Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Merging of neural networks

Master's Thesis

2022
Bc. Martin Pašen

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Merging of neural networks

Master's Thesis

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

Bratislava, 2022
Bc. Martin Pašen

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

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

Názov: Merging of Neural Networks
Spájanie neurónových sietí

Anotácia: Cieľom práce je naimplementovať a otestovať procedúru, ktorá dostane dve neurónové siete s rovnakou architektúrou (natrénované s inou počiatočnou inicializáciou) a vytvorí novú sieť (s rovnakou architektúrou), ktorá bude lepšia ako siete na vstupe. V prípade úspechu je možné sa zaoberať rozšíreniami, kde budeme spájať viacero sietí, prípadne optimalizovať trénovanie neuronových sietí.

Vedúci: Mgr. Vladimír Boža, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Dátum zadania: 22.10.2020

Dátum schválenia: 06.04.2022 prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname: Bc. Martin Pašen
Study programme: 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: Merging of Neural Networks

Annotation:
Our goal is to merge two neural networks with the same architecture (but trained from different initialization) into one network (with the same architecture), which has better performance than the original ones. In case of success, we should explore possible extensions such as: merging multiple networks and optimization of training of neural networks

Supervisor: Mgr. Vladimír Boža, PhD.
Department: FMFI.KAI - Department of Applied Informatics
Head of department: prof. Ing. Igor Farkaš, Dr.

Assigned: 22.10.2020

Approved: 06.04.2022 prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Acknowledgments: I want to express my gratitude to my advisor, Vladimír Boža, for the opportunity to work in a field I had little experience in. I am thankful for his guidance. He was open to my ideas and helpful with my problems. Even though I knew he had little to no time, he was always available when I asked.

Abstrakt

Navrhli sme dve stratégie na spájanie dvoch, už natrénovaných neurónových sietí (učiteľov) do jednej (žiaka) s rovnakou veľkosťou. Na identifikáciu dôležitých častí učiteľov sme použili relaxovanú verziu L_0 regularizácie. V prvej stratégií učíme žiaka napodobňovať dôležité časti učiteľov po vrstvách. V druhej spájame učiteľov do jedného modelu a následne orezávame nepodstatné časti.

Môžeme ich použiť, ak už máme dva natrénované modely. No taktiež sme experimentálne ukázali, že natrénovanie dvoch učiteľov a ich spojenie vedie k lepším výsledkom, ako klasické učenie s rovnakým počtom epôch. Našu stratégiu (nauč dvoch učiteľov a spoj ich) porovnávame so stratégiami *najlepší z troch modelov* (natrénuj tri modely a vyber najlepší) a *jeden model* (použi celý trénovací rozpočet na jeden model). Náš prístup má výrazne lepšie výsledky vo všetkých vykonaných experimentoch.

Kľúčové slová: spájanie neurónových sieti, L_0 regularizácia, orezávanie neurónovej siete, destilovanie vedomostí

Abstract

We propose two simple strategies for merging two trained neural networks (teachers) into one (student) of the same size. We use relaxed L_0 regularization to identify essential parts of the teachers. In the first strategy, we train the student to mimic them layer by layer. In the second, we create the student by concatenating teachers in the channel dimension and pruning the unimportant parts.

We can use these strategies if we already have two trained models. But, we also show that training two teachers and then merging them leads to better performance than classical training in the same number of epochs. We compare our strategy (train two teachers and merge them) with the *bo3 strategy* (train three models, pick the best) and *one model strategy* (use the whole training budget on one model). In all performed experiments, our strategy has significantly better results.

Keywords: merging neural networks, L_0 regularization, pruning neural network, knowledge distillation

Contents

Introduction

A typical neural network training starts with random initialization and ends upon convergence in a local optimum. The performance of the trained network is sensitive to the starting random seed. There was observed a 0.5% difference in accuracy between the worst and the best seed on the Imagenet dataset and a 1.8% difference on the CIFAR-10 dataset [24, 29].

We believe that the discrepancy between starting seeds' performance is a result of different learned features in hidden layers in each initialization. Let's say that we are training a model to distinguish between cars and buses. One model may learn that buses have more wheels than cars and use this fact to classify most of the images. Since only a few of the images can not be classified using this fact (e. g. we can not see the wheels), the model may overfit them. Another model may learn that buses have a vertical windshield and similarly overfit the remaining images. Both models will correctly classify most of the new unseen images, but they will struggle if they can not use the core distinguishing fact.

We ask a question if we have access to these two models, can we train a new model that will know both of the facts? Even though the previous example is probably too extreme and the learned features are not that dissimilar, we show in figure 1 that they differ.

We propose two strategies to train student using a labelled dataset and two teachers. In the first strategy, we train the student to mimic important parts of the teachers layer by layer. In the second, we create the student by concatenating teachers in the channel

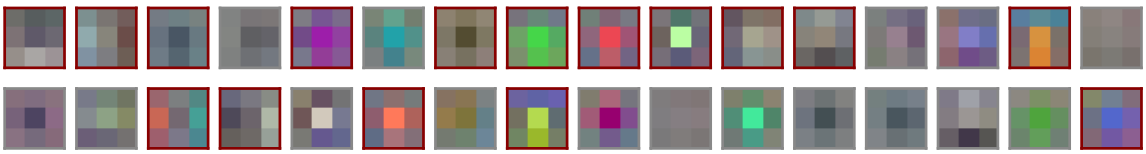

Figure 1: Set of filters in the first layer of two ResNet20 networks trained on CIFAR-100 dataset with different starting seeds. Each row shows filters from one network. We can see that the filters (hidden features) differ even though the networks have the same architecture.

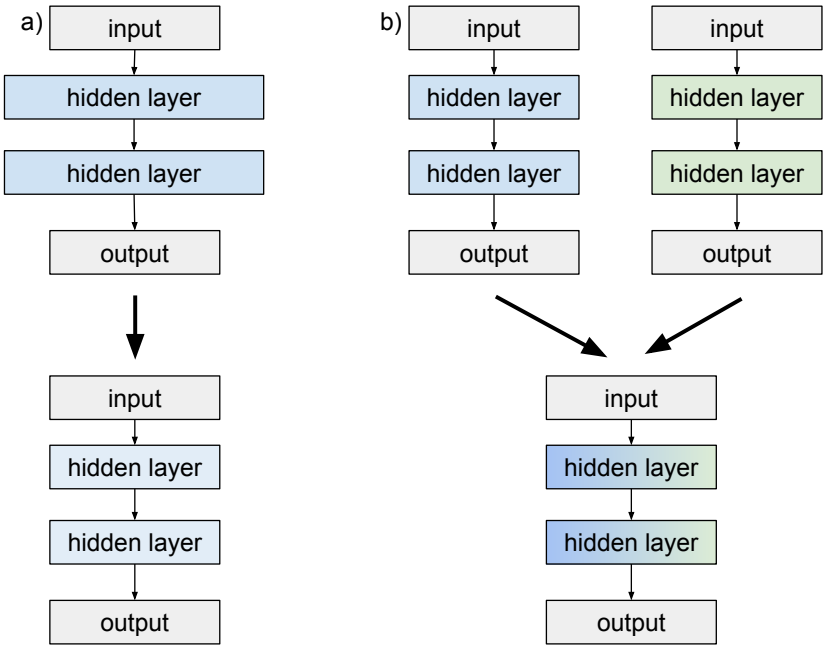

Figure 2: Comparison between a) training a bigger network and then pruning it and b) training two separate networks and then merging them. The width of rectangles denotes the number of channels in the layer.

dimension and pruning the unimportant parts. The opposite approach is training one big model from the start and then selecting the most important channels via channel pruning [22, 21, 18, 30].

Training a big model, which we subsequently prune, might be prohibitive. If we increase the model's width two-fold, we increase the amount of required computation four-fold. To train a big model we might need to change hyper-parameters (e. g. regularization, learning rate). Also, two small models might learn a better combination of hidden features compared to one big model. See figure 2 for the visualization of the two approaches.

Chapter Preliminaries is focused on already known concepts. We start with an intuitive definition of supervised learning and performance. Then we continue with feed-forward neural networks and regularization. In the end, we present a relaxed version of L_0 regularization, introduced by Louizos et. al [17] and architectures LeNet [15] and ResNet [7].

In the following chapter Related work we present two common "training" strategies that utilize teacher, *pruning* and *knowledge distillation*. While pruning modifies the teacher, knowledge distillation trains a new model from scratch. Our approach does not fit into any of the categories. One of our strategies resembles knowledge distillation, another resembles pruning.

The next chapter Methods is devoted to our approach. We introduce two training

strategies. They both merge two teachers into one student with the goal to copy the important parts of the teachers while pruning the non-essential. We do this on the granularity of neurons (linear layer) and channels (convolutional layer). The key task in our strategies is selecting important parts of the teachers. For that, we use L_0 regularization.

In chapter Experimental results, we validate our approach. While one of our strategies had poor results, the other has significantly beaten opposing strategies on all tasks. We have evaluated our strategies on artificial regression problem and image recognition with datasets Imagewoof [11], CIFAR-100 [13], and Imagenet-1k [4]. On the dataset Imagenet-1k, we have beaten the Torchvision [2] training strategy of ResNet-18.

In the last chapter Conclusion we discuss our results and potential research direction.

Chapter 1

Preliminaries

This chapter is focused on the basic concepts essential for understanding our work. We begin with a brief definition of a learning algorithm, feed-forward network, common layers, and regularization. We then proceed with L_0 regularization and architectures LeNet and ResNet.

We expect the reader to be familiar with gradient descent methods and back-propagation. If that is not the case, book [5] provides the necessary introduction.

Tom Mitchel proposed a succinct definition of learning: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."[20]. It is hard to formally define experience, task, and performance measure. And we do not attempt to do so. Instead, we present some examples:

- **experience** - labelled dataset, simulation runs, already trained model ...
- **task** - classification, regression, translation, ...
- **performance measure** - accuracy, mean square error, cross-entropy, ...

We will focus on *supervised learning* - the experience is a labelled dataset. The labelled dataset is a set of pairs (x, y), where x is a vector of input features, and y is a target value. For the task of classification of images, x can be a vector of pixels, and y can be the correct label of the image. For the prediction of tomorrow's weather, y can be tomorrow's temperature, and x can be today's temperature in our and neighbouring countries.

1.1 Feed-forward network

The only model used in this work will be a feed-forward network. It is capable of approximating any continuous function [10]. The feed-forward network with a set

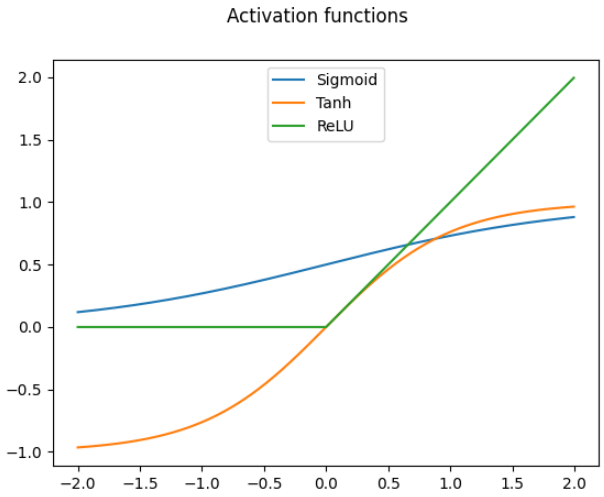

Figure 1.1: A plot of three activation functions - ReLU, Sigmoid, and Tanh. While both ReLU and Tanh have stable point zero, $Sigmoid(0) = 0.5$.

architecture defines mapping $y' = f(x, \theta)$, where x is an input feature vector, θ are parameters, and y' is the output vector.

