basic structural element of an ANN is a *layer* containing multiple units. The first and the last layer are somewhat special, containing the ANN's inputs and, respectively, outputs. All the other layers are denoted as *hidden layers*.

Artificial neural networks come in many flavors (architectures), especially since the resurgence of neural network research during the last decade. We will present in this chapter the three most important architectures: the multilayer perceptron (MLP), the convolutional neural network (CNN) and the recurrent neural network (RNN). MLPs and CNNs are also called feedforward neural networks (because they don't use recurrence in their computation), differently from RNNs. These architectures are based on somewhat different computational primitives. Briefly, they were introduced to deal with different problems: MLPs with general single inputs without any particular structure, CNNs with visual inputs (particularly images) and RNNs with temporal sequences.

As will be highlighted in Chapter 2, these architectures can be mixed and composed according to the problem we need to solve. A widely-used analogy is that of neural network layers as lego blocks, which can be assembled creatively to solve new problems or to improve performance on old ones. Indeed, much (arguably, most) of the research in neural networks involves designing new layers (lego blocks) and / or assembling them into new architectures appropriate for the problems we're trying to solve.

1.3.1 Multilayer perceptrons (MLPs)

Multilayer perceptrons (MLPs) also appear in the literature by the name Deep Neural Networks (DNNs), somewhat ambiguously (since other neural network architectures with multiple layers can also be called deep neural networks).

We will briefly describe each type of layer in the MLP.

1.3.1.1 Input layer

The input layer represents the input as a vector and is connected to the first hidden layer. To make the notation uniform with that of the hidden layers, the input layer could be denoted by h_0 and the first hidden layer as h_1.

1.3.1.2 Hidden layers

In their most common variant, each MLP hidden layer $i \in \{1, ..M\}$ is a vector of units h_i connected to the previous layer h_{i-1} by the weight matrix W_i and vector of biases b_i. The corresponding equation is:

$$h_i = f(W_i * h_{i-1} + b_i) \tag{1.12}$$

where f is an element-wise nonlinearity. In the vast majority of cases, the same nonlinearity is used for every unit in the layer, though different nonlinearities for different units are also possible.

The matrix W_i is most often dense; in this case, the corresponding layer is denoted as fully-connected.

1.3.1.3 Output layer

In the case of classification (which is the most interesting one for this thesis), the output layer can most often be described as a hidden layer with softmax nonlinearity, output weights W_o and biases b_o:

$$o = softmax(W_o * h_M + b_o) \tag{1.13}$$

1.3.1.4 Representational power

A multilayer perceptron with a single hidden layer is an universal approximator [73] of continuous functions: for any function $f(x)$ and any $\epsilon > 0$, there exists a neural network $g(x)$ with one hidden layer and an e.g. sigmoid nonlinearity such

that, for any x, $|f(x) - g(x)| < \epsilon$. The motivation for deep neural networks, even if shallow networks are also universal approximators, comes from the fact that there exist functions which require exponentially more computational resources when approximated by a shallow artificial neural network than by a deep one [156] [158] [155].

1.3.2 Convolutional neural networks (CNNs)

In this subsection we will discuss the Convolutional Neural Network (CNN), which is a specialized neural network architecture initially proposed for image processing. Like MLPs / DNNs, CNNs also contain neurons connected by learnable weights and biases and the operations performed by the neurons (units) are like in MLPs: each neuron will receive some inputs, then perform a dot product and then apply a nonlinearity to the result. The entire system is also differentiable (like MLPs), uses a loss function and the typical optimization procedures (stochastic gradient, backpropagation) apply, as will be discussed in Section 1.5. Although mostly used for image processing, in recent years, the use of different CNN variants has been extended to many more domains, e.g. speech recognition [205] or natural language processing [82]. Our presentation is mostly inspired by the excellent introductions in [10] and [12]. We will assume that the inputs to the CNN are images, to simplify the presentation; extensions to CNNs processing text or other types of inputs are straightforward and many of the ideas and motivations carry over naturally to those cases.

The first significant difference between CNNs and MLPs is the pattern of connectivity of each neuron. While in MLPs neurons in a layer are fully-connected to all the neurons in a consecutive layer, in CNNs the pattern of connectivity is sparse and local. This is inspired by work in neuroscience, where Hubel and Wiesel [113] have discovered that neurons in the visual cortex of cats act as local filters over their input space. The small sub-region of the input each cell is sensitive to is called a receptive field, a term also used to denote the connectivity pattern of CNN neurons.

Another conceptual difference between CNNs and MLPs is that in CNNs the neurons / units are replicated across 2D arrays. A 2D array of replicated units shares the same trainable parameters and is called a *feature map*. This means that the weights and biases of multiple neurons have the same values. The shared trainable parameters are also denoted as a *filter*. As is also the case for RNNs and discussed in more details in Section 1.5, the gradient with regard to a shared trainable parameter is the sum of the gradients with regard to all the parameters being shared.

The replicated units and replicated filters allow for features to be detected regardless of the position in the input image where they appear. This property is called equivariance and roughly means that the resulting feature map changes similarly to how the input image would. If the image were translated one pixel to the left, so would the resulting feature map after applying the filter.

Similarly to MLPs, CNNs are deep neural networks, with multiple layers. A CNN is obtained by stacking repeatedly blocks of convolutional and subsampling layers, which will be discussed below. For classification, the task of most interest for this thesis, the repeated blocks of convolutional and subsampling layers are most commonly followed by fully connected layers and a softmax layer.

For clarity, we'll discuss how CNNs work when a single example (image) is provided as input. Modern CNNs are usually trained on mini-batches containing multiple images. The discussion here can easily be extended to the mini-batch case, by adding an extra dimension (corresponding to the mini-batch) to every tensor.

1.3.2.1 Convolutional layer

For simplicity, we will only discuss stride 1 convolution here, which is the most relevant for the work in this thesis. We will also not discuss zero padding settings. A discussion on strides and paddings for convolutional layers is provided in [12].

The feature maps discussed in the previous subsection will be obtained in a CNN in the following manner. The input image is convoluted with a linear filter, a bias term is added and then a nonlinear function f is applied. The convolution can be interpreted as repeatedly applying the same function across sub-regions of the entire image. To allow for a richer representation of the data, each convolutional or subsampling hidden layer contains multiple feature maps h_k, $k \in 0, ..., K$. We will denote the k-th feature map by h_k and the corresponding filter, composed of trainable weights and biases, by W_k and b_k and the input image by x. A nonlinear function is then applied element-wise to every pixel in every feature map. The corresponding equation for pixel $[i, j]$ in feature map h_k is:

$$h_k[i, j] = f((W_k \circledast x)[i, j] + b_k) \tag{1.14}$$

We have introduced convolution in Section 1.2.1.2. f can be any nonlinearity in Section 1.2.2, with ReLU being a popular modern choice.

The discussion above would apply exactly for a CNN with a single convolutional layer. Similarly to the theoretical results obtained for deep MLPs [158], deep CNNs

are more expressive than shallow ones [149]. We will now extend the discussion of the convolution operation to two consecutive hidden layers, $m-1$ and m. We will denote feature map k in layer m by h_k^m. The input image could be interpreted as layer 0. For an RGB image, it would contain three feature maps (one for each of the red, green and blue channels), while a grayscale image would contain a single feature map. The layers h^m can be convolutional (including the input image) or subsampling (the subsampling layer is presented in the next subsection). By aggregating the weights W_k for every feature map h_k, we obtain a 4D tensor which contains elements for every destination feature map h_k^m, every source feature map h_l^{m-1}, source vertical position i and source horizontal position j. $W_{kl}[i,j]$ denotes the weights connecting each pixel of the k-th feature map at layer m with the pixel coordinates $[i,j]$ of the l-th feature map at layer $m-1$. The biases b can be represented as a vector indexed by the destination feature maps; b_k^m denotes the bias corresponding to destination feature map h_k^m.

1.3.2.2 Subsampling layer

Many different proposals for subsampling layers exist, but we will discuss here the max-pooling layer, which is the most successful and most widely-used subsampling layer. Other notable uses of subsampling are the average pooling layer [53] and using a stride larger than 1 in the preceding convolutional layer [177]. What is common to these layers is that, when applied to a group of pixels of a feature map, they first optionally perform a transformation of the pixels, then select one of them as the result. What is specific to max-pooling is that it selects the pixel with the maximum value (of the group).

The most common choice is to divide the pixels into non-overlapping groups, with the most common choice being groups of 2 pixels width and 2 pixels height. When used with max-pooling, this is called 2×2 max-pooling and is the most widely used setting for a pooling layer.

A more detailed discussion of various types of pooling layers, with different modes to group pixels (with potential overlapping and various subsampling options, as well as the sizes of the corresponding resulting feature maps) is provided in [12].

Given input feature map i, the output feature map o is given by:

$$o[m,n] = \max(i[2{*}m, 2{*}n], i[2{*}m{+}1, 2{*}n], i[2{*}m, 2{*}n{+}1], i[2{*}m{+}1, 2{*}n{+}1]) \quad (1.15)$$

The main motivation behind the max-pooling layer is to provide some translation invariance and robustness to small image distortions. The resulting reduction in feature map size (both the width and the height are divided by 2 in the case of 2×2 max-pooling) also helps to reduce the computational requirements of the CNN.

An interesting feature of the max-pooling layer is that it doesn't contain any trainable parameters.

1.3.2.3 Output layer for classification

When the CNN is used for classification, all the pixels in the feature maps resulting from the last layer (convolutional, nonlinear or subsampling) are reshaped to a 1D array. Fully-connected layers and a softmax layer can then be added to perform classification.

Other possibilities exist in the literature to obtain classification scores from 2D feature maps (e.g. [177]), but we won't discuss them here.

1.3.2.4 CNN design patterns

In this section we will very briefly mention some of the most influential CNN architectures in deep learning. The most influential task on which CNNs are trained is object recognition on the ImageNet dataset [23].

Our simplified description of a CNN above is most in line with the VGG CNN architecture [171], since it is the one closest to the CNNs we have used in our work. This architecture is very homogeneous, containing 3×3 convolutional layers and 2×2 max-pooling layers. It was introduced in the context of the ImageNet Large Scale Visual Recognition Competition (ILSVRC) 2014, where it was the runner-up.

One of the early influential CNN architectures trained with backpropagation was LeNet-5, introduced by Yann LeCun for the task of digit recognition [130], which we discuss in more detail in Section 2.1.1.

AlexNet [126] is the architecture which popularized the use of CNNs in Computer Vision, after winning the ILSRVC challenge in 2012. It also popularized the use of GPUs for training neural networks and led to deep learning becoming the dominant paradigm in Computer Vision.

GoogleNet [185] won the ILSRVC 2014 challenge and introduced Inception Modules, which significantly reduced the number of trainable parameters (compared to e.g. AlexNet). The GoogleNet architecture has gone through several iterations, the most recent one being v4 [184].

Residual CNNs (ResNets) [104] won ILSVRC 2015 and introduced skip connections to ease the training of very deep networks, by reducing the problem of vanishing gradients (see Subsection 1.5.2.5. As of 2018, ResNet variants are often the default choice for using CNNs in practice.

We have only briefly mentioned some of the most famous CNN architectures. A more detailed discussion regarding the architectures mentioned above is provided in [12].

We will discuss in more detail some CNNs which have been influential in the context of handwriting recognition (including LeNet) in Chapter 2.

1.3.3 Recurrent neural networks (RNNs)

Recurrent neural networks (RNNs) extend the deep neural network approach to sequential inputs. Probably the most particular feature of RNNs is that are constrained to use the same 'transition function' for each time step (by 'tying' the weight matrix at each timestep). They can thus construct features which are independent of the particular timestep at which they appear and can learn to predict outputs / output sequences from input sequences independently of sequence length.

1.3.3.1 Standard RNNs

Standard RNNs represent the simplest type of RNN. The equation for a vanilla RNN is:

$$h_t = \sigma(W_{hh} * h_{t-1} + W_{hx} * x_t + b_h) \tag{1.16}$$

Here, h_t is the representation learned by the RNN for each input x_t at time t. W_{hh} and b_h are trainable. σ is an element-wise nonlinearity (for example, sigmoid, tanh or ReLUs - see Subsection 1.2.2).

Predictions can then be computed from the learned representations h_t. For example, in the case where we would want to predict a new output (which we will denote by $\widehat{y_t}$) at every timestep t, we can use the following equations:

$$\widehat{y_t} = W_{yh} * h_t + b_y \tag{1.17}$$

Here, W_{yh} and b_y are trainable parameters.

Some useful loss functions for RNNs are described in section 1.5.1. The RNNs can be trained using gradient-based optimization (see section 1.5).

1.3.3.2 Long short-term memory (LSTM)

In theory, recurrent neural networks are universal sequence approximators (with a sufficient number of hidden units, an RNN can approximate any measurable sequence-to-sequence mapping to arbitrary accuracy) [167] and are Turing-complete [170]. In practice, however, researchers have found it hard to train RNNs, because of optimization problems such as the vanishing gradient problem [110], [42] - see Subsection 1.5.2.5. The Long-short term memory (LSTM) cell [112], which is described in this subsection, is one way of dealing with this issue.

The LSTM equations are:

$$c_t = f_t \bullet c_{t-1} + i_t \bullet \tanh(W_{hc} * h_{t-1} + W_{xc} * x_t + b_c) \tag{1.18}$$

with c_t being the cell state, f_t, the output of the forget gate, and i_t the value of the input gate.

$$f_t = \sigma(W_{xf} * x_t + W_{hf} * h_{t-1} + W_{cf} * c_{t-1} + b_f) \tag{1.19}$$

W_{cf} corresponds to the weight of the peephole connexion between the cell and the input gate.

$$i_t = \sigma(W_{xi} * x_t + W_{hi} * h_{t-1} + W_{ci} * c_{t-1} + b_i) \tag{1.20}$$

$$o_t = \sigma(W_{xo} * x_t + W_{ho} * h_{t-1} + W_{co} * c_t + b_o) \tag{1.21}$$

$$h_t = o_t \bullet \tanh(c_t) \tag{1.22}$$

$\bullet$ signifies element-wise multiplication. These equations correspond to the LSTM variant with added peephole connections. We can simplify the LSTM by removing these connections; the equations of the input, forget and output gate then become:

$$f_t = \sigma(W_{xf} * x_t + W_{hf} * h_{t-1} + b_f) \tag{1.23}$$

$$i_t = \sigma(W_{xi} * x_t + W_{hi} * h_{t-1} + b_i) \tag{1.24}$$

$$o_t = \sigma(W_{xo} * x_t + W_{ho} * h_{t-1} + b_o) \tag{1.25}$$

The equation for the final output (h_t) stays the same.

1.3.3.3 Bidirectional RNN (BRNN)

In a standard RNN, the output at a given time t depends exclusively on the inputs x_0 through x_t (via the hidden layers unrolled in time h_0 through h_{t-1}). However, while this makes sense in some contexts, many sequences have information relevant to output y_t both before, as well as after timestep t. For example, in handwriting recognition, both the sequence of drawn characters before and after the current character provide information about the drawn character currently processed. To use all this information, we need a modified architecture.

Bidirectional RNNs add another set of hidden layers to a recurrent network which goes backwards in time, besides the set of hidden layers which goes forwards in time of a standard RNN. These two sets of hidden layers are entirely separate and do not interact with each other, except for the fact that they are both used to compute the output. Using the trainable parameters, the BRNN is first run forward in time (from time 0 to the end) to compute the forward hidden layers, and then run backward in time (from the end to time 0) to compute the backward hidden layers. Finally, using the values at both of the hidden layers for a given timestep, the output corresponding to every timestep is computed.

Denoting the forward layer by $\overrightarrow{h_t}$ and the backward layer by $\overleftarrow{h_t}$, the equations 1.16 for the network are modified as follows :

$$\overrightarrow{h_t} = \sigma(W_{hh} * \overrightarrow{h_{t-1}} + W_{hx} * x_t + \overrightarrow{b_h}) \tag{1.26}$$

$$\overleftarrow{h_t} = \sigma(W_{hh} * \overleftarrow{h_{t+1}} + W_{hx} * x_t + \overleftarrow{b_h}) \tag{1.27}$$

$$\widehat{y_t} = W_{y\overrightarrow{h}} * \overrightarrow{h_t} + W_{y\overleftarrow{h}} * \overleftarrow{h_t} + b_y \tag{1.28}$$

1.4 Performance measures

We use performance measures to characterize the performance of our recognition systems. Commonly used performance measures for machine learning systems which label examples with a discrete number of potential labels (classes) are the label error rate (LER) for non-sequential data and the character error rate (CER) and word error rate (WER) for sequential data and more particularly handwriting recognition.

1.4.1 Label error rate (LER)

For a dataset S, the LER is the ratio of examples the classifier mislabels:

$$LER = \frac{\text{number of errors}}{\text{number of examples}} \tag{1.29}$$

Another commonly used measure is the accuracy, which can be described as $1 - LER$.

1.4.2 Character error rate (CER)

The formula for the character error rate is:

$$CER = \frac{S + D + I}{N} \tag{1.30}$$

where S is the number of substituted characters, D is the number of deleted characters, I is the number of inserted characters, and N is the number of characters in the ground-truth.

1.4.3 Word error rate (WER)

Similarly to the formula for the character error rate, the formula for the word error rate is:

$$WER = \frac{S + D + I}{N} \tag{1.31}$$

where S is the number of substituted words, D is the number of deleted words, I is the number of inserted words, and N is the number of ground-truth words.

1.5 Gradient-based optimization

In previous sections we have discussed various neural network architectures. Generally, all these architectures can be represented as parameterized functions $f(x, \theta)$, where by x we denote the inputs and by θ we denote the trainable parameters. We can suppose that, initially, the parameters θ are initialized randomly; this is actually commonly used for neural networks (with specific probability distributions often used to accelerate the training).

In the next subsection we will first introduce loss functions, which are used to quantify the quality of the set of trainable parameters θ. The loss function is minimized

by the optimization process, which searches for the corresponding parameters θ. We will then briefly discuss optimization, the process of finding the parameters θ which minimize the loss function L. Our discussion is mostly inspired by the presentation in [13].

1.5.1 Loss functions

Ideally, we would want our machine learning systems to perform as well as possible on new data, according to some performance measures (see Section 1.4).

However, we often don't have access to real test-time data and optimizing the system on test-time data can often lead to overfitting (see Section 1.5.4).

Also, the most commonly encountered performance measures are often not differentiable, a property which is strongly desirable. For these reasons, we will often optimize the machine learning system according to a loss function which is different, but related to the desired performance.

1.5.1.1 Cross-entropy

Here we will place ourselves in the setting of supervised learning. We have a training set S of examples x associated with labels y and want to train a classifier to provide labels for new unlabelled examples x. We will train the classifier to maximize the probability $p(y|x)$ predicted by the classifier (which is equivalent to minimizing the negative log probability). Denoting the loss function by $O(x, y)$, for a single example x with associated ground-truth label y we have:

$$O(x, y) = -\log(p(y|x)) \tag{1.32}$$

By assuming that the data samples (x, y) are independent, the loss function over the entire training set $O(S)$ becomes the sum of the loss functions over each separate sample (x, y):

$$O(S) = \sum_{(x,y)\in S} [-\log(p(y|x))] \tag{1.33}$$

This loss can be minimized through gradient-based optimization.

1.5.1.2 Connectionist temporal classification (CTC)

In this subsection we will present the Connectionist Temporal Classification (CTC) model, as introduced by Alex Graves in [95]. This presentation is heavily inspired by

the blog articles [37] and [119], as well as Alex Graves' thesis [92] and book [93].

Symbol	Meaning
L	finite alphabet of n labels; *blank* (-) not included
L'	$L \cup \text{blank}(-)$
$\mathcal{X}$	$(\mathbb{R}^m)^\star$, m-dimensional input space (sequences of m-dimensional vectors)
$\mathcal{Z}$	$L^\star$, output space, set of all sequences over L
$D_{X\times Z}$	underlying distribution of the data (pairs of input sequences and corresponding label sequences)
S	set of training examples supposed to be drawn from $D_{X\times Z}$
T	length of RNN input/output sequence
U	length of label sequence
(x, l)	example in $S, x = (x_1, .., x_T), l = (l_1, .., l_U)$ and $U <= T$
$h : X \to Z$	temporal classifier (RNN) to be trained
$N_W : (\mathbb{R}^m)^T \to (\mathbb{R}^n)^T$	RNN with m inputs, n outputs and trainable parameters W, as a continuous map applied on a length T sequence
y_k^t	sequence of RNN outputs (the activation of output unit k at time t)
π	*path*, element of L'^T
$l \in L^U$	label sequence or *labeling*
$B : L'^T \to L^U$	function mapping path π to label sequence l
l'	modified label sequence, with blanks added to the beginning and the end and inserted between every pair of labels in l
$l'_{a:b}$	subsequence of l' from a-th to b-th labels
$\alpha_t(s)$	forward variable, the total probability of $l'_{1:s}$ at time t
$\beta_t(s)$	backward variable, the total probability of $l'_{s:\lvert l'\rvert}$ starting from time t
$O^{ML}(S, N_W)$	objective function

Table 1.1 – CTC notation

CTC can be applied to supervised sequence classification tasks, such as handwriting recognition or speech recognition. It helps represent information in an invariant (or at least robust) manner to sequential distortions. It is also interesting for the fact that it doesn't require any pre-segmentation of the input or any post-processing of the output labels to produce the final predicted label sequence.

We will start by introducing the list of symbols used in the algorithm, similarly to the presentation in [119], in Table 1.5.1.2.

In handwriting recognition, the goal is to build a classifier which can convert an image (which can be viewed e.g. as a sequence of pixels or a sequence of pixel windows) to a sequence of labels (e.g. the individual characters or words). The

sequence of pixels represents the input to our classifier, while the sequence of labels represents the desired output of the classifier; thus, the whole problem of recognizing lines, words or characters of handwritten text can be posed as supervised sequence classification (mapping from an input sequence to an output sequence). In this work, we mainly deal with character and word images.

We will use a classifier (an RNN) which takes in the input sequence x and generates outputs y_k^t at every timestep t of the input sequence. y_k^t will be the probability of outputting label k from alphabet L', which is the same as the alphabet L augmented with the - (*blank*) symbol, for reasons which will become clearer later in this section, at timestep t. To have the RNN output probabilities, we can use a softmax layer (see the softmax nonlinearity in Subsection 1.2.2).

We would like to measure the loss of our classifier by considering the RNN classifier as a function transforming the input sequence to the output sequence. For this reason, we would like our loss function to be of the following form:

$$O^{ML}(S, N_W) = -\sum_{(x,l)\in S} \ln(p(l|x)) \tag{1.34}$$

where l is the label sequence and x is the input sequence.

Notice that this CTC loss fuction is the generalization of the cross-entropy loss used for feedforward neural networks (see Subsection 1.5.1.1), except that now both the input and the output are sequences. We can then minimize this loss function via some standard minimization algorithm (such as gradient descent). Minimizing the objective function defined above maximizes the log likelihood of observing our desired label sequence. The problem we have is that the sequence of RNN outputs and the sequence of labels can be of different lengths (denoted by T and U, respectively). Intuitively, we can think of the CTC algorithm as performing alignment as a part of training the classifier (aligning the T inputs to the corresponding U outputs). Other models which also perform alignment have been proposed in the deep learning literature; for a review, see [60].

We can train the RNN using gradient-based optimization, as long as we can differentiate Equation 1.34 with regard to (w. r. t.) y_k^t. Because the training examples are independent, we can rewrite the gradient of the loss function w. r. t. the entire training set as the sum of the gradients of the loss function w. r. t. each separate training example:

$$\begin{aligned}\frac{\partial O^{ML}(S, N_W)}{\partial y_k^t} = \frac{\partial - \sum_{(x,l)\in S} \ln(p(l|x))}{\partial y_k^t} &= - \sum_{(x,l)\in S} \frac{\partial \ln(p(l|x))}{\partial y_k^t} \\ &= - \sum_{(x,l)\in S} \frac{1}{p(l|x)} \frac{\partial p(l|x)}{\partial y_k^t}\end{aligned} \tag{1.35}$$

We now need to map between the loss function $O^{ML}(S, N_W)$ and the RNN outputs y_k^t. We will start by denoting the output of the RNN for an entire input sequence as a *path*. If we assume that the RNN outputs are conditionally independent given the input sequence (an assumption which can be satisfied by not allowing any connections between the RNN's output layer and its hidden or output layer), then the probability of path π can be decomposed into the product of probabilities $y_{\pi_t}^t$, where π_t is the t-th element of path π:

$$p(\pi|x) = \prod_{t=1}^{T} y_{\pi_t}^t \tag{1.36}$$

We can map from path π to a label sequence using only characters from alphabet L by removing all blanks and all duplicate (consecutive) letters. We denote the blank character by '-' and the mapping function by B. For example, the path '-b-ee' will be mapped by B to label sequence 'be', while the path 'b–e-e' will be mapped to label sequence 'bee'. If we denote by l the label sequence resulting from removing all blanks and all duplicate (consecutive) letters through the action of B, we obtain:

$$p(l|x) = \sum_{B(\pi)=l} p(\pi|x) = \sum_{B(\pi)=l} \prod_{t=1}^{T} y_{\pi_t}^t \tag{1.37}$$

By taking the logarithm of the equation above and reversing its sign, we obtain $O^{ML}(S, N_W)$, which we can minimize to train the RNN. The naive computation of $p(l|x)$ using the equation above is intractable, though, since the number of different paths grows exponentially w. r. t. the path length.

Fortunately, $p(l|x)$, as well as the gradients $\frac{\partial p(l|x)}{\partial y_k^t}$, can be computed efficiently using a forward-backward algorithm similar to those used for training Hidden Markov Models (HMMs), as we will describe below. From $\frac{\partial p(l|x)}{\partial y_k^t}$, gradients w. r. t. the RNN's trainable parameters can be computed, so that the trainable parameters can be learned.

Instead of working directly with ground-truth label sequence l, we will construct

a modified ground-truth label sequence l', in which the blank label is inserted at the start and at the end of l' and between every consecutive characters. The length of l' is thus $|l'| = 2 \cdot |l| + 1$.

Denote by $l'_{1:s}$ the subsequence of l' which starts at the first element and ends at the sth element. Let $\alpha_t(s)$ be the total probability of observing $l'_{1:s}$ at time t, which we also denote as the forward variable. This probability can be written as the sum of probabilities over all the paths that contain $l'_{1:s}$ at time t (equivalently, all the $B(\pi_{1:t}) = l_{1:\frac{s}{2}}$, where $\frac{s}{2}$ is rounded down to an integer value):

$$\alpha_t(s) = \sum_{\pi:B(\pi_{1:t})=l_{1:\frac{s}{2}}} p(\pi|x) = \sum_{\pi:B(\pi_{1:t})=l_{1:\frac{s}{2}}} \prod_{i=1}^{t} y^i_{\pi_i} \tag{1.38}$$

The probability of l is the sum of the probabilities of seeing l' with and without the last blank at time T:

$$p(l|x) = \alpha_T(|l'| - 1) + \alpha_T(|l'|) \tag{1.39}$$

The values of α can be computed recursively using dynamic programming. Since all paths have to start with either the *blank* symbol b or the first label l_1, at time $t = 1$ (the base case) we have:

$$\begin{aligned} \alpha_1(1) &= y^1_- \\ \alpha_1(2) &= y^1_{l_1} \\ \alpha_1(s) &= 0, \forall s > 2 \end{aligned} \tag{1.40}$$

All other $\alpha_t(s)$ can be computed recursively using the following equations:

$$\alpha_t(s) = \begin{cases} y^t_{l'_s} \cdot (\alpha_{t-1}(s) + \alpha_{t-1}(s-1)), & \text{if } l'_s = - \text{ or } l'_{s-2} = l'_s \\ y^t_{l'_s} \cdot (\alpha_{t-1}(s) + \alpha_{t-1}(s-1) + \alpha_{t-1}(s-2)), & \text{otherwise} \end{cases} \tag{1.41}$$

The intuition behind this equation goes as follows.

If $l'_s = -$, then the last symbol of the prefix is a *blank*. A *blank* can only be reached at time t in one of two ways. First, the entire prefix could already have been seen by $t - 1$, followed by the repeated last *blank*. The probability for this event is $y^t_{l'_s} \cdot \alpha_{t-1}(s)$ ($y^t_{l'_s}$ is the probability of seeing the *blank* at time t, $\alpha_{t-1}(s)$ is the probability of already seeing the entire prefix at time $t - 1$). The second way is to have seen every symbol of the prefix except for the last *blank* at time $t - 1$, and then

see the last *blank* at time t. The corresponding probability is $y^t_{l'_s} \cdot \alpha_{t-1}(s-1)$.

Otherwise, if $l'_s = l'_{s-2}$, this means that the original sequence has two identical consecutive letters (between which we have added a *blank* in l'). We can (again) see $l'_{1:s}$ by time t in two ways. We could have seen the entire prefix by time $t-1$, so that at t we see l'_s again (the repeated l'_s will be removed when processed by B). The probability corresponding to this event is $y^t_{l'_s} \cdot \alpha_{t-1}(s)$. Otherwise, we could have seen everything but the last letter by time $t-1$ (and have seen the last letter at time t). This corresponds to the probability $y^t_{l'_s} \cdot \alpha_{t-1}(s-1)$. Thus, we get the same equation as for the previous case $l'_s = b$.

Finally, in the case of a non-blank letter l'_s which is also different from the previous non-blank letter l'_{s-2}, we can see the entire prefix by time t in the same two manners described previously, but also in a third manner. l'_s can be observed immediately after l'_{s-2} (without observing a *blank* in between) if l'_{s-2} was observed at time $t-1$ and l'_s at time t. This corresponds to the probability $y^t_{l'_s} \cdot \alpha_{t-1}(s-2)$, the last term for the second case of Eq. 1.41.

The forward variables $\alpha_t(s)$ can thus be computed for any t, s and l'.

We will now detail how to compute the backward variables, $\beta_t(s)$, which define the total probability of observing $l'_{s:|l'|}$ starting from time t. These variables can be described as computing the sum of path probabilities over all paths π with the desired suffix $l'_{s:|l'|}$:

$$\beta_t(s) = \sum_{B(\pi_{t:T})=l_{\frac{s}{2}:|l|}} p(\pi|x) = \sum_{B(\pi_{t:T})=l_{\frac{s}{2}:|l|}} \prod_{i=t}^{T} y^i_{\pi_i} \tag{1.42}$$

Similarly to the case of forward variables, computing the $\beta_t(s)$ variables naively using Eq. 1.42 above is intractable. Instead, we can again compute these variables recursively.

For the base case, at time $t = T$:

$$\begin{aligned} \beta_T(|l'|) &= y^T_- \\ \beta_T(|l'|-1) &= y^T_{l_{|l|}} \\ \beta_T(s) &= 0, \forall s < |l'|-1 \end{aligned} \tag{1.43}$$

The meaning of Eq. 1.43 is that at time T we can only observe the either last letter of the label sequence or the last *blank*.

We can now write the recursive equations:

$$\beta_t(s) = \begin{cases} y^t_{l'_s} \cdot (\beta_{t+1}(s) + \beta_{t+1}(s+1)), & \text{if } l'_s = - \text{ or } l'_s = l'_{s+2} \\ y^t_{l'_s} \cdot (\beta_{t+1}(s) + \beta_{t+1}(s+1) + \beta_{t+1}(s+2)), & \text{otherwise} \end{cases} \tag{1.44}$$

Eq. 1.44 is symmetrical to Eq. 1.41 and to understand the intuition behind it the same type of reasoning can be applied as for the intuition behind Eq. 1.41.

We can thus tractably compute both the forward variables $\alpha_t(s)$ and the backward variables $\beta_t(s)$ for all t, s and l'.

Since the forward variable $\alpha_t(s)$ provides the total probability of prefix $l'_{1:s}$ at time t (see Eq. 1.38) and the backward variable $\beta_t(s)$ provides the total probability of suffix $l'_{s:|l'|}$ at time t (see Eq. 1.42), taking their product $\alpha_t(s) \cdot \beta_t(s)$ provides the probability of all paths corresponding to label sequence l' which visit the sth character at time t, but with $y^t_{l'_s}$ appearing twice:

$$\alpha_t(s) \cdot \beta_t(s) = \sum_{B(\pi_{1:t})=l_{1:\frac{s}{2}}} \prod_{i=1}^{t} y^i_{\pi_i} \cdot \sum_{B(\pi_{t:T})=l_{\frac{s}{2}:|l|}} \prod_{i=t}^{T} y^i_{\pi_i} = \sum_{B(\pi)=l, \pi_t=l'_s} y_{l'^t_s} \cdot \prod_{i=1}^{T} y^i_{\pi_i} \tag{1.45}$$

Dividing by $y_{l'^t_s}$ (to avoid double counting it):

$$\frac{\alpha_t(s) \cdot \beta_t(s)}{y^t_{l'_s}} = \sum_{B(\pi)=l, \pi_t=l'_s} \prod_{i=1}^{T} y^i_{\pi_i} = \sum_{B(\pi)=l, \pi_t=l'_s} p(\pi|x) \tag{1.46}$$

By taking the sum over all possible symbol positions s in Eq. 1.46 (since one of the symbols must appear at time t), we obtain the total probability of l (as the sum of probabilities of any s appearing at time t):

$$p(l|x) = \sum_{s=1}^{|l'|} \frac{\alpha_t(s) \cdot \beta_t(s)}{y^t_{l'_s}} \tag{1.47}$$

This is valid for any t (from 1 to T), so we can obtain $\frac{\partial p(l|x)}{\partial y^t_k}$ for any character k and any time t. To differentiate w. r. t. y^t_k, we only need to consider those paths which go through symbol k at time t (the derivatives for the other paths are zero). Differentiating Eq. 1.46:

$$\frac{\partial \frac{\alpha_t(s)\cdot\beta_t(s)}{y^t_{l'_s}}}{\partial y^t_k} = \frac{\partial \sum_{B(\pi)=l,\pi_t=l'_s} \prod_{i=1}^{T} y^i_{\pi_i}}{\partial y^t_k} = \begin{cases} \sum_{B(\pi)=l,\pi_t=l'_s} \prod_{i=1,i\neq t}^{T} y^i_{\pi_i}, & \text{if } k = l'_s \\ 0, & \text{otherwise} \end{cases} \tag{1.48}$$

Since the same symbol can be repeated for a single label sequence, we will define the set of locations where label k occurs in l' as $loc(l,k) = \{s : l'_s = k\}$. The previous equation can then be rewritten as:

$$\frac{\partial \frac{\alpha_t(s)\cdot\beta_t(s)}{y^t_{l'_s}}}{\partial y^t_k} = \begin{cases} \frac{\alpha_t(s)\cdot\beta_t(s)}{{y^t_k}^2}, & \text{if } s \in loc(l,k) \\ 0, & \text{otherwise} \end{cases} \tag{1.49}$$

Using Eq. 1.47:

$$\frac{\partial p(l|x)}{\partial y^t_k} = \sum_{s\in loc(l,k)} \frac{\alpha_t(s)\cdot\beta_t(s)}{{y^t_k}^2} \tag{1.50}$$

Using Eq. 1.35 and Eq. 1.39:

$$\frac{\partial O^{ML}(S,N_W)}{\partial y^t_k} = \frac{1}{\alpha_T(|l'|-1)+\alpha_T(|l'|)} \cdot \sum_{s\in loc(l,k)} \frac{\alpha_t(s)\cdot\beta_t(s)}{{y^t_k}^2} \tag{1.51}$$

1.5.2 Gradient descent

Gradient descent is an iterative method which starts with an initial set of parameters theta, which are iteratively refined so that the loss function is gradually minimized. It is probably the most widely used method for optimizing machine learning models.

A widely used metaphor to describe the gradient descent process is that of a blindfolded hiker who tries to reach the bottom of a hill. The core intuition behind this approach (instead of e.g. using random search) is that it can be much easier to make a small improvement in the loss function than to come up with the optimal parameters theta in a single step (as would be necessary for random search). We won't discuss here more advanced optimization topics, such as second order optimization methods, but a discussion on these topics can be found in [17].

The core problem we will deal with in this section is how to compute gradients of the loss function with regard to the trainable parameters. These gradients are the main component required to be able to perform gradient descent. In the following subsections we will describe some of the main methods used to compute gradients, starting from the simplest, most general, but also most computationally

expensive method (finite differences) to the most efficient and most commonly used, but which requires further assumptions about the neural network being optimized (backpropagation).

Algorithm 1 Gradient descent

Legend:
t = timestep
θ_t = neural network parameter values at timestep t
lr = learning rate

Algorithm:
while θ_t not converged do:
    g_t = gradient of loss function with regard to θ_t
    $\theta_t = \theta_{t-1} - lr * g_t$

We show the basic gradient descent algorithm in Algorithm 1. At every iteration, some data points from the training set are selected and the gradients of the loss functions with regard to the neural network parameters. Depending on how many examples from the training set are used at each iteration, we get different variants of gradient descent. If a single example is selected randomly at every iteration, we obtain Stochastic Gradient Descent (SGD). If a fixed number of examples (e. g. 100) is selected at each iteration, we obtain mini-batch gradient descent (the examples are denoted as a *mini-batch*). When all the examples in the training set are used at every iteration, we obtain batch gradient descent. The trade-off between selecting more or fewer data points at every iteration goes as follows: using more examples allows for more useful gradients for the optimization process, but they also require more computation. Mini-batch gradient descent is the most often used variant of gradient descent in deep learning research, because it leads to less noisy gradients, while the code operating on the mini-batch examples can be parallelized (so that the computational cost isn't much larger than the cost of processing a single example).

1.5.2.1 Finite differences

The use of finite differences is the simplest and most general idea to compute gradients and is particularly useful in the case of neural network parameterizations which are not differentiable or are stochastic. The gradient will be computed (or, more precisely, approximated) numerically. The intuition behind this method is to compute the gradient $\frac{\partial L}{\partial \theta_i}$ by 'perturbing' every separate parameter θ_i (by adding a small $\epsilon > 0$) to measure the impact on the loss function. Several variants of this process exist.

The forward difference is:

$$\frac{\partial L}{\partial \theta_i} = \frac{L(\theta + \epsilon * e_i) - L(\theta)}{\epsilon} \tag{1.52}$$

where e_i is the unit vector with a 1 in the i-th place.

When ϵ approaches 0, the forward difference approximates the derivative. Notice that this quantity, and the derivative in general, is only informative for very small perturbations to the inputs.

The central difference is given by:

$$\frac{\partial L}{\partial \theta_i} = \frac{L(\theta + \frac{\epsilon}{2} * e_i) - L(\theta - \frac{\epsilon}{2} * e_i)}{\epsilon} \tag{1.53}$$

The central difference approximation is more accurate than the forward difference.

Both the forward and the central differences require a number of function evaluations equal to two times the number of parameters for every update of all the parameters.

Finite differences are also useful to check the implementation of analytic gradients, usually obtained using backpropagation, as detailed in [14].

1.5.2.2 Simultaneous perturbation stochastic approximation (SPSA)

Simultaneous perturbation stochastic approximation (SPSA) is similar to finite differences, in that it is general enough to be useful for parameterizations which are not differentiable or are stochastic. The difference is that it 'perturbs' all the trainable parameters θ at the same time (in a single step). Let δ be the vector of perturbations, with δ_i its i-th component.

Analogously to finite differences, the forward difference is:

$$\frac{\partial L}{\partial \theta_i} = \frac{L(\theta + \epsilon * \delta_i) - L(\theta)}{\epsilon * \delta_i} \tag{1.54}$$

Similarly, the central difference is:

$$\frac{\partial L}{\partial \theta_i} = \frac{L(\theta + \frac{\epsilon}{2} * \delta_i) - L(\theta - \frac{\epsilon}{2} * \delta_i)}{\epsilon * \delta_i} \tag{1.55}$$

SPSA only requires two function evaluations for one update of all the parameters, resulting in much higher computational efficiency per update step compared to finite differences. The price to pay is noisier gradients; on average, though, the gradient approximation is an almost unbiased estimator of the gradient, as shown in [33].

Recently, there has been a revival of interest in SPSA-like optimization algorithms, which have been shown competitive against state of the art reinforcement learning optimization algorithms [165] and even against backpropagation for supervised learning [195]. This is discussed in more details in Chapter 6.

1.5.2.3 Backpropagation

In this subsection we will present the intuition behind the backpropagation procedure, which is the most widely used procedure to obtain the gradients of loss functions with regard to the neural network trainable parameters. We will keep the description simple and focus on toy examples to highlight the intuition behind the method, like in [15]. The more complicated expressions required by the neural network architectures we have implemented are handled by the automatic differentiation provided by modern software like TensorFlow or PyTorch.

Backpropagation is a technique for computing derivatives quickly, which has been used in a variety of technical fields beyond deep learning. If finite differences and SPSA use numerical methods to obtain gradients, backpropagation obtains the gradients analytically. Formulated succintly, backpropagation computes gradients of mathematical expressions through recursive application of the chain rule.

Gradient descent (GD) through backpropagation is much faster than using finite differences. Backpropagation can accelerate training modern neural networks by up to 10 million times [5].

Backpropagation and automatic differentiation are applied most naturally to optimizing a function by decomposing it into modules for which local gradients can be easily derived and then chained (using the chain rule). Automatic differentiation highly simplifies the task of obtaining the analytic gradients (compared to deriving them on paper).

We will exemplify backpropagation on the following toy neural network:

$$p = f(x, w) = \sigma(x_0 * w_0 + x_1 * w_1 + b) \tag{1.56}$$

which corresponds to a neural network with two inputs $x = [x_0, x_1]$ and a single unit, with sigmoid nonlinearity (and trainable parameters $w = [w_0, w_1, b]$ - weights w_0, w_1 and trainable bias b).

We will suppose that the neural network is used for binary classification, with output p signifying the probability of example $x = [x_0, x_1]$ belonging to class 0 (and $1 - p$ the probability of it belonging to class 1). We will use as loss function the

cross-entropy:

$$L = y * \log(p) + (1 - y) * \log(1 - p) \tag{1.57}$$

where y is the ground-truth class corresponding to example x.

