Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Sparse neural network training from scratch

Master Thesis

2023
Marcel Palaj, Bc.

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Sparse neural network training from scratch

Master Thesis

Study Programme:	Applied Computer Science
Field of Study:	Computer Science
Department:	Department of Applied Informatics
Supervisor:	Mgr. Vladimír Boža, PhD.

Bratislava, 2023
Marcel Palaj, Bc.

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Marcel Palaj
Študijný program: aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Sparse neural network training from scratch
Trénovanie riedkych neurónových sietí z ničoho

Anotácia: Neurónové siete majú typicky oveľa viac parametrov ako je potrebné a je ich možné orezať pomocou viacerých techník. Výsledná neurónová sieť potom pozostáva z riedkych váhových matíc.
Našim cieľom je trénovať riedke siete od začiatku trénovania bez nutnosti používať operácie na hustých maticiach.
Očakávame, že počas trénovania sa štruktúra matíc bude meniť (niektoré váhy sa odstránia a niektoré sa pridajú). Predošlé práce buď váhy pridávali náhodne ([1]) alebo používali husté gradienty ([2,4]).
Našim cieľom je vyrobiť novú metódu, ktorá prídava nové váhy rozumných spôsobom a zároveň nepotrebuje husté gradienty.
Voliteľné môžeme tiež vyskúšať rôzne spôsoby inicializácie siete a rôzne spôsoby parametrizácie riedkych matíc.

Literatúra: [1] Mocanu, Decebal Constantin, et al. "Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science." Nature communications 9.1 (2018): 1-12.
[2] Evci, Utku, et al. "Rigging the lottery: Making all tickets winners." International Conference on Machine Learning. PMLR, 2020.
[3] Liu, Shiwei, et al. "Do we actually need dense over-parameterization? in-time over-parameterization in sparse training." International Conference on Machine Learning. PMLR, 2021.
[4] Liu, Junjie, et al. "Dynamic sparse training: Find efficient sparse network from scratch with trainable masked layers." arXiv preprint arXiv:2005.06870 (2020).

Vedúci: Mgr. Vladimír Boža, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: doc. RNDr. Tatiana Jajcayová, PhD.

Dátum zadania: 30.11.2022

Dátum schválenia: 07.12.2022 prof. RNDr. Roman Ďurikovič, PhD.
garant študijného programu

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

....

študent vedúci práce

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Marcel Palaj
Study programme:	Applied Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Sparse neural network training from scratch

Annotation: Neural networks are typically overparametrized and can be pruned using various techniques, resulting in networks with sparse weight matrices.
Our goal is to achieve training of sparse neural networks from scratch without need for dense operations in any stage of training.
During training, we expect that sparsity pattern will change (some weight will be dropped and some weight will be added in).
Previous works, either add new weights randomly ([1]) or use dense gradients ([2,4]).
One of our goals is to develop a new method, which adds new weights in a clever way without using full gradients.
Optionally, we can also experiment with various ways of sparse network initialization and different ways of reparametrizing the network using sparse metrics.

Literature:
[1] Mocanu, Decebal Constantin, et al. "Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science." Nature communications 9.1 (2018): 1-12.
[2] Evci, Utku, et al. "Rigging the lottery: Making all tickets winners." International Conference on Machine Learning. PMLR, 2020.
[3] Liu, Shiwei, et al. "Do we actually need dense over-parameterization? in-time over-parameterization in sparse training." International Conference on Machine Learning. PMLR, 2021.
[4] Liu, Junjie, et al. "Dynamic sparse training: Find efficient sparse network from scratch with trainable masked layers." arXiv preprint arXiv:2005.06870 (2020).

Supervisor:	Mgr. Vladimír Boža, PhD.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	doc. RNDr. Tatiana Jajcayová, PhD.
Assigned:	30.11.2022
Approved:	07.12.2022

prof. RNDr. Roman Ďurikovič, PhD.
Guarantor of Study Programme

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

....

Student Supervisor

Acknowledgments: Tu bude poďakovanie.

Abstrakt

Slovenský abstrakt v rozsahu 100-500 slov, jeden odstavec. Abstrakt stručne sumarizuje výsledky práce. Mal by byť pochopiteľný pre bežného informatika. Nemal by teda využívať skratky, termíny alebo označenie zavedené v práci, okrem tých, ktoré sú všeobecne známe.

Kľúčové slová: jedno, druhé, tretie (prípadne štvrté, piate)

Abstract

Abstract in the English language (translation of the abstract in the Slovak language).

Keywords:

Contents

List of Figures

List of Tables

Introduction

Cieľom tejto práce je poskytnúť študentom posledného ročníka bakalárskeho štúdia informatiky kostru práce v systéme LaTeX a ukážku užitočných príkazov, ktoré pri písaní práce môžu potrebovať. Začneme stručnou charakteristikou úvodu práce podľa smernice o záverečných prácach [**?**], ktorú uvádzame ako doslovný citát.