The feed-forward network is typically a composition of multiple functions $f(x) = f^3(f^2(f^1(x)))$. The individual functions f^i are called layers. The number of layers is called the depth of the network. The dimensionality of the layers determines the width of the network.

There are multiple types of layers, and we will not list them all, only the ones we will use.

1.1.1 Linear layer

A linear layer models affine function $y' = xW + b$, where W and b are learnable parameters called *weight matrix* and *bias*. Since it models an affine function, stacking multiple linear layers does not increase the network's complexity (complexity of the class of the functions it is capable of modelling). Therefore they are usually followed by an activation function. **Activation function** is a fixed (without learnable parameters) function that is supposed to break the affinity of the model. Some of the common activation functions are *ReLU*, *Sigmoid*, and *Tanh*, see figure 1.1.

Hyper-parameters (i. e. the parameters that are not learned and need to be set before the training) of a linear layer are *input size*, *output size*, and *learnability of bias*. The input and output size define the shape of the weight matrix. Learnability of bias says whether the bias will be statically set to zero or learned.

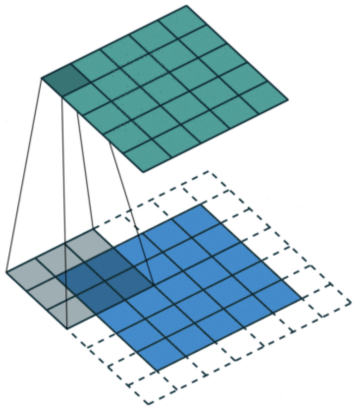

Figure 1.2: Visualization of the computation of convolutional layer's function with kernel 3x3 and padding 1. The output of the convolutional layer has the same dimension as the input. To compute one point of the output we point-wise multiply the kernel with the corresponding input data points and sum it [1].

1.1.2 Convolutional layer

A convolutional layer is used to process data with a grid structure, e. g. 1d time series, 2d images, or 3d videos. It exploits the order of the input data, the fact that the position of a feature bears information as well.

The core learnable parameter of a convolutional layer is a *kernel*. It has the same dimensionality as the input data, but a smaller shape. For example, if we have an image of size 224x224 the kernel may have a size 3x3 or 5x5. The kernel slides over the input data and computes a point-wise multiplication of itself and the corresponding data points, see figure 1.2.

A convolutional layer has multiple hyper-parameters. We will illustrate them on 2d images. Even though the convolutional layer allows for the variability of the size of the image (32x32 or 224x224) the number of values that define one pixel has to be set. For example, a grey-scale image has every pixel defined by one number, while an RGB image has every pixel defined by three numbers. Therefore the shape of the data of the 32x32 grey-scale image is 32x32x1 while the shape of the data of the 32x32 RGB image is 32x32x3. The third dimension is the channel dimension. Hyper-parameters *number of input channels* and *number of output channels* define the channel dimension of input and output data.

Kernel size defines the size of the kernel. For 2d images, the kernel is also a 2d matrix. While usually, its shape is 3x3, 5x5, or 7x7 it does not have to be a square. E. g. a 3x3 kernel is similar to a 1x3 kernel followed by a 3x1 kernel. In both scenarios, a point of the output is computed from the same data points of the input. The difference is in the number of parameters. Kernel 3x3 has nine parameters, while kernel 1x3 and

kernel 3x1 have 6 parameters together. Therefore replacing the 3x3 kernel with 1x3 and 3x1 can decrease the complexity of the network, but it can also help to fight over-fitting.

Stride defines how much we move the kernel between two computations. With stride 2x3 we move the kernel by two pixels to the right until we reach the end of the image and then start again on the left side three pixels lower. The typical strides are 1x1, 2x2, or the same as the size of the kernel. If we use the same stride as the size of the kernel, every input pixel will affect only one point of the output. Stride is a good tool to decrease the size of the data.

If we use a greater kernel than 1x1 the pixels on the edges and in the corners affect fewer output values than pixels in the middle of the image. Also, the size of the output will be smaller by a constant compared to the input. To overcome this we can use *padding*. Padding can be imagined as a frame of the image. We add zeroes (or other values) around the image. Now when we compute the convolution, the previously border pixels are no longer on the edge and are part of multiple convolutions. Typically we pad the image with zeros and with the size of $(kernel\ size - 1)/2$. As a result, the output will have the same shape as the input (as long as the other hyper-parameters such as stride do not brake this). That is why we usually use odd kernel sizes.

Learnability of the bias is analogical to the learnability of the bias in a linear layer. The remaining hyper-parameters *dilation* and *groups* are less frequent and are not used in this work, therefore we skip them.

A convolutional layer may be followed by *pooling*. Usually, it is used to decrease the size of the image. It makes the neural network robust against small shifts in the image. Its computation is similar to the computation of the convolutional layer - it has a kernel, that slides over an image. The difference is that the kernel is set (has no learnable parameters) and can compute more complex functions than the point-wise multiplication of matrices. The most common poolings are *average pooling* and *max pooling*. The average pooling averages the elements under the kernel and max-pooling returns the maximum. The hyper-parameters of pooling are similar to the ones of the convolutional layer: *kernel size, stride*, and *padding*.

1.1.3 Batch normalization layer

A batch normalization layer [12] is used to make networks more stable. It is normalizing the data by re-centring and re-scaling. As the name suggests, for the layer to work we need to train the network in batches (which is practically always the case). Training the network in batches (batch or mini-batch SGD) means that we compute the gradient over multiple (x, y) pairs of the training dataset, average it, and then update the weights.

Let X be a batch of features and $x_{i,j}$ a i-th feature of the j-th batch. Let n to be the size of the batch. First we compute an empirical mean and a variance of every feature:

$$\mu_i = \frac{1}{n}\sum_j x_{i,j} \quad , \quad \sigma_i^2 = \frac{1}{n}\sum_j (x_{i,j} - \mu_i)^2$$

Then we normalize the features:

$$x'_{i,j} = \frac{x_{i,j} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

where ϵ is a small constant added for the numerical stability. The result of the batch normalization layer is:

$$\hat{x}_{i,j} = \gamma_i \cdot x'_{i,j} + \beta_i$$

where γ_i and β_i are either learnable parameters or are set to 1 and 0 respectively.

We have already mentioned hyper-parameters ϵ and learnability of γ and β. Another hyper-parameter is an *input shape*. It defines the number of input features and consequently the shape of γ and β.

We can also define, whether we want to compute the *moving average* of the mean and variance or just the simple mean and variance of the batch. If we use the moving average we need to set *momentum*. Momentum defines the weight of the previous mean and variance in the computation of the current mean and variance. When we use the moving average in the training it is also used in the evaluation.

1.2 Regularization

In supervised learning, the performance of a model m is typically measured as a comparison of the model's prediction $y' = m(x)$ and a correct target value y (ground truth). If we use the same data to measure the performance of the model as we have used to train the model, we will get a positively biased result. For example, imagine a model that stores all the training examples it is shown. At the evaluation, it either knows the correct target (it saw that data point during training) or randomly guesses the target. If we measured its performance on training dataset, it would be perfect while its real-world performance would be poor. To overcome this limitation, we evaluate the model's performance on a separate set of data points, the testing dataset.

Since we use two datasets *training dataset* and *testing dataset*, we can compute two performances:

$$P_{train}(m) = \frac{1}{|TRAIN\_SET|} \sum_{(x,y)\in TRAIN\_SET} p(y', y)$$

$$P_{test}(m) = \frac{1}{|TEST\_SET|} \sum_{(x,y)\in TEST\_SET} p(y', y)$$

where m is a model, $y' = m(x)$ is models prediction and $p(y', y)$ is a performance measure on one data point.

P_{train} tells us, how well our model captures the structure of the training data. If P_{train} is insufficient (model is under-fitting), our model is not complex enough to learn the structure of the data. When our model is under-fitting we want to scale up its complexity (the complexity of the set of functions it is capable of modelling).

If there is a big delta between P_{train} and P_{test}, it is called over-fitting. When a model over-fits, it does not learn to solve the task in general, only on the data points in the training dataset. It is not capable to generalise to unseen data points and is not useful for real-world applications. While the best way to fight over-fitting is to collect new data points, it is usually not feasible. The other options are the decrease of the model's complexity, data augmentation (automatic expansion of dataset) [27] [26], and **regularization**.

In the stochastic gradient descent training, we compute the loss of the model on some data points, then we compute the gradient of the loss with respect to the trainable parameters and we adjust the trainable parameters according to the gradient. We mentioned the loss of the model, not the performance because they can differ. One of the reasons why they can differ is that the performance measure may not be differentiable or its gradient might bare no useful information for the gradient descent training (e. g. accuracy). The other reason can be regularization.

If we optimize a network to maximize its performance on the training dataset, it can overfit. Optimally we would want to optimize the model's testing performance, but since the model can not see the testing dataset during the training, we can not do that in any straightforward fashion. We can try to use some sophisticated ways to at least partially accomplish that goal. For example, if we can compute the model's complexity (in a manner that is useful for the gradient descent), we can define our loss as a sum of the model's error and the model's complexity. Thereby we would optimize the model to have as good training performance as possible while being as simple as possible. This in turn could increase its testing performance, since the simpler model should not be able to memorize the training dataset and it should be forced to learn a general rule to solve the problem.

Another example is *weight decay* (L_2 regularization) [14]. Weight decay penalizes the model for using big weights. Its penalty is computed as:

$$L_2(m) = \frac{1}{|W|} \sum_{w \in W} w^2$$

, where W is a set of weights we want to regularize. It forces the network to use small weights. If we have a network with small weights, a little change in input usually means a little change in output. Therefore the model's function is smoother and that usually means better generalization, see figure 1.3 for the visualization.

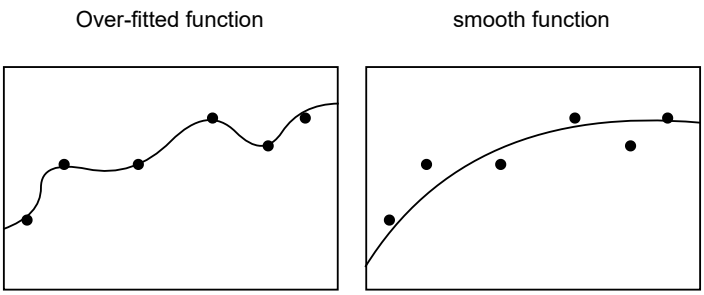

Figure 1.3: While the over-fitted (left) function achieves smaller training loss, the smoother (right) function captures the underlying structure of the data and will probably achieve better testing performance.