The neural network can naturally be decomposed in the following modules (the loss can be considered another module):

$$z = x_0 * w_0 + x_1 * w_1 + b \tag{1.58}$$

$$p = \sigma(z) = \frac{1}{1 + e^{-z}} \tag{1.59}$$

We will now compute the gradients for each module's output w.r.t. all its inputs and parameters:

$$\frac{\partial L}{\partial p} = \frac{y}{p} - \frac{1 - y}{1 - p} \tag{1.60}$$

$$\frac{\partial p}{\partial z} = \frac{\partial \sigma(z)}{\partial z} = \frac{e^{-z}}{(1 + e^{-z})^2} = \frac{1 + e^{-z} - 1}{1 + e^{(} - z)} * \frac{1}{1 + e^{(} - z)} = \sigma(z) * (1 - \sigma(z)) \tag{1.61}$$

$$\frac{\partial z}{\partial w_0} = x_0 \tag{1.62}$$

$$\frac{\partial z}{\partial w_1} = x_1 \tag{1.63}$$

$$\frac{\partial z}{\partial b} = 1 \tag{1.64}$$

$$\frac{\partial z}{\partial x_0} = w_0 \tag{1.65}$$

$$\frac{\partial z}{\partial x_1} = w_1 \tag{1.66}$$

We can now use the chain rule on the gradients of the modules to obtain the gradients of the loss function w.r.t. the trainable parameters ($\frac{\partial L}{\partial w_0}, \frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial b}$). The chain rule specifies the correct way to chain the gradients of the modules - through multiplication:

$$\frac{\partial L}{\partial w_0} = \frac{\partial L}{\partial p} * \frac{\partial p}{\partial z} * \frac{\partial z}{\partial w_0} \tag{1.67}$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial p} * \frac{\partial p}{\partial z} * \frac{\partial z}{\partial w_1} \tag{1.68}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial p} * \frac{\partial p}{\partial z} * \frac{\partial z}{\partial b} \tag{1.69}$$

We can see that the expressions $\frac{\partial L}{\partial p}$ and $\frac{\partial p}{\partial z}$ appear in all three gradient expressions, so it would be redundant to recompute them each time. We can store the intermediate gradients in separate variables to avoid redundancy in computation and gain efficiency by introducing intermediate variables (denoted as $dLdp$ for $\frac{\partial L}{\partial p}$ and $dLdz$ for $\frac{\partial L}{\partial z} = \frac{\partial L}{\partial p} * \frac{\partial p}{\partial z}$). At the same time, we will separate the computation into a forward propagation phase (running the neural network and computing the loss, propagating computation forward from the inputs through the neural network all the way to the loss function) and a backward propagation phase (computing the gradients of the loss function w.r.t. the trainable parameters, propagating values backwards from the loss function to the trainable parameters):

Algorithm 2 Forward and backward propagation for a toy neural network. We use Python-like 'code syntax', with the results of the computation shown to the rightmost, after '#' (corresponding to comments in Python)

e.g. input values: $x_0 = 1, x_1 = 2$, label $y = 0$, trainable parameters $w_0 = 1, w_1 = -1, b = 1$

#forward phase
$z = x_0 * w_0 + x_1 * w_1 + b$ # 0
$p = \sigma(z) = \frac{1}{1+e^{-z}}$ # 0.5
$L = y * \log(p) + (1 - y) * \log(1 - p)$ # -1

#backward phase
$dLdp = \frac{y}{p} - \frac{1-y}{1-p}$ #-2
$dLdz = dLdp * p * (1 - p)$ # -0.5
$dLdw_0 = dLdz * x_0$ # -0.5
$dLdw_1 = dLdz * x_1$ # -1
$dLdb = dLdz * 1$ # -0.5

After the gradients have been computed, they can be directly plugged into gradient descent or variants, like ADAM (see Section 1.5.3.1).

For our toy example, the number of variables is small. For realistic neural networks, operating on thousands or even millions of inputs, vectorizing both the forward and

the backward phases is important for efficient computation (especially on GPU). More details are provided in [16].

1.5.2.4 Backpropagation through time (BPTT)

Backpropagation Through Time (BPTT) is simply an extension of backpropagation, which uses the chain rule, to Recurrent Neural Networks (RNNs), where the same trainable parameters appear repeatedly in multiple expressions.

We will derive BPTT for a simplified, toy model of a RNN, but the extension to full, state of the art architectures like LSTMs is straightforward. Furthermore, modern neural network software like TensorFlow or PyTorch uses automatic differentiation, so we don't need to manually derive the BPTT equations when implementing RNNs.

The model is not supposed to be useful, but is illustrative for how the gradients of a loss function with regard to (w.r.t.) parameters shared across timesteps are computed.

Our toy model starts by taking as input a single real number, x_0; this can be interpreted as the system's initial state. At each timestep t, our toy model will obtain its next state x_{t+1} by multiplying the previous state, x_t, with the trainable parameter (single real number) a_t. After T such timesteps, we obtain the output $y = x_T$. The key feature of this model, to make it similar to RNNs, will be that all the trainable parameters a_t share the same value a ($a_t = a, \forall t \in \{0, ..., T-1\}$). Denoting the trainable parameter a at timestep t as a_t will be useful to illustrate how the derivatives are computed for each separate step t and how they are then combined to output the derivative w.r.t. the shared parameter a. This corresponds to summing the derivatives w.r.t. each separate a_t across all timesteps t.

The equations describing the model can thus be written as:

$$x_{t+1} = a_t * x_t = a * x_t \tag{1.70}$$

$$y = x_T \tag{1.71}$$

The derivative of the state x_{t+1} w.r.t. the previous state x_t:

$$\frac{\partial x_{t+1}}{\partial x_t} = a_t = a \tag{1.72}$$

Using the chain rule, we can obtain the derivative of the output y w.r.t. the state at timestep t x_t:

$$\frac{\partial y}{\partial x_t} = \frac{\partial y}{\partial x_{t+1}} * \frac{\partial x_{t+1}}{\partial x_t} = \frac{\partial y}{\partial x_{t+1}} * a_t = \frac{\partial y}{\partial x_{t+1}} * a \tag{1.73}$$

The derivative of the output y w.r.t the state at the last timestep is 1:

$$\frac{\partial y}{\partial x_T} = 1 \tag{1.74}$$

By induction, we can prove that:

$$\frac{\partial y}{\partial x_t} = a^{T-t} \tag{1.75}$$

The derivative of the output y w.r.t the trainable parameter a_t, at every timestep t can now be derived:

$$\frac{\partial y}{\partial a_t} = \frac{\partial y}{\partial x_{t+1}} * \frac{\partial x_{t+1}}{\partial a_t} = \frac{\partial y}{\partial x_{t+1}} * a^t * x_0 = a^{T-t} * a^t * x_0 = a^T * x_0 \tag{1.76}$$

Because the trainable parameters a_t are shared (they are all equal to a), to obtain the derivate of y w.r.t. a we need to sum all the derivatives of y w.r.t. a_t:

$$\frac{\partial y}{\partial a} = \sum_{t=0}^{t=T-1} \frac{\partial y}{\partial a_t} = T * a^T * x_0 \tag{1.77}$$

Although this toy model is not useful in practice and lacks several features of full RNNs (nonlinearities, multiple inputs, vector of hidden states, inputs at every timestep, loss functions), it is illustrative of full RNNs through its recurrence and shared trainable parameters (which can be interpreted as a temporal invariance) and how the gradient w.r.t. the shared trainable parameters is the sum of the gradients at different timesteps. Another illustrative feature is represented by the vanishing / exploding gradients, which we describe in the following subsection.

1.5.2.5 Vanishing / exploding gradients

The expression of the derivative $\frac{\partial y}{\partial a}$ that we have obtained in Eq. 1.77 illustrates well the problem of vanishing / exploding gradients, which had plagued RNNs [111] particularly before the invention of LSTMs [112] and which can still impede the learning of very long term dependencies, even for modern RNN architectures like LSTM [112] or GRU [61]. We can observe in Eq. 1.77 that the gradient magnitude explodes ($|\frac{\partial y}{\partial a}|$ becomes very large) if $|a| > 1$ and vanishes ($|\frac{\partial y}{\partial a}|$ becomes tiny) if

$|a| < 1$. When the gradient explodes, the values of the trainable parameters change very significantly between learning epochs, leading to unstable training. On the other hand, when the gradient vanishes, the learning can become so slow that it almost stops.

For vanilla RNNs (see Section 1.3.3.1), the main contributor to the vanishing / exploding gradient problem is the hidden-to-hidden transition matrix W_{hh}. The intuition is similar to our toy example: for most matrices W_{hh}, the gradient will either vanish or explode.

1.5.3 State of the art optimization algorithms and heuristics

In this subsection we describe ADAM, a state of the art optimization algorithm for training deep learning systems and gradient clipping, a heuristic often used to diminish the difficulty posed by exploding gradients when training recurrent neural networks.

1.5.3.1 ADAM optimization

ADAM [125] is an optimization algorithm which has been developed for improving the performance of stochastic gradient descent (SGD) on stochastic non-stationary loss functions and sparse gradients.

ADAM is also claimed [125] to allow for less hyper-parameter tuning. When using simpler optimization procedures such as SGD, different learning rates have to be chosen for different neural network layers, leading to more hyper-parameters which need to be set; [125] claims that this is not necessary when using ADAM. On the other hand, the algorithm introduces a few additional hyper-parameters (ϵ and a few other parameters described in detail in [125]).

We describe the basic algorithm in Algorithm box 3. For a more detailed description, see the relevant publication [125].

1.5.3.2 Gradient clipping

Gradient clipping is a method which limits the magnitude of the gradients, used to deal with exploding gradients in RNNs (see Subsection 1.5.2.5). It was introduced by [152].

Multiple variants of gradient clipping exist. The one used in this thesis is clipping by global norm, which is implemented in TensorFlow and described by [34].

Algorithm 3 ADAM

Legend:
t = timestep
θ_t = neural network parameter values at timestep t
lr = learning rate
ϵ = hyper-parameter to make the algorithm numerically stable

Algorithm:
while θ_t not converged do:
g_t = gradient of loss function with regard to θ_t
m_t = moving average of g_t
v_t = moving average of g_t^2
$\theta_t = \theta_{t-1} - lr * \frac{m_t}{\sqrt{v_t}+\epsilon}$

We suppose the gradients g of a neural network's parameters are provided as a list of tensors $\{g_1, ..., g_M\}$. Usually, each tensor contains the gradients of the trainable parameters of one neural network layer (so the neural network in our example would contain M layers). Gradient clipping by global norm also takes in the hyper-parameter *clip_norm*.

The global norm of all the gradients is:

$$global\_norm = \sqrt{(\sum_{i=1}^{M} ||g_i||^2))} \tag{1.78}$$

where $||g_i||$ is the L2 (Euclidean) norm equal to the square root of the sum of the squared elements of the tensor g_i.

Each gradient is then obtained by multiplying its original value (before clipping) by $\frac{clip\_norm}{\max(global\_norm, clip\_norm)}$.

This procedure helps alleviate the problem of exploding gradients and keep magnitudes of the values of the trainable parameters bounded, making the neural network (particularly in the case of RNNs) more numerically stable.

1.5.4 Helpful methods for optimization / regularization

In this subsection we describe dropout, batch normalization and early validation, which are mostly useful for regularization purposes (to prevent overfitting). Batch normalization has also been claimed to ease the optimization of neural networks.

Algorithm 4 Dropout

Legend:
l = layer index
x^l = input of layer l
y^l = output of layer l
p = dropout probability
Equations without dropout during training:
$y^l = W^l * x^l + b^l$
$x^{l+1} = f(y^l)$
Equations with dropout during training:
$r^l \sim Bernoulli(p)$
$\tilde{x}^l = r^l \cdot x^l$
$y^l = W^l * \tilde{x}^l + b^l$
$x^{l+1} = f(y^l)$
Equations without dropout during validation / test:
$y^l = W^l * x^l + b^l$
$x^{l+1} = f(y^l)$
Equations with dropout during validation / test:
$y^l = p * (W^l * x^l + b^l)$
$x^{l+1} = f(y^l)$

1.5.4.1 Dropout

Dropout [178] is a regularization method which allows for much bigger networks to be trained without overfitting. The basic idea is to randomly drop neurons from the neural network during training with probability p sampled from a Bernoulli distribution. The result of this procedure is that this training approximates training an ensemble of much thinner neural networks. During validation or testing, an approximation of using the ensemble of thinned networks can be obtained by scaling the activations by the same probability p.

See Algorithm 4 and [178] for more details.

1.5.4.2 Batch normalization

Batch normalization [116] is a regularization method which allows for much faster training of neural networks because it can allow the optimization algorithm to even work with slightly higher learning rates and to converge in fewer epochs. Batch normalization addresses the so-called problem of internal covariate shift [116], which can be defined as the changing (shifting) of the input distribution when training a classifier. A deep neural network can be seen as a composition of stacked layers,

Algorithm 5 Batch normalization

Legend:
n = number of examples in the mini-batch
i = index of the example in the mini-batch
μ = mean value of the examples in the mini-batch
σ^2 = variance of the examples in the mini-batch
ϵ = hyper-parameter to make the algorithm numerically stable
β, γ = learned parameters to restore the layer's representational power
Algorithm:
$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i$
$\sigma^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2$
$\tilde{x_i} = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$
$BN_{\gamma,\beta}(x_i) = \gamma \cdot \tilde{x_i} + \beta$

each taking as input the output of previous layers. During training, as we modify the parameters of the lower layers, the distribution of the input to the higher layers changes. Batch normalization works by standardizing the inputs between each layer of a neural network (to mean 0 and variance 1) during training, while also adding a few trainable parameters so that the original inputs can be recovered and the trained system does not lose any representational power. In doing so, it also improves the gradient flow through the network, by reducing the dependence of gradients on the scale of the parameters or of their initial values [116].

For more details on batch normalization, see the original publication [116].

1.5.4.3 Early stopping

Early stopping is a regularization technique which can improve the performance of machine learning systems by reducing overfitting. For a provided dataset, the training set is used to minimize a certain loss function. To reduce the possibility of overfitting, the system's performance on the validation set is measured at different points in time. It can happen that performance on the training set keeps improving, while validation set performance deteriorates / stagnates (overfitting). When using early stopping, the state of the system (determined by the values of its trainable parameters) is reverted to the point in time where validation performance was highest.

To make the description more concrete, we describe a setup often used when training neural networks in Algorithm 6.

Algorithm 6 Early stopping

Legend:
max_epochs = maximum number of epochs to train
epoch = current training epoch
max_epochs_no_improvement = maximum number of epochs without performance improvement on validation set during which training can continue
best_epoch = epoch at which the best validation performance was achieved

Algorithm:
while $epoch < max\_epochs$ do:
    update parameters on entire training set
    measure performance on validation set
    if best performance on validation set:
        save system parameters
        $best\_epoch = epoch$
    else if $epoch - best\_epoch > max\_epochs\_no\_improvement$:
        revert to saved system parameters
        break
    $epoch = epoch + 1$

1.6 Conclusion

In this chapter we have provided a short introduction to the deep learning aspects most relevant to this thesis. This included the most popular operations that are used within systems based on deep learning architectures, as well as the most popular architectures (multilayer perceptron, convolutional neural network and recurrent neural network). We have also described the optimization procedures used for training deep learning systems. The handwriting recognition systems presented in the next chapters will make use of the deep learning architectures presented here, particularly recurrent and convolutional, as well as the optimization procedures described.

Chapter 2

Deep learning-based handwriting recognition

Contents

In this chapter, we will first discuss the role played by handwriting recognition tasks in the history and resurgence of neural networks. The most significant role here is played by the highly influential Modified NIST (MNIST) [132] dataset, which has been referred to as 'the drosophila of machine learning' [108], serving as a benchmark for classification. We also describe other tasks, some of which use the same dataset, which have led to influential results in the field of deep learning.

In the second part of this chapter, we will present the historical evolution of end-to-end trained systems for handwriting recognition tasks, all the way up to some

of the most recently-proposed architectures. We divide the architectures into three categories, with corresponding subsections: deep neural networks (DNNs), recurrent neural networks (RNNs) and architectures mixing convolutional and recurrent layers (CNN-RNNs).

These discussions aren't necessarily meant to be exhaustive, as the literatures on deep learning and handwriting recognition are vast, but to at least present the major milestones from the last decades which link the two fields. Furthermore, we review the papers from a 'personalized' perspective, focusing subjectively on aspects we perceive as salient and relevant for this discussion.

2.1 The role of handwriting recognition tasks in the history of neural networks

The recent history of neural networks, starting with the 1990's, has seen the use of several handwriting tasks and datasets as influential benchmarks and showcases of the power of neural networks. We briefly highlight these tasks and datasets in this subsection.

2.1.1 MNIST for classification

The Modified NIST (MNIST) dataset is probably the best known and most influential dataset in all of machine learning.

[133] introduced MNIST and reported human accuracy of around 0.2%. The dataset contains 60000 training grayscale images and 10000 test images of handwritten digits (0 to 9) of size 28×28.

[130] introduced the LeNet-5 CNN architecture as part of a system which was used commercially, reading millions of checks per month in the United States in the late 1990s. This was one of the first major commercial applications of neural networks and the first major commercial success of CNNs. When benchmarked on MNIST, LeNet-5 obtained the best results on MNIST among all methods

LeNet-5 used some architectural design principles similar to those used by modern CNNs, such as alternating between convolutional and subsampling (pooling) layers and progressively reducing the spatial resolution of the feature maps, while compensating this through an increase in the number of feature maps (for deeper layers). Gradient-based learning was used and the CNN features were learned directly from the image pixels. One difference, though, is that LeNet-5 used mean pooling

(differently from most recent CNNs, which use max pooling).

Many of the conclusions of [130] also echo very modern or contemporary findings in the neural network literature.

For example, the main message of the paper is that, in order to build better pattern recognition systems, it's better to rely more on automatic learning than on hand-designed heuristics. This message echoes the contemporary focus on end-to-end learning (instead of hand-designed features). 'Training all the modules to optimize a global performance criterion' (which can be interpreted as the equivalent for the modern term 'end-to-end learning') is also presented as more desirable than 'manually integrating individually designed modules' (hand-designed pipelines). Character recognition (using the MNIST database) was explicitly motivated as a case study for these principles.

The success of end-to-end learning in the paper is motivated similarly to how the current success of deep learning is: by the availability of more data, more compute and better algorithms, all three factors being important.

[130] also discovered an early advantage of training a recognizer at the word level, rather than on pre-segmented characters. The same type of advantage would later be discovered by the handwriting recognition community for many more datasets and also be extended for training a recognizer at the line level, rather than word level.

Another modern feature of [130] is that it used a form of data augmentation, by randomly distorting the original training images. [130] also conjectured that gradient descent learning can have a regularization effect, a conjecture which seems confirmed by recent work (e.g. [58]). It also described specialized hardware to accelerate neural networks, noted the importance of more compute for the obtained state of the art results (larger compute can lead to better performing systems, but also requires more learning data) and conjectured that the comparative advantage of neural networks would increase as the size of the available training data grows.

MNIST has been 'credited' with contributing to the 'neural network winter' around 1995-2005, when research on neural networks declined considerably compared to the previous decade, partially due to the better performance of Support Vector Machines (SVMs) and other relatively simple machine learning methods (compared to neural networks) on this dataset [18]. A list of historical results on the MNIST dataset is compiled at [132].

If MNIST could be credited for the 'neural network winter' of the 1990's and early 2000's, it could also be credited for the current resurgence of neural networks. [109] led to a breakthrough in neural network research and is the paper which arguably started

the current wave of deep learning, by showing that adding depth (more layers) to neural networks can lead to improved performance. Previously, when researchers had tried to train deeper neural networks using backpropagation, they hadn't been able to obtain significant performance gains, most notably due to vanishing gradients. [109] introduced what would become the paradigm of unsupervised generative pretraining followed by discriminative fine-tuning training and showed its effectiveness on MNIST. Although the learning procedure used was quite complicated (especially the specifics for the MNIST task), improved performance on permutation-invariant MNIST was achieved. The permutation-invariant task doesn't take into account the local structure of pixels in the image; it corresponds to permuting the pixels in each image using the same permutation. [109] was explicitly motivated as showing better performance than all purely discriminative algorithms on MNIST, most notably SVMs and neural networks trained discriminatively with backpropagation. Table 1 in [109] showed that for the permutation-invariant MNIST task the proposed method obtained the state of the art results at the time.

[64] reported the first near-human performance on MNIST, by training CNNs on GPUs. The importance of using GPUs and, more generally, significant computational power, is noted in the paper, GPUs being up to 50-100 times faster than CPUs and 2012's computers being 60000 times faster than computers from the 1990's. Training is performed using stochastic gradient descent (SGD); unlike other neural network architectures from the same historical period, no pretraining was required. The CNNs have small receptive fields, a feature which was also found beneficial for more modern CNN architectures like VGG [171], used for object recognition. The CNNs are deep, containing 6-10 layers; [64] notes that these numbers are comparable to the number of layers between the retina and the visual cortex of macaque monkeys. Data augmentation is also used, with random distortions applied to the MNIST images. The ensemble contains 35 neural nets trained on 7 variations of the MNIST dataset with 5 different preprocessors. Its accuracy on the test set is 0.23%, comparable to the 0.2% accuracy of humans [133]. Besides the application on MNIST, state of the art results were also reported for the recognition of Latin and Chinese characters.

Dropout, an influential technique used for regularizing neural networks, which we discuss in Section 1.5.4.1, was also tested on MNIST [178].

[86], which introduced the popular rectified linear unit (ReLU) activation function (see subsection 1.2.2 and equation 1.8), was also benchmarked on MNIST; the ReLU nonlinearity, when used with a feedforward fully-connected neural network, was shown to be significantly better than the tanh and softplus nonlinearities (the softplus

nonlinearity is a smoothed version of ReLU). [86] is also historically important because it showed that purely supervised learning can work just as well as unsupervised pretraining, so unsupervised pretraining isn't really required when enough labeled data is available in a supervised learning setting.

Batch normalization, a popular regularization technique introduced by [116] and presented in subsection 1.5.4.2, was also shown to accelerate the training of a fully-connected three-layer neural network on MNIST, while also leading to higher test accuracy.

ADAM [125], probably the most popular optimization algorithm for neural networks currently, presented in subsection 1.5.3.1, was shown to lead to faster training for two-layer fully-connected neural networks on MNIST, compared to other state of the art gradient-based optimization algorithms.

Many of the recent neural network architectures (and particularly convolutional architectures) have been tested on MNIST. A good reference for the results of modern architectures is [6].

[164] introduced capsule networks, which can be interpreted as a variant of convolutional neural networks with a different type of pooling (than e.g. max pooling used in CNNs), in which multiple neurons (units) in a layer are grouped into 'capsules'. The performance of capsule networks was highlighted on MNIST, where they obtained 0.25% error, compared to 0.39% for a convolutional neural network baseline, while also using less than half of the trainable parameters of the CNN.

2.1.2 Other tasks and datasets

Even if MNIST classification is by far the best-known handwriting-related task in the context of machine learning, it is not the only one. We briefly present four other tasks in this subsection.

2.1.2.1 MNIST for benchmarking generative models

This task consists of binarizing the MNIST images and measuring the log-likelihood (on the test set) of generative models trained to generate samples from the data distribution. It was introduced in [128] and has been quite influential for benchmarking generative models, one of the research areas in deep learning where a lot of effort has been spent recently. A good source of some state of the art results for this task is [89].

2.1.2.2 Pixel by pixel MNIST

The pixel by pixel MNIST task consists of classifying the images which are input (sequentially) pixel by pixel. It was introduced by [129]. The effect of this input procedure is to create long time dependencies: the systems have to remember the inputs for much longer. For images of size 28×28 (like MNIST), this corresponds to sequences of length 784, compared to length 28 if the images were input one column of pixels per timestep. Long-term dependencies are difficult to model by RNNs and other systems working with temporal sequences and they motivated the introduction of many influential neural network architectures, such as LSTMs [112]. This task also has a permuted variant, where the pixels of the image are permuted (all images are permuted using the same permutation) before being input, so as to remove any prior (structural) information contained (e.g. closer pixels are more correlated than distant ones). Permuted pixel by pixel MNIST is more difficult than its unpermuted variant. A large number of recent deep learning papers, most proposing novel RNN architectures, use this task as a benchmark [56] [40] [204][127] [197] [72] [120] [192] [145] [107] [106] [188].

We propose a novel recurrent architecture using quantum computation in Chapter 6 and benchmark it on a similar task (permuted pixel by pixel), but we use a dataset of more complex images - Fashion-MNIST [198].

2.1.2.3 Recognizing multilingual handwritten sequences

In 2009, Multi-dimensional LSTMs (MDLSTMs), a recurrent neural network architecture adapted to processing images, won three competitions at the International Conference on Document Analysis and Recognition (ICDAR) [21], on handwritten Arabic word recognition [142], handwritten Farsi/Arabic character recognition [148] and handwritten French word recognition [99]. These results were very influential in showing that RNNs can outcompete previously established methods such as Hidden Markov Models (HMMs) and that end-to-end learning can surpass handcrafted features.

2.1.2.4 Online handwriting sequential generative models

The IAM On-Line Handwriting Database [20], which contains handwritten English text acquired from a whiteboard, has been used recently to benchmark the generation of long sequences with complex structure (of which handwriting is an example). [94] showed the first impressive results on this task, using LSTMs, with the generated

handwriting often indistinguishable from human handwriting. A demo of the outputs of the system can be accessed at [32]. More recent papers using this benchmark include [63] [62] [57].

2.2 The history of neural networks applied to handwriting recognition

In this subsection, we will first briefly present some of the most significant datasets used to benchmark handwriting recognition systems. We will then briefly discuss the historical evolution of end-to-end trained systems for handwriting recognition tasks. We don't aim to cover the entire literature (which is vast), but present the major (especially recent) milestones.

2.2.1 Datasets

In this subsection we will briefly present the IAM, RIMES and IFN-ENIT datasets, which are some of the most popular handwriting recognition datasets, used by many of the papers we present in this section for benchmarking.

2.2.1.1 IAM

The IAM dataset [143] [19] contains images of handwritten English text. It provides both images of segmented lines and, separately, images of segmented words, split into training, validation and test sets. The lines are partitioned into sets of 6482 (training), 976 (validation) and 2915 (test) lines, respectively. The words are partitioned into sets of 55081 (training), 8895 (validation) and 25920 (test). Of these, only some words are correctly segmented and marked so in the database: 47952, 7558 and 20305 for the training, validation and test sets, respectively.

2.2.1.2 RIMES

The RIMES dataset [31] contains images of handwritten letters in French (simulated scenarios). The training set contains 11333 lines and the test set contains 778 lines. It includes 51739 words in the training set, 7464 words in the validation set and 7776 words in the test set, respectively.

2.2.1.3 IFN-ENIT

The IFN-ENIT dataset [22] [77] contains 32492 images of Tunisian cities and villages names, handwritten by more than 400 writers under a large variety of writing styles. It is divided into five sets, A-E. In the most common setting, sets A-D (containing 26459 images) are used for training and set E (containing 6033 images) is used for testing.

2.2.2 Deep neural networks (DNNs)

DNNs are the simplest architectures (no recurrence), using the least prior information about the data. They can still be effective for handwriting recognition tasks, at least in some cases and under certain conditions, as shown by the papers reviewed here.

[65] showed that DNNs (MLPs) are sufficient to obtain error rates comparable to those obtained using CNNs on MNIST, as long as the MLPs are large enough and sufficiently large quantities of augmented data can be provided. A significant factor in the required scale up of the DNNs is the use of GPU cards. [65] motivate their choice of DNNs as a much simpler architecture than e.g. CNNs and their work as the first trying to answer the question: are really big MLPs sufficient (at least for MNIST)? The MLPs are trained using stochastic gradient descent. Data augmentation is performed online and continuously and parts of the augmentation code are also accelerated on a graphical processing unit (GPU), with the rest implemented on CPU. The operations used for the data augmentation are affine transforms (rotation, scaling and horizontal shearing) and elastic deformations. The GPU is reported to accelerate the neural network by a factor of 40 and the data augmentation by a factor of 10 (compared to a CPU). The test errors obtained are in the range of 0.4%, better than the state of the art at the time (including CNNs). The authors note the importance of progress in compute power, which accelerates by 100-1000 times per decade (Moore's law) and interpret their results as suggesting that hardware improvements might be even more important than software or algorithmic improvements (though, ideally, we want to combine them all).

[50] compared deep MLPs trained with a sequential criterion (state-level Minimum Bayes risk) against BLSTMs as optical models. For both types of systems, handcrafted (geometric and statistical) and pixel features were compared. The comparison was performed on the RIMES and IAM datasets. Overall, similar error rates were observed for all the tested systems. Though the image preprocessing is claimed to be standard, it is quite complicated, involving deskewing, deslanting, some form of contrast

normalization, white pixels added and handwriting region rescaling. The MLP training procedure is also complicated by unsupervised pretraining and GMM-HMM bootstrapping. Adding depth through multiple layers is found to be helpful for both MLPs and RNNs, and pixel and handcrafted features, with higher impact for the pixel features. The performance of the different systems (RNN and deep MLP) and different features (handcrafted and pixels) is found to be comparable; n-gram word language models are used. The RNN and deep MLP are also complementary, as their ROVER combination achieved state of the art performance at the time of the publication.

2.2.3 Recurrent Neural Networks (RNNs)

RNNs came into mainstream use for handwriting recognition tasks after the results obtained by [97], which were also influential in showing the power of RNNs more generally, beyond handwriting recognition. Even if the MDLSTM architecture contains convolutional elements, we shortly discuss it here rather than in Section 2.2.4 because of its historical importance for the adoption of RNNs. We discuss some further papers using MDLSTMs in Section 2.2.4.

[97] showed that Multi-dimensional LSTMs (MDLSTMs), a recurrent neural network architecture adapted to images, obtained state of the art performance on the IFN-ENIT dataset at the ICDAR 2009 Arabic handwriting recognition contest. Multi-dimensional RNNs (MDRNNs), of which MDLSTMs are a variant, were introduced in [96]. The generality of this architecture was highlighted by the fact that none of the authors understood even a single word of Arabic. The proposal of MDLSTMs led to the previous state of the art handwriting recognition systems, based on Hidden Markov Models (HMMs), being superseded by deep learning architectures. As of 2018, MDLSTMs are still strong competitors for obtaining state of the art handwriting recognition systems, second only maybe to Convolutional Recurrent Neural Networks (CRNNs), which we discuss in the next subsection.

[202] showed that a BLSTM can be more accurate than MDLSTMs, when the input images are normalized (while learning directly from pixels). This approach also improved upon methods which use handcrafted features. It was tested on Arabic words from the IFN / ENIT dataset. We discuss this approach in more details in Chapter 5, as we will use a different RNN architecture on the same dataset, processed in the same manner.

[59] proposes a four layer bidirectional Gated Recurrent Unit (GRU) with dropout for Arabic handwriting recognition. GRU is a gated recurrent neural network

architecture similar to the LSTM, but somewhat simplified, making it potentially faster and less prone to overfitting; it was first introduced by [61]. The network is tested on the IFN-ENIT database (similarly to [202] and our own work in Chapter 5), where it is benchmarked against a three-layer BLSTM and shows better performance. The use of dropout is claimed to improve the system's generalization ability. The IFN-ENIT images are preprocessed similarly to [202], by performing dewarping and rescaling to fixed height of 48 pixels. The gain in accuracy is of almost 1% absolute CER (character error rate) compared to [202].

[47] studies the effect of applying dropout at different positions in textline handwriting recognition systems. The systems tested are 3-layered alternating BLSTMs and feedforward layers with tanh activations, trained with the CTC loss function. The datasets used are IAM, RIMES and Bentham [2] and two different setups are used: one for handcrafted and another for pixel features. The results seem somewhat inconclusive, as there's some inconsistency in the results obtained for different datasets. For handcrafted features, dropout seems best applied before the first two layers and after the last one. For pixel features, the best performance is achieved when applying dropout both before and after all of the layers. Overall, the best results seem to be obtained when dropout is applied close to the inputs and close to the outputs.

2.2.4 Architectures mixing convolutional and recurrent layers

More recently, after 2009, architectures mixing convolutional and recurrent layers have been obtaining the state of the art results on the most notable handwriting datasets (like IAM and RIMES) and at ICDAR and ICFHR competitions. Even more recently, after 2016, attention-based models have become competitive against the CTC loss and have shown that full end-to-end paragraph transcription (instead of line transcription) could become feasible in the near term. Also notably, after 2016, MDLSTMs are being outperformed by Convolutional Recurrent Neural Network (CRNN) architectures. We will briefly discuss these developments in this subsection.

[48] proposed the first attention-based model for end-to-end handwriting recognition and the first successful attempt of end-to-end multi-line handwriting recognition, by combining attention and the Multi-dimensional LSTM (MDLSTM) architecture, which contains both convolutional and recurrent computational primitives. The system works directly at the paragraph level, without requiring any explicit segmentation

of the paragraph into lines and outputs a sequence of characters. The motivation is to perform more end-to-end processing of handwriting documents, instead of separately segmenting lines using another system in a more complex pipeline. This can be advantageous, since any segmentation errors are difficult to correct by a system working on segmented lines. Another motivation can be the capability of the system to learn to read documents in different orders (e.g. left-to-right for Latin scripts, right-to-left for Arabic scripts, even mixed when combining the two different types). The attention model is inspired by a similar attention model introduced initially for the problem of machine translation [39]. The algorithm was tested on the IAM dataset. While the CER obtained when working directly on paragraphs was worse than state of the art approaches working at the line level, the results were encouraging and showed the viability of the proof of concept. Another interesting result is the competitiveness of attention, combined with the simple cross-entropy loss function, against CTC; when applied on IAM lines of handwritten text, the system obtains results comparable to state of the art MDLSTMs with the CTC loss. Beyond these results, extensions of this system could be expected to perform integrated end-to-end document layout analysis and text recognition. Furthermore, while the current system doesn't take into account the previous character prediction, it could be extended in this manner; this could also allow integrating a (pre-trained) language model.

[46] (like [48]) is also motivated by the idea of end-to-end processing of handwritten paragraphs, instead of separately segmenting lines in a complex pipeline. Like [48], it also uses MDLSTMs with attention. The main difference (compared to [48]) is that in this paper it is no longer the case that the output characters are predicted one at a time, which would be quite inefficient computationally. Instead, an iterative decoder is used which requires one step per handwriting line (compared to one step per character in [48]), leading to a speedup of a factor of 20-30. Attention is also used here, but only to separate paragraphs into lines, not when predicting the characters corresponding to the image of a handwritten text line. Also differently from [48], the CTC loss function is used (instead of the more simple cross-entropy). The proposed method is tested on the IAM and RIMES datasets. On RIMES, it obtained the state of the art CER at the time of the publication. On IAM, it obtained results comparable to the state of the art at the time, though somewhat worse. Maybe even more significantly, the model was shown to obtain better accuracy than models based on explicit, automatic line segmentation and results comparable or better than when using the ground-truth segmentation.

[49] introduces a model based on gated convolutional layers and BLSTMs to predict character sequences. The convolutional part, which they denote as a convolutional encoder, is useful to produce multilingual features. The gating output y is produced through element-wise multiplication, similar to the gating mechanism in LSTMs (see Subsection 1.3.3.2): $y = g(x) \bullet x$ where $g(x)$ is produced through convolution on the feature maps x, followed by the sigmoid nonlinearity. The convolutional encoder is reused between different languages. It is also motivated by its ease of parallelization (especially on GPUs). This model is shown to improve upon previous models (notably, MDLSTMs) on a large multilingual database, and also produces state of the art results on IAM and RIMES (when combined with hybrid word-character n-gram language models), at both line and paragraph levels.

[76] introduces an attention-based model and tests it on the RIMES dataset. The architecture is composed of an encoder and a decoder with attention. The images are first encoded using a convolutional network, then passed through a three-layer BLSTM. The decoder combines a content-based and a location-based form of attention. Using an attention model is motivated by the difficulty of CTC and HMM models to deal with output sequences which are longer than the input sequences. An element of novelty is using a BLSTM (instead of a LSTM) as the decoder. This is motivated by the BLSTM's improved accuracy on handwriting tasks when compared to LSTMs. To make the decoding possible when using a BLSTM, the sequence length must be known in advance; this is predicted using a separate MLP. The performance of this model is evaluated on the RIMES dataset, for both words and textlines. While the performance is below than that of a hybrid LSTM-HMM system, it is comparable.

In [157], a model composed of convolutional layers, followed by a deep BLSTM, is proposed and shown to be at least as accurate, and even improve upon the results of MDLSTMs. The proposed model is motivated by the improved computational complexity and capability to parallelize, which are also demonstrated empirically. An extensive empirical comparison is performed between the proposed model and MDLSTMs on the IAM and RIMES datasets, without any language model. The model is shown to be at least as accurate with statistical significance, both in terms of CER (character error rate) and WER (word error rate). To compare against the state of the art systems in the literature, image augmentation and word-level language models are used. The performance obtained is slightly worse, but close to the state of the art for IAM. Interestingly, for RIMES, better than state of the art performance is obtained.

[182] proposes an encoder-decoder [61] (also known as sequence to sequence -

Seq2Seq [183] - because it takes as input a sequence and outputs another sequence) architecture with attention. The architecture is composed of a LeNet-like CNN and a Seq2Seq model with attention [183], with the CNN applied over a sequence of image patches obtained from images of handwritten words using a horizontal sliding window. Both LSTM and BLSTM architectures are benchmarked for the Seq2Seq encoder. The model outputs separate characters. To obtain a word (from the closed dictionary), the Levenshtein distance is used and the closest word from the standard dictionary is output. The model obtains competitive results against the state of the art on the IAM and RIMES datasets and, particularly, the best published results without language model and with test lexicon.

The recent competitiveness of CRNNs (particularly against MDLSTMs) has been convincingly demonstrated in the ICFHR2018 Competition on Automated Text Recognition on a READ Dataset [181], targeting the adaptation of recognition engines to small, new, multilingual data. Five of the total six submissions and the best five systems used CRNNs (more precisely, CNNs and BLSTMs), while the sixth system used MDLSTMs. Intriguingly, the winning system didn't use any language model. More generally, the evaluated systems seemed to obtain similar accuracies with or without language models. The submitted systems seemed able to make use of even small new document-specific data, decreasing the CER by 50% with only 16 pages of new document-specific data. The systems winning the first and second place used very similar architectures, combining CNN and BLSTM layers and the CTC loss function. The main difference between the two is the use of more extensive augmented data by the winning system. We illustrate the recognition system which won second place and which is described in [54] in Fig. 2.1. This architecture contains thirteen convolutional layers (architecture inspired by the VGG CNN [171]), followed by three BLSTM layers with 256 units per layer. As shown in Fig. 2.1, it makes use of ReLU nonlinearities, batch normalization and max-pooling. The system is trained end-to-end using the CTC loss function. The training (and validation) data was augmented using multiscale representations. A bigram word-level language model was integrated with this architecture.

2.3 Conclusion

In this chapter, we have discussed the interaction between deep learning and handwriting recognition: how handwriting recognition tasks have played a major part in the resurgence of neural networks and, conversely, how recent neural network

architectures have led to state of the art handwriting recognition performance. In the next chapter, we present our first contribution, applying Convolutional Neural Networks to the recognition of handwritten digits from the MNIST dataset mentioned in this chapter.

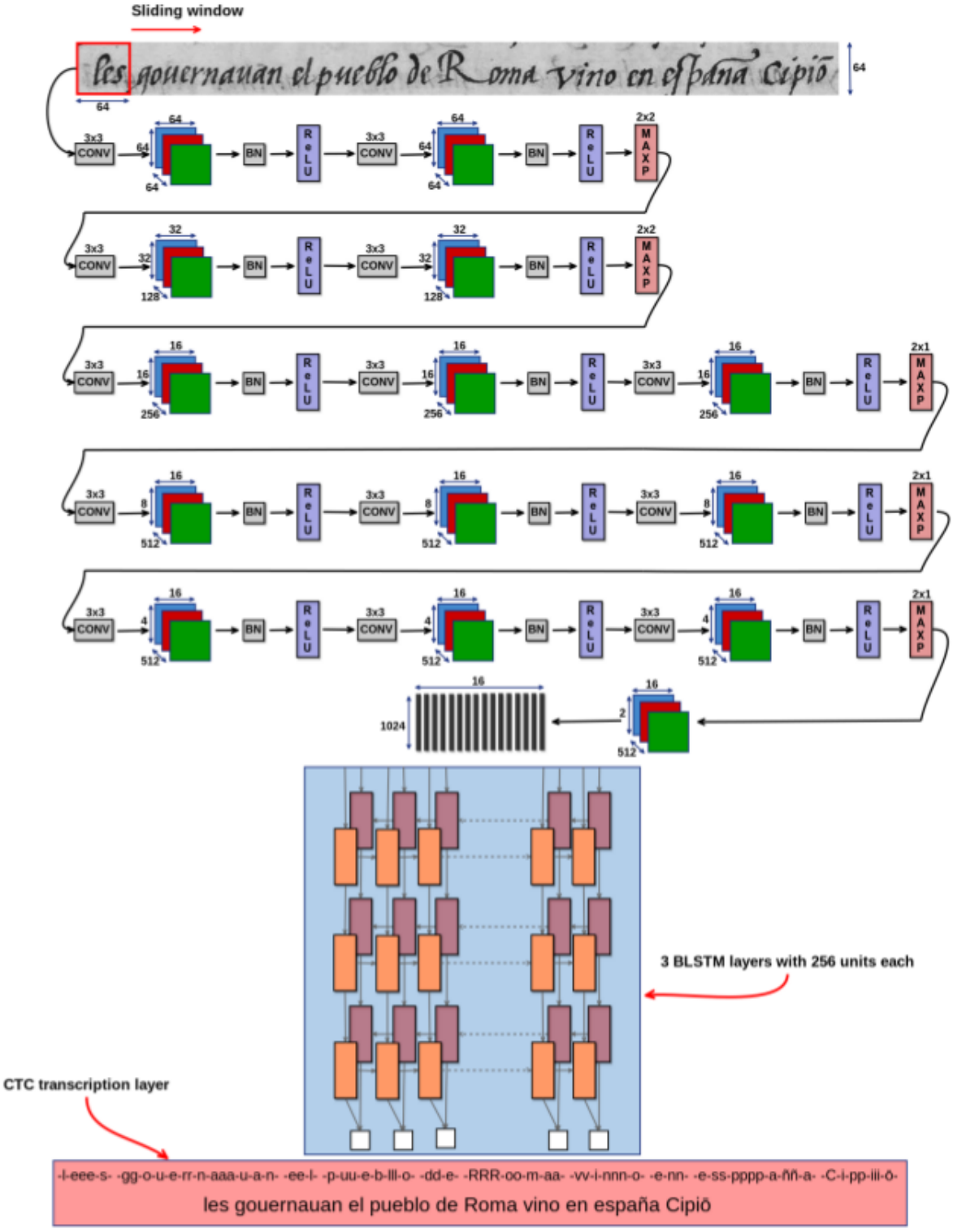


Figure 2.1 – Architecture from [54]. CONV denotes convolutional layers. BN denotes Batch Normalization. MAXP denotes max-pooling.

Chapter 3

Improving a deep convolutional neural network architecture for character recognition

Contents

In this chapter we report the first contribution of this thesis. We start from a deep convolutional architecture and we describe the effect of recent (at the time the work was performed) activation functions, optimization algorithms and regularization procedures when applied to the recognition of handwritten digits from the MNIST dataset [132]. The MNIST dataset is popular for this task and a variety of approaches have been compared using it. This work was presented in [68].

