HAL open science

Hcore-Init: Neural Network Initialization based on Graph Degeneracy

Stratis Limnios, George Dasoulas, Dimitrios M. Thilikos, Michalis Vazirgiannis

To cite this version:

Stratis Limnios, George Dasoulas, Dimitrios M. Thilikos, Michalis Vazirgiannis. Hcore-Init: Neural Network Initialization based on Graph Degeneracy. ICPR 2020 - 25th International Conference on Pattern Recognition, Jan 2021, Milan (Virtual), Italy. pp.5852-5858, 10.1109/ICPR48806.2021.9412940 . hal-03002744v1

HAL Id: hal-03002744

https://hal.science/hal-03002744v1

Submitted on 20 Nov 2020 (v1), last revised 24 May 2022 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Hcore-Init: Neural Network Initialization based on Graph Degeneracy

Stratis Limnios
École Polytechnique
Palaiseau, France

George Dasoulas
École Polytechnique &
Noah's Ark Lab, Huawei Technologies
Paris, France

Dimitrios M. Thilikos
LIRMM, Univ Montpellier, CNRS
Montpellier, France

Michalis Vazirgiannis
École Polytechnique
Palaiseau, France

arXiv:2004.07636v1 [cs.LG] 16 Apr 2020

***Abstract*—Neural networks are the pinnacle of artificial intelligence, as in recent years we witnessed many novel architectures, learning and optimization techniques for deep learning. As neural networks inherently constitute multipartite graphs among neuron layers, we aim to analyze directly their structure to extract meaningful information from the learning processes. To our knowledge graph mining techniques for enhancing learning in neural networks have not been thoroughly investigated. In this paper we propose an adapted version of the k-core structure for the complete weighted multipartite graph extracted from a deep learning architecture. As a multipartite graph is a combination of bipartite graphs, and since bipartite graphs are the incidence graphs of hypergraphs, we design the k-hypercore decomposition, the hypergraph analogue of the k-core degeneracy. We applied hypercore to several neural network architectures, more specifically to convolutional neural networks and multilayer perceptrons for image recognition after a very short pretraining. Then we used the information provided by the core numbers of the neurons to re-initialize the weights of the neural network to give a more specific direction for the gradient optimization scheme. Extensive experiments proved that hypercore outperforms state-of-the-art initialization methods.**

I. Introduction

During the last decade deep learning has been intensely in the focus of the research community. Its applications on a huge variety of scientific and industrial fields highlighted the need for new approaches at the level of neural network design. Researchers have studied until today different aspects of the Neural Network (NN) architectures and how these can be optimal for various tasks, i.e the optimization method used for the error backpropagation, the contribution of the activation functions between the NN layers or normalization techniques that encourage the loss convergence, i.e batch normalization, dropout layer etc.

Weight initialization is one of the aspects of NN model design that contribute the most to the gradient flow of the hidden layer weights and by extension to the ability of the neural network to learn. The main focus on the matter of weight initialization ([4], [5]) is the observation that weights among different layers can have a high variance, making the gradients more likely to explode or vanish. Neural Networks capitalize on graph structure by design. Surprisingly there has been very few work analyzing them as a graph with edge and/or nodes attributes. Recent work [16] introduces graph metrics to produce latent representation sets capitalising on bipartite matching directly implemented in the neural network architecture, which proved to be a very powerful method. Also the work by C. Morris analyzes the expressivity of Graph Neural Networks using the Weisfeiler-Leman isomorphism test [11]. Our interest lies in trying to refine the optimization scheme by capitalizing on graph metrics and decompositions. One natural candidate was the k-core decomposition [15]. Indeed this decomposition method, being very efficient ($O(n \log(n))$ in the best cases [2]), performs very well in state-of-the-art frameworks for enhancing supervised learning methods [12]. Providing key subgraphs, and also extracting very good features.