1.2.1 L_0 regularization

Relaxed version of L_0 regularization is a key piece of our training strategies. It was introduced by Louizos et al. [17] and is derived from the L_0 norm:

$$||x||_0 = \begin{cases} 0 & \text{if } x = 0 \\ 1 & \text{otherwise} \end{cases}$$

Setting a linear/convolutional layer parameter to zero is equivalent to the absence of the parameter. When we apply the L_0 norm to the weights, we penalize the model for every nonzero weight - for every weight it uses. The fever parameters the network uses, the simpler it is.

The problem with L_0 regularization is its combinatorial nature, its optimization is tractable only for trivially small models. Also, it either has no derivative or its derivative is equal to zero. The derivative bares no useful information and the L_0 regularization can not be optimized by the gradient-based methods.

Louizos et al. presented a relaxed version of L_0 regularization. They have used L_0 gates, that can be opened anywhere on the interval $< 0, 1 >$, where 1 means fully opened and 0 means closed.

We can link L_0 gates either to a weight (to force a network to use as few weights as it needs), a neuron of a linear layer, a channel of a convolutional layer, a head of a transformer,...

If a gate is linked to weight and it is fully opened, it is equivalent to the absence of the gate, the model uses the weight and is penalized for it. If the gate is fully closed, it is equivalent to the absence of the weight. The model does not use the weight (its complexity is smaller) and is not penalized.

L_0 gates are modelled using a semi-continuous random variable. It has a nonzero probability to be equal to 0, a nonzero probability to be equal to 1, and is continuous on an interval $(0, 1)$.

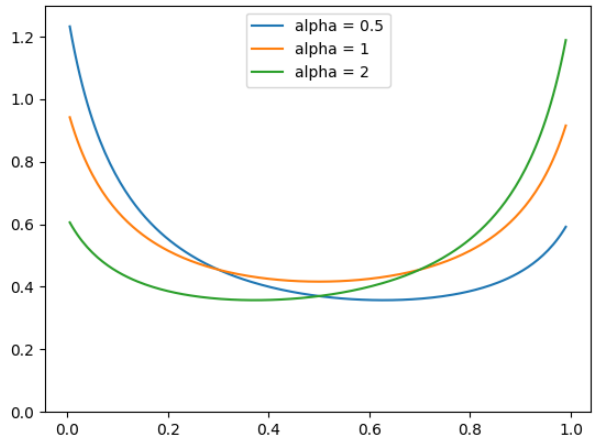

Figure 1.4: Plot of the hard concrete distributions with fixed $\beta = 0.5$ and various location $log\ \alpha$. We can see that location affects whether the gate is mostly opened or closed. It is the core parameter of the gate.

Let W be a matrix of linear layer's weights and b its bias. The ordinary output of the layer is $y = xW + b$. If we link the L_0 gate to every weight the output will be $y = x(W \odot G) + b$, where G is a matrix of realizations of the gates and $\odot$ is a point-wise multiplication. If we link the gates to the output neurons we get $y = (xW + b) \odot g$, where g is a vector of realizations, and if we link the gates to the input neurons we get $y = (x \odot g)W + b$.

We want to note, that we will use the same notation for the gate, its random variable, and the realisation of the random variable. While it may be formally incorrect, we presume that introducing multiple notations for similar concepts would bring undesirable complexity.

The L_0 gates are modelled by random variables, therefore the loss of the model is a random variable as well. Instead of optimizing the model's loss, we optimize its expected value. And instead of computing the exact expected value, we compute its approximation. But that is not atypical in machine learning, since stochastic gradient descent and mini-batch gradient descent also approximate the loss and the gradient of the loss.

In the original L_0 regularization we have penalized the model for using a weight. In the relaxed L_0 regularization we penalize the model for the probability of using a weight - the probability that the gate is not closed. It can be computed using its cumulative distribution function:

$$P[w \text{ is used}] = P[g_w > 0] = 1 - P[g \leq 0] = 1 - F_{g_w}(0)$$

where $P[a]$ is a probability of a being true, g_w is gate linked to the weight w, and F_{g_w}

is cumulative distribution function of the gate g_w.

The distribution function of the gate needs to hold two properties. Its cumulative distribution function and computation of its realization need to be differentiable. The gradient must contain useful information so the gradient-based methods are capable to optimize the L_0 loss.

Louizos et al. proposed a novel distribution called hard concrete. It is obtained by "stretching" and "rectifying" a binary concrete random variable [19] (continuous relaxation of Bernoulli random variable). Its realization can be computed as follows:

$$u \sim U(0,1)$$

$$s = Sigmoid\left((log\ u - log(1-u) + log\ \alpha)/\beta\right)$$

$$\bar{s} = s \cdot (\zeta - \gamma) + \gamma$$

$$z = min(1, max(0, \bar{s}))$$

where:

- u - random variable with uniform distribution on interval $(0, 1)$
- s - binary concrete random variable
- $\bar{s}$ - stretched binary concrete
- z - stretched and rectified binary concrete - hard concrete

Parameters $log(\alpha)$ and β are trainable, parameters γ and ζ are hyper-parameters (usually set to 1.1 and -0.1). Parameter $log(\alpha)$ is the location of the distribution and has analogical function as p in Bernoulli distribution, see figure 1.4. Parameter *beta* determines the level of relaxation. For $\lim \beta \to 0$ hard concrete is equivalent to Bernoulli distribution, while for the $\beta = 1$ the distribution is a straight line.

Since we compute the random variable as a modification of random noise, we can compute the gradient of parameters $log\ \alpha$ and β with respect to the loss and optimize them by gradient descent. The optimization of β has a few challenges though. For example $\beta = 0$ leads to division by zero and small difference between $\beta = \epsilon$ and $\beta = -\epsilon$ leads to big difference in the final realization. As a consequence, β is usually fixed. Another option is to not have β as a parameter, but instead introduce a new parameter δ and compute $\beta = Sigmoid(\beta)$, to make sure that β stays in the safe $(0, 1)$ interval. In our approach, we will use fixed β.

The probability of the gate being not closed (for the hard concrete) is equal to:

$$P[g \text{ is not closed}] = 1 - F_{g_w}(0) = Sigmoid(log(\alpha) - \beta\frac{-\gamma}{\zeta})$$

which is differentiable. For more details, such as derivation and proofs of previously shown facts and combination of L_0 regularization with L_2 regularization see paper [17].

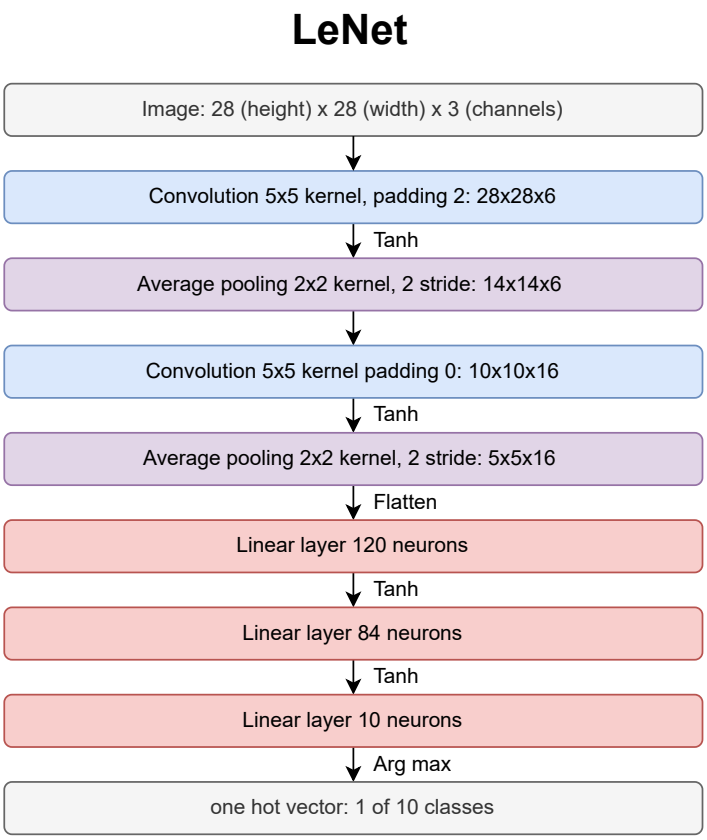

Figure 1.5: An exact LeNet architecture that will be used in experiments.

1.3 LeNet

One of the architectures that we will use to test our approach is *LeNet* specifically *LeNet-5* [15]. LeNet is a simple convolutional feed-forward network proposed in 1989. It is composed of two convolutional layers and three linear layers. Every convolutional layer is followed by average pooling.

Originally LeNet was proposed with Sigmoid as an activation function. But Sigmoid is not compatible with modern initialization. Modern initialization tries to retain the mean of the outputs of the layer around zero (if inputs have a mean around zero). Sigmoid breaks this idea, since $Sigmoid(0) = 0.5$, see figure 1.1. Therefore we have replaced Sigmoid with Tanh.

LeNet has no adaptive pooling or other techniques to cope with the variable size of images. Therefore the sizes of layers vary from problem to problem, based on the shape of the input. For the exact architecture that we will use see figure 1.5.

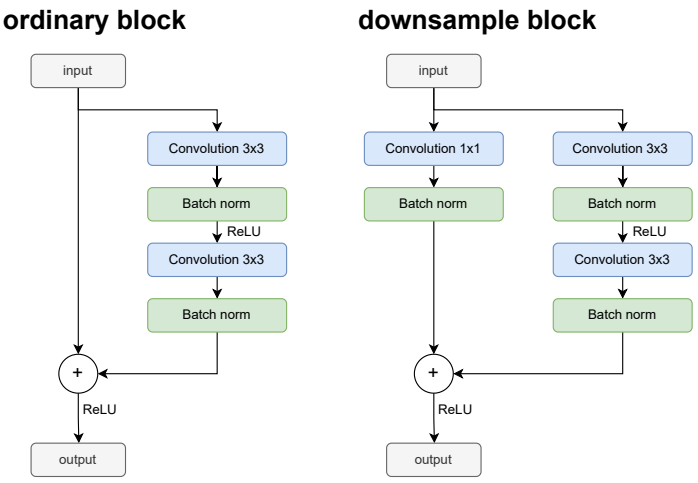

Figure 1.6: On the left side, we can see an ordinary block. On the right side is a downsample block. It is used to change the shape of the data. Instead of an identity, it has a convolutional layer with a 1x1 kernel followed by a batch norm. The convolutional layer offers an option to change all three dimensions of the data - height, width, and number of channels.

1.4 ResNet

ResNet was presented in 2016 by He et al. [7]. The main advantage of the architecture is its depth. Compared to LeNet, it uses three ideas to overcome the problems connected to the training of deep neural networks:

- ReLU activation
- batch normalization
- **skip connections**

The core structure of ResNet is a block. There are few variations of the block, we will present the most common. Block is composed of two convolutional layers c_1 and c_2 and two batch norm layers bn_1 and bn_2. The function modeled by a block is:

$$block(x) = ReLU(\boldsymbol{x} + bn_2(c_2(ReLU(bn_1(c_1(x))))))$$

As highlighted in the equation, the block has two computational flows. One goes through the layers, while the other skips the layers. The later computational path is called *skip connection*, see figure 1.6.