> Úvod je prvou komplexnou informáciou o práci, jej cieli, obsahu a štruktúre. Úvod sa vzťahuje na spracovanú tému konkrétne, obsahuje stručný a výstižný opis problematiky, charakterizuje stav poznania alebo praxe v oblasti, ktorá je predmetom školského diela a oboznamuje s významom, cieľmi a zámermi školského diela. Autor v úvode zdôrazňuje, prečo je práca dôležitá a prečo sa rozhodol spracovať danú tému. Úvod ako názov kapitoly sa nečísluje a jeho rozsah je spravidla 1 až 2 strany.

V nasledujúcej kapitole nájdete ukážku členenia kapitoly na menšie časti a v kapitole **??** nájdete príkazy na prácu s tabuľkami, obrázkami a matematickými výrazmi. V kapitole **??** uvádzame klasický text Lorem Ipsum a na koniec sa budeme venovať záležitostiam záveru bakalárskej práce.

Chapter 1

Overview

In this chapter, we will focus on basic concepts for understanding this work. We will briefly describe feedforward neural networks, their learning methods, and layers which are important for this thesis.

Deep feedforward network

A deep feedforward Network or multilayer perceptron (MLP) is a machine learning model, which goal is to approximate some function f^* [6]. This model can be used for various tasks (classification, regression, ...), but in this work, we only use this model to classificate input vector x to category y. An MLP defines mapping $y = f(x; \theta)$, and during learning, it learns parameters θ [6].

Usually, MLP consists of multiple composed different functions. For example, if the network is composed of two functions $g(x)$ and $h(x)$ chained together, it means that $f(x) = h(g(x))$. Each function can be called *layer*. Rather than thinking about the layer as a function on a vector, we usually think about the layer as many processing units called neurons. Each neuron is connected to some other neurons, and it computes its activation value (which is used for other neurons as input).

The overall length of this chain of functions/layers is called *depth*. The last layer of the model is called *output layer*.

Let's just briefly review the two most important layers of this work: the linear layer and convolutional layers.

Linear layer

The linear layer is the most basic layer in MLP. It models affine transformation. Given input vector x, this layer produces result vector $y = W^T x + b$, where W and b are learnable parameters. Because stacking these layers doesn't increase the model's amount of functions, which can approximate, this layer is followed by some *activation function*. It means, that result of this layer is $y = f(W^T x + b)$, where f is *activation function*.

Linear layer followed by *activation function* is called *dense layer* or *fully connected layer*.

Convolutional layer

The convolutional layers are special kinds of layers for processing data with known grid-like topology [6]. A usual example is processing images because each image is a grid of pixels. This type of layer doesn't use matrix multiplication by full-sized matrix, but instead, it uses a small matrix called *kernel* (which similarly to the linear layer has learnable parameters). This kernel slides over the input data and computes the dot product of the kernel and input data.

//TODO obrazok konvolucie

Learning of MLP

The term "learning" (or "training") in machine learning means process, where learnable parameters are set. There are multiple paradigms, to be followed:

- supervised learning
- unsupervised learning
- semisupervised learning
- reinforcement learning
- ...

These paradigms differ in one biggest thing: whether we have "correct answers" and the model is trying to learn to answer these answers.

Because in this thesis we are dealing only with supervised learning, we will not discuss other paradigms.

In most cases, all learnable parameters (weights) of MLP are learned using gradient descent. In this approach, we have some *loss* or *loss* function, which we are minimizing. Because this function is nonconvex, MLPs are usually trained using iterative methods.

Let's now a little more precisely describe iterative methods of MLP training. For this training, we have a dataset of inputs. For each input, we have the correct "answer". Training consists of multiple rounds of giving model samples from the dataset. For each input, we compare the desired (correct) output with the correct output. Based on the difference between these outputs, we can calculate the gradient and adjust weights.

The gradient for the MLPs is calculated using *back-propagation* algorithm.

We will write back-propagation equations for one linear layer in a multi-layer perceptron. Let x be input for this layer and W matrix of learnable parameters for this

layer. Denote z as the output of this layer. It means, that $z = Wx$. From the next layer, the backpropagation algorithm brings us a gradient of our output z ($\frac{\partial L}{\partial z}$).

If we want to compute the gradient of each weight in this layer, it is computed as

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial z} \cdot x^T$$

However, we are only exceptionally training models by giving them training data one by one. Instead, we are giving multiple data at one time (we are calling these groups "minibatches"). The resulting gradient is the sum of gradients for each data in the minibatch. This method is called "minibatch training".

