# DiagnoseNET: Automatic Framework to Scale Neural Networks on Heterogeneous Systems Applied to Medical Diagnosis

John A García, Frédéric Precioso, Pascal Staccini, Michel Riveill

# DiagnoseNET: Automatic Framework to Scale Neural Networks on Heterogeneous Systems Applied to Medical Diagnosis

John A. García H.[1], Frédéric Precioso[1], Pascal Staccini[2], and Michel Riveill[1]

[1] Université Côte d'Azur, Laboratoire I3S, 06900 SA, France
henao@i3s.unice.fr, frederic.precioso@unice.fr, michel.riveill@unice.fr
[2] Université Côte d'Azur, CHU Nice, 06000 NCE, France
pascal.staccini@unice.fr

**Abstract.** Determine an optimal generalization model with deep neural networks for a medical task is an expensive process that generally requires large amounts of data and computing power. Furthermore, scale deep learning workflows over a wide range of emerging heterogeneous system architecture increases the programming expressiveness complexity for model training and the computing orchestration. We introduce DiagnoseNET, a programming framework designed for scaling deep learning models over heterogeneous systems applied to medical diagnosis. It is designed as a modular framework to enable the deep learning workflow management and allows the expressiveness of neural networks written in TensorFlow, while its runtime abstracts the data locality, micro batching and the distributed orchestration to scale the neural network model from a GPU workstation to multi-nodes. The main approach is composed through a set of gradient computation modes to adapt the neural network according to the memory capacity, the workers' number, the coordination method and the communication protocol (GRPC or MPI) for achieving a balance between accuracy and energy consumption. The experiments carried out allow to evaluate the computational performance in terms of accuracy, convergence time and worker scalability to determine an optimal neural architecture over a mini-cluster of Jetson TX2 nodes. These experiments were performed using two medical cases of study, the former dataset is composed by clinical descriptors collected during the first week of hospitalization of patients in the Provence-Alpes-Côte d'Azur region; the second dataset uses a short ECG records between 30 and 60 seconds, obtained as part of the PhysioNet 2017 Challenge.

## 1 Introduction

Determine an optimal generalization model with Deep Neural Networks (DNN) in healthcare research is an expensive training process, due to the cost of hardware and electricity or the cloud compute time and its carbon footprint required to fuel HPC systems [12]. Therefore, physicians and scientists require alternative High-Performance Computing (HPC) systems to exploit the Artificial Intelligence (AI) methods applied in medical diagnosis within hospitals to accelerate

healthcare research as well as to preserve the patient data privacy with an affordable cost of hardware and electricity. In this context, the motivation of this research is to develop a programming framework to improve the usability, portability and scalability of deep learning workflows over heterogeneous systems and, evaluate low-consumption computing architecture with minimal infrastructure requirements [1], to accelerate clinical-risk-predictive models [10, 14] with an efficient balance between accuracy and energy consumption.

Based on these challenges, this paper analyses the deep learning algorithmic complexity in terms of accuracy, convergence time and worker scalability for training two different neural networks (MLP and CNN) on a mini-cluster with 14 Jetson-TX2 nodes, applied to predict the medical care purpose of hospitalized patients and to atrial fibrillation classification for cardiac arrhythmia diagnosis. Which main contribution is an open-source framework called DiagnoseNET, designed into independent and interchangeable modules for scaling deep learning models over heterogeneous system architecture applied to develop medical risk prediction models. Which increases the developer's productivity, facilitating the programming process to build and finetune a DNN, while its run-time abstracts the data locality, the micro batching and the distributed orchestration to scale the DNN model from a GPU workstation to multi-nodes.

## 2 Background and State of the Art

The main challenge is to minimize the execution time, increase the worker scalability and exploit the computing power of each hybrid processor on-chip (CPU&GPU) with 8GB of host memory capacity by each Jetson TX2. Where the data locality, the communication protocol and the coordination training modes are the key factors for efficient task mapping over the resources but it increases the programming complexity for model training and computing orchestration.

The common data-distributed methods are Bosen [13] and Federated Learning [3,9], the approaches that use iterative-convergent Machine Learning (ML) algorithms for training. They can be applied generically to any ML method if data samples are independent and identically distributed (i.i.d.). The Bosen platform provides a distributed version for a number of well-known ML algorithms (for example, Deep Learning, Sparse Coding, K-means clustering, Random forests or Multi-class Logistic Regression), while Federated Learning is designed to be efficient in setups with a large number of users and unreliable or slow connections. Final classification or prediction models represent a weight matrix that is stored across a large number of clients. Local weight matrix is calculated in the initial step and refined over the rounds, where updates are based on the exchange of parameters with local neighbors or a single master node.

Model-distributed approaches such as Strads platform [13] require ML specialized systems that perform a partition of ML algorithms into a set of parallel tasks, in general scheduled by master node(s) and executed by a set of workers. Schedulers' task is to separate the problem into a non-overlapping set of subproblems, divide a workload and synchronize the updates amongst the workers.

This setup admits non-conflicting model updates that lead to convergence. Numerous algorithms can be deployed in this framework, such as Latent Dirichlet Allocation, Matrix Factorization, Support Vector Machine or Deep Learning algorithm based on Caffe, called Poseidon, to name a few.

Model and data-distributed algorithms for classification and prediction problems. In the literature, there exist only a few works. A hybrid distributed platform known as Angel [6] appropriately combines data partitioning, scheduling and parameter synchronization tasks and demonstrates accuracy improvement in comparison with a Petuum-based data or model distribution. There exist a number of calculus-parallelization methods, such as FlexFlow [5]. It is a hybrid data and model parallel (non-distributed) approach worth of exploring in a distributed setup, because it performs automated search of parallelization strategies that incorporates data, attribute, parameter and operator parallelization for DNN algorithms.

## 3 Material and Methods

### 3.1 DiagnoseNET

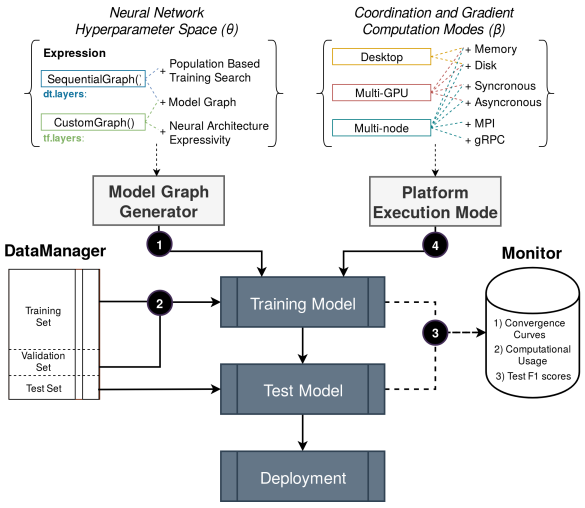



Fig. 1: DiagnoseNET framework scheme.

DiagnoseNET was designed to harmonize the deep learning workflow and to automatize the distributed orchestration to scale the neural network model from a GPU workstation to multi-nodes. Figure 1 shows the schematic integration of the DiagnoseNET modules with their functionalities. The first module is the deep learning model graph generator, which has two expression languages: a

*Sequential Graph* API designed to automatize the hyperparameter search and a *Custom Graph* which support the TensorFlow expression codes for sophisticated neural networks. The second module is the data manager, compose by three classes designed for splitting, batching and multi-task any dataset over GPU workstations and multi-nodes computational platforms. The third module extends the enerGyPU monitor for workload characterization, constitute by a data capture in runtime to collect the convergence tracking logs and the computing factor metrics; and a dashboard for the experimental analysis results [7]. The fourth module is the runtime that enables the platform selection from GPU workstations to multi-nodes whit different execution modes, such as synchronous and asynchronous coordination gradient computations with gRPC or MPI communication protocols.

**DiagnoseNET Model Graph Generator,** In *Sequential Graph* the first step, defines the stacked layers and sets the type of each layer, their neurons numbers, the number of layers and followed by a linear output on top, since the cross entropy will be used as loss function and include the softmax function. Then the neural network hyperparameters are defined as shown in the expression 1. to generate the model graph object. *Custom Graph* uses *tf.layers* to defines the staked layers and the similar expression as the former is used to define the optimizer and loss function for generating the model graph object.

Code Example 1.1: Model definition to generate several graphic-model objects.

```
import diagnosenet as dt
stacked_layer_1 =  [dt.Relu(14637, 2048),
                    dt.Relu(2048, 1024),
                    dt.Relu(1024, 1024),
                    dt.Linear(1024, 14)]

model_1 = dt.sequentialGraph(
                input_size=14637, output_size=14,
                layers=stacked_layer_1,
                loss=dt.CrossEntropy,
                optimizer=dt.Adam(lr=0.001))
```

**DiagnoseNET Data Manager,** This manages the dataset according to the computational architecture, creating an isolated sandbox for each dataset and its transformations in the training process to guarantee the data location. In which, the dataset is splitting into well balance batches over the number of workers, and its worker-batch is micro batching according to the memory or parameter, as shown in the following code expressions:

Code Example 1.2: Dataset splitting and micro-batching over the workers.

```
data_config_1 = dt.Batching(
                dataset_name="medical_D1", valid_size=0.05,
                devices_number=4, batch_size=128)
```

**DiagnoseNET for Distributed Training with gRPC,** It harmonizes the computational resources with the dataset manager to train previously defined models over a multi-node platform, automating the gRPC communication protocol to coordinate the workers with asynchronous gradient computations. In which, the resource manager divide the dataset equally onto the workers nodes of the system where each worker has a copy of the neural network (graph) along with its local weights. Each worker operates on a unique subset of the dataset and updates its local set of weights. These local weights are shared across the cluster to compute a new global set of weights through an accumulation algorithm.

Code Example 1.3: Distribted orchestration with GRPC asynchronous.

```
import diagnosenet as dt
dt.between_graph_replication(
        d_replica_path=/myworkspace,
        d_replica_name="GRPC_replica.py",
        ip_ps="host1",
        ip_workers="host2,host3,host4,host5",
        num_ps=1, num_workers=4)
```

On the side of the replica script, is gives the model graph object, create the dataset batching, and pass both of these to a *Distributed_GRPC* object. This object is responsible for launching the experiment, through its function *asynchronous_training*.

Code Example 1.4: GRPC asynchronous replica.

```
platform = dt.Distibuted_GRPC(
        model=model_1,
        datamanager=data_config_1,
        monitor=enerGyPU(machine_type="arm"),
        max_epochs=20,
        ip_ps=argv[0], ip_workers=argv[1])

platform.asynchronous_training(
        dataset_path=/myworkspace/datasetpath,
        inputs_name="X.npy", targets_name="Y.npy",
        job_name=argv[0], task_index=argv[1])
```

**DiagnoseNET for Distributed Training with MPI,** DiagnoseNET implements synchronous and asynchronous MPI methods to improve performance in the communication between workers. For example, asynchronous gradient updates were optimized with a parameter called weighting, which is responsible to determine the number of workers required in each step to compute the new weights and broadcast it.

Code Example 1.5: MPI Platform Execution Modes.

```
platform = dt.Distibuted_MPI(
        model=model_1, datamanager=data_config_1,
```

```
            monitor=enerGyPU(machine_type="arm"),
            max_epochs=20, early_stopping=3])

    platform.asynchronous_training(
            dataset_name="medical_D1",
            dataset_path=d/myworkspace/datasetpath,
            inputs_name="X.npy", targets_name="Y.npy",
            weighting=1)
```

The specifications of the MPI algorithms are described in the Appendices A, B.

## 4 Case studies and Neural Architectures

**Medical Care Purpose Classification for Inpatients:** The clinical dataset was derived from a program for medicalisation of information systems (PMSI) collection of synthetic medical information in a standardized and anonymized format from hospitalizations that carried out in activities in medical care or rehabilitation settings. In which the patient-feature composition module was used to generate the representations of the patients status in the first week of hospitalization using from one-year of the PMSI data collection. The main clinical descriptors used was demographics, admission details, hospitalization details, physical dependence, cognitive dependence, rehabilitation time, comorbidities, morbidity and etiology. The clinical dataset obtained has 116.831 different inpatients and 14.637 clinical-features embedded in a document-term sparse matrix [4]. In this paper we worked with the high-level group called Clinical Major Category (CMC) obtaining 14 labels-categories to classify the medical care of patients hospitalized as shown in Table 1.

Table 1: Medical Target 1: Care Purpose Description Labels.

| Class | Labels Description | Train | Valid | Test |
|---|---|---|---|---|
| 0 | Other Situations | 4489 | 267 | 515 |
| 1 | Proceedings of Medical Cardiovascular / Respiratory Care | 18299 | 1122 | 2263 |
| 2 | Circulatory system disorders | 13074 | 764 | 1504 |
| 3 | Procedings of Neuro-Muscular Medical Care | 5375 | 295 | 640 |
| 4 | Procedings of Medical Care Mental Health | 2929 | 175 | 344 |
| 5 | Procedings Sensory and Skin Medical Care | 8273 | 456 | 950 |
| 6 | Procedings of Rheumatics / Orthopedic Medical Care | 18080 | 1061 | 2106 |
| 7 | Procedings of Post-Traumatic Medical Care | 14174 | 801 | 1619 |
| 8 | Proceedings of Medical Amputations | 741 | 45 | 89 |
| 9 | Palliative Care | 2056 | 114 | 256 |
| 10 | Placement Expectation | 299 | 20 | 40 |
| 11 | Rehabilitation | 2261 | 114 | 268 |
| 12 | Procedings of Nutritional Medical Care | 9240 | 415 | 1144 |
| 13 | No grouping | 16 | 1 | 3 |

**Atrial Fibrillation Classification for Cardiac Diagnosis:** The ECG dataset was obtained from the 2017 PhysioNet Challenge The dataset was already labeled and the four labels are: Normal, Atrial Fibrillation, Others and Noisy. The Others label means recordings of those similar heart diseases. The total number of source dataset is $8,528$. Each sample is a single short ECG lead recording. Since the length of the sample is inequivalent, samples are transformed into structured input. The position of the peaks $R$ of recordings are extracted to get the centred windows of 260 time steps, which are complete ECG rhythms for a cycle. To better represent the behaviour of the recording, each five consecutive centred windows are concatenated into a training sample as shows in the following table 2.

Table 2: Medical Target 2: Cardiac Arrhythmia Labels

| Class | Label Description | Source dataset | Training Dataset | Small Samples |
|---|---|---|---|---|
| 0 | Normal | 5,050 | 34,303 | 4241 |
| 1 | AF | 738 | 6,542 | 815 |
| 2 | Others | 2,456 | 18,986 | 2424 |
| 3 | Noisy | 284 | 1,382 | 171 |

**Multilayer Perceptron Network:** In DiagnoseNET the network architecture was composed dynamically through fully-connected layers, each neuron is connected to all neurons of the previous layer building a stacked neural network and followed by a softmax layer on top $h_i = f(\sum_{j=1}^{n} w_{ij}x_j + b_{ij})$, where $x_j$ is the output of the previous layer and $w_{ij}$ is the weight value associated with $x_j$ with a bias associated $b_{i,j}$ and $n$ is the number of neurons in the previous layer, while $f$ is the as activation function. Having as a baseline the neural network used in the work called *improving palliative care with deep learning* [2] and after finetune it to classify the medical care purpose with PACA inpatients as shown in the Appendix D. The model used to evaluate the scalability was comprised by an input (of 10,833 dimensions), 4 hidden layers (each 512 dimensions) and a softmax output layer. As activation function was used rectified linear unit (ReLU), as loss function was used categorical cross-entropy and Adam as optimizer [8].

**Convolutional Neural Network:** The neural network baseline for the second medial task is based on a Convolutional Neural Network (CNN) designed to take as input the time-series of ECG signal and generates the sequence of label predictions as outputs [11]. The general neural architecture is composed using DiagnoseNET with 75 layers of convolution followed by a fully-connected layer and a softmax layer on top, as shows in the Appendix D. The major elements in the CNN model are the residual network, the convolutional layers and the regularization methods, such as batch normalization, dropout and activation

which are used to improve the performance and regularization of the CNN model. The convolutional layers are used in order to extract features relative to the form of the traces wave.

## 5 Experiments and Results

The experiments were conducted using the DiagnoseNET self expression codes for training the medical care purpose classification, and for training the atrial fibrillation classification. This implementations was processing over a set of gradient computation modes as synchronous or asynchronous for each communication protocol GRPC or MPI.

**HPC System and Enviroment:** For processing the distributed experiments were built a mini-cluster of 14-nodes Jetson TX2 interconnected by 1 $GigE$ switch Ethernet. The nodes are identical, independent machines and each one runs a separate OS. Every node is composed of one developer kit Jetson TX2, which contains a hybrid processor Nvidia Denver with one ARM Cortex-A57 quad-core with one a Pascal GPU 256-$CUDA@cores$ with a maximun, it has $8GB$ of LPDDR4 memory, $59.7GB/s$ of memory bandwidth and $32GB$ of internal storage.

**Worker Scalability for Training the Medical Task 1:** The baseline got 11.04 hours as convergence time for training the MLP model described in the previous session, which was performed using gRPC asynchronous to coordinate and compute the gradient updates between 2 workers and 1 master. As shown in the Figure fig:worker-scalability, the best setting reduces the convergence time to 1.3 hours, using MPI asynchronous to coordinate and compute the gradient updates between 12 workers and 1 master.

(a) Convergence Time (HRS) (b) Speedup Scalability (Strong)

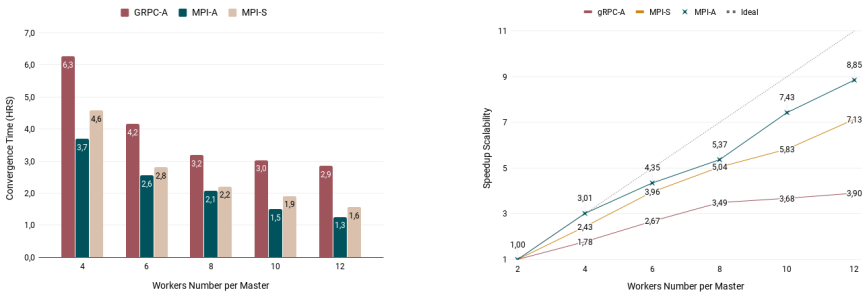
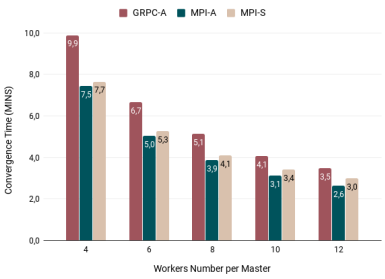



Fig. 2: Worker scalability comparison for distributed training on a mini-cluster of Jetson TX2 to classify the medical care purpose.

**Worker Scalability for Training the Medical Task 2:** For the atrial fibrillation classification was used a small dataset (77MB) with 8.528 patients and a medium model with 72 layers fully-connected of convolutional neural network with residual network connections. Where the baseline uses a gRPC asynchronous training modes with 4 workers take 13 minutes as a time to solution achieving one accuracy of 0,63 F1-score, while the MPI asynchronous training modes with 12 workers take 5 minutes as a time to solution achieving the same accuracy of 0,63 F1 score.

(a) Convergence Time (MINS)




(b) Speedup Scalability (Strong)


Fig. 3: Worker scalability comparison for distributed training on a mini-cluster of Jetson TX2 to the classify atrial fibrillation.

## 6 Conclusions

DiagnoseNET increases developer's productivity facilitating the programming process to build and finetune Deep Learning workflows, while its runtime abstracts the data locality and the distributed orchestration to scale each model from a GPU workstation to multi-nodes. Furthermore, implement a mini-cluster of Jetson TX2 nodes presents a good scalability for distributed training of each neural network by their medical task. Therefore, clusters with embedded computation platforms can be used as a deep learning platform system with minimal infrastructure requirements and low power consumption, offer the computing capacity for processing considerable datasets and models in the HPDA ecosystem. In the way to characterize the deep learning tasks and improve the balance between accuracy, convergence time and worker scalability, MPI asynchronous gradient computations with data parallelism offer an efficient distributed neural network training for early convergence. Likewise adapting the number of records by batch and the model dimensionality helps to minimize the bottleneck of data transfer from host memory to device memory reducing the GPU idle status.

## References


1. Asch M., Moore T. et al.: Big data and extreme-scale computing: Pathways to Convergence-Toward a shaping strategy for a future software and data ecosystem for scientific inquiry. The International Journal of High Performance Computing Applications **32**, 435–479 (2018)
2. Avati, A., Jung, K., Harman, S., Downing, L., Ng, A.Y., Shah, N.H.: Improving Palliative Care with Deep Learning. CoRR **abs/1711.06402** (2017), http://arxiv.org/abs/1711.06402
3. Bonawitz, K., Eichner, H., Grieskamp, W., Huba, D., Ingerman, A., Ivanov, V., Kiddon, C., Konecný, J., Mazzocchi, S., McMahan, H.B., Overveldt, T.V., Petrou, D., Ramage, D., Roselander, J.: Towards Federated Learning at Scale: System Design. CoRR **abs/1902.01046** (2019)
4. Garcia Henao, J.A., Precioso, F., Staccini, P., Riveill, M.: Parallel and Distributed Processing for Unsupervised Patient Phenotype Representation. In: Latin America High Performance Computing Conference (Sep 2018), https://hal.archives-ouvertes.fr/hal-01885364
5. Jia, Z., Zaharia, M., Aiken, A.: Beyond Data and Model Parallelism for Deep Neural Networks. CoRR **abs/1807.05358** (2018), http://arxiv.org/abs/1807.05358
6. Jiang, J., Yu, L., Jiang, J., Liu, Y., Cui, B.: Angel: a new large-scale machine learning system. National Science Review **5**(2), 216–236 (02 2017). https://doi.org/10.1093/nsr/nwx018, https://doi.org/10.1093/nsr/nwx018
7. John A. Garcia H., E.H.B.C.E.M.P.O.N.C.J.B.H.: enerGyPU and enerGyPhi Monitor for Power Consumption and Performance Evaluation on Nvidia Tesla GPU and Intel Xeon Phi (2016)
8. Kingma, D.P., Ba, J.: Adam: A Method for Stochastic Optimization. arXiv e-prints arXiv:1412.6980 (Dec 2014)
9. Konecný, J., McMahan, H.B., Yu, F.X., Richtárik, P., Suresh, A.T., Bacon, D.: Federated Learning: Strategies for Improving Communication Efficiency. CoRR **abs/1610.05492** (2016)
10. Maharlou, H., Niakan Kalhori, S.R., Shahbazi, S., Ravangard, R.: Predicting Length of Stay in Intensive Care Units after Cardiac Surgery: Comparison of Artificial Neural Networks and Adaptive Neuro-fuzzy System. Healthcare informatics research **24**(2), 109—117 (April 2018). https://doi.org/10.4258/hir.2018.24.2.109, http://europepmc.org/articles/PMC5944185
11. Rajpurkar, P., Hannun, A., Haghpanahi, M., Bourn, C., Y. Ng, A.: Cardiologist-Level Arrhythmia Detection with Convolutional Neural Networks (07 2017)
12. Strubell, E., Ganesh, A., McCallum, A.: Energy and Policy Considerations for Deep Learning in NLP. arXiv e-prints arXiv:1906.02243 (Jun 2019)
13. Xing, E.P., Ho, Q., Dai, W., Kim, J.K., Wei, J., Lee, S., Zheng, X., Xie, P., Kumar, A., Yu, Y.: Petuum: A New Platform for Distributed Machine Learning on Big Data. IEEE Transactions on Big Data **1**(2), 49–67 (2015)
14. Ye Chengyin, Wang Oliver, Liu Modi, Zheng Le, Xia Minjie, Hao Shiying, Jin Bo, Jin Hua, Zhu Chunqing, Huang Chao Jung, Gao Peng, Ellrodt Gray, Brennan Denny, Stearns Frank, Sylvester Karl G, Widen Eric, McElhinney Doff B, Ling Xuefeng: A Real-Time Early Warning System for Monitoring Inpatient Mortality Risk: Prospective Study Using Electronic Medical Record Data. Journal of medical Internet research **21**(7), e13719–e13719 (jul 2019), https://www.ncbi.nlm.nih.gov/pubmed/31278734https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6640073/

## Appendix A

# DiagnoseNET MPI Synchronous Algorithm

The algorithm 1 describes the MPI synchronous coordination training with parameter server. It uses the nodes ranks to assign them the role of parameter server or worker, defined the rank 0 as parameter server (PS) and the other ranks as workers. When launching the program, the PS does necessary pre-processing tasks, such as loading the dataset and compiling the model. After these tasks, the PS sends the model to the workers, which are ready to receive it. At each training step, the PS sends a different subset of the data to every worker to be used for loss optimization. At the end of an epoch, the PS will gather the new weights from every worker. Workers receive the collection of weights and compute the average weight for the global update. For the other computing parts, it works as the desktop version.

**Algorithm 1** Synchronous MPI Kernel

```
while ConvergenceCondition do
    if master True then
        for all worker ∈ workers do
            masterGrads ← received(workerGrads)
        averageGrads ← average(masterGrads)
        send(averageGrads)
    else
        workerGrads ← compute(model, batches)
        send(workerGrads)
    if master True then
        for all worker ∈ workers do
            masterLoss ← received(workerLoss)
        averageLoss ← average(masterLoss)
        if overfitting(averageLoss) True then
            send(averageLoss, earlyStopping)
        else
            send(averageLoss, False)
    else
        workerWeights ← received(masterWeights)
        projection ← model.Apply(workerWeights)
        workerLoss ← computeLoss(projection, labels)
        send(workerLoss)
```

## Appendix B

# DiagnoseNET MPI Asynchronous Algorithm

The algorithm 2 allows training multiple model replicas in parallel on different nodes with different subsets of the data. Each model replica processes a mini-batch to compute gradients and sends them to the parameter server which apply a function (mean, weighted average) between previous and received weights, then updates the global weights accordingly and send them back to the workers. In fact, every worker will compute its gradients individually until its convergence; the convergence occurs when we start having overfitting, which means that the training loss is decreasing while the validation loss increased. The master who is responsible for computing the weighted average of received weights and its own weights, will stop when all workers converge. To check the status of convergence of workers, the master has a queue that stores converged workers and when its length is equal to the number of workers, the master knows that all workers converged and stops training. Since each node computes gradients independently and does not require interaction among each other, they can work at their own pace and have greater robustness to machine failure.

**Algorithm 2** Asynchronous MPI Kernel

**while** $ConvergenceCondition$ **do**
  **if** $master\ True$ **then**
    $convergeFlag \leftarrow received(workerCond)$
    $masterGrads \leftarrow received(workerGrads)$
    $collectGrads \leftarrow collection(masterGrads)$
    $averageGrads \leftarrow average(collectGrads)$
    $send(averageGrads)$
  **else**
    **if** $overfitting(averageLoss)\ True$ **then**
      $send(averageLoss, earlyStopping)$
    **else**
      $send(averageLoss, False)$
    **if** $decrease(averageLoss)\ True$ **then**
      $send(Updated(masterWeights))$
    $workerGrads \leftarrow compute(model, workerInput)$
    $send(workerGrads)$
  **if** $master\ False$ **then**
    $workerWeights \leftarrow received(masterWeights)$
    $projection \leftarrow model.Apply(workerWeights)$
    $workerLoss \leftarrow computeLoss(projection, labels)$

## Appendix C

# Hyperparameter Search to Classify the Medical Task 1

A model space contains ($d$) hyperparameters and ($n$) hyperparameters configurations defined in Table 3 and the Table 4 shows the models by number of parameters. We have established some fixed hyperparameters and decided to tune the number of units per layer, the number of layers and batch size, which are the hyperparameters that directly affect the computational cost. Each model was trained using *Adam* as an optimizer with a maximum of 40 epochs and as a loss function is used the *Cross Entropy*.

Table 3: Search Space Model Descriptors.

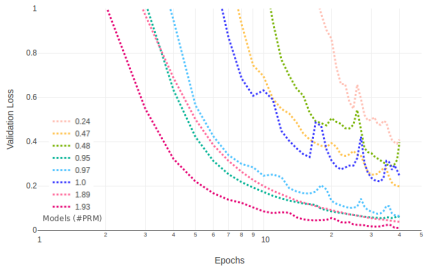
| Hyperparameters ($d$) | Hyper. Configurations ($n$) | State |
|---|---|---|
| Learning rate | 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1 | Fixed: 0.001 |
| Activation function | relu, tanh, linear | Fixed: relu |
| Num. Units per layer | 16, 32, 64, 128, 256, 512, 1024, 2048, 4096 | Search |
| Num. hidden layers | 2, 4, 8, 16 | Search |
| Regularization | Dropout: 0.6, 0.7, 0.8 | Fixed: 0.8 |
| Batch size | 24.576, 12.288, 6.144, 3.072, 1.536, 768 | Search |
| Num. of workers | 4, 6, 8, 10, 12 | Search |

Table 4: Model Dimension Space in Number of Parameters (millions).

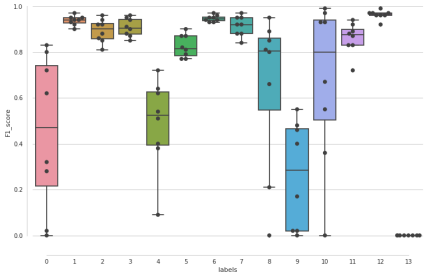
| | | Numbers of layers | | | |
|---|---|---|---|---|---|
| | | 2 | 4 | 8 | 16 |
| Neurons by Layer | 16 | | | | 0.24 |
| | 32 | | | 0.47 | 0.48 |
| | 64 | | 0.95 | 0.97 | 1.0 |
| | 128 | 1.89 | 1.93 | 1.99 | 2.12 |
| | 256 | 3.82 | 3.95 | 4.21 | 4.74 |
| | 512 | 7.76 | 8.29 | 9.34 | 11.44 |
| | 1024 | 16.05 | 18.15 | 22.35 | |
| | 2048 | 34.2 | 42.6 | | |
| | 4096 | 76.8 | | | |

According with the model dimension showed in the Table 4, we are found that is possible divided the models by Fine, middle and course grain. In which, the Figure 4 shows that middle-grain models from 1.99 to 8.29 millions of parameters have a fast convergence in validation loss, and high accuracy levels for the majority of the 14 care purpose labels, in comparison with the other models who present a great variation in accuracy and spent more epochs to convergence.

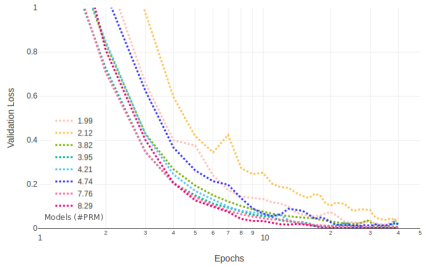
(a) Fine-grain convergence validation.



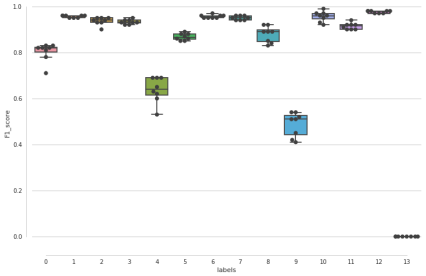

(b) Fine-grain test prediction by class.



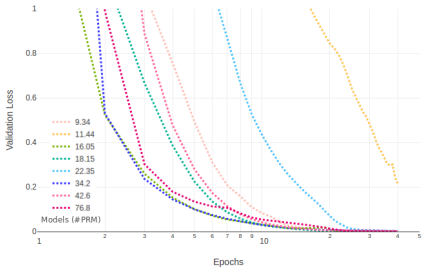

(c) Middle-grain convergence validation.




(d) Middle-grain test prediction by class.




(e) Course-grain convergence validation.




(f) Course-grain test prediction by class.

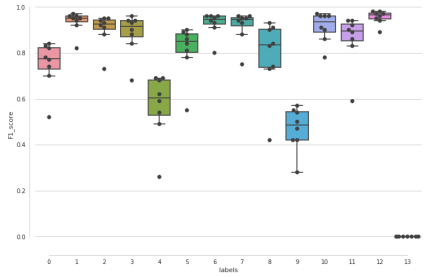



Fig. 4: Experiment results for training a feed-forward neural network, using the hyperparameter model-dimension space.

# Appendix D

# ECG Neural Architecture to Classify the Medical Task 2

The pure CNN model leads to the problem that the last layer of the model may not exploit the original features or the ones extracted in the first layers. The Figure 5 shows the ECG neural architecture implemented using DiagnoseNET framework, which key architecture factor are the residual network connections to solve the information loss problem into the deep layers. To implement this, a second information stream is added in the model. In this way, deeper layers have access to the original features, in addition to the information processed by the previous layers. What else, two different types of residual block are included to access the different states of the information. The normal residual block preserves the size of the input while the sub-sampling residual block lowers the size of the input down to a half. By using max pooling, the network extracts only the high values from an input so that the size of its output is halved.

Fig. 5: ECG Convolutional Neural Architecture.