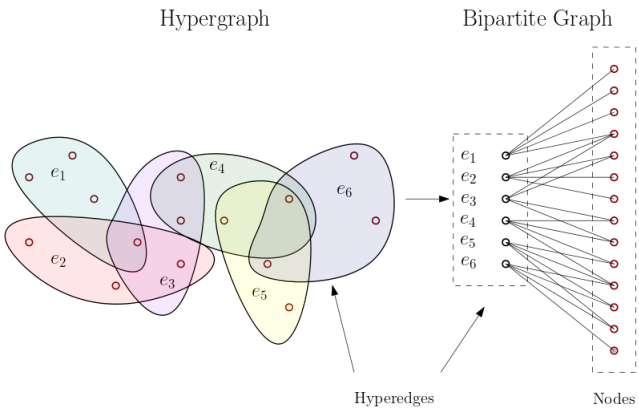

Fig. 1. Hypergraph and the corresponding incidence graph

Unfortunately, in the case of a graph representing a neural network, k-core might lack some features. As a matter of fact, graphs extracted from NNs constitute multipartite complete weighted graphs, in the case of an Multilayer Perceptron and almost complete for Convolutional Neural Networks. As we saw different k-core variants for different types of graphs, such as the k-truss [14] counting triangles, the D-core [3] for directed graphs, were designed this past decade. A natural thought was then to design our own version of the k-core for our precise graph structure.
Hence our contributions are the following:

- We provide a unified method of constructing the graph representation of a neural network as a block composition of the given architecture (see fig. 2). This is achieved

by transforming each part of the network (i.e linear or convolutional layers, normalization/dropout layers and pooling operators) into a subgraph. Having this graph representation, it is possible to apply different types of combinatorial algorithms to extract information from the graph structure of the network.

- Next we design a new degeneracy framework, namely the k**-hypercore**, extending the concept of k-core to bipartite graphs by considering that each pair of layers of the neural network, constituting a bipartite graph, is the incidence graph of a hypergraph (cf. fig.1).
- we propose a novel weight initialization scheme, **Hcore-init** by using the information provided by the weighted version of the k**-hypercore** of a NN extracted graph, to re-initialize the weights of the given neural network, in our case, a Convolutional neural network and a Multilayer Perceptron. Our proposal clearly outperforms traditional initialization methods on classical deep learning tasks.

The rest of this paper is organized as follows, first some preliminary definitions and overview of the state of the art methods in neural network initialization methods. Then we provide the methodology which allows us to transform neural networks to edge weighted graphs. Further on, we proceed to the main contribution of the paper being the definition of the hypercore degeneracy and the procedure which produces our initialization method. Finally we test our method on several image classification datasets, comparing it the main initialization method used in neural networks.

II. Preliminaries

In deep neural networks, weight initialization is a vital factor of the performance of different architectures [10]. The reason is that an appropriate initialization of the weights of the neural network can avert the explosion or vanishing of the layer activation output values.

A. Initialization methods

1) Xavier Initialization: One of the most popular initialization methods is Xavier initialization [4]. According to that, the weights of the network are initialized by drawing them from a normal distribution with $E[W] = 0$ and $\mathsf{Var}(w_i) = \frac{1}{\text{fanin}}$, where fanin is the number of incoming neurons. Also, more generally, we can define variance with respect to the number of outgoing neurons as: $\mathsf{Var}(w_i) = \frac{1}{\text{fanin}+\text{fanout}}$, where fanout is the number of neurons that the output is directed to.

2) Kaiming Initialization: Although Xavier initialization method manages to maintain the variance of all layers equal, it assumes that the activation function is linear. In most of the cases of non-linear activation function that Xavier initialization is used, the hyperbolic tangent activation is employed. The need for taking into account the activation function for the weight initialization led to the Kaiming Initialization [5]. According to this method, in the case that we employ *ReLU* activation functions, we initialize the network weights by drawing samples from a normal distribution with zero mean: $E[W] = 0$ and variance that depends on the order of the layer: $\mathsf{Var}[W] = \frac{2}{n^l}$, where l is the index of the l-th layer.

One main assumption for weight initialization is that the mean of the random distribution used for initialization needs to be 0. Otherwise, the calculation of the variances presented above could not be done and we won't be able to have a fixed way to initialize the variance.
Since in our work we want to capitalize on the k-hypercore decomposition to *bias* those distributions we will have to face the fact that we might not be able to control the variance of the weights we initialize. Thankfully the fact that the initial distribution has 0 mean will ensure that our method respects as well this condition on every layer of the neural network.

Moreover, since the k-hypercore decomposition is defined over hypergraphs, let us remind some properties between a hypergraph and a bipartite graph.