So, if we are training the model using the minibatch method, the gradient for each weight can be calculated as

$$\frac{\partial L}{\partial W} = \sum_{x \in batch} \frac{\partial L}{\partial z} \cdot x^T$$

If we write down this equation for one particular weight W_{ij}, we get:

$$\frac{\partial L}{\partial W_{ij}} = \sum_{x \in batch} \frac{\partial L}{\partial z_j} \cdot x_i$$

Also, we need to pass the gradient of our input x to the previous layer (if such exists). We can calculate this gradient as

$$\frac{\partial L}{\partial X} = W^T \cdot \frac{\partial L}{\partial z}$$

// regularizácia

Sparse neural networks

Nowadays, deep neural networks are achieving state-of-the-art models in machine learning. However, the results scale with model and dataset size [9]. Current state-of-the-art models have multiple millions of parameters, which require huge computational power even for model training, and much more for hyperparameter tuning.

By the term "sparsity", we mean that the subset of the model's (or layer's) learnable parameters is set to zero [5]. If we have somewhere in the model zero-valued weights results of any multiplications (which dominate calculations in neural network training and testing) are zero, and thus can be skipped. Of course, we also don't need to store these zero-valued parameters, so model storing requires fewer parameters. An opposite of "sparse model" is usually used "dense model", term which means, that none of the model's weights are zero-valued on purpose. Of course, if the result of the training is that some weights are zero-valued (and it isn't our intention), we usually don't refer to this model as a "sparse model".

The term "sparsity" also refers to the fraction of weights in the model, which are set to zero. So if some model has "higher sparsity", it means, that it has more zero-valued weight [5]. And, the last term that we are going to use is "pruning". By "pruning" we mean removing connections (weights) from the network, i.e. processes that make the network more sparse.

Chapter 2

Related work

In this chapter, we will discuss methods, which is related to sparse neural networks. We can divide these methods into two parts: methods, that use dense training, and methods that use sparse training. We will now describe both of them in more detail.

Dense training, sparse inference

Research on sparse networks isn't a new topic. Firstly, there were introduced methods, which started on dense networks, and throughout the training process, weights were removed to achieve the desired sparsity.

All methods which will be mentioned in this section will have one thing in common: they are using dense networks in some stages of training (usually in the beginning). But, even with dense training, models produced by these methods are sparse, therefore in the inference phase they have fewer weights, so they require fewer operations. Also, they require less space to store or transfer. Both these improvements can result in wider usage of these networks in fields like mobile apps, where large storage overhead can prevent deep neural networks from being used in these situations.

It's worth noting, that the motivation of the earliest research on this topic wasn't driven only by reducing model size or by reducing operations needed for model training, but as another regularization method for improve model generalization ability [15].

There is a large amount of methods and techniques, that use dense training, perhaps it is because these methods are somehow easier to develop. If at the beginning of the training, the model is dense (it contains all weights), it means, that weights which are in our *best sparse model* (that we want to find), are for sure a subset of weights of the dense model. So, in simple terms, we just want to find which weights we want to prune to achieve the desired sparsity.

There were works ([15], [8], [2]) that prove the possibility of pruning small magnitude weight with none or very minimal accuracy loss. The idea behind these methods is shared across multiple methods and is somehow intuitive: the magnitude of each

weight is proportional to weight importance. Removing weight with the smallest magnitude doesn't change the result of the model so much (or at least for the result of this layer). So, all these methods remove the least important weights during training according to some sparsification schedule.

The sparsification schedule in these methods can be some iterative pruning, when they were removing a small amount of weight regularly during training, or they were doing some "train-prune-retrain" scheme.

Examples of some "train-prune-retrain" methods can be found in [7] and [14]. While the first method was pruning model only once, the second method was doing multiple rounds of alternating sparse and dense phases during the training process.

Another interesting example of the "train-prune-retrain" method is [8], which is a method, that trains dense networks, and when training is (already) finished, weights with the smallest magnitude are removed. This first training can be viewed as "learning which weights are important". However, this step would have a significant impact on network accuracy, so it is followed by a retraining network, but now with sparse topology.

There also had been techniques, that used second-order approximation of the loss function, and based on this approximation were removing unimportant weights [11]. However, calculating second-order loss approximation is less computationally efficient (compared to calculating first-order loss), and modern techniques achieved comparable results with only first-order loss.

One of the methods, that brings valuable insight into the topic of sparse networks is Lottery ticket hypothesis[4]. To quote this work:

> **The Lottery Ticket Hypothesis.** A randomly-initialized, dense neural network contains a subnetwork that is initialized such that—when trained in isolation—it can match the test accuracy of the original network after training for at most the same number of iterations.

