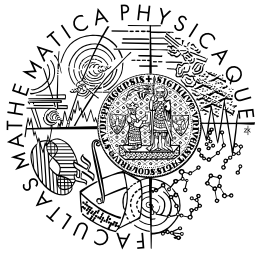


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Ondřej Hübsch

Reducing Number of Parameters in Convolutional Neural Networks

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis:	Mgr. Martin Pilát, Ph.D.
Study programme:	Computer Science
Study branch:	Theoretical Computer Science

Prague 2021

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

I would like to thank my supervisor, Mgr. Martin Pilát, Ph.D., for all his time, patience, guidance and insights that he gave me while I was working on this thesis.

I would not be able to complete this thesis without my wife, who watched over our two daughters whenever I was writing, kept me motivated and supported me the entire time.

I would also like to thank my parents for their support throughout my life.

Computational resources were supplied by the project "e-Infrastruktura CZ" (e-INFRA LM2018140) provided within the program Projects of Large Research, Development and Innovations Infrastructures.

Title: Reducing Number of Parameters in Convolutional Neural Networks

Author: Bc. Ondřej Hübsch

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Martin Pilát, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: In the current deep learning era, convolutional neural networks are commonly used as a backbone of systems that process images or videos. A lot of existing neural network architectures are however needlessly overparameterized and their performance can be closely matched by an alternative that uses much smaller amount of parameters. Our aim is to design a method that is able to find such alternative(s) for a given convolutional architecture. We propose a general scheme for architecture reduction and evaluate three algorithms that search for the optimal smaller architecture. We do multiple experiments with ResNet and Wide ResNet architectures as the base using CIFAR-10 dataset. The best method is able to reduce the number of parameters by 75-85% without any loss in accuracy even in these already quite efficient architectures.

Keywords: parameter reduction, convolutional neural networks, reinforcement learning, neural architecture search, pruning

Contents

1 Introduction **3**
- 1.1 Motivation . . . 3
- 1.2 Thesis Scope and Goals . . . 4

2 Background **5**
- 2.1 Machine Learning . . . 5
  - 2.1.1 General Concepts . . . 5
  - 2.1.2 Classification Problem . . . 6
  - 2.1.3 Gradient Descent . . . 6
- 2.2 Feedforward Neural Networks . . . 7
  - 2.2.1 Backpropagation . . . 9
  - 2.2.2 Fully Connected Layers . . . 10
  - 2.2.3 Convolutional Neural Networks . . . 11
- 2.3 Recurrent Neural Networks . . . 13
  - 2.3.1 Principle . . . 13
  - 2.3.2 Long Short-Term Memory Cells . . . 14
- 2.4 Reinforcement Learning . . . 15
  - 2.4.1 REINFORCE Algorithm . . . 15
- 2.5 Evolutionary Algorithms . . . 16

3 Related Work **18**
- 3.1 Architecture Search . . . 18
  - 3.1.1 Manual Architecture Design . . . 18
  - 3.1.2 Automatic Architecture Search . . . 22
- 3.2 Neural Network Pruning . . . 22
  - 3.2.1 Magnitude Pruning . . . 23
  - 3.2.2 The Lottery Ticket Hypothesis . . . 24
  - 3.2.3 Pruning Based on Hessian Matrix . . . 24
  - 3.2.4 Pruning by Evolution Strategies . . . 25
- 3.3 Knowledge Distillation . . . 26
- 3.4 Reduction of Arithmetic Precision . . . 27

4 Proposed Approach **29**
- 4.1 Overview . . . 29
- 4.2 Parameterizing Blocks of the Architecture . . . 29
  - 4.2.1 Decomposition of the Architecture . . . 29
  - 4.2.2 Scaling the Blocks . . . 31
  - 4.2.3 Block Replacement . . . 33
- 4.3 Optimization Stage . . . 34
  - 4.3.1 Common Properties . . . 34
  - 4.3.2 Evolutionary Approaches . . . 35
  - 4.3.3 Reinforcement Learning . . . 37
- 4.4 Final Training, Pruning and Fine Tuning . . . 39

5 Experiments **41**
5.1 Experiment Setup 41
5.2 Used Architectures 41
5.3 Baseline Experiments 42
5.3.1 Global Pruning 42
5.3.2 Optimization by Random Search 42
5.4 Evo and Evo-NSGA-II 43
5.5 Hyperparameters in Evo-W 46
5.6 Weight Initialization and Layer Freezing in Evo-W 49
5.7 Experiments with REINFORCE method 49
5.8 Comparisons to Existing Methods 53
5.8.1 Pruning Filters for Efficient ConvNets 53
5.8.2 Other Approaches 57
5.9 Overall Results 58

Conclusion **60**

Bibliography **62**

1. Introduction

Since the beginning of the deep learning era a lot of neural network architectures were developed. A common trend in their design is to stack a lot of layers and a lot of parameters to obtain a gain in accuracy. In this thesis we aim to go in the other direction: to take an existing architecture and make it smaller, perhaps sacrificing some of its performance in the process.

In this chapter we first describe our motivation in more detail and highlight some of the advantages of smaller network architectures. Then we specify scope of this thesis and a general direction that we will follow.

1.1 Motivation

In many practical scenarios we can choose from multiple models: some might have higher test set accuracy, while other models might be faster to evaluate or need less memory. Let us now only consider models that have an acceptable accuracy (or other similar metric) for our use case.

Among them, some might be very large (have many parameters) and we might not be able to use them in practice (e.g. directly on a mobile device) or even train them in acceptable time. Therefore, aside from the resulting accuracy, there are other technical factors that have to considered as well: hardware required to train the network in a reasonable amount of time, the training time itself, inference time, required memory footprint, storage requirements and perhaps many others, depending on the particular situation.

Some of these requirements may be more strict than others: we might need to deliver the trained model in some time frame and have only limited hardware available for the training, or we need it to run fast even on an old mobile device and without internet connection. In these cases, simply taking the state-of-the-art model with the best accuracy might not be an option, as the best models often require a lot of resources for their performance.

In such situations, we will likely have to settle for a smaller model that performs well-enough yet also meets our other criteria. Few other interesting advantages of smaller models are mentioned in SqueezeNet paper by Iandola et al. [16]: they allow for more efficient distributed training (the authors mention that communication among servers is often a bottleneck for scalability), there is less overhead when the use case requires sending updated weights to customers[1] and they are also easier to store directly on-chip, leading to faster evaluation.

The motivation for our thesis is based upon these two facts: that state-of-the-art approaches commonly train a large model and that there are use cases when size of the model matters. Therefore, we could take an existing architecture and modify it in a certain way to trade some of its performance for its practical usability. However, such modifications can degrade the model too much, so we have to be careful to avoid such situations.

[1] For instance, when Tesla sends over-the-air updates to their self-driving cars.

1.2 Thesis Scope and Goals

In this thesis we only consider reduction of parameters in convolutional neural networks and we will evaluate them on image classification task.

We decided to focus solely on approaches that take an existing network architecture and somehow amend it with the aim to reduce its size, specifically the number of its parameters, according to some target value. There is some overlap with general network architecture search, but our goal is not to construct entirely new architectures (whether by hand or by some algorithm), but rather to replace some parts of an existing architecture with more parameter-efficient alternative or to even remove some parameters without replacement.

We have chosen number of parameters as a proxy metric for both the speed of evaluation and storage requirements of the models: it should reflect both of these two primary objectives. Number of parameters is also easy to compute, and very objective: it does not depend on the used hardware or specific implementation details. Moreover, some of the existing approaches also optimize for it, or at least mention the resulting number of parameters. Finally, optimizing for some other objective instead should be relatively easy in the proposed methods and certain use cases may have a more suitable alternative.

Our goal is therefore to design a method (or methods) for optimizing existing architectures in such a way that we keep as close as possible to their original accuracy but reduce the number of parameters that is required, and to evaluate it and compare it to existing approaches.

We describe the proposed methods in Chapter 4 and then evaluate them in Chapter 5.

2. Background

In this chapter we give an introduction to machine learning concepts that are relevant for this thesis. More detailed information about each of these topics can be found in a book written by Goodfellow et al. [7].

2.1 Machine Learning

2.1.1 General Concepts

Machine Learning (ML) is a problem-solving approach that essentially tries to let the machine learn some problem structure from data that we have available. The assumption there is that the past can tell us something about the future - we can use the learnt problem structure to predict answers to inputs that we have not yet seen. This generally does not have to be the case: one classical example is that if somebody tells us that first five elements of certain sequence are 1, 2, 4, 8, 16, it does not necessarily mean that the sixth element is 32 (as the sixth element could have absolutely no connection to the previous five). However, machine learning can perform well for problems where it is reasonable to expect some connection between past and future data.

One typical problem reasonably solvable by machine learning might be this one: given a 32x32 pixel black and white image containing a single handwritten uppercase letter from the English alphabet, recognize what letter is in the image. This problem is often hard even for humans: different people might not agree on the letter that is supposed to be there. Theoretically, an engineer could handcraft some set of rules that might work reasonably well in practice, but it is clear that designing such set of rules would be a tedious task to do ("if there is a horizontal line approximately in the middle of the image, answer is one of A, E, R, P, F, G, H, B"). One very simple machine learning approach might be to collect a sample image of each possible letter, and then for new images look for the closest letter according to some metric (e.g. number of pixels that are the same).

More generally, we have some *training data* (ideally thousands of images of handwritten letters) and we want to train a model that would take an image and answer with a letter. Our hope is that the trained model would be able to answer correctly even for handwriting of person that did not contribute to the training data, i.e., it would *generalize* to new data (new styles of handwriting).

Our training data might or might not contain *labels* (correct answers) for the images. ML approaches are then often divided into *supervised learning* (we use the labels during training) and *unsupervised learning* (we don't have/use the labels). However the boundary between them might not be that clear - we might have labels for some part of the training input, and also some not labeled data that we want to use as well - then the commonly used term for that situation is *semi-supervised learning*.

Obtaining the labels usually requires a lot of manual work - some human has to manually label the images (unless the images are generated or there is some similar way to obtain them). Note that we might not necessarily need the labels to reasonably solve the sample handwriting problem as even without them we

could divide the images into 26 categories based on their "similarity" to each other. In other words, using some ML algorithm we would learn that "these two images contain the same letter" and obtain 26 equivalence classes. Then we could manually look at an example image from each equivalence class and define mapping between an equivalence class and corresponding uppercase letter and likely obtain a well-performing model.

2.1.2 Classification Problem

The handwriting example is a classification problem - the labels always belong to one of m categories. In this thesis we will always deal with classification problems and we will always have labels for all of our training data. In classification problems we have a data set $\mathcal{T}$ that contains $(\mathbf{x}, y)$: pairs of input vector $\mathbf{x} \in \mathbb{R}^n$ and integer label $y \in \{1..m\}$. We want to train a model f' that would predict a probability distribution $P(Y = y|X = x)$. When we then need a single label, we can take the one with the maximum predicted probability. We usually consider the training labels to be one-hot encoded vectors (i.e. $\mathbf{e_y}$ instead of y) - we will denote that by $\mathbf{y}$.

As we want to obtain model that *generalizes* well (is able to make reasonable predictions even on inputs not present in training data), the data set $\mathcal{T}$ is often split into three disjoint sets: *training set* (the part used only for training), *validation set* and *test set*. Internally, we can use validation set to guide training (to tune *hyperparameters* of algorithms that are used to optimize $\boldsymbol{\theta}$) and selection of "best model". We should not use the test set performance before reporting the final values (one exception might be checking if it is very bad). This is to limit the risk of indirectly optimizing $\boldsymbol{\theta}$ so that it performs well on the test set (as we want to use the test set to compare different models by possibly different people against each other). What could happen otherwise if we for example selected a model with best performance on the test set is that it could perform very bad on other data - it could indirectly *overfit* (performing well on a particular set of data but failing to generalize outside of it).

A common very simple performance measurement used to evaluate the models is *accuracy*. We can define accuracy on a subset T of $\mathcal{T}$ as:

$$\frac{|\{i \mid \arg\max f'(\mathbf{x}^{(i)}, \boldsymbol{\theta}) = \arg\max_j y_j^{(i)}, (\mathbf{x}^{(i)}, \mathbf{y}^{(i)}) \in T\}|}{|T|}$$

Note that accuracy can be very misleading - if there some label is very frequently occuring in the data (e.g. in 95% of data points), a very simple model that would always predict that label would obtain 95% accuracy. For many problems there exist publicly available collections of training and test data. In this thesis we will use the CIFAR-10 [19] dataset for training and measuring performance of our models. For it accuracy typically serves as a good proxy of model quality.

2.1.3 Gradient Descent

Gradient descent is a simple algorithm that was originally proposed by Cauchy in 1847: given some objective function $J(\boldsymbol{\theta})$ for which we want to find local

minimum, we can start with some initial vector $\boldsymbol{\theta}$ and keep making small steps in the negative direction of $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ (assuming that J is defined and differentiable in the neighborhood of $\boldsymbol{\theta}$). While accuracy is useful to compare different models, it is not very practical for training using gradient descent as it is not differentiable.

Therefore during training we usually aim to minimize expected *loss* over our data set $J(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x},\mathbf{y})}[L(\mathbf{x}, \mathbf{y}, \theta)]$ where L is a *loss function* for a single training data point. We can approximate the expectation by average loss over the training data. One commonly used loss function is *squared error* between output of model and label $\mathbf{y}$ (then approximation of $J(\boldsymbol{\theta})$ corresponds to Mean Squared Error (MSE) over the training data):

$$L(\mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) = ||f'(\mathbf{x}) - \mathbf{y}||_2^2$$

For classification problems currently the more commonly used loss function is cross entropy:

$$L(\mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) = -\sum_{i=1}^{m} y_i \log(f^{'}(\mathbf{x})_i)$$

Here m refers to the number of possible classes/categories.

Algorithm 1: One iteration of training by Gradient Descent

Input: current values of $\boldsymbol{\theta}$, learning rate α, training data $T = \{\{\mathbf{x}^(1), y^(1)\}, \dots\}$

accumulated_grad $\leftarrow (0, \dots, 0)$;

for $i \leftarrow 1$ **to** $|T|$ **do**

  accumulated_grad $\leftarrow$ accumulated_grad $+ \nabla_{\boldsymbol{\theta}} L(\mathbf{x_i}, y_i, \boldsymbol{\theta})/|T|$;

end

$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \cdot$accumulated_grad;

Because nowadays the training data can contain a lot of samples, a stochastic version of gradient descent (*Stochastic Gradient Descent*) is used to train the network - a random tiny[1] subset of training data called minibatch is chosen and used as T in the algorithm. Such approximation of the gradient of the loss function works well in practice. Typical implementation randomly shuffles the entire training data at the beginning, and then cycles through that shuffled data: first minibatch consists of first M data points, next M datapoints as next minibatch and so on. When we go through all the data, we can reshuffle the array or just consider it to be cyclic and continue, depending on the implementation.

2.2 Feedforward Neural Networks

Neural network is a computational model that is loosely inspired by how human brain works: they consist of multiple units called *neurons* and *connections* between some neurons. In the original model introduced by Rosenblatt [29] and its subsequent multilayer modifications, each neuron had a weight for each incoming neuron. To compute value of a neuron, values of incoming neurons were multiplied by their corresponding weight and then summed together. Each neuron

[1] Commonly about 100 data points

also had an *activation function* that was applied to the result of this weighted sum to obtain the value. See Figure 2.1 for an illustration of this concept.

However, nowadays the structure can be more complicated, so we will describe slightly more general version. A *feedforward neural network* can be any directed acyclic graph; we will then refer to nodes in that graph as neurons. Nodes without incoming edges (with in-degrees of 0) are called *input neurons*. Some neurons are considered as outputs of the network - *output neurons*. Each neuron x uses some internal function f_x to calculate its value from values of all neurons that are connected to x and from some internal weights (parameter vector $\boldsymbol{\theta}_x$ for f_x that will be adjusted during training).

To calculate the output of the network, we initialize values of input neurons to corresponding input value of the network. We will use a queue to keep track of neurons that are ready to compute their value (neurons for which all incoming neurons already calculated their value). We start by putting input neurons to the queue. When we pick a neuron from the queue, we calculate its final value, go across all connections from it and mark for the corresponding connected neuron that another one of their inputs has calculated value. Once we notice that for a neighboring neuron all its inputs are ready, we can add that neuron to the queue and proceed further (we keep the invariant that for every neuron we put to the queue all incoming neurons have computed their value). See Algorithm 2 for a description in pseudocode.

Algorithm 2: Forward propagation of a general neural network

Input: DAG $G = (V, E)$, array of initial values I for each input neuron
Output: Array Out containing computed values for all neurons
$Q \leftarrow$ initialize new queue;
for $i \leftarrow 1$ **to** $|V|$ **do**
  if *i is input neuron (has in-degree 0)* **then**
    $Out[i] \leftarrow I[i]$;
    Q.push(i);
  end
end
while *Q is not empty* **do**
  $node \leftarrow Q$.pop();
  if *Neuron node is not an input neuron* **then**
    denote all neurons connected into $node$ as $x_1, x_2, \ldots, x_k$;
    $Out[node] \leftarrow f_{node}(Out[x_1], Out[x_2], \ldots, Out[x_k], \boldsymbol{\theta}_{node})$;
  end
  for *for all edges going from node to u* **do**
    $ComputedInputs[u] \leftarrow ComputedInputs[u] + 1$;
    if *$ComputedInputs[u]$ equals in-degree of u* **then**
      Q.push(u);
    end
  end
end

The complexity of this algorithm is $O(V + E)$ where V is the number of vertices (neurons) in the graph and E the number of edges (connections). A similar

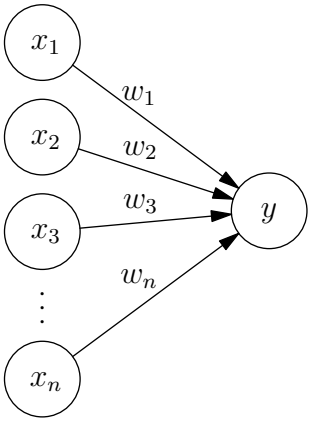


Figure 2.1: Neuron y with incoming connections from neurons $x_1, \ldots, x_n$. The value of y is calculated as $y = f\left(b + \sum_{i=1}^{n} w_i x_i\right) = f(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + b)$ where f is some scalar function. The calculated value can then be fed into neurons in subsequent layers or it can be one of the outputs of the network.

version of this approach is implemented in modern machine learning frameworks: an arbitrary computation is represented using a graph of simple arithmetic operations and the graphs are then called *computational graphs*.

2.2.1 Backpropagation

We described an algorithm to compute the output of the network for given input neuron values (called *forward propagation*). In order to train the network, some version of Gradient Descent algorithm as described in Section 2.1.3 is commonly used. For that we need to calculate the $\nabla_\theta L(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta})$ for a given $\boldsymbol{x}$ input and $\boldsymbol{y}$ expected output and from loss function L (we consider all the parameters the network uses as a parameter vector $\boldsymbol{\theta}$). For output neurons we can just directly compute the corresponding partial derivatives from the loss function. For other neurons, let us assume that we already know $\frac{\partial L}{\partial v_i}$ for all neurons v_i that neuron u is connected into (to simplify notation, by v_i we mean a variable corresponding to value of neuron v_i). According to the Chain rule we then have:

$$\frac{\partial L}{\partial u} = \sum_i \frac{\partial L}{\partial v_i} \frac{\partial v_i}{\partial u}$$

As we also know the logic that neuron u uses to compute its output, we can use Chain rule once again to calculate $\frac{\partial L}{\partial \theta_i}$ for all parameters θ_i that neuron u uses to compute its output:

$$\frac{\partial L}{\partial \theta_i} = \frac{\partial L}{\partial u} \frac{\partial u}{\partial \theta_i}$$

Using these insights, we can use almost the same algorithm as for forward propagation to calculate $\nabla_\theta L(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta})$. We will start from output neurons and proceed very similarly on a graph with reversed edges[2]. Instead of calculating

[2]The obtained graph has to be acyclic as well: if there were a cycle, we could reverse the edges again to obtain the original graph and it would then also contain a cycle - but we knew it was an acyclic graph.

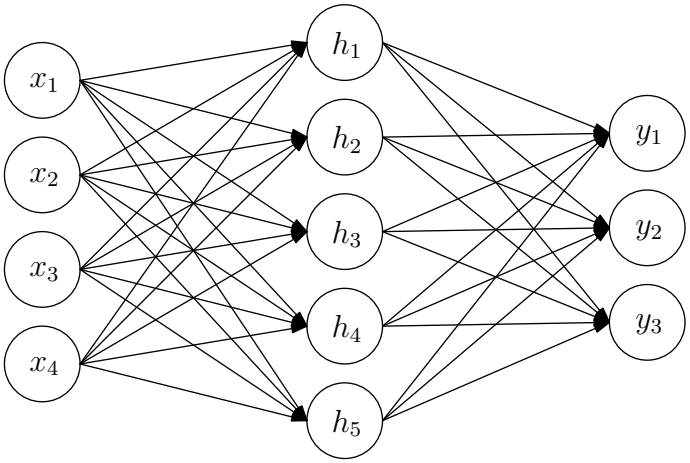


Figure 2.2: Simple neural network with 3 layers. All layers are fully connected to the next one, which means that we can use matrix-vector multiplication to obtain $\boldsymbol{h} = (h_1, \ldots, h_5)^{\mathrm{T}}$ from $\boldsymbol{x} = (x_1, \ldots, x_4)^{\mathrm{T}}$ as $\boldsymbol{h} = f(W\boldsymbol{x} + \boldsymbol{b})$. Here f is applied element-wise, $W \in \mathbb{R}^{5\times4}$ is a weight matrix and $\boldsymbol{b} \in \mathbb{R}^5$ is a vector of bias weights. The result of $W\boldsymbol{x}$ is a column vector where the i-th value corresponds to the value of dot product between the i-th row of W and $\boldsymbol{x}$. The matrix-vector product lets us to compute value of multiple neurons at once - see Figure 2.1 for comparison.

value of neuron, we will use these two equations to calculate partial derivatives with respect to the neuron's function and then with respect to the parameters that the neuron uses to compute its output.

2.2.2 Fully Connected Layers

There are several common building blocks or patterns that modern neural network architectures use. A group of neurons that together represent some logical block in the network is often called a *layer*. For example, all input neurons can be referred to as *input layer*, similarly with all output neurons.

A common transition from one layer A to another layer B is to make connections from all neurons in layer A to all neurons in layer B. We then say that layer A is fully connected to B, or that B forms a fully connected layer of x neurons. An example of neural network with three layers is shown in Figure 2.2.

Fully connected layers that use the standard weighted sum have one advantage: all weights can be represented in a matrix and all values in layer B can be computed by a simple matrix-vector product (of the weight matrix and vector that contains all values in layer A). In computational graphs such fully connected transitions are represented by a connection between two single nodes - each node calculates a vector of output values. Using a GPU such operations can be easily made in parallel and this then allows for significant speed ups.

A significant disadvantage is the number of required parameters (weights): fully connecting layer A of a neurons and B of b neurons requires at least ab weights (commonly b bias parameters are added on top of that).

2.2.3 Convolutional Neural Networks

Motivation

Let us assume that the input of our network is a 2D image. If we represented each pixel as a single neuron and connected all input neurons to all neurons in a following layer, the resulting network would have an interesting property: any pixel permutation applied to all input images – even such permutation that humans would no longer be able to classify the modified images – would not affect the network's potential performance in any way. That is simple to show: we can take any network trained on the original images and permute the input neurons in the corresponding way. In other words, the network is not forced to leverage the fact that the input image is composed of multiple regions of neighboring pixels that together capture certain information.

Another problem of these fully connected layers is that the number of used parameters grows quite fast with the image dimensions. For example, for a 32×32 input image (1024 input neurons) and n neurons in the following hidden layer that is fully-connected to the input layer, we need at least $1024n$ parameters for the connections (and perhaps n more for the bias terms). Now let us imagine that such network takes an input image of dimensions 1024×1024. That is not something unrealistic as photos produced by modern cameras often have way more than 1024×1024 pixels. However, in such a case we would quickly need billions of parameters. Related drawback is that the input image size is fixed after training and we have to scale inputs with different dimensions[3].

For these reasons, modern architectures that have an image input commonly use *convolutional layers* instead of fully connected ones, at least at the beginning of the network.

Convolution

Convolution is an operation that has many applications, we will focus on its usage in digital image processing. For input matrices K (called *kernel*) and I (input image) let us define output of convolution $K * I$ as matrix O where:

$$O_{rc} = \sum_{i=0}^{K_r-1} \sum_{j=0}^{K_c-1} I_{r-i,c-j} K_{ij}$$

where K_r and K_c are the number of rows and columns of the kernel matrix. Informally, every pixel of the output image is a weighted sum of pixels in a neighboring rectangular region in the input image, where kernel matrix defines weights in that weighted sum. Note that certain region on the border is not well-defined as we run out of the input image - usually the input image is padded with zeros, or the output image is slightly smaller than the input image.

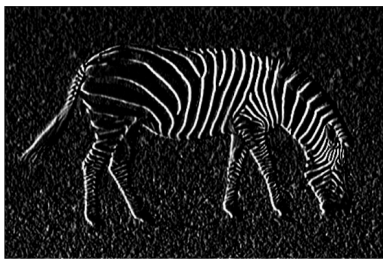
There are many known convolution kernels that apply some interesting transformation to the input image, e.g. highlighting edges (*edge detection*), sharpening or blurring. See Figure 2.3 for an example of one such kernel application.

[3]Some convolutional neural networks also require that all inputs have certain fixed dimensions, but using global average pooling it is possible to avoid that. More details about global average pooling are in the following sections.

(a)

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

(b)



(c)

Figure 2.3: Input photo (a), convolution kernel (b) and result of convolution (c) = (b) $*$ (a). Photo of the zebra is taken from Set14 dataset by Huang et al. [15].

Convolutional Layers

LeCun et al. [20] proposed to use backpropagation to directly learn weights in multiple convolution kernels. A single *convolutional layer* is then composed of multiple such kernels that are all applied to an image from the previous layer.

More precisely, because input images are typically composed of three *channels* (red, green and blue) and therefore are 3-dimensional, every convolutional layer has multiple output channels (not necessarily three). Then, for a convolutional layer with i input channels and o output channels, there are in total $i \times o$ convolution kernels being applied: every output channel is computed as a sum of i different convolutions. More formally, the t-th output channel of output image O is computed as a sum of i convolutions using kernels $K_{t,1}, K_{t,2}, \ldots, K_{t,i}$ by:

$$O_{r,c,t} = \sum_{p=0}^{i} K_{t,p} * I_{r,c,p}$$

The number of required parameters in a single layer is therefore $o \cdot i \cdot k_w \cdot k_h$ where k_w and k_h are the kernel's height and width (we assume these are the same for all performed convolutions). So to produce one output channel using 3×3 kernel for an input image with 3 channels we need only 27 parameters. However as the number of parameters grows linearly with both the number of input and output channels, let us consider a situation when the previous layer produces 512 output channels and the current layer is supposed to produce 512 output channels. Assuming we use 3×3 kernels, we now need 2,359,296 parameters in total for that single layer-to-layer transition. On the other hand, notice that the number of parameters now does not depend on the input image dimension at all: regardless of whether the input has dimensions 4096×4096 or 32×32, the

number of parameters stays the same, only the processing time will differ as we will need to apply the kernels at more places.

To go back from images to neurons, convolutional layers arrange them in a cuboid. The input neurons are arranged in a $h \times w \times 3$ cuboid (assuming RGB image), therefore the number of neurons in the network is now dynamic and depends on the size of the input. Value of every neuron is computed exactly as described with images, we just reuse the same parameter value in multiple connections. To do backpropagation we can just accumulate gradients from all the corresponding usages. After calculating the value of every neuron in the $h \times w \times o$ cuboid using convolutions, commonly a bias term is added (the same one for all neurons that are together in a single output channel) and then a non-linearity is applied. Common choice for the non-linearity is ReLU function.

Pooling Layer

To reduce the *spatial dimensions* – the height and width of the intermediate images/neuron cuboids – one common approach is to employ *pooling layers*. The number of channels in such operations stays exactly the same and these layers do not typically use any parameters. Instead, they consider at a region of certain size (like 2×2) and produce a single number out of that region. In *max-pooling* such number is produced by taking a maximum over the region, or in *avg-pooling* the arithmetic mean is taken. These operations also use a *stride* greater than 1 to reduce overlapping of the regions: a stride of s means that only every s-th row and column are considered as top left corners of some region. To achieve similar effects as pooling (reducing spatial dimension), convolution with stride greater than 1 can be used as well, yet it requires extra parameters and computations.

Using pooling layers, we can design the entire network to avoid fully connected layers altogether. Assuming we need to do classification over c classes, we can just design the last convolution to produce c channels and then add a final pooling operation that reduces the dimensions to $1 \times 1 \times c$. This can be done statically, i.e. by a pooling layer with a fixed region size in the pooling layer that is calculated based on the image size, but modern architectures implement this dynamically: at the end they contain a *global average pooling layer*, that computes the average value of every channel, thus ending with c total numbers regardless of the input dimensions. Then a standard softmax can be applied to produce a probability distribution over the c classes.

2.3 Recurrent Neural Networks

2.3.1 Principle

Recurrent neural networks (RNN) are neural networks that are tailored to process sequential data. What feed forward networks do not allow are loops: connections of some outputs back to some other neuron. When processing sequences it is however desirable for the network to keep some kind of information about what it has already seen in the past and what were its previous outputs.

Consider that we want to process some sequence of inputs $\boldsymbol{x}^{(0)}, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n)}$ and expect some seqeuence $\boldsymbol{y}^{(0)}, \boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(n)}$ as an output. In RNNs the networks

are allowed to contain cycles, but the connection does not go into exactly the same neuron: rather, for each of the n *time steps* a new copy of the network is created. The cycle passing output of the network back to some neuron s then just means that every output $\boldsymbol{y}^{(i-1)}$ is connected into $s^{(i)}$: neuron s present in i-th version of the network. The key is that all the copies of the neural network share the same parameters at the corresponding places. The backpropagation algorithm can be adjusted to handle such situations and it is then called *backpropagation through time*.

2.3.2 Long Short-Term Memory Cells

There is one particular problem with RNNs unless we use some more sophisticated architecture: for long time series, the training suffers from a vanishing or exploding gradient problem. Long Short-Term Memory (LSTM, Hochreiter and Schmidhuber [13]) cells were designed to avoid such problem by using *gates* and a gated self-loop.

LSTM cells are parameterized by dimension d and produce at each time step two *state* vectors: $\boldsymbol{h}$ (hidden state) and c (memory cell state), both from $\mathbb{R}^d$. Input of the cell at time step t is a vector $\boldsymbol{x}^{(t)} \in \mathbb{R}^d$ and the two state vectors from previous time step: $\boldsymbol{h}^{(t-1)}$ and $\boldsymbol{c}^{(t-1)}$. In the very first time step they can be initialized to zero.

In an extension by Gers et al. [6] there are three types of gates: an *input gate*, *forget gate* and *output gate*. To compute the two outputs $\boldsymbol{h}^{(t)}$ and $\boldsymbol{c}^{(t)}$, we do the following:

- The three gates are prepared; all of them apply their own linear transformations of x and another transformation of h, add them together with a bias term and apply element-wise sigmoid function. For instance, for the input gate:
$$\boldsymbol{i}^{(t)} = \sigma\left(W\boldsymbol{x}^{(t)} + V\boldsymbol{h}^{(t-1)} + \boldsymbol{b}\right)$$
The $\boldsymbol{f}^{(t)}$ and $\boldsymbol{o}^{(t)}$ – forget and output gates – are determined in analogous way, they just use their own parameters for the transformations and bias.

- The memory cell is updated by first element-wise multiplication cell with the forget gate: this allows the memory cell to intentionally forget some pieces of information. Then, input information to the memory cell is prepared in a similar way to the gates, only tanh is used instead of the sigmoid activation. This input information is then element-wise multiplicated (gated) by the input gate $\boldsymbol{i}^{(t)}$ and simply added to the memory cell:
$$\boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} \cdot \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \cdot \tanh\left(W'\boldsymbol{x}^{(t)} + V'\boldsymbol{h}^{(t-1)} + \boldsymbol{b}'\right)$$
The important thing is that there is no non-linearity applied to $\boldsymbol{c}$, which helps with the vanishing/exploding gradient problem.

- Finally, the new hidden state is computed by and applying non-linearity to the new memory cell state and gating the result by the output gate:
$$\boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \cdot \tanh\left(\boldsymbol{c}^{(t)}\right)$$

2.4 Reinforcement Learning

In this section we will briefly introduce few concepts of *reinforcement learning* (RL) that are relevant to our thesis. More details can be found in a book by Sutton and Barto [30]. Generally, in RL problems, there is an *agent* that interacts with a certain *environment*. The environment is in a certain *state* and based on the state the agent has to choose an *action* and the environment in return changes its state and gives the agent some *reward* for the action. This is repeated for T time steps or even indefinitely; in the finite case the T steps form an *episode* that ends in some terminal state. This interaction is visualized in Figure 2.4.

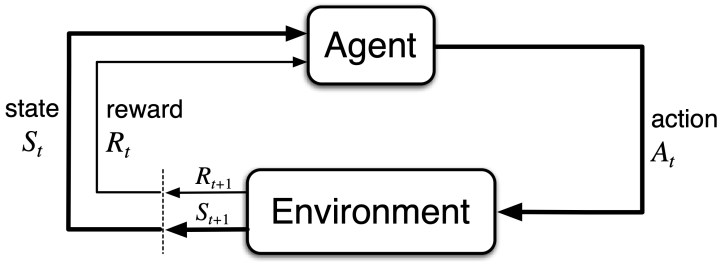


Figure 2.4: Interactions in Reinforcement Learning. Figure adopted from Sutton and Barto [30].

The agent aims to maximize his *return*: sum of obtained rewards. Because the agent needs to decide on an action in each state, his strategy at time t defines a *policy* $\pi^{(t)}$ for which $\pi^{(t)}(a|s)$ corresponds to the probability that the agent selects action a when the environment is currently in state s. Based on the past rewards the agent can *learn* and adjust his policy: this process is called reinforcement learning. Ideally, the agent would eventually learn an *optimal policy*, a policy that maximizes his expected return. In order to do that, the agent has to balance between exploitation and exploration: between choosing the action that currently seems to be the most promising, and some other action in order to learn more about the result for that action.

2.4.1 REINFORCE Algorithm

One commonly used algorithm is called REINFORCE (Williams [36]). It is a *policy gradient method* that uses some parameterized model (e.g. a neural network) to produce a policy (a distribution over actions) and based on the observed reward(s) updates the parameters of the model using gradient ascent to hopefully improve the policy. For given parameters $\boldsymbol{\theta}$ we denote $\pi(a|s, \boldsymbol{\theta})$ as the policy produced by such model. Also, by J(θ) we denote performance of that policy: the expected reward obtained by following the policy π. Then, by *policy gradient theorem* the following holds:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s, \boldsymbol{\theta})$$

where $\propto$ means proportional to, μ is distribution over states obtained by following π and $q_\pi(s,a)$ is *q-value of state action pair* s, a: expected return if we started within the environment in state s, chose action a and then followed policy π.

REINFORCE directly applies the policy gradient theorem by first rewriting:

$$\nabla J(\boldsymbol{\theta}) \propto \mathbb{E}_{s\sim\mu}\left[\sum_a q_\pi(s,a)\nabla\pi(a|s,\boldsymbol{\theta})\right]$$

if we follow the policy π, we can just treat each encountered state as a sample from μ. Similarly, we can use the currently taken action as a sample from distribution over actions and use that to estimate expected value over actions:

$$\nabla J(\boldsymbol{\theta}) \propto \mathbb{E}_{s\sim\mu}\left[\sum_a \frac{\pi(a|s,\boldsymbol{\theta})}{\pi(a|s,\boldsymbol{\theta})} q_\pi(s,a)\nabla\pi(a|s,\boldsymbol{\theta})\right] =$$

$$= \mathbb{E}_{s\sim\mu}\mathbb{E}_{a\sim\pi}\left[q_\pi(s,a)\frac{\nabla\pi(a|s,\boldsymbol{\theta})}{\pi(a|s,\boldsymbol{\theta})}\right] = \mathbb{E}_{s\sim\mu}\mathbb{E}_{a\sim\pi}\left[q_\pi(s,a)\ln\nabla\pi(a|s,\boldsymbol{\theta})\right]$$

Finally, $q_\pi(s,a)$ can be estimated by the partial return that was obtained between reaching s and the end of the episode. Therefore, the algorithm first samples an episode by interaction with the environment according to the policy π; then it iterates backwards over states in the episode from the terminal state to the initial state and at each step adds current reward to the partial return G: an estimate of $q_\pi(s,a)$. After we have updated the estimate G for $q_\pi(s,a)$, a gradient ascent update of $\boldsymbol{\theta}$ is performed using gradient of the policy parameters given that the policy produced action a for the currently considered state s:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha G \nabla \ln \pi(a|s,\boldsymbol{\theta})$$

where α is learning rate, G current estimate of $q_\pi(s,a)$, s is the currently considered state and a is the taken action. In our applications, we will obtain only a single reward at the end, then G is just equal to this reward for all steps.

Due to all the sampling involved, REINFORCE has a high variance; to reduce that variance and speed up the training process commonly a *baseline* is subtracted from $q_\pi(s,a)$. The baseline can be arbitrary as long as it does not depend on a:

$$\sum_a b(s)\nabla\pi(a|s,\boldsymbol{\theta}) = b(s)\nabla\sum_a \pi(a|s,\boldsymbol{\theta}) = b(s)\nabla 1 = 0$$

Therefore subtracting $b(s)$ from $q(s,a)$ does not change direction of the gradient, but variance of the estimations is reduced and therefore the algorithm converges faster[4]; how fast depends on used $b(s)$. Generally, $b(s)$ can be approximated by another parameterized model. When neural networks are used to approximate both $\pi(a|s)$ and $b(s)$, some part of the two networks may be shared.

2.5 Evolutionary Algorithms

Evolutionary algorithms are optimization algorithms that use ideas and terminology inspired by evolution of species in nature. They keep a *population* (a set)

[4]Consider for example situation where we are in state S and can take one of two available actions, A and B. Taking A in S generates an expected return of 0.5 while B in S has an expected return of 50. Without any baseline, selecting A sends a positive signal and the update raises probability of selecting A next time. If we however use a baseline $b(S) = 40$, suddenly taking action A in S has an adjusted expected return of -39.5 and the update would significantly decrease probability of taking A in S next time.

of *individuals* (one possible solution) that are composed of multiple *genes* (a gene is some part of the solution). The quality of an individual is defined by a *fitness function* that is specific for the given problem. Evolutionary algorithms typically work by iterating the same logic over and over, each time updating the current population, until a solution of desired quality is obtained or we hit a limit on the number of iterations. State of the population in an iteration is also called a *generation*; however, one individual may be present in multiple generations (in contrast to individuals in real world).

Therefore, evolutionary algorithms have to define how to obtain the initial population and how to transition from the current population to the new one. This typically involves the following steps:

- Selection: some number of individuals from the current population are selected to be *parents* and produce an offspring. This process is often biased to prefer individuals with higher fitness.
- Crossover: in some evolutionary algorithms a crossover process is performed and the selected parents exchange some part(s) of their genes.
- Mutation: some genes of offspring are (randomly) altered.
- Evaluation: the new offspring are assigned fitness according to the fitness function.
- Environmental selection: some individuals from the current population and from the produced offspring are selected to become the new population. Again, individuals with high fitness are favored.

During the selection and environmental selection the algorithms have to balance between selecting the best individuals (elitism) and not selecting them often enough (leading to worse overall results). The desired ideal is to achieve a certain level of *diversity* within the population, so that we have both very strong and also average/weaker individuals in the population. This is desired because too much focus on the best individuals may make the population converge to a local optimum; the currently suboptimal individuals may still carry some good subsequence of genes that may eventually improve other individuals in the population.

3. Related Work

In this chapter, we will go through some of the existing and relevant work and we will also briefly mention few closely related problems or approaches that are out of the scope of this thesis.

3.1 Architecture Search

A more general problem than the one that we are concerned with in this thesis is a general architecture search. Architecture of the network describes how to interconnect neurons, what activation functions are used and where, number of neurons at each layer and so on. The architecture should also define where to put weights – but note that while the actual values of these weights are not a part of the architecture, the trainability of such weights often guides the architecture search. In other words, quality of the architecture is commonly measured by training weights and measuring performance.

As the general space of architectures is enormous, a first common step in architecture search approaches limits the search space (or design space) over which the actual search is performed. Also, another typical pattern is to design some larger blocks that are then chained together as opposed to adding a single neuron to the currently considered architecture. For example, the architecture might be constrained to contain a few convolution layers at the beginning and then in the search process one option might be to insert a new convolution layer that uses 3x3 filters.

We can try to divide approaches to architecture search into two buckets - manual and automatic. However, there is a lot of potential ambiguity: for example, if we fix a certain composition of convolutional and pooling layers, and then run some algorithm that searches for the number of kernel sizes, filters and strides, such approach could be considered both automatic and manual.

In this thesis, we will describe certain fixed small blocks that were found useful in the manual section (even though they might have been found by some algorithm) and in the automatic architecture search we will describe approaches that search for composition of such blocks together.

3.1.1 Manual Architecture Design

As we mentioned earlier, in this section we will show few hand-crafted modules, blocks or techniques that can be commonly found in modern convolutional neural networks. Our focus will be on ideas that are designed to reduce the number of parameters in the network.

Residual Connections

He et al. [10] experimented with very deep convolutional networks and found a construction that helps with training of such networks. They propose to add *residual* or *shortcut* connections that compute channel-wise addition of all input channels of one convolutional layer to outputs of another convolutional layer

present later in the network (and use that in all places where the previous output channels were used). Figure 3.1 shows two types of blocks: a residual and a residual bottleneck block.

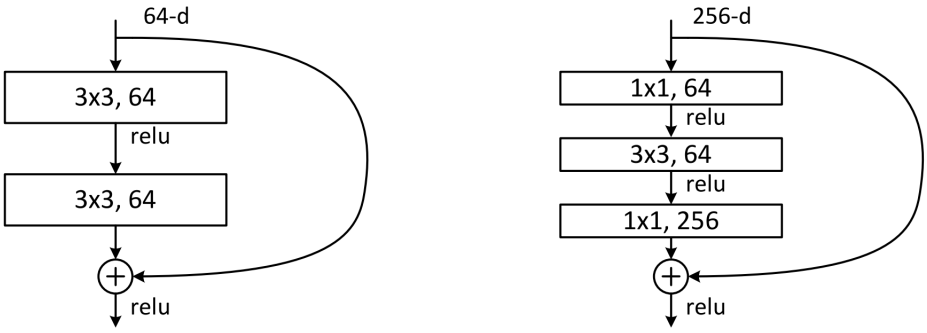


Figure 3.1: Left: a residual block, the residual connection connects the input of the block to the output using addition and ReLU. Right: a residual bottleneck block, that uses 1x1 (point-wise) convolutions to first reduce number of channels and then expand later, in an attempt to keep number of parameters at a reasonable level. These are the main blocks of the ResNet architectures. Figure adopted from He et al. [10].

The residual block consists of two convolutional layers that ends with a residual connection of the input, while the residual bottleneck block consists of three layers: two 1x1 convolutions that reduce/expand channel dimensions and one 3x3 convolution on small number of channels. The residual bottleneck blocks were introduced to keep the number of parameters at a reasonable level.

In the same paper, He et al. [10] proposed multiple ResNet architectures based on the number of layers – in all of them multiple copies of such blocks (with different dimensions) are chained together to form the network. Large ResNets won ILSVRC 2015, and for us especially the bottleneck blocks show one particular way how number of parameters can be reduced in the architecture.

Fire modules and SqueezeNet

Similarly to our thesis, Iandola et al. [16] attempted to construct an architecture that requires a small number of parameters yet achieves a competitive accuracy. They do not attempt to reduce a given network, but rather construct from scratch a new architecture called SqueezeNet that is based upon the following three design strategies:

- Replacing 3x3 filters with 1x1 filters: 3x3 filters are very prevalent in standard CNN architectures. Every time we use a 1x1 filter instead of a 3x3 filter we reduce the number of parameters in that part of the network 9 times.

- Decreasing the number of input channels in layers that use larger kernel sizes. This is again directed at reducing the total number of parameters: consider one convolution layer with 512 input channels that are fed into next convolution layer that also has 512 input channels. The total number of required parameters for the first convolution is then $K_w \times K_h \times 512 \times 512$,

where K_w and K_h are width and height of the filter. Halving number of input channels in both layers would then decrease the number of parameters required for the transition by 4 times.

- Downsampling late in the network: they use the assumption that if the network quickly decreases input image size to a small one, the accuracy will be severely impacted. To avoid this, they propose to keep large activation maps and defer the downsampling to the latest layers in the network.

Applying these principles, they proposed a new building block called *Fire module*. It consists of two convolutional layers, a *squeeze layer* that only has 1x1 filters, followed by an *expand layer* that has some number of 1x1 filters and some number of 3x3 filters. A specific Fire module block can therefore be described using three numbers that correspond to these three number of filters: they label them as s_{1x1} for number of 1x1 filters in the squeeze layer, and e_{1x1}, e_{3x3} for 1x1 and 3x3 filters in the expand layer respectively. From the second design strategy they also infer a constraint $s_{1x1} < e_{1x1} + e_{3x3}$ to limit the number of input channels to the 3x3 filters.

The entire hand-crafted SqueezeNet architecture that they proposed is then composed of a single standard convolution at the beginning, 8 Fire modules and a final standard convolution. They obtain a model that is 50x smaller than AlexNet and meets or exceeds AlexNet accuracy on ImageNet. Adding residual connections yielded another small improvement in accuracy.

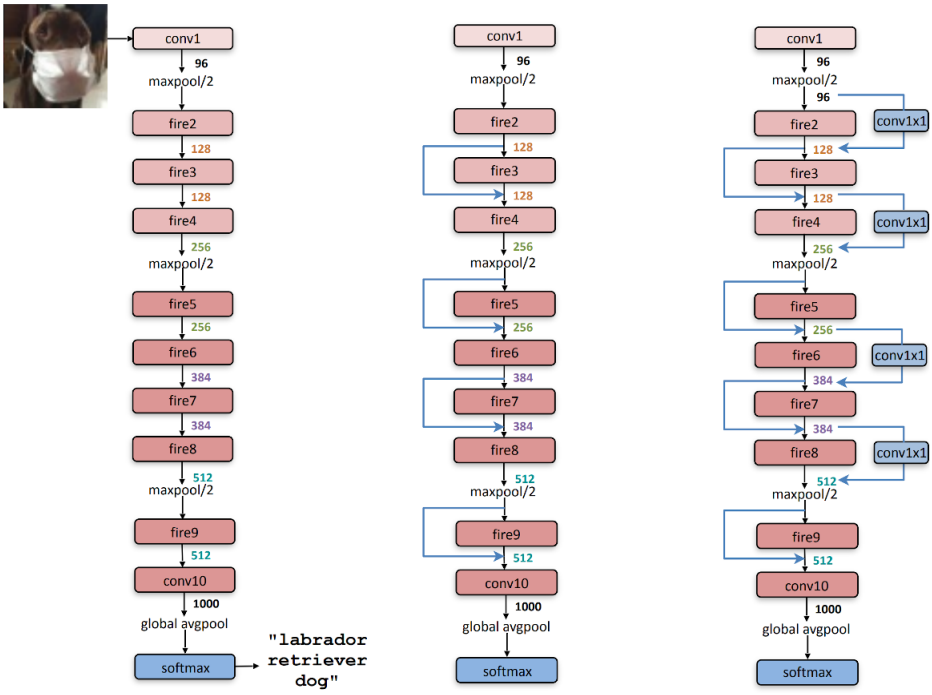


Figure 3.2: Visualization of SqueezeNet architecture. Left: plain SqueezeNet, middle: SqueezeNet with residual connections, right: Squeezenet with residual connections and 1x1 residual convolutions. Middle architecture achieved highest accuracy. Figure adopted from Iandola et al. [16].

Depth-wise Convolution and Depth-wise Separable Convolution

Standard convolution requires a lot of parameters especially when the number of input and output channels is high as it is proportional in both of them. To combat this, Howard et al. [14] in their MobileNet paper replace most convolution layers with *depth-wise separable convolutions*. Depth-wise separable convolutions are a combination of two parts: a *depth-wise convolution* and a standard 1x1 convolution. Depth-wise convolution does not combine produced channels at all – it simply contains $k_w \times k_h$ parameters for each input channel, applies the corresponding convolution on that single channel and therefore produces a single output channel for every input channel. Because each output channel depends only on a single input channel, depth-wise separable convolutions use a standard 1x1 convolution (also called *point-wise convolution*) afterwards. For every output channel, this step then combines all output channels of the depth-wise convolution multiplied by a corresponding weight (for a total number of input $\times$ output parameters).

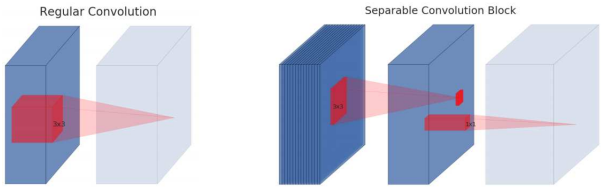


Figure 3.3: Regular convolution vs Depth-wise separable convolution. Figure adapted from Howard et al. [14].

Consider for example a simple 3x3 convolution layer with 128 input channels and 256 output channels. It needs a total of $3 \times 3 \times 128 \times 256$ parameters. Replacing it by a 3x3 depth-wise separable convolution with the same number of input and output channels allows us to reduce the number of parameters to:

$$\underbrace{3 \times 3 \times 128}_{\substack{\#\text{params for} \\ \text{depth-wise convolution}}} + \underbrace{128 \times 256}_{\substack{\#\text{params for 1x1} \\ \text{(point-wise) convolution}}}$$

In this case swapping convolution for depth-wise separable convolution results in roughly 8x reduction of the number of parameters, yet empirically they work similarly well[1].

Network Width and Input Resolution Scaling

Another two techniques that Howard et al. [14] considered were scaling *width of the network* – number of channels in each convolutional layer – by a hyperparameter $\alpha \in (0; 1]$ and similarly, scaling the input image dimensions by a hyperparameter $\rho \in (0; 1]$. While scaling input resolution reduces the number of required floating-point operations (the convolutions are applied to smaller images), it does not reduce the number of parameters in the network. Also, in this thesis we will mostly do experiments on the CIFAR-10 dataset, which contains 32x32 images -

[1] MobileNet with depth-wise separable convolutions had 1.1% worse top-1 accuracy on ImageNet than the same architecture with plain convolutions.

so there is probably no room for this sort of optimization. Therefore, we will not consider resolution scaling for our purposes.

However, width scaling seems like a very simple and effective method to quickly scale the architecture. Scaling the number of channels in all layers by a factor of α scales the number of parameters in both convolutional and depth-wise separable convolutional layers by a factor of α^2, as the number of parameters depends multiplicatively on number of both input and output channels, and both of them are scaled.

3.1.2 Automatic Architecture Search

Recently there were significant advancements in neural architecture search - multiple models close to state of the art for image classification on ImageNet are currently models that are based on EfficientNets - a family of networks that contains multiple scaled version of a base model which was obtained by a neural architecture search. EfficientNets were proposed by Tan and Le [31] and they look for the base model using reinforcement learning similarly to their previous work on MnasNet (Tan et al. [33]).

In MnasNet they first let a recurrent neural network (*controller*) produce a sequence of distributions over various architectural choices and then they sample out of them to obtain an architecture. They train it and measure both its accuracy on a validation set and latency on a real mobile phone. Using these two metrics, they form a reward for the sampled model and improve the sampling controller by Proximal Policy Optimization algorithm. For EfficientNet this procedure is slightly modified: they use number of floating point operations instead of latency on a real device.

We will not go into further details here for two reasons: in Chapter 4 we will describe our modification of a similar neural architecture search procedure adapted to our task; and because we believe that most of the details are more related to the problem of neural architecture search and less to our goal: they need to design the controller so that it can produce a complicated graph with many possible options for each considered operation.

There are also other approaches based on evolutionary algorithms (Elsken et al. [4]) that search for a neural architecture. State of the art approaches are able to find networks with 98.4% accuracy on CIFAR-10 test set with 7M parameters (Lu et al. [25]).

3.2 Neural Network Pruning

One common and natural approach to our problem is called *network pruning*. To prune a neural network is to take its architecture and permanently discard some parts of it: connections between neurons, entire neurons along with all connections, or even entire filters in the convolutional layers. The obtained network after pruning can be *fine-tuned* (doing few extra training steps with very small learning rate). Also, the entire pruning process may be iterated multiple times (*iterated pruning*) - the already pruned and fine-tuned network is pruned again, then fine-tuned and so on and on, until we hit some stop condition (e.g. desired number of parameters was reached).

We can also think of pruning as element-wise multiplication of parameter vector by some binary vector (mask) $\boldsymbol{m}$: instead of $\boldsymbol{\theta}$ we would then use $\boldsymbol{m} \odot \boldsymbol{\theta}$ (element-wise multiplication). Ones would correspond to weights that were kept and zeros to weights that are pruned. This allows for a simple representation of currently pruned weights, but does not reduce the actual number of computations that need to be performed.

In the deep learning era Han et al. [8] demonstrate that in experiments on ImageNet dataset that roughly 90% of connections can be pruned in AlexNet and VGG-19 networks without significant effect on accuracy, although these are architectures with large fully connected layers that are easier to prune. However, many pruning techniques were already described by Cun et al. [2] in 1990.

We can either prune connections without any structural relationships (*unstructured pruning*), or remove multiple connections that together form a larger unit in the network (*structured pruning*). It might be more challenging to remove entire structures like neurons or convolution filters without a degradation in accuracy, but it allows for more efficient calculations when computing output of the network: the matrices that are multiplied when going from layer to layer have smaller dimensions, similarly the number of required convolution operations would decrease. In contrast, pruning a single connection just means that one of the numbers in a corresponding matrix becomes zero.

Han et al. [8] showed that fine-tuning at the end helps a lot: they were able to prune 90% of weigths at almost no hit to accuracy when using fine tuning compared to 80% parameters pruned at 1% hit to accuracy without fine tuning. Therefore, even without fine tuning a significant number of weights can be pruned away. One advantage of not training further is that almost zero computational resources are needed to obtain a much smaller network that still performs reasonably well in practice (assuming that the pruning algorithm is not computationally demanding). Having a good pruning algorithm can help us transform a huge network trained on gigabytes of data to a much smaller network that should still retain most of the knowledge learned on the original data set. Because the new network will be much smaller, one could more easily fine tune it on another data set (*transfer learning*), or just directly use it on slower devices with less available storage (like mobile phones).

3.2.1 Magnitude Pruning

Probably one of the simplest yet very effective approaches is to prune weights that are closest to zero in the network. This can be done locally (in a specific layer), or globally across the entire network (e.g. discard 90% of weights that are closest to zero). The reasoning behind this is quite natural - these weights are already close to zero, so actually making them zero should not have too negative impact. However, even low value weights can be important in the network as noted by Hassibi and Stork [9] and this method might often discard them anyway.

Despite that, some of the recent experiments with magnitude pruning by Han et al. [8] show that magnitude pruning still performs well in practice for deep neural networks. They prune weights locally (with varying rates across layers), added L1/L2 regularization into the initial training to push weights towards zero, they prune different layers by different ratios. According to them L1 regulariza-

tion works better than L2 regularization if no fine-tuning is performed, but with fine-tuning L2 regularization performs better than L1. Also, they show that using both fine-tuning and iterative pruning allows them to prune over 90% of weights at almost no degradation in accuracy in their experiments on ImageNet dataset using AlexNet and VGG-16 networks.

3.2.2 The Lottery Ticket Hypothesis

Recently, Frankle and Carbin [5] proposed The Lottery Ticket Hypothesis:

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.*

They support this hypothesis by giving an algorithm to find such a subnetwork whose weights "won the initialization lottery" by iteratively pruning a large trained network and then resetting the remaining weights to values **before** training (to their original random initialization). Their experiments show that they can consistently find such subnetwork for shallow architectures, but it does not entirely generalize for deep networks - for them they can achieve good results as well, but they need to manually adjust learning rate for training of the subnetwork. They also note an interesting fact: when they prune VGG-16 by (only) 80%, they are able to retrain the network from scratch to similar performance as the original - but more importantly, when they prune the architecture by more than 98.5%, they find out that the found subnetwork performs much worse when trained from scrach than when they keep the initial "lottery weights".

The given algorithm is just an iterative version of global magnitude pruning.

3.2.3 Pruning Based on Hessian Matrix

Cun et al. [2] noticed that low value weights can still be important in the network and instead proposed a more principled approach to pruning weights. Let us consider the second order Taylor approximation of the loss function around $\boldsymbol{\theta}$ when we shift parameters by some vector $\boldsymbol{d}$:

$$J(\boldsymbol{\theta} + \boldsymbol{d}) \approx J(\boldsymbol{\theta}) + \sum_i \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i} \boldsymbol{d} + \frac{1}{2} \boldsymbol{d}^T H \boldsymbol{d}$$

where $H = \frac{\partial^2 J}{\partial \boldsymbol{\theta}^2}$ is the Hessian matrix of the loss function.

Consider that we want to make some weight zero, i.e. set $\theta_i = 0$. According to that approximation for $\boldsymbol{d} = -\theta_i \boldsymbol{e}_i$, the change in loss will be:

$$J(\boldsymbol{\theta} - \theta_i \boldsymbol{e}_i) - J(\boldsymbol{\theta}) \approx -\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i} \theta_i + \frac{1}{2} H_{ii} \theta_i^2$$

Note that this approximation would be even less precise when multiple weights are pruned: Hassibi and Stork [9] instead proposed to work with the entire Hessian matrix (and its inverse), but at the time (1990s) neural networks had thousands of parameters. Now even networks that contain hundreds of thousands parameters

could be considered small, and the Hessian matrix for larger networks is extremely large.

To avoid computations related to the Hessian matrix, Cun et al. [2] proposed to make two further approximations: we can pretend that it is diagonal, and that we are also pruning the network when it is at a local optimum. Then all the partial first derivatives are zero, and the approximation simplifies to:

$$J(\boldsymbol{\theta} - \theta_i \boldsymbol{e}_i) - J(\boldsymbol{\theta}) \approx \frac{1}{2} H_{ii} \theta_i^2$$

Theis et al. [34] noted that they got worse results in their experiments when they included the first derivative term. Also, they proposed to approximate the Hessian H_{ii} through empirical estimate of Fisher information:

$$H_{ii} \approx \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})} \left[\left(\frac{\partial}{\partial \theta_i} \log P_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x}) \right)^2 \right]$$

where P is distribution produced by our model, and expectation is taken over distribution of our data set. This approximation becomes exact when the model is twice differentiable and the data set distribution is equal to distribution produced by our model. Because we use cross entropy loss in our training setup, we can further estimate from N real data samples:

$$H_{ii} \approx \sum_{j=1}^{N} \left(\frac{\partial}{\partial \theta_i} L(\boldsymbol{x}_j, \boldsymbol{y}_j, \boldsymbol{\theta}) \right)^2$$

$$J(\boldsymbol{\theta} - \theta_i \boldsymbol{e}_i) - J(\boldsymbol{\theta}) \approx \frac{1}{2N} \theta_i^2 \sum_{j=1}^{N} \left(\frac{\partial}{\partial \theta_i} L(\boldsymbol{x}_j, \boldsymbol{y}_j, \boldsymbol{\theta}) \right)^2$$

This is both very simple to compute and more theoretically grounded than simple magnitude pruning. Theis et al. [34] call this method *Fisher pruning* and reference very similar approach by Molchanov et al. [27] which uses absolute gradients instead of squared gradients – but they note that Molchanov et al. [27] did not provide similarly principled motivation.

Also, note that if we assumed $H = cI$ (Hessian is a constant multiple of identity matrix), all these second order methods that assume first derivatives are zero would collapse into magnitude pruning.

3.2.4 Pruning by Evolution Strategies

Junior and Yen [18] propose an approach based on evolution strategies that they call *DeepPruningES*. Their approach is structured and targeted at convolutional neural networks: they only prune entire filters in convolutional layers. They use a simple binary mask to represent an individual: one bit for every filter in the network – if some bit is set to zero, corresponding filter is pruned.

For initialization, they create some number of identical copies of the original model that is supposed to be pruned and then randomly mutate these copies according to a probability p_m that a single bit is flipped.

To evaluate an individual, they construct the network according to the representation, and run fine-tuning for some number of epochs and some learning rate

(these are hyperparameters of their algorithm, and they stay constant). After that, they measure the number of floating-point operations (FLOPs) and training error on some data set (training error is complement of accuracy; data set used for evaluation is subset of entire training data set). The goal is to optimize for both, and balance between the two: obviously we expect networks with very low FLOPs to perform worse, and likewise networks with small training error to require more FLOPs to achieve that error.

Their algorithm runs for certain number of generations. In every generation they select three individuals, called *Boundary Heavy*, *Boundary Light* and *Knee*. Boundary Heavy is the individual with minimum training error in the population, Boundary Light is the individual with minimum number of FLOPs. A Knee is the individual with the smallest Manhattan distance from the origin, where this distance is calculated as:

$$\mathrm{D}(i) = \frac{f_1(P_i) - \min(f_1)}{\max(f_1) - \min(f_1)} + \frac{f_2(P_i) - \min(f_2)}{\max(f_2) - \min(f_2)}$$

where i is index of an individual, P_i is the corresponding individual, f_1 is function measuring number of FLOPs and f_2 is function measuring training error.

To generate offspring, they randomly select one of the three marked solutions as a parent, clone that individual and mutate his genes according to probability p_m. This is done until desired number of offspring are generated. These offspring together with the three initially marked solutions form a new population.

To finish the algorithm, they fine-tune all three marked solutions for a few epochs, this time on the entire training set (number of epochs and learning rate are hyperparameters different from the ones used to evaluate an individual).

The authors had limited computational power available, yet they still report interesting results. Given that it is quite expensive to evaluate an individual, they had to use small hyperparameters: population size of only around 20 and 10 generations. For an example from the results, for ResNet-56 and initial test error of 6.63% (and $1.27 \cdot 10^8$ FLOPs) the produced Boundary Heavy, Boundary Light and Knee solutions reached on average test errors of 8.77%, 13.36% and 9.98% respectively. The average number of FLOPs were $1.01 \cdot 10^8$, $0.244 \cdot 10^8$ and $0.432 \cdot 10^8$. However, the number of parameters did not get reduced a lot - they only prune on average 77% of parameters for the Boundary Light solutions, 59% for the knee solutions and 15% for the Boundary Heavy solutions.

3.3 Knowledge Distillation

A different approach commonly known as *knowledge distillation* (or *teacher-student architecture*) was proposed by Hinton et al. [12] and consists of training *a student network* of possibly very different architecture by an already trained *teacher network* (or multiple teacher networks). Student network's weights are initialized randomly. Generally the only constraint there is that both networks need to make predictions over the same number of classes.

The loss function typically tries to balance between two objectives: mimicking the teacher's network output distribution for a given input and individual loss corresponding to the correct label for that same input (as if we were training the student network in a standard way). Consider that the correct label for an image

is that it contains a car. Then we expect the teacher network to predict that it is more likely that the image contains a bus than that it contains a plant.

To help the student learn these "relative nuances", both the student's and the teacher's network output distributions are smoothed out using a temperature parameter T in the softmax function. Given that the output logit for i-th class is o_i the corresponding probability p_i is now computed as:

$$p_i = \frac{\exp(o_i/T)}{\sum_j \exp(o_j/T)}$$

Note that for $T = 1$ this just corresponds to the standard softmax function. Setting T to higher value produces more smooth distribution over the classes (instead of probabilities 95% for car and 2% for truck, the new probabilities could be 76% for car and 11% for truck). Again, this is used only for training - once the model is trained, we will fix $T = 1$.

Note that the student network could also be trained on completely unlabeled data set; then there would not be a component for the individual loss and the student would be trained using teacher's output distribution only.

If we have labeled data for the student's training set, the authors used a weighted average of two different objective functions. One of them is cross entropy between the student's and teacher's output distributions (with T set to the same value), and the other one is just standard cross entropy against distribution with $p_i = 1$ for the correct label (using $T = 1$ in the student). The authors also note that it is important to scale the gradients produced by the smooth distributions by T^2, as they scale with T differently than the other part of the loss. This is done so that if we changed T in experiments, the relative contributions of the two parts of the objective function would also change.

Another interesting thing that Hinton et al. [12] prove is that training the student to minimize MSE loss against logits of the teacher network is a special case of distillation. This was another known teacher-student training approach.

In the context of Lottery Ticket Hypothesis ("small networks are hard to train from scratch"), knowledge distillation tries to pass some additional knowledge from the large network to help the training as opposed to relying on good initial values for the weights.

3.4 Reduction of Arithmetic Precision

Reducing precision of most computations in the network is probably the most popular method of neural network compression at the time of writing this thesis. This is because quite surprisingly the accuracy of an already trained model does not significantly drop after reducing precision - so for embedded devices or mobiles this allows to reduce necessary storage and computations at little cost in accuracy.

Modern NVIDIA GPUs even contain Tensor Cores that are supposed to accelerate mixed-precision training of neural networks (Markidis et al. [26]). In this mixed-precision training most computations are done in 16-bit floating point arithmetic (half precision), with some accumulations (like in the softmax layers) still stored as 32-bit floating point numbers.

This approach is not limited to half precision, there are successful experiments with training done in 8-bit (Wang et al. [35]) floating point numbers as well, and generally it is possible to do n-bit quantization of the weights.

4. Proposed Approach

In this chapter we will describe the proposed approach to reduction of parameters in convolutional neural networks that is based on some of the related work from Chapter 3. This approach will be evaluated in Chapter 5.

4.1 Overview

We propose a relatively simple scheme to reduce the number of parameters in a given architecture. As modern convolutional architectures are commonly built out of several building blocks that are then stacked on top of each other, we aim to scale the architecture on a block-level.

The scheme consists of three stages that are illustrated in Figure 4.1. In the first stage, we need to define how to perform the scaling: the network should be decomposed into multiple blocks, and for all possible values of scaling coefficient $s \in \{1, \ldots, 10\}$ and a particular block b of the architecture we need to define b_s: block that will replace b in the architecture whenever b should be scaled by coefficient s. We propose to do this stage manually, but we also describe an option how this could be done automatically.

The second stage is responsible for optimization of the scaling coefficients: for each block we need to determine s so that the resulting architecture achieves a good balance between the number of parameters and the resulting accuracy. We propose and in Chapter 5 compare three algorithms responsible for this stage: two evolutionary algorithms and one algorithm based on reinforcement learning.

Finally, in the last stage our goal is to obtain weights for the architecture found in the second stage. To further reduce the number of parameters, we propose to do this in three steps: training the obtained architecture according to the same procedure that we would use for the input architecture, pruning it using global pruning, and then fine-tuning it for a few more epochs.

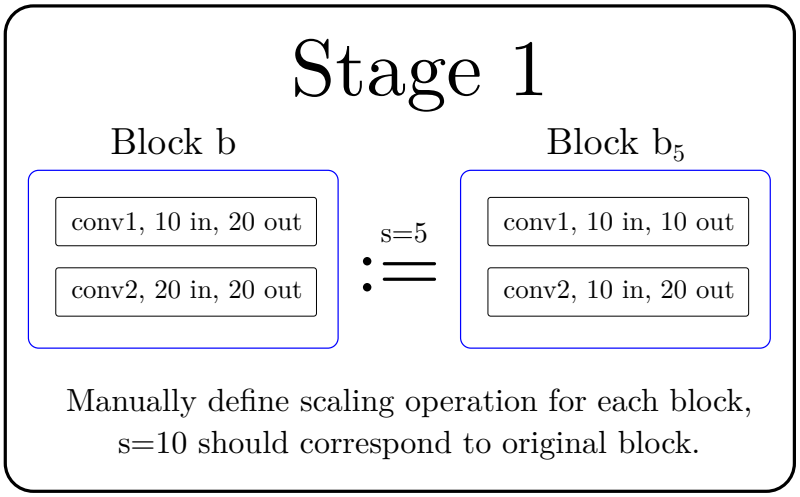
In the following sections, we will describe each stage in more detail.

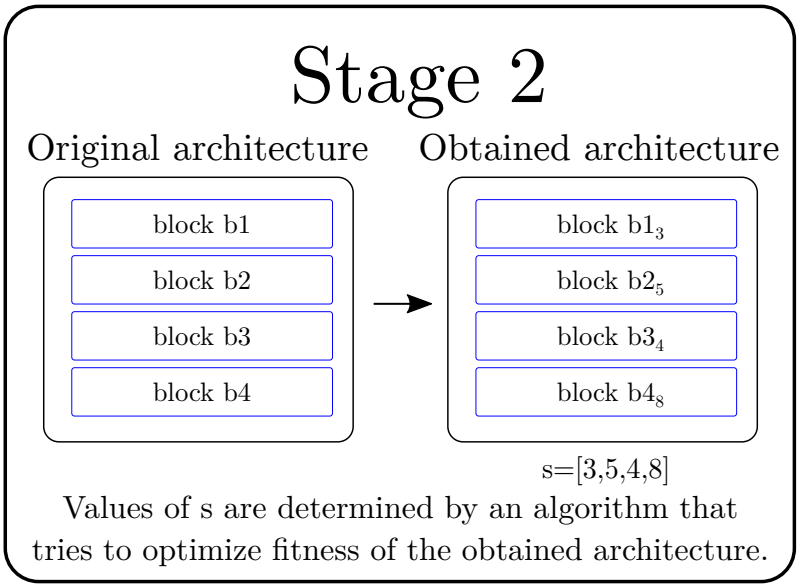
4.2 Parameterizing Blocks of the Architecture

As we mentioned, the first problem we have to solve is how to decompose the network into blocks and how to design the scaling operation. We believe this is best done manually, but at the end of the section we also discuss few automatic options.

4.2.1 Decomposition of the Architecture

To reiterate, we propose to scale the architecture on a block level, therefore we first need to decompose the architecture into multiple blocks. By block we understand some sequence of convolutions and other operations that eventually changes the number of convolutional channels from C_{in} to C_{out} such that the intermediate operations are not connected to some other place outside of the block. Not all





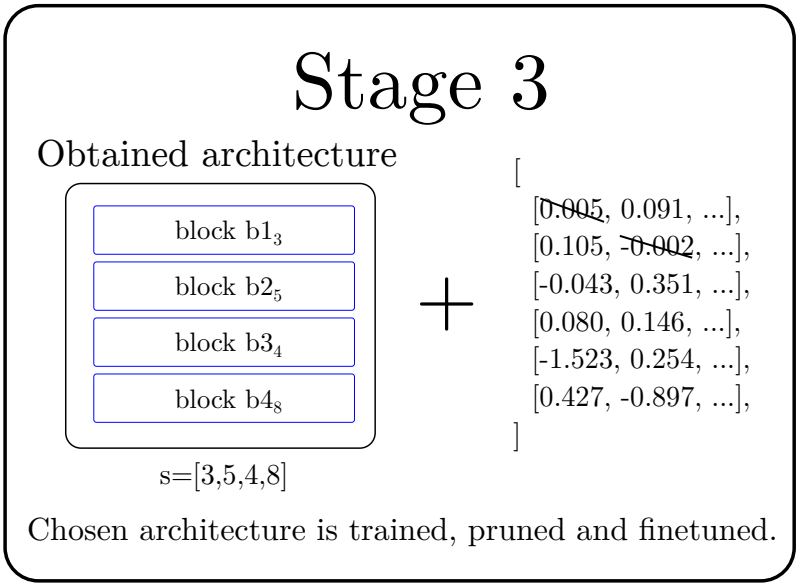


Figure 4.1: Overview of stages of the proposed scheme.

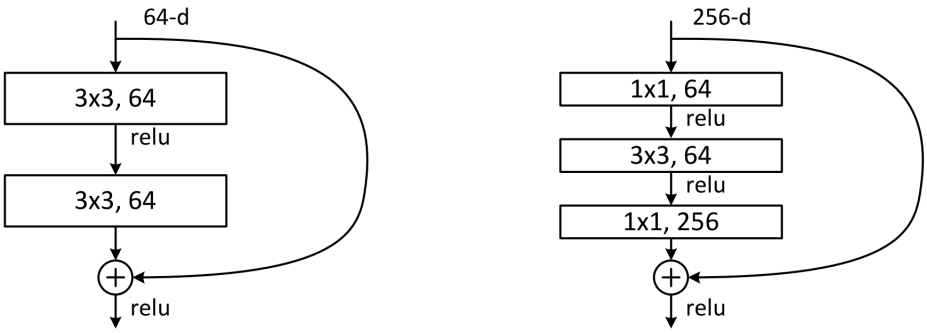


Figure 4.2: Building blocks used in ResNets.

blocks need to be of the same *type*[1], but in most cases they should contain at least 2 convolutions, otherwise it will be hard to scale them as we need to keep C_{in} and C_{out} unchanged; we will discuss this later in more detail.

Note that almost the entire architecture can be understood as a single block by this definition. Therefore, choosing the granularity of "what is a block" can be an important decision. However, given that modern architectures first define multiple small blocks and then chain them together in some way, we will probably not have to make that decision. For instance, in ResNets a reasonable choice seems to be to scale the *building blocks* that they use: a sequence of two convolutions followed by a residual connection - see Figure 4.2.

4.2.2 Scaling the Blocks

Once we decomposed the network into multiple blocks, we propose to define multiple scaled versions of each block. We use 10 as the number of versions. Therefore, for each block b and $s \in \{1, \dots, 10\}$ the scaled version of b by s (b_s) needs to be defined. We do not prescribe how exactly these scaled blocks need to be produced as long as the following conditions are met:

- The original architecture with n_B blocks can be constructed for each possible vector $\boldsymbol{s} \in \{1, \dots, 10\}^{n_B}$. By constructing the architecture we mean substitution of i-th block b by its corresponding scaled version $b_{\boldsymbol{s}_i}$. Therefore, the number of input and output channels to each block shall remain the same for all possible s.
- For each block b and for all $s_1, s_2 \in \{1, \dots, 10\}$, $s_1 < s_2 \Rightarrow \text{ParamCnt}(b_{s_1}) \leq \text{ParamCnt}(b_{s_2})$.
- For each block b, it holds that $b_{10} = b$, i.e. for $s = 10$ the block shall remain unchanged.

Note that we do not restrict potential big architectural changes to the block itself: the scaled versions of b can look very differently to the original b, e.g. they may possibly use depth-wise convolutions instead of normal convolutions or add some point-wise convolutions to change the number of intermediate channels and

[1]By type we mean what operations are performed within the block: e.g. some blocks can contain 5 convolutions while other blocks may just do 3 of them.

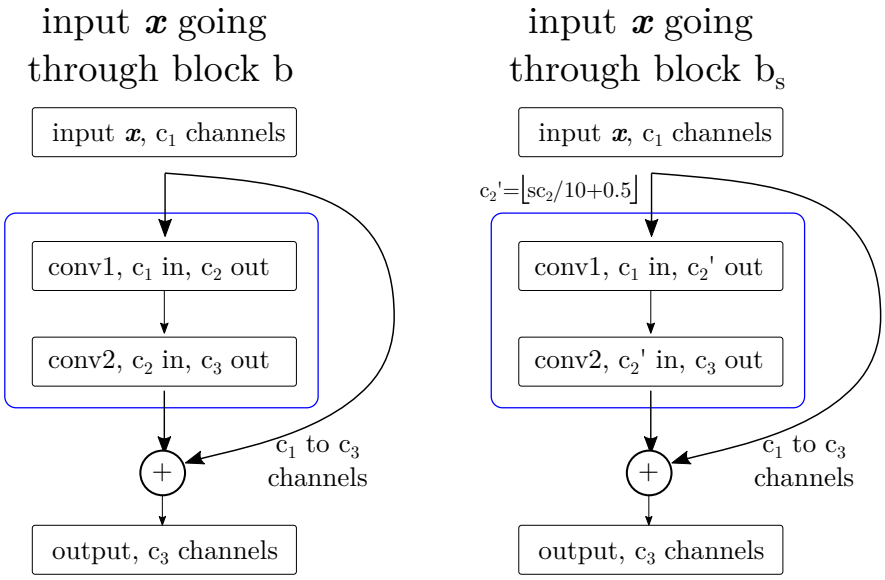


Figure 4.3: Scaling of ResNet blocks. The intermediate channels c_2 are not connected outside of the blocks and therefore there are no restrictions on their number in b_s. We scale c_2 proportionally with s so that the value for $s = 10$ corresponds to c_2 and round to the nearest integer.

so on. Also, the scaled versions themselves do not need to stick to the same structure; for example, block b_3 can use different operations than b_5. That said, in our main experiments we decided to stick as close as possible to the original architecture as our primary goal is to reduce an existing architecture and not to design a new one – but later in this section we show a comparison of few block types.

The first condition is needed to avoid situations as in the following example: consider an architecture that contains two blocks A and B so that the output of A is the input of B. Block A takes 16 and produces 32 channels while block B takes 32 and produces 32 channels. If we substituted A for A_2 that changes the number of output channels to 20, and B for $B_{10} = B$, then B would still expect 32 input channels but only get 20 - there is a mismatch that causes problems. Note that A_2 can however produce additional 12 zero channels. Given that blocks were defined so that intermediate computations are not connected to somewhere outside of the block in the network, this means that there is no longer a potential mismatch if we change the number of channels in some intermediate convolution inside of a block.

For our experiments in Chapter 5 we want to stick to the original architecture as close as possible. Therefore, we scale the ResNet blocks as illustrated in Figure 4.3: for a given s we change the number of intermediate channels from p to $\lfloor \frac{1}{10}ps + 0.5 \rfloor$. This is a very simple way that meets all the required conditions and decreases the number of parameters in the network linearly: if we set $s = 1$ for all blocks, we will get a network that uses approximately one tenth of all parameters that were used in the convolutional layers.

4.2.3 Block Replacement

As we left room for architectural changes, some scaled versions of b might use very different operations compared to b. This leads to a possible automatic block scaling: we can treat each convolution as a block and for $s = 10$ return that convolution unchanged. For $s < 10$, we can replace the convolution by a sequence of convolutions and scale the intermediate convolutions. This unfortunately did degrade both accuracy and epoch training time of the resulting network compared to scaling performed inside of the blocks. Another option is to again treat each convolution as a block, scale its number of outputs channels and then pad it to the original number of outputs; yet again, this performed very poorly in our preliminary experiments.

We also tried substituting the original ResNet block for a different one in ResNet-20 to roughly evaluate the effect a different block can have on performance and training time. We tried the following blocks: 1-3 Block - where block transforming channels from c_i to c_o is replaced by a $1x1$ convolution ($c_i \rightarrow 2c_o$) and then 3x3 convolution ($2c_o \rightarrow c_o$), Wrapped Convs: block where each convolution ($c_i \rightarrow c_o$) is replaced by a two convolutions: 1x1 ($c_i \rightarrow c_o$) and 3x3 ($c_o \rightarrow c_o$).

Block	Param Cnt	Training time	Top-1 Accuracy
Original Block	269,722	8s/epoch	90%
1-3 Block	177,978	10s/epoch	87,9%
MBConv6	206,170	28s/epoch	90.8%
MBConv6 FP16	206,170	60s/epoch	x
Wrapped Convs	256,842	14s/epoch	89.5%

Few importants things to note: a single epoch pass over the CIFAR-10 dataset using ResNet-20 for a single MBConv6 on our GPU roughly 4 times longer, despite the fact that it uses approximately 63,000 less parameters and also needs fewer floating point operations. Some implementations should be able to compute depthwise separable convolutions much faster when using half-precision arithmetic, but on our GPU (GeForce GTX 1070) this takes even longer: about a minute per epoch. On other GPUs we expect the MBConv6 block using half-precision arithmetic to be faster than the original ResNet block. Another important thing is that using MBConv6 blocks resulted in a higher accuracy and lower parameter count, which is in line with MBConv6 usage in modern efficient architectures like EfficientNet or MobileNets.

We decided not to proceed with the automatic approaches: they require extra parameters, may not recognize layers that should not be altered (like the very first convolution), prolong training time as more convolution operations are performed, significantly affect accuracy and also make comparisons to performance of the original architecture less fair as there is a substantial change of the architecture. Also, the analysis by Zagoruyko and Komodakis [38] done for Wide ResNets shows that particular arrangement of convolutions inside of a block can significantly affect quality of the model. Given the differences between various architectures, we believe that some manual decisions are necessary - and in most cases they will be simple enough to justify doing them.

4.3 Optimization Stage

We propose three different algorithms for finding the desired vector $\boldsymbol{s} \in \{1, 10\}^{n_B}$. In this section we will first define few terms and then describe each one of the algorithms: two of them will be based on evolution and one on reinforcement learning.

4.3.1 Common Properties

Individual Representation

We refer to candidates for the optimal vector $\boldsymbol{s}$ as individuals. We represent an individual by the corresponding vector $\boldsymbol{s}$ of scaling coefficients and in evolution with weights also by a set of all weights necessary for construction of the architecture for any vector $\boldsymbol{s}$.

Therefore, if block b in its original version uses p parameters to compute output of some convolution, we always store these p parameters in all individuals, even in the ones that currently use s other than 10 for b and the architecture that corresponds to their $\boldsymbol{s}$ does not use most of the p parameters. This is done so that if we take any two individuals and do crossover, both individuals can provide weights for the entire architecture regardless of how we mix their two vectors $\boldsymbol{s}$.

Fitness Function

We decided to balance between two objectives: accuracy of the resulting network and the number of parameters. In most cases, we will transform these two objectives into a single one, but we also experimented with application of multi-objective algorithm NSGA-II (Deb et al. [3]).

We decided to use a similar formula to the one used in search for the baseline network of EfficientNet (Tan and Le [31]) and EfficientNetV2 (Tan and Le [32]) family. For EfficientNet, the authors turn the multi-objective optimization of accuracy and number of floating point operations into a single objective optimization by defining fitness of network m as:

$$\text{Fitness}(m) = \text{Acc}(m) \times (\text{FLOPS}(m)/T)^w$$

where $\text{Acc}(m)$ and $\text{FLOPS}(m)$ are accuracy and number of floating point operations of model m respectively, T is a parameter that determines the target number of FLOPS and w is a hyperparameter that controls trade-off between accuracy and FLOPS. They set $w = -0.07$ since for Pareto-optimal solutions[2] a 5% relative gain in accuracy can be obtained by doubling the number of allowed FLOPS, and $1/2^{-0.07} \approx 1.05$. Thus, when model A uses half the number of FLOPS as model B, it is compensated for its possibly worse accuracy by a factor of 1.05. In EfficientNetV2 they also multiply the objective function by the number of parameters with a similar balancing exponent $v = -0.05$.

Based on their findings, we define:

$$\text{Fitness}(m) = \text{Acc}(m) \times (\text{Params}(m)/T)^w$$

[2]Architectures where accuracy cannot be improved without increasing FLOPS, or FLOPS cannot be decreased without hitting accuracy.

where T corresponds to a target number of parameters and w is again a hyperparameter that controls the trade-off. Note that setting $w = -0.05$ for all values of Params(m) makes the division by T^w unnecessary - it just normalizes the fitness, but does not change the relative ordering between any two individuals. If the parameter T is considered to be a strict target, we can lower w for networks m where Params(m) $> T$ to penalize this overstep but still provide some additional information for such networks. If we on the other hand just set Fitness(m) $= 0$ for such cases, we would not be able to distinguish between individual that exceeds the bound T by 50 parameters and individual that exceeds the bound by 2 million parameters.

Evaluation of an Individual

To evaluate an individual, we simply take the corresponding vector $\boldsymbol{s}$ and construct the architecture with its i-th block replaced by b_{s_i}. Then, if the individual also contains weights, we load them as the initial weights. Since we defined that individuals carry all weights, we know that we will have initial values for all the weights. If the algorithm does not carry weights within the individuals, we simply initialize weights randomly according to the standard initialization routine as defined by the architecture. Then, we train the network for a defined number of epochs, and after that training we measure the accuracy on validation set that was not used for training. Finally, the fitness of the individual is computed according to the formula defined in previous section.

Due to the way we scale down blocks, we can employ one trick that seemed to improve our results significantly: for a given block b, and some operation in b that uses weight tensor w, we store just a single tensor in the individual; it has dimensions equal to the tensor that is used in the original block b. To load weights from that tensor into the smaller blocks that may need smaller tensors, we just use first few entries across all dimensions. For instance, if some operation in the original block uses a tensor w of shape $[32, 32, 3, 3]$ and it is scaled down so that the same operation in some smaller block uses a tensor t of shape $[16, 16, 2, 2]$, we would use $w[1 \dots 16, 1 \dots 16, 1 \dots 2, 1 \dots 2]$ to initialize t.

4.3.2 Evolutionary Approaches

We propose two slightly different evolutionary algorithms to search for the optimal $\boldsymbol{s}$: **Evo** and **Evo-W**. The difference between the two is that individuals in **Evo-W** also carry the set of weights as described in Section 4.3.1. Pseudocode that applies to both of them is given in Algorithm 3, but some details in the used operations differ between the two variants. We will now describe each of the operations in more detail and emphasize the difference between **Evo** and **Evo-W**.

We also experiment with NSGA-II (Deb et al. [3]) multi-objective variants of **Evo** and **Evo-W**; we will describe changes relevant for them as well.

Algorithm 3: Pseudocode of the evolutionary approaches

```
population ← InitPopulation();
while terminal condition is not met do
    new_population ← few individuals with highest fitness from
     population;
    while size(new_population) < size(population) do
        a, b ← Selection(population);
        with probability p_c do: a, b ← Crossover(a, b);
        for all s in s_a: a ← Mutate(a, s) with probability p_m;
        for all s in s_b: b ← Mutate(b, s) with probability p_m;
        compute fitness for both a and b;
        add both a and b to new_population
    end
    population ← new_population;
end
```

Initialization

We initialize the initial population by sampling each value of s uniformly randomly from the set $\{1, \ldots, 10\}$. In **Evo-W** we also initialize weights randomly according to the routine defined by the corresponding architecture, or load the same weights to all individuals from a pretrained network. Then we train each individual for one epoch on the training data and at the same re-evaluate fitness of the individual.

Selection

We do a standard roulette selection: each individual of the previous generation has probability of selection proportional to his fitness.

In NSGA-II variants of the two algorithms we do the standard NSGA-II tournament selection that compares ranks and then crowding distance to break ties.

Crossover

In **Evo** we employ a standard one-point crossover on the $\boldsymbol{s}$ vectors: we uniformly randomly sample a location and then swap all s-values between the two individuals from that location forward.

For **Evo-W** we also exchange the two sets of weights with 50% probability: a and b keep its weight set with 50% probability, otherwise a will now use weight set of b and b will now use weight set from a.

See Figure 4.4 for illustration of the crossover operation. This operation works in the same way in the NSGA-II variants.

Mutations

To mutate an individual, we go over each of the s-values and first toss a coin with probability 1/2, whether we will increase or decrease that value. Then we do increase or decrease that value with probability 1/4 by 2, otherwise by 1.

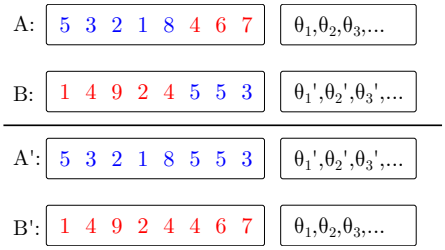


Figure 4.4: Crossover operation in our evolutionary algorithms. With probability $1/2$ the new offspring A' keeps all weights from A, otherwise it inherits them from B, as shown in the picture.

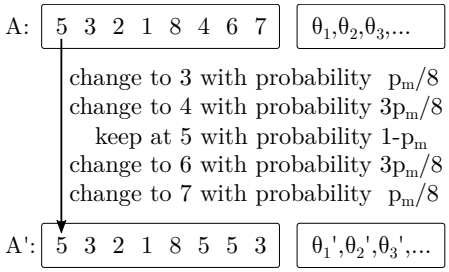


Figure 4.5: Mutation operation in our evolutionary algorithms. For each gene one outcome is chosen according to the probabilities and its value either stays the same, or it is slightly shifted up or down. If the new value was less than 1 or more than 10, it is bounded to 1 or 10 based on which bound was exceeded.

After that we make sure that the new s value is still within the set $\{1, \ldots, 10\}$ by clipping it to the bounds if needed.

See Figure 4.5 for illustration of the mutation operation. This operation also works in the same way in the NSGA-II variants.

We set probability of mutation p_m to $1/n_B$, so that we modify on average one s-value in an individual, which is a common choice in similar algorithms.

Difference in Individual Evaluation

In **Evo-W** we always recompute fitness of all individuals in the population which in turn causes weight set updates (of weights that the individual uses). Because the weights are continually improved over time, we do just one epoch training of the weights and then evaluate the performance on the validation set.

On the other hand, in **Evo** we need to evaluate the architecture for more epochs, because we do not have a starting point for the weights. We chose 10 as the number of epochs to evaluate the individual, consistently with other evolutionary approaches like Deep Pruning by Evolution Strategies (Junior and Yen [18]). Because this takes considerable amount of time, we only re-evaluate individuals that have changed in any way.

4.3.3 Reinforcement Learning

We propose another approach that is closely inspired by Efficient Neural Architecture Search (ENAS, Pham et al. [28]). In ENAS the authors use a standard

reinforcement learning approach to neural architecture search: RNN-based controller guides architecture sampling, the sampled architecture is trained and fitness of the architecture is used as a reward for the controller. However, they share weights among all sampled architectures, and interleave two kinds of steps: training of shared weights and training of the RNN controller. They are able to find good architectures in about half a day on a single GPU that have a test error around 3% on CIFAR-10. This is computationally much less demanding than previous work, as other neural architecture search approach need way more GPU time to search through the architecture space. As we have a limited computational power available, this seems like a good method to adapt. Algorithm 4 shows a pseudocode of the adapted ENAS-like reinforcement learning algorithm and Figure 4.7 shows an illustration of the process.

Algorithm 4: Pseudocode of the Reinforcement Learning approach.

```
Initialize controller and shared_weights;
baseline ← 0 while terminal condition is not met do
    network ← SampleArchitecture(controller);
    Load shared_weights to network and train network for one epoch;
    Store back changed weights from network to shared_weights;
    foreach mini-batch mb of validation data do
        eval_network ← SampleArchitecture(controller);
        Load shared_weights to eval_network;
        accuracy ←Accuracy of eval_network on mb;
        Compute fitness from accuracy and ParamCnt(eval_network);
        baseline ← 0.99baseline + 0.01fitness;
        Use REINFORCE with baseline baseline to update controller
         with reward fitness
    end
end
```

LSTM Controller

We use 11 distinct "characters": 1 special character to start the sampling process, and one for each possible value of s. These characters are embedded[3] into a 64-dimensional space as we use a 64-dimensional LSTM.

To sample an architecture, we pass embedding of the special character to the LSTM as input, and obtain hidden state $\boldsymbol{h}$ and memory $\boldsymbol{c}$, both 64-dimensional vectors. Then, we do as many steps as there are blocks in the architecture (n_B), and each time we:

- Linearly transform current $\boldsymbol{h}$ to a 10-dimensional space, apply softmax over that space to get a distribution and then sample a value s out of that distribution:

  $$s \sim \text{softmax}(W\boldsymbol{h})$$

  where matrix entries $W \in \mathbb{R}^{64\times 10}$ belong to learnable parameters of the controller.

[3] A 64x11 matrix is randomly initialized and when we embed i-th character, we take its i-th column; the backpropagation algorithm can then update values stored in this matrix.

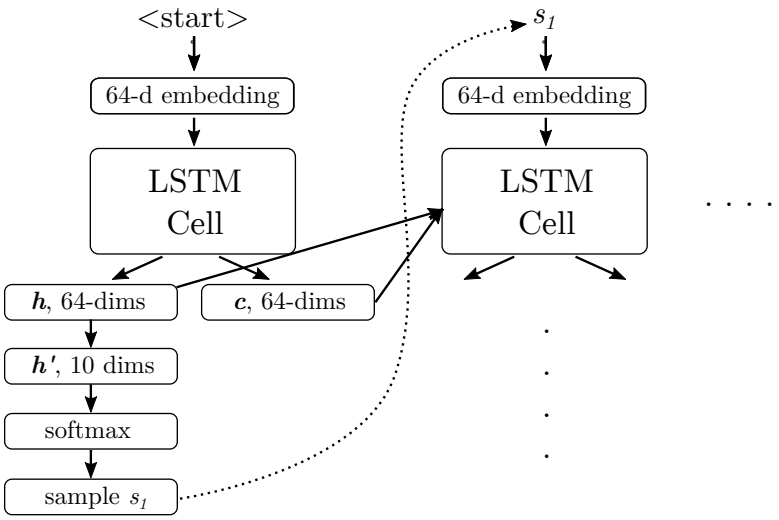


Figure 4.6: Sampling $\boldsymbol{s} = (s_1, \ldots, s_{n_B})$ from the LSTM controller. In the first step we pass embedding of a special character to the LSTM, and finally sample s_1. Afterwards, we pass embedding of s_1 and repeat the entire process to obtain s_2. We proceed similarly until we obtain s_{n_B}.

- Afterwards, we feed embedding of s as next input to the LSTM cell, and obtain new values of $\boldsymbol{h}$ and $\boldsymbol{c}$.

The values of s obtained in this process are stored in the vector $\boldsymbol{s}$ that we use to scale the blocks in the currently sampled architecture. See Figure 4.6 for illustration of the process. To train the controller, REINFORCE with baseline is used; therefore we also keep track of negative log likelihood loss where s plays the role of a correct class. We can then backpropagate the reward through them.

Training Procedure

As we mentioned, two interleaved steps are repeated until we ran the algorithm for enough time: first we sample an architecture to improve shared weights, then we sample one architecture per each mini-batch of the validation data and backpropagate their fitness as a reward to the controller.

4.4 Final Training, Pruning and Fine Tuning

After we have concluded the optimization stage and have an individual with highest fitness, we propose to first train the corresponding architecture for a large number of epochs. Then, we can choose a desired target number of parameters and apply global magnitude pruning[4] to further reduce size of the obtained model. After that, we fine tune the remaining weights to recover some of the model's performance.

These steps will likely not speed up the network anymore, but we found that the obtained networks can still be pruned by a large amount while keeping their performance. Therefore this step yields further savings in space needed to store the network for very low cost.

[4]Pruning of weights with the smallest ℓ_1 norm globally across the network.

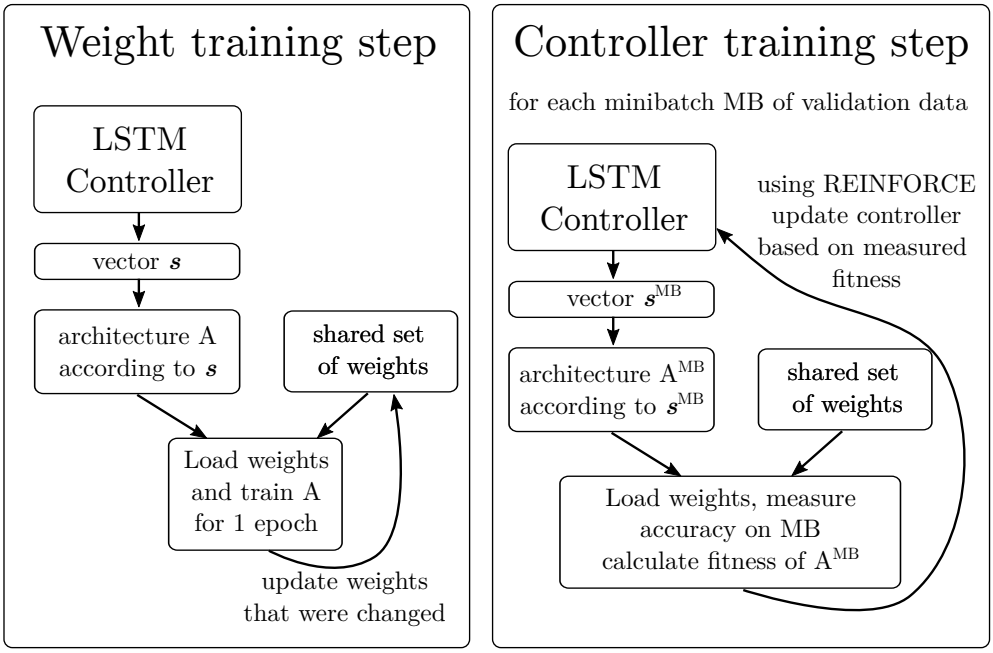


Figure 4.7: RL-based approach: two interleaved steps are performed throughout the algorithm. For each iteration, we first improve the shared weights by sampling an architecture and doing one epoch of training. Afterwards, we iterate over mini batches of validation data and for each mini-batch we sample an architecture, measure its performance on that minibatch and update the controller by REINFORCE algorithm, using fitness as a reward.

5. Experiments

In this chapter we first describe a common experiment setup and network architectures that we used for our experiments. Then we define two baseline methods, afterwards we evaluate the three proposed methods individually and finally, we do an overall comparison of all the methods.

5.1 Experiment Setup

We perform our experiments on architectures that are described in the following section; we use standard augmentations for their training: random horizontal flip of the input image, normalization of mean and standard deviations according to values measured on the training set and finally first randomly padding the input by 4 pixels and then cropping it to a randomly positioned 32×32 window. On validation and test sets, only the normalization is performed (again using values measured on the training set).

Our training procedure performs 150 training epochs[1] using batch size of 128. We start with a high learning rate 0.1 and then decay it according to a cosine annealing schedule (Loshchilov and Hutter [24]) over the 150 epochs.

In most of the experiments we then perform pruning and fine tuning. To evaluate that stage, we try pruning the network with each pruning ratio[2] from the set $\{0.0, 0.1, 0.2, 0.3, 0.4, 0.41, 0.42, \ldots, 0.99\}$. For every such ratio we start with the network obtained after training for the 150 epochs, prune the corresponding amount of weights with lowest ℓ_1 norm and then do fine-tuning.

To fine tune a network, we do 10 training epochs with initial learning rate set to 0.003 that is decayed over the 10 epochs using the same cosine annealing schedule as in the regular training procedure.

Most of our experiments are repeated 5 times. In the graphs, we plot the median value together with error bars that show minimum and maximum values over the 5 runs. We will explicitly highlight experiments that were not performed in this manner.

5.2 Used Architectures

We perform our experiments on ResNets and Wide ResNets (Zagoruyko and Komodakis [38]) in their versions for CIFAR-10. The used architectures are described in more detail in the following section; we used implementation by Idelbayev [17] as a starting point and modified it to support the block scaling operations as described in Section 4.2.2: for a given s we change the number of output channels of the first convolution layer in a block from p to $\lfloor \frac{1}{10} ps + 0.5 \rfloor$.

To give a reference point for the performance of the architectures when trained according to our procedure, we trained each one of them for three times and show the resulting test set accuracy in Table 5.1. The table also lists the number of

[1] One epoch corresponds to one pass over the entire training data

[2] Ratio 0.2 means we prune 20% of the weights in the trained architecture.

parameters and training time per epoch. Training time per epoch was measured on the same GPU: GeForce GTX 1070.

Architecture	Params	Time/Epoch (s)	Test Set Accuracy		
			1	2	3
ResNet-20	269,722	8s	91.04%	91.35%	91.48%
ResNet-32	464,154	13s	92.28%	92.47%	92.89%
ResNet-44	658,586	16s	92.40%	91.88%	92.54%
ResNet-56	853,018	22s	92.66%	92.58%	93.00%
WRN 16-4	2,750,250	32s	93.87%	93.62%	93.76%
WRN 28-2	1,468,746	22s	93.77%	93.91%	94.14%

Table 5.1: Summary of used architectures. Column time/epoch contains training time in seconds for a single pass over the entire training data. All architectures were trained using the procedure described in Section 5.1 three times.

5.3 Baseline Experiments

5.3.1 Global Pruning

In this experiment we take the original architectures as described in Section 5.2 and perform only the last stage of our pipeline: training, pruning and fine tuning. We expect this unstructured method to work well: existing research (e.g. Han et al. [8]) shows that in some networks[3] it is possible to prune around 90% of all parameters at a small cost in accuracy with this technique. Therefore, this method should serve as a relatively strong baseline for reachable accuracy at a given number of remaining parameters.

Obtained results are shown in Figure 5.1. Depending on the size of the initial architecture, we can prune between 75% and 90% of parameters and keep accuracy within 2% of the initial values. A longer fine-tuning procedure might be able to further improve these results, but given that all methods are subject to the same fine-tuning schedule, the relative comparisons should be fair.

5.3.2 Optimization by Random Search

The experiment from the previous section is intended to set a good overall baseline on reachable accuracy for a given number of remaining parameters. On the other hand, in this experiment we aim to verify whether the three proposed variants are able to beat a random algorithm that samples random individuals out of the search space. There are tasks where random algorithms perform comparably to methods similar to ENAS, but also tasks, where ENAS significantly outperforms random search as shown by Bender et al. [1]. Therefore, this baseline may also help us asses the situation for our problem.

We propose to use the following simple random algorithm as a baseline: we select few uniformly random individuals (each gene is assigned value from 1 to 10

[3]It is however easier to prune networks that use large fully connected layers at the end as there are then many redundant parameters.

uniformly randomly), train the corresponding architectures for few epochs and finally pick the one with the highest fitness. We then train the obtained architecture in the standard way, as described in Section 5.1. Note that this algorithm exactly corresponds to the simple proposed evolutionary algorithm when run for 0 generations: we just pick the best individual from the initial population.

We will replace the optimization stage in our algorithm by this random baseline and try two different configurations: **Rand 50C-10E** refers to configuration where we take candidate with the best fitness out of 50 randomly initialized candidates that were trained for 10 epochs, and **Rand 500C-1E**, where we pick the candidate with the best fitness after training for just one epoch (out of 500 candidates).

The results are shown together with global pruning in Figure 5.1. For the random baselines, it interestingly seems that **Rand-50C-10E** performs better than **Rand 500C-1E** on ResNets, while the opposite seems to be true on Wide ResNets. This may be due to the fact that Wide ResNets contain over 2 times more parameters than ResNets and thus 10 epoch estimation does not provide enough information to outperform the larger pool of candidates of **Rand 500C-1E**. Global pruning seems to be much stronger until a certain point, where the original model is already pruned too much and the smaller networks are still not pruned to a high extent (relatively to their initial sizes).

We also show graphs of "prunability" in Figure 5.2: how prunable is the obtained model relative to its initial number of parameters as opposed to the number of parameters of the base architecture. From them it seems that **Rand 50C-10E** is less prunable than the global pruning baseline and **Rand 500C-1E**, while these two seem to be similarly prunable: **Rand 500C-1E** is better at smaller pruning ratios and worse at the end. Worse performance of **Rand 50C-10E** is likely caused by the fact that for most architectures it produced the smallest candidates out of the three methods; results on ResNet-44 and ResNet-56 support that explanation as on these two architectures **Rand 50C-10E** produced similarly sized candidates to **Rand 500C-1E** and the plots show better results.

5.4 Evo and Evo-NSGA-II

Unfortunately, the proposed algorithm called **Evo** that does not store (and use) weights within the individuals takes a lot of time to evaluate. Given that the chosen hyperparameters[4] are already quite low, we decided to perform only a single run per architecture in this experiment.

We compare two ways to do the multi-objective optimization: using the transformation into a single objective optimization by the defined fitness formula and using the NSGA-II algorithm to do the multi-objective optimization directly. Even for the NSGA-II version we internally compute the scalar value of fitness and plot it in the graphs; we just do not use it anywhere within the algorithm itself.

Change in fitness over generations can be seen in Figure 5.4 and Figure 5.5. The interesting thing is that both algorithms seem to reach individuals with

[4]To reiterate, we have 30 individuals in the population, run the algorithm for 50 generations and use 10 epochs to evaluate an individual.

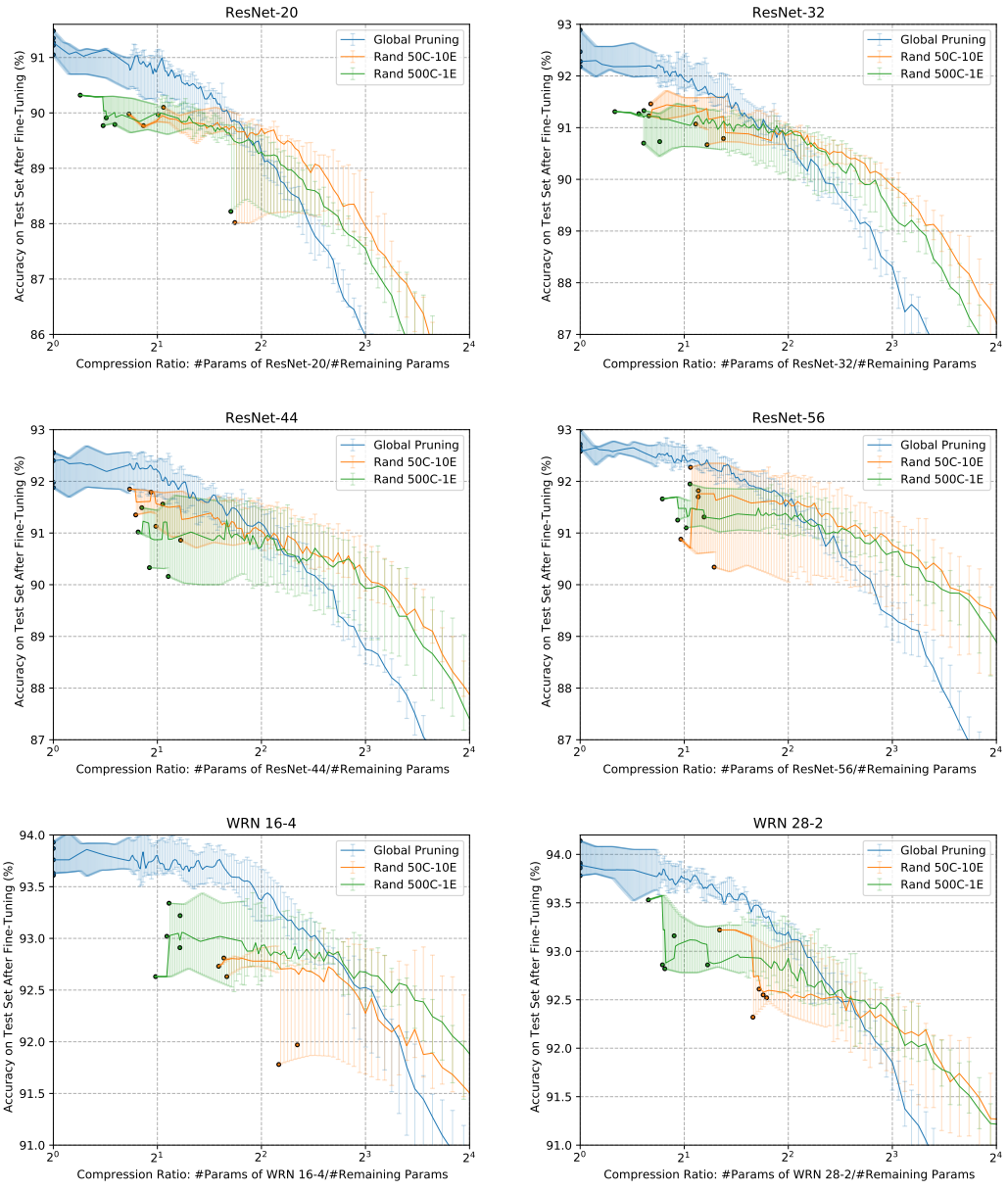


Figure 5.1: Results of global pruning and random baselines. The shown dots correspond to initially trained individuals without any pruning. The x-axis is logarithmic and corresponds to compression ratios as explained in the subfigures.

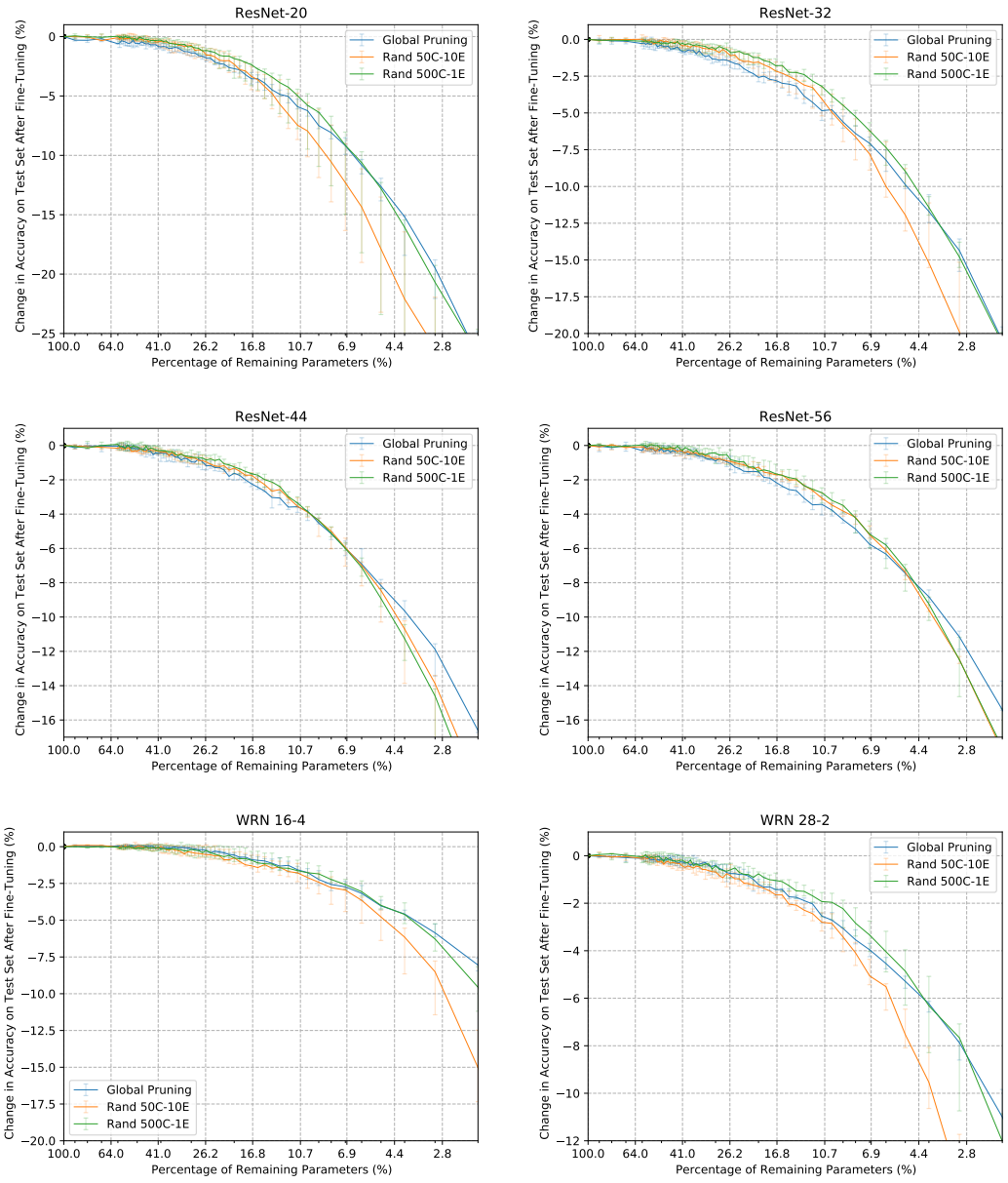


Figure 5.2: Prunability of global pruning and random baselines.

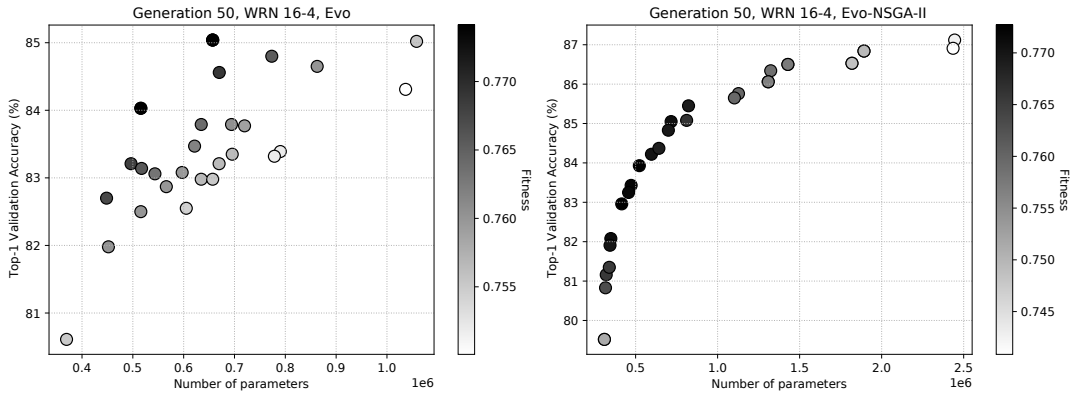


Figure 5.3: Individuals in the population for Wide ResNet 16-4 after 50 generations of **Evo** and **Evo-NSGA-II**.

similar scalar fitness, even though the maximum fitness only changes a few times in **Evo**, while in **Evo-NSGA-II** the individual with best fitness is updated more often.

Figure 5.3 shows population state after 50 generations of **Evo** and **Evo-NSGA-II**. The population in the NSGA-II version shows that after the end of the algorithm we can choose from a wide variety of suitable candidates, depending on the desired parameter count as the individuals approximate the Pareto front of the optimization problem.

5.5 Hyperparameters in Evo-W

To evaluate the role of hyperparameters in **Evo-W**, we did a grid search over cross-over probability and over expected number of mutated genes. We did 5 runs for each combination of $p_c \in \{0.0, 0.2, 0.4, 0.6, 0.8, 0.9\}$ and $m \in \{0.5, 1.0, 1.5, 2\}$, using only ResNet-20 as the base architecture. The results are shown in Figure 5.6.

From the graphs it does not seem that there is a hyperparameter choice that would significantly outperform others as there seems to be a high variance for all choices of hyperparameters. However, as the combination of cross-over probability 0.2 and one average gene mutation seems to reach the best results, we decided to use it for further experiments. We expect most of the other choices to perform similarly. It is possible that on other architectures or for different versions of **Evo-W** the situation might change and there might be a better combination of hyperparameters, but we decided to instead focus on other experiments.

To illustrate the difference between **Evo-W** and **Evo**, we also include two sample graphs of fitness evolution over generations and final generation state in Figure 5.7. Compared to **Evo**, the reached accuracy and fitness is higher and all the individuals improve in (almost) all generations; this is expected as we train the weights of all individuals in every generation for a single epoch.

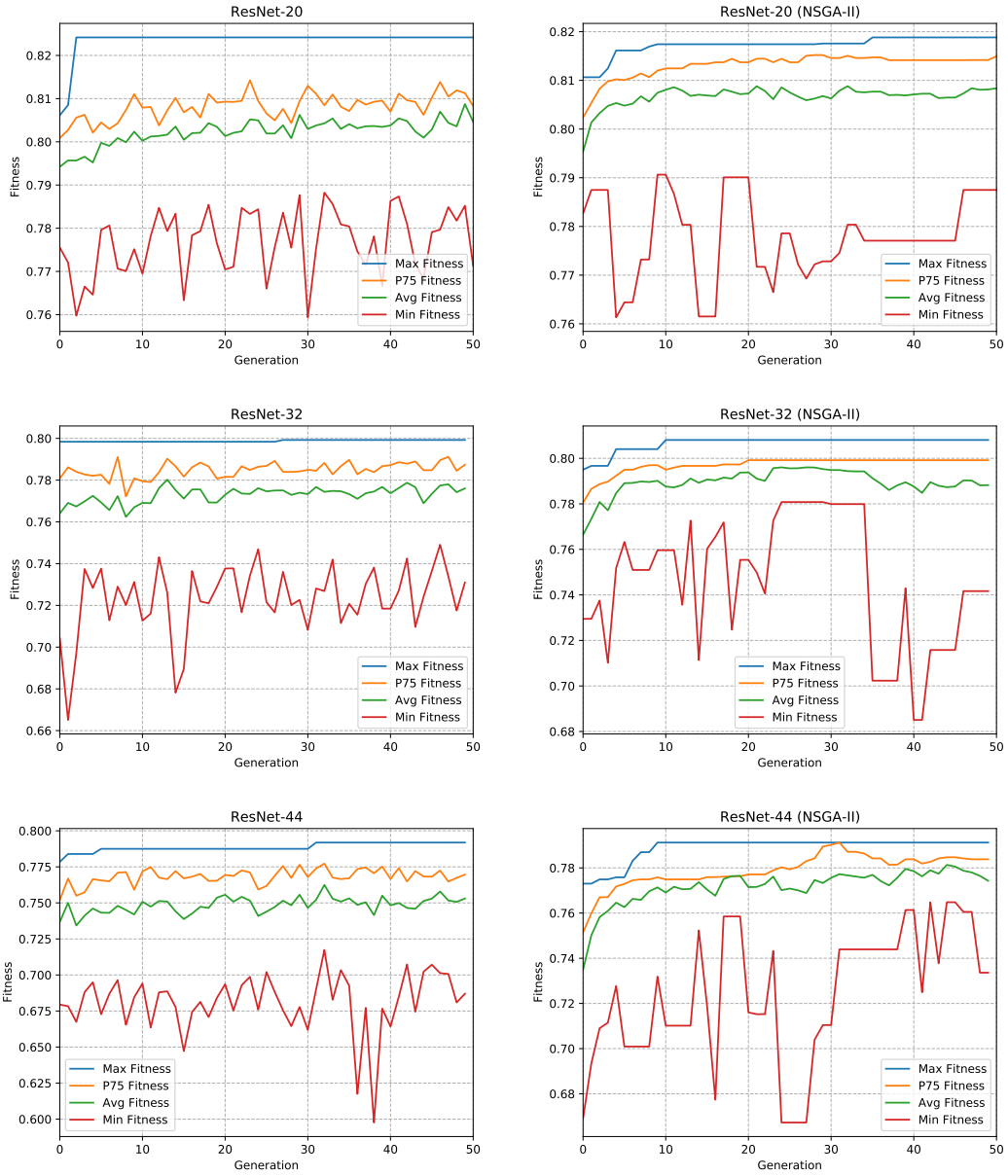


Figure 5.4: Maximum, P75, mean and minimum fitness of individuals over generations in **Evo** and **Evo-NSGA-II** on ResNet-20, ResNet-32 and ResNet-44.

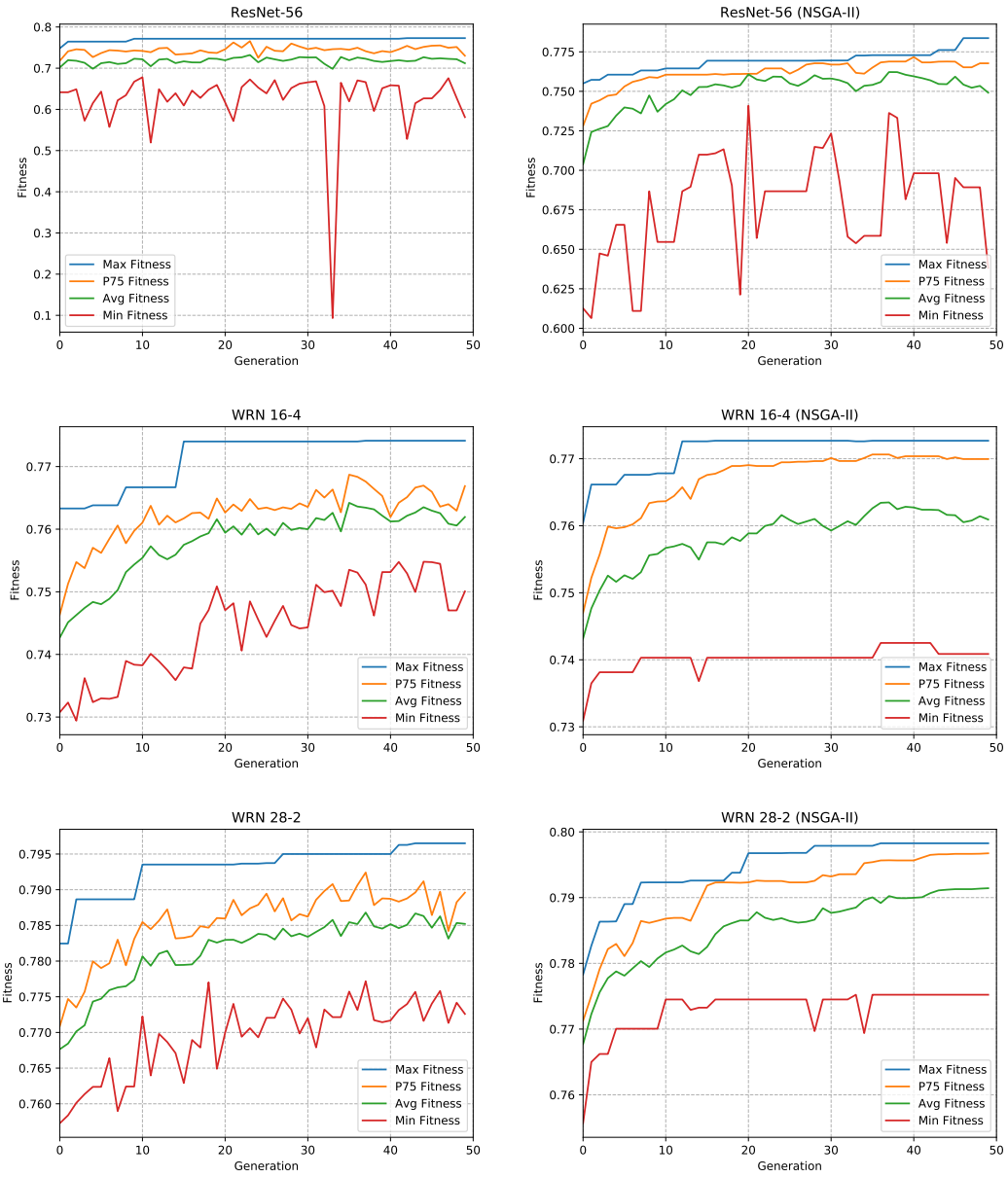


Figure 5.5: Maximum, P75, mean and minimum fitness of individuals over generations in **Evo** and **Evo-NSGA-II** on ResNet-56, Wide ResNet 16-4 and Wide ResNet 28-2.

5.6 Weight Initialization and Layer Freezing in Evo-W

In this experiment we evaluate the impact of weight initialization and layer freezing in **Evo-W**. We experiment with the following 3 modifications:

- **Evo-W-FREEZE**: When doing the 1 epoch training of each individual, we only update weights in blocks for which s has changed. Weight of other layers are frozen.
- **Evo-W-INITW**: All individuals in the starting population are initialized with the same weights: we first train the base architecture according to the training procedure and use it for the initialization.
- **Evo-W-INIT+FREEZE**: A combination of the previous two versions.

The results are shown in Figure 5.8. This time we decided to hide the error bars as the variance was similarly high to the previous experiments and they did not provide much information. Therefore, we only plot the median values. We also include the two random baselines for comparison.

It again seems that there is not a clear "winning option" among the considered versions. Also, in some cases the random baselines seem to outperform the **Evo-W** method. More specifically, on some architectures (ResNet-20, ResNet-44, ResNet-56) both random baselines seem to be outperformed by most **Evo-W** variants. On the other hand, on WRN 16-4 both random baselines seem perform the best. Experiments with the remaining architectures show a comparable result for all versions, including the random baselines.

5.7 Experiments with REINFORCE method

Initially, we made a mistake and implemented the proposed RL algorithm in a slightly different way to the ENAS version: in the main loop we first sampled an architecture, trained the shared network weights on it and then measured validation accuracy of that same architecture. The intended training routine however samples an architecture, trains the shared network weights, and then trains the controller by iterating over mini-batches of the validation set. For each mini-batch a new architecture is sampled and without any training the accuracy on that mini-batch is measured right after loading the shared weights. The controller is therefore updated on each mini-batch. We decided to compare these two methods and refer to the first "wrong" one as **REINFORCE** and to the other as **REINFORCE-VAL-MB** (or R-V-MB).

Figure 5.9 shows that **REINFORCE** produces smaller models for the same hyperparameters and models that eventually have better test set accuracy than simple pruning when more than approximately 75% parameters are pruned away. On the other hand, **REINFORCE-VAL-MB** seems to commonly reduce the number of parameters by 20-40% by itself without almost any loss in accuracy; in fact, the obtained slightly smaller architectures actually often outperform the initial architecture on reached accuracy. Both variants seem to consistently outperform all of the baselines.

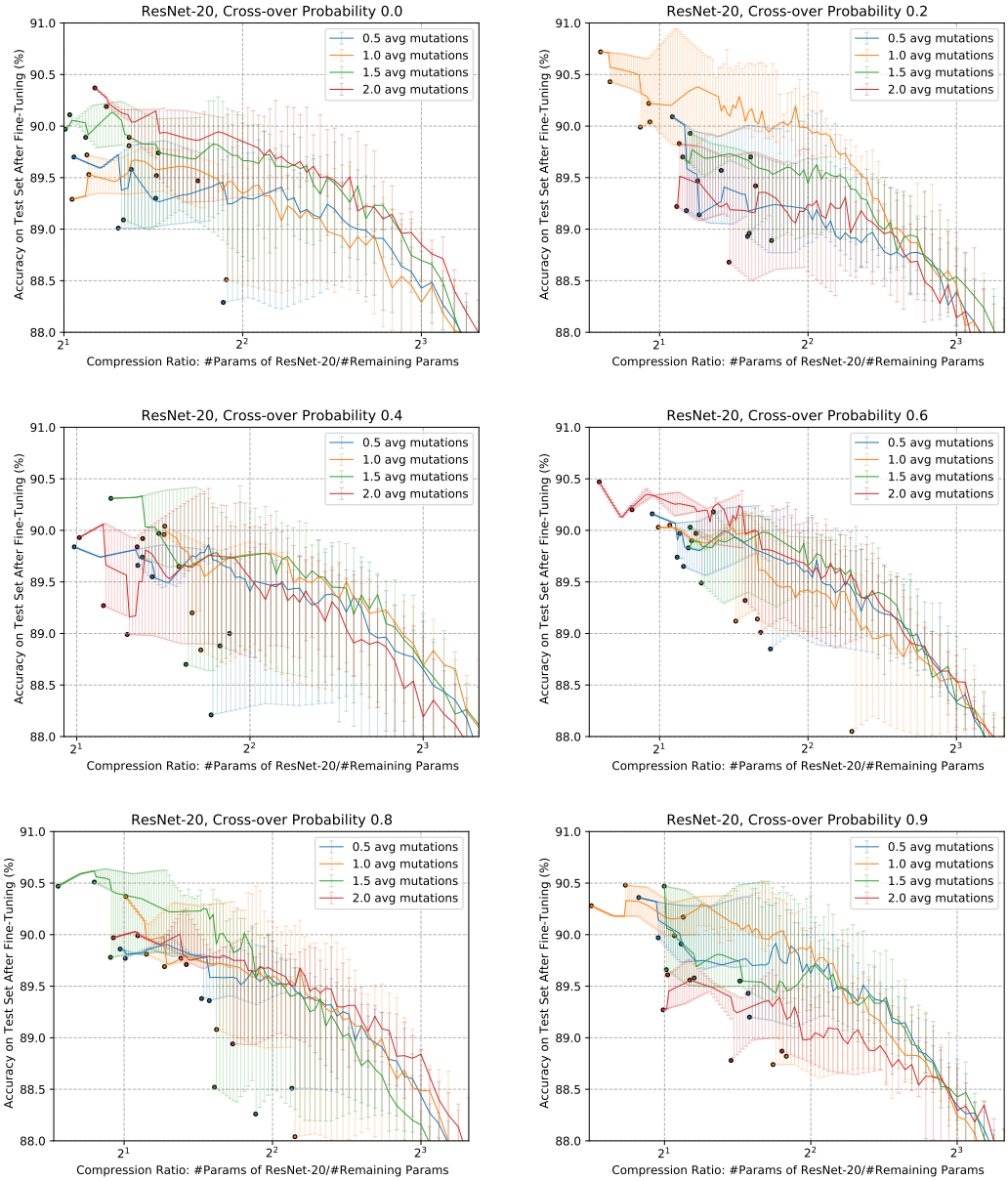


Figure 5.6: Role of cross-over and mutation probabilities in **Evo-W** on ResNet-20. The shown dots correspond to initially trained individuals without any pruning. The x-axis is logarithmic and corresponds to compression ratios as explained in the subfigures.

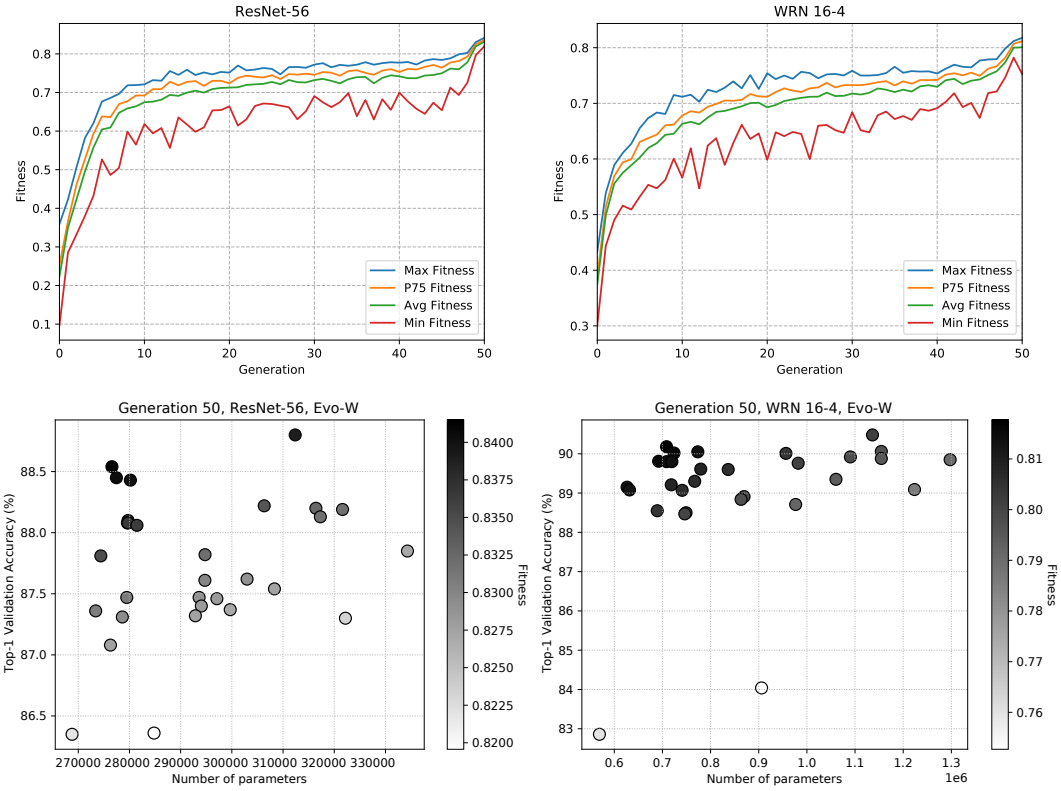


Figure 5.7: **Evo-W** plots on ResNet-56 and WRN 16-4. First row: Maximum, P75, average and minimum fitness in each generation. Second row: state of the population after 50 generations. See Figure 5.5 and Figure 5.3 for comparison with **Evo**.

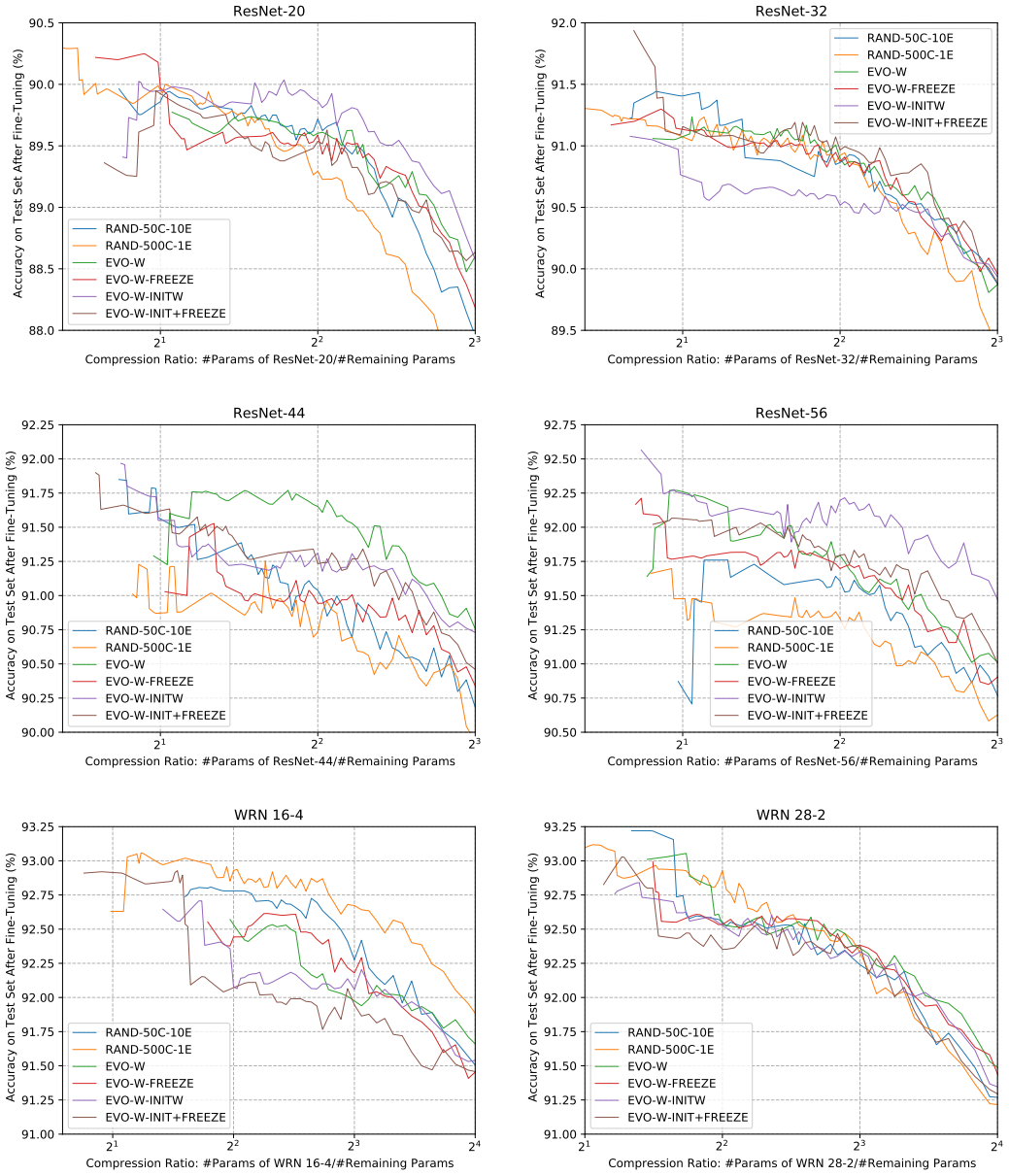


Figure 5.8: Comparison of **Evo-W**, **Evo-W-FREEZE**, **Evo-W-INITW**, and **Evo-W-INIT+FREEZE**. The x-axis is logarithmic and corresponds to compression ratios as explained in the subfigures. The lines show the current value of median among available runs for the given compression ratio.

Another interesting thing is that despite having less parameters, the architectures obtained by these methods seem to be slightly more prunable than the base architectures as shown in Figure 5.11.

To further explore capabilities of **REINFORCE-VAL-MB**, we decided to force it to produce smaller architectures by setting the number of target parameters T to first half and then quarter of the base architecture size and also by using $w = -1$ in the fitness formula for candidates that exceed T. This was proposed by Tan et al. [33] in MnasNet's neural architecture search to enforce T more strictly. We denote these two restricted methods by **REINFORCE-VAL-MB-0.5** and **REINFORCE-VAL-MB-0.25**.

The results of these two more constrained versions against the two former versions are shown in Figure 5.10. **REINFORCE-VAL-MB-0.5** seems to achieve similar or better results compared to **REINFORCE** even though it often starts with fewer parameters. **REINFORCE-VAL-MB-0.25** seems to be too affected by the constraints compared to the other variants, but from Table 5.3 it also seems that it is generally hard to structurally prune 75% or more of the parameters.

The results of **REINFORCE-VAL-MB-0.25** might potentially be improved by taking more fine-grained scaling steps: a different block scaling definition that produces blocks reduced by 75% or more for all $s \leq 5$ might help as the algorithm is currently constrained to choose from very few options (to keep the average below 2.5, the algorithm "needs to pay" for every block with $s \geq 3$ with one or more blocks for which $s \in \{1, 2\}$).

5.8 Comparisons to Existing Methods

5.8.1 Pruning Filters for Efficient ConvNets

Li et al. [21] in their paper "Pruning Filters for Efficient ConvNets" propose a method that is based on removing entire convolution kernels: they sort them according to the sum of absolute weight values and then discard the ones with lowest sum. The amount that is discarded varies from layer to layer, based on their manually determined sensitivity to pruning. Also, for ResNets they add a specific logic that ensures that the residual connections and output of each block have the same dimension: the last convolution layer and the residual projection shares the pruning information.

On CIFAR-10 they experiment with VGG-16, ResNet-56 and ResNet-110 architectures. They use the same version of ResNet-56 as we do that uses 8.5×10^7 parameters. They report that they are able to prune around 13.7% of parameters and at the same time improve accuracy by 0.02% from 93.04% to 93.06%. They also reduce number of required FLOPS by 27.6%.

In our experiments the original ResNet-56 architecture reaches slightly lower accuracy: $92.66\%, 92.58\%, 93.00\%, 92.58\%, 92.72\%$ in the 5 corresponding training runs and therefore our training procedure might be weaker. If we compare drop of accuracy of the original architecture, our results seem stronger on ResNet-56: we can prune almost half of the model, raise mean accuracy by 0.1% and reduce the number of required FLOPS by 42.52%.

We show the measured values in Table 5.2. Our method also does not need manual analysis of sensitivity and then corresponding choice of pruning ratios

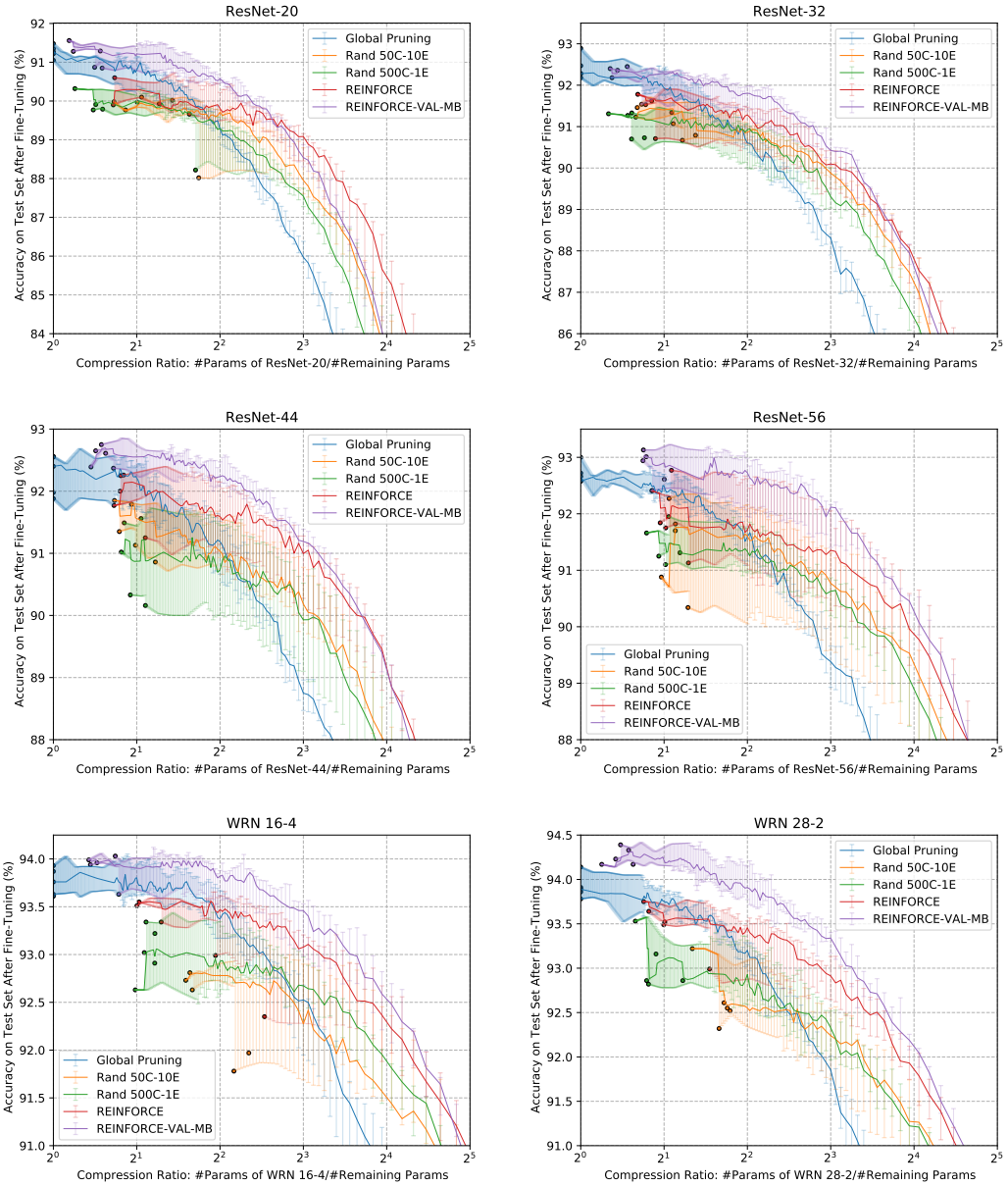


Figure 5.9: Comparison of global pruning and random baselines, **REINFORCE** and **REINFORCE-VAL-MB**. The shown dots correspond to initially trained individuals without any pruning. The x-axis is logarithmic and corresponds to compression ratios as explained in the subfigures.

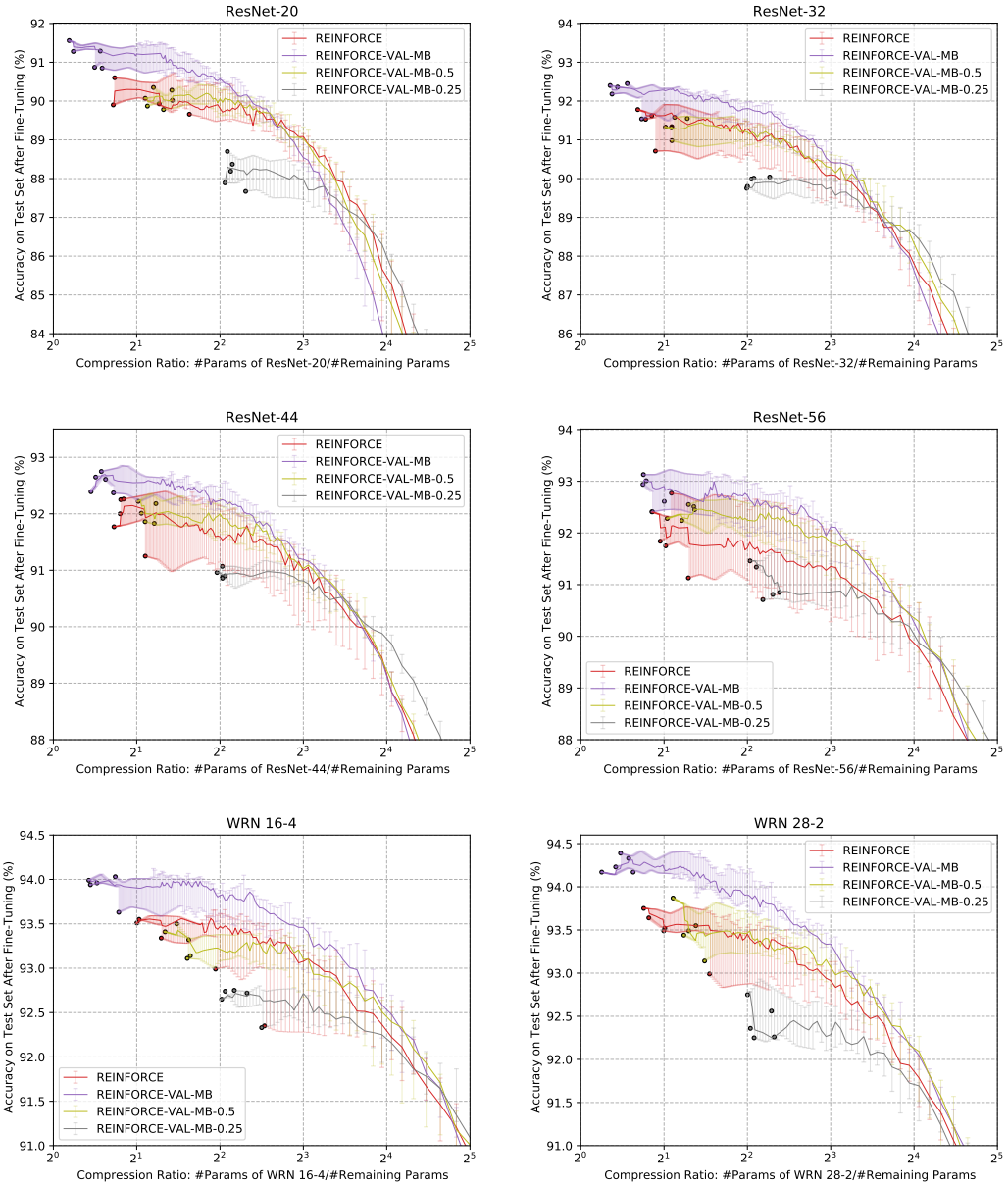


Figure 5.10: Comparison of all evaluated versions that are based on REINFORCE and ENAS. The shown dots correspond to initially trained individuals without any pruning. The x-axis is logarithmic and corresponds to compression ratios as explained in the subfigures.

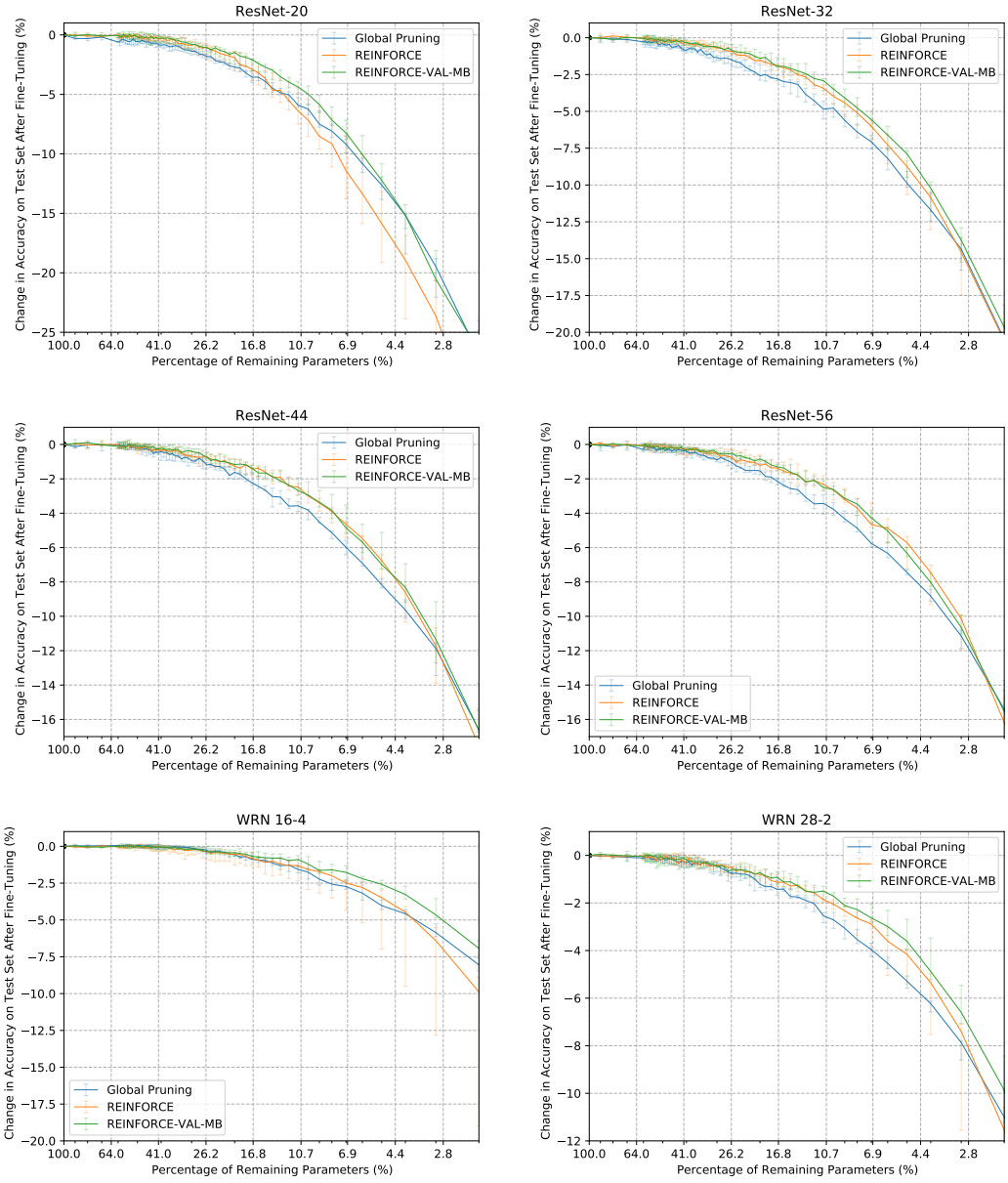


Figure 5.11: Prunability comparison of **REINFORCE** and **REINFORCE-VAL-MB**. Both methods achieve similar levels of prunability, while also outperforming the initial architecture in prunability on architectures other than ResNet-20.

across layers. However, our method needs to train the architecture roughly 7 times more often: we do 1000 training epochs in REINFORCE-based methods and around 150 epochs for training the obtained architecture. On the other hand, Li et al. [21] after the manual analysis perform only one training and one fine tuning session, i.e. an equivalent of 160 epochs in our setting.

Model / Run #	Params	Pruned Par. (%)	Accuracy	FLOPS
ResNet-56 (ours)	853,018	0%	92.71%	127,066,762
ResNet-56 ([21])	853,018	0%	93.04%	127,066,762
Pruned-B	730,000	13.7%	93.06%	90,900,000
R-V-MB / 1	**425,788**	**50.01%**	92.61%	**67,923,466**
R-V-MB / 2	469,928	44.91%	92.41%	70,581,002
R-V-MB / 3	506,482	40.62%	**93.13%**	80,062,858
R-V-MB / 4	508,316	40.41%	92.94%	73,707,530
R-V-MB / 5	494,640	42.01%	93.01%	76,799,370
R-V-MB / avg	481,031	43.59%	92.82%	73,814,845
R-V-MB-0.5 / 1	332,854	60.98%	**92.61%**	54,799,498
R-V-MB-0.5 / 2	415,510	51.29%	92.28%	61,676,938
R-V-MB-0.5 / 3	330,946	**61.20%**	92.45%	**50,697,226**
R-V-MB-0.5 / 4	348,006	59.20%	92.55%	57,908,106
R-V-MB-0.5 / 5	367,882	56.87%	92.24%	55,074,058
R-V-MB-0.5 / avg	359,040	57.91%	92.41%	56,031,165
R-V-MB-0.25 / 1	163,270	**80.86%**	90.85%	35,922,442
R-V-MB-0.25 / 2	198,014	76.79%	91.34%	33,493,898
R-V-MB-0.25 / 3	172,604	79.77%	90.81%	**28,254,218**
R-V-MB-0.25 / 4	208,734	75.53%	**91.46%**	44,266,506
R-V-MB-0.25 / 5	187,470	78.02%	90.71%	31,462,794
R-V-MB-0.25 / avg	186,018	78.19%	91.03%	34,679,972

Table 5.2: Comparison of ResNet-56-Pruned-B (Li et al. [21]) and **REINFORCE-VAL-MB** variants on ResNet-56. No further pruning or fine tuning is performed after training the obtained architecture for 150 epochs.

5.8.2 Other Approaches

As there are many pruning approaches that emerged in the last few years, we adapt a table from 2020 by Lin et al. [22] that compares multiple recent pruning approaches on a combination of CIFAR-10 dataset and ResNet-56 architecture. It includes the following methods:

- HRank: Filter Pruning using High-Rank Feature Map by Lin et al. [22]
- L1: Approach from Section 5.8.1 by Li et al. [21]
- NISP: Pruning Networks using Neuron Importance Score Propagation by Yu et al. [37]
- GAL: Towards Optimal Structured CNN Pruning via Generative Adversarial Learning by Lin et al. [23]

- CHAN: Channel Pruning for Accelerating Very Deep Neural Networks by He et al. [11]

The data are displayed in Table 5.3 and show that our methods based on REINFORCE and ENAS obtain competitive results. The accuracy of any two different methods might not be directly comparable due to differences in training procedure and accuracy of the initial pruned model: for instance, Lin et al. [22] pruned a baseline model that reached 93.26% accuracy, while our mean accuracy on plain ResNet-56 over 5 runs is 92.71%.

Despite that, we believe that they show that our method performs similarly well across the entire range of FLOPS, and outperforms other methods around 55% reduction in FLOPS. We believe this is a very good result given that number of FLOPS is only our indirect objective: our objective other than accuracy was minimization of the number of parameters, while the other methods specifically focus on optimization of the number of FLOPS.

Model	Accuracy	FLOPS(% pruned)	Params(% pruned)
ResNet-56	93.26%	125.49M(0%)	0.85M(0.0%)
L1 [21]	93.06	90.90M(27.6%)	0.73M(14.1%)
HRank [22]	93.52%	88.72M(29.3%)	0.71M(16.8%)
NISP [37]	93.01%	81.00M(35.5%)	0.49M(42.4%)
GAL-0.6 [23]	92.98%	78.30M(37.6%)	0.75M(11.8%)
R-V-MB	92.82%	73.81M(40.95%)	0.48M(43.61%)
HRank [22]	93.17%	62.72M(50.0%)	0.49M(42.4%)
CHAN [11]	90.80%	62.00M(50.6%)	N/A
R-V-MB-0.5	92.41%	56.03M(55.4%)	0.36M(57.91%)
GAL-0.8 [23]	90.36%	49.99M(60.2%)	0.29M(65.9%)
R-V-MB-0.25	91.03%	34.68M(72.4%)	0.19M(78.2%)
HRank [22]	90.72%	32.52M(74.1%)	0.27M(68.1%)

Table 5.3: Results of the three versions of **REINFORCE-VAL-MB** compared to multiple recent pruning approaches on a CIFAR-10 dataset using ResNet-56 architecture as a base. Data of the other models are taken from Lin et al. [22]. Our data are mean values over 5 experiment runs.

5.9 Overall Results

To summarize performance of all of our methods, we include Figure 5.12 that shows each baseline and one or more representatives of each method in one graph per architecture. It also shows that **REINFORCE-VAL-MB** performs almost in all cases and across most of the parameter range the best. Even though it starts with a relatively large number of parameters, it does not drop accuracy as much as global pruning of the base architecture.

For **Evo-NSGA-II** we have chosen as a representative the individual with the highest scalar fitness. On ResNet-20, ResNet-32 and ResNet-44 **Evo** generally performs relatively well and better than **Evo-NSGA-II**; on other architectures it is worse than **Evo-NSGA-II** and both of them are among the worst methods.

We have chosen **Evo-W-INITW** as a representative for **Evo-W**. On ResNet-56 it achieves a good result, otherwise it struggles to overcome the random baselines. It seems to perform comparably to the better one from **Evo** and **Evo-NSGA-II**, even though it performs approximately 10 times less training.

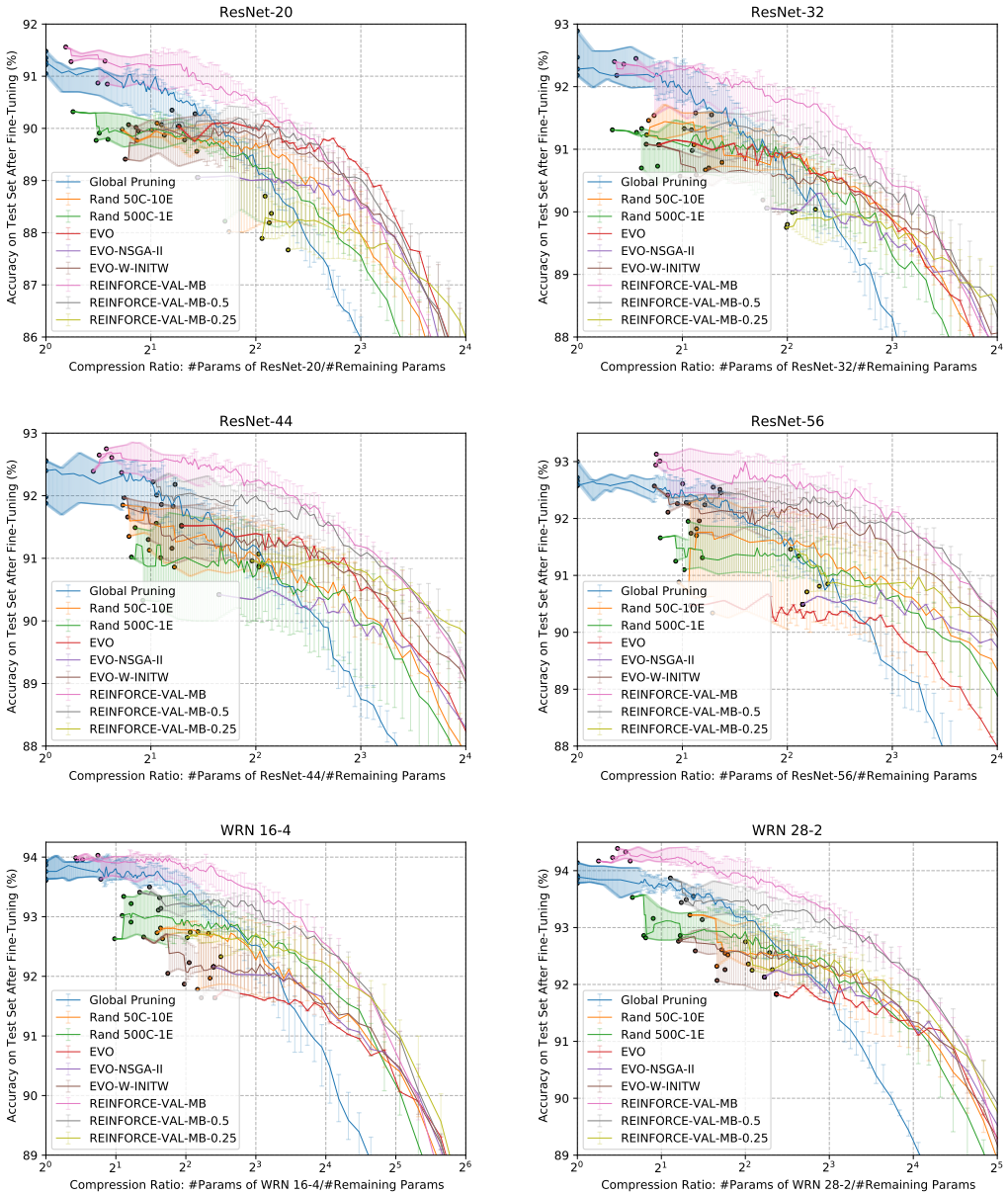


Figure 5.12: Overall Results. The x-axis is logarithmic and corresponds to compression ratios as explained in the subfigures.

Conclusion

Our goal was to develop new methods that take an existing convolutional neural network architecture and produce a smaller architecture that has comparable accuracy. We proposed a new general three stage reduction scheme that is based on manual parameterization of the initial architecture, search for the optimal architecture within the parameterization and finally a training stage, where we perform training, further pruning and fine tuning of the found architecture.

We focused on the architecture search stage in our experiments and evaluated three algorithms: a simple evolutionary algorithm, evolutionary algorithm with weights and a reinforcement learning method. The simple evolutionary algorithm unfortunately requires a lot of GPU-time as accuracy needs to be estimated from scratch every time we change an individual. Also, its results are not strong enough to justify the time spent on the computation.

The evolutionary algorithm that uses weights is roughly 10 times faster than the simple one. On some architectures (like ResNet-56) it is able to beat the random baselines and also reduce number of parameters by 50% and keep accuracy within 0.5% of the base architecture.

The method based on reinforcement learning and extensive weight sharing performed the best in our experiments. On all tested architectures it pruned a significant amount (20-40%) of parameters and at the same time it improved accuracy compared to the base architecture. We tried to push the algorithm to its limits by forcing it to prune larger portions of the architecture. The results of that experiment show that this method actually finds more efficient architectures than the other two methods and does not merely produce big architectures that have higher chance to reach high accuracy. By itself it is competitive with recent pruning approaches, and with further pruning the number of parameters can be reduced by 75-85% while preserving accuracy of the base architecture.

Future Work

As we intentionally designed our method to be relatively general, there are many ways in which it can be extended or further explored.

In the parameterization stage, different block types can be substituted for the original ones. We briefly mentioned and described this, but decided not to proceed further: mainly because blocks that used depth-wise separable convolutions were on our GPU much slower than the original blocks and also because it did not seem fair to compare the new architectures to the base one. However, employing block substitution might definitely improve the results.

Another thing that could be explored in this stage is the block scaling routine. Different mapping of s to the number of intermediate channels. Similarly, new layers that change the number of channels may be considered, different ranges of s (not just 1 to 10), etc.

For the optimization stage, new algorithms could be evaluated. For the existing ones, a more in-depth hyperpararameter search could be performed. More experiments on larger architectures and ImageNet would also be helpful in further evaluation of the proposed methods.

Finally, even the training and pruning could be done in some other way: fine tuning setup of more epochs could improve the results, different algorithm that performs pruning or perhaps even an iterative version of the entire scheme may all further improve the results.

Bibliography

[1] Gabriel Bender, Hanxiao Liu, Bo Chen, Grace Chu, Shuyang Cheng, Pieter-Jan Kindermans, and Quoc Le. Can weight sharing outperform random architecture search? An investigation with TuNAS, 2020. Cited on page 42.

[2] Yann Le Cun, John S. Denker, and Sara A. Solla. *Optimal Brain Damage*, page 598–605. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990. ISBN 1558601007. Cited on pages 23, 24, and 25.

[3] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *Trans. Evol. Comp*, 6(2): 182–197, April 2002. ISSN 1089-778X. doi: 10.1109/4235.996017. URL `https://doi.org/10.1109/4235.996017`. Cited on pages 34 and 35.

[4] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient Multi-objective Neural Architecture Search via Lamarckian Evolution. *arXiv preprint arXiv:1804.09081*, 2018. Cited on page 22.

[5] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018. Cited on page 24.

[6] Felix A. Gers, Jürgen A. Schmidhuber, and Fred A. Cummins. Learning to forget: Continual prediction with lstm. *Neural Comput.*, 12(10):2451–2471, October 2000. ISSN 0899-7667. doi: 10.1162/089976600300015015. URL `https://doi.org/10.1162/089976600300015015`. Cited on page 14.

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

[8] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015. Cited on pages 23 and 42.

[9] Babak Hassibi and David G Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in neural information processing systems*, pages 164–171, 1993. Cited on pages 23 and 24.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. Cited on pages 18 and 19.

[11] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks, 2017. Cited on page 58.

[12] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015. Cited on pages 26 and 27.

[13] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL https://doi.org/10.1162/neco.1997.9.8.1735. Cited on page 14.

[14] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017. Cited on page 21.

[15] Jia-Bin Huang, Abhishek Singh, and Narendra Ahuja. Single image super-resolution from transformed self-exemplars. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5197–5206, 2015. Cited on page 12.

[16] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016. Cited on pages 3, 19, and 20.

[17] Yerlan Idelbayev. Proper ResNet implementation for CIFAR10/CIFAR100 in PyTorch. https://github.com/akamaster/pytorch_resnet_cifar10. Cited on page 41.

[18] Francisco Erivaldo Fernandes Junior and Gary G Yen. Pruning Deep Neural Networks Architectures with Evolution Strategy. *arXiv preprint arXiv:1912.11527*, 2019. Cited on pages 25 and 37.

[19] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009. Cited on page 6.

[20] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989. Cited on page 12.

[21] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning Filters for Efficient Convnets. *arXiv preprint arXiv:1608.08710*, 2016. Cited on pages 53, 57, and 58.

[22] Mingbao Lin, Rongrong Ji, Yan Wang, Yichen Zhang, Baochang Zhang, Yonghong Tian, and Ling Shao. Hrank: Filter pruning using high-rank feature map, 2020. Cited on pages 57 and 58.

[23] Shaohui Lin, Rongrong Ji, Chenqian Yan, Baochang Zhang, Liujuan Cao, Qixiang Ye, Feiyue Huang, and David Doermann. Towards optimal structured cnn pruning via generative adversarial learning, 2019. Cited on pages 57 and 58.

[24] Ilya Loshchilov and Frank Hutter. SGDR: Stochastic Gradient Descent with Restarts. *CoRR*, abs/1608.03983, 2016. URL http://arxiv.org/abs/1608.03983. Cited on page 41.

[25] Zhichao Lu, Gautam Sreekumar, Erik Goodman, Wolfgang Banzhaf, Kalyanmoy Deb, and Vishnu Naresh Boddeti. Neural Architecture Transfer. *arXiv preprint arXiv:2005.05859*, 2020. Cited on page 22.

[26] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter. Nvidia tensor core programmability, performance precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531, 2018. Cited on page 27.

[27] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning Convolutional Neural Networks for Resource Efficient Inference. *arXiv preprint arXiv:1611.06440*, 2016. Cited on page 25.

[28] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing, 2018. Cited on page 37.

[29] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958. Cited on page 7.

[30] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL `http://incompleteideas.net/book/the-book-2nd.html`. Cited on page 15.

[31] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks, 2020. Cited on pages 22 and 34.

[32] Mingxing Tan and Quoc V. Le. EfficientNetV2: Smaller Models and Faster Training, 2021. Cited on page 34.

[33] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile, 2019. Cited on pages 22 and 53.

[34] Lucas Theis, Iryna Korshunova, Alykhan Tejani, and Ferenc Huszár. Faster Gaze Prediction with Dense Networks and Fisher Pruning. *arXiv preprint arXiv:1801.05787*, 2018. Cited on page 25.

[35] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Advances in neural information processing systems*, pages 7675–7684, 2018. Cited on page 28.

[36] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992. Cited on page 15.

[37] 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, 2018. Cited on pages 57 and 58.

[38] Sergey Zagoruyko and Nikos Komodakis. Wide Residual Networks. *CoRR*, abs/1605.07146, 2016. URL `http://arxiv.org/abs/1605.07146`. Cited on pages 33 and 41.