Besides ordinary blocks, ResNet is also composed of downsample blocks. The goal of the downsample block is to change the shape of the data (usually to decrease the height and the width, and increase the number of channels). In downsample block the skip connection can not be an identity, because of the miss-match of the shapes. It

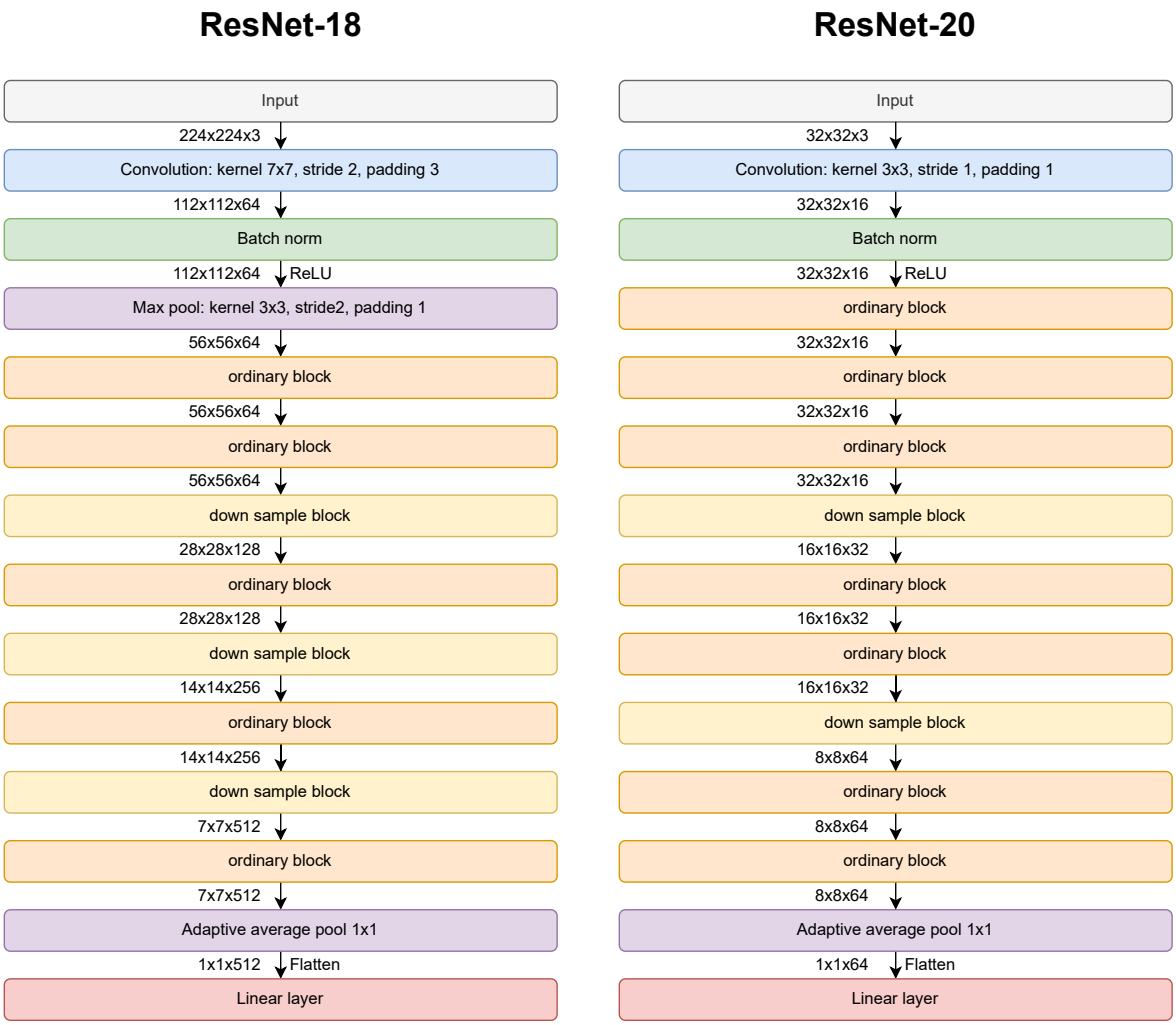

Figure 1.7: ResNet-18 and ResNet-20. The number says how many convolutional/linear layers are in the architecture, disregarding 1x1 downsample convolutions. We can see the shape of the flowing data on the left side of the arrows.

is composed of a convolution c_{ds} with kernel $1x1$ and batch norm bn_{ds}. The function modelled by a downsample block is:

$$down\_sample\_block(x) = ReLU(bn_{ds}(c_{ds}(x)) + bn_2(c_2(ReLU(bn_1(c_1(x))))))$$

There are many ResNet architectures. Because of the computational constraints, we will ResNet18 and ResNet20 (the smaller ResNet networks). The difference in the architectures is a result of the difference in the datasets they are used for. While ResNet-18 is built for images with shape 224x224, ResNet-20 is used for images with shape 32x32. ResNet-20 uses fewer channels and only two downsample blocks compared to the three that are in ResNet-18, see figure 1.7.

Chapter 2

Related work

Pruning and **knowledge distillation** are two main methods that utilize a teacher (already trained model) to "train" a student, typically a smaller model. The goal is to decrease the size of the model without significantly decreasing its performance.

There are many reasons why we want to decrease the size of the network. Mainly to decrease the storage requirements, increase the speed of evaluation, and decrease the energy consumption [3].

2.1 Pruning

Pruning "trains" the student by modifying the teacher, i.e. by cutting out some of its parts while keeping others. The important decision in pruning is the granularity, the smallest part/structure of the network you can cut out. **Weight pruning** is the most general approach, it prunes individual weights in linear layers or kernels. In **channel pruning** we are pruning whole neurons (with all their input and output weights) in the linear layer or channels in the convolutional layer.

2.1.1 Weight pruning

Weight pruning is the most general approach and channel pruning is its constrained version. So in the theory weight pruning is stronger. But in the practice, its generality brings challenges. It is harder to find the optimal solution and the result of the pruning is **sparse neural network**.

While sparse neural networks are a superset of dense networks and they are more akin to the natural neural networks, we do not have efficient algorithms and data structures to profit from the reduced number of weights. For example, we can store weights of the linear sparse layer as a sparse matrix (and save space if it is sparse enough), but the evaluation will probably be slower (current hardware is optimized for dense tensor multiplication). On the other hand, we can have a sparse matrix stored

as a dense matrix with many zeros. This approach may be a bit faster (multiplication by zero is faster than multiplication by nonzero float), but it does not save any space.

The finding of the optimal solution has combinatorial nature and is intractable for non-trivial networks. It is analogical to feature selection and we use similar greedy algorithms and heuristics to solve this problem.

In the iterative greedy approach, we either go top-down and prune weights or bottom-up and decide which weights to keep one by one until we satisfy the stopping condition. There are multiple tactics to choose one weight in an iteration. We can compute the loss/gain in the performance if we prune/keep the weight. But most of the time that is infeasible since the complexity of the whole algorithm is quadratic with respect to the number of the weights. By complexity, we mean the number of passes through the dataset. Another approach is to approximate the drop/gain in the performance based on the gradient of the loss [22, 21]. The gradient is computed for all weights at once. The complexity is linear with respect to the number of weights.

The stopping condition is set based on the goal of the pruning. It can be the maximal allowed size of the network, minimal allowed performance, it can be based on the previous drop/gain, ...

Another approach to pruning is to use regularization. L_0 and L_1 regularizations push the weights of the model to zero ($L_1(x) = |x|$). If the weight is zero, we can prune it without changing the function of the model. L_2 regularization also has the smallest penalization for zero weights, but it is not adequate for this problem. When we regularize weights by L_2 regularization, the model is incentified to use small but non-zero weights. It is more advantageous to use two small weights than one big and one zero:

$$L_2(0.5) + L_2(0.5) = 0.25 + 0.25 = 0.5 < 1 = 1 + 0 = L_2(1) + L_2(0)$$

Which is not the case for the L_0 and L_1 regularizations. We can control the size-performance trade-off by changing the multiplicator of the regularization.

2.1.2 Channel pruning

Channel pruning [22, 21, 18, 30] solves the problem with sparse neural networks at the cost of generality. We do not prune the individual weights, but we prune the neurons of the linear layers and channels of the convolutional layers. So instead of deleting elements from the weight matrix in the linear layer, we delete columns (or rows). Therefore the resulting weight matrix is still dense and the same holds for the network.

We can use the same iterative approaches as in the weight pruning. L_0 gates can also be linked to whole neurons and channels instead of the weights. Or even more

complex structures, such as transformer heads [28]. Therefore the L_0 regularization can be used to push the network to use only the important neurons and channels and the less important ones can be pruned.

The replacement of a L_1 regularization for channel pruning is *multi-task lasso*[23]. It is a mix of L_1 and L_2 regularization. Let W be a weight matrix of a linear layer, where $w_{i,j}$ is a weight between i-th output neuron and j-th input neuron. The multi-task lasso penalty of the layer is:

$$multi\_task\_lasso(W) = \frac{1}{|W|} \sum_j \sqrt{\sum_i w_{i,j}^2}$$

2.2 Knowledge distillation

Compared to pruning, knowledge distillation [9, 6] does not modify the teacher but uses it to train the student. The idea is that the bigger model may be more capable to extract the relevant information from the training dataset, or that it may be more robust against outliers and wrong labels (mistakes in the dataset).

In the basic knowledge distillation we train student to learn to predict ground truth as well as the predictions of the teacher. The student loss is composed of the error in prediction of the ground truth and the error in prediction of the teachers predictions:

$$loss(s) = \frac{1}{|TRAIN\_SET|} \sum_{(x,y) \in TRAIN\_SET} p(s(x), y) + \lambda \cdot p(s(x), t(x))$$

, $s(x)$ and $t(x)$ are predictions of the student and the teacher and $p(y', y)$ is a performance measure on one data point (e. g. mean square error or cross-entropy). Hyper-parameter λ is a ratio of the importance of the two parts of the loss. It controls whether student should focus more on the dataset or the teacher.

There are many variations of the basic knowledge distillation [6]. We can push the student to also learn the teacher's intermediate results, not only the final prediction. We can push the student to do a similar transformation in blocks/layers as the teacher, we can use multiple teachers, we can train teacher and student at the same time, ...

Chapter 3

Methods

We introduce two training strategies, that utilize labelled dataset as well as two teachers (already trained models). The goal is to train the best model with set architecture. There are two variants of the strategies. Either the teachers are part of the input, or we train them ourselves.

Both strategies are composed of three steps:

1. Training of two teachers (if they are not a part of the input)
2. **Merging**
3. Fine-tuning of the student

The first and the third steps are standard training of neural networks and are the same for both of the strategies. The second step is where the strategies differ. The **greedy strategy** uses the greedy merging and the **loss-driven strategy** uses the loss-driven merging.

3.1 Greedy merging

Greedy merging has two stages. In the first stage, we identify important parts of the teachers. In the second stage, we train student to mimic these parts.

What a part means depends on the layer. A part of the linear layer is a neuron and a part of the convolutional layer is a filter. Merging works the same way for both layers. For the lucidity of the text, we will explain how it works only for the linear layer, not in a general sense. To get an explanation of how it works for the convolutional layer it is enough to replace "neuron" with "channel" and "linear layer" with "convolutional layer".

Both stages are done in a greedy layer-wise fashion, hence the name greedy merging.

3.1.1 Identification of important neurons

The goal of this stage is to identify important neurons of teachers which will be later mimicked by the student. The identification is done in a greedy layer-wise fashion from the end (the output layer) to the start (the input layer).

We need to identify an exact number of neurons in every layer, so the requirements for the student's architecture will be met. E. g. if the student is supposed to have k neurons in the i-th layer, we have to select in total k important neurons from both teachers. It does not have to be $k/2$ from the first teacher and $k/2$ from the second.

The identification is done layer by layer. Let's say we have already identified the important neurons in the i-th layer and we want to identify important neurons in the $i-1$-th layer. Let's call important neurons in the i-th layer a target neurons, and the $i-1$-th layer analyzed layer. To identify important neurons in the analyzed layer we create a model that:

- simulates all layers of both teachers up to the analyzed layer (including)
- predicts values of target neurons
- has frozen weights in the first $i-1$ layers
- has L_0 gates linked to neurons in the analyzed layers
- is trained to minimize a loss that is composed of error loss and L_0 loss

Error loss forces the model to predict the values of target neurons while L_0 loss forces the model to use only half of the neurons in the analyzed layer, see figure 3.1.

Let G be a set of all gates. L_0 loss is equal to:

$$L_0\_loss(G) = \left(\frac{1}{2} - \frac{1}{|G|}\sum_{g\in G} P(g > 0)\right)^2$$

where $P(g > 0)$ is a probability that gate g will be opened. We can see, that the L_0 *loss* forces student to have:

$$\frac{1}{2} - \frac{1}{|G|}\sum_{g\in \mathbb{G}_\ell} P[g > 0]$$

equal to zero, which can be rewritten as:

$$\sum_{g\in \mathbb{G}_\ell} P[g > 0] = \frac{|G|}{2}$$

It forces the student to use at average half of the neurons. In an optimal scenario, the model will learn to use exactly half of the neurons with probability one. The choice of

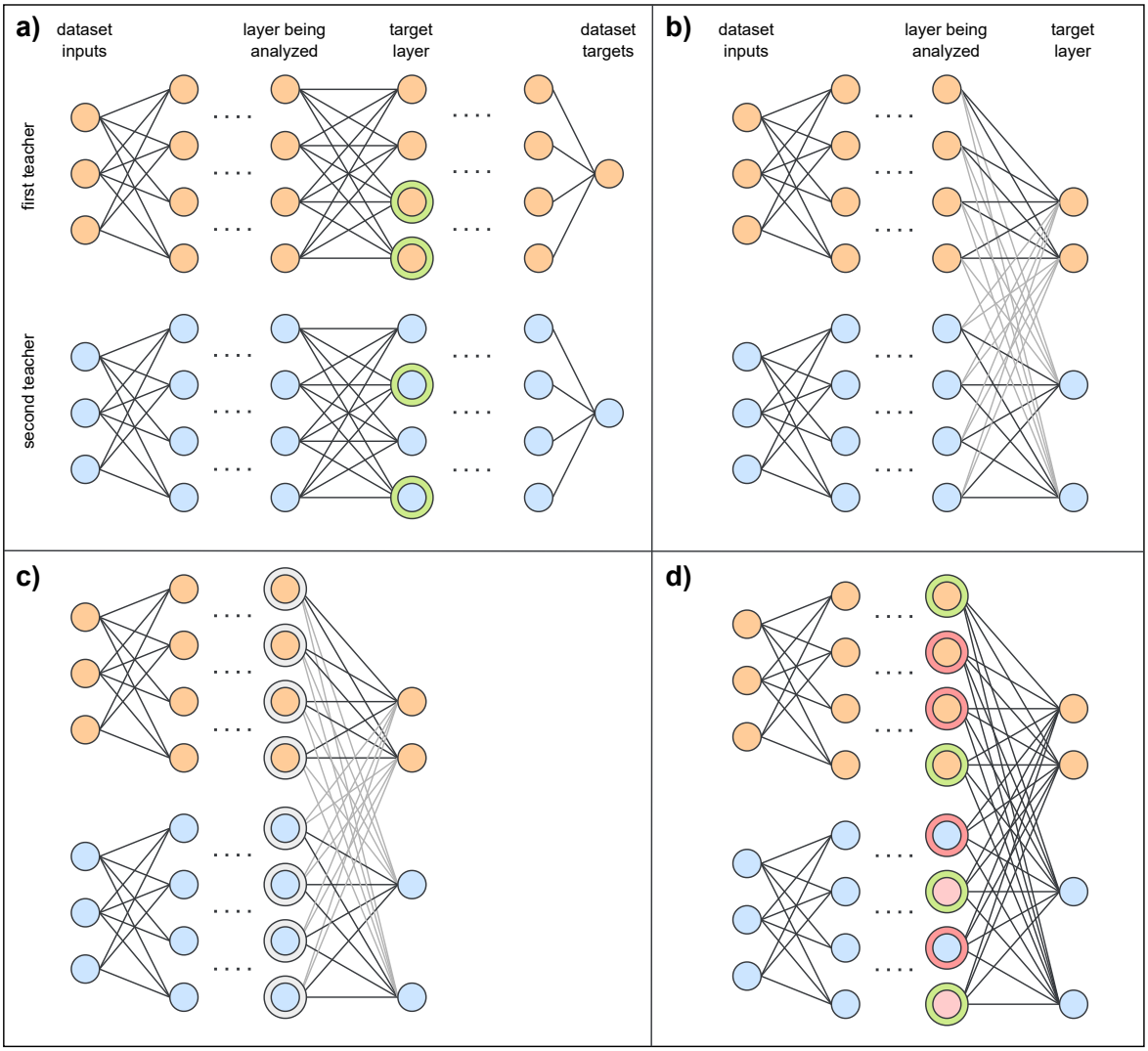

Figure 3.1: Identification of the important neurons. a) Start of the new iteration. Half of the neurons in the target layer are selected as important (highlighted by green background). b) We have gotten rid of all layers after the target layer and of unimportant neurons in the target layer. Also, we have initialized cross-teacher weights to zero. c) We have linked L_0 gates to every neuron in the analyzed layer. L_0 gates are initialized to a neutral position, hence the grey colour. d) Result of the training. In the optimal scenario, half of the gates will be always opened and half will be always closed. We select the important neurons, based on the probability of the linked gate being opened.

the important neurons in the analyzed layer is based on the probability of the linked gate being opened.

The first iteration of the identification is special. Instead of using teachers' output neurons, we use the ground truth as the target values.

3.1.2 Train student to mimic important neurons

The second stage is again layer-wise. First, we train the student to mimic important neurons of the teachers in the first layer, then the second, third, ... To do so we use classical SGD.

During the training of the i-th layer, we do not change the previous layers. Therefore it is enough to evaluate the outputs of the $i-1$-th layer (inputs of the i-th layer) once. If we can store the inputs of the i-th layer the complexity of the second stage is linear, not quadratic, with respect to the depth of the network. A similar trick can be used to speed up the first stage as well.

If we do not need to speed up the second stage or we are unable to store precomputed inputs of the i-th stage, we can unfreeze the previous layers. It may lead to a little less greedy algorithm, but on the other hand, it may increase the complexity of the hyper-parameters (e. g. we may need a different learning rate for the previous layers).

3.2 Loss-driven merging

Loss-driven merging is composed of three stages:

1. Layer-wise concatenation of teachers into a big student
2. Learning importance of big student's neurons
3. Compression of the big student

3.2.1 Layer-wise concatenation of teachers into a big student

First, we create a big student by layerwise concatenating the teachers. The big student simulates the two teachers in two separate computational flows and averages their predictions. We call the model "big student" because it has a doubled width. To get the required architecture we will need to prune it.

This stage is just network transformation without any training, see Fig. 3.2. Concatenation of the linear layer and the batch norm layer is done in the feature dimension. Concatenation of the convolutional layer is done in the channel dimension, see code fragment 3.3.

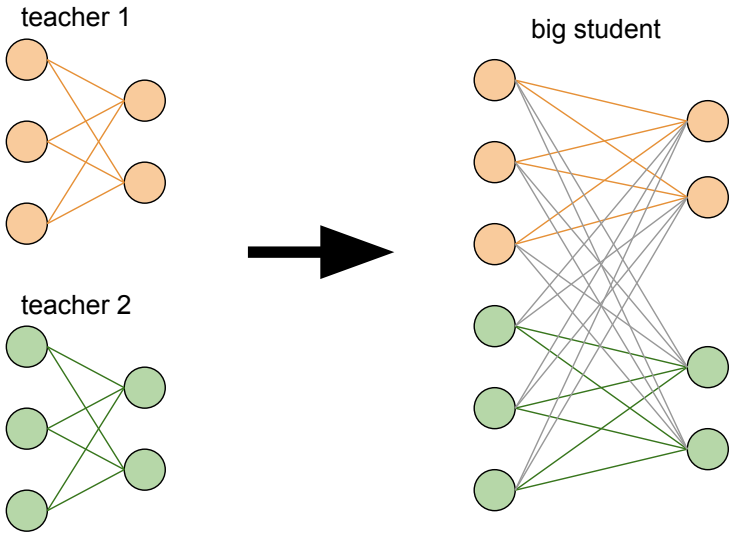

Figure 3.2: Concatenation of a linear layer. Orange and green weights are copies of the teachers' weights. Grey weights are initialized to zero. In the beginning, big student simulates two separate computational flows, but during the training, they can be interconnected.

```
def merge(conv1: nn.Conv2d, conv2: nn.Conv2d) -> nn.Conv2d:
    in_channels = conv1.in_channels
    out_channels = conv1.out_channels
    conv = nn.Conv2d(in_channels * 2, out_channels * 2,
                     kernel_size=conv1.kernel_size,
                     stride=conv1.stride,
                     padding=conv1.padding, bias=False)
    conv.weight.data *= 0
    conv.weight.data[:out_channels, :in_channels] = \
        conv1.weight.data.detach().clone()
    conv.weight.data[out_channels:, in_channels:] = \
        conv2.weight.data.detach().clone()
    return conv
```

Figure 3.3: PyTorch code for concatenation of a convolutional layer in a ResNet. Since convolutions are followed by batch normalization, they do not use bias.

3.2.2 Learning the importance of the big student's neurons

We want the big student to learn to use only half of the neurons in every layer, so after the removal of the unimportant neurons, we end up with the required architecture. Besides learning the relevance of neurons, we also want the two computational flows to interconnect.

Compared to the previous approach, we do not learn the importance layer by layer, but we learn it all at once. To do so we use similar L_0 loss simultaneously for all layers. It is equal to:

$$L_0\_loss(s) = \frac{1}{|\mathbb{L}|} \sum_{\ell \in \mathbb{L}} \left(\frac{1}{2} - \frac{1}{|G|} \sum_{g \in \mathbb{G}_\ell} P[g > 0] \right)^2$$

where s is a big student, $\mathbb{L}$ is a set of student's layers, $\mathbb{G}_\ell$ is a set of gates linked to the layer ℓ, and $P[g > 0]$ is a probability, that the gate g will be nonzero (the neuron will be used).

Same as in the previous strategy, the L_0 loss forces the big student to use at average half of the neurons in every layer. In an optimal scenario, the big student will learn to use half of the neurons with probability one, and not use the rest. In this scenario, we can compress the big student (prune the unused neurons) without affecting the performance.

Note, that our L_0_loss is different from the one used in [17]. While our forces the model to use exactly half of the neurons, theirs forces the model to use as few neurons as possible.

The complete loss we optimize the big student to minimize is:

$$loss(s) = error\_loss(s) + \lambda \cdot L_0\_loss(s)$$

where $error\_loss$ is an error in prediction (e. g. mean square error or cross-entropy) and λ is a hyper-parameter that controls the importance ratio of the two losses.

Hyper-parameter λ is sensitive and needs proper tuning. At the beginning of the training, it can not be too large, or the big student will set every gate to be opened with a probability of 0.5. At the end of the training, it can not be too small, or the big student will ignore the L_0_loss in favour of the $error\_loss$. It will use more than half of the neurons and it will significantly drop performance after we prune it. We found that using the quadratic increase of λ during big student's training works sufficiently well, see figure 3.4.

We have implemented the L_0 gates as a separate layer, not as a part of the linear (convolutional) layer. This allows us to variable position the L_0 gates, which is crucial in some scenarios. If we would position the L_0 layer before batch normalization, batch normalization would counter its effect $bn(x) = bn(x \cdot g)$ (for non-zero g). As a result, all non-zero gates would act as fully opened. The gradient of the gate g would be multiplied by g^{-1} and could explode.

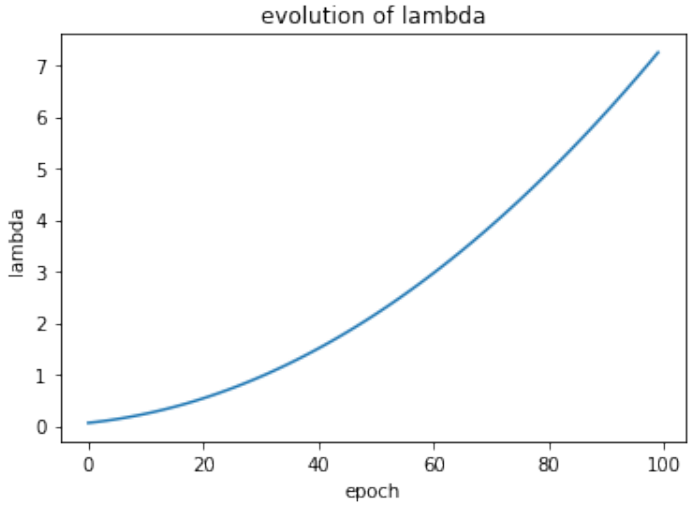

Figure 3.4: Evolution of λ during the training of the big student. It is problem sensitive and needs to be properly set. For the sine problem (presented later as one of the experiments) we have used $\lambda_{t+1} = \lambda_t + 0.05 * \sqrt{\lambda_t}$, where λ_t is used in the t-th epoch.

Also positioning the L_0 layer after ReLU activation can speed up the computation without having any effect on learning or result. The reason is that the output of the ReLU has a higher chance to be zero (compared to the input) and the multiplication by zero is faster. For the positioning of the gate layers in architectures that will be used in experiments see figure 3.5.

The output of the 2d convolutional layer has four dimensions (batch, height, width, channels), while the output of the linear layer has two (batch, neurons). As a result, we can not use identical L_0 layers. For the details of the implementation see appendix.

3.2.3 Compression of the big student

The result of the compression of the big student is a **student**, model with the required architecture. Compression is done by pruning half of the neurons of every layer. The neurons to be cut out are selected based on the probability, that the linked gate is open. Again, this stage is only network transformation without any learning. See figure 3.6 for the visualization.

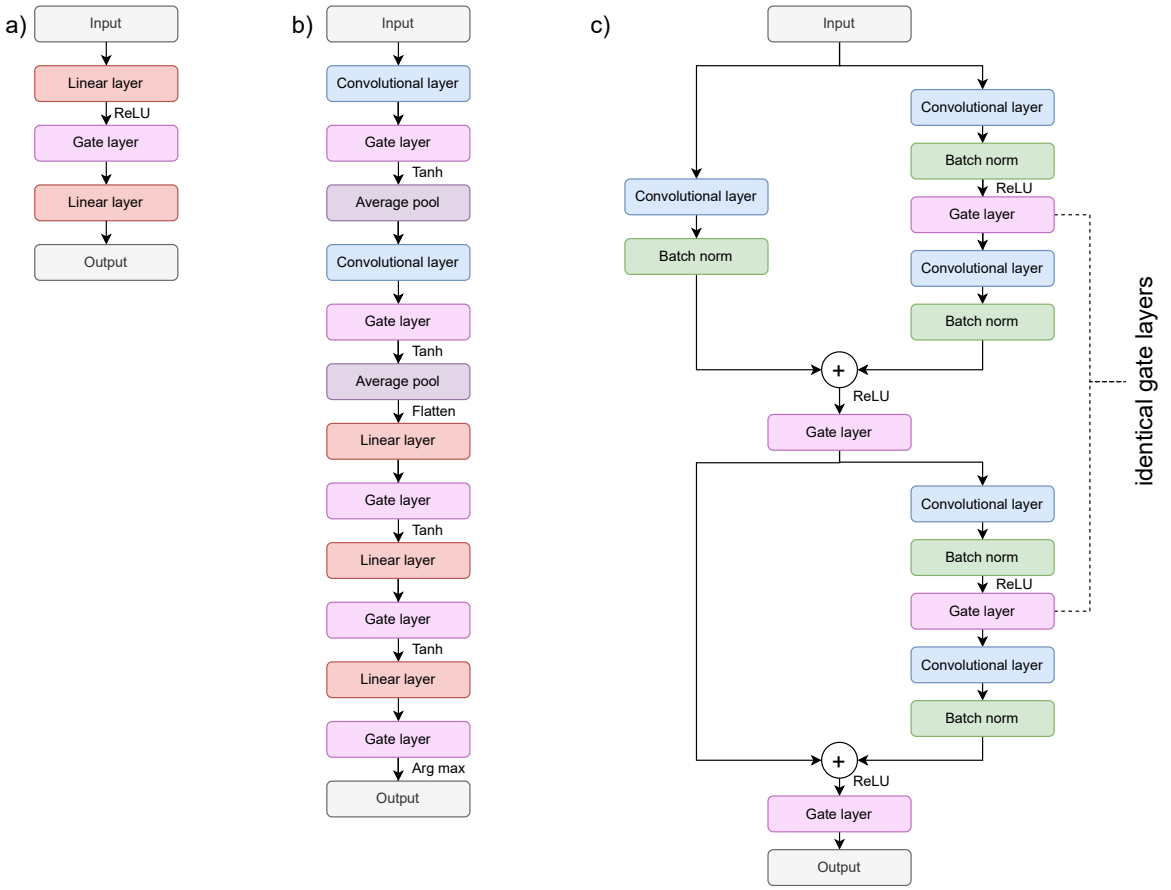

Figure 3.5: Positioning of the gate layers. a) simple fully-connected network b) LeNet c) two consecutive blocks of ResNet-18. Two of the gate layers have to be identical (the same parameters and realizations of random variables). If the gate layers would not be identical and for some channel i, the first gate would be closed while the second gate would be opened, the output value of the i-th channel would be $0 + f(x)$ instead of $x_i + f(x)$ which would defeat the whole purpose of skip connections.

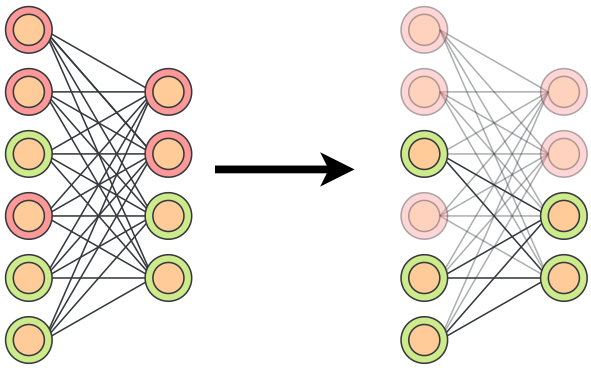

Figure 3.6: Compression of the big student. On the left side is a linear layer of a big student. The gate layers decide which neurons are important based on the probability they are used. On the right side is a compressed layer, it consists only of the important neurons.

Chapter 4

Experimental results

Right at the start of the chapter, we need to say, that we will only present the results of the loss-driven strategy. The reason is that we were not able to make the greedy strategy work. There was no difference between training student with a greedy strategy, or training a new model using only the third step of the strategy (classical SGD training). Also, the student's performance after the second stage was significantly worse than the performance after the third stage.

The strategy did not work either because its concept is flawed (e. g. the first stage is too greedy), or because our implementation was flawed. When we were working on the loss-driven strategy (after the greedy strategy has failed), we learned some new tricks. The implementation of these tricks (e. g. non-static λ) to the greedy strategy could make it work and is a potential idea for the subsequent research.

First, we have tested a loss-driven strategy on a simple artificial regression problem to demonstrate that it can learn better features than standard training. Then we tested it on the Imagewoof dataset (Imagenet-1k using only 10 classes) [11, 25] with LeNet [16] and ResNet-18 [8] architectures. We have also tested our approach on the CIFAR-100 dataset [13] using ResNet-20 [8]. Finally, we have evaluated our approach on the Imagenet-1k dataset [4].

In all experiments except full Imagenet-1k, we compare our strategy with the *bo3 model* strategy and *one model* strategy. Strategy bo3 model trains three models (all with the same amount of epochs) and picks the best. Strategy one model uses the whole training budget on one model. All strategies have the same training budget (some number of epochs). In our strategy, we use two-thirds of epochs to train teachers and one-third to train student (one-sixth to merge the teachers and one-sixth to fine-tune).

To make the comparison as fair as possible, all hyper-parameters in all strategies are either the same or similar. For example, the teachers in our strategy are trained the same way as the first two models of the bo3 model strategy.

In the table 4.1 we present a summary of all experiments except full Imagenet-1k.

Task	Strategy	Min	Max	Median	Mean	Std
Sine problem	student	**0.049**	**0.116**	**0.078**	**0.077**	**0.015**
dense network	bo3 model	0.105	0.373	0.253	0.240	0.076
test loss	one model	0.053	0.363	0.281	0.249	0.097
Imagewoof	student	**0.380**	**0.411**	**0.399**	**0.399**	0.008
LeNet	bo3 model	0.369	0.394	0.387	0.385	**0.007**
test acc	one model	0.365	0.397	0.377	0.378	0.010
Imagewoof	student	**0.824**	**0.828**	**0.825**	**0.826**	**0.002**
ResNet18	bo3 model	0.801	0.809	0.807	0.806	0.003
test acc	one model	0.810	0.818	0.813	0.813	0.003
CIFAR-100	student	**0.688**	**0.691**	**0.688**	**0.689**	0.002
ResNet20	bo3 model	0.670	0.672	0.670	0.670	**0.001**
test acc	one model	0.670	0.679	0.675	0.675	0.004

Table 4.1: Summary of experimental results. Considering a specific task, all strategies used the same number of epochs. Strategy *student* (our strategy) used 2/3 of epochs to train two teachers and 1/3 to train student. Strategy *bo3 model* trained three models (all for the 1/3 of epochs) and picks the best. Strategy *one model* uses all epochs to train one model. We can see, that our strategy has meaningfully better performance in all experiments, on all architectures.

The training on full Imagenet-1k costs significantly more and we could not afford to do the same type of an experiment as on the previous datasets. The results of the experiment on the full Imagenet-1k are shown in the table 4.2.

4.1 Sine problem

We have generated an artificial dataset, five sine waves with a noise. The input is a uniformly distributed scalar $x \sim \mathcal{U}(0, 1)$, and the target is $y = sin(10 \cdot \pi \cdot x) + z$, where $z \sim \mathcal{N}(0,\ 0.2)$, see figure 4.1. Our architecture is composed of two linear layers, i. e. one hidden layer with 100 neurons. In this experiment, we want to confirm the idea, that network trained from a random initialization might end up in a worse local optimum compared to our strategy.

For the placement of the gates in the big student, see Fig. 3.5. In every strategy, we have used 900 epochs and SGD with starting learning rate 0.01 and momentum 0.9. We have decreased the learning rate to 0.001 after the 100th epoch of the fine-tuning student, the 250th epoch of the training of the teachers and bo3 models, and the 800th epoch of the training in one model strategy.

We have trained 50 models with every strategy. Our strategy has significantly

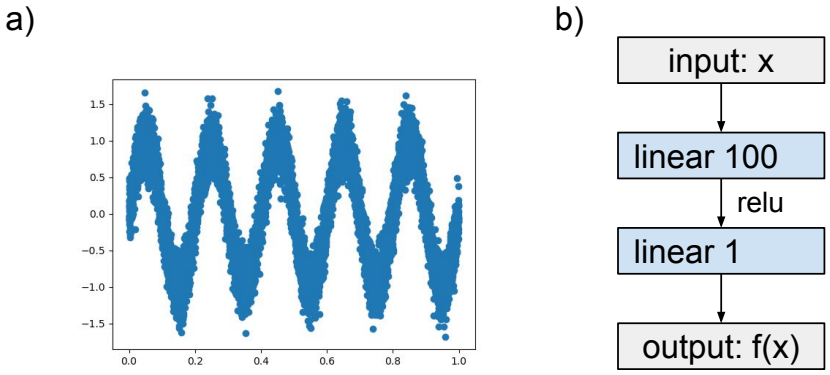

Figure 4.1: a) Training dataset for the sine problem consisting of 10000 samples: $x \sim \mathcal{U}(0,1)$; $y = sin(10 \cdot \pi \cdot x) + z$; $z \sim \mathcal{N}(0, 0.2)$. b) Architecture of a network used for the sine problem.

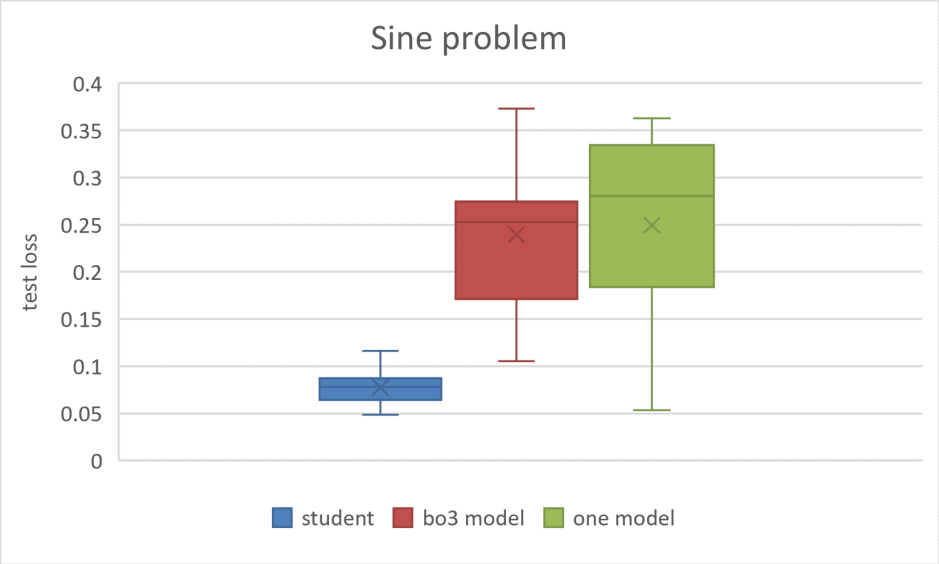

Figure 4.2: Box plot of the testing loss; 50 models for every strategy. The horizontal line inside the box shows the median, and the cross represents the mean.

smaller error than other strategies. For a more comprehensive comparison see table 4.1, for the visualization of test losses see figure 4.2 and for the visualization of the learned curves see figure 4.3. It shows, that student avoids local collapses and better fits the sine peaks.

4.2 Imagewoof

Imagewoof [11] is a subset of Imagenet (10 classes of dog breeds). On this recognition problem, we have tested two architectures - LeNet and ResNet-18.

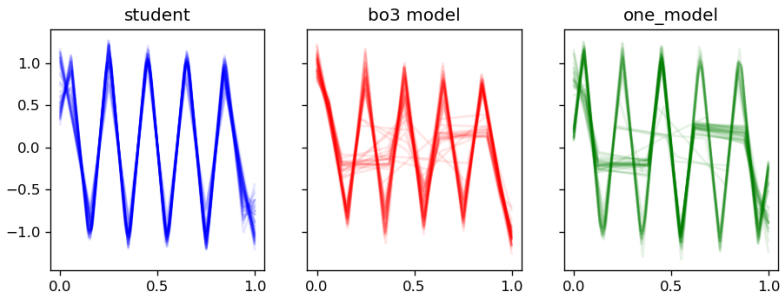

Figure 4.3: A plot of models' learned curves. One line represents one model, and there are 50 lines for every strategy. As we can see, every student has learned all middle peaks, which is not the case for other strategies. The border peaks learned by students are comparable to one model strategy, while they are worse for the bo3 strategy.

4.2.1 LeNet

We have already presented the LeNet architecture in chapter Preliminaries. We have also mentioned that we have replaced a Sigmoid with Tanh. While their shape is similar, their main difference is that Tanh has a stable point in zero $Tanh(0) = 0$, while Sigmoid does not $Sigmoid(0) = 0.5$. Throughout the thesis, we have already mentioned some advantages of zero being a stable point, e. g. compatibility with modern initialization or deeper stable networks. Another advantage of zero being a stable point is compatibility with L_0 gates.

When the gate layer is positioned before the activation function and a neuron has its gate closed (the value of the gate is zero, so also the value of the neuron is zero), the stable point keeps the value of the neuron at zero. If the output value of a neuron is zero, it does not affect the computation of the following layer (it is equivalent to not being a part of the computation). If some neuron has its gate always closed, we want that neuron to not affect the computation of the network. So when we cut it out, the network's function will not change and the performance will not drop. If the activation function does not have a stable point in zero, it breaks this property.

In chapter Methods we have mentioned, that we can place the gate layer after the ReLU activation function. We could do it because the relative position of ReLU and gate layer does not affect the result:

$$ReLU(x \cdot g) = ReLU(x) \cdot g \ ; \ x \in \mathbb{R}, \ g \in < 0, 1 >$$

This property does not hold for all activation functions, e. g. for values $x = 5, \ g = 0.2$:

$$Tanh(x \cdot g) \approx 0.76 \ ; \ Tanh(x) \cdot g \approx 0.2$$

$$Sigmoid(x \cdot g) \approx 0.73 \ ; \ Sigmoid(x) \cdot g \approx 0.2$$

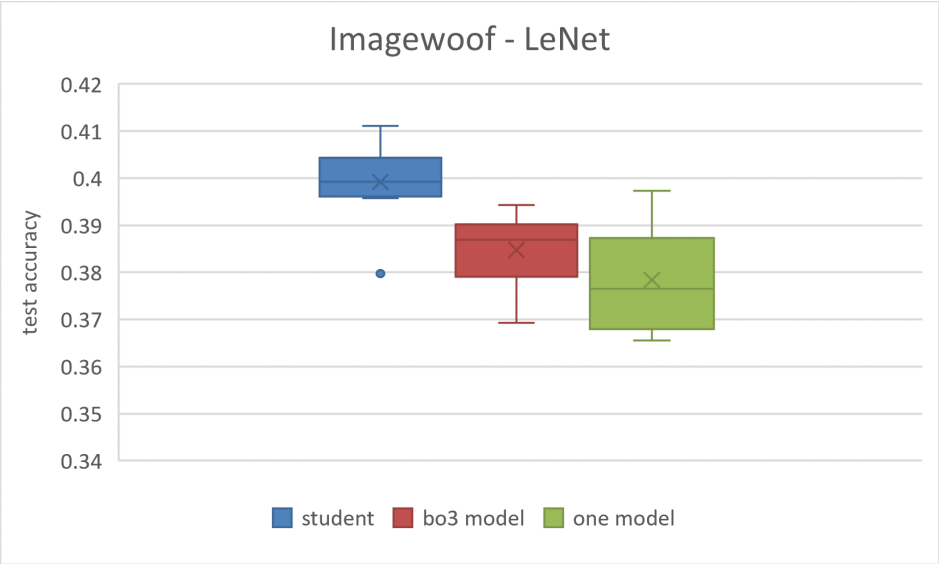

Figure 4.4: Box plot of testing accuracy; 10 models trained by every strategy.

So in this experiment, we have used Tanh instead of Sigmoid, and we have placed the activation function after the gate layer.

The training budget for this problem was 6000 epochs. All strategies have used SGD with starting learning rate 0.01 and momentum 0.9. Every training (teachers, student fine-tuning, bo3 models, and one model) except the learning of the important neurons decreased the learning to 0.001 in the third quarter and 0.0001 in the last quarter of the training.

We have trained 10 models by every strategy. See figure 4.4 for the visualization of accuracy and table 4.1 for detailed statistics. Our strategy has better min, max, median, and mean testing accuracy.

4.2.2 ResNet-18

As we have pointed out before in chapters Preliminaries and Methods, ResNet has two computational flows and we have to be careful to not break them with gate layers. For the exact placement of the gate layers and reasoning see figure 3.5.

The training budget was 600 epochs. The optimizer and the learning rate scheduler used are analogical to the LeNet experiment.

We have trained 5 models using every strategy. Our strategy had better results in every computed statistic. For the visualization of accuracies see figure 4.5, and for the detailed statistics see table 4.1.