B. Hypergraphs and Bipartite graphs

A hypergraph is a generalization of graph in which an edge can join any number of vertices. It can be represented and we keep this notation for the rest of the paper as $\mathcal{H} = (V, E_{\mathcal{H}})$ where V is the set of nodes, and $E_{\mathcal{H}}$ is the set of hyperedges, i.e. a set of subsets of V. Therefore $E_{\mathcal{H}}$ is a subset of $\mathcal{P}(V)$. Moreover a bipartite graph is the incidence graph of a hypergraph [13]. Indeed, a hypergraph $\mathcal{H}$ may be represented by a bipartite graph $\mathcal{G}$ as follows: the sets X and E are the partitions of $\mathcal{G}$, and (x_1, e_1) are connected with an edge if and only if vertex x_1 is contained in edge e_1 in $\mathcal{H}$. Conversely, any bipartite graph with fixed parts and no unconnected nodes in the second part represents some hypergraph in the manner described above. Hence, we can consider that every pair of layers in the neural network can be viewed as a hypergraph, where the left layer represent the hyperedges and the right the nodes (see fig.1).

III. Graph characterization of neural network

We will now describe how we transpose the two classic neural network architectures we investigate, to graphs, and more specificaly bipartite ones. Also, from now on, we are going to refer to a fully-connected neural network as FCNN and to a convolutional neural network as CNN [8].

A. Fully-Connected Neural Networks

Let a FCNN $\mathcal{F}$ with L hidden layers, n_i, $i = 1, .., L$ number of hidden units per layer and $W_i \in \mathbb{R}^{n_i, n_{i+1}}$ the weight matrix of the links between the units of the layers i and $i + 1$.
We define the graph $G_{\mathcal{F}} = (V, E, W)$ as the graph representation of the FCNN $\mathcal{F}$, where the set of nodes V corresponds to the $\sum_{i=1}^{L} n_i$ number of hidden units of $\mathcal{F}$, the set of edges E contains all the links of unit pairs across the layers of $\mathcal{F}$ and the edge weight matrix W corresponds to the link weight matrices $W_i, i = 1, ..., L - 1$. We note that the graph representation $G_{\mathcal{F}}$ does not take into account any activation functions σ used in $\mathcal{F}$.

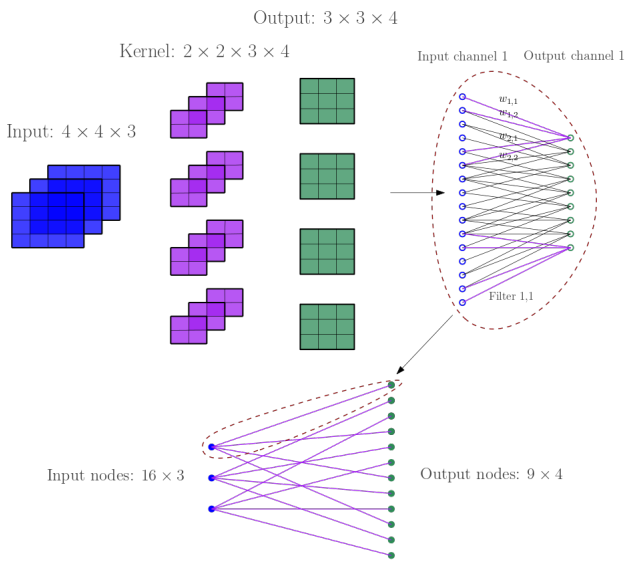

Fig. 2. Illustration of the transformation of a CNN to graph.

Remark. It is easy to see that G_F is a k-partite graph (i.e a graph whose vertices can be partitioned into k independent sets) and more specifically a union of $L-1$ complete bipartite graphs.

B. Convolutional Neural Networks

After showing the correspondence between a FCNN $\mathcal{F}$ and its graph representation $G_{\mathcal{F}}$, we are ready to define the graph representation of a CNN layer. Let a CNN layer $\mathcal{C}$. The convolutional layer is characterized by the input information that has I input channels where each channel provides $n \times n$ features (i.e an 24×24 image characterized by the 3 RGB channels), the output information that has O output channels, where each channel has $m \times m$ features and the matrix of the convolutional kernel $F \in \mathbb{R}^{w \times h \times I \times O}$, where w, h are the width and height of the kernel.
In order to define the graph $G_{\mathcal{C}} = (V, E, W)$ as the graph representation of the CNN $\mathcal{C}$, we have to flatten the 3 and 4-dimensional input, output, and filter matrices correspondingly. Specifically, the $G_{\mathcal{C}}$ is a bipartite graph, where the first partition of nodes P_1 is the flattened input information of the CNN layer ($|P_1| = I \times n \times n$) , the second partition of nodes P_2 is the flatten output information ($|P_2| = O \times m \times m$).