We only focus here on those approaches which take into account and exploit information about the spatial structure of images and which use deep learning, while discarding the permutation-invariant MNIST task (in the permutation-invariant task, no prior information about the spatial arrangement of the input pixels is available). Many of the most recent and most successful proposals also use deep convolutional neural networks, as we do [134] [141] [90] [88] [136] [64]. Other approaches use recurrent neural network variants, such as the Multi-dimensional long short-term memory (MDLSTM) [93] or ReNet [191]. While these approaches encode information about the spatial structure of the images mainly through the architecture design, other approaches do so mainly through data augmentation, by using image pixels shift, scaling, or elastic distortion, in order to help the trained systems become more invariant to these transforms and to artificially augment the training set. The most notable of these approaches is [66], where the authors train big simple feedforward neural networks (also known as multilayer perceptrons - MLPs) using elastic distortions. As is true for many other tasks, training ensembles of classifiers often results in better performance than training a single classifier [64] [66] [194]. We focus here on the setting of a single classifier (no ensemble) and no data augmentation.

3.1 Architecture

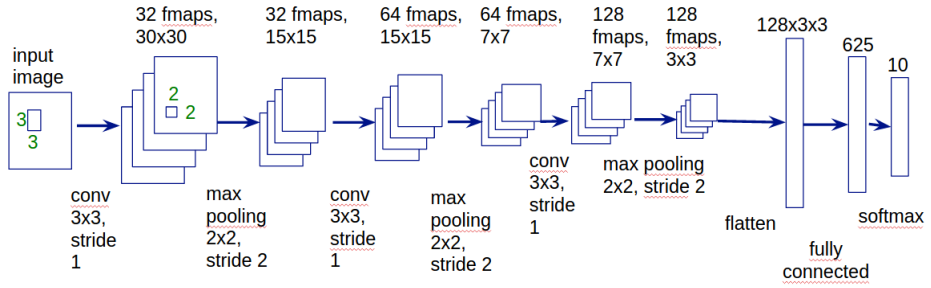


Figure 3.1 – CNN architecture

The convolutional architecture we use is shown in Figure 3.1. It follows many of the guidelines from [171], which has obtained the best single-model performance in the ILSVRC 2014 object classification competition [163]. Each of the convolutional layers has filters of size 3x3, with stride 1 (the stride is the distance between the centers of the receptive fields of neighboring neurons in an activation map). The subsampling layers are always 2x2 max-pooling with stride 2, which results in subsampling both image height and width by 2. When choosing the number of units in the convolutional

Input size	**Convolutional Layer 1**
1 x 28 x 28	conv: 3 x 3 full, stride 1, 32 feature maps
32 x 30 x 30	batch normalization
32 x 30 x 30	PReLU
32 x 30 x 30	2 x 2 max pooling, stride 2
32 x 15 x 15	0.5 dropout
	Convolutional Layer 2
32 x 15 x 15	conv: 3 x 3, stride 1, 64 feature maps
64 x 15 x 15	batch normalization
64 x 15 x 15	PReLU
64 x 15 x 15	2 x 2 max pooling, stride 2
64 x 7 x 7	0.5 dropout
	Convolutional Layer 3
64 x 7 x 7	conv: 3 x 3, stride 1, 128 feature maps
128 x 7 x 7	batch normalization
128 x 7 x 7	PReLU
128 x 7 x 7	2 x 2 max pooling, stride 2
128 x 3 x 3	flatten
128 x 3 x 3	0.5 dropout
	Fully Connected Layer
128 x 3 x 3	fully connected layer matrix multiplication
625	batch normalization
625	PReLU
625	0.5 dropout
	Softmax Layer
10	softmax

Table 3.1 – Proposed CNN architecture

layers, we are once more inspired by the guidelines outlined in [171]: we double the number of units between each consecutive layer, from 32 in the first convolutional layer to 64 then 128 in the last one. The architecture also contains a fully-connected layer of 625 units and a softmax layer of 10 classes (corresponding to the 10 digits). We show the architecture in Table 3.1.

3.2 Nonlinear activation functions

We have tried several nonlinear activation functions, such as ReLU 1.8 (see Equation 1.9, LReLU (see Equation 1.9) and PReLU. For more details about activation functions in general, as well as the specific ones mentioned above, see Subsection

1.2.2. We have obtained the best results using the PReLU activation function, by allowing each convolutional filter to have its own a value, but tying the values between different spatial locations of a same convolutional filter.

3.3 Gradient-based optimization and loss function

The loss function we minimize is the average negative log-likelihood of the conditional distribution of the correct label given the input, $\log(p(y|x)$, across training examples (x, y) from dataset S. This is equivalent to maximizing the multinomial logistic regression objective. For more details, see Subsection 1.5.1.1. The optimization is gradient-based and the gradients are computed using the backpropagation algorithm [162]. We use mini-batches, which can reduce the computational time required by the optimization procedure, especially when using modern computational architectures and even more so for GPUs than for CPUs.

3.4 Initialization

Initialization is important because bad initialization can lead to instability in the gradients (vanishing or exploding gradients) and consequently poor model performance after training. In contrast, good initialization can accelerate convergence speed.

We have obtained the best results using the initialization method described in [105]. For simplicity, we initialize the PReLU a parameters with zero values (right after this initialization, PReLUs behave just like ReLUs). We initialize the batch normalization γ parameters to 1, and the β parameters to 0 (see Section 3.7). The neuron weight matrices are initialized using values drawn from Gaussian distributions with mean 0 and standard deviation $\sqrt{\frac{2}{n_i}}$, where n_i is the number of inputs to the neuron, following the procedure in [105], and the biases are initialized to 0.

This procedure helps avoid both the exponential vanishing and exploding of gradients, as well as of inputs to each layer during the feedforward phase. We refer you to [105] for the full mathematical treatment.

3.5 ADAM variant

For an introduction to ADAM, see Subsection 1.5.3.1.

ADAM is supposed to help with optimizing stochastic functions. In our case, the stochasticity comes from mini-batch training and batch normalization, from dropout,

Algorithm 7 ADAM (variant)

Legend:
t = timestep
θ_t = neural network parameter values at timestep t
lr = learning rate
$decay$ = learning rate decay
$init$ = initial learning rate value
ϵ = hyper-parameter to make the algorithm numerically stable

Algorithm:
$lr = init$
while θ_t not converged do:
    g_t = gradient of loss function with regard to θ_t
    m_t = moving average of g_t
    v_t = moving average of g_t^2
    $\theta_t = \theta_{t-1} - lr * \frac{m_t}{\sqrt{v_t}+\epsilon}$
    $lr = lr * decay$

as well as from the inherent noise in the function to be learned (which takes as inputs images and outputs labels). We briefly describe the algorithm as we have used it in Algorithm 7. The only modification we have made from the basic algorithm as described in Subsection 1.5.3.1 and [125] is using a schedule for decaying the learning rate. A detailed description of ADAM is beyond the scope of this paper; we refer the reader to the relevant publication [125].

One aspect we have found empirically is that the learning schedule (the way the learning rate is adjusted during the optimization procedure) is critical in determining both the speed of learning (the number of epochs required), as well as the final performance (as measured by the validation accuracy). In spite of the fact that the authors of [125] claim that this optimization method is quite robust to step size, we have found that adapting the learning rate during learning still helps. We thus propose a variant of ADAM which consists in using annealing (multiplying the learning rate by a constant after each epoch of training). This corresponds to the last equation in Algorithm 7. An intuitive explanation for why this procedure works well is that, as the learning rate decays, the optimization takes shorter steps, doing less exploration, so that it eventually settles into a local minimum [178].

3.6 Dropout

For a short introduction to dropout, see Subsection 1.5.4.1. For more details, see the original paper [178].

3.7 Batch normalization

For a short introduction to batch normalization, see Subsection 1.5.4.2. For more details, see the original publication [116].

We apply batch normalization to every training mini-batch and we shuffle the training set, as recommended in [116]. Training with batch normalization leads to each example being seen in conjunction with other examples in the mini-batch, so that the training procedure no longer produces deterministic values for a given training example. This can be seen as having a regularizing effect similar to using dropout. Applied naively, batch normalization only adds two extra parameters (γ and β) for each neuron, so the risk of over-fitting as a result of an increase in the number of parameters is low. As a result of adding the batch normalization β parameters, the bias parameters are no longer necessary, so we no longer use them in the layers in which we use batch normalization; therefore, the effective number of parameters only increases by 1 per neuron. When using the validation set, we no longer split it into mini-batches, but use it in its entirety. This has the advantage of removing the randomization introduced by batch normalization (when using mini-batches) when estimating the system's accuracy on the validation set. The same procedure is applied when testing the system's final performance (on the test set).

3.8 Early stopping

We have used early stopping (see Subsection 1.5.4.3 for a description) when performing optimization: we run the optimization algorithm for a maximum 300 epochs (maximum 300 passes through the entire dataset using mini-batches) and stop after 30 epochs without improvement on the validation set. Another method we have tried, but which has worked worse empirically, was to go through a fixed set of learning rates (e.g 0.1, 0.01, 0.001) in decreasing order, by decreasing the learning rate when the validation accuracy doesn't improve after a given number of epochs (e.g. 30). Note that we do not retrain the model using both the training and validation samples.

3.9 Experiments

The MNIST dataset of 28 x 28 images is split in 3 separate sets: a training set of 50000 images, a validation set of 10000 images and a test set of 10000 examples.

As is standard in the literature ([134], [136]), the only preprocessing we perform is scaling the inputs to [0, 1] values. We have also tried standardizing the inputs using global contrast normalization, but haven't seen improved results.

For the optimization procedure, we have used ADAM, keeping all its hyper-parameters to their default values in [125], except for the learning rate, which was annealed during training. We obtain our best system using an initial learning rate of 0.005 and a learning rate decay of 0.98.

As regularization, in the best system we have used dropout [178] with probability of removing a neuron $p = 0.5$, in both the convolutional and the fully connected layers (we don't apply dropout to the input images). Though one might expect that overfitting is not a problem for convolutional layers, since they have few parameters (as a result of the local connection patterns and the tied weights), dropout can still help by providing noisy inputs to the higher fully connected layers, preventing them from overfitting [178]. We have also tried values of $p = 0$ (corresponding to no dropout), $p = 0.2$ and $p = 0.8$.

Notice that we haven't performed a systematic, extensive hyper-parameter search, due to limitations in access to computation (GPUs). Using random search [44] or Bayesian optimization [172] might lead to improved results, as has been observed empirically in [172]

Model	Test error (%)
Ours	**0.38**
Deeply-Supervised Nets [134]	0.39
Fractional max-pooling [90]	0.44
Maxout Networks [88]	0.45
Network in Network [136]	0.47

Table 3.2 – Comparison with state-of-the-art results on the MNIST test set (single system, no data augmentation) at the time this work was originally performed (May 2015)

The error rate obtained using our system is shown in Table 3.2. The same table also shows results of other deep learning approaches which obtain state-of-the-art results on MNIST without using data augmentation and model ensembles. The misclassified digits are shown in Fig. 3.2.

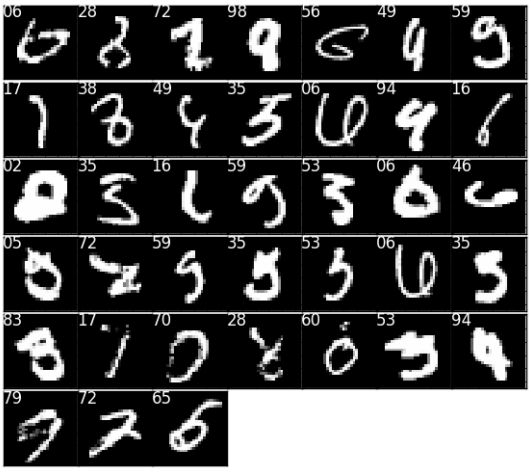


Figure 3.2 – All misclassified samples of the MNIST test set. The first number is the estimated label, the second one is the ground truth.

The network achieves a 0.38 % error rate, matching and slightly improving the best known performance of a single model trained without data augmentation at the time the experiments were performed, in May 2015. As of February 2016, the best performance in this setting was 0.24 % [55].

3.10 Conclusions

We interpret our results as an argument for engineering systems which can perform automatic (learned) feature extraction, rather than using hand-engineered features. Using a deep learning approach also allows for deep learning innovations from other applications domains to be imported and applied with ease and the deep learning contributions can also be used in other application domains.

In the next chapter, we use CNNs as building blocks of more complex neural network architectures, which can perform both localization and classification.

Chapter 4

Tied Spatial Transformer Networks for Digit Recognition

Contents

We now report our second contribution: a new approach applied to CNNs, where we use spatial transformer networks (STNs) [117]. This work was published in ICFHR 2016 [70]. This chapter closely follows the presentation in [70].

In this chapter we use STNs (Spatial Transformer Networks), deep learning architectures that include two CNNs coupled using a Spatial Transformer module, which was first introduced by [117]. The first one, the so-called *localization* CNN, is dedicated to spatial transformations. The second one, the *classification* CNN, is dedicated to the classification of the transformed images. The input images are

transformed using the parameters estimated by the localization CNN and then fed into the classification CNN, which performs the class prediction.

The advantage of using such coupled systems (localization coupled to classification) is that the trainable parameters of the localization network are learned during the training phase, along with those of the classification network, so that images can be transformed and classified in an unified manner. Since localization and classification can be regarded as different tasks, the trainable parameters of the two networks would generally differ. We propose here to tie the trainable parameters of both networks in order to obtain a regularization effect which can improve performance. We apply our approach to the recognition of noisy digits, using the cluttered MNIST database [146], derived from the MNIST database [132] but including additional sources of noise.

This chapter is organized as follows. In Section 4.1, we describe the common elements of the three CNN-based classifier systems: the simple classification CNN, the untied STN and the TSTN. In Section 4.2, we provide comparative results for the three systems, using two different convolutional architectures (architecture 1 and architecture 2), the second architecture being more complex and powerful than the first one. Interpretations of our results are discussed in Section 4.3 and we conclude in Section 4.4.

4.1 Common elements

In this section, we present the common elements used between the classification CNN, the untied STN, and the TSTN. For simplicity and to make the comparison of the results easier, we have tried to keep as many elements as possible (convolutional architectures, activation function, optimization procedure, regularization) identical or as similar as possible between the three systems.

4.1.1 Convolutional architectures

All the classification systems presented in this paper are CNN-based. The CNN classifier includes a single block composed of convolutional and fully connected layers, while the STN and TSTN include two blocks, one for the localization part and another for the classification part (see Figures 4.1, 4.2 and 4.3). We perform experiments using two different architectures, the second one being more complex and more powerful than the first. These architectures are shown in Table 4.1 and

Table 4.2 and follow many of the guidelines outlined in [171]. The classification CNN with architecture 1 is shown in more details in Figure 4.4.

Figure 4.1 – Classification CNN architecture

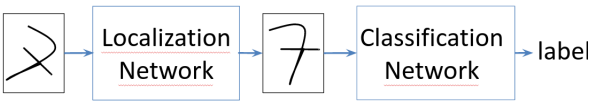


Figure 4.2 – Untied STN architecture, composed of two coupled CNNs, one dedicated to localization, the other to classification

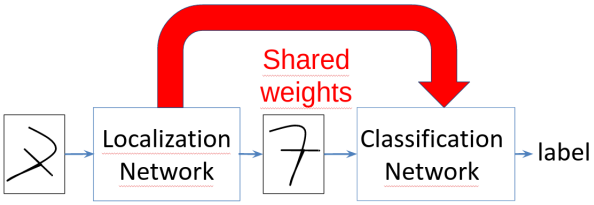


Figure 4.3 – TSTN architecture, similar to the untied STN architecture, but using shared weights

Each of the convolutional layers contains filters of size 3x3. 'Same' mode convolution (which allows the size of the output of each convolutional map to be equal to the size of the input) is used in all the layers. The subsampling layers are always 2x2 max-pooling with stride 2, which results in subsampled images with both height and width divided by 2. We double the number of convolutional feature maps between each consecutive convolutional layer, from 32 in the first layer to 64 then 128 in the last one, drawing once more inspiration from [171].

The fully connected parts are identical between CNN architectures 1 and 2, but different between the localization and classification CNNs. The convolutional parts are also very similar. The difference is that for architecture 2 we use two consecutive operations of convolution followed by nonlinearity instead of a single one; these are followed by pooling operations, for each of the three convolutional layers. This makes architecture 2 more powerful and also allows the implicit size of its convolutional

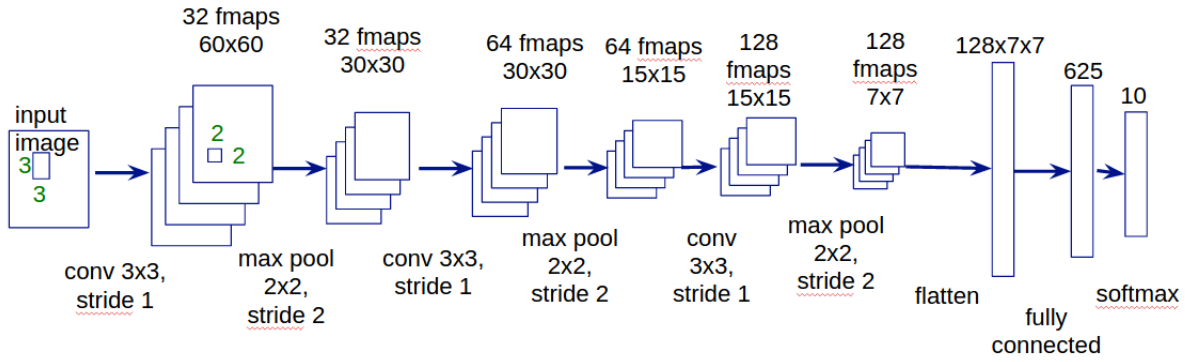


Figure 4.4 – Detailed classification CNN architecture 1 (fmap = feature map)

filters to be larger. In both architectures, we remove the pooling operation in the last convolutional layer of the localization CNN. The motivation for this architectural choice is to keep more of the spatial information that the fully connected layer of the localization CNN might need in order to output the estimated parameters of the affine transformation performed by the Spatial Transformer module. Notice also that since the max-pooling layers do not contain any trainable parameters, this means that the convolutional layers of the classification and localization CNNs can still have the same (tied) trainable parameters. For both architectures (1 and 2), the last layer of the localization CNN contains less units than that of the classification CNN, imitating the design of the architectures used in [176] and [174].

Generally, the transformation performed on the input image using the Spatial Transformer can include scaling, cropping, rotations, as well as non-rigid deformations [117]. In all of our architectures, we only use the affine Spatial Transformer, introduced in [117] and also used in [176]. It allows for all the transformations previously mentioned, except for non-rigid deformations, and only requires 6 parameters to be estimated by the localization network. The implementation we use, introduced and described in [176], also includes a subsampling factor d, by which both the height and the width of the image are scaled, after applying the affine transform. For simplicity and to make comparisons between different models (including between CNNs and STNs) easier, we use $d = 1$ in all of our experiments with systems containing Spatial Transformers. By performing affine transforms, systems integrating the Spatial Transformer module can both select relevant portions of an image (a form of attention) and transform those regions into poses which can simplify the recognition task for the following system components. For all the technical details about Spatial Transformers, we refer the reader to the relevant publication [117].

Localization CNN	**Classification CNN**
Convolutional Layer 1	**Convolutional Layer 1**
conv: 3 x 3 same, stride 1, 32 feature maps	conv: 3 x 3 same, stride 1, 32 feature maps
ReLU	ReLU
2 x 2 max pooling, stride 2	2 x 2 max pooling, stride 2
Convolutional Layer 2	**Convolutional Layer 2**
conv: 3 x 3 same, stride 1, 64 feature maps	conv: 3 x 3 same, stride 1, 64 feature maps
ReLU	ReLU
2 x 2 max pooling, stride 2	2 x 2 max pooling, stride 2
Convolutional Layer 3	**Convolutional Layer 3**
conv: 3 x 3 same, stride 1, 128 feature maps	conv: 3 x 3 same, stride 1, 128 feature maps
ReLU	ReLU
	2 x 2 max pooling, stride 2
flatten	flatten
Fully Connected Layer	**Fully Connected Layer**
200 neurons linear layer	625 neurons linear layer
ReLU	ReLU
	0.5 dropout
Affine Transform Layer	**Softmax Layer**
6 parameters	10-class softmax

Table 4.1 – Architecture 1: localization CNN (left) and classification CNN (right)

4.1.2 Activation functions and parameter initialization

For the activation function, we use in all of our experiments the Rectified linear unit (ReLU) [86], one of the most successful activation functions in deep learning.

For simplicity, we initialize all weight matrices using a method similar to the one proposed in [105] and which is also used in [174]. Weight matrices are initialized using values drawn from uniform distributions with mean 0 and standard deviation $\sqrt{\frac{2}{n_i}}$, where n_i is the number of inputs to the neuron. The biases are initialized to 0. The procedure, presented in detail in [105], helps avoid gradient instabilities during the training phase, as well as instabilities of inputs to each layer during inference. The affine transform layer parameters are initialized to the identity transform, like in [174].

Localization CNN	**Classification CNN**
Convolutional Layer 1	**Convolutional Layer 1**
conv: 3 x 3 same, stride 1, 32 feature maps	conv: 3 x 3 same, stride 1, 32 feature maps
ReLU	ReLU
conv: 3 x 3 same, stride 1, 32 feature maps	conv: 3 x 3 same, stride 1, 32 feature maps
ReLU	ReLU
2 x 2 max pooling, stride 2	2 x 2 max pooling, stride 2
Convolutional Layer 2	**Convolutional Layer 2**
conv: 3 x 3 same, stride 1, 64 feature maps	conv: 3 x 3 same, stride 1, 64 feature maps
ReLU	ReLU
conv: 3 x 3 same, stride 1, 64 feature maps	conv: 3 x 3 same, stride 1, 64 feature maps
ReLU	ReLU
2 x 2 max pooling, stride 2	2 x 2 max pooling, stride 2
Convolutional Layer 3	**Convolutional Layer 3**
conv: 3 x 3 same, stride 1, 128 feature maps	conv: 3 x 3 same, stride 1, 128 feature maps
ReLU	ReLU
conv: 3 x 3 same, stride 1, 128 feature maps	conv: 3 x 3 same, stride 1, 128 feature maps
ReLU	ReLU
	2 x 2 max pooling, stride 2
flatten	flatten
Fully Connected Layer	**Fully Connected Layer**
200 neurons linear layer	625 neurons linear layer
ReLU	ReLU
	0.5 dropout
Affine Transform Layer	**Softmax Layer**
6 parameters	10-class softmax

Table 4.2 – Architecture 2: localization CNN (left) and classification CNN (right)

4.1.3 Loss function and optimization

As loss function, we minimize the standard negative log-likelihood of the conditional distribution of the correct label given the input across mini-batches of training examples.

Each one of the systems we experiment with (classification CNN, untied STN, TSTN, for each of the two architectures) represents a differentiable function. To optimize each such function, we use gradient-based optimization methods, with the gradients being computed from mini-batches of examples, using the backpropagation algorithm.

We use as optimization algorithm a variant of ADAM [125], in which we multiply the learning rate by a fixed amount after a certain number of epochs. ADAM is a state of the art algorithm developed for improving the performance of stochastic gradient descent (SGD) on stochastic non-stationary loss functions and sparse gradients. We have presented ADAM in Section 1.5.3.1. Using a good optimizer can be important for the untied STN and TSTN systems, since each of them is approximately twice deeper than the corresponding classification CNNs alone, potentially leading to vanishing gradients as a consequence of the increased depth.

ADAM introduces a few additional hyper-parameters, described in detail in [125], which we have kept fixed, using the values recommended in [125]. The only parameter we vary is the learning rate, which we multiply by a fixed amount after a fixed number of epochs. We have followed the implementation provided by [174] and multiplied the learning rate by 0.7 after each 20 epochs of training. We use mini-batches of size 256 in all the training procedures.

4.1.4 Regularization

To reduce the risk of overfitting, we use dropout, which we have presented in Section 1.5.4.1. During validation and testing, the activations of the neurons are scaled by the same probability p. We use $p = 0.5$ dropout in the fully connected layer of the classification CNN only, because it is the layer with the largest number of trainable parameters, where regularization is potentially most useful. We don't use dropout in the fully connected layer of the localization network. The intuition behind this choice is that dropout, as a noisy form of regularization, tends to remove some of the spatial information present in the unit activations. This spatial information might be more valuable for the localization task (as compared to the classification one), so we prefer to preserve more of it (at the risk of overfitting).

4.2 Experiments

We have applied TSTNs to the recognition of noisy handwritten digits. The experiments are conducted on the cluttered MNIST database [173] which is derived from the MNIST database [132] but includes additional noise and transformations, which make the classification task more difficult. The digits are distorted using random translation, scale, rotation, and clutter, and the entire image corresponding to each digit is larger (60 x 60, compared to 28 x 28 for MNIST). The number of training / validation / test examples replicates the numbers in MNIST (50000 / 10000 / 10000). We show upscaled image samples from cluttered MNIST and the processing performed by a Spatial Transformer in Figure 4.5.

The cluttered MNIST database was introduced by [146]. The best test error obtained on this dataset, to the best of our awareness, is 1.7%, using a CNN with a Spatial Transformer module [117]. This result is not directly comparable to ours, due to differences in the training procedure (e. g. the optimization algorithm). Furthermore, we do not necessarily aim to achieve the state of the art, but rather to provide preliminary proof that tying the trainable parameters of the localization and the classification CNNs can provide some benefit.

For our implementation, we use Theano [186] and Lasagne [75] and rely heavily on the following implementation [175] and example [174] of Spatial Transformer Networks.

Figure 4.5 – Cluttered MNIST input images (left) and the same images processed by a Spatial Transformer as part of a STN1 system (right)

4.2.1 CNN, STN and TSTN comparison

We compare in Table 4.3 the proposed TSTN network (tied trainable parameters) against the untied STN and the classification CNN alone, using each of the two different architectures on the cluttered MNIST dataset.

We have stopped the training of each system manually, after checking that the training accuracy is (almost) perfect. In a single case, the untied STN architecture 2 (entry 8 in Table 4.3), we have not been able to get the training procedure to converge so as to perform comparably to the other systems. To highlight that the number of epochs of training is comparable and that we have allowed each system a reasonable amount of training to achieve good performance, we also show in Table 4.3 the number of training epochs and the best training accuracy (achieved up to the maximum training epoch). For each system and training procedure, we choose the parameters for which the best validation accuracy is obtained for evaluation on the test set. A single experiment is performed for each system, with the same random seed used in all experiments.

As can be seen in Table 4.3, the TSTN systems obtain better results compared to CNNs and untied STNs, for the same architecture (by comparing between entries 1-3; and between entries 7-9, respectively).

Entry	Maximum training epochs	Best training error (%)	Validation error (%)	Test error (%)
1. CNN1	145	0	4.39	4.44
2. STN1	179	0	4.38	4.33
3. TSTN1	196	0.002	3.02	3.14
4. TSTN1-2	127	0.032	4.14	4.25
5. TSTN1-5-20	135	0	3.11	3.15
6. TSTN1-2-20	127	0.002	2.73	2.74
7. CNN2	95	0	1.82	2.18
8. STN2	66	88.64	88.74	88.81
9. TSTN2	231	0	1.30	1.74
10. TSTN2-5-20	141	0	1.24	1.50
11. TSTN2-2-20	178	0	1.22	1.31

Table 4.3 – Results on the cluttered MNIST database for different systems, architectures and training procedures

4.2.2 The regularization hypothesis

One hypothesis we propose for the improved performance of the TSTN is that, compared to the untied STN, it is better regularized. To test this hypothesis, we have conducted a series of experiments where we train each of the two TSTN architectures using a curriculum like-approach: we alternate between training the entire TSTN during some epochs and training only its classification CNN component during some different epochs. We test the regularization effect on the tied trainable parameters of the TSTNs by using only the classification CNN components during the validation and test phases. The results of these experiments are shown in entries 4-6, 10 and 11 in Table 4.3.

The first such experiment is conducted using architecture 1, by training the entire TSTN during the odd epochs and training only the classification CNN during the even epochs. The results, shown in entry 4 (TSTN1-2) in Table 4.3, are slightly better than for entries 1 and 2, supporting the regularization hypothesis, but they are worse than those of entry 3.

We have conjectured that the performance could be improved if we allowed for some initial training epochs for the entire TSTN before alternating between training the TSTN and classification CNN. Since the TSTN architecture is twice as deep as the classification CNN, the vanishing / exploding gradient problem is potentially worse; this is especially true for the parameters of the localization CNN, which is the farthest from the error signal provided during backpropagation at the softmax layer. To test this hypothesis, we perform the training procedure according to the following scheme: the entire TSTN is trained for the first 20 epochs, then we alternate the training of the entire TSTN every 5 epochs and the training of the classification CNN during the remaining epochs. This corresponds to the TSTN1-5-20 system (entry 5) of Table 4.3; the performance seems to be improved for architecture 1, becoming similar to the TSTN's.

The effect of the frequency of training the entire TSTN system, while only using the classification CNN during validation / test could also be related to the trade off between how much we weigh a regularization term against the likelihood term in the training objective. Training the entire TSTN more often would correspond to using more regularization. Viewed in this manner, we can compound the regularization provided using the approaches described in the previous two paragraphs by training the entire TSTN during the first 20 epochs, then alternating between training the entire TSTN and the classification CNN every 2 epochs. The results (TSTN1-2-20) for architecture 1, shown in entry 6, seem to be even better than for the TSTN.

We have also tested the last two approaches using architecture 2 (entries 10 and 11). The same conclusions seem to hold.

4.3 Discussion

In this section we provide several perspectives to interpret the results of our experiments with TSTNs.

First, the parameters learned by the localization CNN can be interpreted as encoding the knowledge of how to transform input samples (the transformations being conditioned on each individual sample) so as to simplify the task of recognition for the following layers of the entire system. By tying the parameters of the classification and localization CNNs, the TSTN is constrained to become good at both tasks (transforming the input image and classifying the transformed image).

As already mentioned in the previous section, the gradients of the localization CNN can become smaller than the gradients of the classification CNN, as a result of the vanishing gradient problem. This means that the localization CNN can have less influence in 'guiding' the search for the optimum parameters (than the classification CNN), when the two CNNs share parameters. This is somewhat similar to the problem of vanishing gradients in RNNs, given that almost the same computation is repeated twice.

Using the localization network can be seen as adding an auxiliary objective, different from the one that we are finally interested in (cross-entropy as a proxy for classification error), but which might help improve it [134]. Using the TSTN objective during certain training epochs can be seen as deviating from the optimal trajectory which the training procedure would follow if we were to only train the classification CNN. This makes the optimization of the classification network itself harder and we have observed empirically that, especially early during training, the validation performance of the classification CNN alone is worse after the epochs during which the TSTN objective is used. However, our experiments show that using the auxiliary TSTN objective leads to better final test performance, even when using the classification network alone at test time (and the performance is also better than that of untied STNs).

Another manner to interpret the results is in relationship to curriculum learning [43]. First training a localization network together with the classification network could help the classification network by providing it with easier images to classify (e.g. if the localization network is trained so that it can deslant the digits in the

images). After this initial period, the classification network is then either trained alone during all the epochs, or the percentage of training epochs with the localization network could be diminished (different schedules can be imagined, but we have only explored very simple ones in this work). Further experiments should be performed to verify this hypothesis.

An advantage of tying trainable parameters in the TSTNs is that we no longer have to perform a separate hyper-parameter search for the localization CNN. A disadvantage of STNs (both untied and tied), however, is that the computational time needed to use them is approximately twice the computation cost of the classification CNN alone. From this perspective, training procedures where we alternate between training the entire TSTN and training the classification CNN, while only using the classification CNN at test time, can be useful to reduce the computational burden for systems deployed in production.

4.4 Conclusion

In this chapter, we have proposed a new manner of using Spatial Transformer Network architectures, by tying the trainable parameters between the localization and classification networks and applied it to the recognition of noisy handwritten digits.

In the next chapter, we address a more challenging problem, the recognition of handwritten words, using a recently proposed recurrent neural network architecture.

Chapter 5

Associative LSTMs for handwriting recognition

Contents

In this chapter we present results for recognizing Arabic handwriting sequences, comparing recently-proposed Associative Long Short-term Memory (LSTM) [74] recurrent neural networks (RNNs) against a benchmark Long Short-term Memory (LSTM) RNN.

Our motivation is to study the potential benefits of adding extra memory capacity in RNNs without significantly increasing the number of trainable parameters for

recognizing Arabic handwriting. [74] also motivates introducing Associative LSTMs as enhancing (augmenting) LSTMs with extra memory. The extra memory is added by creating redundant copies of the stored information in the Associative LSTM cells and by using key-value data storage and retrieval mechanisms. The redundant copies allow for reduced noise when the retrieval process is performed.

As presented in [74], LSTMs have several limitations:

1. The number of memory cells (units) is linked to the size of the recurrent weight matrices. More precisely, N_h units require $O(N_h^2)$ weights, leading to high memory and computational costs.

2. LSTMs have difficulties reading common data structures, because of lacking explicit mechanisms for memory indexing during reads and writes.

While previous proposals, such as Neural Turing Machines [98], use external memory and soft or hard attention mechanisms to address the memory locations for reading or writing, Associative LSTMs don't use locations, but distributed storage, like LSTMs, and the associative mechanism is implemented using key-value pairs. [74] also motivates Associate LSTMs by comparing them to Redundant Arrays of Inexpensive Disks (RAID). Similarly to how RAID is used for reliable storage, Associative LSTMs make use of redundant memory to reduce the noise, without increasing the number of trainable parameters.

In our study on applying Associative LSTMs to handwriting recognition, we will follow a similar methodology to that of [202], which showed that a (1D) Bidimensional LSTM (BLSTM) can be more accurate than Multidimensional LSTMs, if the input images are normalized (while learning directly from pixels). This approach also improves upon methods which use handcrafted features. All the above-mentioned systems were benchmarked on the dataset of handwritten Arabic words IFN / ENIT [22].

We will use the same dataset and normalization method to compare between Associative and vanilla LSTMs and will compare multiple configurations of both types of architectures: single layer, multilayer, unidirectional, bidirectional (and combinations, e.g. multilayer bidirectional).

In the next section, we will present the theory and motivation behind Associative LSTMs (including the complex key-value mechanism and redundant associative memory) and shortly present LSTMs, for comparison. We will then present our results on the IFN-ENIT dataset, discuss how our work compares to the literature and conclude.

5.1 Methods

In this section we will describe the ideas behind Associative LSTMs, following closely the presentation in [74].

5.1.1 Holographic Reduced Representations

We will first describe the idea of Holographic Reduced Representations, which provide a key-value associative mechanism allowing to represent an indefinite number of key-value pairs using a vector of fixed size (an associative array). We will use three types of complex vectors which all have the same size ($N_h/2$ complex numbers, which can also be represented as a real vector of size N_h) and represent the keys, their associated values and the associative array, respectively. For a complex value vector denoted by x, which can be written as $x = (a_x[1]e^{i\phi_x[1]}, ..., a_x[N_h/2]e^{i\phi_x[N_h/2]})$ in polar form, its associated complex key vector r can be written as $r = (a_r[1]e^{i\phi_r[1]}, ..., a_r[N_h/2]e^{i\phi_r[N_h/2]})$. $a_r[k]$ denotes the absolute value of $r[k]$ and $\phi_r[k]$ denotes the phase of $r[k]$ (similarly for x). r and x have the same size, so they can be 'bound' together using element-wise complex multiplication (which multiplies corresponding moduli and adds corresponding phases):

$$c = r \otimes x = (a_r[1]a_x[1]e^{i(\phi_r[1]+\phi_x[1])}, ..., a_r[N_h/2]a_x[N_h/2]e^{i(\phi_r[N_h/2]+\phi_x[N_h/2])}) \quad (5.1)$$

c, *the associative array*, is also denoted by the term *memory trace*. Given multiple keys r_1, r_2 and r_3 (and multiple associated values x_1, x_2 and x_3), the memory trace c is obtained by adding the 'bindings' resulting from the element-wise complex multiplication:

$$c = r_1 \otimes x_1 + r_2 \otimes x_2 + r_3 \otimes x_3 \quad (5.2)$$

Given a key r, the *key inverse* r^{-1} is defined as:

$$r^{-1} = (a_r[1]^{-1}e^{-i\phi_r[1]}, ..., a_r[N_h/2]^{-1}e^{-i\phi_r[N_h/2]}) \quad (5.3)$$

The value x_k associated with the key r_k can be retrieved (approximately) by multiplying the memory trace c element-wise with the inverse key r_k^{-1}. For example, for the previous case of keys r_1, r_2 and r_3 and values x_1, x_2 and x_3, to approximately recover x_1:

$$r_1^{-1} \otimes c = r_1^{-1} \otimes (r_1 \otimes x_1 + r_2 \otimes x_2 + r_3 \otimes x_3) = x_1 + r_1^{-1} \otimes (r_2 \otimes x_2 + r_3 \otimes x_3) = x_1 + noise \tag{5.4}$$

The noise term will be 0 in expectation, if the phases of the elements of the key vector are randomly distributed.

Instead of using the inverse r_k^{-1} for retrieval, [154] and [74] suggest using the conjugate $\overline{r_k} = (a_k[1]e^{-i\phi_k[1]}, ..., a_k[N_h/2]e^{-i\phi_k[N_h/2]})$. The intuition behind this choice is that the moduli of the inverse $(a_k[1]^{-1}, ..., a_k[N_h/2]^{-1})$ could have a negative effect, by magnifying the noise term (a problem which doesn't occur for the moduli of the conjugate).

5.1.2 Redundant Associative Memory

One problem of the previously described mechanism is that, as the number of items to be stored is increased, the noise in Eq. 5.4 also grows. This noise can be reduced by using multiple memory traces, each of which will store the input vectors after a certain transformation has been applied to them. During retrieval, an average of the copies stored in each separate memory trace will be returned.

We will denote the number of copies by N_{copies} and the number of items by N_{items}. To each value $x_k \in \mathbb{C}^{N_h/2}$ is associated the key $r_k \in \mathbb{C}^{N_h/2}$. Each value x_k and key r_k can also be represented as vectors of size N_h.

The equation corresponding to the s-th memory trace c_s can now be written as:

$$c_s = \sum_{k=1}^{N_{items}} (P_s r_k) \otimes x_k \tag{5.5}$$

$P_s \in \mathbb{R}^{N_h/2 \times N_h/2}$ is the constant random permutation matrix specific to the memory trace c_s. Applying the permutation matrix P_s to the key r_k decorrelates the retrieval noise from the memory trace copy c_s.

A noisy version $\widetilde{x}_k$ of the value x_k can be retrieved using the following equation:

$$\widetilde{x}_k = \frac{1}{N_{copies}} \sum_{s=1}^{N_{copies}} \overline{(P_s r_k)} \otimes c_s \tag{5.6}$$

$\overline{(P_s r_k)}$ is the complex conjugate of $P_s r_k$ and the average over all memory trace copies $c_s, s \in \{1, ..., N_{copies}\}$ is taken.

We will analyze how the noise scales with the number of copies and the number

of items when $\widetilde{x}_k$ is retrieved. We will assume that each complex element of the vector r_k has modulus 1, so that the complex conjugate acts as an inverse and we can retrieve x_k (without any scaling factor $\neq 1$) and some additional noise:

$$\widetilde{x}_k = x_k + \frac{1}{N_{copies}} \sum_{s=1}^{N_{copies}} \sum_{k' \neq k}^{N_{items}} \overline{(P_s r_k)} \otimes (P_s r_{k'}) \otimes x_{k'}$$

$$= x_k + \sum_{k' \neq k}^{N_{items}} x_{k'} \otimes \frac{1}{N_{copies}} \sum_{s=1}^{N_{copies}} P_s(r_k \otimes r_{k'}) = x_k + noise$$

If the terms $P_s(r_k \otimes r_{k'})$ are independent, the sum over the copies $\sum_{s=1}^{N_{copies}} P_s(r_k \otimes r_{k'})$ will be 0 in expectation. If we also assume that the noise due to one item $x_{k'} \otimes \frac{1}{N_{copies}} \sum_{s=1}^{N_{copies}} P_s(r_k \otimes r_{k'})$ is independent of the noise due to the other items, the variance of the total noise for a retrieved item will be of the order $O(\frac{N_{items}}{N_{copies}})$. By keeping a number of copies similar to the number of items, this variance can be kept constant as the number of items is increased.

Redundant associative memory has some nice properties, as described in [74]:

1. The number of copies can be modified at any time, since it is independent from the number of units and other Associative LSTM hyperparameters. By increasing the number of copies, the retrieval noise can be reduced and the memory capacity can be increased.

2. Since no location mechanism is used, like in Neural Turing Machines, but the storage is distributed, it is not necessary to add any mechanism to search for free locations to write to or mechanisms to decide which previous locations to overwrite (if all the locations had previously been written to).

3. More items than the number of copies can be stored, at the cost of increased retrieval noise.

In [74], a specific task is described to demonstrate how redundant memory copies can reduce the noise as the number of items is kept fixed, using ImageNet examples [23]. Each image of width and height 110 (3 color channels) is vectorised, starting from the data format $3 \times 110 \times 110$. The first half of the resulting vector is interpreted as the real part and the second half as the imaginary part of a complex vector. Random complex keys with moduli equal to 1 are used for encoding the vectors into the random associative memory.

We have reproduced the results which show that, as the number of copies is increased (with the number of items fixed), the noise diminishes.

5.1.3 LSTM

For convenience, we more briefly describe the LSTM (already presented in Section 1.3.3.2) here, using a notation more similar to that in [74], to make the comparison between the LSTM and the Associative LSTM simpler.

The LSTM equations are:

$$\hat{g}_f, \hat{g}_i, \hat{g}_o, \hat{u} = W_{xh}x_t + W_{hh}h_{t-1} + b_h \tag{5.7}$$

$$g_f = \sigma(\hat{g}_f) \tag{5.8}$$

$$g_i = \sigma(\hat{g}_i) \tag{5.9}$$

$$g_o = \sigma(\hat{g}_o) \tag{5.10}$$

$$u = \tanh(\hat{u}) \tag{5.11}$$

$$c_t = g_f \odot c_{t-1} + g_i \odot u \tag{5.12}$$

$$h_t = g_o \odot \tanh(c_t) \tag{5.13}$$

σ is the sigmoid function. g_f, g_i and g_o are the forget, input and output gates. c is the cell state and u is the proposed update. W_{xh} and W_{hh} are weight matrices, b_h is a bias vector and $\odot$ denotes element-wise multiplication. It can be noted that the multiple elements at the left hand side of equation 5.7 means that weight matrices W_{xh} and W_{hh} include different sub matrices from which each element can be obtained.