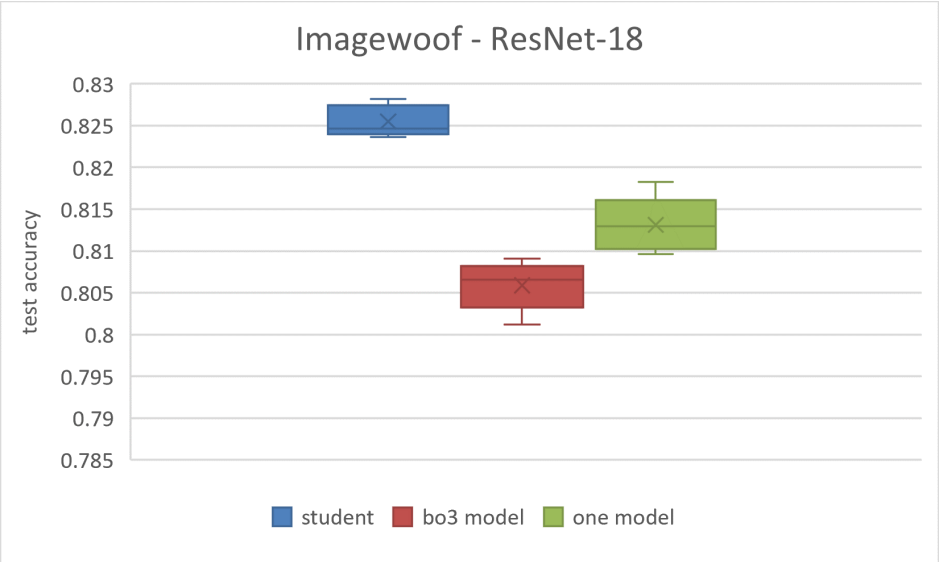

Figure 4.5: Box plot of testing loss; 5 models trained by every strategy. The worst student has better accuracy (0.819) than the best long teacher (0.818).

4.3 CIFAR-100

For the CIFAR-100 we have used architecture ResNet-20. Even though ResNet-20 and ResNet-18 have a similar number of layers, they were built for different shapes of the input. For the exact differences see figure 1.7.

The training budget was 900 epochs. We have trained models using SGD with a learning rate 0.1 in the first two quarters, 0.01 in the third quarter, and 0.001 in the last quarter of the training. For the visualization of accuracies see figure 4.6, and for the detailed statistics see table 4.1. We can see, that our strategy is more than 1% better compared to the one model strategy (better of the two opposing strategies).

4.4 Imagenet

We have also tested our training strategy on the Imagenet-1k dataset [4]. However as seen in [29], high-quality training requires 300 to 600 epochs, which is quite prohibitive for such a large data set. We have opted for an approach from Torchvision [2], which achieves decent results in 90 epochs. We have trained networks using SGD with starting learning rate 0.1. We decrease the learning rate by a factor of 10 after the first and the second third of the training. For the final fine-tuning of the student, we have used a slightly smaller starting learning rate 0.07.

In this experiment, we have used different allocation of epochs. We have trained teachers only for 20 epochs, and we have also used only 20 epochs in the merging step. The remaining 90 epochs were used for the fine-tuning. As a result, the teachers had

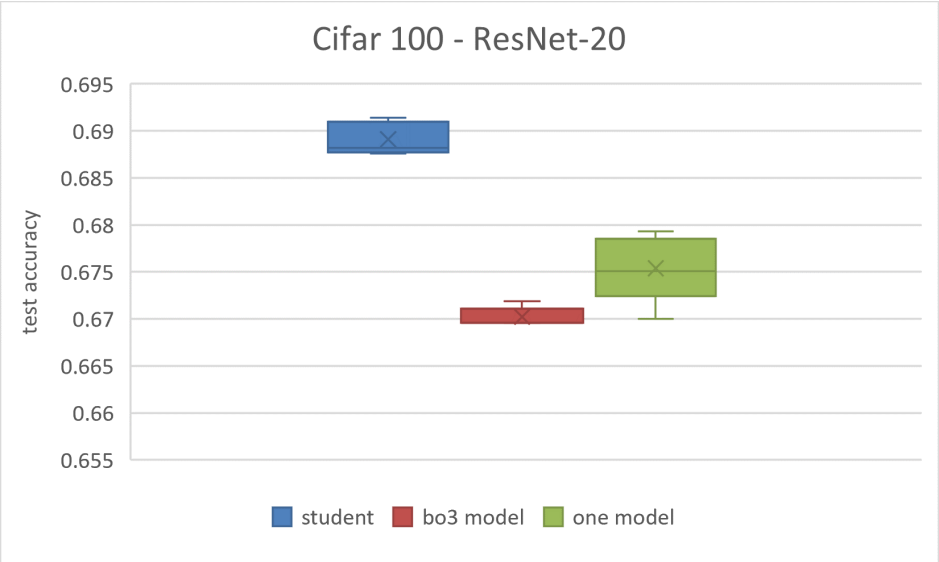

Figure 4.6: Box plot of testing accuracy; 5 models trained by every strategy.

Teacher epochs	Merging epochs	Fine-tuning epochs	Total epochs	Accuracy
-	-	90	90	0.6976
-	-	150	150	0.7028
20	20	90	150	**0.7047**

Table 4.2: Results on Imagenet benchmark with ResNet-18 architecture. Accuracy shown in table is a testing accuracy. Even though our model has better accuracy only by 0.2%, taking in the perspective the 0.5% gain in classical training by adding 60 epochs, it is not insignificant.

an accuracy of only 65%, but the final student had better accuracy as a classically trained model for 90 and 150 epochs. The results are summarized in table 4.2.

Conclusion

In chapter Methods, we have proposed two strategies for merging two trained neural networks. We can use them to merge already trained networks or start from scratch and train better networks than classical SGD with the same budget. In chapter Experimental results we have demonstrated the second use-case for the loss-driven strategy.

Our loss-driven strategy was significantly better than the *bo3 strategy* and *one model* strategy on every problem we have tried. Also, we did not have to customize the architecture of the network or hyper-parameters such as learning rate, momentum, learning rate scheduler, ... The only problem-specific excessive work was setting up the hyper-parameter λ.

Even though all strategies had the same training budget, the comparison was not fully fair from a practical standpoint. An epoch of our merging is slower than an epoch of classical training. The main reason is that we generate a random number for every realization of the gate. We believe it is not necessary. Reducing the amount of generated random numbers could increase the speed without decreasing the performance. Optimization of the speed of merging is a potential research direction.

At the beginning of the chapter Experimental results, we have noted that we were not able to make the greedy merging work. Networks trained by the greedy strategy did not have better results than classical training. It could be a result of a flawed concept or of a wrong implementation. Implementing the greedy strategy with new tricks, that we have learned while working on the loss-driven strategy (such as non-static λ and positioning of the gate layer) could make the greedy strategy work and is another possibility for future work.

In both strategies, we have always used two teachers, but the strategies can be generalized to utilize more. Exploring the effect of the number of teachers is also a potential research direction. In addition to that, we can try to use hierarchical merging. Create two students by merging four teachers, and then use the two students as teachers to train the third student.

We can also use our merging strategy in the transfer learning setting. We can try merging teachers trained on different data sets or different tasks.

Bibliography

[1] Convolutional Layer. https://colab.research.google.com/drive/1S8SJvH4bqhPvurG4gjh3-t-XulX4S8JX. Accessed: May 5, 2022.

[2] Torchvision. https://pytorch.org/vision/stable/index.html. Accessed: May 5, 2022.

[3] Davis W. Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John V. Guttag. What is the State of Neural Network Pruning? *CoRR*, abs/2003.03033, 2020.

[4] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: a Large-Scale Hierarchical Image Database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

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

[6] Jianping Gou, Baosheng Yu, Stephen John Maybank, and Dacheng Tao. Knowledge Distillation: A Survey. *CoRR*, abs/2006.05525, 2020.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *CoRR*, abs/1512.03385, 2015.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *CoRR*, abs/1512.03385, 2015.

[9] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the Knowledge in a Neural Network, 2015.

[10] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer Feedforward Networks are Universal Approximators. *Neural Networks*, 2(5):359–366, 1989.

[11] Jeremy Howard. Imagenette. https://github.com/fastai/imagenette. Accessed: May 5, 2022.

[12] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR*, abs/1502.03167, 2015.

[13] Alex Krizhevsky and Geoffrey Hinton. Learning Multiple Layers of Features from Tiny Images. Technical Report 0, University of Toronto, Toronto, Ontario, 2009.

[14] Anders Krogh and John Hertz. A Simple Weight Decay Can Improve Generalization. In J. Moody, S. Hanson, and R.P. Lippmann, editors, *Advances in Neural Information Processing Systems*, volume 4. Morgan-Kaufmann, 1991.

[15] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1(4):541–551, 1989.

[16] Yann LeCun, Bernhard Boser, John Denker, Donnie Henderson, R. Howard, Wayne Hubbard, and Lawrence Jackel. Handwritten Digit Recognition with a Back-Propagation Network. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann, 1989.

[17] Christos Louizos, Max Welling, and Diederik P. Kingma. Learning Sparse Neural Networks through L_0 Regularization, 2017.

[18] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression. *CoRR*, abs/1707.06342, 2017.

[19] Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables. *CoRR*, abs/1611.00712, 2016.

[20] Tom M. Mitchell. *Machine Learning.* McGraw-Hill, New York, 1997.

[21] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance Estimation for Neural Network Pruning. *CoRR*, abs/1906.10771, 2019.

[22] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning Convolutional Neural Networks for Resource Efficient Transfer Learning. *CoRR*, abs/1611.06440, 2016.

[23] Guillaume Obozinski, Ben Taskar, and Michael Jordan. Multi-Task Feature Selection. *Statistics Department, UC Berkeley, Tech. Rep 2.2.2*, 2006.

[24] David Picard. Torch.manual_seed(3407) Is All You Need: On the Influence of Random Seeds in Deep Learning Architectures for Computer Vision. *CoRR*, abs/2109.08203, 2021.

[25] Sam Shleifer and Eric Prokop. Using Small Proxy Datasets to Accelerate Hyperparameter Search. *CoRR*, abs/1906.04887, 2019.

[26] Connor Shorten and Taghi M. Khoshgoftaar. A survey on Image Data Augmentation for Deep Learning. *Journal of Big Data*, 6(1):60, Jul 2019.

[27] David A. van Dyk and Xiao-Li Meng. The Art of Data Augmentation. *Journal of Computational and Graphical Statistics*, 10(1):1–50, 2001.

[28] Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned. *CoRR*, abs/1905.09418, 2019.

[29] Ross Wightman, Hugo Touvron, and Hervé Jégou. ResNet Strikes Back: An Improved Training Procedure in Timm. *CoRR*, abs/2110.00476, 2021.

[30] Ruichi Yu, Ang Li, Chun-Fu Chen, Jui-Hsin Lai, Vlad I. Morariu, Xintong Han, Mingfei Gao, Ching-Yung Lin, and Larry S. Davis. NISP: Pruning Networks using Neuron Importance Score Propagation. *CoRR*, abs/1711.05908, 2017.

Appendix: electronic attachment

All the relevant source codes can be found in the electronic attachment. A potentially updated version can be found on `https://github.com/fmfi-compbio/neural-network-merging`.

The core script *l0module.py* consists of all important functions and classes needed for the work with L_0 gate layers and L_0 blocks. Script *student.py* shows how to create a network with L_0 gate layers. Script *train_student.py* is an implementation of our training strategy.