IV. Weight initialization based on Hcore

As degeneracy frameworks have proven to be very efficient at extracting influential individuals in a network [1], why not using the structural information provided by the hcore decomposition of the network to identify "influential" neurons.

Having a neural network graph, we provide a definition of degeneracy specifically for bipartite graphs, where standard k-core does not provide much relevant information.

Definition 1 (Hypercore)**.** *Given a hypergraph $\mathcal{H} = (V, E_{\mathcal{H}})$ We define the (k, l)-**hypercore** as a maximal connected subgraph of $\mathcal{H}$ in which all vertices have hyperdegree at least k and all hyperedges have at least l incident nodes.*

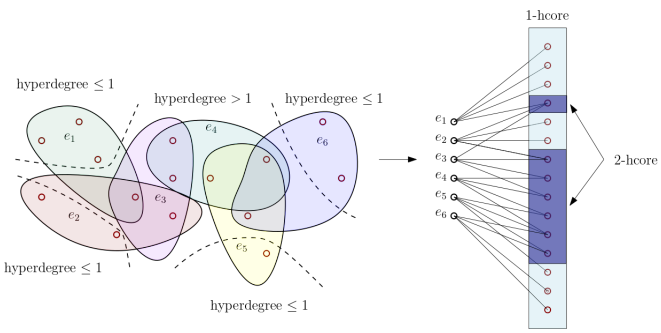

Fig. 3. Example of a k-hcore decomposition of a hypergraph

As for now on, we will refer to the $(k, 2)$-hypercore as the k-hcore and similarly, the *hcore number* of the node will be the largest value of k for which the given node belongs to the k-hcore.

This, henceforth provides a hypergraph decomposition and in our case a decomposition of the right part of the studied bipartite graph, as we do not care about the hcore of the hyperedges (cf. fig. 3).

Since we deal with edge-weighted bipartite graphs, we will use the weighted degree to define the hcore ranking of the nodes given the following weighted-hypercore definition:

Definition 2 (Weighted-hypercore)**.** *Given an edge weighted hypergraph $\mathcal{H} = (V, E_{\mathcal{H}})$, we define the (k, l)-**weighted-hypercore** as a maximal connected subgraph of $\mathcal{H}$ in which all vertices have hyper-weighted-degree at least k and all hyperedges have at least l incident nodes.*

Again, we will refer to the $(k, 2)$-weighted-hypercore as k-WHcore. Now that we have this weighted version, we need to define a way to initialize the weights of the neural network. Indeed, since the WHcore is a value given to the nodes of the network and not the edges, being the weights we aim to initialize. The WHcore shows us which neurons gather the more information, positive on the one hand and negative on the other. After a quick pretraining, we learn the weights just enough to show which neurons have a higher impact on the learning. This information is then grouped by the WHcore into influential neurons and less influential ones.
Moreover, since weights in neural networks are sampled from centered normal law, we have positive and negative weights. Since the hypercore framework operates on positive weighted degrees, we provide two graph representations of the neural network, namely G^+ and G^-. The G^+ graph is built upon the positive weights of the neural network, and the edge weights of the G^- graph are the absolute values of the negative weights of the neural network. Indeed if between neuron x_i and neuron y_j, $w_{ij} > 0$ then we add an edge with weight w_{ij} between node x_i and y_j in graph G^+, otherwise we add an edge with weight $|w_{ij}|$ between node x_i and y_j to graph G^-.

Algorithm 1 Hcore decomposition algorithm

```
procedure HCORE(G, rnodes)
    Input G: bipartite graph, rnodes: right layer nodes
    Output hcore: dictionary of hcore values

    hcore ← dict((node, 0) for node in rnodes)
    tokeep ← rnodes
    while tokeep ≠ ∅ do
        state ← True
        while state == True do
            state ← False
            tokeep ← []
            for node ∈ rnodes do
                if G.degree(node) > k then
                    tokeep.append(node)
                else
                    hcore[node] = k
                    graph.remove[node]
                    state ← True
                end if
            end for
            for node ∈ G.nodes \ rnodes do
                if G.degree(node) = 1 then
                    G.remove(node)
                end if
            end for
        end while
        k ← k + 1
    end while
end procedure
```