5.1.4 Associative LSTM

Associative LSTMs are obtained by combining Holographic Reduced Representations with LSTMs. To implement an Associative LSTM, we will need to perform complex

vector element-wise multiplication. a complex vector $z = h_{real} + ih_{imaginary}$ can be represented as a real vector using the following format:

$$h = \begin{bmatrix} h_{real} \\ h_{imaginary} \end{bmatrix} \tag{5.14}$$

where $h \in \mathbb{R}^{N_h}$, $h_{real}, h_{imaginary} \in \mathbb{R}^{N_h/2}$. Using this transformation, all the vectors and matrices in the Associative LSTM can be represented using real numbers.

The Associative LSTM also uses gates, like the LSTM, but additionally introduces associative keys $\hat{r}_i, \hat{r}_o$ (and the additional trainable parameters required, corresponding to extending the matrices W_{xh} and W_{hh} and biases b_h to the necessary dimensions). The same gates are applied identically to the real and imaginary parts of the complex vectors, similarly to how σ is applied to the real vectors in the LSTM in Eqs. 5.8, 5.9 and 5.10.

$$\hat{g}_f, \hat{g}_i, \hat{g}_o, \hat{r}_i, \hat{r}_o = W_{xh}x_t + W_{hh}h_{t-1} + b_h \tag{5.15}$$

$$\hat{u} = W_{xu}x_t + W_{hu}h_{t-1} + b_u \tag{5.16}$$

$$g_f = \begin{bmatrix} \sigma(\hat{g}_f) \\ \sigma(\hat{g}_f) \end{bmatrix} \tag{5.17}$$

$$g_i = \begin{bmatrix} \sigma(\hat{g}_i) \\ \sigma(\hat{g}_i) \end{bmatrix} \tag{5.18}$$

$$g_o = \begin{bmatrix} \sigma(\hat{g}_o) \\ \sigma(\hat{g}_o) \end{bmatrix} \tag{5.19}$$

The Associative LSTM formulation introduces a new activation function which only modifies the modulus of the entries of a complex vector, by restricting it to the interval [0, 1]. We will first define the vector $d \in \mathbb{R}^{N_h/2}$, which corresponds to element-wise normalization by the modulus of each complex number when the modulus is greater than one:

$$d = \max(1, \sqrt{h_{real} \odot h_{real} + h_{imaginary} \odot h_{imaginary}}) \tag{5.20}$$

The function *bound* is then defined as:

$$bound(h) = \begin{bmatrix} h_{real} \oslash d \\ h_{imaginary} \oslash d \end{bmatrix} \tag{5.21}$$

where $\oslash$ denotes element-wise division. The introduction of this nonlinearity is motivated in [74] as having worked better than applying a more standard nonlinearity such as tanh separately to the moduli of the complex numbers.

We will next compute the input key $r_i \in \mathbb{R}^{N_h}$, which will act as a storage key in the associative array (denoted further by c), the output key $r_o \in \mathbb{R}^{N_h}$, which corresponds to a lookup key and the update u, which will produce the value to be stored by multiplying it with g_i, similarly to Eq. 5.12 in the LSTM.

The *bound* nonlinearity is used for all these computations, analogously to how the *tanh* nonlinearity is applied in the original LSTM in Eq. 5.11:

$$u = bound(\hat{u}) \tag{5.22}$$

$$r_i = bound(\hat{r}_i) \tag{5.23}$$

$$r_o = bound(\hat{r}_o) \tag{5.24}$$

We will next describe how redundant storage is introduced and the procedure to use it during memory retrieval. For each copy indexed by $s \in \{1, ..., N_{copies}\}$ and every timestep t, there will be a different cell state (representing a memory trace) $c_{s,t}$.

The input keys r_i are first permuted using the permutation matrices $P_s \in \mathbb{R}^{N_h/2 \times N_h/2}$:

$$r_{i,s} = \begin{bmatrix} P_s & 0 \\ 0 & P_s \end{bmatrix} r_i \tag{5.25}$$

The cell state indexed by s at time t will be computed using the following expression:

$$c_{s,t} = g_f \odot c_{s,t-1} + r_{i,s} \circledast (g_i \odot u) \tag{5.26}$$

where $\circledast$ denotes element-wise complex multiplication. Element-wise complex multiplication between two complex vectors r and u can be written as the following expression with regard to their real and imaginary parts r_{real}, $r_{imaginary}$, u_{real} and

$u_{imaginary}$:

$$r \circledast u = \begin{bmatrix} r_{real} \odot u_{real} - r_{imaginary} \odot u_{imaginary} \\ r_{real} \odot u_{imaginary} + r_{imaginary} \odot u_{real} \end{bmatrix} \tag{5.27}$$

Notice the similarity of Eq. 5.26 to the LSTM Eq. 5.12. The left side of the equation, $g_f \odot c_{s,t-1}$, is the same as in the LSTM, except for the extension to multiple cells c_s. The right side of the equation, $r_{i,s} \circledast (g_i \odot u)$, performs memory 'bounding', like in Eq. 5.2 (without redundant storage) and in Eq. 5.5 (when redundant storage is added). The memory cell c in the LSTM Eq. 5.26 can then be interpreted as playing the same role as the associative memory trace in Eqs. 5.2 (without redundant storage) and 5.5 (when redundant storage is added).

The same matrix P_s is used to obtain the permuted output keys $r_{o,s}$ from the output key r_o:

$$r_{o,s} = \begin{bmatrix} P_s & 0 \\ 0 & P_s \end{bmatrix} r_o \tag{5.28}$$

The cells are read out by averaging the copies indexed by s:

$$h_t = g_o \odot bound \left(\frac{1}{N_{copies}} \sum_{s=1}^{N_{copies}} r_{o,s} \circledast c_{s,t} \right) \tag{5.29}$$

The expression $\frac{1}{N_{copies}} \sum_{s=1}^{N_{copies}} r_{o,s} \circledast c_{s,t}$ is the same as the retrieval from Redundant Associative Memory in Eq. 5.6. The rest of Eq. 5.29 is the same as Eq. 5.13 for the LSTM, with the *tanh* nonlinearity replaced by *bound*.

Because the matrices P_s are fixed (not learned), using them does not lead to an increase in the number of trainable parameters. The averaging over the different copies also means that the resulting vector's size is independent of the number of copies. Each permutation can be performed in asymptotic computational time $O(N_h)$ and all the N_{copies} different operations can be parallelized over tensors of size $N_{copies} \times N_h$.

In [74], for some tasks W_{hu} in Eq. 5.16 is set to 0 and this is observed to lead to faster learning.

5.2 Results

We will next describe the results we have obtained on the IFN-ENIT dataset. We will follow a methodology similar to that of [202], training for a fixed number of epochs and reporting the performance of the trained system on the validation set.

5.2.1 Dataset

The IFN-ENIT dataset contains contains 32492 images of Tunisian cities and villages names, handwritten by more than 400 writers under a large variety of writing styles. It is divided into five sets, A-E. In the most common setting, which we will also follow, sets A-D (containing 26459 images) are used for training and set E (containing 6033 images) is used for testing.

The images vary both in height and in width and range from 85×84 to 162×1069 in the training set and from 40×170 to 139×977 in the test set. All the images are grayscale.

5.2.2 Image normalization

In this subsection we will describe the image normalization procedure we have used for the IFN-ENIT handwriting images.

We first perform Otsu thresholding [151] using the OpenCV implementation [27]. We then use the image normalization procedure provided by ocropy [26] which performs dewarping and image size normalization to the default ocropy image height of 48 pixels (with the width rescaled proportionally, so as to preserve the aspect ratio). We then perform contrast inversion on the resulting image and, finally, normalize the pixel values to [0, 1].

We display IFN-ENIT images before and after Otsu thresholding and ocropy normalization in Fig 5.1.

5.2.3 System details

We initialize the recurrent weights for all the architectures using the orthogonal initializer [166]. The forget biases are initialized to 1 and all the other recurrent biases are initialized to 0. For all the architectures, we have initialized the weights of the output layer using Xavier initialization and the biases to 0.

The loss optimized is the CTC loss, which is presented is Section 1.5.1.2. All the results presented are obtained using greedy CTC decoding, presented in Section

1.5.1.2.

For optimization, we have used the ADAM [125] gradient-based optimizer with default settings (including the default 0.001 learning rate). We also apply gradient clipping by global norm, with global norm 10. We have used batch size 64 for all the experiments.

We also present results for bidirectional architectures. Bidirectional RNNs are discussed in Section 1.3.3.3. The extension to Bidirectional LSTMs and Bidirectional Associative LSTMs is straightforward.

5.2.4 Results

All the results presented are obtained on the validation set E, with training performed on the sets A-D.

Our Associative LSTMs use a single copy, since early experiments with multiple copies didn't show any gains.

Because the LSTM is about $4-5\times$ faster, we allow it to run for more training epochs (150) than the Associative LSTM (100). In all of our experiments with Associative LSTMs, W_{hu} was set to 0.

System	CER	Number of trainable parameters	Number of training epochs
LSTM-128	0.5451	106233	150
LSTM-256	0.5200	343417	150
ALSTM-64	0.5314	40121	100
BLSTM-64	0.3545	73465	150
BALSTM-64	0.3598	80121	100
BLSTM-128	0.3197	212345	150
BALSTM-128	0.2919	233849	100

Table 5.1 – Character error rates (CERs) for various LSTM and Associative LSTM systems. We denote LSTM RNNs with n units by LSTM-[n] and Associative LSTM (ALSTM) with n units by ALSTM-[n]. We add the prefix B for Bidirectional RNNs.

We display the results for different LSTM and Associative LSTM settings in Table 5.1. By LSTM-[n] we denote LSTM RNNs with n units and analogously for Associative LSTM (ALSTM). For Bidirectional RNNs, we add the prefix B.

We can see that the ALSTM-64 performs better than the LSTM-128 with about $2.5\times$ less trainable parameters, and somewhat worse than LSTM-256, which contains

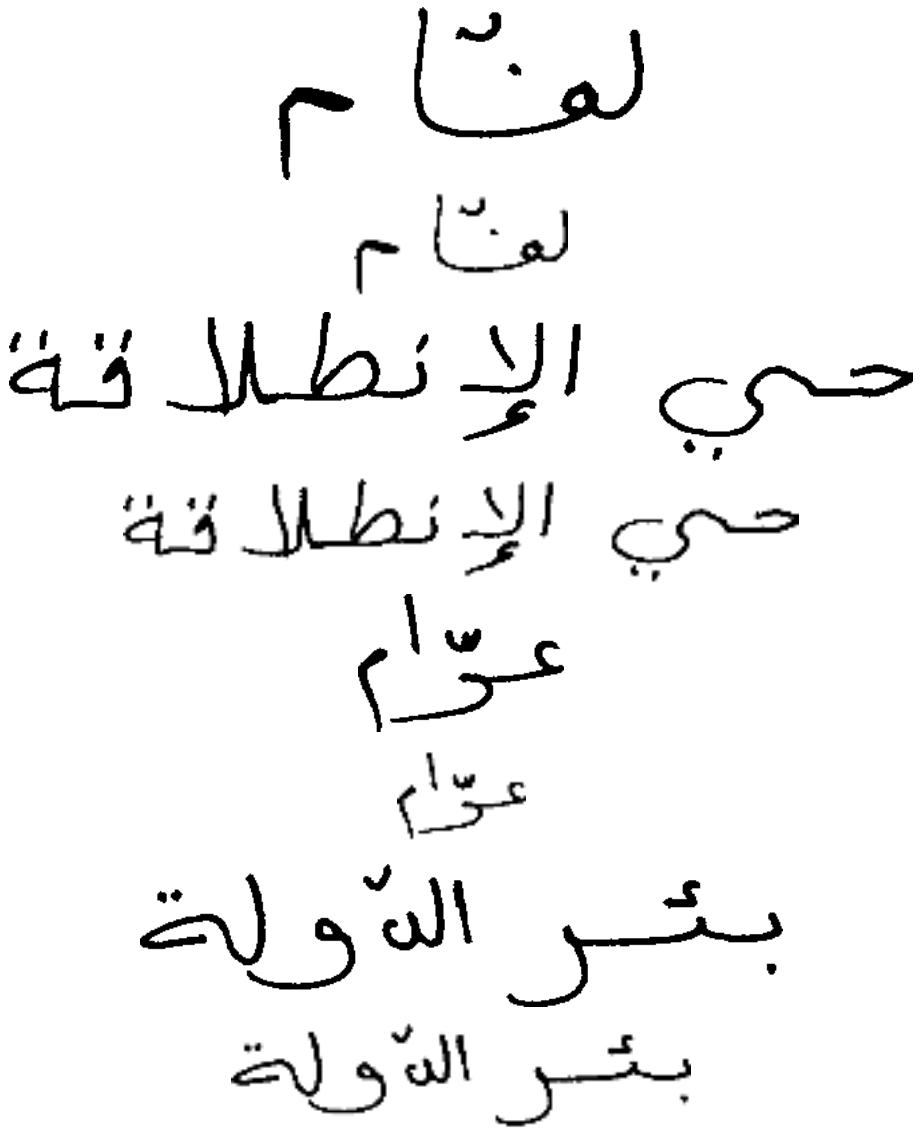


Figure 5.1 – Example images from IFN-ENIT before and after Otsu thresholding and ocropy normalization. The odd images, from top to bottom, are the inputs (unnormalized), while the even ones are thresholded and normalized. For better visualization, the input images are scaled 0.5×.

about 8.5× more trainable parameters.

The BALSTM-128 performs better than the BLSTM-128, while the BALSTM-64 doesn't seem to do better than BLSTM-64. In both cases, the BALSTM and BLSTM systems have comparable numbers of trainable parameters.

In our implementation, the Associative LSTM seems to be about 4-5 times slower than an LSTM, for equal number of units. This slowdown is similar to that reported in [74]. This diminished computational efficiency partly negates the accuracy gains and motivates us to try other approaches to increase the memory of an RNN, without

increasing the computational cost, as will be discussed in the next chapter.

5.3 Discussion

Other RNN architectures have been proposed motivated by the benefits of increased memory without increasing the number of trainable parameters and while potentially reducing the computational cost.

Unitary evolution RNNs [38] replace the full-rank hidden-to-hidden transition matrices W_{hh} in vanilla RNNs with a factorization of complex matrices:

$$W_{hh} = D_3 R_2 F^{-1} D_2 \Pi R_1 F D_1 \tag{5.30}$$

In this equation F and F^{-1} are the Fourier Transform and Inverse Fourier Transform and are fixed (not learned). Π denotes a fixed random index permutation matrix. Each of the D_1, D_2, D_3 matrices is diagonal and learned, parameterized as $D_{j,j} = e^{iw_j}$, where w_j are real numbers. R_1 and R_2 are reflection matrices parameterized as $R = I - 2\frac{vv^*}{||v||^2}$, with I the identity matrix and $v \in \mathbb{C}^n$ a complex vector. F and F^{-1} require no storage and require $O(n \log(n))$ computation. The D, R and Π matrices require $O(n)$ storage and $O(n)$ computation. Since a typical vanilla RNN W_{hh} matrix requires $O(n^2)$ storage and $O(n^2)$ computation, this parameterization can be much cheaper, as n, the number of units, grows. By increasing n, unitary evolution RNNs could allow for potentially vast memory representations, as also discussed in [38].

This is motivated by the fact that the memory capacity of RNN architectures is usually correlated with the number of units, as shown theoretically in the reservoir computing literature [118] and empirically in [71].

The parameterization in [121] goes even further in reducing the number of trainable parameters in the hidden-to-hidden transition matrix, by proposing a Kronecker factorization which only requires $O(\log n)$ parameters and storage and $O(n \log n)$ computational cost. Although the main motivation presented in [121] is reduced computational cost, the reduced storage and computational cost allow for increased number of units and increased memory. In the next chapter we will describe a similar approach, inspired by the work in [121], but which allows the computational cost to be reduced by using quantum computation.

5.4 Conclusion

In this chapter we have studied the impact of using a mechanism through which more memory can be added to a RNN without significantly increasing the number of trainable parameters on a handwriting recognition task. In the next chapter we will show how quantum computation seems to provide the opportunity for up to exponentially more memory capacity in an RNN.

Chapter 6

Hybrid classical-quantum deep learning

Contents

In this chapter, we propose a hybrid classical-quantum neural network parameterization which is inspired by previously proposed work on hypernetworks [100]. The idea of hypernetworks is that a neural network, which we will call *the hypernetwork*, can be used to generate the parameters of another network, denoted as *the main network*, which is used to make the required predictions. We instantiate this proposal using recurrent neural network (RNNs) on a binary classification task for sequences. More specifically, we propose that a classical RNN (implemented on a classical computer) generates the parameters of a quantum RNN (implemented on a quantum computer), which will provide the label prediction. We simulate the training of both components of our system (the classical and the quantum RNN) on a classical computer (GPU), using stochastic gradient descent (SGD) and backpropagation. RNNs are Turing-complete [115] [170], so, in some sense, fully general deep learning computing models. Addressing RNNs differentiates our work from most previous quantum neural network proposals and using quantum networks differentiates our work from classical hypernetworks. Section 6.8 provides a more detailed discussion of this point. In principle, our proposal is general and can address more types of architectures than RNNs.

The intuition behind our proposal is that quantum computers can allow us to perform some operations on very high-dimensional spaces with exponentially less computation compared to classical computers. In the case of RNNs, this high dimensionality could allow for more memory, so that the quantum RNN remembers inputs from the distant past with significant computational gains compared to classical proposals. The hyper-RNN in our proposal can be interpreted as *learning to encode* the inputs into the high-dimensional space of the quantum RNN.

To simplify the problem of generating outputs from the quantum computer, in this work we will only address many-to-one problems [135], in which the input is presented as a sequence and a single output is required. We will show results suggesting that our proposed hybrid classical-quantum RNN can perform similarly to a classical RNN in a long-term sequential dependency task, with potentially exponentially less computation required. For this, we will choose a task similar to the permuted pixel MNIST task, a standard benchmark for the long-term memory capabilities of RNN architectures [121] [38], but using the Fashion-MNIST dataset [198] [199].

We simulate our proposal on GPU, but show the approximate computational costs required if our proposal were run on a quantum computer.

To the best of our awareness, our work is the first to show promising results suggesting a potential quantum advantage for RNNs on a large, industrial-scale dataset.

We now summarize the next sections of this chapter. We will begin by briefly discussing the motivation behind our use of quantum computing in Section 6.1. We then (in Section 6.2) shortly introduce and illustrate the best-known model of quantum computation and how some of its features can be related neural network concepts. In Section 6.3, we discuss a natural implementation of quantum neural networks, but which is difficult to simulate classically using backpropagation. This difficulty motivated our proposal for hybrid classical-quantum neural networks in the framework of hypernetworks, a framework which we introduce in Section 6.4. In Section 6.5 we describe our proposed parameterization, including details about the main (quantum) RNN component, with its output layer and the loss function we have used, and the classical RNN which generates the quantum RNN parameters. In Section 6.6 we start by motivating our choice of task and dataset, then provide implementation details for our system and the classical benchmark and the methods we have used to approximate the computational costs of both systems. We then describe how we estimate the accuracy of our proposal under ϵ-sampling approximations. In Section 6.7 we present the results we have obtained. In Section 6.8, we discuss how our proposal is related to the literature in quantum and classical machine learning, as well as limitations and potential improvements of our method, followed by conclusions in Section 6.9.

6.1 Motivation for using quantum computing

In this section, we briefly discuss the motivation behind our use of quantum computing in the neural network architecture we have proposed.

Quantum computing is the most general and most powerful form of computing known as of today, while classical computing can be seen as a specialization, implementing a restricted subset of the operations that quantum computers can implement [30]. An illustration of this generality and power of quantum computation comes from computational complexity theory. We will first introduce the classes of decisions problems which can be solved or verified efficiently using classical and quantum computation, respectively.

In the theory of computational complexity, '**bounded-error quantum polynomial time (BQP)** is the class of decision problems solvable by a quantum computer in polynomial time, with an error probability of at most 1/3 for all instances' [4]. Meanwhile, the class **P** 'contains all decision problems that can be solved by a deterministic Turing machine using a polynomial amount of computation time' [28]. At the same time, '**NP** is the set of decision problems for which the problem instances, where the answer is "yes", have proofs verifiable in polynomial time' [25].

So, to summarize, **P** corresponds to decisions problems **efficiently solvable classically**, **BQP** to decisions problems **efficiently solvable using quantum computation** and **NP** to decisions problems **efficiently verifiable classically**.

BQP is known to contain P, while it is suspected that BQP and NP intersect, but none is completely contained in the other [4]. The intuitive interpretation of BQP containing P is that there exist problems which quantum computers can solve efficiently, but classical computers can't (these problems would belong to BQP, but not to P) and that, at the same time, any problem which can be solved efficiently classically can also be solved efficiently using quantum computing. More counter-intuitively, BQP not contained in NP would mean that there exist problems which quantum computers can solve efficiently and whose solution can't even be proved efficiently classically.

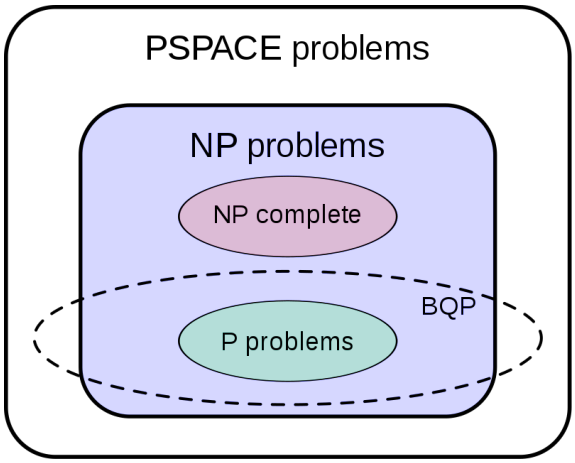


Figure 6.1 – Suspected relationship between the BQP, P and NP complexity classes. PH is a complexity class which can be interpreted as a generalization of NP. Figure from [4].

BQP has long been suspected not to be included in NP [4] and recently a problem has been found which is in BQP but not in NP [159]. The findings of [159] also imply that, even if it were true that P = NP, quantum computers would still be able to efficiently solve some problems that classical computers can't (and whose

solution classical computers can't even check). We illustrate the suspected relationship between BQP, P and NP in Fig. 6.1.

6.2 Introduction to the quantum circuit model of quantum computing with discrete variables

In this section, we shortly introduce the best-known model of quantum computation and discuss and illustrate the most important concepts and operations available in this model (which we have also used in our proposal). We will also briefly discuss the similarity between some of these concepts and operations to some concepts and operations from the field of deep learning.

Several models exist for implementing quantum computation, including adiabatic quantum computation [1], continuous-variable quantum circuits [7] and discrete-variable quantum circuits [30]. We will use the discrete-variable quantum circuit model. This is an universal model of quantum computation (a problem which can be solved in polynomial time using any other model of quantum computation can also be solved in polynomial time using the discrete-variable quantum circuit model). Arguably, this model is also the most natural for computer scientists, due to the similarities between the classical bit and its quantum version, the qubit, as will be seen in more details in the next section. In the following sections, we will shortly introduce the operations available when using discrete-variable quantum circuits which are relevant for our discussion about quantum neural networks: unitary evolution and measurement (including full and partial quantum measurements).

6.2.1 The qubit

The quantum equivalent of the classical bit is the qubit. A qubit's quantum state can be represented as a complex vector $|\Psi\rangle = \begin{bmatrix} \alpha_0 \\ \alpha_1 \end{bmatrix}$ of norm 1: $|\alpha_0|^2 + |\alpha_1|^2 = 1$. Another representation for the qubit's quantum state is as a **superposition** of the **basis states** $|0\rangle$ and $|1\rangle$: $|\Psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle$, where α_0 and α_1 can be interpreted as the amounts of the qubit $|\Psi\rangle$ being in the configurations $|0\rangle$ and $|1\rangle$, respectively. $|0\rangle$ and $|1\rangle$ represent the computational basis $\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. α_0 and α_1 can be linked to a probabilistic interpretation, as will be discussed in Section 6.2.3.

6.2.1.1 Multiple qubits

When multiple (n) qubits are put together, the quantum state of the entire ensemble can be written as a vector of 2^n complex numbers of complex norm 1.

In some cases, the quantum state of the entire ensemble can be expressed as a tensor product. For example, suppose each one of the qubits i, with $i \in \{1, ..., n\}$, has been initialized to the state $\begin{bmatrix} \alpha_i \\ \beta_i \end{bmatrix} = \alpha_i |0\rangle + \beta_i |1\rangle$. The quantum state of the entire ensemble of n qubits, when put together, is then the vector of 2^n complex numbers resulting from the tensor product of the n qubit states (vectors): $\bigotimes_{i=1}^{n} \begin{bmatrix} \alpha_i \\ \beta_i \end{bmatrix} = \bigotimes_{i=1}^{n} (\alpha_i |0\rangle + \beta_i |1\rangle)$.

We will illustrate the tensor product for two qubits $\begin{bmatrix} \alpha_1 \\ \beta_1 \end{bmatrix} = \alpha_1 |0\rangle + \beta_1 |1\rangle$ and $\begin{bmatrix} \alpha_2 \\ \beta_2 \end{bmatrix} = \alpha_2 |0\rangle + \beta_2 |1\rangle$:

$$\begin{aligned} \begin{bmatrix} \alpha_1 \\ \beta_1 \end{bmatrix} \otimes \begin{bmatrix} \alpha_2 \\ \beta_2 \end{bmatrix} &= (\alpha_1 |0\rangle + \beta_1 |1\rangle) \otimes (\alpha_2 |0\rangle + \beta_2 |1\rangle) = \begin{bmatrix} \alpha_1 \begin{bmatrix} \alpha_2 \\ \beta_2 \end{bmatrix} \\ \beta_1 \begin{bmatrix} \alpha_2 \\ \beta_2 \end{bmatrix} \end{bmatrix} \\ &= \begin{bmatrix} \alpha_1\alpha_2 \\ \alpha_1\beta_2 \\ \beta_1\alpha_2 \\ \beta_1\beta_2 \end{bmatrix} = \alpha_1\alpha_2 |00\rangle + \alpha_1\beta_2 |01\rangle + \beta_1\alpha_2 |10\rangle + \beta_2\alpha_2 |11\rangle \end{aligned} \tag{6.1}$$

Here $|00\rangle, |01\rangle, |10\rangle, |11\rangle$ form the computational basis $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$.

Because the quantum state of a quantum system of n qubits is a complex vector of 2^n numbers, a quantum system containing a relatively small number of qubits can represent gigantic vectors. For example, the quantum state of a system of 300 qubits is a vector of 2^{300} complex numbers (more than the number of atoms in the universe). Moreover, each time an extra qubit is added, the size of the quantum state (and of the classical memory which would be required just to store it) doubles. The advantage that quantum computing provides (compared to classical computing) is the efficient

manipulation of these huge vectors (but only for certain operations, mostly for matrix-vector multiplications, where the matrices are structured, as discussed in Section 6.2.2). The entire information in the quantum state also can't be accessed directly, but only limited amounts of information can be obtained through measurements, as will be discussed in Section 6.2.3.

Not all quantum states can be expressed as tensor products. An example of a quantum state which can't be expressed as a tensor product is the Bell state. The Bell state is the simplest example of **quantum entanglement** of two qubits (quantum entanglement means that the quantum state of each qubit cannot be described independently of the state of the others): $|\Psi\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$. It is easy to prove that this state can't be factored as a tensor product of the states of the two qubits: $|\Psi\rangle \neq (\alpha_1|0\rangle + \beta_1|1\rangle) \otimes (\alpha_2|0\rangle + \beta_2|1\rangle)$.

6.2.2 Unitary evolution

Most of the operations which can be applied to an ensemble of n qubits (whose state can be written as a vector of size 2^n) can be described as multiplying its state by a unitary matrix of size $2^n \times 2^n$. However, unitary matrices of size $2^n \times 2^n$ are usually computationally intractable for large n (even for quantum computers), unless the matrix has some special structure. When these matrices can be implemented efficiently using quantum computers, they can be expressed efficiently as quantum circuits composed of **quantum gates**. Efficiently usually means, in this case, that only a polynomial number of quantum gates is required. Quantum gates are quantum circuits operating on a small number of qubits and they represent the building blocks of quantum circuits, similarly to how classical logic gates are used in conventional digital circuits [29]. Simple quantum gates can be composed to form more complex unitary operations.

We will first describe the unitary evolution of single qubits and then discuss some interesting cases for the unitary evolution of multiple qubits.

6.2.2.1 Single qubit

In the case of a single qubit, if we describe the qubit's state as $|\Psi\rangle$, it will evolve to $|\Psi'\rangle$, when a 1-qubit unitary U is applied, through matrix-vector multiplication (between U and $|\Psi\rangle$):

$$|\Psi'\rangle = U * |\Psi\rangle \tag{6.2}$$

In the most general case, for a single qubit, U is a 2×2 unitary matrix which can be parameterized as:

$$U = e^{\frac{i\alpha}{2}} \begin{bmatrix} e^{i\beta}\cos(\phi) & e^{i\gamma}\sin(\phi) \\ -e^{-i\gamma}\sin(\phi) & e^{-i\beta}\cos(\phi) \end{bmatrix} \tag{6.3}$$

We will use this parameterization in our quantum RNN proposal in section 6.5.1. General 1-qubit unitaries can be implemented using 4 consecutive elementary 1-qubit gates [52] [150].

6.2.2.2 Multiple qubits

In the case of a system containing multiple (n) qubits, as mentioned previously, the most interesting unitaries are those which can be expressed using only a polynomial number of quantum gates. The structure of these matrices is such that they can be expressed as tensor products of unitaries applied to (small) subsets of the n qubits. For example, in the case of n fully general 1-qubit unitaries $U_j, j \in \{1, .., n\}$, U_j applied to qubit j, this is equivalent to a $2^n \times 2^n$ unitary which is the tensor product of the n 1-qubit unitaries $U_j, j \in \{1, ..., n\}$ multiplying the quantum state of the n qubits (vector of size 2^n):

$$\bigotimes_{j=1}^{n} U_j = \bigotimes_{j=1}^{n} e^{\frac{i\alpha_j}{2}} \begin{bmatrix} e^{i\beta_j}\cos(\phi_j) & e^{i\gamma_j}\sin(\phi_j) \\ -e^{-i\gamma_j}\sin(\phi_j) & e^{-i\beta_j}\cos(\phi_j) \end{bmatrix} \tag{6.4}$$

Because general 1-qubit unitaries can be implemented using 4 consecutive elementary 1-qubit gates [52] [150], the computational complexity of implementing the above $2^n \times 2^n$ unitary matrix using quantum computation is only $O(n)$.

We illustrate here the tensor product of two 2×2 matrices, resulting in a $2^2 \times 2^2$ matrix. This simple example shows the result of applying 2 single-qubit gates acting independently on each of the qubits of a 2-qubit system (the resulting $2^2 \times 2^2$ matrix acts on the quantum state - a vector of size 2^2):

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \otimes \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} & a_{12} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \\ a_{21} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} & a_{22} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} \\ a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22} \\ a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} \\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} \end{bmatrix} \tag{6.5}$$

This example illustrates how rapidly the size of the unitary matrices increases when multiple quantum gates operating on different qubits are used. Analogously to the case of adding multiple qubits, when multiple gates act on different qubits, the size of the resulting unitary matrix can grow very fast (for 300 qubits and 300 single qubit unitaries, each unitary acting on a different qubit, we would get a matrix of size $2^{300} \times 2^{300}$ multiplying a vector of size 2^{300}). This highlights the tremendous power of quantum computation: with only 300 quantum gates (operations), we can operate on 2^{300} vector entries.

In Fig. 6.2, we illustrate a 3-qubit system in quantum state $|\Psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle + \alpha_2 |2\rangle + \alpha_3 |3\rangle + \alpha_4 |4\rangle + \alpha_5 |5\rangle + \alpha_6 |6\rangle + \alpha_7 |7\rangle = \alpha_0 |000\rangle + \alpha_1 |001\rangle + \alpha_2 |010\rangle + \alpha_3 |011\rangle + \alpha_4 |100\rangle + \alpha_5 |101\rangle + \alpha_6 |110\rangle + \alpha_7 |111\rangle = \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ ... \\ \alpha_7 \end{bmatrix}$, which undergoes unitary evolution under 3 general 1-qubit unitaries, resulting in the output quantum state $|\Psi'\rangle = \alpha'_0 |0\rangle + \alpha'_1 |1\rangle + \alpha'_2 |2\rangle + \alpha'_3 |3\rangle + \alpha'_4 |4\rangle + \alpha'_5 |5\rangle + \alpha'_6 |6\rangle + \alpha'_7 |7\rangle = \alpha'_0 |000\rangle + \alpha'_1 |001\rangle + \alpha'_2 |010\rangle + \alpha'_3 |011\rangle + \alpha'_4 |100\rangle + \alpha'_5 |101\rangle + \alpha'_6 |110\rangle + \alpha'_7 |111\rangle = \begin{bmatrix} \alpha'_0 \\ \alpha'_1 \\ ... \\ \alpha'_7 \end{bmatrix}$. The collective action of the 3 general 1-qubit unitaries can be interpreted as multiplying the quantum state (vector) $|\Psi\rangle$ of size 2^3 by the $2^3 \times 2^3$ unitary matrix generated by taking the tensor product of the three 2×2 unitary matrices representing the general 1-qubit unitaries: $\bigotimes_{j=1}^{3} e^{\frac{i\alpha_j}{2}} \begin{bmatrix} e^{i\beta_j} \cos(\phi_j) & e^{i\gamma_j} \sin(\phi_j) \\ -e^{-i\gamma_j} \sin(\phi_j) & e^{-i\beta_j} \cos(\phi_j) \end{bmatrix}$, resulting in the quantum state (vector) $|\Psi'\rangle$ (also of size 2^3).

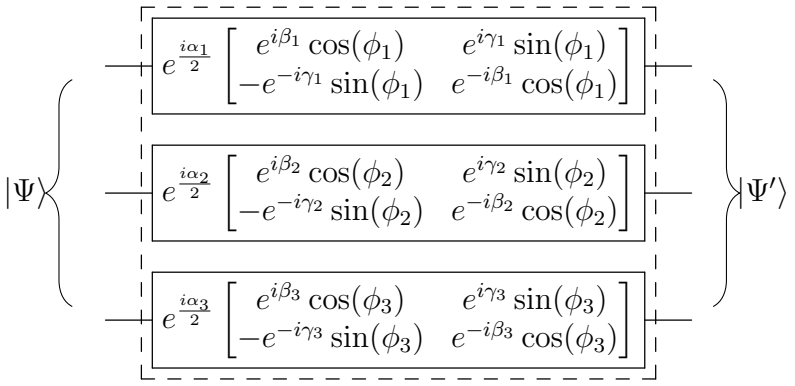


Figure 6.2 – Illustration of a 3-qubit system which undergoes unitary evolution under 3 general 1-qubit unitaries. The three qubits are illustrated as wires and the general 1-qubit unitaries as rectangle boxes. We illustrate the general 1-qubit unitaries as unitary gates (even though, technically, they would each be decomposed into multiple simpler 1-qubit gates) for simplicity. The 3 general 1-qubit unitaries are enclosed in a dashed box to indicate that their collective action can be interpreted as a single unitary transform, generated by taking the tensor product of the three general 1-qubit unitary matrices.

6.2.3 Measurement

To output information from a quantum system, we need to use quantum measurement. Measurement can be described as a stochastic, nonlinear, irreversible operation applied to one or multiple qubits. From the perspective of programming a quantum computer, measurement (whether it's full or partial) is just another primitive which can be used, similar to quantum gates.

In this section we describe two types of measurement of a quantum system: full measurement (in which all the qubits a qunatum system are measured) and partial measurement (in which only a subset of the qubits of a system are measured). We will discuss measurement in the computational basis $\{|0\rangle, |1\rangle\}$. Measurement in other basis is equivalent to first performing an unitary operation, and then measuring in the $\{|0\rangle, |1\rangle\}$ basis. In this sense, the case of the $\{|0\rangle, |1\rangle\}$ computational basis is fully general. Both types of quantum measurement can be interpreted as neural network (stochastic) nonlinear activation functions and such proposals have been previously put forward, as we will discuss in this section and the next one.

6.2.3.1 Full measurement

Full measurement means that all the qubits of a quantum system are measured. Full measurement can be used as a stochastic nonlinearity for a quantum neural network (and this was used in one of the earliest proposals for implementing a quantum neural

network [203]), as will be discussed in more detail in the next section.

In the case of a single qubit in superposition $|\Psi\rangle = \begin{bmatrix} \alpha_0 \\ \alpha_1 \end{bmatrix} = \alpha_0 |0\rangle + \alpha_1 |1\rangle$, full measurement (in the computational basis $|0\rangle$ and $|1\rangle$) collapses the qubit in a non-deterministic manner, to the state $|0\rangle$ with probability $|\alpha_0|^2$ or, respectively, to the state $|1\rangle$ with probability $|\alpha_1|^2$. This collapse means that no further information about the numbers α_0 and α_1 can be gathered by further measurement (in the computational basis $|0\rangle$ and $|1\rangle$): the qubit would remain in the state it collapsed to initially (with probability 1). Furthermore, the **no-cloning theorem** [24] prevents the copying of an arbitrary unknown quantum state (so as to measure multiple copies of the quantum state). To estimate the probabilities $|\alpha_0|^2$ and $|\alpha_1|^2$, we would have to repeatedly recreate the qubit state $|\Psi\rangle = \begin{bmatrix} \alpha_0 \\ \alpha_1 \end{bmatrix} = \alpha_0 |0\rangle + \alpha_1 |1\rangle$ and then measure the qubit. The probability $|\alpha_1|^2$ can then be estimated by taking the expectation (mean) of the successive measurements (and $|\alpha_0|^2$ can then be estimated by substracting the estimate of $|\alpha_1|^2$ from 1).

In the case of n qubits, the superposition $|\Psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle + ... + \alpha_{2^n-1} |2^n - 1\rangle = \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ ... \\ \alpha_{2^n-1} \end{bmatrix}$ would **collapse**, after measurement, to the state $|i\rangle$ with probability $|\alpha_i|^2$, $i \in \{0, ..., 2^n - 1\}$. Here, we denote by $|i\rangle$ the state of the n qubits which would correspond to the binary representation of the number i (e.g., for 2 qubits, $|\Psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle + \alpha_2 |2\rangle + \alpha_3 |3\rangle = \alpha_0 |00\rangle + \alpha_1 |01\rangle + \alpha_2 |10\rangle + \alpha_3 |11\rangle$).

6.2.3.2 Partial measurement

Partial measurement means that only a subset of the qubits of a quantum system is measured. This could be any subset, including all the qubits (this case reduces to full quantum measurement). We will focus in this section on a simple illustration of how partial quantum measurement is similar to neural network nonlinearities, which will be helpful when discussing a natural implementation of a quantum neural network (but whose training is difficult to simulate) in the next section. Partial quantum measurement has recently been used to implement a quantum neural network [161], as will be discussed in more details in the next section.

The simplest scenario to illustrate partial measurement is for a 2-qubit system in superposition $|\Psi\rangle = \alpha_{00} |0\rangle + \alpha_{01} |1\rangle + \alpha_{10} |2\rangle + \alpha_{11} |3\rangle = \alpha_{00} |00\rangle + \alpha_{01} |01\rangle +$

$\alpha_{10}|10\rangle + \alpha_{11}|11\rangle = \begin{bmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{bmatrix}$. Suppose that we measure the first qubit. The outcome 0 will be obtained with probability $|\alpha_{00}|^2 + |\alpha_{01}|^2$ (analogously, 1 would be obtained with probability $|\alpha_{10}|^2 + |\alpha_{11}|^2$). In the case of the outcome 0, the state of the 2-qubit system becomes $|\Psi'\rangle = \frac{\alpha_{00}|00\rangle + \alpha_{01}|01\rangle}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}} = \begin{bmatrix} \frac{\alpha_{00}}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}} \\ \frac{\alpha_{01}}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}} \\ 0 \\ 0 \end{bmatrix}$. This corresponds to zeroing out all the vector entries corresponding to states of the first qubit incompatible with the outcome measurement (in this case, all the vector entries corresponding to state 1 for the first qubit) and normalizing the remaining non-zero vector entries (compatible with the outcome measurement, in this case 0 for the first qubit) so that the vector has norm 1.

For simplicity, we will omit presenting the most general case for partial quantum measurement here. It is sufficient to note that the same intuition would hold as for the simple 2-qubit system presented previously: the vector entries corresponding to states incompatible with the outcome measurement would be zeroed out, while the remaining non-zero entries (compatible with the outcome measurement) would be normalized (to vector norm 1).

Partial measurement bears some similarities to the ReLU nonlinearity (see Section 1.5.4.2), in the sense that it zeroes out some entries of the input vector, and to some methods similar to batch normalization (see Section 1.2.2), in the sense that the output vector stays normalized (to vector norm 1) after quantum measurement. Since full measurement can be interpreted as a special case of partial measurement, it bears the same similarities to ReLU and batch normalization.

6.3 Discrete-variable quantum neural networks using partial measurement

In this section, we will discuss a natural implementation of quantum neural networks, but which is difficult to simulate classically using backpropagation. It is this difficulty that motivated our proposal for hybrid classical-quantum neural networks.

Arguably, as of today, there is no standard quantum neural network architecture

Neural network concept	Quantum computing concept
Activations of a single neural network layer	Quantum state
Matrix-vector multiplication	Unitary evolution operator
Nonlinearity (stochastic)	Measurement (partial or full)
Output layer (e.g. for classification)	Expectation of repeated partial measurements

Table 6.1 – Potential implementations for neural network concepts using quantum computing concepts. [203] proposed using full quantum measurement to implement a neural network's nonlinearity (activation function), while [161] used partial measurement. [79] used the expectation of repeated partial quantum measurements as output layer for a binary classification task (to provide the predicted class probabilities).

(as would be the case in classical deep learning for e.g. multi-layer perceptrons, convolutional neural networks, or recurrent neural networks). This is both due to the fact that quantum machine learning is a nascent research field and to the lack of wide, easy access to quantum computers to experiment with. As of today, various quantum neural network models have been proposed [193], [161], [124] [189], including some attempts to unify previous proposals in a coherent framework [189].

One can argue that the most natural implementation of a quantum neural network, using the discrete-variable quantum circuit model of quantum computation, would include the components described in Table 6.1. We will now discuss in more details each of these components and some papers which used them.