These networks, that can match the test accuracy were called "winning tickets" (in the lottery). To identify these "winning tickets", they perform the usual network training (starting with the usual random initialization). After some iterations, they prune the lowest magnitude weights, to reach desired network sparsity. After this pruning, they train the network again from scratch, but using only nonpruned weights. They also explore iterative pruning, when they apply that "train-prune-reset-train" more times. This iterative pruning was able to find smaller networks that with their accuracy match their dense counterparts.

But, there is also another interesting thing, in this paper: it turns out, that it doesn't only matter on model sparsity layout (i.e. which weight has to be nonzero-valued), but also on weight initialization. When "winning tickets" are randomly reini-

tialized, they learn slowly and achieve lower test accuracy as if they are initialized exactly as they were at the beginning of the previous training.

The first possible explanation - these values are close to their final values - turns out to be wrong. The explanation provided by the authors is, that "winning ticket" initialization is connected to the optimization algorithm, dataset, and model. For example, this initialization can be in some good regions of the loss landscape, where the optimization algorithm quickly finds some good local minima.

Sparse training, sparse inference

Now, let's describe methods that are training sparse neural networks from the beginning.

The problem with sparse training is that we simply don't know which weights are important for the network (for their results). It means that is not possible to decide, which weights will be zero-valued, and which will have a nonzero value at the beginning of the training, and leave that unchanged during the whole training process.

This means, that all methods, which are using sparse training are not only removing/deleting weights to networks, they are also adding/growing new weights.

We can divide them into two groups: either they use dense gradient information, or they use randomization to find the best connection distributions.

Sparse Evolutionary Training (SET)

Sparse Evolutionary Training (SET)[13] is an example of the second method, which uses randomization.

This method starts with a random sparse network. More precisely, the weight matrix is Erdős-Rényi random graph. In this graph, the probability of a connection between neuron h_i^k and h_j^{k-1} (there is n_k neurons in layer k and n_{k-1} neurons in layer $k-1$) is given by:

$$P(W_{ij}^k) = \frac{\epsilon(n^k + n^{k-1})}{n^k n^{k-1}} \tag{2.1}$$

where $\epsilon \in R^+$ is a parameter controlling sparsity level. Of course, this initial randomly generated topology cannot be well-suited for all datasets and all models. So, during training this topology changes. After each training epoch new weight grows and some weights are removed (to preserve sparsity level). Weights are removed based on the smallest magnitude, which was discussed in methods with dense training. New connections are grown randomly (new connections are chosen randomly from zero-valued weight, with exceptions to weight which we removed right now).

//TODO update schedule

This approach is (as can be guessed from the name) an example of some evolutionary algorithm, where removing the smallest magnitude weight can be viewed as the "selection" phase in evolution. Also, growing new weights can be viewed as the "mutation" phase in evolution.

The interesting thing, that this work discovers is that even if the sparsity layout is not updated (it is fixed from the beginning), the results of the network are not as bad as one would guess. Of course, they are worse than dense models and also worse than models trained using SET, but not that bad. It confirms that the networks can overcome removed weights and learn despite being heavily sparsified.

Rigging the lottery

Paper [3] is also removing connections with the smallest magnitude weight. However, this method grows new connections based on their highest magnitude gradients. The idea behind this method is that if the magnitude gradient is really large for some (not existing) connections, it means that these connections are important for network results and that we should add this connection to our network. Newly added connections are initialized to zero, so they don't affect the output of the network.

This method also doesn't update sparsity layout after each training step, instead only each ΔT step. There also was the decay of the amount of weight updated applied. The method that slightly outperforms the other methods considered is cosine annealing.

$$f_{decay}(t; \alpha, T_{end}) = \frac{\alpha}{2}\left(1 + \cos\left(\frac{t\pi}{T_{end}}\right)\right)$$

where α, T_{end} are hyperparameters.

This work also explored sparsity distribution between layers. They consider 3 strategies: *uniform* (where the sparsity of each layer is equal to the total sparsity of the network), *Erdős-Rényi* (where sparsities were distributed the same way as in the previous method, using Erdős-Rényi random graph created using equation 2.1), and *Erdős-Rényi-Kernel* (ERK), which modified equation 2.1 to include also kernel dimensions. The reasoning behind the second and third options is that they enable lower sparsities in the smaller layers (layers with fewer parameters), and higher sparsities for larger layers. Using these distributions, it is unlikely that some "bottleneck" happens, where we prune nearly all weights from some small layer, and we nearly stop information flow through the model. Experimental results also confirm this hypothesis. The difference between these options became larger with a bigger sparsity of models.

However, this method requires a dense gradient in each weight update step, of course, this method doesn't require storing this gradient, it can be immediately discarded. If the gradient is too large to fit into memory, it can even be calculated in an online manner, because we only need to store top-k values of the gradient vector.