Remark. It is important to note that the **WHcore number** of a node is the largest k in which a node is contained in the k-WHcore. Also, the WHcore number of a node is a function of the degree of the node. As the degree depends on the weights, there exists two functions g and h such that $g(W, x)$ outputs the weighted degree of a node x, thus being a linear combination of the weights W. Then $c(W, x) = h(g(W, x))$ is the WHcore number of the node x. For convinience, we now write $c(W^+, x_k) = c_k^+$ where W_k are the positive weights of the weight matrix W.
Moreover, the following initialization schemes are done after a small amount of pretraining of the neural network, in order to have a preliminary information over the importance on the task of the neurons.

A. Initialization of the FCNN

The initialization then is then dependent on the architecture we are looking at, indeed for an FCNN as the graph construction is fairly straight forward we proceed as follows:
For every pair of layers for both positive and negative graphs, we have nodes x_i, with $i \in \{1, \dots, \text{fanin}\}$ in the left side of the bipartite graph, and y_j nodes, with $j \in \{1, \dots, \text{fanout}\}$ nodes on the right side. As for every node y_i we compute their WHcore from the graph G^-, c_j^- and from the graph G^+, c_j^+. Then the given layer weights $w_i j$, are initialized, depending on their sign, with a normal law with expectancy:

- for all i if $w_{ij} \geq 0$, $m = \frac{c_j^+}{\sum_{1 \leq k \leq \text{fanout}} c_k^+}$,
- else $M = \frac{c_j^-}{\sum_{1 \leq k \leq \text{fanout}} c_k^-}$

and with the same variance used in Kaiming initialization. We prove later that the overall mean value of the new random variable obtained in this fashion is 0 as well, justifying the use of the Kaiming variance to be optimal.

B. Initialization of the CNN

For the CNN, since the induced graph is more intricate and the filter weights must follow the same distribution, the initialization framework has to be adapted. We still compute the WHcore on a pair of layers but keeping the filters in mind, the left layer nodes are $x_i^{(k)}$ with $i \in \{1, \dots, n \times n\}$ the input size and $k \in \{1, \dots, I\}$ the number of input filters. Similarly the left layer nodes are $y_j^{(k')}$, where $j \in \{1, \dots, m \times m\}$ the output size, and $k' \in \{1, \dots, O\}$ the output channels. We remind as well that we have two WHcores, one for the positive graph c^+ and one for then negative c^-. Then for a given filter $w^{(k,k')}$ its values are initialized with the following method:

- we define f for a given filter W as $m(W^+) = \frac{1}{H^2} \sum_j c_j^+$ and $m(W^-) = \frac{1}{H^2} \sum_j c_j^-$, if $m(W^+) - m(W^+) > 0$ then $M = m(W^+)$
- else $M = -m(W^-)$.

Using the notations given in the previous remark we can write m in the following general form :

$$m = \text{sign}(\text{argmax}(m(W^+), f(W^-))) \max(m(W^+), m(W^-))$$

where $\text{sign}(W^+) = 1$ and $\text{sign}(W^-) = -1$.

This initialization is done for every filter and with variance given by the Kaiming initialization method. Now we will prove that for the CNN the overall expectancy of the mean value produced is indeed 0.

Proposition 1. *Given a measurable function f and two positive i.i.d. random variables X^+ and X^-, the random variable Z:*

$$Z = \text{sign}\left(\arg\max(f(X^+), f(X^-))\right) \max\left(f(X^+), f(X^-)\right)$$

has mean 0.

Proof. We remind that the function $\mathbb{I}_{\{x \in X\}}$ is the Euler indicator function:

$$\mathbb{I}_{\{x \in X\}} = \begin{cases} 1 & \text{if } x \in X \\ 0 & \text{otherwise.} \end{cases}$$

Let us proceed to evaluate the expectancy of Z provided that X^+ and X^- are i.i.d.:

$$\mathbb{E}[Z] = \mathbb{E}[Z\mathbb{I}_{\{f(X^+) > f(X^-)\}}] + \mathbb{E}[Z\mathbb{I}_{\{f(X^+) \leq f(X^-)\}}] = \mathbb{E}[f(X^+)\mathbb{I}_{\{f(X^+) > f(X^-)\}}] - \mathbb{E}[f(X^-)\mathbb{I}_{\{f(X^+) \leq f(X^-)\}}] = \mathbb{E}[(f(X^+) + f(X^-))\mathbb{I}_{\{f(X^+) > f(X^-)\}}] - \mathbb{E}[f(X^-)].$$

Given the initial assumptions , we can expand the first term $\mathbb{E}[f(X^+)\mathbb{I}_{\{f(X^+)>f(X^-)\}}]$ as follows:

$$\mathbb{E}[f(X^+)\mathbb{I}_{\{f(X^+)>f(X^-)\}}] = \iint f(X^+)\mathbb{I}_{\{f(X^+)>f(X^-)\}} dP(X^+)dP(X^-)$$

As X^+ and X^- follow the same distribution, and f is a measurable function, we use the *Fubini* theorem to intervert the integrals as follows:

$$\mathbb{E}[f(X^+)\mathbb{I}_{\{f(X^+)>f(X^-)\}}] = \iint f(X^-)\mathbb{I}_{\{f(X^-)\geq f(X^+)\}} dP(X^-)dP(X^+).$$

Now replacing this in the original equation gives us:

$$\begin{aligned}
\mathbb{E}[Z] &= \iint f(X^-)\mathbb{I}_{\{f(X^-)\geq f(X^+)\}} dP(X^-)dP(X^+) \\
&\quad + \iint f(X^-)\mathbb{I}_{\{f(X^+)>f(X^-)\}} dP(X^-)dP(X^+) \\
&\quad - \mathbb{E}[f(X^-)] \\
&= \mathbb{E}[(f(X^-)\mathbb{I}_{\{f(X^-)\geq f(X^+)\}}] \\
&\quad + \mathbb{E}[(f(X^-)\mathbb{I}_{\{f(X^+)>f(X^-)\}}] - \mathbb{E}[f(X^-)] \\
&= 0
\end{aligned}$$

This completes our proof that Z is a centered random variable. □

Notice that setting the function $m = l \circ g \circ h$ we can write $l \circ g = f$ and $X = h(W)$. As we defined previously h to be the weighted degree function of a node :

$$h(W_j^+) = \sum_i W_{ij}\mathbb{I}_{\{W_{ij}>0\}}$$

$$h(W_j^-) = \sum_i |W_{ij}|\mathbb{I}_{\{W_{ij}\leq 0\}}$$

which ensures that $h(W^+)$ and $h(W^-)$ follow the same distribution by linear combination of absolute value of the same normal distribution. Replacing these function in the previous proposition, i.e. $f = l \circ g$, $X^+ = h(W^+)$ and $X^- = h(W^-)$ proves that our initialization method has mean 0. This proof allows us to justify the use of the Kaiming variance in our initialization method as it was proven to be the optimal one.

V. Experiments

We will now evaluate our proposed weight initialization method in three standard image classification datasets, CIFAR-10, CIFAR-100, and MNIST and compare it to Kaiming initialization. It is important to note that we do not experiment on state-of-the-art architectures for each dataset. We want to show, as our method can be used separately on different architecture blocks, i.e. only convolutional layers, or only on the FCNN part, or both, that it out performs standard initialization methods, regardless of the refinement of the architecture. Hence in this section, we will evaluate the classification accuracy on the aforementioned datasets with simple CNN architectures presented in this section.

A. Dataset specifications.

The CIFAR-10 and CIFAR-100 are labeled subsets of the 80 million tiny images dataset. They were collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton [7]. We also use the MNIST dataset

- The CIFAR-10 dataset consists of 60000 32×32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.
- The CIFAR-100 is just like the CIFAR-10, except it has 100 classes containing 600 images each. There are 500 training images and 100 testing images per class.

We also test our model on the MNIST database of handwritten digits, which has a training set of 60000 examples, and a test set of 10000 examples. The digits have been size-normalized and centered in a fixed-size image [9].

The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

B. Model setup and baseline.

Next, we present the models that were trained and evaluated for the image classification task. We note that for every case, we compare two scenarios:

1) Initialization of the model with Kaiming initialization [5], training on the train set for 150 epochs and evaluation on the test set.
2) Pretraining of the model (using Kaiming initialization) for N epochs, re-initialization of the model with Hcore-Init, training on the train set for the rest $150-N$ epochs and evaluation on the test set. N has been set as a hyper-parameter.

For the CIFAR-10 and CIFAR-100 datasets, we applied 2 convolutional layers with sizes $3 \times 6 \times 5$ and $6 \times 15 \times 5$ respectively, where 5 is the kernel size and the stride was set to 1. Moreover, after each convolutional layer, we applied two 2×2 max-pooling operators and finally three fully connected layers with corresponding sizes 400×120, 120×84, $84\times$ #classes, where #classes $=$ 10 and 100 respectively for the two datasets. Furthermore, we used *ReLU* as activation function among the linear layers and *tanh* for the convolution layers.

For the MNIST dataset, we applied again 2 convolutional layes of size $1 \times 10 \times 5$ and $10 \times 20 \times 5$, where again the filter size was set to 5 and the stride was set to 1. As in the other dasets, we employed two 2×2 max-pooling operators and we performed dropout [17] on the output of the 2nd convolutional layer with probability $p = 0.5$. Finally, we applied 2 fully

connected layers of size 320×50 and 50×10 and *ReLU* as an activation function throughout the layers.

In all cases, we employed stochastic gradient descent [6] with momentum set to 0.9 and learning rate set to 0.001. As we mentioned before, we chose 2 rather simple models, as we intend to highlight the contribution of Hcore-Init in comparison to its competitor and not to achieve state-of-the-art results for the given datasets, which are exhaustively examined.

C. Settings of the weight initialization.

Next, we present the contribution of Hcore-Init in the performance of the neural network architecture with respect to its application on different types of layers. Specifically, we applied the configurations of the initialization methods (a) exclusively on the set of the linear layers (b) exclusively on the set of the convolutional layers (c) on the combined set of linear and convolutional layers of the model.

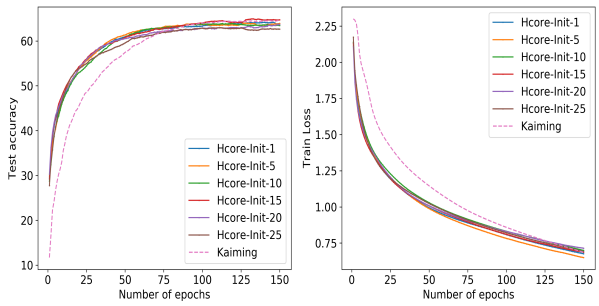

Fig. 4. Test accuracy (left) and train loss (right) on CIFAR-10 for the combined application of the initialization on the linear and the convolutional layers. For the curves Hcore-init-x, x stands for the number of pretraining epochs.

On Figure 4, we can observe that for 15 pretraining epochs, the model initialized with Hcore-Init outperforms the model initialized with Kaiming initialization. It is, also, noteworthy that the loss convergence is faster when applying Hcore-Init. This highlights empirically our initial motivation of encouraging the "important" weights by using the graph information from the model architecture.

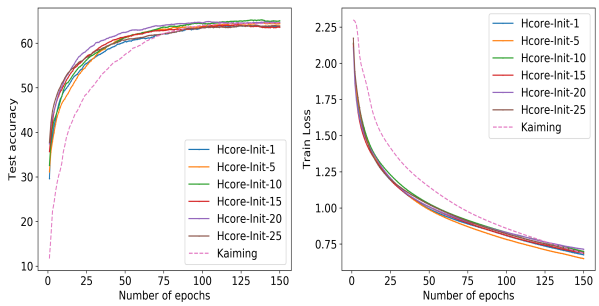

Fig. 5. Test accuracy and train loss on CIFAR-10 for the initialization applied **only** on the linear layers.

On Figures 5 and 6, we can notice the contribution again of Hcore-Init in the performance of the network, when the former is applied on the fully connected and convolutional layers respectively. We can see that in both cases, Hcore-Init with different numbers of pretraining epochs (10 and 20 correspondly) achieves better accuracy results in comparison to Kaiming.

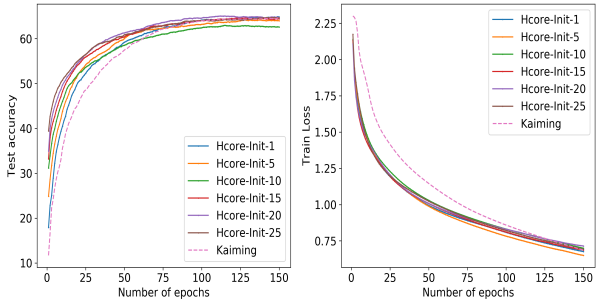

Fig. 6. Test accuracy and train loss on CIFAR-10 for the initialization applied **only** on the convolutional layers.

TABLE I

Top Accuracy results over initializing the full model, only the CNN and only the FCNN for CIFAR-10, CIFAR-100, and MNIST. **Hcore-Init*** represent the top performance over all the pretraining epochs configurations up to 25

	CIFAR-10	**CIFAR-100**	**MNIST**
Kaiming	64.62	32.56	98, 71
Hcore-Init*	**65.22**	**33.48**	**98.91**
Hcore-Init-1	64.91	32.87	98.59
Hcore-Init-5	64.41	32.96	98.70
Hcore-Init-10	**65.22**	33.41	98.81
Hcore-Init-15	64.94	33.45	98.64
Hcore-Init-20	65.05	33.39	98.87
Hcore-Init-25	64.72	**33.48**	**98.91**

Finally, we report the results of the experiments conducted on the 3 datasets in I. Those results correspond to an ablation study over the different number of pretraining epochs as well as the different initialization scenarios, i.e. initializing only on the linear layers, convolutional layers, and the whole architecture. We kept for each mentioned scenario the best performance, and **Hcore-Init*** corresponds to the best overall accuracy. It is important to notice that we do not necessarily need a long pretraining phase to achieve the best results, in fact, only 10 epochs is usually more than enough to outperform in a significant way the Kaiming initialization. We remind that this pretraining corresponds to less than 10% of the total training which is proportional in terms of computation time to 10% of the time to train the model. Furthermore it is interesting to notice that in the early stages of pretraining we are more likely to lose some accuracy as the gradient direction in this stage of the training might be wrong. This justifies as well the consistency of our method.

VI. Conclusion

In this paper, we propose **Hcore-Init**, an initialization method applicable on the most common blocks of neural network architectures, i.e. convolutional and linear layers. This method capitalizes on a graph representation of the neural network and more importantly the definition of hypergraph degenarcy providing a neuron ranking for the bipartite architecture of the neural network layers. Our method, learning with a small pretraining of the neural network, outperforms the standard Kaiming initialization, under the condition that the initialization distribution has zero expectancy. This work is intended to be used as a framework to initialize specific blocks in more complex architectures that might bear more information and are more valuable for the task at hand.

References

[1] Mohammed Ali Al-garadi, Kasturi Dewi Varathan, and Sri Devi Ravana. Identification of influential spreaders in online social networks using interaction weighted k-core decomposition method. *Physica A: Statistical Mechanics and its Applications*, 468:278–288, 2017.

[2] Vladimir Batagelj and Matjaz Zaversnik. An o (m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049*, 2003.

[3] Christos Giatsidis, Dimitrios M Thilikos, and Michalis Vazirgiannis. D-cores: measuring collaboration of directed graphs based on degeneracy. *Knowledge and information systems*, 35(2):311–343, 2013.

[4] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS10). Society for Artificial Intelligence and Statistics*, 2010.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, 2015.

[6] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *Ann. Math. Statist.*, 23(3):462–466, 09 1952.

[7] Alex Krizhevsky et al. Learning multiple layers of features from tiny images. 2009.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[9] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[10] Dmytro Mishkin and Jiri Matas. All you need is a good init. *arXiv preprint arXiv:1511.06422*, 2015.

[11] Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4602–4609, 2019.

[12] Giannis Nikolentzos, Polykarpos Meladianos, Stratis Limnios, and Michalis Vazirgiannis. A degeneracy framework for graph similarity. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 2595–2601. International Joint Conferences on Artificial Intelligence Organization, 7 2018.

[13] Tomaz Pisanski and Milan Randic. Bridges between geometry and graph theory. *MAA NOTES*, pages 174–194, 2000.

[14] Maria-Evgenia G Rossi, Fragkiskos D Malliaros, and Michalis Vazirgiannis. Spread it good, spread it fast: Identification of influential nodes in social networks. In *Proceedings of the 24th International Conference on World Wide Web*, pages 101–102, 2015.

[15] Stephen B Seidman. Network structure and minimum degree. *Social networks*, 5(3):269–287, 1983.

[16] Konstantinos Skianis, Giannis Nikolentzos, Stratis Limnios, and Michalis Vazirgiannis. Rep the set: Neural networks for learning set representations. *arXiv preprint arXiv:1904.01962*, 2019.

[17] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):19291958, January 2014.