One of the earliest attempts at implementing nonlinearity in a quantum neural network (QNN) is [203] (this approach is also reviewed in [169]). [203] proposed using full quantum measurement as nonlinear activation function, after performing a unitary transform on the quantum state (the unitary transform being the equivalent of matrix-vector multiplication in classical neural networks). The proposal in [161], on the other hand, uses partial quantum measurement (also after performing a unitary transform on the quantum state) to construct quantum autoencoders. This proposal could be interpreted as a generalization of the proposal in [203] (since partial quantum measurement can be seen as a generalization of full quantum measurement). [169] remarked that, at least at the time the paper was written (around 2014), 'the idea to use quantum measurements to simulate the nonlinear convergence of associative memories seems to be the most mature solution to the dynamics incompatibility problem encountered in the quest for a QNN model.'

For classification purposes, information needs to be output from the quantum computer, which means that some measurement needs to be performed. For binary classification (the simplest case and the most interesting one for this chapter), the simplest solution is to perform partial measurement of one qubit. By running the quantum circuit multiple times and taking repeated measurements, we can then obtain an estimate of the probability of measuring the qubit in the state 1 and interpret this as the probability of class 1 (for binary classification). An estimate of the probability of class 0 can then be obtained by substracting the estimate of the probability of class 1 from 1. [79] uses the expectation of repeated partial measurements (of a single qubit) to provide the predicted probability for a binary classification task. For an estimation of the predicted probability of $O(\frac{1}{\epsilon})$ accuracy, the method from [79] requires $O(\frac{1}{\epsilon^2})$ repeated measurements (and, thus, $O(\frac{1}{\epsilon^2})$ repeated runs of the full QNN circuit).

The QNN proposal in Table 6.1, while intuitive, is difficult to simulate classically. Because of the stochasticity of quantum measurement (whether it is full or partial), this nonlinearity is stochastic and, in our experience, this leads to difficulty in simulating the training of this proposal with simple backpropagation for multiple hidden layers. It is this difficulty in simulating, arguably, the most natural model of a discrete-variable quantum neural network, which led to our proposal combining classical and quantum computation. Our proposal keeps the analogy between unitary evolution and matrix-vector multiplication, but we no longer use partial measurement for nonlinearity. Instead, a classical neural network generates the parameters of the quantum circuit dynamically based on the input, leading to nonlinear dynamics in the evolution of the quantum state.

6.4 Introduction to hypernetworks

In this section, we concisely present classical hypernetworks [100], the classical deep learning proposal which inspired our hybrid classical-quantum proposal, as well as a previous related work which provided inspiration for our proposed system [80].

Our first source of inspiration for our hybrid classical-quantum proposal is the work of [80], which showed that nonlinear dynamics can be achieved without using any explicit (element-wise) nonlinearities. [80] introduces a RNN architecture composed of input-switched affine transformations, where the recurrent weights are input-dependent. More precisely, the RNN's hidden state h_t (at time t) depends on the input x_t as described in the following equation:

$$h_t = W_{x_t} h_{t-1} + b_{x_t} \tag{6.6}$$

This equation illustrates how the trainable parameters (the weights W_{x_t} and the biases b_{x_t}) depend on the input x_t at every timestep t.

In [80], this architecture was used for character-level and, respectively, word-level language modelling, and was shown to achieve results comparable to the LSTM architecture, especially as the number of trainable parameters is increased.

[100] proposed what can be interpreted as an extension of [80] and is the work from classical deep learning closest to our own proposal. In [100], a new architecture is proposed, named hypernetworks, in which one network, also known as a hypernetwork, is used to generate the weights for another network (the main network). This architecture is shown to provide state-of-the-art results at the time it was proposed for both convolutional and recurrent neural networks. We will focus on the adaptation to recurrent neural networks, since it is the most relevant to our own work.

When adapted to RNNs, the hypernetwork architecture will use a hyper RNN to generate the weights of a main RNN. These weights can be different at each timestep and for each input sequence. This is illustrated in Figure 1. The motivation behind this design, as presented in [80], is to challenge the paradigm of tying the RNN weights at each timestep, while potentially allowing for increased expressiveness. In this manner, a large set of weights can be generated for the main RNN, while only using a small number of trainable parameters (in the hyper RNN).

The RNN cell used for both the hyper and the main RNN in [100] is the LSTM. The hyper LSTM will get as input both the previous hidden states of the main LSTM and the actual input sequence of the main LSTM (concatenated). The outputs of the hyper LSTM can be interpreted as embedding vectors (denoted by z_t in Fig. 2), which will be used to generate the weights of the main LSTM (denoted by $W(z_t)$ in Fig. 2).

The main result of the paper is that hyper LSTMs can achieve results close to the state of the art for a variety of sequence modelling tasks, which include character-level language modelling, handwriting generation and neural machine translation. The hyper LSTM (and hyper RNNs, in general) can be seen as an extension of input-switched affine networks [80], in which the weights are defined at every timestep using an additional RNN, rather than just being chosen from a discrete set, based on the input from the current timestep.

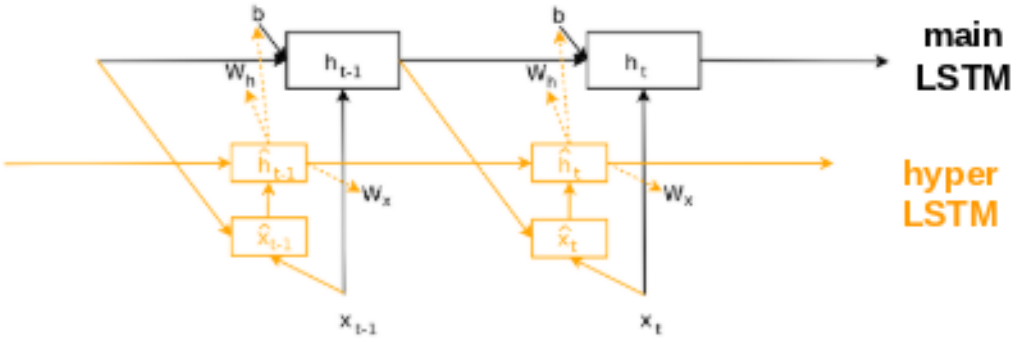


Figure 6.3 – Recurrent hypernetwork architecture: the hyper LSTM generating the weights of the main LSTM (at every timestep). Adapted from [100].

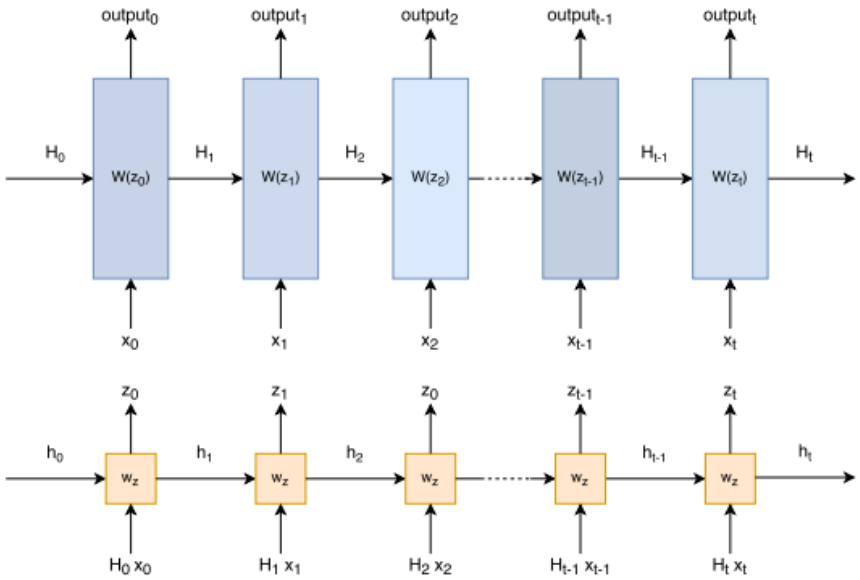


Figure 6.4 – A more detailed view of the hyper RNN in Fig. 1. The hyper RNN (shown in orange) takes in, at every timestep, the previous hidden state of the main RNN and the current input of the main RNN, and outputs the embedding vector z_t, from which the weights of the main RNN will be generated (denoted by $W(z_t)$). Figure from [100].

6.5 Proposed parameterization

In this section we will describe the parameterization of our hybrid classical-quantum RNN. We will first describe the quantum component, followed by the output layer, the loss function and, finally, the classical RNN which generates the parameters of the quantum RNN. A simplified illustration of our proposal is shown in Fig. 6.5.

6.5.1 Quantum (main) RNN

We will denote the state of our main (quantum) RNN at time t by h_t. The main RNN's initial state h_0 is the uniform superposition. This uniform superposition can be obtained starting from the n-qubit $|0..0\rangle$ state with $O(n)$ computational cost using n Hadamard gates. At every time step t, our quantum RNN's state h_t is obtained by multiplying its previous hidden state h_{t-1} with the matrix U_t, which is generated from the classical RNN's hidden state:

$$h_{t+1} = U_t * h_t \tag{6.7}$$

We decompose the matrix U_t into the following parameterization:

$$U_t = A_t * F * D_t * F^{-1} \tag{6.8}$$

where F is the Quantum Fourier Transform (QFT), F^{-1} is the Inverse Quantum Fourier Transform (IQFT) and A_t and D_t are parameterized unitary matrices, with their parameters generated by a classical hyper-RNN. More precisely, A_t and D_t are tensor products of general 1-qubit unitary transforms. These 1-qubit unitaries are general 2 x 2 unitary matrices [35], so A_t and D_t can be written as the tensor products of n 2×2 unitary matrices, as described in Sections 6.2.2.1 and 6.2.2.2.

For readability, we rewrite the equation of a general 2×2 unitary A already presented in Section 6.2.2.1:

$$A = e^{\frac{i\alpha}{2}} \begin{bmatrix} e^{i\beta}\cos(\phi) & e^{i\gamma}\sin(\phi) \\ -e^{-i\gamma}\sin(\phi) & e^{-i\beta}\cos(\phi) \end{bmatrix} \tag{6.9}$$

where α, β, γ and ϕ are modifiable parameters. We will denote all the modifiable (i.e. trainable) parameters used at time t to generate A_t and D_t by θ_t.

The QFT and IQFT can be implemented with $O(n\log(n))$ [101] computational cost. As discussed in Section 6.2.2.1, general 1-qubit unitaries can be implemented using 4 consecutive elementary 1-qubit gates [52] [150]. Thus, A_t and D_t are of size

$2^n \times 2^n$, include $4 * n$ trainable parameters only and their respective computational complexities are $O(n)$ (similarly to the discussion in Section 6.2.2.2). We will use these complexities when we approximate the computational cost of our hybrid classical-quantum RNN if it were implemented using a quantum computer.

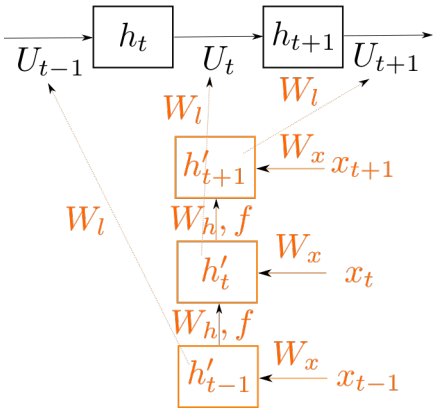


Figure 6.5 – Example of a hybrid classical-quantum recurrent neural network similar to those we simulate. A recurrent hypernetwork (in orange) implemented on a classical computer dynamically generates unitary transforms for a main network (in black), implemented on a quantum computer. The unitary matrices U_t are dynamically generated by the classical network for each input (sequence) from the classical network activations h'_t, using the fixed matrix W_l. No nonlinearity is used in the quantum neural network. We illustrate the hypernetwork as a typical RNN, with the same weight matrix W_h applied at every time step, followed by elementwise nonlinearity f, but, in principle, any architecture could be used. The inputs x_t are only provided to the hyper-RNN, processed through the fixed matrix W_x. The figure is simplified for illustration purposes. In our experiments, an LSTM is used as the hyper-RNN.

6.5.2 Output layer

For simplicity and to match our simulations which will be described in Section 6.6, we will suppose that our hybrid classical-quantum RNN is used to solve a many-to-one (T inputs, single output) sequential task with 2 output classes (binary classification setting). We denote the probability of predicting the class 0 as p_0 and the probability of predicting the class 1 as p_1.

To predict the probability p_1 of the class 1, we compute the complex dot product between the last state of the main (quantum) RNN h_T and a complex vector of norm 1 v (this vector contains trainable parameters and its norm is fixed at value 1 during the entire training procedure by renormalizing it after every update of its trainable parameters). We then substract the absolute value of the resulting complex number

from 1 (we could also have used the absolute value directly, but we have found the previously described procedure to work better empirically, similarly to [161]):

$$p_1 = 1 - |h_T \cdot \frac{v}{||v||}| \tag{6.10}$$

h_T being the hidden state of the quantum network at the final time step T. The dot product of complex unit vectors which we use is equal up to constants of proportionality with the Euclidean distance of such vectors [196], which can be implemented using quantum RAM (QRAM), as described in [139], with computational cost $O(n)$ for n qubits. As described in [139] [150], p_1 can be estimated to accuracy ϵ using quantum counting with computational complexity $O(\frac{1}{\epsilon})$. To obtain this estimate to accuracy ϵ, $O(\frac{1}{\epsilon})$ repeated runs of the entire quantum RNN have to be executed.

The predicted probability of the class 0 is then $p_0 = 1 - p_1$. An example's predicted label will then be 1 if $p_1 > 0.5$ (and 0, otherwise). We discuss how we approximate the classification accuracy (on the validation and test sets) when estimating p_1 to accuracy ϵ in Section 6.6.4.

We could extend this approach to discriminate between more classes by using one-vs.-rest classifiers with separate v as in Eq. (6.10) (which we would denote by v_c) for each class c. If we denote the number of classes by C, we would also obtain C different scores p_c using the following equation, similarly to (6.10):

$$p_c = |h_T \cdot \frac{v_c}{||v_c||}| \tag{6.11}$$

The different scores p_c could be normalized to sum to 1 (by dividing each p_c by $\sum_{c'} p_{c'}$). The final prediction would be the class c corresponding to the highest score p_c and the loss function to be minimized could be the negative log-likelihood of the ground-truth class. This would increase the computational complexity (both for inference and for training) by a factor equal to C, because the final measurement would have to be repeated for each separate class.

6.5.3 Loss function

The loss we have used is a max-margin loss similar to the one used in [164]. The mathematical expression for the loss function L is:

$$L = y * \max(0, m^+ - p_1)^2 + (1 - y) * \max(0, p_1 - m^-)^2 \tag{6.12}$$

where y is the ground-truth label, which we assume 1 for class 1 and 0 for class 2 and m^+ and m^- are hyperparameters.

The intuition behind this expression is that, in the case of ground-truth label 1, we only penalize the probability p_1 if it is lower than m^+ (a similar argument applies for p_0 and ground-truth label 0). This corresponds to maximizing the absolute difference (margin) between the prediction probabilities for the two classes, $|p_1 - p_0|$. Intuitively, we want the predictions of our classifier to be as confident as possible, so that they are less affected by the noise of the ϵ-accuracy sampling obtained using quantum counting. A classifier with more confident predictions would allow us to choose higher values for ϵ and, thus, we would need to repeat the quantum measurement fewer times, leading to reduced computational costs.

A similar loss function and similar motivation were used to train quantum networks in [114], concurrently to our own work.

6.5.4 Classical (hyper) RNN

We use as the classical (hypernetwork) RNN a single layer Long Short-term Memory (LSTM) architecture [112], without any peephole connections, to generate the modifiable parameters of the quantum RNN. We denote the hidden state of the network at time t by h'_t and the inputs at time t by x_t.

The LSTM equations are:

$$c_t = f_t \bullet c_{t-1} + i_t \bullet \tanh(W_{hc} * h'_{t-1} + W_{xc} * x_t + b_c) \tag{6.13}$$

$$f_t = \sigma(W_{xf} * x_t + W_{hf} * h'_{t-1} + b_f) \tag{6.14}$$

$$i_t = \sigma(W_{xi} * x_t + W_{hi} * h'_{t-1} + b_i) \tag{6.15}$$

$$o_t = \sigma(W_{xo} * x_t + W_{ho} * h'_{t-1} + b_o) \tag{6.16}$$

$$h'_t = o_t \bullet \tanh(c_t) \tag{6.17}$$

where $\bullet$ signifies element-wise multiplication and σ denotes the sigmoid nonlinearity often used in neural networks. The multiplications by the weight matrices W_{hc}, W_{hf}, W_{hi} and W_{ho} depends on the previous hidden state h'_{t-1}, so cannot be

parallelized across time steps. On the other hand, the multiplications of the matrices W_{xc}, W_{xf}, W_{xi} and W_{xo} with x_t can be performed in parallel across all time steps $t \in 0, ..., T$. We denote the action of the matrices W_{hc}, W_{hf}, W_{hi} and W_{ho} as *hidden-to-hidden transition* and the action of the matrices W_{xc}, W_{xf}, W_{xi} and W_{xo} as *the input-to-hidden transition*. The action of U_t in Eq (6.7) can also be interpreted as a *the hidden-to-hidden transition*, since it depends on the previous quantum state h_t. This observation will become relevant when we discuss the estimated computational cost of our proposal andof the classical LSTM benchmark in Section 6.6.

A new unitary matrix U_t as parameterized by Eq. (6.7) is generated at every time step t, with the free parameters generated from the state of the classical hypernetwork h'_t. The free parameters of the quantum RNN at time t, θ_t (used to construct A_t and D_t), are obtained from the hiddens units h'_t using a linear layer (composed of matrix W_l and biases b_l):

$$\theta_t = W_l * h'_t + b_l \tag{6.18}$$

We will denote the application of the matrix W_l (and biases b_l) as *the hidden-to-output-parameters transition*.

The same architecture is used as a baseline for standard classical RNN architectures to compare our parameterization against in Section 6.6.

6.6 Simulation results

In this section, we will describe the experiments we have performed. We will start with the reasoning behind our choices of task and dataset, followed by detailed descriptions of the systems implementing our proposal and the baseline LSTM. We then present the calculation of the approximation computational costs of these systems and accuracy estimations under ϵ-sampling approximation.

6.6.1 Task and dataset

We are interested in showing that our proposed hybrid classical-quantum RNN can perform similarly to a classical RNN in long-term sequential dependency tasks, with potentially exponentially less computation required. For this, we will choose a task similar to the permuted pixel MNIST task, a standard benchmark for the long-term memory capabilities of RNN architectures [121] [38].

We will show results on the Fashion-MNIST dataset [198] [199]. The motivation

for choosing this dataset is that it is similar to the well-known MNIST dataset (same input dimensions, same number of datapoints and same splits), but it is more challenging. The task we consider consists of first permuting the input pixels of the images (using the same permutation for the entire dataset). The pixels are then input one by one and the system has to classify the resulting sequence.

This task (Fashion MNIST) is the same as the permuted pixel MNIST task. RNN architectures (GRU) have already been benchmarked on the Fashion-MNIST dataset [198], but in a setting where a full column of pixels is input at every time step (instead of a single pixel), leading to shorter time dependencies.

To reduce training time (especially for the simulation of the hybrid classical-quantum systems), we will also perform simulations where we resize the images and / or reduce the size of the training set. We will also occasionally vary the batch size. This is motivated by the fact that when simulating quantum RNNs with high number of qubits (e. g. 13), larger batch sizes (e. g. 100) no longer fit in our GPU memory. In these cases, we reduce the batch size when simulating our parameterization. To provide a fair comparison, we then also show results with the same batch size for the classical baseline.

To simplify the problem of having to produce outputs from the quantum computer, we will also simplify the tasks to only discriminating between 2 classes of Fashion-MNIST. This simplifies the problem of outputting a result from the very high-dimensional space of the quantum RNN. At the same time, we will try to choose the most difficult 2 classes to discriminate between, so that the differences between the capabilities of different RNNs become easier to distinguish and less affected by the inherent noise of the random parameter initialization, training procedure, etc. In [114] a similar methodology has been used, for providing simulation results for a quantum tensor network on the MNIST dataset (simplifying the problem to distinguishing 2 classes instead of the original 10).

We will thus use classes 4 (representing coats) and 6 (representing shirts) in this task [198]. Some example images from these classes can be seen in Fig. 6.6 and Fig. 6.7. The training sets will vary from 1000 images from each class up to 5000 images from each class (corresponding to totals of 2000 and 10000 training samples). The validation sets contain 1000 images from each class. Both of these sets are composed of images picked randomly without replacement from the entire original training set (of 6000 images for each class). We keep the entire test set unmodified (consisting of 1000 images from each class).

We will denote each separate setting with specific image size (sequence length),

Figure 6.6 – Examples of coats from Fashion-MNIST

Figure 6.7 – Examples of shirts from Fashion-MNIST

training set size and batch size by the term *task setting*.

6.6.2 System details

In this subsection we describe the details of the systems we have simulated, namely the baseline classical LSTM and our hybrid classical-quantum parameterization.

6.6.2.1 Baseline classical LSTM

As the classical baseline, we have used a single layer Long Short-term Memory (LSTM) architecture [112], without any peephole connections. We have presented the LSTM architecture in Section 1.3.3.2. To predict the probability of the first of the two classes, we compute the dot product between the hidden state at the last time step h_T and a vector of same dimension of real trainable parameters v (which includes the weights of the output layer) and interpret the result as a logit. The loss function we use is the logistic loss. If we denote the probability of class 1 by p_1, we have:

$$\log(p_1) = \sigma(h_T \cdot v) \tag{6.19}$$

where σ is the sigmoid function.

The logistic loss L is defined by:

$$L = -y * \log(p_1) - (1 - y) * \log(1 - p_1) \tag{6.20}$$

The probability of class 0 is $p_0 = 1 - p_1$.

We have verified empirically that this approach performs similarly to the usual

approach of using a softmax output and cross-entropy loss function (without any significant loss in prediction accuracy).

We initialize all the LSTM weights using an orthogonal initializer [166] and initialize the biases of the forget gates to 1, to ease the learning of long-term dependencies, as suggested for example in [122]. The initial state h_0 is initialized to 0 and the rest of the biases are initialized to 0. The weights and biases of the output layer are initialized, for simplicity, using a Gaussian distribution (of mean 0 and variance 1).

6.6.2.2 Hybrid classical-quantum parameterization

We have used the same initialization for the classical hyper-RNN (LSTM) in our proposed parameterization as for the classical benchmark LSTM. The real and imaginary components of the output layer v in Eq. (6.10) were initialized using a truncated Gaussian distribution with mean 0 and standard deviation 0.01. The weights W_l in Eq. (6.18) are initialized to 0 and the biases b_l to 1.

For all the results we report, we have used the values $m^+ = 0.7$ and $m^- = 0.3$ for the margin loss function of our proposal. These values seemed to work best in our short hyperparameter search trials. Empirically, in our simulations we have found that increasing m^+ to higher values than 0.7 and decreasing m^- below 0.3 leads to more confident predictions, but makes training more difficult. It is possible that better values can be found with a much more extensive hyperparameter search.

6.6.2.3 Common settings

For optimization, we have used ADAM [125] with default hyperparameters (including learning rate 0.001) for both the classical baseline and our proposed parameterization. For both systems, gradient clipping by global norm is used [152], with global norm 5.

For most task settings, we use batches of size 100 (unless specified otherwise). The optimization is always run for 100 epochs. Validation is performed after every 10000 examples. We keep the best validation accuracy and the corresponding test accuracy.

For the baseline LSTM system, we will run multiple configurations for each task setting. We double the number of units for each new configuration, corresponding to approximately 4 times more trainable parameters. Increasing the number of units increases the recurrent memory and increasing the number of trainable parameters leads to more expressive models, but also increased risk of overfitting and higher computational cost. For each configuration of the baseline system, we train the system

multiple times (corresponding to different random seeds from which the trainable parameters are initialized) for each setting with a fixed number of units and select the resulting system with the best accuracy on the validation set, for which we also report the test accuracy. This is a more challenging setup for our proposal, since the baseline has several chances to obtain good performance, while we only run our proposal once for each task setting.

6.6.3 Approximate computational cost

In this subsection, we describe the approach we have used for approximating the computational cost for our proposal and for the baseline. For simplicity, we will use a quick 'back-of-the-envelope' computational cost approximation. Obviously, in the case of an implementation on quantum computers (not a simulation, as in our case), a lot more factors would come into account which we neglect here for simplicity, most notably the extra cost of quantum error correction / noise management. We will focus on the cost at inference / deployment and discuss the training cost in Section 6.8.

Our approach is similar to how the computational cost is estimated in [121], which proposes a Kronecker-factored RNN similar to our own, but doesn't make use of quantum computation. Notably, we will only take into account the dominant cost of applying the recurrent transition (corresponding to hidden-to-hidden matrices both for the quantum and the classical systems, as discussed in Section 6.5.4). For a sequence of T inputs, the hidden-to-hidden transition is applied T times, in contrast to the output operation, which is only applied once. The hidden-to-hidden transition is also different from the input-to-hidden transitions in the classical LSTMs and the hidden-to-output-parameters transition in the hyper-RNN because it can't be parallelized across the time steps (the hidden-to-hidden transition must be applied sequentially, in order, at every time step). We further neglect the cost of using any biases, because asymptotically it is quadratically smaller than the cost of the matrix multiplications.

In the case of the benchmark single-layer LSTM comprising h hidden units, we will approximate the cost of running it at every time step by $4 * h^2$. We use this approximation because an LSTM has 4 hidden-to-hidden matrices of size h^2: W_{hc}, W_{hf}, W_{hi} and W_{ho}, as described in Section 6.5.4.

The approximate computational cost of our hybrid classical-quantum RNN parameterization is obtained in the following manner. First, the cost of the classical hyper-RNN (an LSTM) is obtained like for the classical benchmark.

We decompose the cost of the quantum RNN in the cost of the QFT and IQFT and the cost of the parameterized unitaries. Because the QFT and IQFT can be implemented on a quantum computer in time $O(n\log(n))$ [101] and we simulate between 8 and 16 qubits, for simplicity, we will approximate these costs as $4 * n$. Because the parameterized unitaries only use single-qubit gates and each such gate can be decomposed into 4 elementary gates, as described in Section 6.5.1, we will also approximate these costs as $4 * n$. The entire cost per time step of the quantum RNN for one run is then approximated as $4 * 4 * n$, the sum of the costs for the QFT, the IQFT and the parameterized unitaries.

Since we need to perform repeated measurements after the last time step to obtain the approximate probability p_1 (and thus, repeated runs of the entire quantum RNN), as described in Section 6.5.2, we also need to consider the costs of these repeated runs. As described in Section 6.5.2, p_1 can be estimated to accuracy ϵ with computational complexity $O(\frac{1}{\epsilon})$. For simplicity, we will approximate this cost as $\frac{1}{\epsilon}$. The approximate computational cost per time step for the quantum RNN will then be $\frac{4 * 4 * n}{\epsilon}$ and the same cost for the entire hybrid classical-quantum system will be $4 * h^2 + \frac{4 * 4 * n}{\epsilon}$.

6.6.4 Accuracy estimation under ϵ-sampling approximation

We estimate the validation and test accuracies of our hybrid systems when repeated measurements to ϵ approximation by considering that the number of correct predictions is the sum of the number of correct, confident predictions ($\text{sign}(p_1 - 0.5) = \text{sign}(2 * l - 1)$ and $|p_1 - 0.5| > \epsilon$) and half of the number of predictions which are not confident enough ($|p_1 - 0.5| < \epsilon$). Here l denotes the ground-truth label, which can be 0 or 1 (for the case of binary classification). The intuition behind this procedure is that, for simplicity, we suppose that for half of all the examples for which the prediction is not confident enough, the ϵ approximation would lead to a correct prediction (and for the other half, the prediction would be incorrect).

6.7 Experimental results

In this section we present the results obtained for the various task settings (number of pixels of input image, number of training examples, batch size) we have considered. We present these settings and the corresponding results in the order of increasing temporal dependencies length and in the order of increasing training set size and

increasing batch size. We denote the classical benchmark systems by LSTM-[h], where h is the number of hidden units. For our proposal, we use the notation LSTM-[h']-[n]q, where h' is the number of units of the hyper-LSTM and n the number of qubits in our quantum parameterization. When presenting the results of the ϵ-sampling approximations, we add the notation ϵ = [value]. We use the notation AFDF-[n]q to denote the quantum component of our system.

We first show the approximate computational cost per time step for each of our classical and quantum system components in Table 6.2. In table 6.3 we display the extra cost factors resulting from repeated measurements under different ϵ approximations. We have described how the total approximate cost per time step for a system is obtained from the cost of the classical and quantum subsystems, taking into account the ϵ-sampling approximation, in Section 6.6.3. For convenience, we will display the total approximate cost per time step for each system in each task setting below (except for the results obtained for our proposal when no ϵ-accuracy is used, since the cost would depend on ϵ).

System	Approximate computational cost
LSTM-7	196
LSTM-16	1024
LSTM-32	4096
LSTM-64	16384
LSTM-128	65536
LSTM-256	262144
LSTM-512	1048576
AFDF-9q	144
AFDF-10q	160
AFDF-11q	176
AFDF-13q	208

Table 6.2 – Approximate computational cost (operations) per time step for each system component (classical LSTM and quantum AFDF).

ϵ approximation	0.01	0.02	0.03
Extra cost factor	100	50	34

Table 6.3 – Extra cost factor from repeated measurements under different ϵ approximations.

In Table 6.4, we show the results for the task setting of 2000 training examples of 14 x 14 pixel images and batch size 25. If no ϵ-sampling approximation were used, our

proposed system (LSTM-16-13q) would obtain the best validation accuracy and only slightly worse test accuracy than LSTM-256. The approximate computational cost when $\epsilon = 0.03$ sampling is used is around 32 times lower than for LSTM-256. The validation and test accuracy for this approximation, though, are comparable to the LSTM-32 (which would be of comparable computational cost to the approximation).

System	Best validation accuracy	Test accuracy	Approximate computational cost
LSTM-16	0.7675	0.7505	1024
LSTM-32	0.8330	0.8030	4096
LSTM-64	0.8020	0.7920	16384
LSTM-128	0.8125	0.7725	65536
LSTM-256	**0.8345**	0.8120	262144
LSTM-512	0.8295	**0.8125**	1048576
LSTM-16-13q	**0.8385**	0.8075	-
LSTM-16-13q $\epsilon = 0.03$	0.8335	0.8015	8096

Table 6.4 – Comparison in terms of accuracy and computational costs between classical LSTMs and our proposal. Results for 2000 training examples, 14 x 14 pixels, batch size 25.

Table 6.5 shows the results for a training set of 2000 samples of 14 x 14 pixel images and batch size 100. Our proposal (LSTM-7-11q) obtains validation and test accuracies better than LSTM-128 (but worse than LSTM-256 and LSTM-512), for a computational cost which would be e.g around 10 times lower than for LSTM-128, under a 0.03 sampling approximation.

In Table 6.6, we show the results obtained for 10000 training examples of 14 x 14 pixel images and batch size 100. LSTM-16-9q obtains a validation accuracy slightly worse that that of LSTM-64 and test accuracy somewhat worse than LSTM-128. For a 0.01 sampling approximation, the accuracy would be comparable to LSTM-64, for comparable cost.

In Table 6.7, we show the results for 2000 training examples of 20 x 20 pixels and batch size 100. Our proposed hybrid system (LSTM-16-10q) obtains the best validation and test accuracies, with LSTM-64 performance second best. Under a 0.03 sampling approximation, our proposal would be about 3 times more computationally efficient than the LSTM-64.

We show the best results our proposal obtains in Table 6.8, for 2000 training examples of 28 x 28 pixel images and batch size 100. Our proposed system (LSTM-

System	Best validation accuracy	Test accuracy	Approximate computational cost
LSTM-7	0.7650	0.7555	196
LSTM-16	0.7330	0.7020	1024
LSTM-32	0.7700	0.7450	4096
LSTM-64	0.7700	0.7710	16384
LSTM-128	0.7925	0.7665	65536
LSTM-256	0.8235	0.8100	262144
LSTM-512	**0.8300**	**0.8170**	1048576
LSTM-7-11q	0.7995	0.7880	-
LSTM-7-11q $\epsilon = 0.03$	0.7910	0.7805	6180

Table 6.5 – Comparison in terms of accuracy and computational costs between classical LSTMs and our proposal. Results for 2000 training examples, 14 x 14 pixels, batch size 100.

System	Best validation accuracy	Test accuracy	Approximate computational cost
LSTM-16	0.8045	0.7900	1024
LSTM-32	0.8570	0.8280	4096
LSTM-64	0.8660	0.8475	16384
LSTM-128	**0.8840**	**0.8590**	65536
LSTM-16-9q	0.8655	0.8510	-
LSTM-16-9q $\epsilon = 0.01$	0.8625	0.8490	15424
LSTM-16-9q $\epsilon = 0.02$	0.8590	0.8432	8224
LSTM-16-9q $\epsilon = 0.03$	0.8560	0.8365	5920

Table 6.6 – Comparison in terms of accuracy and computational costs between classical LSTMs and our proposal. Results for 10000 training examples, 14 x 14 pixels, batch size 100.

System	Best validation accuracy	Test accuracy	Approximate computational cost
LSTM-16	0.7520	0.7430	1024
LSTM-32	0.7725	0.7465	4096
LSTM-64	**0.8115**	**0.8005**	16384
LSTM-128	0.7875	0.7880	65536
LSTM-256	0.7895	0.7900	262144
LSTM-512	0.7940	0.7945	1048576
LSTM-16-10q	**0.8280**	**0.8035**	-
LSTM-16-10q $\epsilon = 0.03$	**0.8125**	0.7955	6464

Table 6.7 – Comparison in terms of accuracy and computational costs between classical LSTMs and our proposal. Results for 2000 training examples, 20 x 20 pixels, batch size 100.

16-9q) obtains the best validation and test accuracies, with LSTM-256 performance second best. Under a 0.03 sampling approximation, our proposal would be about 44 times more computationally efficient than the LSTM-256. Under a 0.01 sampling approximation, our proposal would still be about **17 times more computationally efficient** than the LSTM-256 and would obtain the **best accuracy both on the validation and on the test sets** among all the benchmarked system for this task.

Finally, in Table 6.9, we show the results obtained for 10000 training examples of 28 x 28 pixels and batch size 100. Here LSTM-16-9q obtains a validation accuracy better than all LSTM systems with less than 128 units and test accuracy better than all LSTM systems with less than 512 units. LSTM-512 obtains the best validation and test accuracies, with LSTM-128 obtaining equal validation accuracy (but worse test accuracy). Interestingly, the 9 qubits of our proposal correspond to exactly 512 states of the quantum system ($\log_2(512) = 9$). If we take into account the ϵ-sampling approximation, for $\epsilon = 0.01$ the test accuracy would be equal to that of LSTM-128, for a computational cost which would be around 4 times lower. Compared to LSTM-512, our method would be about 68 times more efficient for $\epsilon = 0.01$. This result shows that our proposal would still be computationally efficient even as the number of training examples is increased, requiring the statistical models to have more capacity, and suggests that the computational gains could be asymptotically exponential.

Generally speaking, we can observe that our quantum proposal does comparatively better (relatively to the classical benchmarks) as the tasks contain longer temporal dependences and, thus, require more memory. They also seem to do somewhat better

System	Best validation accuracy	Test accuracy	Approximate computational cost
LSTM-16	0.7260	0.7160	1024
LSTM-32	0.8115	0.7880	4096
LSTM-64	0.7735	0.7590	16384
LSTM-128	0.8055	0.7880	65536
LSTM-256	**0.8155**	**0.8010**	262144
LSTM-16-9q	**0.8205**	**0.8040**	-
LSTM-16-9q $\epsilon = 0.01$	**0.8180**	**0.8030**	15424
LSTM-16-9q $\epsilon = 0.02$	**0.8122**	0.7955	8224
LSTM-16-9q $\epsilon = 0.03$	0.8067	0.7935	5920

Table 6.8 – Comparison in terms of accuracy and computational costs between classical LSTMs and our proposal. Results for 2000 training examples, 28 x 28 pixels, batch size 100.

System	Best validation accuracy	Test accuracy	Approximate computational cost
LSTM-16	0.7920	0.7875	1024
LSTM-32	0.8560	0.8315	4096
LSTM-64	0.8585	0.8495	16384
LSTM-128	**0.8870**	0.8585	65536
LSTM-256	0.8840	0.8565	262144
LSTM-512	**0.8870**	**0.8745**	1048576
LSTM-16-9q	0.8675	0.8620	-
LSTM-16-9q $\epsilon = 0.01$	0.8642	0.8585	15424
LSTM-16-9q $\epsilon = 0.02$	0.8625	0.8557	8224
LSTM-16-9q $\epsilon = 0.03$	0.8578	0.8515	5920

Table 6.9 – Comparison in terms of accuracy and computational costs between classical LSTMs and our proposal. Results for 10000 training examples, 28 x 28 pixels, batch size 100.

when the training set is smaller, so that the statistical models require less capacity. Also notable should be the fact that the computational cost grows with the square of the number of units for the classical LSTM systems, while for the quantum AFDF parameterizations it is almost linear with the number of qubits, exponentially more efficient compared to the classical system, as can be observed in the tables presenting the results. The computational costs of the classical LSTM systems would quickly become much higher than those of our quantum parameterization, if only single runs (corresponding to single measurements) were required for the quantum system. This highlights the potential benefits of optimizing losses which encourage confident predictions, leading to fewer required repetitions followed by measurements of the quantum systems.

We also want to highlight the fact that for all of the presented simulations, our hybrid classical-quantum proposals obtain much higher performance than the purely classical baselines with the same numbers of neurons (e.g LSTM-16-11q vs. LSTM-16).

The losses in accuracies incurred from the ϵ-sampling approximations are relatively small and get smaller as smaller ϵ's are used. Furthermore, it seems reasonable to expect these results could be improved by doing a more thorough hyper-parameter search over the max-margin loss parameters. The extra computational cost resulting from repeated measurements could also be addressed if the quantum systems were run on multiple quantum processors, since the measurements are independent and trivially parallelizable.

Another potential advantage of our method could be its stability at training time, compared to the benchmark LSTMs. We have observed multiple times the LSTMs becoming unstable during training, due to exploding gradients, even when gradient clipping is used. On the other hand, we have never observed such behavior with our current hybrid classical-quantum RNN proposal during training.

6.8 Discussion

In this section we will discuss how our proposal compares to some recent works from the literatures of classical and quantum machine learning. Because both of these fields are vast, we won't aim for extensive reviews, but only a discussion of the most salient points.

Our work could address some potential limitations of relatively near-term quantum computers when implementing neural networks, such as limited input size and

difficulty of implementing nonlinear behaviors, by having a classical neural network generate the parameters of the quantum neural twork. Somewhat similar approaches (hybrid classical-quantum proposals which try to address limited input size for near-term quantum computers) were also presented in [153] and [41]. [79] proposed a QNN model aimed at being realisable on near-term quantum processors and used partial measurement to provide the prediction of the binary label (for binary classification). This proposal focused on feedforward QNNs whose training was simulated using stochastic gradient descent (SGD). [79] performed classical simulation of quantum neural networks for a binary classification task, on a downsampled version of MNIST restricted to two classes. The authors reported that they couldn't find any quantum advantage over classical neural networks for this task. Our proposal differs from the above-mentioned approaches by using a classical neural network which generates the parameters of the quantum neural network and by presenting results for RNNs.

A quantum version of a Hopfield network has been proposed recently [160]. [160] proposes encoding the network into the amplitudes of quantum states, resulting in potential exponential improvements in computational cost compared to the classical algorithm. They show that the quantum algorithm they propose obtains comparable accuracy to that of the classical algorithm. While we also use the idea of encoding a network into the amplitudes of quantum states, our classical hypernetwork *learns* this encoding. Furthermore, we present results for truly sequential tasks (while the inputs in [160] are fixed size) and compare against a baseline which is much closer to the state of the art (LSTMs, as opposed to Hopfield networks).

The most similar quantum neural network proposal to ours that we are aware of is [124]. Concurrent to our work, [124] proposed a continuous-variable quantum feedforward neural network whose parameters are generated dynamically by a classical feedforward neural network. In the framework of hypernetworks, this would correspond to a classical hyper feedforward network generating the parameters of a quantum main feedforward network. [124] also shows results for a classical simulation of this proposal for a credit card fraud detection task. Our work is different from [124], most significantly because we show results for recurrent networks (hyper - classical RNN and main - quantum RNN), but also because we use discrete-variable quantum circuits.

Our current main (quantum) network parameterization could be somewhat sub-optimal for implementation on a physical quantum computer in the following sense. Because the QFT and IQFT have $\mathrm{O}(n \log(n))$ computational cost and don't contain trainable parameters, while the parameterized components have $O(n)$ parameters

and computational cost $O(n)$, the computational cost of the entire parameterization comes down to $O(\log(n))$ / parameter. Ideally, a computational cost of $O(1)$ / parameter would be desirable. A similar argument has been used to motivate the introduction of other neural network parameterizations, such as the one in [120] with $O(1)$ / parameter computational cost (claimed to be an improvement over the parameterization in [147], with $O(\log(n))$ / parameter cost). We have chosen the current parameterization because it is relatively easy and computationally cheap for implementation and simulation on GPU using TensorFlow [36]. More expressive variations of our parameterization, with $O(1)$ / parameter cost and e. g. $O(n^2)$ parameters for each quantum matrix U_t, such as the one proposed in [161], should be straightforward to integrate into our framework.

Our approach also alleviates the requirement for quantum RAM of other quantum machine learning proposals, such as [139], and could potentially completely remove this requirement. The need for quantum RAM is problematic because, to this day, no large scale implementation exists and its scalability is uncertain. More precisely, quantum RAM requires exponential physical resources, even if access times are sub-exponential [84]. If our proposal were implemented exactly as in our simulations, we would already only require the use of quantum RAM once for every sequence, when we need to output the final prediction. Our output layer can be interpreted as a classifier working on inputs of very high dimensionality. We conjecture that we could replace the current output layer with a tensor network like Multiscale Entanglement Renormalization Ansatz (MERA) [190], without significant losses in statistical accuracy and while maintaining the computational efficiency of the quantum RNN, while no longer requiring quantum RAM. Promising results for classifiers working in very high dimensional spaces, using quantum tensor network parameterizations like MERA or Matrix Product States (MPS), have already been reported in the machine learning literature [138] [102] [180] [179]. We are interested in addressing this in future work.

Up to this point, we have addressed the challenges and opportunities of using our model to provide a quantum advantage only for inference (at deployment, after the training phase). We will now discuss how training could be performed.