Dynamic sparse training

Lastly, work [12] has been using a different approach. In this method, they make sparsity an integral part of the training process. After each weight matrix in the network, they add a "mask layer". This layer was just returning input (i.e. weight) if the magnitude of input was greater than some threshold t_h, or else it was returning 0, according to the equations:

$$Q_{ij} = F(W_{ij}, t_i) = |W_{ij}| - t_i$$

$$M_{ij} = S(Q_{ij})$$

$$S(x) = \begin{cases} 1, & \text{if } x \geq 1, \\ 0, & \text{otherwise.} \end{cases}$$

And weight value W_{ij} will be masked by value M_{ij}, so result weight will be $W_{ij} \cdot M_{ij}$

In this work, they tried three options for thresholds (for one layer): have only one scalar threshold (i.e. one threshold for each *layer*), have a vector of thresholds (i.e. one threshold for each *neuron*), and have a matrix of thresholds (i.e. one threshold for each *weight*). Matrix and vector thresholds have similar results, but the scalar threshold was less robust. Finally, they decided to have vector thresholds, because of the smaller amount of additional parameters.

However, these thresholds (due to the usage in step function $S(x)$) are not trainable using back-propagation, because we require the derivative of step function $S(x)$. This derivative is an impulse function, which looks like this:

$$\delta(x) = \begin{cases} \infty, & \text{if } x = 0, \\ 0, & \text{otherwise.} \end{cases}$$

Which is not usable for back-propagation. So, they decided to use only an approximation of this derivation of this function, long-tailed estimator.

$$\frac{d}{dx}S(x) \approx H(x) = \begin{cases} 2 - 4|x|, & -0.4 \leq |x| \leq 0.4, \\ 0.4, & 0.4 < |x| \leq 1, \\ 0, & \text{otherwise.} \end{cases}$$

With this estimator, the thresholds can be trained via back-propagation. Each network parameter is during training receiving two types of gradients. One is the performance gradient for better model performance (for higher model accuracy), and the second is the structure gradient for better sparse structure.

Even if the network thresholds are trainable, there is nothing, that is pruning weights to make the network more sparse. So, this method introduced a penalty called

"sparsity regularization term", which was added to the loss function. For each trainable masked layer, the regularization term is

$$R = \sum exp(-t_i)$$

and sparse regularization term for the whole network is the sum of these regularization terms:

$$L_S = \sum R_i$$

This brings us to the desired result: the more sparse the network, the smaller the penalty. Of course, this sparse regularization term is added to the overall loss multiplied by some hyperparameter α.

This approach allows the network to somehow "decide on their own way" how to make sparsity distribution, and which parameters will be pruned, depending on how important they seem to be.

Chapter 3

Research

In our work, we are developing a new method to update connections in sparse neural networks. This method (like other methods) consists of two parts:

1. drop old connections
2. grow new connections

Part, when we are dropping old connections is shared across multiple sources (e.g. [3], [13]). We are dropping connections with the smallest absolute value of weights, because these connections are somehow least important for network output and overall performance, because their influence on network results is probably minimal.

The second part is growing new connections. We are growing new connections based on the magnitude gradient.

If we have one layer in a multi-layer perceptron. Where output z of this layer is $z = Wx$. The gradient for some weight in the layer is calculated as $\frac{\partial L}{\partial W_{ij}} = \sum_{x \in batch} \frac{\partial L}{\partial z_j} \cdot x_i$.

If we want to find connections with the largest magnitude gradients, we basicaly want to find some i, j for which is value $|\sum_{x \in batch} \frac{dL}{dz_j} \cdot x_i|$ maximal.

But, we can also look at this problem as finding the most similar vectors between two groups of vectors, where all vectors have the same size as batch size. Each vector in the first group of vectors consists of values $\frac{dL}{dz_j}$ for each value $x \in batch$. Each vector in the second group consists of values x_i for each value $x \in batch$.

In each update step, we are updating some fraction of connections. We denote this hyperparameter as K, where $K \in \langle 0, 1 \rangle$. This number means, that if denote the number of all connections in this layer as c, we are updating $c \cdot K$ connections.

Because finding most similar vectors, is quite a common task, there exists a couple of projects and libraries, that perform this task quite fast. In this thesis, we consider two of them: faiss[10] and nmslib[1]. These libraries create an object called *index*, where are all vectors from one group inserted. After this step, *index* can perform operation *search(v, k)* of finding k most similar vectors from *index* to vector v. Both these

libraries support batch queries, which means, that if we have some array of vectors V, operation *search(V, k)* will quickly find k most similar vectors to each vector in array V.

Also, using a similarity search between two groups of vectors brings us another hyperparameter to explore: k, which is the number of most similar vectors for *index* to find. Because the sizes of groups of vectors will vary between layers in the model, we introduced different hyperparameter $k_{similar}$. Similar to hyperparameter K, this hyperparameter denotes a fraction of vectors, which we are finding. It means, that $k_{similar} \in \langle 0, 1 \rangle$. More precisely, let us consider that we have two groups of vectors. Let's say that in the first group is a vectors and in the second in b vectors. We add create *index* from the first group of vectors (a vectors) and perform a search on group b, we will set $k = a \cdot k_{similar}$.

If we set $k_{similar} = 1$, the result of this query will be identical to a brute force search, where we will have distances to all vectors from another group. We have to also note, that in this case, this approach would be slower because index creation is also an operation that costs some time. But, we believe, that it will be sufficient to set $k_{similar}$ to some small value (e.g. something like 0.2), which can bring us to speed up in comparison to the brute-force search.

The difference between searching for most similar vectors using brute force search and searching with faiss is shown in figure 3.1. Both methods start with a similarity matrix (in the figure on the left top). Searching using faiss finds for each vector some number (this number is determined by hyperparameter $k\_similar$) of the most similar vector (in the figure it finds 2 most similar vectors for each row). These vectors are flattened together, and sorted, and we choose K most similar vectors as new connections. Contrary, brute-force search flattens these similarities as the first step, sorting them and directly choosing the K most similar vectors. Thus, searching using faiss (with these similarities and these hyperparameters) wasn't able to find the most similar vectors (in this example searching using faiss doesn't find vector with similarity 33.1)

So, to sum up our algorithm: we start with a random sparse network. After some training steps (determined by hyperparameter ΔT) we update connections using the methods described above. We are removing connections with the smallest absolute value of weights. Connections are added based on the largest vector similarity (largest magnitude of gradient). We need to decide how are we going to initialize newly added connections. The pseudocode for our method can be seen at algorithm 3.

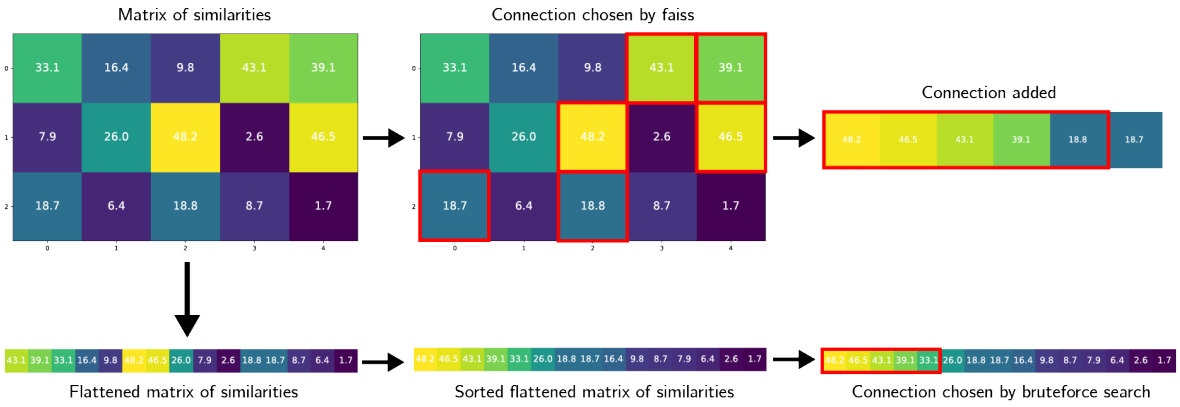

Figure 3.1: Difference between searching for most similar vectors using brute-force search and searching using faiss.

Algorithm 1 Training model with one layer

Input network f_W, dataset D
connection update hyperparameters: K, dT

randomly sparsify W
for each training step t **do**
  x_i, y_i = input and output of minibatch from D
  $L_t = L(f_W(x_i), y_i)$
  $W = W - \alpha \nabla_W L_t$
  if $t \pmod{dT} == 0$ **then**
    remove K weights with lowest $|W|$
    add K weights with largest similarity between ∇_W and x_i
  end if
end for

We ran several experiments with various hyperparameters to compare training time for finding connections for updates using brute-force search and using vector similarity search.

All experiments have been running on server with 12 CPUs *Intel Xeon CPU E5-1650 v3 @ 3.50GHz*, 32 GB RAM and two GPUs: *Nvidia RTX 3090* and *Nvidia GTX 1080*.

All experiments were run on WideResNet28x5 model on CIFAR 100 dataset. In this model, all convolution and linear layers were replaced by random sparse layers. During training, we updated connections in all convolution layers.

The first experiment was a training model with connections updated each minibatch ($\Delta T = 1$). Results from training can be seen in tables 3.1 and 3.3.

From these results, we can clearly see, that connection updates based on inner

Model	**Sparsity**	K	$k_{similar}$	**Training time (70 epochs)**	**Val. acc**	**Note**
basic	0%	—	—	$0.61h$	74.83%	
pruned	80%	—	—	$0.65h$	69.94%	
pruned + bruteforce	80%	0.01	—	$3.59h$	72.68%	*GPU 0*
pruned + bruteforce	80%	0.05	—	$3.65h$	72.01%	*GPU 0*
pruned + bruteforce	80%	0.1	—	$3.81h$	71.17%	*GPU 0*
pruned + faiss	80%	0.01	0.15	$3.72h$	73.65%	*GPU 1*
pruned + faiss	80%	0.05	0.15	$3.74h$	72.13%	*GPU 1*
pruned + faiss	80%	0.1	0.15	$4.02h$	70.59%	*GPU 1*
pruned + faiss	80%	0.01	0.125	$2.32h$	72.53%	*GPU 0*
pruned + faiss	80%	0.05	0.125	$2.43h$	73.28%	*GPU 0*
pruned + faiss	80%	0.1	0.125	$2.55h$	68.21%	*GPU 0*

Table 3.1: Results from training model with $abs(IP)$ connection updates after each minibatch

Model	**Sparsity**	K	$k_{similar}$	**Training time (70 epochs)**	**Val. acc**	**Note**
basic	0%	—	—	$0.61h$	74.83%	
pruned	80%	—	—	$0.65h$	69.94%	
pruned + bruteforce	80%	0.01	—	$1.94h$	67.53%	*GPU 0*
pruned + bruteforce	80%	0.05	—	$2.21h$	66.70%	*GPU 0*
pruned + bruteforce	80%	0.1	—	$2.34h$	55.63%	*GPU 0*
pruned + faiss	80%	0.01	0.15	$2.36h$	68.73%	*GPU 1*
pruned + faiss	80%	0.05	0.15	$2.53h$	69.68%	*GPU 1*
pruned + faiss	80%	0.1	0.15	$2.63h$	65.35%	*GPU 1*
pruned + faiss	80%	0.01	0.2	$3.84h$	68.95%	*GPU 1*
pruned + faiss	80%	0.05	0.2	$3.99h$	67.59%	*GPU 1*
pruned + faiss	80%	0.1	0.2	$4.07h$	64.17%	*GPU 1*

Table 3.2: Results from training model with $L2$ connection updates after each minibatch

Model	**Sparsity**	K	$k_{similar}$	**Training time (70 epochs)**	**Val. acc**	**Δ acc**	**Note**
basic	0%	—	—	$0.61h$	74.83%	—	
pruned	80%	—	—	$0.65h$	69.94%	—	
pruned + bruteforce	80%	0.01	—	$0.84h$	71.92%	−0.69%	*GPU 0*
pruned + faiss	80%	0.01	0.15	$0.78h$	71.99%	−1.66%	*GPU 0*
pruned + faiss	80%	0.05	0.15	$0.83h$	73.04%	+0.91%	*GPU 0*
pruned + faiss	80%	0.1	0.15	$0.84h$	71.08%	+0.49%	*GPU 0*
pruned + faiss	80%	0.01	0.125	$0.77h$	71.91%	−0.62%	*GPU 0*
pruned + faiss	80%	0.05	0.125	$0.81h$	72.01%	−1.27%	*GPU 0*
pruned + faiss	80%	0.1	0.125	$0.76h$	69.90%	+1.69%	*GPU 0*

Table 3.3: Results from training model with IP connection updates each epoch.

product similarity provides much better results than connection updates based on L2 distance. Even the best model using $L2$ metrics is worse than a model with the fixed sparsity mask. With this knowledge, we perform other experiments using solely $abs(IP)$ metrics as a similarity measure.

Also, if we compare training time (on the same GPU) for the model using bruteforce similarity search and the model using faiss similarity search, we can see, that similarity search using faiss is faster for up to approximately $k\_similar = 0.3$. It is worth noting, that in these experiments we use only the CPU version of faiss. We tried using GPU version of faiss in the latter part of this thesis.

We decided to take a closer look at the connection update schedule.

As a first experiment, we update weights only after each epoch (which is 196× less than the update after each minibatch, mentioned in 3.1 and 3.3).

From these results, we can formulate behavior, which is somewhat intuitive: *the more often we are doing connection updates, the fewer connections we need to update* (to achieve the same accuracy). Of course, this statement doesn't hold for extreme cases, such as if we want to do only one connection update during training, when we replace all connections.

An interesting thing, that also seems intuitive is, that doing connections update not after each training step, but after a few steps can raise model accuracy. It is because, similar to the usage of momentum in the training process in neural networks if we are doing connection updates based on the average of more training steps, we can better overcome some local patterns in the training set.

Model	**Sparsity**	K	$k_{similar}$	$bits$	**Training time (70 epochs)**	**Val. acc**
basic	0%	—	—	—	$0.61h$	74.83%
pruned	80%	—	—	—	$0.61h$	69.94%
pruned + faiss	80%	0.05	0.15	0.5	$2.41h$	72.46%
pruned + faiss	80%	0.05	0.15	1	$2.49h$	72.8%
pruned + faiss	80%	0.05	0.15	1.5	$2.68h$	72%
pruned + faiss	80%	0.05	0.15	2	$3.01h$	72.21%
pruned + faiss	80%	0.05	0.15	4	$6.08h$	72.1%
pruned + faiss	80%	0.05	0.15	8	$17.17h$	71.48%

Table 3.4: Results from training model with LSH connection updates after each mini-batch

Model	**Sparsity**	K	$k_{similar}$	$bits$	**Training time (70 epochs)**	**Val. acc**
basic	0%	—	—	—	$0.61h$	74.83%
pruned	90%	—	—	—	$0.61h$	65.92%
pruned + faiss	90%	0.01	0.15	0.5	$2.23h$	67.65%
pruned + faiss	90%	0.01	0.15	1	$2.31h$	69.5%
pruned + faiss	90%	0.01	0.15	1.5	$2.48h$	70.02%
pruned + faiss	90%	0.01	0.15	2	$2.79h$	69.73%
pruned + faiss	90%	0.025	0.15	0.5	$2.29h$	68.39%
pruned + faiss	90%	0.025	0.15	1	$2.36h$	68.12%
pruned + faiss	90%	0.025	0.15	1.5	$2.53h$	68.85%
pruned + faiss	90%	0.025	0.15	2	$2.85h$	68.44%

Table 3.5: Results from training model with LSH connection updates after each mini-batch

Conclusion

Tu budú výsledky práce.

Bibliography

[1] Leonid Boytsov and Bilegsaikhan Naidan. Engineering efficient and effective non-metric space library. In Nieves R. Brisaboa, Oscar Pedreira, and Pavel Zezula, editors, *Similarity Search and Applications - 6th International Conference, SISAP 2013, A Coruña, Spain, October 2-4, 2013, Proceedings*, volume 8199 of *Lecture Notes in Computer Science*, pages 280–293. Springer, 2013.

[2] Maxwell D. Collins and Pushmeet Kohli. Memory bounded deep convolutional networks, 2014.

[3] Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. Rigging the lottery: Making all tickets winners. In *International Conference on Machine Learning*. PMLR, 2020.

[4] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks, 2019.

[5] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks, 2019.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[7] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2016.

[8] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[9] Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md. Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. Deep learning scaling is predictable, empirically, 2017.

[10] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[11] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. *Advances in neural information processing systems*, 2, 1989.

[12] Junjie Liu, Zhe Xu, Runbin Shi, Ray C. C. Cheung, and Hayden K. H. So. Dynamic sparse training: Find efficient sparse network from scratch with trainable masked layers, 2020.

[13] Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H. Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature Communications*, 9(1), jun 2018.

[14] Alexandra Peste, Eugenia Iofinova, Adrian Vladu, and Dan Alistarh. Ac/dc: Alternating compressed/decompressed training of deep neural networks. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 8557–8570. Curran Associates, Inc., 2021.

[15] Georg Thimm and Emile Fiesler. Evaluating pruning methods. In *Proceedings of the International Symposium on Artificial neural networks*, pages 20–25, 1995.

Príloha A: obsah elektronickej prílohy

V elektronickej prílohe priloženej k práci sa nachádza zdrojový kód programu a súbory s výsledkami experimentov. Zdrojový kód je zverejnený aj na stránke `http://mojadresa.com/`.

Ak uznáte za vhodné, môžete tu aj podrobnejšie rozpísať obsah tejto prílohy, prípadne poskytnúť návod na inštaláciu programu. Alternatívou je tieto informácie zahrnúť do samotnej prílohy, alebo ich uviesť na obidvoch miestach.

Príloha B: Používateľská príručka

V tejto prílohe uvádzame používateľskú príručku k nášmu softvéru. Tu by ďalej pokračoval text príručky. V práci nie je potrebné uvádzať používateľskú príručku, pokiaľ je používanie softvéru intuitívne alebo ak výsledkom práce nie je ucelený softvér určený pre používateľov.

V prílohách môžete uviesť aj ďalšie materiály, ktoré by mohli pôsobiť rušivo v hlavnom texte, ako napríklad rozsiahle tabuľky a podobne. Materiály, ktoré sú príliš dlhé na ich tlač, odovzdajte len v electronickej prílohe.