In our simulations, we have performed gradient-based optimization, with gradients obtained using the backpropagation algorithm. The backpropagation (backprop) algorithm was not known to be generally feasible for quantum computing when our experiments were performed. Alternatives, such as finite differences, exist, but they would have been slower to simulate classically - and also to implement on

a quantum computer: finite differences requires a number of function evaluations (quantum neural network runs) which is linearly dependent on the number of trainable parameters. Recently, a feasible quantum backpropagation variant (named 'Baqprop') has been proposed by [189], which only requires a constant number of function evaluations, independently of the number of trainable parameters. This procedure would, in principle, be straightforward to use for training our quantum RNN (and, consequently, our entire hybrid classical-quantum proposal).

The results obtained in simulations suggest that our proposal could provide a quantum advantage for sequential classification tasks, in a many-to-one framework [135] (sequence of inputs, single output required), such as sentiment analysis or video classification [201]. Addressing many-to-one recurrent tasks using quantum computation can also be motivated intuitively using the following arguments. First, recurrent tasks might be easier to address because they result in computational graphs with very long sequences of operations and quantum computation has been shown to be especially advantageous (e.g. in proving quantum supremacy for near-term devices) for deeper circuits [51]. Furthermore, because many-to-one recurrent tasks only require outputs once at the end of the sequence, the difficulty of using outputs from a quantum computer while maintaining a computational advantage with regard to classical computation is also diminished.

The authors of [121] have already shown that the number of parameters in the recurrent weight matrix can be reduced by orders of magnitude compared to state of the art models like LSTMs, while keeping comparable statistical performance. Furthermore, their parameterization is similar to ours, in the sense of only requiring $O(\log(n))$ parameters for n hidden units (neurons) in the hidden-to-hidden RNN transition and of using a complex tensor product of the hidden-to-hidden RNN transition. The differences, though, come from the fact that they use a standard nonlinearity (the complex ReLU from [38]) and the unitarity of the transformation is only encouraged through a regularization penalty, not strictly enforced. Most importantly, the computational complexity of the approach in [121] would be exponentially higher than ours, since quantum computers are not used. The authors of [121] suggest that their results show that there are advantages for RNNs working in high-dimensional spaces, but the capacity of the recurrent part of the model can be drastically reduced without much degradation in model performance. Quantum computers (and particularly our quantum parameterization) could allow us to push this paradigm of high-dimensional recurrent space (with relatively low capacity recurrent dynamics) much further, potentially allowing for dimensions larger than the

number of visible particles in the universe for a reasonably-sized quantum computer of several hundreds of qubits.

We want to highlight the fact that decades of engineering have gone into making classical neural networks obtain state of the art performance for many tasks, while work on quantum neural networks is only in its infancy. It seems reasonable to expect that there is a lot of room for improving the performance of hybrid classical-quantum neural networks, including but not restricted to RNNs, and successfully extending this approach to more settings and tasks. If the quantum improvements our work suggests held and generalized to more settings and tasks, universal fault-tolerant quantum computers could present us with the opportunity of unparalleled neural network scalability, but also new challenges in terms of system transparency, interpretability and safety. Recent results in the literature provide more examples of machine learning settings and tasks for which quantum advantages seem promising. [81] shows that quantum computing can be used to build generative models which are as general as some standard deep learning classical counterparts but can be exponentially more expressive. Furthermore, these quantum models are intractable for classical computation (the computational cost of any classical algorithm approximating the quantum algorithm closely enough would be exponentially higher). The recently-proposed quantum Hopfield network [160] is another example of a neural network with potential exponential improvements compared to the classical algorithm. There have also been multiple proposals for discriminative algorithms; [123], for example, proposes a quantum classifier with running time polylogarithmic in the dimension and number of data points which, when simulated (on a classical compute), obtains 98.5% accuracy on MNIST (comparable to state of the art classical algorithms).

The results most interesting for general handwriting recognition are likely those pertaining to quantum computation advantages for sequential tasks in AI, due to the sequential nature of handwriting recognition. Multiple theoretical results suggest the quantum computing systems require less memory resources (than classical computation) to accurately model dynamical systems for sequential tasks [45] [78] [187]. It seems feasible to take advantage of these features of quantum computation to construct more efficient recurrent neural networks. Our own empirical results also suggest that quantum computation could lead to (potentially exponentially) more efficient recurrent neural networks.

6.9 Conclusion

In this chapter, we have presented preliminary results for a new hybrid classical-quantum neural network parameterization. To the best of our awareness, this is the first approach aimed at building quantum recurrent neural networks which shows potential improvements over a near state of the art classical RNN architecture. Though our results are preliminary, we believe this is one of the first studies which suggests the potential for quantum advantages for real-world machine learning tasks and datasets, especially in the case of sequence processing and recurrent neural networks.

Conclusions

In this last part of the thesis, we will first summarize our work, then provide a more speculative perspective on the subjects discussed in this thesis, discussing the promise of neural networks and quantum computation for solving handwriting recognition (at a human level of performance) and potential implications for artificial general intelligence.

In Chapter 3, we have improved the performance of CNNs for handwritten digit recognition using recent (at the time the work was performed) deep learning techniques.

In Chapter 4, we have introduced Tied Spatial Transformer Networks (TSTNs), Spatial Transformer Networks (STNs) with shared weights and different training variants resulting in improved performance on a distorted variant of the MNIST dataset.

In Chapter 5, we have compared the performance of Associative Long Short-Term Memory (ALSTM), a recently introduced recurrent neural network (RNN) architecture, against Long Short-Term Memory (LSTM), on the Arabic handwriting recognition IFN-ENIT dataset.

Finally, in Chapter 6, we have shown that integrating quantum computing with neural networks could provide computational gains (in terms of time and memory) for sequence recognition on the Fashion-MNIST dataset. Our results suggest that exponential improvements in computational requirements might be achievable, especially for recurrent neural networks trained for sequence classification.

We will now discuss how our work fits into the larger context of handwriting recognition and, more speculatively, artificial general intelligence. Some influential deep learning researchers have argued that supervised learning has effectively been solved using deep learning, as long as enough data is available [87].

Superhuman performance has already been achieved for some tasks such as object recognition and speech recognition, on large datasets (ImageNet [104] [206] for

object recognition and Switchboard [103] for speech recognition) . Given the relatedness between the domains (object recognition, speech recognition and handwriting recognition) and the success of similar neural network architectures in all three, superhuman performance for handwriting recognition tasks should also be feasible with similar methods. Given the growing amount of computation (Moore's law) and algorithmic innovations (often imported from related domains like computer vision, speech recognition or machine translation), the most significant bottleneck might be the available amount of labeled handwriting recognition data.

How does quantum computation fit into this picture? Quantum computing is likely not necessary to solve handwriting recognition, or even human-level AI, since the human brain most likely does not use quantum computation [137]. However, many recent theoretical and empirical results suggest that quantum computing could provide benefits for many AI problems.

The results most interesting for general handwriting recognition are likely those pertaining to quantum computation advantages for sequential tasks in AI, due to the sequential nature of handwriting recognition. Our empirical results in Chapter 6 suggest that quantum computation could lead to (potentially exponentially) more efficient recurrent neural networks. Multiple theoretical results also suggest that quantum computing systems require less memory resources (than classical computation) to accurately model dynamical systems for sequential tasks [45] [78] [187]. It seems feasible to take advantage of these features of quantum computation to construct more efficient recurrent neural networks in time and memory and, conversely, RNNs which obtain improved accuracy for similar computational resources.

The capacity to discriminate ordered sequences of stimuli and, more generally, the ability to represent and process sequential information and the increased memory capacity have been proposed as fundamental features which differentiate humans from other animals. The observed superior human performance on sequential tasks can be useful for a large variety of problems, some of which humans uniquely can solve, such as language understanding [83]. In this sense, solving handwriting recognition, for which deep learning and quantum computation seem appropriate and advantageous approaches, might be one of the important steps towards human-level AI. The requirements for less internal memory of quantum systems might also suggest (much more speculatively) that (potentially strongly) superhuman performance is plausible both for handwriting recognition and for artificial general intelligence.

Publications

Journal articles

1. Emilio Granell, Edgard Chammas, Laurence Likforman-Sulem, Carlos-D. Martínez-Hinarejos, Chafic Mokbel, Bogdan-Ionuţ Cîrstea, 'Transcription of Spanish Historical Handwritten Documents with Deep Neural Networks'; in Journal of Imaging, vol. 4, no. 1, p. 15, Jan. 2018 [91].

International conference papers

1. Bogdan-Ionuţ Cîrstea and Laurence Likforman-Sulem, 'Improving a deep convolutional neural network architecture for character recognition'; in Document Recognition and Retrieval XXIII (DRR 2016, San Francisco) [68].
2. Bogdan-Ionuţ Cîrstea and Laurence Likforman-Sulem, 'Tied Spatial Transformer Networks for Digit Recognition'; in 2016 15th International Conference on Frontiers in Handwriting Recognition (ICFHR), Shenzhen [70].

International symposiums and national conferences

1. Bogdan-Ionuţ Cîrstea, 'Recognition and information extraction in multi-lingual documents with Recurrent Neural Networks and Deep Neural Networks'; in the Doctoral Consortium of the ICDAR 2015 Conference, Nancy [67].
2. Bogdan-Ionuţ Cîrstea and Laurence Likforman-Sulem, 'Tied Spatial Transformer Networks for Character Recognition'; in Rencontres Jeunes Chercheurs of CIFED-CORIA 2016, Toulouse [69].

Appendices

Annex A

Other Achievements

I have participated to a common work with UPV (Polytechnic University of Valencia, Spain) and Telecom ParisTech, on the transcription of Spanish historical handwritten documents with deep neural networks. This work has been published in [91].

In this work, we propose combining optical recognition systems with language models based on sub-lexical units (characters or hyphenation sub-word units). The main motivation for using sub-lexical units language models is that they can naturally model Out-Of-Vocabulary (OOV) words, which can be especially problematic for handwriting recognition in historical documents. The optical recognition systems are based, in a set of experiments, on Hidden Markov Models (HMMs), and, in another set, on deep learning systems, namely Bidirectional Long Short-Term Memory (BLSTMs) and Convolutional Recurrent Neural Nets (CRNNs). The experiments are performed on the *Rodrigo* dataset of ancient Spanish manuscript. When HMM optical recognition systems are used, the language models based on sub-word units are shown to outperform those based on words for all the metrics studied, namely Word Error Rate (WER), Character Error Rate (CER) and Out-Of-Vocabulary (OOV) word accuracy recognition rate. When the deep learning optical recognition systems are used, the CRNNs are shown to outperform both the HMMs and the BLSTMs, reaching the lowest WER and CER for the *Rodrigo* dataset and they also significantly improve the OOV word accuracy recognition rate.

Annex B

Contributions à la reconnaissance de l'écriture manuscrite en utilisant des réseaux de neurones profonds et le calcul quantique

B.1 Introduction

Au cours des dernières années, l'apprentissage profond, le domaine d'étude des réseaux de neurones artificiels à couches multiples, a eu un fort impact sur de nombreux domaines de l'intelligence artificielle, dont la reconnaissance de l'écriture manuscrite. L'informatique quantique, en revanche, bien qu'elle ait une histoire qui s'étend sur plusieurs décennies, n'a été étudiée que très récemment pour des applications d'apprentissage machine (discutablement, pendant plusieurs années seulement). Une raison importante pour laquelle l'apprentissage par machine quantique en est à ses débuts est le manque d'ordinateurs quantiques pratiques et à grande échelle.

Dans cette thèse, nous apportons plusieurs contributions des domaines de l'apprentissage profond et de l'informatique quantique à la reconnaissance de l'écriture manuscrite.

Dans la première section, nous présentons notre travail qui intégre certaines des techniques d'apprentissage profond les plus récentes (en particulier au moment où le travail a été effectué), comme dropout, la normalisation par lots (batch normalization) et différentes fonctions d'activation dans des réseaux neuronaux convolutionnels et nous montrons une amélioration des performances sur le jeu de données MNIST.

Nous décrivons ensuite les Réseaux de Transformateurs Spatiaux Liés (RTSLs; en anglais: Tied Spatial Transformer Networks - TSTNs), une variante des Réseaux de Transformateurs

Spatiaux (RTSs; en anglais: Spatial Transformer Networks - STNs) précédemment introduits avec poids partagés. Nous décrivons également différentes variantes d'entraînement des Réseaux de Transformateurs Spatiaux Liés et montrons de meilleures performances sur une variante déformée du jeu de données MNIST.

Dans la troisième section, nous comparons la performance de l'architecture ALSTM (Associative Long Short-Term Memory), une architecture de réseau neuronal récurrent (RNN) récemment introduite, à celle de la LSTM (Long Short-Term Memory), sur le jeu de données IFN-ENIT de reconnaissance d'écriture arabe.

Nous décrivons ensuite brièvement les principes de base de l'informatique quantique que nous utilisons et proposons une architecture de réseau neuronal qui peut intégrer et tirer profit de l'informatique quantique. Bien que nos simulations soient effectuées à l'aide de calculs classiques (sur GPU), nos résultats sur le jeu de données Fashion-MNIST suggèrent que des améliorations exponentielles de la complexité algorithmique pourraient être réalisables, en particulier pour les réseaux neuronaux récurrents entraînés pour la classification des séquences.

Nous concluons par une discussion sur le potentiel ultime des réseaux de neurones pour résoudre la reconnaissance de l'écriture manuscrite et sur la contribution potentielle de l'information quantique à cet effort, comme un pas vers l'IA de niveau humain.

B.2 Amélioration d'une architecture de réseau neuronal convolutionnel profond pour la reconnaissance de caractères

Dans cette section, nous présentons la première contribution de cette thèse. Nous partons d'une architecture convolutionnelle profonde et décrivons l'effet des fonctions d'activation récentes (au moment où le travail a été effectué), des algorithmes d'optimisation et des procédures de régularisation appliquées à la reconnaissance des chiffres manuscrits de l'ensemble de données MNIST [132]. L'ensemble de données MNIST est populaire pour cette tâche et une variété d'approches ont été comparées en l'utilisant.

B.2.1 Architecture

L'architecture convolutionnelle que nous utilisons est illustrée dans la Figure B.1. Elle suit plusieurs des lignes directrices de [171], qui a obtenu la meilleure performance d'un modèle unique dans le concours de classification d'objets ILSVRC 2014 [163]. Chacune des couches convolutives possède des filtres de taille 3x3, avec stride 1 (le stride est la distance entre les centres des champs réceptifs des neurones voisins dans une carte d'activation). Les couches de sous-échantillonnage correspondent toujours à une opération de max pooling

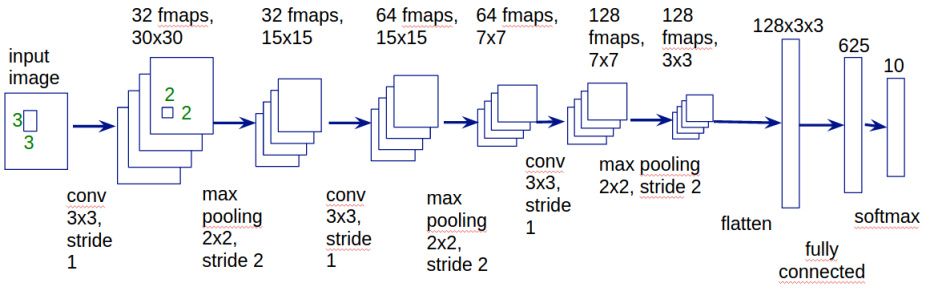


Figure B.1 – Achitecture du réseau convolutionnel

2x2 avec stride 2, ce qui permet de sous-échantillonner la hauteur et la largeur de l'image de 2. Lors du choix du nombre d'unités dans les couches convolutives, nous nous inspirons une fois de plus des directives décrites dans [171] : nous doublons le nombre d'unités entre chaque couche consécutive, de 32 dans la première couche convolutive à 64, puis 128 dans la dernière. L'architecture comprend également une couche entièrement connectée de 625 unités et une couche softmax de 10 classes (correspondant aux 10 chiffres). Nous montrons l'architecture dans le tableau B.1.

B.2.2 Détails d'implémentation

Nous avons essayé plusieurs fonctions d'activation non linéaires, telles que ReLU [86], LReLU [140] et PReLU [105]. Nous avons obtenu les meilleurs résultats en utilisant la fonction d'activation PReLU, en permettant à chaque filtre convolutionnel d'avoir sa propre valeur a, mais en liant les valeurs entre différents emplacements spatiaux d'un même filtre convolutif.

La fonction de perte que nous minimisons est la log-vraisemblance négative moyenne de la distribution conditionnelle de l'étiquette correcte, en fonction de l'entrée. Pour la procédure d'optimisation, nous avons utilisé ADAM, en conservant tous ses hyper-paramètres à leur valeur par défaut [125], sauf pour le taux d'apprentissage, qui a été diminué pendant l'entraînement. Nous obtenons notre meilleur système en utilisant un taux d'apprentissage initial de 0,005 et une décroissance du taux d'apprentissage de 0,98.

Nous avons obtenu les meilleurs résultats en utilisant la méthode d'initialisation décrite dans [105]. Pour simplifier, nous initialisons les paramètres PReLU a avec des valeurs nulles (juste après cette initialisation, les PReLU se comportent comme les ReLU). Nous initialisons les paramètres de normalisation de lot γ à 1, et les paramètres β à 0. Les matrices de poids des neurones sont initialisées en utilisant des valeurs tirées de distributions gaussiennes avec une moyenne 0 et un écart-type $\sqrt{\frac{2}{n_i}}$, où n_i est le nombre d'entrées du neurone, suivant la procédure dans [105], et les biais sont initialisés à 0.

Nous avons utilisé l'arrêt anticipé (early stopping) lors de l'exécution de l'optimisation:

Input size	**Convolutional Layer 1**
1 x 28 x 28	conv: 3 x 3 full, stride 1, 32 feature maps
32 x 30 x 30	batch normalization
32 x 30 x 30	PReLU
32 x 30 x 30	2 x 2 max pooling, stride 2
32 x 15 x 15	0.5 dropout
	Convolutional Layer 2
32 x 15 x 15	conv: 3 x 3, stride 1, 64 feature maps
64 x 15 x 15	batch normalization
64 x 15 x 15	PReLU
64 x 15 x 15	2 x 2 max pooling, stride 2
64 x 7 x 7	0.5 dropout
	Convolutional Layer 3
64 x 7 x 7	conv: 3 x 3, stride 1, 128 feature maps
128 x 7 x 7	batch normalization
128 x 7 x 7	PReLU
128 x 7 x 7	2 x 2 max pooling, stride 2
128 x 3 x 3	flatten
128 x 3 x 3	0.5 dropout
	Fully Connected Layer
128 x 3 x 3	fully connected layer matrix multiplication
625	batch normalization
625	PReLU
625	0.5 dropout
	Softmax Layer
10	softmax

Table B.1 – Architecture du réseau convolutionnel proposé

nous exécutons l'algorithme d'optimisation pour un maximum de 300 époques (utilisant des mini-lots) et nous nous arrêtons après 30 époques sans amélioration sur l'ensemble de validation.

B.2.3 Expériences

Nous avons effectué nos expériences sur le jeu de données MNIST, qui contient des images de taille 28 x 28 et est divisé en 3 ensembles distincts : un ensemble d'entraînement de 50000 images, un ensemble de validation de 10000 images et un ensemble de test de 10000 exemples. Comme c'est la norme dans la littérature ([134], [136]), le seul prétraitement que nous effectuons est de mettre à l'échelle les entrées à des valeurs [0, 1]. Nous avons également essayé de normaliser les entrées en utilisant la normalisation globale du contraste (global contrast normalization), mais nous n'avons pas obtenu de meilleurs résultats.

Comme régularisation, dans le meilleur système, nous avons utilisé dropout [178] avec probabilité d'enlever un neurone $p = 0.5$, tant dans la couche convolutive que dans la couche entièrement connectée (nous n'appliquons pas dropout sur les images en entrée).

Model	Test error (%)
Ours	**0.38**
Deeply-Supervised Nets [134]	0.39
Fractional max-pooling [90]	0.44
Maxout Networks [88]	0.45
Network in Network [136]	0.47

Table B.2 – Comparaison avec les résultats les plus récents sur l'ensemble de test MNIST (système unique, pas d'augmentation de données) au moment où ce travail a été effectué (Mai 2015)

Le taux d'erreur obtenu en utilisant notre système est indiqué dans le tableau B.2. Le même tableau montre également les résultats d'autres approches d'apprentissage profond qui permettent d'obtenir des résultats de pointe sur MNIST sans utiliser l'augmentation des données ni les ensembles de modèles. Les chiffres mal classés sont indiqués dans la figure B.2.

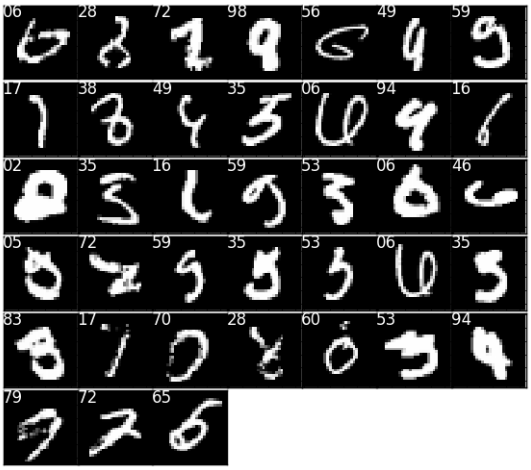


Figure B.2 – Tous les échantillons mal classés de l'ensemble de test MNIST. Le premier numéro est l'étiquette estimée, le second est la vérité-terrain.

Le réseau atteint un taux d'erreur de 0,38%, améliorant légèrement les meilleures performances connues d'un seul modèle entraîné sans augmentation de données au moment

où les expériences ont été réalisées, en mai 2015. En février 2016, la meilleure performance dans ce contexte était 0.24% [55].

B.2.4 Conclusions

Nous interprétons nos résultats comme un argument en faveur de systèmes qui peuvent effectuer l'extraction automatique (apprise) des caractéristiques, plutôt que d'utiliser des caractéristiques conçues à la main. L'utilisation d'une approche d'apprentissage profond permet également d'importer et d'appliquer facilement des innovations d'apprentissage profond d'autres domaines d'application et les contributions d'apprentissage profond pour l'écriture manuscrite peuvent également être utilisées dans d'autres domaines d'application.

Dans la section suivante, nous utilisons les réseaux convolutionnels comme éléments constitutifs d'architectures de réseaux de neurones plus complexes, qui peuvent effectuer à la fois la localisation et la classification.

B.3 Réseaux de Transformateurs Spatiaux Liés pour la reconnaissance de chiffres

Dans cette section, nous utilisons des Réseaux de Transformateurs Spatiaux Liés (RTSLs; Tied Spatial Transformer Networks - TSTNs, en anglais), des architectures d'apprentissage profond qui incluent deux réseaux convolutionnels couplés à l'aide d'un module Transformateur Spatial (Spatial Transformer, en anglais), qui a été introduit par [117]. Le premier réseau, qu'on va appeler réseau de *localisation*, est dédié aux transformations spatiales. Le deuxième, le réseau de *classification*, est dédié à la classification des images transformées. Les images d'entrée sont transformées à l'aide des paramètres estimés par le réseau de localisation, puis introduites dans le réseau de classification, qui effectue la prédiction de la classe.

L'avantage d'utiliser de tels systèmes couplés (localisation couplée à la classification) est que les paramètres entraînables du réseau de localisation sont appris pendant la phase d'entraînement, de même que ceux du réseau de classification, afin que les images puissent être transformées et classifiées de manière unifiée. Étant donné que la localisation et la classification peuvent être considérées comme des tâches différentes, les paramètres entraînables des deux réseaux seraient généralement différents. Nous proposons ici de lier les paramètres entraînables des deux réseaux afin d'obtenir un effet de régularisation qui peut améliorer la performance. Nous appliquons notre approche à la reconnaissance des chiffres bruités, en utilisant la base de données cluttered MNIST [146], dérivée de la base de données MNIST [132] mais incluant des sources supplémentaires de bruit.

B.3.1 Architectures convolutives

Tous les systèmes de classification présentés dans cette section sont basés sur les réseaux convolutionnels. On va comparer trois réseaux: un réseau convolutionnel classique de classification, le RTS et le RTSL. Le réseau convolutionnel comprend un seul bloc composé de couches convolutionnelles et de couches entièrement connectées, tandis que le RTS et le RTSL comprennent deux blocs, un pour la partie localisation et un autre pour la partie classification (voir les figures B.3, B.4 et B.5). Nous avons fait des expériences avec deux architectures différentes, la seconde étant plus complexe et plus puissante que la première. Ces architectures sont montrées dans le tableau B.3 et le tableau B.4 et suivent plusieurs des directives décrites dans [171]. Le réseau de classification utilisant la premiere architecture est montrée plus en détail dans la Figure B.6.

Figure B.3 – Architecture du réseau convolutionnel (CNN) de classification

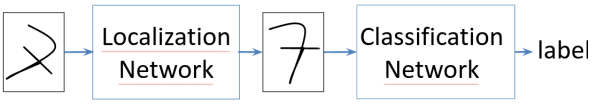


Figure B.4 – Architecture de type Réseau de Transformateur Spatial (RTS), composée de deux réseau convolutionnels (CNN) couplés, l'un dédié à la localisation, l'autre à la classification

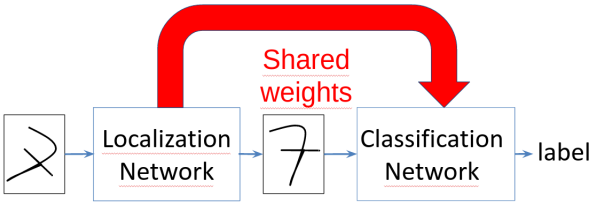


Figure B.5 – Architecture de type Réseau de Transformateur Spatial Lié (RTSL), similaire à l'architecture RTS (non lié), mais avec des poids partagés (liés)

Chacune des couches convolutionnelles contient des filtres de taille 3x3. La convolution de type 'same' - qui permet que la taille de la feature map de sortie soit égale à la taille de l'entrée - est utilisée dans toutes les couches. Les couches de sous-échantillonnage sont toujours 2x2 max-pooling avec stride 2, ce qui fait que les images soient sous-échantillonnées

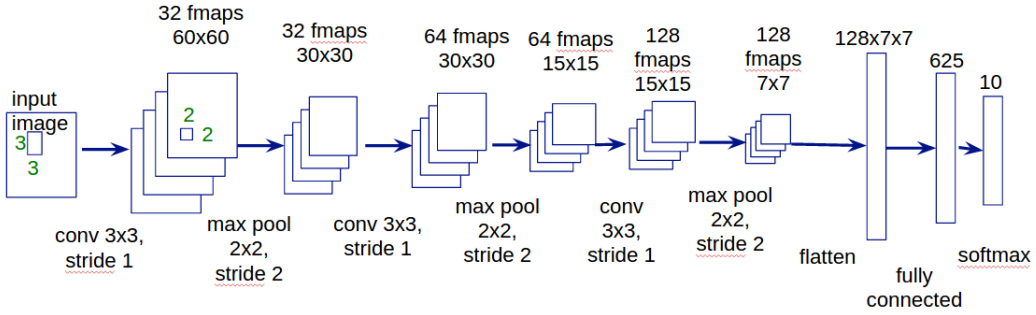


Figure B.6 – Architecture détaillée du réseau convolutionnel de classification 1 (fmap = feature map = carte des caractéristiques)

en divisant à la fois hauteur et largeur par 2. Nous doublons le nombre de feature maps entre chaque couche convolutionnelle consécutive, de 32 dans la première couche à 64 puis 128 dans la dernière, en nous inspirant une fois encore de [171].

Les couches entièrement connectées sont identiques entre les architectures 1 et 2, mais différentes entre les réseaux de localisation et de classification. Les couches convolutives sont également très similaires. La différence est que pour l'architecture 2, nous utilisons deux opérations consécutives de convolution suivies d'une non-linéarité au lieu d'une seule avant chaque max-pooling. Cela rend l'architecture 2 plus puissante et permet également d'augmenter la taille implicite de ses filtres convolutifs. Dans les deux architectures, nous supprimons l'opération de pooling dans la dernière couche convolutionnelle du réseau de localisation. La motivation de ce choix architectural est de conserver plus d'informations spatiales dont la couche entièrement connectée du réseau de localisation pourrait avoir besoin afin de fournir les paramètres estimés de la transformation affine effectuée par le module Transformateur Spatial. Notez également que puisque les couches max-pooling ne contiennent pas de paramètres entraînables, cela signifie que les couches convolutives des réseaux de classification et de localisation peuvent toujours avoir les mêmes valeurs pour leurs paramètres entraînables (liés). Pour les deux architectures (1 et 2), la dernière couche du réseau localisation contient moins d'unités que celle du réseau de classification, imitant la conception des architectures utilisées dans [176] et [174].

Généralement, la transformation effectuée sur l'image d'entrée à l'aide du Transformateur Spatial peut inclure la mise à l'échelle, le recadrage, les rotations, ainsi que les déformations non rigides [117]. Dans toutes nos architectures, nous n'utilisons que le Transformateur Spatial affine, introduit dans [117] et également utilisé dans [176]. Il permet toutes les transformations mentionnées précédemment, à l'exception des déformations non rigides, et ne nécessite que 6 paramètres à estimer dans les sorties du réseau de localisation. L'implémentation que nous utilisons, introduite et décrite dans [176], inclut également un facteur de sous-échantillonnage d, par lequel la hauteur et la largeur de l'image sont

Localization CNN	**Classification CNN**
Convolutional Layer 1	**Convolutional Layer 1**
conv: 3 x 3 same, stride 1, 32 feature maps	conv: 3 x 3 same, stride 1, 32 feature maps
ReLU	ReLU
2 x 2 max pooling, stride 2	2 x 2 max pooling, stride 2
Convolutional Layer 2	**Convolutional Layer 2**
conv: 3 x 3 same, stride 1, 64 feature maps	conv: 3 x 3 same, stride 1, 64 feature maps
ReLU	ReLU
2 x 2 max pooling, stride 2	2 x 2 max pooling, stride 2
Convolutional Layer 3	**Convolutional Layer 3**
conv: 3 x 3 same, stride 1, 128 feature maps	conv: 3 x 3 same, stride 1, 128 feature maps
ReLU	ReLU
	2 x 2 max pooling, stride 2
flatten	flatten
Fully Connected Layer	**Fully Connected Layer**
200 neurons linear layer	625 neurons linear layer
ReLU	ReLU
	0.5 dropout
Affine Transform Layer	**Softmax Layer**
6 parameters	10-class softmax

Table B.3 – Architecture numéro 1: réseau convolutionnel de localisation (à gauche) et réseau convolutionnel de classification (à droite)

mises à l'échelle, après l'utilisation de la transformation affine. Par souci de simplicité et pour faciliter les comparaisons entre les différents modèles (y compris entre CNN et STN), nous utilisons $d = 1$ dans toutes nos expériences avec des systèmes contenant des Transformateurs Spatiaux. En effectuant des transformations affines, les systèmes intégrant le module Spatial Transformer peuvent à la fois sélectionner des parties pertinentes d'une image (une forme d'attention) et transformer ces régions en poses qui peuvent simplifier la tâche de reconnaissance pour les composants suivants du système. Pour tous les détails techniques en ce qui concerne les Transformateurs Spatiaux, nous renvoyons le lecteur à la publication correspondante [117].

B.3.2 Details d'implementation

Pour la fonction d'activation, nous utilisons dans toutes nos expériences l'unité linéaire rectifiée (ReLU) [86], l'une des fonctions d'activation les plus réussies en apprentissage profond.

Localization CNN	**Classification CNN**
Convolutional Layer 1	**Convolutional Layer 1**
conv: 3 x 3 same, stride 1, 32 feature maps	conv: 3 x 3 same, stride 1, 32 feature maps
ReLU	ReLU
conv: 3 x 3 same, stride 1, 32 feature maps	conv: 3 x 3 same, stride 1, 32 feature maps
ReLU	ReLU
2 x 2 max pooling, stride 2	2 x 2 max pooling, stride 2
Convolutional Layer 2	**Convolutional Layer 2**
conv: 3 x 3 same, stride 1, 64 feature maps	conv: 3 x 3 same, stride 1, 64 feature maps
ReLU	ReLU
conv: 3 x 3 same, stride 1, 64 feature maps	conv: 3 x 3 same, stride 1, 64 feature maps
ReLU	ReLU
2 x 2 max pooling, stride 2	2 x 2 max pooling, stride 2
Convolutional Layer 3	**Convolutional Layer 3**
conv: 3 x 3 same, stride 1, 128 feature maps	conv: 3 x 3 same, stride 1, 128 feature maps
ReLU	ReLU
conv: 3 x 3 same, stride 1, 128 feature maps	conv: 3 x 3 same, stride 1, 128 feature maps
ReLU	ReLU
	2 x 2 max pooling, stride 2
flatten	flatten
Fully Connected Layer	**Fully Connected Layer**
200 neurons linear layer	625 neurons linear layer
ReLU	ReLU
	0.5 dropout
Affine Transform Layer	**Softmax Layer**
6 parameters	10-class softmax

Table B.4 – Architecture numéro 2: réseau convolutionnel de localisation (à gauche) et réseau convolutionnel de classification (à droite)

Pour simplifier, nous initialisons toutes les matrices de poids en utilisant une méthode similaire à celle proposée dans [105] et qui est également utilisée dans [174]. Les matrices de poids sont initialisées à l'aide de valeurs tirées de distributions uniformes avec une moyenne 0 et un écart-type $\sqrt{\frac{2}{n_i}}$, où n_i est le nombre d'entrées dans le neurone. La procédure, présentée en détail dans [105], permet d'éviter les instabilités de gradient pendant la phase d'entraînement, ainsi que les instabilités des entrées de chaque couche pendant l'inférence. Les paramètres de la couche de transformation affine sont initialisés à la transformation d'identité, comme dans [174].

Chacun des systèmes que nous expérimentons (réseau convolutionnel de classification, RTS, RTSL, pour chacune des deux architectures) représente une fonction différentiable. Pour optimiser chacune de ces fonctions, nous utilisons des méthodes d'optimisation basées sur les gradients, les gradients étant calculés à partir de mini-lots d'exemples, en utilisant l'algorithme de rétropropagation.

Nous utilisons comme algorithme d'optimisation une variante d'ADAM [125], dans laquelle nous multiplions le taux d'apprentissage par un montant fixe après un certain nombre d'époques. L'utilisation d'un bon optimiseur peut être importante pour les systèmes RTS et RTSL, car chacun d'entre eux est environ deux fois plus profond que les réseaux de classification correspondants, ce qui peut entraîner la disparition de gradients en raison de la profondeur accrue.

ADAM introduit quelques hyper-paramètres supplémentaires, décrits en détail dans [125], que nous avons maintenu fixes, en utilisant les valeurs recommandées dans [125]. Le seul paramètre que nous modifions est le taux d'apprentissage, que nous multiplions par un montant fixe après un nombre fixe d'époques. Nous avons suivi l'implémentation fournie par [174] et multiplié le taux d'apprentissage par 0,7 après chaque 20 époques. Nous utilisons des mini-lots de taille 256 dans toutes les procédures d'entraînement.

Pour réduire le risque de surapprentissage, nous utilisons dropout. Nous utilisons dropout avec $p = 0.5$ uniquement dans la couche entièrement connectée du réseau de classification, car c'est la couche avec le plus grand nombre de paramètres entraînables, où la régularisation est potentiellement la plus utile. Nous n'utilisons pas de dropout dans la couche entièrement connectée du réseau de localisation. L'intuition derrière ce choix est que dropout, en tant que forme bruyante de régularisation, tend à supprimer une partie de l'information spatiale présente dans les activations de l'unité. Cette information spatiale peut être plus utile pour la tâche de localisation (par rapport à celle de la classification), nous préférons donc en préserver une plus grande partie (au risque de surapprentissage).

B.3.3 Expériences

Nous avons appliqué les RTSLs à la reconnaissance des chiffres manuscrits bruités. Les expériences sont menées sur la base de données cluttered MNIST [173] [146] qui est dérivée

de la base de données MNIST [132] mais inclut des bruits et transformations supplémentaires qui rendent la tâche de classification plus difficile. Les chiffres sont déformés à l'aide de la translation, de l'échelle, de la rotation et de l'encombrement aléatoires, et l'image entière correspondant à chaque chiffre est plus grande (60 x 60, comparé à 28 x 28 pour MNIST). Le nombre d'exemples d'apprentissage / de validation / de test reproduit celles de MNIST (50000 / 10000 / 10000). Nous montrons des échantillons d'image agrandies provenant de cluttered MNIST et le traitement effectué par un Transformateur Spatial dans la figure B.7.

Pour notre implémentation, nous utilisons Theano [186] et Lasagne [75] et nous nous appuyons fortement sur l'implémentation de [175] et sur l'exemple de RTS de [174].

Figure B.7 – Images d'entrée de cluttered MNIST (à gauche) et les mêmes images traitées par un module Transformateur Spatial dans le cadre d'un système RTS1 (à droite)

B.3.3.1 Comparaison des réseaux convolutionnels de classification, des RTSs et des RTSLs

Nous comparons dans le tableau B.5 le réseau RTSL proposé (paramètres entraînables liés) avec le RTS (paramètres entraînables non liés) et le réseau convolutionnel de classification (seul), en utilisant les architectures 1 et 2 sur le jeu de données cluttered MNIST.

Nous avons arrêté l'entraînement de chaque système manuellement, après avoir vérifié que la précision de la classification sur les données d'entraînement est (presque) parfaite. Dans un seul cas, l'architecture RTS non liée 2 (entrée 8 dans le tableau B.5), nous n'avons pas réussi à faire converger l'entraînement pour qu'elle fonctionne de manière comparable aux autres systèmes. Pour souligner que le nombre d'époques d'entraînement est comparable et que nous avons permis à chaque système un entraînement raisonnable

pour atteindre une bonne performance, nous montrons également dans le tableau B.5 le nombre d'époques d'entraînement et la meilleure précision d'entraînement (obtenue jusqu'à l'époque d'entraînement maximale). Pour chaque système et pour chaque procédure d'entraînement, nous choisissons les paramètres pour lesquels nous obtenons la meilleure précision de validation avant de faire l'évaluation sur l'ensemble de test. Une seule expérience est effectuée pour chaque système, avec la même random seed utilisée dans toutes les expériences.

Comme on peut le voir dans le tableau B.5, les systèmes RTSL obtiennent de meilleurs résultats que les réseaux convolutionnels de classification et les RTS (non liés), pour la même architecture (en comparant les entrées 1-3 et 7-9, respectivement).

Entry	Maximum training epochs	Best training error (%)	Validation error (%)	Test error (%)
1. CNN1	145	0	4.39	4.44
2. STN1	179	0	4.38	4.33
3. TSTN1	196	0.002	3.02	3.14
4. TSTN1-2	127	0.032	4.14	4.25
5. TSTN1-5-20	135	0	3.11	3.15
6. TSTN1-2-20	127	0.002	2.73	2.74
7. CNN2	95	0	1.82	2.18
8. STN2	66	88.64	88.74	88.81
9. TSTN2	231	0	1.30	1.74
10. TSTN2-5-20	141	0	1.24	1.50
11. TSTN2-2-20	178	0	1.22	1.31

Table B.5 – Résultats sur la base de données cluttered MNIST pour différents systèmes, architectures et procédures d'entraînement

B.3.3.2 L'hypothèse de la régularisation

Une hypothèse que nous proposons pour la meilleure performance du RTSL est que, par rapport au RTS (non lié), il est mieux régularisé. Pour tester cette hypothèse, nous avons mené une série d'expériences où nous entraînons chacun des deux sous-systemès RTSL à l'aide d'une approche curriculum: nous alternons entre la formation de l'ensemble du RTSL à certaines époques et la formation de sa seule composante de classification à des époques différentes. Nous testons l'effet de régularisation sur les paramètres entraînables liés du RTSL en utilisant uniquement sa composante de classification pendant les phases de validation et de test. Les résultats de ces expériences sont présentés aux entrées 4-6, 10 et 11 du tableau B.5.

La première expérience de ce type est réalisée en utilisant l'architecture 1, en entraînant l'ensemble du RTSL pendant les époques impaires et en n'entraînant que sa composante de classification pendant les époques paires, montrée à l'entrée 4 (TSTN1-2) du tableau B.5.

Nous avons émis l'hypothèse que les performances pourraient être améliorées si nous pouvions prévoir des périodes d'entraînement initial pour l'ensemble du RTSL avant d'alterner entre l'entraînement du RTSL et de son sous-système de classification. Puisque l'architecture RTSL est deux fois plus profonde que son sous-système de classification, le problème des gradients qui disparaissent ou explosent (vanishing / exploding gradients) est potentiellement pire; ceci est particulièrement vrai pour les paramètres du sous-système de localisation, qui est le plus éloigné du signal d'erreur fourni lors de la rétropropagation. Pour tester cette hypothèse, nous effectuons la procédure d'entraînement selon le schéma suivant: l'ensemble du RTSL est entraîné pour les 20 premières époques, puis nous alternons l'entraînement de l'ensemble du RTSL toutes les 5 époques et l'entraînement de son sous-système de classification pendant les autres époques. Cela correspond au système RTSL1-5-20 (entrée 5) du tableau B.5; la performance semble être améliorée pour l'architecture 1, devenant similaire à celle du RTSL classique.

L'effet de la fréquence de l'entraînement de l'ensemble du système RTSL, tout en n'utilisant que le réseau de classification pendant les phases de validation / test pourrait également être lié à l'arbitrage entre la pondération d'un terme de régularisation et la fonction de perte. Un entraînement plus fréquent de l'ensemble du RTSL correspondrait à utiliser plus de réguliarisation. De cette manière, nous pouvons compléter la régularisation fournie en utilisant les approches décrites dans les deux paragraphes précédents en entraînant le RTSL entier pendant les 20 premières époques, puis en alternant entre l'entraînement du RTSL entier et de son sous-système de classification toutes les 2 époques. Les résultats (RTSL1-2-20) pour l'architecture 1, présentés à l'entrée 6, semblent être encore meilleurs que pour le RTSL.

Nous avons également testé les deux dernières approches en utilisant l'architecture 2 (entrées 10 et 11). Les mêmes conclusions semblent s'appliquer.

B.3.4 Discussion

Dans cette section, nous présentons plusieurs perspectives pour interpréter les résultats de nos expériences avec les RTSLs.

Tout d'abord, les paramètres appris par le réseau de localisation peuvent être interprétés comme codant la connaissance de la transformation des images d'entrée (les transformations étant conditionnées sur chaque image individuelle) afin de simplifier la tâche de reconnaissance pour les couches suivantes du système entier. En liant les paramètres des réseaux de classification et de localisation, le RTSL est contraint de devenir bon aux deux tâches (transformation de l'image d'entrée et classification de l'image transformée).

L'utilisation du réseau de localisation peut être vue comme l'ajout d'un objectif auxiliaire, différent de celui qui nous intéresse finalement (l'entropie croisée), mais qui pourrait aider à améliorer l'erreur de classification [134].

B.3.5 Conclusion

Dans cette section, nous avons décrit une nouvelle manière d'utiliser les architectures de Réseaux de Transformateurs Spatiaux, en liant les paramètres entraînables entre les réseaux de localisation et de classification, et de les appliquer à la reconnaissance des chiffres manuscrits bruités. Dans la section suivante, nous abordons un problème plus complexe, la reconnaissance des mots manuscrits, en utilisant une architecture de réseau de neurones récurrent récemment proposée.

B.4 Associative LSTMs pour la reconnaissance de l'écriture manuscrite

Dans cette section, nous présentons les résultats de la reconnaissance des séquences d'écriture arabe, comparant le réseau récurrent Associative Long Short-Term Memory (ALSTM) récemment proposé [74] à un LSTM de référence [112].

Notre motivation est d'étudier les avantages potentiels de l'ajout d'une capacité mémoire supplémentaire dans les réseaux récurrents sans augmenter significativement le nombre de paramètres entraînables pour la reconnaissance de l'écriture arabe. [74] motive également l'introduction des ALSTMs comme améliorant (augmentant) les LSTMs avec mémoire supplémentaire. La mémoire supplémentaire est ajoutée en créant des copies redondantes des informations stockées dans les cellules ALSTM et en utilisant des mécanismes de stockage et de récupération des données à valeur clé. Les copies redondantes permettent de réduire le bruit lorsque le processus de récupération est effectué.

Dans notre étude sur l'application des ALSTMs à la reconnaissance de l'écriture manuscrite, nous suivrons une méthodologie similaire à celle de [202], qui a montré qu'un LSTM bidimensionnel (Bidimensional LSTM - BLSTM) peut être plus précis que des LSTM multidimensionnels, si les images saisies sont normalisées (en apprenant directement des pixels). Cette approche améliore également les méthodes qui utilisent des caractéristiques artisanales. Tous les systèmes susmentionnés ont été comparés en utilisant l'ensemble de données des mots arabes manuscrits IFN / ENIT [22].

Nous utiliserons le même jeu de données et la même méthode de normalisation pour comparer les ALSTMs et les LSTMs et nous comparerons des configurations multiples des deux types d'architectures : monocouche, multicouche, unidirectionnelle, bidirectionnelle (et combinaisons, par exemple multicouche bidirectionnelle).

B.4.1 Ensemble de données

On utilise le jeu de données IFN-ENIT, qui contient 32492 images de noms de villes et villages tunisiens, écrits à la main par plus de 400 scripteurs dans une grande variété de styles. Il est divisé en cinq ensembles, de A à E. Dans le cas le plus courant, que nous allons également suivre, les jeux A-D (contenant 26459 images) sont utilisés pour l'entraînement et le jeu E (contenant 6033 images) est utilisé pour les tests.

Les images varient à la fois en hauteur et en largeur et varient de 85×84 à 162×1069 dans le set d'entraînement et de 40×170 à 139×977 dans le set de test. Toutes les images sont en niveaux de gris.

B.4.2 Normalisation d'image

Nous effectuons d'abord le seuillage d'Otsu [151] en utilisant l'implémentation OpenCV [27]. Nous utilisons ensuite la procédure de normalisation d'image fournie par ocropy [26], qui effectue la mise au point (dewarping) et la normalisation de la taille d'image à la hauteur par défaut de 48 pixels (avec une largeur redimensionnée proportionnellement, afin de préserver le rapport hauteur / largeur). Nous effectuons ensuite une inversion de contraste sur l'image résultante et, enfin, normalisons les valeurs des pixels à [0, 1].

B.4.3 Détails des systèmes

Nous initialisons les poids récurrents pour tous les systèmes en utilisant l'initialisateur orthogonal [166]. Pour tous les systèmes, nous avons initialisé les poids de la couche de sortie en utilisant l'initialisation Xavier et les biais à 0 [85].

La perte optimisée est la perte CTC [95] et tous les résultats présentés sont obtenus en utilisant un décodage CTC gourmand (greedy) [95].

Pour l'optimisation, nous avons utilisé ADAM [125] avec les paramètres par défaut (y compris le taux d'apprentissage par défaut de 0.001). Nous appliquons également l'écrêtage de gradient par norme globale, avec la norme globale 10. Nous avons utilisé la taille de lot 64 pour toutes les expériences.

Nous présentons également des résultats pour les architectures bidirectionnelles. L'extension aux LSTM bidirectionnelles et aux ALSTM bidirectionnelles est simple.

B.4.4 Résultats

Tous les résultats présentés sont obtenus sur l'ensemble de validation E, avec l'entraînement effectué sur les ensembles A-D.

Nos LSTM associatives utilisent une seule copie, puisque les premières expériences avec des copies multiples n'ont montré aucun gain.

Parce que le LSTM est environ 4-5 fois plus rapide que l'ALSTM associatif nous l'entraînons pendant plus d'époques d'entraînement (150) que l'ALSTM (100). Dans toutes nos expériences avec les ALSTMs, W_{hu} a été mis à 0 (voir [74] pour plus de détails).

System	CER	Number of trainable parameters	Number of training epochs
LSTM-128	0.5451	106233	150
LSTM-256	0.5200	343417	150
ALSTM-64	0.5314	40121	100
BLSTM-64	0.3545	73465	150
BALSTM-64	0.3598	80121	100
BLSTM-128	0.3197	212345	150
BALSTM-128	0.2919	233849	100

Table B.6 – Taux d'erreur de caractères (Character Error Rate - CER) pour divers systèmes LSTM et ALSTM. Nous désignons les RNNs LSTM avec n unités par LSTM-[n] et les ALSTMs avec n unités par ALSTM-[n]. Nous ajoutons le préfixe B pour les RNNs bidirectionnels.

Nous affichons les résultats pour différents réseaux LSTM et ALSTM dans le tableau B.6. Par LSTM-[n], nous désignons les LSTMs avec n unités et de façon analogue pour le ALSTM. Pour les RNN bidirectionnels, nous ajoutons le préfixe B.

Nous pouvons voir que l'ALSTM-64 fonctionne mieux que le LSTM-128 avec environ 2,5 fois moins de paramètres entraînables, et un peu moins bien que le LSTM-256, qui contient environ 8,5 fois plus de paramètres entraînables.

Le BALSTM-128 est plus performant que le BLSTM-128, tandis que le BALSTM-64 ne semble pas faire mieux que le BLSTM-64. Dans les deux cas, les systèmes BALSTM et BLSTM ont un nombre comparable de paramètres entraînables.

Dans notre implémentation, l'ALSTM semble environ 4 à 5 fois plus lent qu'un LSTM, pour un nombre égal d'unités. Ce ralentissement est similaire à celui signalé dans [74]. Cette diminution de l'efficacité des calculs annule en partie les gains de précision et nous incite à essayer d'autres approches pour augmenter la mémoire des réseaux récurrents, sans augmenter le coût de calcul, comme nous le verrons dans la section suivante.

B.5 Apprentissage profond hybride classique-quantique

Dans cette section, nous proposons une paramétrisation hybride classique-quantique des réseaux de neurones, qui s'inspire des travaux précédemment proposés sur les hyper-réseaux

(hypernetworks) [100]. L'idée des hypernetworks est qu'un réseau neuronal, que nous appellerons *réseau hyper* (hyper network), peut être utilisé pour générer les paramètres d'un autre réseau, que nous désignerons comme *réseau principal* (main network), qui est utilisé pour résoudre la tâche (par exemple, classifier). Nous instancions cette proposition en utilisant des réseaux neuronaux récurrents (recurrent neural networks - RNNs). Plus spécifiquement, nous proposons qu'un réseau neuronal récurrent classique (implémenté sur un ordinateur classique) génère les paramètres d'un réseau neuronal récurrent quantique (implémenté sur un ordinateur quantique), qui résoudra la tâche. Les réseaux neuronaux récurrents sont Turing-complets [115] [170], donc, dans un certain sens, des modèles de calcul d'apprentissage profond entièrement généraux. L'adressage des réseaux neuronaux récurrents différencie notre travail de la plupart des propositions de réseaux neuronaux quantiques précédentes et l'utilisation des réseaux quantiques différencie notre travail des hypernetworks classiques. En principe, notre proposition est générale et peut couvrir plus de types d'architectures que juste les réseaux neuronaux récurrents.

L'intuition derrière notre proposition est que les ordinateurs quantiques peuvent nous permettre d'effectuer certaines opérations dans des espaces de très haute dimension avec un nombre de calculs exponentiellement inférieur à celui des ordinateurs classiques. Dans le cas des RNNs, cette dimensionnalité élevée pourrait permettre davantage de mémoire, de sorte que le RNN quantique se souvienne des entrées du passé lointain avec des gains de calcul significatifs par rapport aux propositions classiques. L'hyper-RNN dans notre proposition peut être interprété comme *apprenant à coder* les entrées dans l'espace de grande dimension du RNN quantique.

Pour simplifier le problème de la génération de sorties à partir de l'ordinateur quantique, dans ce travail, nous ne traiterons que des problèmes many-to-one [135], dans lesquels l'entrée est présentée sous forme de séquence et une seule sortie est requise. Nous montrerons des résultats suggérant que notre RNN hybride classique-quantique proposé peut fonctionner de la même façon qu'un RNN classique dans une tâche de dépendance séquentielle à long terme, avec un besoin de calcul potentiellement exponentiellement moindre. Pour cela, nous choisirons une tâche similaire à la tâche MNIST à pixels permutés, un repère standard pour les capacités mémoire à long terme des architectures RNN [121] [38], mais en utilisant le jeu de données Fashion-MNIST [198] [199]. Nous simulons notre proposition sur carte graphique (Graphics Processing Unit - GPU), mais montrons les coûts de calcul approximés requis si notre proposition était exécutée sur un ordinateur quantique.

Au meilleur de notre connaissance, nos travaux sont les premiers à montrer des résultats prometteurs suggérant un avantage quantique potentiel pour les RNNs sur un ensemble de données à grande échelle (à échelle industrielle).

B.5.1 Architecture proposée

Dans cette section, nous allons décrire le paramétrage de notre RNN hybride classique-quantique. Nous décrirons d'abord la composante quantique, puis la couche de sortie, la fonction de perte et, enfin, le RNN classique qui génère les paramètres du RNN quantique. Une illustration simplifiée de notre proposition est présentée dans la Fig. B.8.

B.5.1.1 RNN quantum (principal)

Nous indiquerons l'état de notre RNN principal (quantique) au temps t par h_t. L'état initial du RNN principal h_0 est la superposition uniforme. Cette superposition uniforme peut être obtenue à partir de l'état n-qubit $|0..0\rangle$ avec un coût de calcul $O(n)$ en utilisant n portes de Hadamard. A chaque pas de temps t, l'état h_t de notre RNN quantique est obtenu en multipliant son état précédent h_{t-1} par la matrice U_t, qui est générée par l'état h'_t du RNN classique:

$$h_{t+1} = U_t * h_t \tag{B.1}$$

Nous décomposons la matrice U_t avec le paramétrage suivant:

$$U_t = A_t * F * D_t * F^{-1} \tag{B.2}$$

où F est la transformée de Fourier quantique (Quantum Fourier Transform - QFT) pour n qubits, F^{-1} est la transformée de Fourier quantique inverse (Inverse Quantum Fourier Transform - IQFT) pour n qubits et A_t et D_t sont des matrices unitaires paramétrées (entraînables), avec leurs paramètres générés par un hyper-RNN (la QFT et la IQFT ne contiennent pas des paramètres entraînables). Plus précisément, A_t et D_t sont des produits tensoriels de transformées unitaires générales à un qubit. Ces transformées unitaires pour un qubit sont des matrices unitaires 2 x 2 générales [35], de sorte que A_t et D_t peuvent être écrites comme des produits tensoriels de n matrices 2×2.

Une matrice unitaire générale 2×2 A peut s'écrire comme suit:

$$A = e^{\frac{i\alpha}{2}} \begin{bmatrix} e^{i\beta}\cos(\phi) & e^{i\gamma}\sin(\phi) \\ -e^{-i\gamma}\sin(\phi) & e^{-i\beta}\cos(\phi) \end{bmatrix} \tag{B.3}$$

où α, β, γ et ϕ sont des paramètres modifiables. Nous indiquerons par θ_t l'ensemble des paramètres modifiables (entraînables) utilisés au temps t pour générer A_t et D_t.

Le QFT et l'IQFT peuvent être implémentés avec un coût de calcul de $O(n\log(n))$ [101]. Les matrices unitaires générales sur un qubit peuvent être implémentées en utilisant 4 portes élémentaires consécutives sur un qubit [52] [150]. Ainsi, A_t et D_t sont de taille $2^n \times 2^n$, comprennent uniquement $4 * n$ paramètres entraînables et leurs complexités de calcul respectives sont $O(n)$. Nous utiliserons ces complexités lorsque nous approximerons

le coût de calcul de notre RNN hybride classique-quantique s'il était implementé sur ordinateur quantique.

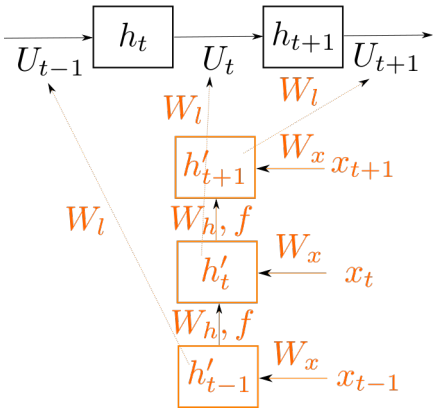


Figure B.8 – Exemple d'un réseau neuronal récurrent hybride classique-quantique similaire à ceux que nous simulons. Un hypernetwork récurrent (en orange) implémenté sur un ordinateur classique génère dynamiquement des transformations unitaires pour un réseau principal (en noir), qui serait implémenté sur un ordinateur quantique (mais qu'on simule classiquement). Les matrices unitaires U_t sont générées dynamiquement par le réseau classique pour chaque exemple (séquence) à partir des activations du réseau classiques h'_t, en utilisant la matrice entraînable W_l. Aucune non-linéarité n'est utilisée dans le réseau neuronal quantique. Nous illustrons l'hypernetwork comme un RNN typique, avec la même matrice entraînable de poids W_h appliquée à chaque pas de temps, suivie de la non-linéarité par élément f, mais, en principe, toute architecture pourrait être utilisée. Les entrées x_t ne sont fournies qu'à l'hyper-RNN, traitées par la matrice entraînable W_x. La figure est simplifiée à des fins d'illustration. Dans nos expériences, un LSTM est utilisé comme hyper-RNN.

B.5.1.2 Couche de sortie

Pour simplifier et pour relier cette partie à nos simulations qui seront décrites dans la section B.5.2, nous supposerons que notre RNN hybride classique-quantique est utilisé pour résoudre une tâche séquentielle many-to-one (T entrées, une sortie) avec 2 classes de sortie (classification binaire): 0 et 1. Nous indiquons la probabilité de prédire la classe 0 comme p_0 et la probabilité de prédire la classe 1 comme p_1.

Pour calculer la probabilité p_1 de la classe 1, nous calculons le produit scalaire complexe entre le dernier état du RNN principal (quantique) h_T et un v complexe de norme 1 (ce vecteur v contient des paramètres entraînables et sa norme est fixée à la valeur 1 pendant tout l'entraînement en renormalisant v après chaque mise à jour de ses paramètres entraînables). Nous soustrayons ensuite de 1 la valeur absolue du nombre complexe résultant (nous aurions aussi pu utiliser directement la valeur absolue $|h_T \cdot \frac{v}{||v||}|$, mais nous avons

constaté que la procédure décrite précédemment fonctionnait mieux de manière empirique, de la même manière que [161]):

$$p_1 = 1 - |h_T \cdot \frac{v}{||v||}| \tag{B.4}$$

h_T étant l'état du réseau quantique au pas de temps final T. Le produit scalaire des vecteurs unitaires complexes que nous utilisons peut être obtenu en utilisant la distance euclidienne de tels vecteurs [196]. La distance euclidienne peut être implémenté à l'aide de la RAM quantique (QRAM - Quantum Random Access Memory), comme décrit dans [139], avec un coût de calcul $O(n)$ pour n qubits. Comme décrit dans [139] [150], p_1 peut être estimé avec précision ϵ en utilisant l'algorithme de comptage quantique, avec une complexité de calcul $O(\frac{1}{\epsilon})$. La probabilité de la classe 0 est $p_0 = 1 - p_1$.

B.5.1.3 Fonction de perte

La perte que nous avons utilisée est une perte de marge maximale similaire à celle utilisée dans [164]:

$$L = y * \max(0, m^+ - p_1)^2 + (1 - y) * \max(0, p_1 - m^-)^2 \tag{B.5}$$

où y est l'étiquette de la vérité terrain et m^+ et m^- sont des hyperparamètres.

L'intuition derrière cette expression est que, dans le cas de l'étiquette vérité-terrain 1, nous ne pénalisons la probabilité p_1 que si elle est inférieure à m^+ (un argument similaire s'applique pour p_0 et l'étiquette vérité-terrain 0). Cela correspond à la maximisation de la différence absolue (marge) entre les probabilités de prédiction pour les deux classes, $|p_1 - p_0|$. Intuitivement, nous voulons que les prédictions de notre classificateur soient aussi fiables que possible, afin qu'elles soient moins affectées par le bruit de l'échantillonnage de précision ϵ obtenu en utilisant l'algorithme de comptage quantique. Un classificateur avec des prédictions plus sûres nous permettrait de choisir des valeurs plus élevées pour ϵ et, par conséquent, nous aurions besoin de répéter la mesure quantique moins souvent, ce qui réduirait les coûts de calcul.

B.5.1.4 Hyper RNN classique

Nous utilisons comme RNN classique (hyper-réseau) une architecture LSTM à couche unique [112], sans aucune connexion de type peephole, pour générer les paramètres modifiables du RNN quantique. Nous notons l'état caché de l'hyper-réseau (classique) au temps t par h'_t et les données d'entrée au temps t par x_t. Nous utilisons la même architecture comme base de référence pour les architectures RNN classiques standard auxquelles nous comparons notre proposition.

Une nouvelle matrice unitaire U_t paramétrée par l'eq. B.1 est générée à chaque pas de temps t, avec les paramètres générés à partir de l'état de l'hyper-réseau classique h'_t.

Les paramètres modifiables du RNN quantique au temps t, θ_t (utilisés pour construire A_t et D_t), sont obtenus à partir de l'état h'_t en utilisant une couche linéaire (composée de la matrice W_l et des biais b_l):

$$\theta_t = W_l * h'_t + b_l \tag{B.6}$$

Nous dénoterons l'application de la matrice W_l (et des biais b_l) comme *la transition état-vers-paramètres-de-sortie*.

B.5.2 Résultats de la simulation

Dans cette section, nous allons décrire les expériences que nous avons effectuées. On va commencer avec le raisonnement derrière nos choix de tâche et de jeu de données, suivi par des descriptions détaillées des systèmes mettant en oeuvre notre proposition et le LSTM de base. Nous présentons ensuite le calcul des coûts de calcul approximés de ces systèmes et l'estimation de la précision de classification sous approximation de l'échantillonnage de précision ϵ.

B.5.2.1 Tâche et ensemble de données

Nous sommes intéressés à montrer que notre RNN hybride classique-quantique proposé peut fonctionner de la même façon qu'un RNN classique dans les tâches de dépendance séquentielle à long terme, avec un besoin de calcul potentiellement exponentiellement moindre. Pour cela, nous choisirons une tâche similaire à la tâche pixels permutés de MNIST, un benchmark standard pour les capacités mémoire à long terme des architectures RNN [121] [38].

Nous afficherons des résultats sur l'ensemble de données Fashion-MNIST [198] [199]. Le choix de cet ensemble de données est motivé par le fait qu'il est similaire à l'ensemble de données bien connu du MNIST (mêmes dimensions d'entrée, et mêmes nombres d'exemples d'entraînement, de validation et de test), mais il est plus difficile. La tâche que nous considérons consiste d'abord à permuter les pixels d'entrée des images (en utilisant la même permutation pour l'ensemble entier des données). Les pixels sont ensuite entrés un par un et le système doit classer la séquence résultante. Cette tâche (Fashion-MNIST) est la même que la tâche pixels permutés MNIST (mais utilise un autre ensemble de données). Afin de réduire le temps d'entraînement (en particulier pour la simulation des systèmes hybrides classiques-quantiques), nous effectuerons également des simulations où nous réduirons la taille de l'ensemble d'entraînement.

Pour simplifier le problème de devoir produire des sorties du RNN quantique, nous

simplifierons également les tâches pour ne distinguer que 2 classes de Fashion-MNIST. Cela simplifie le problème de produire une sortie à partir de l'espace en très haute dimension du RNN quantique. Dans le même temps, nous essaierons de choisir les 2 classes les plus difficiles à distinguer, afin que les différences entre les capacités des différents RNN deviennent plus faciles à distinguer et moins affectées par le bruit inhérent à l'initialisation des paramètres aléatoires, à la procédure d'entraînement, etc.

Nous utiliserons donc les classes 4 (représentant des manteaux) et 6 (représentant des chemises) dans cette tâche [198]. Quelques exemples d'images de ces classes peuvent être vues dans la figure B.9 et la figure B.10. Les ensembles d'entraînement varieront de 1000 images pour chaque classe jusqu'à 5000 images pour chaque classe (correspondant au total de 2000 et 10000 images d'entraînement). Les ensembles de validation contiennent 1000 images de chaque classe. Ces deux ensembles sont composés d'images choisies au hasard sans remplacement sur l'ensemble de la série d'entraînement originale (de 6000 images pour chaque classe). Nous conservons l'ensemble du jeu de test non modifié (composé de 1000 images de chaque classe).

Figure B.9 – Exemples de manteaux de Fashion-MNIST

Figure B.10 – Exemples de chemises de Fashion-MNIST

B.5.2.2 Détails des systèmes

Dans cette sous-section, nous décrivons les détails des systèmes que nous avons simulés, à savoir le LSTM classique de référence et notre proposition hybride classique-quantique.

B.5.2.2.1 LSTM classique de référence Comme base de référence classique, nous avons utilisé une architecture LSTM (Long Short-Thort-Term Memory) à couche unique [112], sans aucune connexion de type peephole. Pour prédire la probabilité de la première des deux classes, nous calculons le produit scalaire entre l'état à la dernière étape temporelle h_T et un vecteur de même dimension des paramètres réels entraînables v (qui

contient les poids de la couche de sortie) et interprétons le résultat comme un logit. La fonction de perte que nous utilisons est la perte logistique.

Nous avons vérifié de manière empirique que cette approche fonctionne de manière similaire à l'approche habituelle qui consiste à utiliser une fonction de sortie softmax et de perte d'entropie croisée (sans perte significative de la précision de la prédiction).

Nous initialisons tous les poids LSTM à l'aide d'un initialiseur orthogonal [166] et initialisons les biais des portes d'oubli à 1, afin de faciliter l'apprentissage des dépendances à long terme, comme suggéré par exemple dans [122]. L'état initial h_0 est initialisé à 0 et le reste des biais sont initialisés à 0. Les poids et les biais de la couche en sortie sont initialisés, par souci de simplicité, en utilisant une distribution gaussienne (moyenne 0 et variance 1).

B.5.2.2.2 Système hybride classique-quantique Nous avons utilisé la même initialisation pour l'hyper-RNN classique (LSTM) dans notre proposition de système que pour la référence classique LSTM. Les composantes réelles et imaginaires de la couche de sortie v dans l'eq. B.4 ont été initialisées en utilisant une distribution gaussienne tronquée avec une moyenne 0 et un écart-type 0,01. Les poids W_l dans l'eq. B.6 sont initialisés à 0 et les biais b_l à 1.

Pour tous les résultats que nous rapportons, nous avons utilisé les valeurs $m^+ = 0.7$ et $m^- = 0.3$ pour la fonction de perte de marge de notre proposition. Ces valeurs semblaient donner les meilleurs résultats dans nos essais limités de recherche d'hyperparamètres.

B.5.2.2.3 Paramètres communs Pour l'optimisation, nous avons utilisé ADAM [125] avec les hyperparamètres par défaut (y compris le taux d'apprentissage 0,001) à la fois pour la réference classique et pour notre proposition de système. Pour les deux systèmes, l'écrêtage de gradient par norme globale [152] est utilisé, avec norme globale 5.

Pour la plupart des paramètres de tâche, nous utilisons des lots de taille 100 (sauf indication contraire). L'optimisation est toujours exécutée pour 100 époques. La validation est effectuée tous les 10000 exemples. Nous conservons la meilleure précision de validation et la précision de test correspondante.

Pour le système LSTM de référence, nous essayerons plusieurs configurations pour chaque tâche. Nous doublons le nombre d'unités pour chaque nouvelle configuration, ce qui correspond à environ 4 fois plus de paramètres entraînables. L'augmentation du nombre d'unités augmente la mémoire récurrente et l'augmentation du nombre de paramètres entraînables conduit à des modèles plus expressifs, mais également à un risque accru de surapprentissage et à un coût de calcul plus élevé. Pour chaque configuration du système de référence, nous entraînons le système plusieurs fois (correspondant à différentes graines aléatoires - random seed - à partir desquelles les paramètres entraînables sont initialisés) pour chaque système avec un nombre fixe d'unités et sélectionnons le système résultant avec

la meilleure précision sur l'ensemble de validation, pour lequel nous rapportons également la précision de test. Ceci est une configuration plus difficile pour notre proposition de système, car le système de référence a plusieurs chances d'obtenir de bonnes performances, alors que notre proposition aura une seule chance.

B.5.2.3 Approximation du coût de calcul

Dans cette section, nous décrivons l'approche que nous avons utilisée pour approximer les coûts de calcul de notre proposition et du système classique de référence (LSTM). Par souci de simplicité, nous utiliserons une approximation rapide des coûts de calcul. Évidemment, dans le cas d'une implémentation sur des ordinateurs quantiques (et non d'une simulation, comme dans notre cas), il faudrait prendre en compte un plus grand nombre de facteurs que nous négligeons ici pour des raisons de simplicité, notamment le surcoût de la correction d'erreur et de la gestion du bruit. Nous nous concentrerons sur les coûts d'inférence/déploiement.

Notre approche est similaire à la façon dont le coût de calcul est estimé dans [121], qui propose un RNN où les matrices de poids sont factorisées par Kronecker (similaire au nôtre, sans faire appel au calcul quantique). Notamment, nous ne prendrons en compte que le coût dominant de l'application de la transition récurrente (entre deux états consecutifs d'un réseau récurrent, qu'il soit classique ou quantique). Pour une séquence d'entrées T, la transition récurrente est appliquée T fois, contrairement à l'opération de sortie (qui génère les probabilités des classes), qui n'est appliquée qu'une seule fois. La transition récurrente est également différente des autres opérations (la multiplication des entrées par une matrice de poids dans le LSTM classique ou la géneration des paramètres du RNN quantique à partir de l'état de l'hyper-RNN classique), car elle ne peut pas être parallélisée (elle doit être appliquée séquentiellement, dans l'ordre, à chaque pas de temps). Nous négligeons en outre le coût d'utilisation de tout biais, car asymptotiquement, ils sont quadratiquement plus petits que le coûts des multiplications matrice - vecteur.

Dans le cas du modèle LSTM monocouche de référence comprenant des unités cachées h, nous estimons le coût de son exécution à chaque pas de temps par $4 * h^2$. Nous utilisons cette approximation car un LSTM a 4 matrices récurrentes de taille h^2.

Le coût de calcul approximé de notre RNN hybride-classique est obtenu de la manière suivante. Tout d'abord, le coût de l'hyper-RNN classique (un LSTM) est obtenu comme pour la référence classique.

Nous décomposons le coût du RNN quantique en coût du QFT et du IQFT et le coût des matrices unitaires paramétrées. Comme le QFT et l'IQFT peuvent être implémentés sur un ordinateur quantique avec un temps $O(n \log(n))$ [101] et que nous simulons entre $n = 8$ et $n = 16$ qubits, pour simplifier, nous estimons ces coûts à $4 * n$ (parce que $log_2(n) \leq 4$ pour $8 \leq n \leq 16$). Comme les matrices unitaires paramétrées n'utilisent que des portes à

un seul qubit et que chacune de ces portes peut être décomposée en 4 portes élémentaires, comme décrit dans la section B.5.1.1, nous estimons également ces coûts à $4 * n$. Le coût total par pas de temps du RNN quantique pour une éxécution est alors estimé à $4 * 4 * n$, somme des coûts du QFT, de l'IQFT et des matrices unitaires paramétrées.

Puisque nous devons effectuer des mesures répétées après le dernier pas de temps pour obtenir une approximation de la probabilité p_1, comme décrit dans la section B.5.1.2, nous devons répéter l'éxécution du RNN quantique pour chaque nouvelle mesure. Comme décrit dans la section B.5.1.2, p_1 peut être estimé avec précision ϵ si on mesure $O(\frac{1}{\epsilon})$ fois (et on répéte l'éxécution du RNN quantique $O(\frac{1}{\epsilon})$ fois). Pour des raisons de simplicité, nous approcherons le nombre de répétitions de l'éxécution du RNN quantique à $\frac{1}{\epsilon}$. Le coût de calcul approximé par pas de temps pour le RNN quantique sera alors $\frac{4 * 4 * n}{\epsilon}$ et le coût approximé par pas de temps pour l'ensemble du système hybride classique - quantique sera $4 * h^2 + \frac{4 * 4 * n}{\epsilon}$.

B.5.2.4 Estimation de la précision de classification sous approximation de l'échantillonnage de précision ϵ

Nous estimons la précision de classification de validation et de test de nos systèmes hybrides lorsque des mesures répétées $O(\frac{1}{\epsilon}$ fois (à précision ϵ) sont effectuées comme suit.

Nous comptons pour combien d'exemples $|p_1 - 0.5| < \epsilon$ (correspondant à tous les exemples pour lesquels la prédiction n'est pas suffisamment fiable). Par souci de simplicité, supposons alors que pour la moitié des exemples pour lesquels la prédiction n'est pas suffisamment fiable, l'approximation ϵ conduirait à une prédiction correcte (et pour l'autre moitié, la prédiction serait incorrecte). Nous recalculons ensuite les précisions de validation et de test en utilisant cette hypothèse (exactement la moitié de tous les exemples pour lesquels la prédiction n'est pas suffisamment fiable serait prédite correctement, au lieu d'utiliser le nombre initial de prédictions correctes, mais pas suffisamment fiables).

B.5.3 Résultats

Dans cette section, nous présentons les meilleurs résultats obtenus pour les différents paramètres de tâche (nombre de pixels de l'image d'entrée, nombre d'exemples d'entraînement, taille du lot) pris en compte. Nous présentons ces paramètres et les résultats correspondants dans l'ordre croissant des dépendances temporelles, dans l'ordre croissant de la taille du jeu d'apprentissage et de la taille du lot. Nous désignons les systèmes de référence classiques par LSTM-[h], où h est le nombre d'unités cachées. Pour notre proposition, nous utilisons la notation LSTM-[h']-[n]q, où h' est le nombre d'unités de l'hyper-LSTM et n le nombre de qubits de notre RNN quantum. Lors de la présentation des résultats des approximations d'échantillonage de précision ϵ, nous ajoutons la notation ϵ = [value]. Nous utilisons la

notation AFDF-[n]q pour désigner la composante quantique de notre système.

Nous montrons d'abord le coût approximé de calcul par pas de temps pour chacun des composants classiques et quantiques de notre système dans le tableau B.7. Dans le tableau B.8, nous affichons les facteurs de coûts supplémentaires résultant de mesures répétées sous différentes approximations de précision ϵ. Nous avons décrit comment le coût approximé total par pas de temps d'un système est obtenu à partir du coût des sous-systèmes classiques et quantiques, en tenant compte de l'approximation de précision ϵ, dans la section B.5.2.3. Pour des raisons de commodité, nous afficherons l'approximation du coût total par pas de temps pour chaque système dans chaque paramètre de tâche ci-dessous (à l'exception des résultats obtenus pour notre proposition lorsqu'aucune précision ϵ-accuracy n'est utilisée, puisque le coût dépendrait de ϵ).

System	Approximate computational cost
LSTM-7	196
LSTM-16	1024
LSTM-32	4096
LSTM-64	16384
LSTM-128	65536
LSTM-256	262144
LSTM-512	1048576
AFDF-9q	144
AFDF-10q	160
AFDF-11q	176
AFDF-13q	208

Table B.7 – Coût de calcul approximé (opérations) par pas de temps pour chaque composant de système (LSTM classique et AFDF quantique).

ϵ approximation	0.01	0.02	0.03
Extra cost factor	100	50	34

Table B.8 – Facteur de coût supplémentaire à partir de mesures répétées sous différentes approximations ϵ.

Nous montrons les meilleurs résultats obtenus par notre proposition dans le tableau B.9, pour 2000 exemples d'entraînement de 28 x 28 pixels et d'une taille de lot de 100 exemples. Notre proposition de système (LSTM-16-9q) obtient la meilleure précision de validation et de test, avec la performance LSTM-256 deuxième meilleure. Avec une approximation d'échantillonnage de 0,03, notre proposition serait environ 44 fois plus efficace en calcul que le LSTM-256. Sous une approximation d'échantillonnage de 0,01, notre proposition

System	Best validation accuracy	Test accuracy	Approximate computational cost
LSTM-16	0.7260	0.7160	1024
LSTM-32	0.8115	0.7880	4096
LSTM-64	0.7735	0.7590	16384
LSTM-128	0.8055	0.7880	65536
LSTM-256	**0.8155**	**0.8010**	262144
LSTM-16-9q	**0.8205**	**0.8040**	-
LSTM-16-9q $\epsilon = 0.01$	**0.8180**	**0.8030**	15424
LSTM-16-9q $\epsilon = 0.02$	**0.8122**	0.7955	8224
LSTM-16-9q $\epsilon = 0.03$	0.8067	0.7935	5920

Table B.9 – Résultats pour 2000 exemples d'entraînement, 28 x 28 pixels, taille des lots 100 exemples.

serait toujours environ **17 fois plus efficace en calcul** que le LSTM-256 et obtiendrait **la meilleure précision à la fois sur les ensembles de validation et de test** pour cette tâche.

Dans le tableau B.10, nous montrons les résultats obtenus pour 10000 exemples d'entraînement de 28 x 28 pixels et de taille de lot 100. Ici LSTM-16-9q obtient une meilleure précision de validation que tous les systèmes LSTM avec moins de 128 unités et une meilleure précision de test que tous les systèmes LSTM avec moins de 512 unités. Le LSTM-512 obtient les meilleures précisions de validation et de test, tandis que le LSTM-128 obtient la même précision de validation (mais une précision d'essai inférieure). Fait intéressant, les 9 qubits de notre proposition correspondent exactement à 512 états du système quantique ($\log_2(512) = 9$). Si l'on tient compte de l'approximation de l'échantillonnage à ϵ-près, pour $\epsilon = 0,01$ la précision de test serait égale à celle du LSTM-128, pour un coût de calcul qui serait environ 4 fois inférieur. Par rapport à LSTM-512, notre méthode serait environ 68 fois plus efficace pour $\epsilon = 0,01$. Ce résultat montre que notre proposition resterait efficace du point de vue calcul même si le nombre d'exemples d'entraînement augmente, ce qui exige que les modèles statistiques aient plus de capacité, et suggère que les gains sur le plan du calcul pourraient être asymptotiquement exponentiels.

B.5.4 Conclusions

Dans cette section, nous avons présenté des résultats préliminaires d'une nouvelle paramétrisation de réseau neuronal hybride classique-quantique. À notre connaissance, il s'agit

System	Best validation accuracy	Test accuracy	Approximate computational cost
LSTM-16	0.7920	0.7875	1024
LSTM-32	0.8560	0.8315	4096
LSTM-64	0.8585	0.8495	16384
LSTM-128	**0.8870**	0.8585	65536
LSTM-256	0.8840	0.8565	262144
LSTM-512	**0.8870**	**0.8745**	1048576
LSTM-16-9q	0.8675	0.8620	-
LSTM-16-9q $\epsilon = 0.01$	0.8642	0.8585	15424
LSTM-16-9q $\epsilon = 0.02$	0.8625	0.8557	8224
LSTM-16-9q $\epsilon = 0.03$	0.8578	0.8515	5920

Table B.10 – Résultats pour 10000 exemples d'entraînement, 28 x 28 pixels, taille du lot 100.

de la première approche visant à construire des réseaux récurrents quantiques de neurones qui présente des améliorations potentielles par rapport à une architecture RNN classique proche de l'état de l'art. Bien que nos résultats soient préliminaires, nous pensons qu'il s'agit d'une des premières études montrant des résultats empiriques prometteurs indiquant des avantages quantiques potentiels pour des tâches et des jeux de données d'apprentissage automatique de grande échelle.

B.6 Conclusions

Dans cette dernière partie, nous allons d'abord résumer notre travail, puis fournir une perspective plus spéculative sur les sujets abordés dans cette thèse, en discutant de la promesse des réseaux neuronaux et du calcul quantique pour résoudre la reconnaissance de l'écriture manuscrite (à un niveau humain de performance) et des implications potentielles pour l'intelligence générale artificielle.

Dans notre première contribution, nous avons amélioré les performances des réseaux convolutionnels pour la reconnaissance manuscrite des chiffres à l'aide de techniques récentes (au moment où le travail a été effectué) d'apprentissage profond.

Nous avons ensuite introduit les Réseaux Transformateurs Spatiaux Liés (RTSL), des Réseaux Transformateurs Spatiaux (RTS) avec des poids partagés, et différentes variantes d'entraînement permettant d'améliorer les performances sur une variante déformée de l'ensemble de données MNIST.

Dans notre contribution suivante, nous avons comparé la performance de l'ALSTM (Associative Long Short-Term Memory), une architecture de réseau neuronal récurrent récemment introduite, avec celle du LSTM (Long Short-Term Memory), sur le jeu de données IFN-ENIT (pour la reconnaissance d'écriture arabe).

Enfin, nous avons montré que l'intégration de l'informatique quantique aux réseaux de neurones pourrait fournir des gains de calcul (en termes de temps et de mémoire) pour la reconnaissance de séquences sur l'ensemble de données Fashion-MNIST. Nos résultats suggèrent que des améliorations exponentielles de la complexité de calcul pourraient être réalisées, en particulier pour les réseaux de neurones récurrents entraînés pour la classification des séquences.

Nous allons maintenant examiner comment notre travail s'intègre dans le contexte plus large de la reconnaissance de l'écriture manuscrite et, plus spéculativement, de l'intelligence artificielle générale. Certains chercheurs influents en apprentissage profond ont soutenu que l'apprentissage supervisé a été efficacement résolu grâce à l'apprentissage en profondeur, tant qu'il existe suffisamment de données [87].

Des performances surhumaines ont déjà été obtenues pour certaines tâches telles que la reconnaissance d'objets et la reconnaissance vocale, sur des jeux de données volumineux (ImageNet [104] [206] pour la reconnaissance d'objet et Switchboard [103] pour la reconnaissance vocale). Etant donné la corrélation entre les domaines (reconnaissance d'objet, reconnaissance de la parole et reconnaissance de l'écriture) et le succès d'architectures de réseau de neurones similaires dans les trois domaines, des performances surhumaines pour les tâches de reconnaissance de l'écriture devraient également être réalisables avec des méthodes similaires. Étant donné le puissance croissante en matière de calcul (loi de Moore) et d'innovations algorithmiques (souvent importées de domaines connexes tels que la vision par ordinateur, la reconnaissance vocale ou la traduction automatique), le goulot d'étranglement le plus important pourrait être la quantité disponible de données étiquetées de reconnaissance d'écriture manuscrite.

Comment le calcul quantique s'inscrit-il dans cette perspective? L'informatique quantique n'est probablement pas nécessaire pour résoudre le problème de la reconnaissance de l'écriture manuscrite, ni même de l'IA au niveau humain, car le cerveau humain n'utilise probablement pas l'informatique quantique [137]. Cependant, de nombreux résultats théoriques et empiriques récents suggèrent que l'informatique quantique pourrait apporter des avantages à de nombreux problèmes d'IA.

Les résultats les plus intéressants pour la reconnaissance générale de l'écriture manuscrite sont probablement ceux qui concernent les avantages du calcul quantique pour les tâches séquentielles en IA, en raison de la nature séquentielle de la reconnaissance de l'écriture manuscrite. Nos résultats empiriques présentés dans la section précédente suggèrent que le calcul quantique pourrait conduire à des réseaux neuronaux récurrents (potentiellement exponentiellement) plus efficaces. De multiples résultats théoriques suggèrent également

que les systèmes d'informatique quantique nécessitent moins de ressources mémoire (que le calcul classique) pour modéliser avec précision des systèmes dynamiques pour des tâches séquentielles [45] [78] [187]. Il semble possible de tirer parti de ces caractéristiques du calcul quantique pour construire des réseaux neuronaux récurrents plus efficaces en temps et en mémoire et, inversement, des RNNs qui obtiennent une meilleure précision pour des ressources de calcul similaires.

La capacité de discriminer des séquences ordonnées de stimuli et, plus généralement, la capacité de représenter et de traiter l'information séquentielle et la capacité de mémoire accrue ont été proposées comme des caractéristiques fondamentales qui différencient l'homme des autres animaux. Les performances humaines supérieures observées sur des tâches séquentielles peuvent être utiles pour une grande variété de problèmes, dont certains peuvent être résolus seulement par l'homme, comme la compréhension du langage [83]. En ce sens, la résolution de la reconnaissance de l'écriture manuscrite, pour laquelle l'apprentissage profond et le calcul quantique semblent des approches appropriées et avantageuses, pourrait être l'une des étapes importantes vers l'IA au niveau humain. Les exigences de moins de mémoire interne pour la mémoire interne des systèmes quantiques pourraient également suggérer (beaucoup plus spéculativement) que la performance surhumaine (potentiellement fortement) est plausible tant pour la reconnaissance de l'écriture manuscrite que pour l'intelligence artificielle générale.

References

[1] "Adiabatic quantum computation." [Online]. Available: https://en.wikipedia.org/wiki/Adiabatic_quantum_computation

[2] "Bentham collection | tranScriptorium." [Online]. Available: http://transcriptorium.eu/datasets/bentham-collection/

[3] "Biological neuron model." [Online]. Available: https://en.wikipedia.org/wiki/Biological_neuron_model

[4] "BQP." [Online]. Available: https://en.wikipedia.org/wiki/BQP

[5] "Calculus on Computational Graphs: Backpropagation – colah's blog." [Online]. Available: http://colah.github.io/posts/2015-08-Backprop/

[6] "Classification datasets results." [Online]. Available: http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html

[7] "Continuous-variable quantum information." [Online]. Available: https://en.wikipedia.org/wiki/Continuous-variable_quantum_information

[8] "Convolution as matrix multiplication." [Online]. Available: https://en.wikipedia.org/wiki/Toeplitz_matrix#Discrete_convolution

[9] "Convolution theorem." [Online]. Available: https://en.wikipedia.org/wiki/Convolution_theorem

[10] "Convolutional Neural Networks (LeNet) - DeepLearning 0.1 documentation." [Online]. Available: http://deeplearning.net/tutorial/lenet.html

[11] "CS231n Convolutional Neural Networks for Visual Recognition." [Online]. Available: http://cs231n.github.io/classification/

[12] "CS231n Convolutional Neural Networks for Visual Recognition." [Online]. Available: http://cs231n.github.io/convolutional-networks/

[13] "CS231n Convolutional Neural Networks for Visual Recognition." [Online]. Available: http://cs231n.github.io/optimization-1/

[14] "CS231n Convolutional Neural Networks for Visual Recognition." [Online]. Available: http://cs231n.github.io/optimization-1/#numerical

[15] "CS231n Convolutional Neural Networks for Visual Recognition." [Online]. Available: http://cs231n.github.io/optimization-2/

[16] "CS231n Convolutional Neural Networks for Visual Recognition." [Online]. Available: http://cs231n.github.io/optimization-2/#mat

[17] "Deep learning book. Optimization chapter." [Online]. Available: http://www.deeplearningbook.org/contents/optimization.html

[18] "Discriminative learning for DBNs [9 mins] - Deep neural nets with generative pre-training | Coursera." [Online]. Available: https://www.coursera.org/lecture/neural-networks/discriminative-learning-for-dbns-9-mins-nLRJy

[19] "IAM Handwriting Database - Computer Vision and Artificial Intelligence." [Online]. Available: http://www.fki.inf.unibe.ch/databases/iam-handwriting-database

[20] "IAM On-Line Handwriting Database - Computer Vision and Artificial Intelligence." [Online]. Available: http://www.fki.inf.unibe.ch/databases/iam-on-line-handwriting-database

[21] "ICDAR 2009 Competitions." [Online]. Available: http://www.cvc.uab.es/icdar2009/competitions.html

[22] "IFN/ENIT - database Arabic OCR handwritten arabic word recognition, Arabic database." [Online]. Available: http://www.ifnenit.com/

[23] "ImageNet." [Online]. Available: http://www.image-net.org/

[24] "No-cloning theorem." [Online]. Available: https://en.wikipedia.org/wiki/No-cloning_theorem

[25] "NP complexity class." [Online]. Available: https://en.wikipedia.org/wiki/NP_(complexity)

[26] "Ocropy." [Online]. Available: https://github.com/tmbdev/ocropy

[27] "OpenCV." [Online]. Available: https://opencv.org/

[28] "P complexity class." [Online]. Available: https://en.wikipedia.org/wiki/P_(complexity)

[29] "Quantum logic gate." [Online]. Available: https://en.wikipedia.org/wiki/Quantum_logic_gate

[30] "Qubit." [Online]. Available: https://en.wikipedia.org/wiki/Qubit

[31] "RIMES database." [Online]. Available: http://www.a2ialab.com/doku.php?id=rimes_database:start

[32] "RNN handwriting generation demo." [Online]. Available: https://www.cs.toronto.edu/~graves/handwriting.html

[33] "Simultaneous perturbation stochastic approximation." [Online]. Available: https://en.wikipedia.org/wiki/Simultaneous_perturbation_stochastic_approximation

[34] "TensorFlow clip by global norm." [Online]. Available: https://www.tensorflow.org/versions/r1.1/api_docs/python/tf/clip_by_global_norm

[35] "Unitary matrix wiki," https://en.wikipedia.org/wiki/Unitary_matrix.

[36] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," 2016. [Online]. Available: https://research.google.com/pubs/pub45381.html

[37] Andrew Gibiansky, "Speech Recognition with Neural Networks." [Online]. Available: http://andrew.gibiansky.com/blog/machine-learning/speech-recognition-neural-networks/

[38] M. Arjovsky, A. Shah, and Y. Bengio, "Unitary Evolution Recurrent Neural Networks," nov 2015. [Online]. Available: http://arxiv.org/abs/1511.06464

[39] D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," sep 2014. [Online]. Available: http://arxiv.org/abs/1409.0473

[40] S. Bai, J. Z. Kolter, and V. Koltun, "An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling," mar 2018. [Online]. Available: http://arxiv.org/abs/1803.01271

[41] M. Benedetti, J. Realpe-Gómez, and A. Perdomo-Ortiz, "Quantum-assisted Helmholtz machines: A quantum-classical deep learning framework for industrial datasets in near-term devices," aug 2017. [Online]. Available: https://arxiv.org/abs/1708.09784

[42] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, mar 1994. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/18267787http://ieeexplore.ieee.org/document/279181/

[43] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," *Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09*, pp. 1–8, 2009. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.192.9439

[44] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for Hyper-Parameter Optimization," in *Advances in Neural Information Processing Systems*, 2011, pp. 2546–2554. [Online]. Available: http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization

[45] F. C. Binder, J. Thompson, and M. Gu, "A practical, unitary simulator for non-Markovian complex processes," sep 2017. [Online]. Available: http://arxiv.org/abs/1709.02375

[46] T. Bluche, "Joint Line Segmentation and Transcription for End-to-End Handwritten Paragraph Recognition," apr 2016. [Online]. Available: http://arxiv.org/abs/1604.08352

[47] T. Bluche, C. Kermorvant, and J. Louradour, "Where to apply dropout in recurrent neural networks for handwriting recognition?" in *2015 13th International Conference on Document Analysis and Recognition (ICDAR)*. IEEE, aug 2015, pp. 681–685. [Online]. Available: http://ieeexplore.ieee.org/document/7333848/

[48] T. Bluche, J. Louradour, and R. Messina, "Scan, Attend and Read: End-to-End Handwritten Paragraph Recognition with MDLSTM Attention," apr 2016. [Online]. Available: http://arxiv.org/abs/1604.03286

[49] T. Bluche and R. Messina, "Gated Convolutional Recurrent Neural Networks for Multilingual Handwriting Recognition," in *ICDAR 2017*, 2017.

[50] T. Bluche, H. Ney, and C. Kermorvant, "A Comparison of Sequence-Trained Deep Neural Networks and Recurrent Neural Networks Optical Modeling for Handwriting Recognition," pp. 199–210, oct 2014. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-11397-5_15

[51] S. Boixo, S. V. Isakov, V. N. Smelyanskiy, R. Babbush, N. Ding, Z. Jiang, M. J. Bremner, J. M. Martinis, and H. Neven, "Characterizing Quantum Supremacy in Near-Term Devices," jul 2016. [Online]. Available: http://arxiv.org/abs/1608.00263

[52] A. Bouland, J. F. Fitzsimons, and D. E. Koh, "Quantum Advantage from Conjugated Clifford Circuits," sep 2017. [Online]. Available: http://arxiv.org/abs/1709.01805

[53] Y.-L. Boureau, J. Ponce, J. P. Fr, and Y. Lecun, "A Theoretical Analysis of Feature Pooling in Visual Recognition," Tech. Rep., 2010. [Online]. Available: http://yann.lecun.com/exdb/publis/pdf/boureau-icml-10.pdf

[54] E. Chammas, C. Mokbel, and L. Likforman-Sulem, "Handwriting Recognition of Historical Documents with Few Labeled Data," in *2018 13th IAPR International Workshop on Document Analysis Systems (DAS)*. IEEE, apr 2018, pp. 43–48. [Online]. Available: https://ieeexplore.ieee.org/document/8395169/

[55] J.-R. Chang and Y.-S. Chen, "Batch-normalized Maxout Network in Network," 2015. [Online]. Available: http://arxiv.org/abs/1511.02583

[56] S. Chang, Y. Zhang, W. Han, M. Yu, X. Guo, W. Tan, X. Cui, M. Witbrock, M. Hasegawa-Johnson, and T. S. Huang, "Dilated Recurrent Neural Networks," oct 2017. [Online]. Available: http://arxiv.org/abs/1710.02224

[57] K. B. Charbonneau and O. Shouno, "Neural Trajectory Analysis of Recurrent Neural Network In Handwriting Synthesis," apr 2018. [Online]. Available: http://arxiv.org/abs/1804.04890

[58] P. Chaudhari and S. Soatto, "Stochastic gradient descent performs variational inference, converges to limit cycles for deep networks," oct 2017. [Online]. Available: http://arxiv.org/abs/1710.11029

[59] L. Chen, R. Yan, L. Peng, A. Furuhata, and X. Ding, "Multi-layer recurrent neural network based offline Arabic handwriting recognition," in *2017 1st International Workshop on Arabic Script Analysis and Recognition (ASAR)*. IEEE, apr 2017, pp. 6–10. [Online]. Available: http://ieeexplore.ieee.org/document/8067749/

[60] K. Cho, A. Courville, and Y. Bengio, "Describing Multimedia Content using Attention-based Encoder–Decoder Networks," jul 2015. [Online]. Available: http://arxiv.org/abs/1507.01053

[61] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation," Tech. Rep. [Online]. Available: https://www.aclweb.org/anthology/D14-1179

[62] J. Chung, S. Ahn, and Y. Bengio, "Hierarchical Multiscale Recurrent Neural Networks," sep 2016. [Online]. Available: http://arxiv.org/abs/1609.01704

[63] J. Chung, K. Kastner, L. Dinh, K. Goel, A. Courville, and Y. Bengio, "A Recurrent Latent Variable Model for Sequential Data," jun 2015. [Online]. Available: http://arxiv.org/abs/1506.02216

[64] D. Cireşan, U. Meier, and J. Schmidhuber, "Multi-column Deep Neural Networks for Image Classification," 2012. [Online]. Available: http://arxiv.org/abs/1202.2745

[65] D. C. Cireşan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Deep, Big, Simple Neural Nets for Handwritten Digit Recognition," *Neural Computation*, vol. 22, no. 12, pp. 3207–3220, 2010. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/20858131

[66] D. C. Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Handwritten Digit Recognition with a Committee of Deep Neural Nets on GPUs," *CoRR*, vol. abs/1103.4, 2011. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1103.html#abs-1103-4487

[67] B.-I. Cîrstea, "Recognition and information extraction in multi-lingual documents with recurrent neural networks and deep neural networks," in *Doctoral Consortium of the ICDAR 2015 Conference, Nancy*, 2015.

[68] B.-I. Cîrstea and L. Likforman-Sulem, "Improving a deep convolutional neural network architecture for character recognition," in *Document Recognition and Retrieval XXIII 2016, San Francisco*, 2016.

[69] ——, "Tied spatial transformer networks for character recognition," in *Rencontres Jeunes Chercheurs of CIFED-CORIA 2016, Toulouse*, 2016.

[70] ——, "Tied spatial transformer networks for digit recognition," in *2016 15th International Conference on Frontiers in Handwriting Recognition (ICFHR)*. IEEE, oct 2016, pp. 524–529. [Online]. Available: http://ieeexplore.ieee.org/document/7814118/

[71] J. Collins, J. Sohl-Dickstein, and D. Sussillo, "Capacity and Trainability in Recurrent Neural Networks," nov 2016. [Online]. Available: https://arxiv.org/abs/1611.09913

[72] T. Cooijmans, N. Ballas, C. Laurent, Ç. Gülçehre, and A. Courville, "Recurrent Batch Normalization," mar 2016. [Online]. Available: http://arxiv.org/abs/1603.09025

[73] G. Cybenkot, "Mathematics of Control, Signals, and Systems Approximation by Superpositions of a Sigmoidal Function*," Tech. Rep., 1989. [Online]. Available: http://www.dartmouth.edu/~gvc/Cybenko_MCSS.pdf

[74] I. Danihelka, G. Wayne, B. Uria, N. Kalchbrenner, and A. Graves, "Associative Long Short-Term Memory," feb 2016. [Online]. Available: http://arxiv.org/abs/1602.03032

[75] S. Dieleman, J. Schlüter, C. Raffel, E. Olson, S. K. Sønderby, D. Nouri, D. Maturana, M. Thoma, E. Battenberg, J. Kelly, J. D. Fauw, M. Heilman, Diogo149, B. McFee, H. Weideman, Takacsg84, Peterderivaz, Jon, Instagibbs, D. K. Rasul, CongLiu, Britefury, and J. Degrave, "Lasagne: First Release." *doi.org*, pp. –, aug 2015. [Online]. Available: https://zenodo.org/record/27878#.WgDcAHUrJCU

[76] P. Doetsch, A. Zeyer, and H. Ney, "Bidirectional decoder networks for attention-based end-to-end offline handwriting recognition," in *Frontiers in Handwriting Recognition (ICFHR), 2016 15th International Conference on*, 2016, pp. 361–366.

[77] H. El Abed and V. Margner, "The IFN/ENIT-database - a tool to develop Arabic handwriting recognition systems," in *2007 9th International Symposium on Signal Processing and Its Applications*. IEEE, feb 2007, pp. 1–4. [Online]. Available: http://ieeexplore.ieee.org/document/4555529/

[78] T. J. Elliott and M. Gu, "Superior memory efficiency of quantum devices for the simulation of continuous-time stochastic processes," *npj Quantum Information*, vol. 4, no. 1, p. 18, dec 2018. [Online]. Available: http://www.nature.com/articles/s41534-018-0064-4

[79] E. Farhi and H. Neven, "Classification with Quantum Neural Networks on Near Term Processors," feb 2018. [Online]. Available: http://arxiv.org/abs/1802.06002

[80] J. N. Foerster, J. Gilmer, J. Chorowski, J. Sohl-Dickstein, and D. Sussillo, "Input Switched Affine Networks: An RNN Architecture Designed for Interpretability," nov 2016. [Online]. Available: https://arxiv.org/abs/1611.09434

[81] X. Gao, Z. Zhang, and L. Duan, "An efficient quantum algorithm for generative machine learning," nov 2017. [Online]. Available: http://arxiv.org/abs/1711.02038

[82] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, "Convolutional Sequence to Sequence Learning," may 2017. [Online]. Available: http://arxiv.org/abs/1705.03122

[83] S. Ghirlanda, J. Lind, and M. Enquist, "Memory for stimulus sequences: a divide between humans and other animals?" *Royal Society Open Science*, vol. 4, no. 6, p. 161011, jun 2017. [Online]. Available: http://rsos.royalsocietypublishing.org/lookup/doi/10.1098/rsos.161011

[84] V. Giovannetti, S. Lloyd, and L. Maccone, "Quantum random access memory," aug 2007. [Online]. Available: http://arxiv.org/abs/0708.1879

[85] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[86] X. Glorot, A. Bordes, and Y. Bengio, "Deep Sparse Rectifier Neural Networks," in *International Conference on Artificial Intelligence and Statistics*, 2011, pp. 315–323.

[87] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[88] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, "Maxout Networks," pp. 1319–1327, 2013. [Online]. Available: http://arxiv.org/abs/1302.4389

[89] A. Goyal, A. Sordoni, M.-A. Côté, N. R. Ke, and Y. Bengio, "Z-Forcing: Training Stochastic Recurrent Networks," nov 2017. [Online]. Available: http://arxiv.org/abs/1711.05411

[90] B. Graham, "Fractional Max-Pooling," *CoRR*, vol. abs/1412.6, 2014. [Online]. Available: https://arxiv.org/abs/1412.6071

[91] E. Granell, E. Chammas, L. Likforman-Sulem, C.-D. Martínez-Hinarejos, C. Mokbel, and B.-I. Cîrstea, "Transcription of Spanish Historical Handwritten Documents with Deep Neural Networks," *Journal of Imaging*, vol. 4, no. 1, p. 15, jan 2018. [Online]. Available: http://www.mdpi.com/2313-433X/4/1/15

[92] A. Graves, "Supervised sequence labelling with recurrent neural networks." Ph.D. dissertation, 2008.

[93] ——, *Supervised Sequence Labelling with Recurrent Neural Networks*, ser. Studies in Computational Intelligence. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 385. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24797-2

[94] ——, "Generating Sequences With Recurrent Neural Networks," aug 2013. [Online]. Available: http://arxiv.org/abs/1308.0850

[95] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber, "Connectionist temporal classification," in *Proceedings of the 23rd international conference on Machine learning - ICML '06*. New York, New York, USA: ACM Press, 2006, pp. 369–376. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1143844.1143891

[96] A. Graves, S. Fernandez, and J. Schmidhuber, "Multi-Dimensional Recurrent Neural Networks," may 2007. [Online]. Available: http://arxiv.org/abs/0705.2011

[97] A. Graves and J. Schmidhuber, "Offline Handwriting Recognition with Multi-dimensional Recurrent Neural Networks," in *Advances in Neural Information Processing Systems*, 2009, pp. 545–552. [Online]. Available: http://papers.nips.cc/paper/3449-offline-handwriting-recognition-with-multidimensional-recurrent-neural-networks

[98] A. Graves, G. Wayne, and I. Danihelka, "Neural Turing Machines," oct 2014. [Online]. Available: http://arxiv.org/abs/1410.5401

[99] E. Grosicki and H. El Abed, "ICDAR 2009 Handwriting Recognition Competition," 2009.

[100] D. Ha, A. Dai, and Q. V. Le, "HyperNetworks," 2016. [Online]. Available: http://arxiv.org/abs/1609.09106

[101] L. Hales, L. Hales, and S. Hallgren, "An Improved Quantum Fourier Transform Algorithm and Applications," *IN PROCEEDINGS OF THE 41ST ANNUAL SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE*, pp. 515–525, 2000. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.4161

[102] A. Hallam, E. Grant, V. Stojevic, S. Severini, and A. G. Green, "Compact Neural Networks based on the Multiscale Entanglement Renormalization Ansatz," nov 2017. [Online]. Available: http://arxiv.org/abs/1711.03357

[103] K. J. Han, A. Chandrashekaran, J. Kim, and I. Lane, "The CAPIO 2017 Conversational Speech Recognition System," dec 2017. [Online]. Available: http://arxiv.org/abs/1801.00059

[104] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," dec 2015. [Online]. Available: https://arxiv.org/abs/1512.03385

[105] ——, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," feb 2015. [Online]. Available: http://arxiv.org/abs/1502.01852

[106] Z. He, S. Gao, L. Xiao, D. Liu, H. He, and D. Barber, "Wider and Deeper, Cheaper and Faster: Tensorized LSTMs for Sequence Learning," nov 2017. [Online]. Available: http://arxiv.org/abs/1711.01577

[107] K. Helfrich, D. Willmott, and Q. Ye, "Orthogonal Recurrent Neural Networks with Scaled Cayley Transform," jul 2017. [Online]. Available: http://arxiv.org/abs/1707.09520

[108] G. Hinton, N. Srivastava, and K. Swersky, "Neural Networks for Machine Learning Lecture 1a Why do we need machine learning?" Tech. Rep. [Online]. Available: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec1.pdf

[109] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," Tech. Rep. [Online]. Available: https://www.cs.toronto.edu/~hinton/absps/fastnc.pdf

[110] S. Hochreiter, "The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 06, no. 02, pp. 107–116, apr 1998. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0218488598000094

[111] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber, "Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies," 2001. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.24.7321

[112] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, nov 1997. [Online]. Available: http://www.mitpressjournals.org/doi/10.1162/neco.1997.9.8.1735

[113] D. H. Hubel and T. N. Wiesel, "Receptive fields and functional architecture of monkey striate cortex." *The Journal of physiology*, vol. 195, no. 1, pp. 215–43, mar 1968. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/4966457

[114] W. Huggins, P. Patel, K. B. Whaley, and E. M. Stoudenmire, "Towards Quantum Machine Learning with Tensor Networks," mar 2018. [Online]. Available: http://arxiv.org/abs/1803.11537

[115] H. Hyotyniemi, "Turing Machines are Recurrent Neural Networks," 1996. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.5161

[116] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[117] M. Jaderberg, K. Simonyan, A. Zisserman, and K. Kavukcuoglu, "Spatial Transformer Networks." *CoRR*, vol. abs/1506.0, jun 2015. [Online]. Available: http://arxiv.org/abs/1506.02025

[118] H. Jaeger, "Echo state network." [Online]. Available: http://www.scholarpedia.org/article/Echo_state_network

[119] W. Jian, "RNNLIB: Connectionist Temporal Classification and Transcription Layer." [Online]. Available: http://wantee.github.io/2015/02/08/rnnlib-connectionist-temporal-classification-and-transcription-layer/

[120] L. Jing, Y. Shen, T. Dubček, J. Peurifoy, S. Skirlo, Y. LeCun, M. Tegmark, and M. Soljačić, "Tunable Efficient Unitary Neural Networks (EUNN) and their application to RNNs," dec 2016. [Online]. Available: http://arxiv.org/abs/1612.05231

[121] C. Jose, M. Cisse, and F. Fleuret, "Kronecker Recurrent Units," may 2017. [Online]. Available: http://arxiv.org/abs/1705.10142

[122] R. Jozefowicz, W. Zaremba, and I. Sutskever, "An empirical exploration of recurrent network architectures," pp. 2342–2350, 2015. [Online]. Available: https://dl.acm.org/citation.cfm?id=3045367

[123] I. Kerenidis and A. Luongo, "Quantum classification of the MNIST dataset via Slow Feature Analysis," may 2018. [Online]. Available: http://arxiv.org/abs/1805.08837

[124] N. Killoran, T. R. Bromley, J. M. Arrazola, M. Schuld, N. Quesada, and S. Lloyd, "Continuous-variable quantum neural networks," 6 2018. [Online]. Available: http://arxiv.org/abs/1806.06871

[125] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," dec 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[126] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems*, 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks

[127] D. Krueger, T. Maharaj, J. Kramár, M. Pezeshki, N. Ballas, N. R. Ke, A. Goyal, Y. Bengio, A. Courville, and C. Pal, "Zoneout: Regularizing RNNs by Randomly Preserving Hidden Activations," jun 2016. [Online]. Available: http://arxiv.org/abs/1606.01305

[128] H. Larochelle and I. Murray, "The Neural Autoregressive Distribution Estimator," pp. 29–37, jun 2011. [Online]. Available: http://proceedings.mlr.press/v15/larochelle11a.html

[129] Q. V. Le, N. Jaitly, and G. E. Hinton, "A Simple Way to Initialize Recurrent Networks of Rectified Linear Units," apr 2015. [Online]. Available: http://arxiv.org/abs/1504.00941

[130] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=726791

[131] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, may 2015. [Online]. Available: http://www.nature.com/doifinder/10.1038/nature14539

[132] Y. LeCun, C. Cortes, and C. J. C. Burges, "MNIST," http://yann.lecun.com/exdb/mnist/.

[133] Y. Lecun, Y. Lecun, L. D. Jackel, H. A. Eduard, N. Bottou, C. Cartes, J. S. Denker, H. Drucker, E. Sackinger, P. Simard, and V. Vapnik, "Learning Algorithms For Classification: A Comparison On Handwritten Digit Recognition," *NEURAL NETWORKS: THE STATISTICAL MECHANICS PERSPECTIVE*, pp. 261—-276, 1995. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.21.4628

[134] C.-Y. Lee, S. Xie, P. Gallagher, Z. Zhang, and Z. Tu, "Deeply-Supervised Nets," sep 2014. [Online]. Available: http://arxiv.org/abs/1409.5185

[135] F.-F. Li, J. Johnson, and S. Young, "Recurrent Neural Networks," http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture10.pdf.

[136] M. Lin, Q. Chen, and S. Yan, "Network In Network," p. 10, 2013. [Online]. Available: http://arxiv.org/abs/1312.4400

[137] A. Litt, C. Eliasmith, F. W. Kroon, S. Weinstein, and P. Thagard, "Is the Brain a Quantum Computer?" Tech. Rep., 2006. [Online]. Available: https://onlinelibrary.wiley.com/doi/pdf/10.1207/s15516709cog0000_59

[138] D. Liu, S.-J. Ran, P. Wittek, C. Peng, R. B. García, G. Su, and M. Lewenstein, "Machine Learning by Two-Dimensional Hierarchical Tensor Networks: A Quantum Information Theoretic Perspective on Deep Architectures," oct 2017. [Online]. Available: http://arxiv.org/abs/1710.04833

[139] S. Lloyd, M. Mohseni, and P. Rebentrost, "Quantum algorithms for supervised and unsupervised machine learning," jul 2013. [Online]. Available: http://arxiv.org/abs/1307.0411

[140] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," *Proc. ICML*, vol. 30, 2013.

[141] J. Mairal, P. Koniusz, Z. Harchaoui, and C. Schmid, "Convolutional Kernel Networks." in *NIPS*, 2014, pp. 2627–2635. [Online]. Available: http://dblp.uni-trier.de/db/conf/nips/nips2014.html#MairalKHS14

[142] V. Märgner and H. El Abed, "ICDAR 2009 Arabic Handwriting Recognition Competition," 2009. [Online]. Available: www.ifnenit.com

[143] U.-V. Marti and H. Bunke, "The IAM-database: an English sentence database for offline handwriting recognition," *International Journal on Document Analysis and Recognition*, vol. 5, no. 1, pp. 39–46, nov 2002. [Online]. Available: http://link.springer.com/10.1007/s100320200071

[144] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115–133, dec 1943. [Online]. Available: http://link.springer.com/10.1007/BF02478259

[145] Z. Mhammedi, A. Hellicar, A. Rahman, and J. Bailey, "Efficient Orthogonal Parametrisation of Recurrent Neural Networks Using Householder Reflections," dec 2016. [Online]. Available: http://arxiv.org/abs/1612.00188

[146] V. Mnih, N. Heess, A. Graves, and K. Kavukcuoglu, "Recurrent Models of Visual Attention," jun 2014. [Online]. Available: http://arxiv.org/abs/1406.6247

[147] M. Moczulski, M. Denil, J. Appleyard, and N. de Freitas, "ACDC: A Structured Efficient Linear Layer," nov 2015. [Online]. Available: http://arxiv.org/abs/1511.05946

[148] S. Mozaffari and H. Soltanizadeh, "ICDAR 2009 Handwritten Farsi/Arabic Character Recognition Competition," in *2009 10th International Conference on Document Analysis and Recognition*. IEEE, 2009, pp. 1413–1417. [Online]. Available: http://ieeexplore.ieee.org/document/5277795/

[149] Q. Nguyen and M. Hein, "Optimization Landscape and Expressivity of Deep CNNs," oct 2017. [Online]. Available: http://arxiv.org/abs/1710.10928

[150] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*. Cambridge University Press, 2010.

[151] N. Otsu, "A Threshold Selection Method from Gray-Level Histograms," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 9, no. 1, pp. 62–66, jan 1979. [Online]. Available: http://ieeexplore.ieee.org/document/4310076/

[152] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *Proceedings of The 30th International Conference on Machine Learning*, 2013, pp. 1310–1318. [Online]. Available: http://jmlr.csail.mit.edu/proceedings/papers/v28/pascanu13.html

[153] A. Perdomo-Ortiz, M. Benedetti, J. Realpe-Gómez, and R. Biswas, "Opportunities and challenges for quantum-assisted machine learning in near-term quantum computers," aug 2017. [Online]. Available: http://arxiv.org/abs/1708.09757

[154] T. A. Plate and T. A., *Holographic reduced representation : distributed representation for cognitive structures*. CSLI Publications, 2003. [Online]. Available: https://dl.acm.org/citation.cfm?id=862030

[155] T. Poggio, H. Mhaskar, L. Rosasco, B. Miranda, and Q. Liao, "Why and When Can Deep – but Not Shallow – Networks Avoid the Curse of Dimensionality: a Review," nov 2016. [Online]. Available: http://arxiv.org/abs/1611.00740

[156] B. Poole, S. Lahiri, M. Raghu, J. Sohl-Dickstein, and S. Ganguli, "Exponential expressivity in deep neural networks through transient chaos," jun 2016. [Online]. Available: http://arxiv.org/abs/1606.05340

[157] J. Puigcerver, "Are Multidimensional Recurrent Layers Really Necessary for Handwritten Text Recognition?" in *ICDAR 2017*, 2017.

[158] M. Raghu, B. Poole, J. Kleinberg, S. Ganguli, and J. Sohl-Dickstein, "On the Expressive Power of Deep Neural Networks," pp. 2847–2854. [Online]. Available: http://arxiv.org/abs/1606.05336

[159] R. Raz and A. Tal, "Oracle Separation of BQP and PH." [Online]. Available: https://eccc.weizmann.ac.il/report/2018/107/

[160] P. Rebentrost, T. R. Bromley, C. Weedbrook, and S. Lloyd, "A Quantum Hopfield Neural Network," oct 2017. [Online]. Available: http://arxiv.org/abs/1710.03599

[161] J. Romero, J. P. Olson, and A. Aspuru-Guzik, "Quantum autoencoders for efficient compression of quantum data," dec 2016. [Online]. Available: http://arxiv.org/abs/1612.02806http://dx.doi.org/10.1088/2058-9565/aa8072

[162] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," pp. 318–362, jan 1986. [Online]. Available: http://dl.acm.org/citation.cfm?id=104279.104293

[163] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, apr 2015. [Online]. Available: http://dblp.uni-trier.de/db/journals/ijcv/ijcv115.html#RussakovskyDSKS15

[164] S. Sabour, N. Frosst, and G. E. Hinton, "Dynamic Routing Between Capsules," oct 2017. [Online]. Available: http://arxiv.org/abs/1710.09829

[165] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution Strategies as a Scalable Alternative to Reinforcement Learning," mar 2017. [Online]. Available: http://arxiv.org/abs/1703.03864

[166] A. M. Saxe, J. L. McClelland, and S. Ganguli, "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks," dec 2013. [Online]. Available: http://arxiv.org/abs/1312.6120

[167] A. M. Schäfer and H. G. Zimmermann, "Recurrent Neural Networks Are Universal Approximators," in *Proceedings of the 16th international conference on Artificial Neural Networks - Volume Part I*. Springer-Verlag, 2006, pp. 632–640. [Online]. Available: http://link.springer.com/10.1007/11840817{_}66

[168] J. Schmidhuber, "Deep Learning in Neural Networks: An Overview," apr 2014. [Online]. Available: http://arxiv.org/abs/1404.7828

[169] M. Schuld, I. Sinayskiy, and F. Petruccione, "The quest for a Quantum Neural Network," aug 2014. [Online]. Available: http://arxiv.org/abs/1408.7005http://dx.doi.org/10.1007/s11128-014-0809-8

[170] H. T. Siegelmann and E. D. Sontag, "On the computational power of neural nets," in *Proceedings of the fifth annual workshop on Computational learning theory - COLT '92*. New York, New York, USA: ACM Press, 1992, pp. 440–449. [Online]. Available: http://portal.acm.org/citation.cfm?doid=130385.130432

[171] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," 2014. [Online]. Available: http://arxiv.org/abs/1409.1556

[172] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian Optimization of Machine Learning Algorithms," in *Advances in Neural Information Processing Systems*, 2012, pp. 2951–2959. [Online]. Available: http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms

[173] S. K. Sønderby, "Cluttered MNIST dataset," https://s3.amazonaws.com/lasagne/recipes/datasets/mnist_cluttered_60x60_6distortions.npz.

[174] ——, "Spatial Transformer Network code example ('recipe')," https://github.com/Lasagne/Recipes/blob/master/examples/spatial_transformer_network.ipynb.

[175] ——, "Spatial Transformer Network code repository," https://github.com/skaae/transformer_network.

[176] S. K. Sønderby, C. K. Sønderby, L. Maaløe, and O. Winther, "Recurrent Spatial Transformer Networks," sep 2015. [Online]. Available: http://arxiv.org/abs/1509.05329

[177] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for Simplicity: The All Convolutional Net," dec 2014. [Online]. Available: http://arxiv.org/abs/1412.6806

[178] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting." *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[179] E. M. Stoudenmire, "Learning Relevant Features of Data with Multi-scale Tensor Networks," dec 2017. [Online]. Available: http://arxiv.org/abs/1801.00315

[180] E. M. Stoudenmire and D. J. Schwab, "Supervised Learning with Quantum-Inspired Tensor Networks," 2016. [Online]. Available: http://arxiv.org/abs/1605.05775

[181] T. Strauss, G. Leifert, R. Labahn, and G. Mühlberger, "ICFHR2018 Competition on Automated Text Recognition on a READ Dataset."

[182] J. Sueiras, V. Ruiz, A. Sanchez, and J. F. Velez, "Offline continuous handwriting recognition using sequence to sequence neural networks," *Neurocomputing*, vol. 289, pp. 119–128, may 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231218301371

[183] I. Sutskever Google, O. Vinyals Google, Q. V. Le Google, I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks." in *NIPS*, 2014, pp. 3104–3112. [Online]. Available: https://arxiv.org/pdf/1409.3215.pdf

[184] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," feb 2016. [Online]. Available: http://arxiv.org/abs/1602.07261

[185] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper with Convolutions," sep 2014. [Online]. Available: http://arxiv.org/abs/1409.4842

[186] T. T. D. The Theano Development Team, R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, Y. Bengio, A. Bergeron, J. Bergstra, V. Bisson, J. B. Snyder, N. Bouchard, N. Boulanger-Lewandowski, X. Bouthillier, A. de Brébisson, O. Breuleux, P.-L. Carrier, K. Cho, J. Chorowski, P. Christiano, T. Cooijmans, M.-A. Côté, M. Côté, A. Courville, Y. N. Dauphin, O. Delalleau, J. Demouth, G. Desjardins, S. Dieleman, L. Dinh, M. Ducoffe, V. Dumoulin, S. E. Kahou, D. Erhan, Z. Fan, O. Firat, M. Germain, X. Glorot, I. Goodfellow, M. Graham, C. Gulcehre, P. Hamel, I. Harlouchet, J.-P. Heng, B. Hidasi, S. Honari, A. Jain, S. Jean, K. Jia, M. Korobov, V. Kulkarni, A. Lamb, P. Lamblin, E. Larsen, C. Laurent, S. Lee, S. Lefrancois, S. Lemieux, N. Léonard, Z. Lin, J. A. Livezey, C. Lorenz, J. Lowin, Q. Ma, P.-A. Manzagol, O. Mastropietro, R. T. McGibbon, R. Memisevic, B. van Merriënboer, V. Michalski, M. Mirza, A. Orlandi, C. Pal, R. Pascanu, M. Pezeshki, C. Raffel, D. Renshaw, M. Rocklin, A. Romero, M. Roth, P. Sadowski, J. Salvatier, F. Savard, J. Schlüter, J. Schulman, G. Schwartz, I. V. Serban, D. Serdyuk, S. Shabanian, É. Simon, S. Spieckermann, S. R. Subramanyam, J. Sygnowski, J. Tanguay, G. van Tulder, J. Turian, S. Urban, P. Vincent, F. Visin, H. de Vries, D. Warde-Farley, D. J. Webb, M. Willson, K. Xu, L. Xue, L. Yao, S. Zhang, and Y. Zhang, "Theano: A Python framework for fast computation of mathematical expressions," may 2016. [Online]. Available: http://arxiv.org/abs/1605.02688

[187] J. Thompson, A. J. P. Garner, V. Vedral, and M. Gu, "Using quantum theory to reduce the complexity of input-output processes," jan 2016. [Online]. Available: http://arxiv.org/abs/1601.05420

[188] J. van der Westhuizen and J. Lasenby, "The unreasonable effectiveness of the forget gate," apr 2018. [Online]. Available: http://arxiv.org/abs/1804.04849

[189] G. Verdon, J. Pye, and M. Broughton, "A Universal Training Algorithm for Quantum Deep Learning," 6 2018. [Online]. Available: http://arxiv.org/abs/1806.09729

[190] G. Vidal, "Entanglement Renormalization: an introduction," dec 2009. [Online]. Available: http://arxiv.org/abs/0912.1651

[191] F. Visin, K. Kastner, K. Cho, M. Matteucci, A. C. Courville, and Y. Bengio, "ReNet: A Recurrent Neural Network Based Alternative to Convolutional Networks," *CoRR*, vol. abs/1505.0, 2015. [Online]. Available: https://arxiv.org/abs/1505.00393

[192] E. Vorontsov, C. Trabelsi, S. Kadoury, and C. Pal, "On orthogonality and learning recurrent networks with long term dependencies," jan 2017. [Online]. Available: http://arxiv.org/abs/1702.00071

[193] K. H. Wan, O. Dahlsten, H. Kristjánsson, R. Gardner, and M. S. Kim, "Quantum generalisation of feedforward neural networks," dec 2016. [Online]. Available: http://arxiv.org/abs/1612.01045

[194] L. Wan, M. D. Zeiler, S. Zhang, Y. LeCun, and R. Fergus, "Regularization of Neural Networks using DropConnect," in *ICML (3)*, 2013, pp. 1058–1066. [Online]. Available: http://dblp.uni-trier.de/db/conf/icml/icml2013.html#WanZZLF13

[195] Y. Wen, P. Vicol, J. Ba, D. Tran, and R. Grosse, "Flipout: Efficient Pseudo-Independent Weight Perturbations on Mini-Batches," mar 2018. [Online]. Available: http://arxiv.org/abs/1803.04386

[196] N. Wiebe, A. Kapoor, and K. Svore, "Quantum Algorithms for Nearest-Neighbor Methods for Supervised and Unsupervised Learning," 2014. [Online]. Available: http://arxiv.org/abs/1401.2142

[197] S. Wisdom, T. Powers, J. R. Hershey, J. L. Roux, and L. Atlas, "Full-Capacity Unitary Recurrent Neural Networks," oct 2016. [Online]. Available: http://arxiv.org/abs/1611.00035

[198] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST." [Online]. Available: https://github.com/zalandoresearch/fashion-mnist

[199] ——, "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms," aug 2017. [Online]. Available: http://arxiv.org/abs/1708.07747

[200] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical Evaluation of Rectified Activations in Convolutional Network," 2015. [Online]. Available: http://arxiv.org/abs/1505.00853

[201] Y. Yang, D. Krompass, and V. Tresp, "Tensor-Train Recurrent Neural Networks for Video Classification," jul 2017. [Online]. Available: http://arxiv.org/abs/1707.01786

[202] M. R. Yousefi, M. R. Soheili, T. M. Breuel, and D. Stricker, "A comparison of 1D and 2D LSTM architectures for the recognition of handwritten Arabic," E. K. Ringger and B. Lamiroy, Eds., feb 2015, p. 94020H. [Online]. Available: http://proceedings.spiedigitallibrary.org/proceeding.aspx?doi=10.1117/12.2075930

[203] M. Zak and C. P. Williams, "Quantum Neural Nets," *International Journal of Theoretical Physics*, vol. 37, no. 2, pp. 651–684, 1998. [Online]. Available: http://link.springer.com/10.1023/A:1026656110699

[204] S. Zhang, Y. Wu, T. Che, Z. Lin, R. Memisevic, R. Salakhutdinov, and Y. Bengio, "Architectural Complexity Measures of Recurrent Neural Networks," feb 2016. [Online]. Available: http://arxiv.org/abs/1602.08210

[205] Y. Zhang, M. Pezeshki, P. Brakel, S. Zhang, C. L. Y. Bengio, and A. Courville, "Towards End-to-End Speech Recognition with Deep Convolutional Neural Networks," jan 2017. [Online]. Available: http://arxiv.org/abs/1701.02720

[206] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning Transferable Architectures for Scalable Image Recognition," jul 2017. [Online]. Available: http://arxiv.org/abs/1707.07012

Contributions à la reconnaissance de l'écriture manuscrite en utilisant des réseaux de neurones profonds et le calcul quantique

Bogdan-Ionuţ CÎRSTEA

RESUME: Dans cette thèse, nous fournissons plusieurs contributions des domaines de l'apprentissage profond et du calcul quantique à la reconnaissance de l'écriture manuscrite.

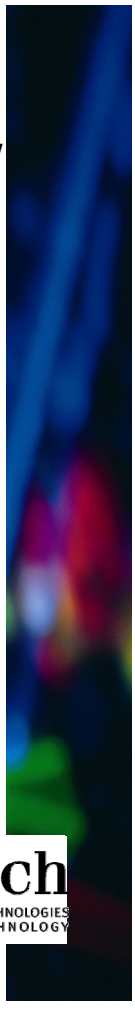
Nous commençons par intégrer certaines des techniques d'apprentissage profond les plus récentes (comme dropout, batch normalization et différentes fonctions d'activation) dans les réseaux de neurones à convolution et obtenons des meilleures performances sur le fameux jeu de données MNIST. Nous proposons ensuite des réseaux TSTN (Tied Spatial Transformer Networks), une variante des réseaux STN (Spatial Transformer Networks) avec poids partagés, ainsi que différentes variantes d'entraînement du TSTN. Nous présentons des performances améliorées sur une variante déformée du jeu de données MNIST. Dans un autre travail, nous comparons les performances des réseaux récurrents de neurones Associative Long Short-Term Memory (ALSTM), une architecture récemment introduite, par rapport aux réseaux récurrents de neurones Long Short-Term Memory (LSTM), sur le jeu de données de reconnaissance d'écriture arabe IFN-ENIT. Enfin, nous proposons une architecture de réseau de neurones que nous appelons réseau hybride classique-quantique, capable d'intégrer et de tirer parti de l'informatique quantique. Alors que nos simulations sont effectuées à l'aide du calcul classique (sur GPU), nos résultats sur le jeu de données Fashion-MNIST suggèrent que des améliorations exponentielles en complexité computationnelle pourraient être réalisables, en particulier pour les réseaux de neurones récurrents utilisés pour la classification de séquence.

MOTS-CLEFS: apprentissage profond, calcul quantique, reconnaissance d'écriture manuscrite, réseau de neurones, réseau récurrent de neurones

ABSTRACT: In this thesis, we provide several contributions from the fields of deep learning and quantum computation to handwriting recognition.

We begin by integrating some of the more recent deep learning techniques (such as dropout, batch normalization and different activation functions) into convolutional neural networks and show improved performance on the well-known MNIST dataset. We then propose Tied Spatial Transformer Networks (TSTNs), a variant of Spatial Transformer Networks (STNs) with shared weights, as well as different training variants of the TSTN. We show improved performance on a distorted variant of the MNIST dataset. In another work, we compare the performance of Associative Long Short-Term Memory (ALSTM), a recently introduced recurrent neural network (RNN) architecture, against Long Short-Term Memory (LSTM), on the Arabic handwriting recognition IFN-ENIT dataset. Finally, we propose a neural network architecture, which we name a hybrid classical-quantum neural network, which can integrate and take advantage of quantum computing. While our simulations are performed using classical computation (on a GPU), our results on the Fashion-MNIST dataset suggest that exponential improvements in computational requirements might be achievable, especially for recurrent neural networks trained for sequence classification.

KEY-WORDS: deep learning, quantum computation, handwriting recognition, neural network, recurrent neural network



TELECOM ParisTech

EDITE

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY