FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Anytime Learning with Auto-Sizing Neural Networks
Student:	Bc. Vojtěch Cahlík
Supervisor:	doc. Ing. Pavel Kordík, Ph.D.
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2022/2023

Instructions

Survey existing work on auto-sizing and similar approaches in the area of feedforward neural networks. Design and implement anytime learning algorithms that utilize auto-sizing. Training should be efficient in regard to selected criteria, and the generalization performance of the models should keep improving as long as training progresses. Explore dynamic regularization methods and heuristic methods that enable to control the progress of training. Evaluate the performance of selected methods with fully-connected and convolutional models and discuss the results.

Electronically approved by Ing. Karel Klouda, Ph.D. on 12 October 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Anytime Learning with Auto-Sizing Neural Networks

Bc. Vojtěch Cahlík

Department of Applied Mathematics
Supervisor: doc. Ing. Pavel Kordík, Ph.D.

May 3, 2022

Acknowledgements

First and foremost, I would like to thank my supervisor, Mr. Kordík, for his help and guidance in writing this thesis. I am also grateful for my colleagues being very sympathetic and allowing me to focus on the writing as well as on other study activities. Last but not least, I would like to thank my partner, family, and friends for their patience and hearty support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 3, 2022 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Cahlík, Vojtěch. *Anytime Learning with Auto-Sizing Neural Networks.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Algoritmy označované jako anytime slouží k produkování aproximativních výsledků, jejichž kvalita se s výpočetním časem zlepšuje. Tato diplomová práce se zaměřuje na aplikaci anytime algoritmů v úlohách strojového učení za využití metody auto-sizing, která umožňuje efektivní prořezávání komponent umělých neuronových sítí pomocí gradientní optimalizace. V rámci diplomové práce je původní auto-sizing rozšířen do metody nazvané dynamický auto-sizing, která umožňuje měnit velikost a strukturu modelů během tréninku upravováním aplikované síly regularizace, a tato technika je dále začleněna do několika anytime algoritmů strojového učení. Výsledky experimentů ukazují, že dynamický auto-sizing může být úspěšně použit v různorodých klasifikačních a regresních úlohách, často s lepšími výsledky než za použití tradičních přístupů.

Klíčová slova anytime strojové učení, hluboké učení, auto-sizing, regularizace, strukturovaná řídkost, anytime algoritmy, AutoML

Abstract

Anytime algorithms produce approximative results whose quality improves with computation time. The thesis focuses on applying anytime algorithms

on machine learning tasks with use of auto-sizing neural networks, which are deep learning models that can efficiently prune their components during training and are trainable with gradient-based optimization methods. As part of the thesis, auto-sizing is extended into a novel technique called dynamic auto-sizing, which allows to dynamically change the size and structure of the models during training according to the applied regularization strength, and the technique is incorporated into several anytime learning algorithms. The experimental evaluation shows that dynamic auto-sizing models can successfully be used in various classification and regression tasks and often provide an improvement in predictive performance over traditional approaches.

Keywords anytime learning, deep learning, auto-sizing, regularization, structured sparsity, anytime algorithms, AutoML

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With today's prevalent approach to training deep learning models, it is necessary to explicitly set the hyperparameters controlling the sizes of hidden layers of the model, specifically the numbers of units such as neurons or filters. Computationally intensive tuning of these hyperparameters is then required in order to find an efficient architecture. Moreover, such models can not automatically prune unnecessary components or adjust their size to the problem domain. To address this limitation, methods such as *auto-sizing* have been proposed for efficiently choosing the layer sizes as well as for parameter pruning [1, 2, 3, 4]. However, as these methods still operate on top of a static model with a fixed architecture, they are still somewhat computationally ineffective and also can not grow the models beyond their original sizes. More complex techniques such as *MorphNet* exist that allow the models to grow new units in hidden layers [5, 6], however the sizes of the resulting models are then dependent on explicit resource constraints such as the number of FLOPS, instead of the difficulty of the task.

This thesis builds upon original auto-sizing by introducing *dynamic auto-sizing*, a novel method for producing feed-forward artificial neural network models that can dynamically change their structure during training. Dynamic auto-sizing enables the hidden layers of such models to automatically shrink or grow by adjusting the numbers of hidden units during training, using the novel *weighted* l_1 *regularization* technique, which induces structured sparsity in the model by penalizing every additional unit more, until some units are regularized so much that they effectively do not contribute to the model's outputs. During training, new units are periodically grown and unnecessary units are periodically pruned by being completely removed from the hidden layers, until the model stabilizes at a final architecture. The size of the resulting model typically increases with growing complexity of the problem domain, and can be further controlled by adjusting the strength of regularization.

In the thesis, dynamic auto-sizing is experimentally analyzed on selected tasks, with focus on *anytime learning* settings in which the quality of the

obtained models gradually improves as training progresses, thus introducing a trade-off between computational costs and predictive performance of the models. Novel anytime learning algorithms that utilize dynamic auto-sizing are designed, with close relation to the AutoML subfields of hyperparameter optimization and neural architecture search. The algorithms are experimentally evaluated with promising results, which demonstrate the potential for use of dynamic auto-sizing in everyday machine learning tasks.

The thesis is structured as follows. Chapter 2 explains the theoretical background. Chapter 3 expands on chapter 2 by presenting the related existing research. Chapter 4 then fully describes the dynamic auto-sizing method as well as the underlying weighted l_1 regularization technique. Afterwards, chapter 5 describes both existing and novel anytime learning algorithms, and proposes approaches to integrate the algorithms with dynamic auto-sizing. Chapter 6 then experimentally evaluates the dynamic auto-sizing technique as well as its proposed applications in anytime learning. Finally, chapters 7 and 8 discuss the results and conclude the thesis.

Chapter 2

Background

2.1 Anytime Algorithms

Anytime algorithms are defined by [7] as algorithms whose quality of solutions gradually improve with computation time. Anytime algorithms can in general be interrupted at any point and still return a valid solution to the problem, as opposed to traditional algorithms that, if interrupted before completion, do not produce any useful output. As such, anytime algorithms are useful in problem domains in which computing optimal or exact solutions is not computationally or economically feasible, as they allow to trade off the quality of solutions with execution time or computational cost.

The term anytime algorithms was first used by Thomas Dean and Mark Boddy in a study on time-dependent planning in robotics [8]. Anytime algorithms have since been successfully applied to various fields such as evaluation of Bayesian networks [9], constraint satisfaction problems [10], heuristic state-space search [11], and path planning [12].

2.1.1 Properties of Anytime Algorithms

Anytime algorithms can be characterized by several typical properties, which have been gathered by [7]. The remainder of this section lists and informally defines these properties; it should be noted that these properties are merely desirable features and a concrete algorithm does not have to comply with all of them to be characterized as anytime.

Measurable quality of results. The solutions returned by anytime algorithms are approximate in general, and establishment of a suitable metric is crucial to the design of such algorithms. This metric can be as simple as the value of objective function in optimization problems.

Recognizable quality of results. With a chosen metric, it is desirable to have the ability to determine the quality of approximate results at run-

time. In practice, this would be difficult for example with a metric defined as distance between the approximate and correct result.

Monotonicity. The quality of solutions should be a non-decreasing function of elapsed time. When the quality of results is recognizable, this is trivial to achieve as the algorithm can simply store the best result.

Interruptibility. Interruptible algorithms can be stopped at virtually any time and still return an admissible solution. Algorithms without this property are called *contract*. Contract algorithms receive the amount of allocated resources as a parameter, and are not guaranteed to yield any useful results if interrupted sooner than was promised according to the allocation [13].

Preemptability. Preemptible algorithms can be suspended and resumed at will with negligible computational overhead.

Diminishing returns. It is common for anytime algorithms to improve rapidly at first, but with the improvements diminishing over time.

Consistency. The quality of results is often correlated with computation time (and possibly also with the quality of inputs). This is usually desirable as it can allow to predict the quality of results.

2.1.2 Performance Measurement

For many applications of anytime algorithms, it is crucial to define a quality measure to monitor the progress of the algorithm so that computation resources can be allocated effectively. As long computational time will typically reduce the overall utility in the vast majority of problem domains, the decision of at which point to interrupt the anytime algorithm and act on the current solution must often be made in practice. In autonomous systems, this is known as the meta-level control problem. [14]

2.1.2.1 Metrics

As anytime algorithms vary greatly in the ways they approach the exact results, no universal metric of quality of solutions exists. Three common anytime algorithm metrics have been listed by [7] as useful in different scenarios. Arguably the most commonly used metric is **accuracy**, which measures the distance between the approximate and exact solution. However in some settings, the anytime algorithm always produces the optimal or exact solution, only with the level of detail increasing with time. The **specificity** metric is useful in such circumstances, as it measures the level of detail of the result. Finally with certain anytime algorithms, the metric of **certainty** is utilized, which measures the degree of certainty (such as probability or fuzzy set membership) that the result is correct.

Quality measurement can be problematic in some cases. For example, if the function that evaluates quality is computationally costly or unavailable

during algorithm execution, the algorithm may need to use an approximation function instead. This introduces uncertainty into the monitoring process and can even lead to the algorithm replacing an existing solution with a solution of worse quality. [15]

2.1.2.2 Performance Traces

A *performance trace* describes a single execution of an anytime algorithm on a particular instance. For a particular run r of an anytime algorithm, the performance trace is defined by [15] as a function $K_r : T \to Q$ that maps t, the execution time, to q, the quality of result that would be returned if the algorithm was interrupted at that time. In most cases, performance traces can be made non-decreasing simply by adjusting the algorithm so that it stores the best result, with the exception of problematic quality metrics described in section 2.1.2.1.

2.1.2.3 Performance Profiles

As anytime algorithms are often stochastic in nature, it is important to study not only individual runs, but to also characterize the behavior of the algorithm over different runs. One option is to create a *quality map*, which simply plots performance traces of multiple runs of the algorithm. An alternative way is to use **performance profiles**, which have been defined by [7] as functions that map execution time of an algorithm to the expected output quality. An alternative definition is given by [15], which defines the performance profile of an algorithm over a set of runs R as a function

$$P_R(t, q) = \frac{1}{|R|} \sum_{r \in R} [K_r(t) \geq q] \tag{2.1}$$

that denotes the frequency of a run of the algorithm returning solution of quality at least q if interrupted at time t, where "[]" is the Iverson bracket. Performance profiles can thus be used to measure the overall trade-off between execution time and quality of solutions of an anytime algorithm.

2.1.3 Anytime Learning

The term *anytime learning* was coined in 1992 by John Grefenstette [16]. Originally it denoted a concrete approach for continuous learning in changing environments, which utilized two integrated running modules - an execution system, which controls the interactions of the agent with the external environment, and a learning system, which provides the execution system with a knowledge base by continuously testing new strategies against a simulated model of the environment. However, today the term anytime learning is much more general and refers to machine learning algorithms that share characteristics with anytime algorithms [17, 13], usually the ones that learning can be

interrupted at any time while still producing an answer, that the returned answers improve over time, and that the learning process can be suspended and resumed at will with little overhead [17].

It is not immediately clear whether gradient-based optimization methods, which are common for solving machine learning problems, can be regarded as anytime. They can be regarded as contract anytime algorithms in the sense that they always produce a non-trivial answer (that is, not merely a randomly initialized model) if allowed to run at least for a single epoch. Moreover with many implementations, the training can be interrupted and resumed at will without significant computational overhead. However, the individual runs will usually quickly stop improving because of overfitting, and more advanced techniques are thus necessary for truly anytime learning that gives the potential to improve indefinitely.

2.2 Deep Learning

Deep learning is a broad subfield of machine learning which constructs models from complex algebraic components with tunable parameters. Such components are typically organized into many layers so that the computation paths from input to output of the model are "deep". [18, p. 801] The origins of the field lie in deep **artificial neural networks**, more specifically deep multi-layer perceptrons, which are structured as a sequence of mutiple layers. However, the common contemporary definitions of the field usually do not refer to artificial neural networks but rather define deep learning as the study of methods that use multiple layers to automatically extract higher-level features from raw input data. [19, p. 264] Deep learning has become a popular field of research especially in recent years.

Artificial neural networks, on which much of the field of deep learning is based upon, are a class of machine learning models that emerged as an attempt to imitate the functionality of biological neural networks present in animal or human brains, although artificial neural networks have gradually deviated from their biological counterparts as the field matured. It is widely considered that the first models that took inspiration from computation in biological brains were introduced by Warren McCulloch and Walter Pitts in the year of 1943 [20], followed later by Frank Rosenblatt's **perceptron** algorithm from 1957 that can be roughly described as training a single artificial neuron. [21]. Later, models known as multilayer perceptrons were proposed that organize multiple perceptrons into layers, and have become popular especially with the introduction of the **backpropagation** training algorithm [22] in 1986.

2.2.1 Multilayer Perceptrons

Multilayer perceptrons, often presented as fully-connected feedforward artificial neural networks, are composed of multiple **units** (often called *artificial*

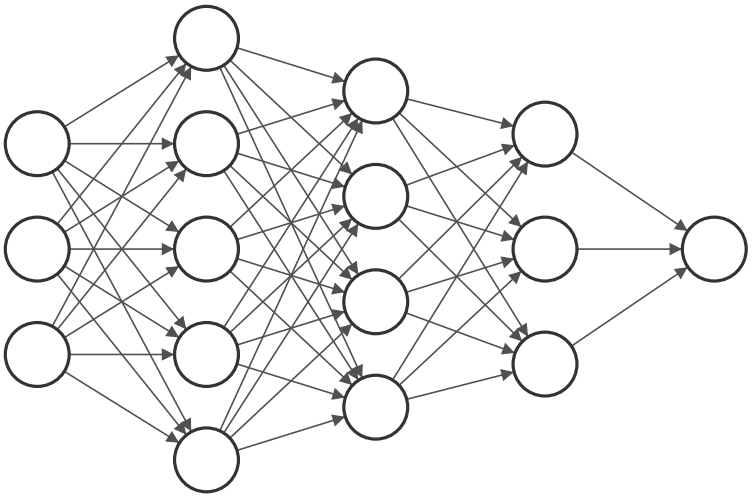

Figure 2.1: Diagram of a multilayer perceptron with three units in the input layer, five, four and three units in the following hidden layers, and a single output unit. Generated using the *NN-SVG* tool [23].

neurons) which are arranged into multiple layers. The structure of a multilayer perceptron is described in figure 2.1. The first layer of a multilayer perceptron is called the input layer, the last layer is called the output layer, and the intermediate layers are called hidden layers. [24, p. 289] Every unit in each hidden layer is connected to all units in the previous and following hidden layer by a connection that is is represented by a parameter called *weight*. The units in the input layer do not perform any computation but merely represent the input data; on the other hand, the output units do perform computations, and their outputs represent the output of the whole model.

The role of each unit is to calculate the weighted sum of inputs from units in the preceding layer, and apply a nonlinear *activation* function to this weighted sum to produce an output that is passed to units in the subsequent layer. Every neuron also includes an additional parameter called *bias*, which is added to the weighted sum before the activation function is applied; however, it is common to represent the bias terms of all units in a layer as weights of connections from a special "dummy" neuron in the previous layer, whose output is fixed to the value 1. With this abstraction, the function of a single unit can be described by the formula

$$y = \varphi(\sum_i w_i x_i) = \varphi(\boldsymbol{w}^\top \boldsymbol{x}), \tag{2.2}$$

where x_i the output of the i-th neuron in the preceding layer, w_i is the weight of the connection from the i-th neuron in the preceding layer, φ is the activation

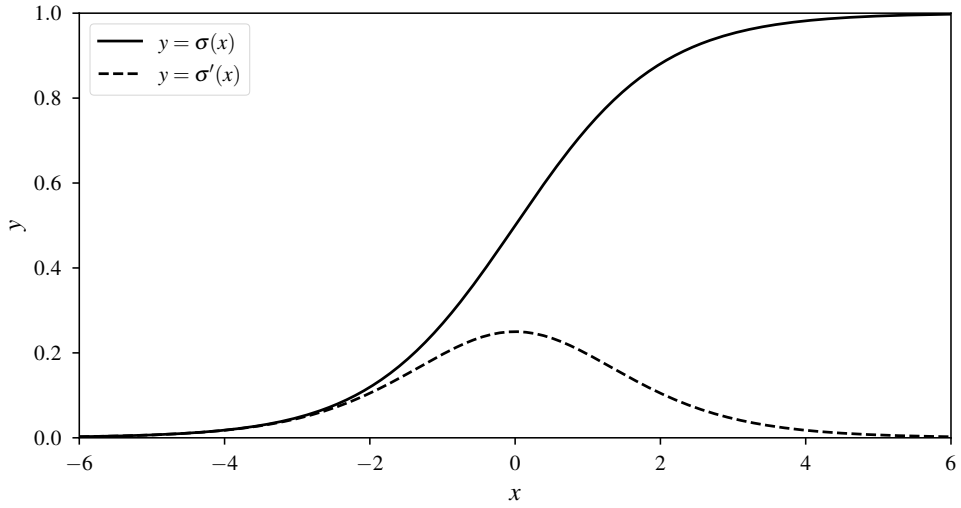

Figure 2.2: The sigmoid function and its derivative. For values of x farther from zero, the derivative becomes nearly zero, leading to difficulties with gradient-based optimization techniques.

function, and y is the unit's output[1] [18, p. 803].

2.2.2 Activation Functions

The role of a nonlinear activation function is important both in multilayer perceptrons and in most deep learning models in general. If this function was simply linear, then multilayer perceptrons would be limited to representing linear functions, no matter how the units would be arranged into layers [18, p. 803]. On the other hand, multilayer perceptrons with at least a single hidden layer and a finite number of units can approximate any continuous function with arbitrary precision [25, 26], provided that a non-polynomial activation function is utilized [27, 28]. This is known as the *universal approximation theorem*.

The activation function used in the original "perceptron" artificial neuron was the *Heaviside step function*:

$$H(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{2.3}$$

However, use of this activation function makes training impossible using gradient descent or similar methods, as there is no useful gradient. Historically,

[1] When we explicitly include the bias term b in the formula, the output of a single neuron becomes $\varphi(\sum_i w_i x_i + b)$.

more popular activation functions in multilayer perceptrons included the **sigmoid** function (also known as logistic function), defined as $\sigma(x) = 1/(1+e^{-x})$, and the **hyperbolic tangent** function, defined as $tanh(x) = (e^{2x}-1)/(e^{2x}+1)$ [18, p. 804]. However, these functions still suffer from *saturation*, meaning that they contain flat regions in which these functions are insensitive to small changes to the input [29, p. 68]. To elaborate, the sigmoid and tanh function are only strongly sensitive to changes in input when x is near 0. With large or small values of x, the derivative of these functions becomes near zero, leading to the *problem of vanishing gradients* [29, p. 290].

2.2.2.1 ReLU Activation

The *rectified linear unit* or **ReLU** is arguably the most widely used activation function in deep learning models, and is at the same time usually recommended for use in most applications [29, p. 174]. This simple activation function is defined as $ReLU(x) = max(0, x)$ [18, p. 803], therefore this function is piecewise linear with two linear pieces and as such it is advantageous for use with gradient-based optimization methods; however, the resulting transformation is still non-linear. Morevoer, the ReLU activation function is non-saturating for positive values of x as the corresponding partial derivative for this region is 1. [29, p. 175]

Unfortunately, as the gradient of the ReLU function for negative values of x is 0, units that use this activation function may sometimes become stuck at outputting only zeros. This issue is known as the *dying ReLU problem* [24, p. 335], as such units do not contribute to the outputs of the model and also are unlikely to "come back to life" by returning to positive values. Remedies have been proposed that generalize the ReLU function to the form of $f(x) = max(0, x) + \alpha min(0, x)$, in which α is a parameter that controls the slope of the linear part for negative values of x. Examples of such activation functions include *absolute value rectification* that sets $\alpha = -1$, the *leaky ReLU* function that sets α to a small positive value, or the *parametric ReLU* function which treats α as a learnable parameter. [29, p. 193]

2.2.2.2 ELU Activation

The *exponential linear unit* or simply **ELU** is another popular activation function, which similarly to the ReLU function can be defined piecewise [30]:

$$ELU(x) = \begin{cases} x & x > 0 \\ \alpha(e^x - 1) & x \leq 0 \end{cases} \tag{2.4}$$

The parameter α controls the amount of saturation for negative values of x, and when it is set to the usual value of 1, the function is smooth even around 0. For negative inputs, the outputs are also negative, which leads to the average output of units that utilize ELU activation being closer zero. These

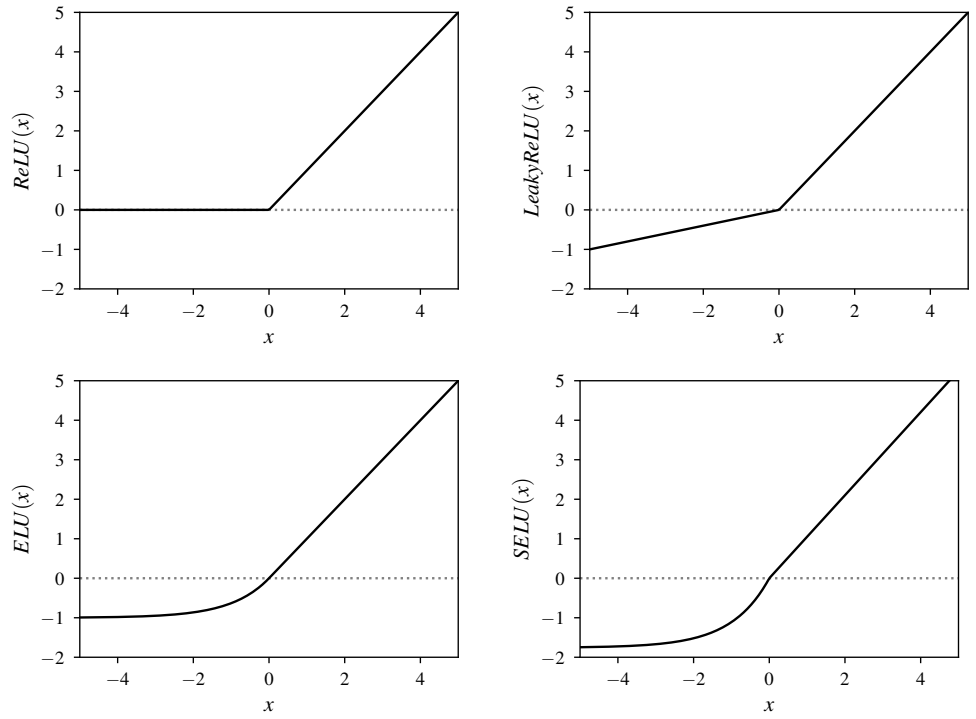

Figure 2.3: Graphs of various activation functions. Note that only *ELU* is smooth at $x = 0$.

properties are beneficial with gradient-based optimization methods. Moreover, the derivative is non-zero even for negative values of x, mitigating the problem of dying units. [24, p. 337] All in all, it has been shown that ELU activation typically outperforms ReLU and its common variants in both training time and predictive performance of models [30].

2.2.2.3 SELU Activation

The *scaled exponential linear unit*, or **SELU**, is a scaled variant of the ELU activation function. This activation function is defined by the formula

$$SELU(x) = \lambda \begin{cases} x & x > 0 \\ \alpha(e^x - 1) & x \leq 0, \end{cases} \tag{2.5}$$

with λ and α having fixed predefined values rather than being parameters[2] [31]. When used exclusively as the activation function of all units in a model, SELUs have the important property of inducing *self-normalization*, which

[2]For the SELU activation function, the values of λ and α are defined as approximately 1.0507 and 1.6733, respectively.

means that during training, the outputs of every layer will be close to mean of 0 and standard deviation of 1, thus strongly mitigating the problem of vanishing gradients. This has been demonstrated to hold true with sequential models when the input features of the model are standardized and the weights of every layer are initialized using LeCun normal initialization. [24, p. 337]

2.2.2.4 Softmax Activation

The softmax activation function is typically used at the output layer of classification models to represent a multinoulli probability distribution over a categorical variable. In most applications, softmax activation is used together with cross-entropy loss. The function is defined as

$$softmax(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}}, \tag{2.6}$$

where $\boldsymbol{z}$ is a vector of log probabilities produced by a linear layer, that is, $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$ [29, p. 185]. The outputs represent a valid probability distribution, as the value of each element is between 0 and 1 and all the values sum to 1. In theory, a similar normalization could be achieved by dividing each input by the sum of all inputs, however the exponential function roughly cancels the logarithm in the cross-entropy loss [29, p. 185]. This leads to a roughly constant gradient with incorrectly classified instances, that is, instances where the true class has not obtained the highest output probability.

2.2.3 Layers

Deep learning models are composed of components called layers that output mathematical transformations of their inputs. Numerous layer types of varying complexity exist, ranging from simple *fully-connected* layers used in multilayer perceptrons to the more complex *convolutional* and *attention* layers, with different layer types being suitable for different use-cases. All of the mentioned layer types are examples of **feedforward** layers, that is layers in which information flows only in the direction from input to output; nonetheless, **recurrent** layers also exist, which feed their outputs back into their own inputs so that a single data instance is processed for multiple time steps. However, this section will be limited to the study of fully-connected and convolutional layers.

2.2.3.1 Fully-Connected Layers

Fully-connected or *dense* layers are layers known from multilayer perceptrons and as such have already been described in section 2.2.1. To recapitulate, a fully-connected layer is composed of multiple units known as artificial neurons, with every unit being connected to all of the layer's inputs. Instead of modeling

each neuron individually, today's software implementations of deep learning models usually represent the whole fully-connected layer by a *weight matrix*, which is composed of connection weights belonging to all of the layer's units, and usually by an addidional vector of all of the units' biases. Mathematically, the transformation of a fully-connected layer can then be described by the formula

$$\boldsymbol{y} = g(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}), \tag{2.7}$$

where $\boldsymbol{W}$ is the weight matrix with $\boldsymbol{W}_{i,j}$ being the weight of connection from input j to output i, $\boldsymbol{b}$ is the vector of biases, $\boldsymbol{x}$ is a vector of inputs, and $\boldsymbol{y}$ is a vector of outputs [24, p. 286]. The weight matrix notation of $\boldsymbol{W}_{i,j}$ representing the connection from input i to output j yields the slightly altered formula

$$\boldsymbol{y}^\top = g(\boldsymbol{x}^\top \boldsymbol{W} + \boldsymbol{b}^\top), \tag{2.8}$$

which is useful when describing operations on batches of data and will be used in the later parts of the thesis.

Fully-connected layers are very common in deep learning models. This is partly because of multilayer perceptrons still being popular all-purpose machine learning models, however fully-connected layers are frequently used even in more complicated deep learning architectures as separate building blocks.

2.2.3.2 Convolutional Layers

Convolutional layers were originally inspired by neuroscientific study of the visual cortex in biological brains, and as such are mostly used in image data processing. However, convolutional layers are not limited only to applications in computer vision, as they have successfully been applied to tasks such as voice recognition and natural language processing. [24, p. 445] As opposed to fully-connected layers, convolutional layers are designed for parameter efficiency, as each hidden unit is connected only to a small local region from the previous layer, thus respecting adjacency in image data. Moreover, *spatial invariance* of image data is exploited by the weights of the hidden units of a layer being shared for all local regions. [18, p. 811].

A pattern of connection weights that is replicated across all local regions is called *kernel* or *filter*. Each convolutional layer typically consists of many such kernels. Both the input and output of a convolutional layer are multidimensional arrays which are usually referred to as tensors, although the terminology is problematic from a mathematical perspective. [18, p. 814] With image data, tensors are typically three-dimensional[3]. For example, a three-channel RGB image with width of 256 pixels and height of 128 pixels would typically form a tensor of size $256 \times 128 \times 3$, where the three *channels* are in the context of convolutional layers usually known as *feature maps*. Each

[3]Four-dimensional if we consider batches of data.

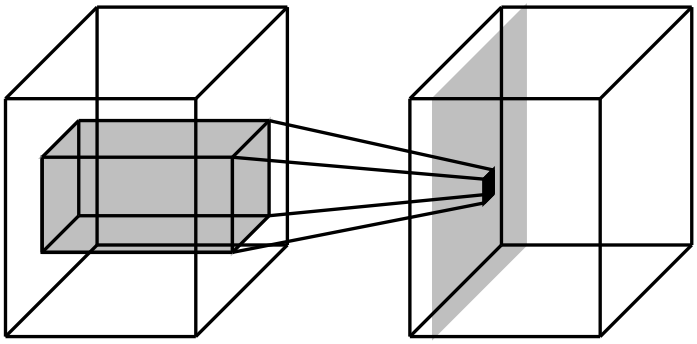

Figure 2.4: Visualization of the operation performed by a convolutional layer. Each of the two blocks is a three-dimensional tensor, composed of multiple two-dimensional feature maps. Each value of a single feature map, as indicated by the black square, is calculated as the weighted sum of the values in the corresponding local regions of all feature maps from the previous layer (on the left), where each feature map is weighted by the values in the corresponding filter.

convolutional layer produces a tensor with the number of feature maps equal to the number of the layer's kernels.

Convolutional layers are built around **discrete convolution**, which is a mathematical operation on two functions defined by [29] as

$$(f * g)(n) = \sum_{m=-\infty}^{\infty} f(m)g(n-m). \tag{2.9}$$

When we denote a two-dimensional feature map as X and a two-dimensional kernel as K, and assume that the functions are zero for elements outside of the feature map and kernel, the convolution operation can be written as

$$(X * K)(i,j) = \sum_{m}\sum_{n} X(m,n)K(i-m, j-n), \tag{2.10}$$

where the summation is over all values of m and n for which the indexing is valid [29, p. 332]. In this notation, it is said that the kernel is "flipped", as when a summation index increases, the index into the feature map is increased but the index into the kernel is decreased [29, p. 332]. It is common to omit the flipping of the kernel in software implementations of convolutional layers, however the resulting mathematical operation is then formally called *cross-correlation* and not convolution [29, p. 333]. Moreover, it is common to include a bias term, similarly to fully-connected layers. Bias can be *tied*, with one bias per kernel, and *untied*, where one bias term is specific not only to a single kernel but to a single output location in the feature map as well.

Because the convolution operation produces a feature map that is smaller than the input, the input feature maps are commonly *padded* with zeros so that the outputs match their original dimensions. In case we wish to decrease the size of feature maps produced by a convolutional layer, some of the elements of the produced feature maps can be omitted by using "strides", which specify how many positions of the kernel should be skipped in each dimension of the feature map [29, p. 348]. An alternative to using strides is to include *pooling layers*, which aggregate sets of adjacent units from the previous convolutional layer into single values, thus also decreasing the size of the feature maps [18, p. 811].

The whole transformation defined by a standard convolutional layer can be described by the formula

$$Y_{j,k,l} = \sum_{i,m,n} X_{i,k+m-1,l+n-1} K_{i,j,m,n}, \tag{2.11}$$

where $X_{i,k,l}$ is an input value within feature map i, row k and column l; $Y_{j,k,l}$ is an output value with equivalent indexing; $K_{i,j,m,n}$ is the element of the kernel tensor $\boldsymbol{K}$ defining the weight of connection between input unit in feature map i with offset of m rows and n columns, and the output unit in feature map j; and the summation is over all values of i, m, and n for which the tensor indexing is valid [29, p. 348]. Note that the kernel is not flipped in this notation, and bias is omitted.

Convolutional neural networks are deep learning models that include convolutional layers. Traditional convolutional neural network architectures such as LeNet-5 [32], AlexNet, [33] or VGGNet [34] are usually formed by several convolutional and pooling layers, which are followed by a small number of fully-connected layers. During training, the first convolutional layers then typically learn filters that detect simple features such as edges or textures, while the later layers tend to learn filters that detect high-level features such as human faces [35]. Several important enhancements to these traditional convolutional architectures have been proposed, such as inception networks [36], residual networks [37], and fully convolutional networks [38].

2.2.4 Loss Functions

Loss functions are functions that machine learning models minimize during training. According to [29], the terms loss function, cost function, objective function and error function may often be used interchangeably, with the convention that objective functions can be both minimized or maximized, according to the chosen form.

A loss function typically includes the ground truth data and the corresponding predictions of a model as inputs, and outputs a single number as a measure of the error. Nonetheless, other forms of losses exist, as for example a total cost function for training a model often combines a loss function with

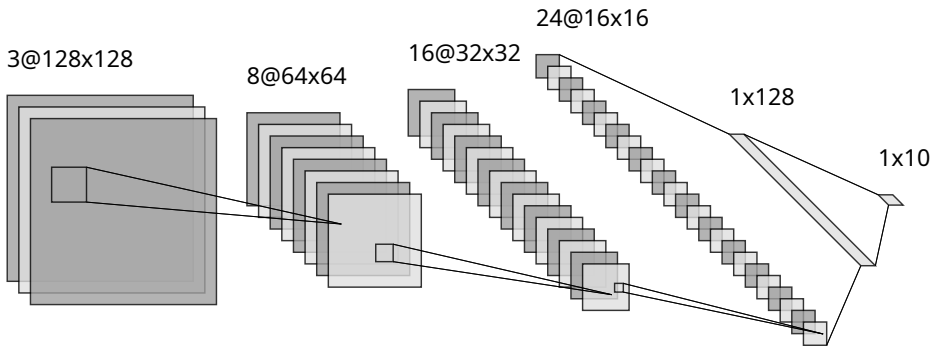

Figure 2.5: Diagram of a convolutional neural network composed of three convolutional and two fully-connected layers. The input is a three channel 128×128 image. The convolutional layers are then composed of a growing number of filters, and utilize 2-by-2 strides, which progressively halve the sizes of feature maps produced by the layers. The following fully-connected hidden layer then *flattens* the feature maps into a vector, and the 10-unit output layer can be perceived as returning the probabilities of the input image belonging to each of 10 classes. Generated using the *NN-SVG* tool [23].

a *regularization* term that includes the model's parameters as inputs. [29, p. 178] In this section, two basic loss functions will be discussed: **mean squared error**, which is popular for use in regression tasks, and **cross-entropy**, which is commonly utilized in classification tasks.

2.2.4.1 Mean Squared Error

The mean squared error is arguably the most popular loss function for training models on regression problems. The mean squared error measures the average of the squares of the errors between predictions of the model and the ground truth. The measure is given by

$$MSE(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})^2, \tag{2.12}$$

[29, p. 134]with $y^{(i)}$ being the target value of instance i, $\hat{y}_k^{(i)}$ being the corresponding prediction of a model, and m being the number of instances.

Closely related to mean squared error are other popular losses for use in regression settings, such as the *root mean squared error*, which measures error in units of the quantity being estimated, and *mean absolute error*, which also uses the same scale as the estimated quantity, but additionally is less sensitive to outliers [24, p. 41].

2.2.4.2 Cross-Entropy

In statistics, it is common to measure how much two probability distributions are different from each other using the **Kullback-Leibler (KL) divergence**, which is defined as

$$D_{KL}(P||Q) = \boldsymbol{E}_{x\sim P}[\log \frac{P(x)}{Q(x)}] = \boldsymbol{E}_{x\sim P}[\log P(x) - \log Q(x)]. \tag{2.13}$$

The KL divergence is non-negative, being 0 if and only if P and Q represent the same probability distribution in case of discrete random variables[4]. [29, p. 75] However, as minimizing the KL divergence with respect to Q does not depend on the $\log P(x)$ term, it can be omitted for such tasks, leading to the related statistical measure of **cross-entropy** [29, p. 75] [18, p. 809]:

$$H(P, Q) = -\boldsymbol{E}_{x\sim P} \log Q(x) = -\int P(x) \log Q(x) dx. \tag{2.14}$$

Therefore, minimizing cross entropy is equivalent to minimizing KL divergence between an empirical distribution defined by the training set and the distribution defined by the model. [29, p. 132] In a machine learning task, cross-entropy can be used as the loss function for both binary and multiclass classification tasks, as it can be perceived to measure how well a set of estimated class probabilities matches the ground truth. The concept is more clear when the equation for cross-entropy is written in an alternative notation from the perspective of a classification task,

$$H(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} \log(\hat{y}_k^{(i)}), \tag{2.15}$$

where $y_k^{(i)}$ is the target probability that instance i belongs to class k, $\hat{y}_k^{(i)}$ is the respective probability predicted by a model, m is the number of instances, and K is the number of classes. [24, p. 149] It is worth noting that in most machine learning tasks, the target probability is binary with value always at 0 or 1, as training datasets typically consist of instances whose labels always correspond to a single target class.

2.2.5 Optimization Techniques

The topic of optimization of deep learning models is very broad and, although crucial for successful applications, not directly related to this thesis. For completeness, this section briefly describes two important optimization techniques: the backpropagation algorithm, and the Adam optimizer.

[4] A similar property holds for continuous random variables.

2.2.5.1 Backpropagation

Backpropagation was introduced by David Rumelhart et al. in a famous 1986 paper [22], which revolutionized the way in which artificial neural networks were trained. The algorithm can be described as gradient descent, with the gradients computed by two passes through the network. The instances from the training set are fed to the algorithm in groups called *mini-batches*, with the whole training set iterated over multiple times in so-called *epochs*. Each mini-batch of the training set instances is first passed to the input layer of the model, with the computed result being successively sent to the following layers in the so-called *forward pass*. The error of the output layer is then calculated using the cost function, and the contribution of each of the preceding layers is consecutively measured in the *backward pass*, which calculates the partial derivatives with respect to the individual parameters using the chain rule. The parameters of the model are then updated in a gradient descent step. [24, p. 289]

2.2.5.2 Adam Optimization

As the cost functions with respect to parameters of deep learning models often contain numerous local optima and plateaus, use of plain gradient descent is usually not sufficient in practice. The Adam optimization algorithm, which stands for *adaptive moment optimization*, combines the ideas of two other optimization techniques, namely *momentum optimization* [39] and *RMSProp*. Momentum optimization does not update the model parameters using the computed gradient directly, but instead uses the gradient to update *momentum*, which aggregates the gradients from the previous steps and is then used to update the model parameters in the usual manner. This approach gives the ability to escape some local optima and overcome plateaus faster. On the other hand, RMSProp builds upon the *AdaGrad* algorithm [40] and uses so-called *adaptive learning rate* to more effectively navigate the optimizer to the optimum. These two approaches make Adam a powerful all-purpose optimizer. [24, p. 356]

2.3 Regularization

Complex machine learning models can have thousands, millions or even billions of parameters, which gives them the ability to formulate very complicated hypotheses about the world. Large numbers of parameters can allow such models to represent very complex functions that can fit complicated datasets - as John von Neumann used to say:

> "With four parameters I can fit an elephant, and with five I can make him wiggle his trunk". [41]

However, models with a lot freedom can easily overfit the training dataset by only performing well on the data seen during training; it can therefore be desirable to explicitly penalize complex hypotheses. Such a process is called regularization, with the name derived from the usual goal of obtaining functions that are more regular. [18, p. 689]

The *No Free Lunch* theorem implies that machine learning algorithms must be designed for specific tasks, as no single algorithm will be perfect for all optimization problems. Therefore every machine learning algorithm is built with an underlying set of preferences, and only performs well when these preferences are aligned with the specific task to be solved. A certain regularity of the function represented by the obtained model is often among such preferences. In general, regularization techniques are designed to reduce error on the test dataset, possibly while increasing the error on the training dataset. It can be said that regularization works by trading increased bias of a machine learning algorithm for reduced variance, with this method only being effective when variance is reduced significantly while the bias is kept at a satisfactory amount. [29, p. 229]

2.3.1 Parameter Norm Penalties

A typical approach to regularization is based on limiting the capacity of models by searching for a hypothesis that minimizes an objective function with the added penalty term that represents the complexity of the hypothesis. If we denote the original objective function by J and the regularization term Ω, the formula for the regularized objective function $\tilde{J}$ is

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta}), \tag{2.16}$$

[29, p. 230] where $\boldsymbol{\theta}$ are the parameters of the model and α is a hyperparameter controlling the emphasis on the regularization term Ω in contrast to the rest of the original objective function J. In general, parameter norm penalties are designed to penalize the model for large values of its parameters, as such parameters would usually contribute to more complicated, non-regular functions.

2.3.1.1 l_2 Regularization

l_2 regularization, also known as Tikhonov regularization or weight decay, functions by pushing the parameters of a model towards zero. The formula for the l_2 regularization term is

$$\Omega_{l_2}(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|_2^2 = \boldsymbol{\theta}^\top \boldsymbol{\theta} = \sum_i \theta_i^2, \tag{2.17}$$

[29, p. 231] where $\boldsymbol{\theta}$ is a vector containing all of the model's parameters. Therefore the objective function of a model regularized with l_2 regularization

becomes

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha \boldsymbol{\theta}^\top \boldsymbol{\theta}, \tag{2.18}$$

with the corresponding gradient being

$$\nabla_{\boldsymbol{\theta}} \tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + 2\alpha \boldsymbol{\theta}, \tag{2.19}$$

[5]

which causes the learning of the model to be modified so that during each step, every parameter is reduced ("decayed") towards zero proportionally to its value. This means that parameters with large values are pushed towards zero much more than parameters that are already close to zero.

2.3.1.2 l_1 Regularization

Similarly to l_2 regularization, l_1 regularization (sometimes referred to as *lasso*) also pushes the parameters of the regularized model towards zero. The formula for the l_1 regularization term is

$$\Omega_{l_1}(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|_1 = \sum_i |\theta_i|, \tag{2.20}$$

[29, p. 234] therefore the model's objective function becomes

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha \|\boldsymbol{\theta}\|_1, \tag{2.21}$$

with the corresponding gradient

$$\nabla_{\boldsymbol{\theta}} \tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha sgn(\boldsymbol{\theta}). \tag{2.22}$$

By analyzing the gradient term, we can see that compared to l_2 regularization, the parameters are pushed towards zero by an additive constant, independently of the size of the parameter's value.

Adding an l_1 regularizer to a model together with sufficient regularization strength α causes the important effect of inducing sparsity in the parameters of the model. More specifically, l_1 regularization in general leads to more sparse solutions than l_2 regularization. Sparsity in this context means that a number of parameters has an optimal value of zero. [29, p. 236] This can be explained by the observation that in the l_1 regularization cost function, all parameters contribute linearly, even ones with values that are close to zero, as can be seen in figure 2.6. On the other hand, as the parameters approach zero, the corresponding limits of the partial derivatives of the l_2 regularization loss function are equal to zero, and the parameters thus typically end up with values further from zero than when using l_1 regularization.

In many types of models, such as linear regression, logistic regression, or basic types of artificial neural networks, parameters with a value of zero do

[5] Sometimes the l_2 regularization term is defined as $\frac{1}{2}\|\boldsymbol{\theta}\|_2^2$ so that the corresponding gradient simply becomes $\alpha\boldsymbol{\theta}$.

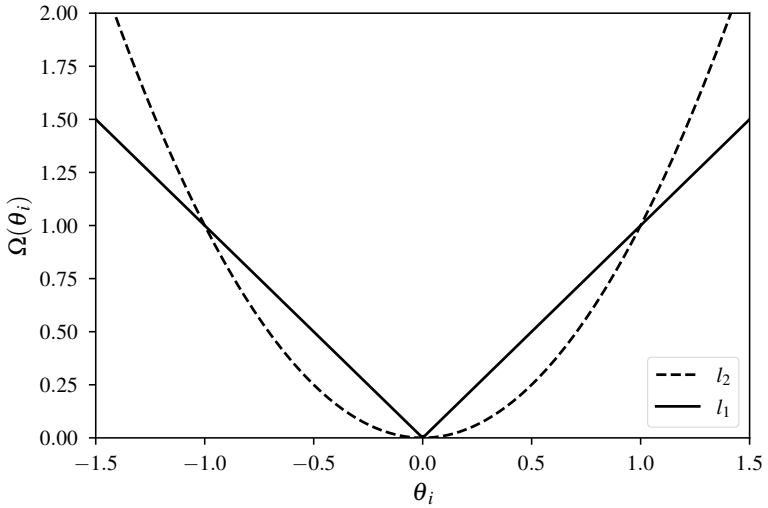

Figure 2.6: Loss functions of l_2 and l_1 regularization when simplified to a single parameter. As opposed to l_1 regularization, l_2 regularization leads to negligible loss with parameters that are already close to zero, which thus usually end up with non-zero values.

not contribute to the outputs of the model and can thus be safely omitted. This implies that l_1 regularization can often be utilized as a feature selection technique, as parameters that are unimportant for the task being solved will typically be set to values close to zero by most optimization methods.

2.3.1.3 Structured Sparsity Regularization

Structured sparsity regularizers are a generalization of standard sparsity inducing regularizers such as the l_1 norm, in which structures such as groups or networks are defined on the model parameters or, in some cases, on the input features. [43] One example for use with structures in form of non-overlapping groups is *group lasso* [42], in which every group of parameters is regularized using the unsquared l_2 norm, which is sparsity-inducing as opposed to the squared l_2 norm used in l_2 regularization. In principle, this can lead to only some groups of parameters being "selected" as the other groups are zeroed out.

The group lasso technique can be perceived as application of the regularizer defined as the $l_{2,1}$ matrix norm. For a parameter matrix $\mathbf{\Theta}$ of a fully-connected neural network layer in which each column forms a group, the corresponding regularization term can be written as

$$\Omega_{l_{2,1}}(\mathbf{\Theta}) = \sum_j \|\mathbf{\Theta}_{:j}\|_2 = \sum_j \left(\sum_i \Theta_{ij}^2 \right)^{\frac{1}{2}}, \tag{2.23}$$

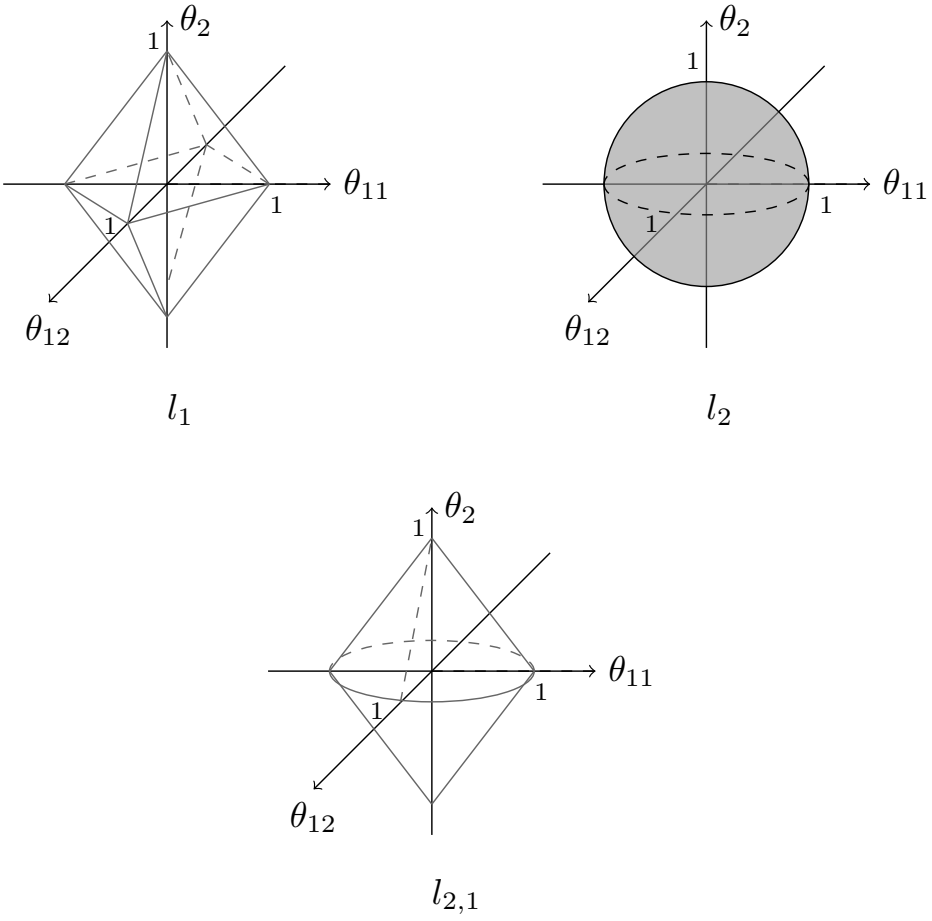

Figure 2.7: Comparison of the l_1, l_2, and $l_{2,1}$ (group lasso) penalties. Each figure depicts the surface at which the corresponding penalty is equal to 1. In case of the $l_{2,1}$ penalty, the parameters θ_{11} and θ_{12} form a group, and if both parameters are set to 1, the resulting penalty is lower than if each parameter formed a separate group. Created with inspiration from [42].

as demonstrated by [1]. The group sparsity effect of $l_{2,1}$ norm is described in figure 2.7. Similarly, the $l_{\infty,1}$ norm, defined as

$$\Omega_{l_{\infty,1}}(\boldsymbol{\Theta}) = \sum_j \|\boldsymbol{\Theta}_{:j}\|_\infty = \sum_j \max_i |\Theta_{ij}|, \tag{2.24}$$

can also be used as a structured sparsity regularizer [1].

2.3.2 Dropout

Dropout [44] is a powerful artificial neural network regularization technique that introduces a hyperparameter p, called the *dropout rate*, which represents the probability for each neuron to be omitted from the computations in each training step. More specifically, for every training step[6], dropout randomly samples a binary mask that excludes a portion of units in input and hidden layers of the model from the forward propagation, back-propagation, and the learning update of this step. When making predictions after training is complete, all parameters are utilized without any masking. Typically the empirical *weight scaling inference rule* is utilized to compensate for the fact that the units suddenly sum up more input data than during training, by multiplying the weights of every unit by the dropout rate p.

Dropout shares certain similarities with *bagging*, which is an ensemble learning technique[7] which constructs multiple datasets by sampling from the original training set with replacement, and upon each sampled dataset trains an independent machine learning model. Similarly to bagging, dropout can also be perceived as a method that trains an ensemble of models, with the major difference the models sampled by dropout share parameters, whilst bagging works with independent models trained on their respective datasets. [29, p. 265]

An important advantage of dropout is the simplicity of the method, which results in both low computational costs and the benefit that dropout can be used with many types of layers and optimization techniques. It has also been shown that dropout is a more effective regularizer than traditional techniques such as weight decay, and that dropout may additionally be coupled with other regularizers to produce further improvements. [29, p. 265] Utilizing dropout during training often results in a significant boost in predictive performance of the trained models, making it a popular method in the deep learning community.

2.3.3 Early Stopping

Arguably the most popular form of regularization is the strategy known as *early stopping*. This simple strategy is built on the common observation in

[6] Dropout is designed for use with minibatch-based learning algorithms.

[7] Bagging can also be perceived as a regularization technique, as it attempts to reduce the test set error similarly to traditional regularizers.

training models with higher representational capacity that during training, at some point the model starts to overfit the training set, with the error on the validation set going up while the training error keeps decreasing. With early stopping, the validation error is thus measured after every training epoch and if it stops improving for a certain number of epochs, the model corresponding to the best epoch (regarding validation error) is returned as the result. Besides formally being a regularization technique, early stopping can also be perceived as a hyperparameter selection algorithm that chooses the optimal number of training steps. [29, p. 246]

2.4 Hyperparameter Tuning

A typical machine learning algorithm is parameterized by various variables controlling different aspects of the learning process. Such parameters, belonging not to individual models but to whole model classes, are known as *hyperparameters* and must be set prior to training, instead of being set by the optimization technique.

The typical aim is to select hyperparameters that lead to satisfactory performance, either on validation sets or in a cross-validation setting, while still keeping the time and memory cost of training at acceptable values. In artificial neural networks, possibly the most important hyperparameter is the learning rate, while most other important hyperparameters adjust the representational capacity of the model so that it suits the complexity of the task. These hyperparameters usually include the number of hidden layers and the corresponding numbers of units, the regularization strength α, and dropout rate. Hyperparameters can be even categorical, such as when controlling the type of activation function utilized in hidden units, or binary, such as indicators of the use of batch normalization or custom preprocessing steps.

Approaches to hyperparameter tuning can be classified as either manual or automatic. The advantage of automatic hyperparameter tuning methods is that they require less understanding of the function of the hyperparameters than manual tuning. On the other hand, automatic hyperparameter methods are often more computationally expensive. [29, p. 428]

2.4.1 Manual Hyperparameter Tuning

The simplest and possibly also most common hyperparameter tuning technique is manual tuning, which is also known as *hand-tuning*. Manual tuning is usually iterative - first some initial hyperparameter values are manually chosen according to past experience of the engineer, then tested by training the model and observing the performance on the validation set, and the results are then used to choose the next combination of hyperparameter values. [18, p. 428]

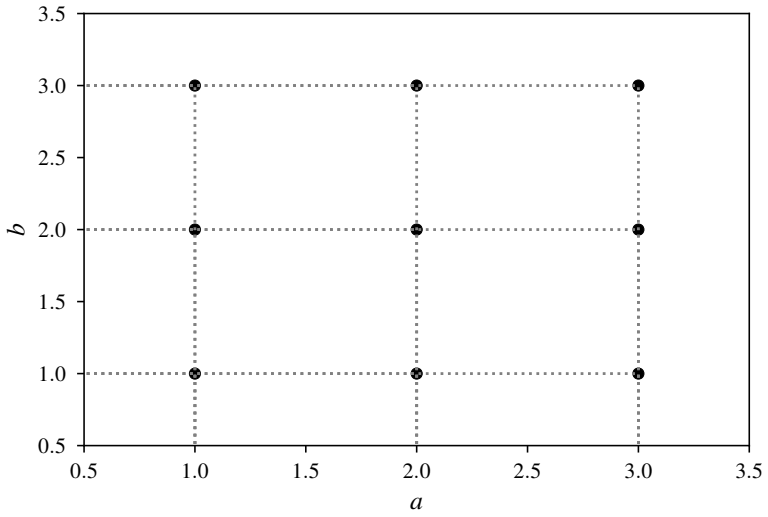

Figure 2.8: Visualization of a run of a typical configuration of grid search for two hyperparameters. Although 9 trainings were performed in total, only three unique values of each hyperparameter were tested.

2.4.2 Automatic Hyperparameter Tuning

2.4.2.1 Grid Search

Grid search is a systematic approach to hyperparameter tuning, suitable in cases when it is sufficient to consider only a few selected hyperparameters. For each hyperparameter, first a set of values to be explored is selected by the user, after which the grid search algorithm trains a model for every combination of hyperparameter values in the Cartesian product of the individual sets of hyperparameter values. The combination that yielded the best validation error is then returned as result. [29, p. 432]

Grid search is not practically usable in cases when there is a large number of hyperparameters and each hyperparameter can take a larger number of values. If we denote the number of hyperparameters as m and the maximum number of values that a single parameter can take as n, the number of requred training and evaluation steps grows as $O(n^m)$. [29, p. 434] Therefore, the time complexity grows exponentially with the number of parameters, which is a well known phenomenon known as the *curse of dimensionality*.

2.4.2.2 Random Search

A more powerful alternative to grid search is random search [45]. Similarly to grid search, random search tests every combination of hyperparameters by training a model and evaluating it on the validation dataset. However,

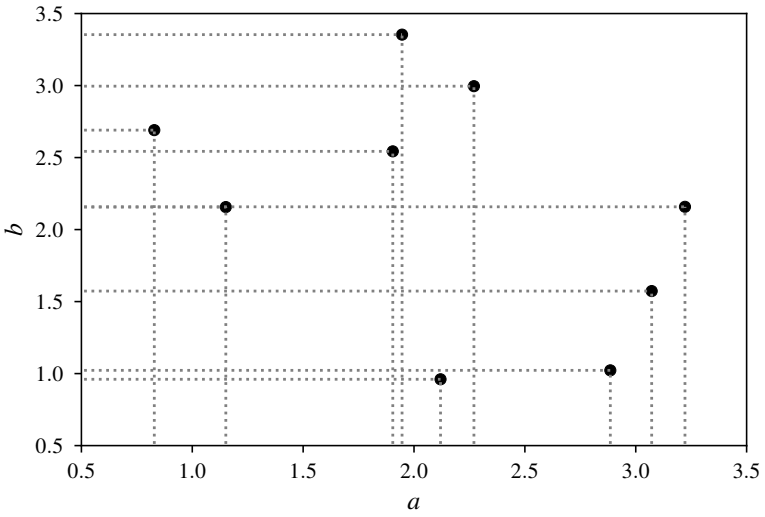

Figure 2.9: Visualization of a simulated run of random search for two hyperparameters, with values of each hyperparameter randomly sampled from a uniform distribution. In total, 9 trainings were performed, and 9 unique values of each hyperparameter were tested.

instead of choosing the combinations systematically from a Cartesian product of possible hyperparemeter values, random search operates stochastically by randomly sampling a unique combination of hyperparameter values for every iteration.

In random search, the values of every hyperparameter are sampled from a specified marginal distribution. Unlike with grid search, the values ideally should not be discretized, as sampling from a continuous disctrubtion allows to explore a larger set of values for every hyperparameter than with grid search at no additional cost. This property is important as in many machine learning problems, only some hyperparameters have a significant influence on the training. Therefore even in problems with a larger number of hyperparameters, merely a limited number of random search iterations will often be sufficient to obtain satistfactory values of the most important hyperparameters.

It has been demonstrated that random search often converges faster to satisfactory hyperparameter combinations than grid search. [45] The main reason behind this advantage is that as the number of non-influential parameters increases, the amount of computational resources wasted by grid search grows exponentially, while random search wastes less resources as new values of influential parameters are tested on nearly every trial. [29, p. 433]

2.5 Evolutionary Algorithms

Evolutionary algorithms are algorithms inspired by biological evolution, although the analogy is often loose. The scientific theory of evolution was developed independently by Alfred Russel Wallace in 1858 and Charles Darwin in 1859, with the central thesis being that characteristics of individuals in biological populations are preserved in proportion to their effect on reproductive fitness [18, p. 136]. From the perspective of computer science, evolutionary algorithms can be perceived as *local search* optimization algorithms in that they operate by searching locally between neighboring states of the state space, without storing the path from the initial state. More specifically, evolutionary algorithms can be perceived as a version of *local beam search* called *stochastic beam search*, which operates with multiple states at once and chooses successor states stochastically based on their values. [18, p. 133]

From a biological perspective, states in evolutionary algorithms can be seen as **individuals** of a **population**. Moreover, evolutionary algorithms introduce *operators* such as *selection*, *recombination*, and *mutation* that operate on encoded representations of the states[8]. Another common feature of evolutionary algorithms is that the individuals are scored by a **fitness function**, whose main role is to determine the quality of solutions represented by individuals. Evolutionary algorithms operate iteratively on **generations** of individuals, with the goal to produce individuals of progressively better quality. The common procedure is that first an initial population is created, then the individuals are recombined into new individuals, mutated, and afterwards a new generation is selected according to the individuals' fitnesses. An important concept is *selective pressure*, which can be roughly described as the probability of selection of the best individuals [46]. Selective pressure that is too strong can lead to degeneration of the population in the long term.

Concrete evolutionary algorithms differ in various aspects, such as in the representation of each individual, the recombination procedure, and the mutation and selection processes. [18, p. 134] This section will discuss two common classes of evolutionary algorithms, namely **genetic algorithms**, which encode individuals as binary strings, and **evolution strategies**, which encode individuals as real-valued vectors. Other popular branches of evolutionary algorithms include *genetic programming*, which evolves computer programs encoded into tree structures [47], and *evolutionary programming*, which evolves finite state machines [48].

2.5.1 Genetic Algorithms

Genetic algorithms are likely the most famous class of evolutionary algorithms, popularized largely by the work of John Holland in the 1970s [49] but with

[8] Recombination and mutation are examples of *variation operators*, which directly change the information encoded in the individuals.

efforts dating back as early as 1950s [50, 51, 52]. In genetic algorithms, the individuals are usually represented as binary strings, which are sometimes called *chromosomes* and the individual values are sometimes called *genes* [47]. On the other hand, variants of genetic algorithms exist which operate on permutations or vectors with values from discrete domains.

As for variation operators in genetic algorithms, the recombination operator is usually called **crossover** and typically operates on a pair of individuals, called *parents*, by combining their chromosomes into an *offspring* individual. For example, *single-point crossover* operates by randomly generating an index with value between 1 and the length of the chromosomes, and changing the parents' genes with positions in the chromosome greater than the index [47], thus producing two offspring. Mutation is performed by randomly flipping the value of each gene with probability p, which is called the *mutation rate*.

The usual choice of the selection operator in genetic algorithms is either *roulette wheel selection* or *tournament selection*. In roulette wheel selection, an individual is selected for the next generation randomly with probability being proportional to its fitness, which is usually linearly scaled to improve the selective pressure [48]. On the other hand, *tournament selection* operates by holding a "tournament" among k randomly selected individuals (with k oten set to 2 or 3) and returning the fittest one. Rank-based selection mechanisms such as tournament selection are often considered to be superior over fitness-proportional selection mechanisms [53, 54].

2.5.2 Evolution Strategies

In general, evolution strategies focus on evolving individuals encoded as vectors of real numbers (which are usually called *objective parameters*), although the concrete implementations vary significantly. First efforts towards evolution strategies took place in the 1960s and 1970s by Hans-Paul Schwefel and Ingo Rechenberg [48].

In the most basic version of evolution strategy, mutation is performed by adding a normally distributed random vector $\mathbf{N}(\mathbf{0}, \mathbf{I})$ to the objective parameters. Additional techniques can be used to control the mutation process, such as *self-adaptation*, which adds an additional *strategy parameter* σ to each individual, thus specifying mutation strength. In whole, mutation with self-adaptation can be defined as

$$\begin{aligned} \sigma_l &\leftarrow \sigma_l \mathrm{e}^{\tau \mathrm{N}(0,I)}, \\ \boldsymbol{y}_l &\leftarrow \boldsymbol{y}_l + \sigma_l \mathbf{N}(\mathbf{0}, \mathbf{I}), \end{aligned} \tag{2.25}$$

where σ_l is the strategy parameter for individual l, $\boldsymbol{y}_l$ is the vector of objective parameters for individual l, and τ is the *learning hyperparameter* controlling the rate of self-adaptation [55]. Various recombination mechanisms can be used, most notably arithmetical averaging [55], which is typically performed

Algorithm 1 $(\mu/\rho, \lambda)$ evolution strategy with self-adaptation [55]

1: $\mathbf{P} \leftarrow$ randomly initialized parent population of size μ
2: **repeat**
3: $\quad \tilde{\mathbf{P}} \leftarrow$ empty offspring population
4: $\quad$ **while** $|\tilde{\mathbf{P}}| < \lambda$ **do**
5: $\quad\quad \boldsymbol{r} \leftarrow$ individual recombined from ρ parent individuals randomly sampled from $\mathbf{P}$
6: $\quad\quad \boldsymbol{r} \leftarrow \boldsymbol{r}$ with mutated strategy parameters
7: $\quad\quad \boldsymbol{r} \leftarrow \boldsymbol{r}$ with mutated objective parameters
8: $\quad\quad \tilde{\mathbf{P}} \leftarrow \tilde{\mathbf{P}} \cup \{\boldsymbol{r}\}$
9: $\quad$ **end while**
10: $\quad \mathbf{P} \leftarrow$ new parent population selected from offspring population $\tilde{\mathbf{P}}$
11: **until** termination criterion fulfilled
12: **return** best objective parameters from $\mathbf{P}$

on the objective parameters as well as on the strategy parameters if present [48].

Two ways of performing selection in evolution strategies are commonly used. *Comma selection* selects individuals for new parent population only from the offspring population, and evolution strategies that utilize this method are usually denoted as $(\mu/\rho, \lambda)$-ES, where μ is the size of parent population, ρ is the size of offspring population, and $\rho \leq \mu$ is the *mixing number*, that is, the number of parents used for recombination. On the other hand, *plus selection* selects individuals from both the parent and the offspring population; evolution strategies utilizing plus selection are commonly denoted as $(\mu/\rho + \lambda)$-ES. [55] For both methods, it is common to use deterministic truncation selection which selects top k individuals [55]. The pseudocode for $(\mu/\rho, \lambda)$-ES can be found under algorithm 1.

Chapter 3

Related Work

3.1 Anytime Learning

In the field of anytime learning, a classical work is [16] which, besides coining the term anytime learning, focused on using a genetic algorithm for controlling interactions of an agent with a changing environment using a two-module system. A similar approach was also investigated in [56]. Regarding interactions of agents with external environments, anytime learning was also utilized together with a hexapod robot in [57], [58] and [59].

[13] introduced an anytime framework for induction of decision trees that traded computational time for higher quality of trees, most notably in a novel interruptible algorithm that starts with a greedily generated tree and continuously improves individual subtrees. Anytime learning with decision trees was also studied in [60], which introduced an anytime technique for producing *anycost classifiers*, that is, classifiers whose misclassification errors decrease with allocation of more resources at inference time. Anytime classification was also the topic of [61], a paper which focused on classification of streaming data in which the varying time between two streams must be utilized in the best possible way.

A different approach was investigated in [62], which introduced an anytime learning technique in which an anytime algorithm is used for creating feature representations, which can then be used to turn conventional models into anytime models.

3.2 Parameter Pruning

A classical technique on neural network parameter pruning is *Optimal Brain Damage* [63]. In this method, the *saliency* of every parameter, i.e. change in the objective function caused by deleting that parameter, is estimated using the second derivative of the objective function. The parameters with the lowest saliency are then pruned.

A more recent approach to pruning is to use a sparsity-inducing regularizer, such as the l_1 norm, which is also known as *lasso* and was originally used mainly as a technique for feature selection [64]. *Connection pruning* was utilized for example in [65] and [66] for significantly reducing the sizes of AlexNet and VGG-16 convolutional models. However, no practical computational speedups were observed. In some cases, sparse representations of artificial neural models can be utilized to achieve computational efficiency [67].

With dense representations of artificial neural networks, pruning individual parameters is typically not sufficient, as these parameters are grouped into units such as neurons or filters. As was described in section 2.3.1.3, such groups of parameters can be pruned by using a structured sparsity regularizer such as group lasso. Regarding deep learning models, structured sparsity was utilized in [68] with convolutional neural networks to reduce the model sizes and computation costs.

[69] focused on an approach to pruning deep neural networks in which there is not produced a fixed pruned model for deployment, but instead the model is pruned dynamically at runtime using a technique based on reinforcement learning.

3.3 Auto-Sizing

3.3.1 Original Paper

Auto-sizing is a technique that was introduced by Kenton Murray and David Chiang in a 2015 paper [1]. Auto-sizing automatically determines the numbers of neurons in hidden layers of an artificial neural network, and can additionally be used to prune previously trained models.

The method works as follows: first the weights of incoming connections of every neuron in a fully-connected model are grouped, and the model is regularized using a structured sparsity regularizer. The model is then trained. If the regularization term is sufficiently large, then at the end of training some of the neurons have all of the weights of the incoming connections set to a value close to zero, and thus do not contribute to the outputs of the model. Such neurons can then be pruned from the model altogether, resulting in a model that is smaller than the initial one.

In the original paper, two structured sparsity regularizers were used, namely the $l_{2,1}$ and $l_{\infty,1}$ norm. Experiments were performed on simple n-gram natural language models, which were composed of two hidden layers and trained using the proximal gradient method. It was shown that auto-sizing can lower the perplexity of the models while decreasing the number of parameters.

3.3.2 Related Research

An approach very similar to auto-sizing was investigated in [2] on large-scale image classification datasets with deeper convolutional models, and it was shown that the method can prune the number of parameters of a model by up to 80% while retaining its predictive accuracy. In [3], the authors took a similar approach with comparable results. The algorithm used for training was forward-backward splitting, and for inducing sparsity, besides using the $l_{2,1}$ norm, the authors also experimented with tensor low rank constraints. Auto-sizing was also examined further by the original authors in [4], who successfully utilized the original auto-sizing method for neural architecture search with Transformer models.

In [5], the authors presented a neural network architecture design method called *MorphNet*, which aims to find an optimal model under the specified resource constraint (such as model size or inference speed). The method works by iteratively shrinking and growing the model while training. Shrinking is achieved by utilization of a sparsity-inducing regularizer, more specifically the l_1 norm on the γ_L variables of batch normalization [70]. Growing is performed by uniformly expanding all hidden layer sizes as much as the resource constraint allows, thus restructuring the model while approximately preserving its original size. Training was carried out using gradient descent. An approach somewhat similar to MorphNet is proposed in [6], where the authors iteratively grow a small initial artificial neural network model, again according to a specified budget in form of number of parameters or FLOPS. The technique allows for growth of new units as well as of entire layers, by using indicator variables indicating the presence of each component. These variables are continuous for most of the training, but effectively approach binary values as training progresses.

3.4 Neuroevolution

Neuroevolution is the field of application of principles of evolution to artificial neural networks. A common approach is to evolve both the parameters as well as the topology of the networks, known as TWEANN (topology and weight evolving artificial neural networks) [71]. An important TWEANN method is NEAT [72]. The technique starts with a population of small and simple artificial neural networks and *complexifies* them via evolution, thus employing growth of new units. Each model is represented by a genome of variable length, in which genes correspond to individual neurons or connections. Crossover is made possible by the introduction of *historical markings*, which align two genomes by their historical origins. Diversity in the population is enhanced using niching, more specifically by the *explicit fitness sharing* technique, which forces similar individuals to share their fitness payoff. HyperNEAT [73] and

CoDeepNEAT [74] are similar techniques suitable for larger and deeper models.

Another noteworthy TWEANN method was presented under the name *Symbiotic, Adaptive Neuro-Evolution* (SANE). SANE utilizes an approach called *symbiotic evolution*, which evolves a population neurons while promoting both cooperation and specialization of the neurons. Another population of neural networks constructed from these neurons is evolved simultaneously. As the function of a neuron can change depending on where it is placed in the network, [75] improved upon SANE by introducing separate populations of neurons for different positions in the network in a technique called *Enforced Sub-Populations* (ESP).

Another interesting approach is to work with ensembles of models. Evolution can then be performed on trees representing hierarchical algorithmic ensembles, as investigated in [76].

3.5 AutoML

Automated machine learning, broadly known as *AutoML*, can be described as a field that aims to automate the process of construction of machine learning pipelines [77]. Individual AutoML approaches usually focus on problems such as data preparation, feature engineering, model selection, selection of optimization algorithms, and model evaluation [78], although complete AutoML solutions such as TPOT [79], Auto-Sklearn [80], Auto-Keras [81], and NNI [82] are also available.

A notable subfield of AutoML is *neural architecture search* (NAS), which focuses on automating the process of designing artificial neural network architectures. An important NAS technique related to this thesis is the differentiable neural architecture search (DARTS) framework presented in [83], which allows to find an optimal architecture of deep learning models by working with high-level cells composed of arbitrary operations, and selecting over possible combinations of these cells using the softmax function. Similarly to the method presented in this thesis, differentiability implies that an optimizer based on gradient descent can be utilized instead of more complex techniques such as evolution.

Another important AutoML topic is hyperparameter optimization, with many of the techniques based on Bayesian optimization. One of the popular subfields is sequential model-based optimization, which iterates between fitting models and using the results to choose the next combination of hyperparameters to investigate [84]. A notable related method is the *Hyperband* algorithm [85], which perceives hyperparameter optimization as an infinite-armed bandit problem and creates a tradeoff between resource budgets and the quality of the hyperparameters. This is done by allocating the resources

only to the most promising hyperparameter combinations, as the algorithm iteratively discards the worse half of the pool of combinations [78].

Chapter 4

Dynamic Auto-Sizing

Similarly to the original auto-sizing technique, dynamic auto-sizing requires a structured sparsity regularizer for limiting the size of the model during training. Although standard regularizers that induce structured sparsity (such as the $l_{2,1}$ and $l_{\infty,1}$ penalty) can be used together with dynamic auto-sizing, as part of the work on this thesis a novel suitable regularization method was developed, which is presented here under the name *weighted* l_1 *regularization*. This chapter starts with a discussion of l_1 regularization, which induces non-structured sparsity, and is then in the next section modified into the weighted l_1 regularization method, which already induces structured sparsity.

In artificial neural networks, the l_1 regularization belonging to the parameters $\mathbf{W}$ of l-th fully-connected or convolutional layer can be written as

$$\alpha\Omega(\mathbf{W}) = \alpha \sum_{i=1}^{N_l} \sum_{j=1}^{N_{l+1}} \sum_{k=1}^{P_l} |\mathrm{W}_{ijk}|, \tag{4.1}$$

where W_{ijk} is the k-th parameter of the connection from the i-th unit (neuron or filter) in the l-th layer to the j-th unit in the $(l+1)$-th layer, N_l is the number of units in the l-th layer, P_l is the number of parameters per outgoing connection from the l-th layer. This notation will be useful in the following section. l_1 regularization induces sparsity, meaning that using it within a neural network typically leads to the unimportant parameters ending up with a value very close to zero. If we consider only fully-connected models, then connections with only such parameters virtually do not contribute to the outputs of the model, meaning that they could be removed from the model altogether. However, with the omnipresent dense tensor representations of neural network models, individual connections can't be removed - this is only possible with whole units.

4.1 Weighted l_1 Regularization

Weighted l_1 regularization is a novel regularization technique that can be used for inducing structured sparsity. As the name suggests, weighted l_1 regularization is based on standard l_1 regularization. If we number the units (neurons or filters) in every hidden layer, then the trick for inducing structured sparsity in the model is to set a higher strength of l_1 regularization to the parameters belonging to all incoming connections of units with a higher index - the intuition is that every additional unit in a hidden layer is more costly for the network than the previous unit in the same hidden layer. The general formula for weighted l_1 regularization term is

$$\alpha\Omega(\mathbf{W}) = \alpha \sum_{i=1}^{N_l} \sum_{j=1}^{N_{l+1}} \sum_{k=1}^{P_l} w(j)|\mathrm{W}_{ijk}|, \tag{4.2}$$

where w is an arbitrary non-decreasing function which sets regularization strength for the unit with index j. Therefore in every hidden layer, the parameters of every connection leading to the j-th unit are l_1-regularized with the coefficient of $\alpha w(j)$. Note that weighted l_1 regularization typically should not be applied to the output layer.

In the experiments presented in this thesis, an identity $w(j) = j$ is used for the function w. Combined with the non-negative term α controlling the strength of regularization, w can take the form of any non-decreasing linear function passing through origin, while experiments with non-linear functions remain for future research. Regarding the bias terms, in this work they are considered parameters belonging to special connections and are regularized as well, as this is necessary for the pruning of whole units.

4.2 Auto-Sizing

Section 3.3.1 described the original auto-sizing method. Weighted l_1 regularization can be directly utilized in the auto-sizing setting. This section describes how weighted l_1 regularization induces structures sparsity necessary for the correct functioning of auto-sizing, and at the same time recapitulates the original auto-sizing technique.

Training a neural network regularized with weighted l_1 regularization generally leads to the state in which for units with a high-enough index (and therefore with a large-enough strength of l_1 regularization), the optimizer has set the parameters of all of the incoming connections to values very close to zero. At the end of training, such units can be "pruned", i.e. removed from the weight matrices, without any relevant changes in the model's outputs.[9] The trained and pruned model can then be used in the usual "static" manner,

[9] In this work, the use of activation functions that satisfy $f(0) = 0$ is assumed.

e.g. for inference. In theory, the layer sizes of such a pruned model are not much dependent on the initial layer sizes, provided that the initial layer sizes were large enough so that enough neurons were regularized with a very high amount of l_1 regularization and therefore ultimately pruned. This is approximately the principle of the original auto-sizing method, the main difference being that original auto-sizing utilized the $l_{2,1}$ and $l_{\infty,1}$ regularizers.

4.3 Dynamic Auto-Sizing

Dynamic auto-sizing brings two major improvements to the original auto-sizing method. The first enhancement is that units that do not contribute to the model outputs are pruned periodically instead of only at the end of training, which improves performance as calculations are performed with smaller tensors. The second enhancement, which addresses the problem of original auto-sizing that training had to start with large layer sizes, is to periodically "grow" new units by adding them to the corresponding weight tensors with small initial random values of parameters - this at first does not change the outputs of the model, but if the new units are beneficial for the optimized task, the optimizer eventually increases the corresponding parameters and they become a natural part of the model (otherwise, the newly grown units are removed during the next pruning step). Therefore, it is possible to start the training with a small model and only gradually increase its size if necessary, leading to computational efficiency and relaxed requirements on the initial layer sizes.

A model trained using dynamic auto-sizing therefore starts with the initially set layer sizes and gradually keeps growing or shrinking. The size of the model typically stabilizes after some number of epochs, in a state near to an equilibrium in which all of the new units added during a growing step would be pruned during the subsequent pruning step. Utilizing larger regularization strength α typically leads to a smaller resulting model.

The complete pseudocode for dynamic auto-sizing can be found under algorithm 2. In theory, it is problematic that the initial parameter values of the newly grown units are multiplied by the pruning threshold τ, as this changes the average values of the units' outputs and negates some of the beneficial properties of parameter initialization techniques. However in practice, this does not seem to significantly impair the performance of the models, although the topic certainly remains up for debate.

Algorithm 2 Dynamic auto-sizing

1: *model* ← randomly initialized model, regularized with suitable sparsity inducing regularizer
2: $\tau \leftarrow$ pruning threshold
3: $\gamma_{perc} \leftarrow$ growth percentage
4: $\gamma_{min} \leftarrow$ minimum growth
5: **for** each epoch **do**
6: **for** each fully-connected or convolutional hidden layer l **do**
7: $n \leftarrow$ number of units in l
8: $\gamma \leftarrow max(n \cdot \gamma_{perc}, \gamma_{min})$
9: Grow l by adding γ new units with parameters randomly initialized and multiplied by τ
10: **end for**
11: Fit *model* once on the train set
12: **for** each fully-connected or convolutional hidden layer l **do**
13: Prune l by removing units with all parameters smaller in absolute value than τ
14: **end for**
15: **end for**
16: **return** *model*

Chapter 5

Anytime Learning with Dynamic Auto-Sizing

This chapter presents several anytime learning algorithms that can work together with dynamic auto-sizing models. Some of the algorithms are directly based on dynamic auto-sizing, while the rest of the algorithms can in theory benefit from the use of dynamic auto-sizing models, as it is argued in the corresponding sections. All of the algorithms are interruptible and produce a trained model upon interruption, and as all of the algorithms can also be perceived as hyperparameter optimization algorithms, they could also be trivially modified to additionally produce the corresponding hyperparameter combinations. Moreover, the algorithms are monotonic as the quality of the solutions can only increase with time. The performances of the algorithms are experimentally evaluated in chapter 6.2

5.1 Random Search

Although random search emerged mainly as a hyperparameter optimization technique, it can be considered an anytime algorithm if we apply the simple modification of always returning the overall best model. In this way, the algorithm can continue improving indefinitely, although the rate of improvements can be expected to slow down over time.

As was described in section 2.4.2.2, random search can be considered a relatively powerful algorithm, with two major disadvantages. The first disadvantage is that random search tries hyperparameters merely randomly, without focusing on regions that have been found to be promising. As many machine learning methods are stochastic, it is possible to switch from exploration to exploitation simply by restarting the training with the best hyperparameter combination with a good chance that the training will result in a better model. The second issue with random search is that in its basic variant with a fixed

number of training epochs, random search does not monitor the progress of training, and thus does not interrupt the training of models that give bad results from the very start of the training. This problem is alleviated by the introduction of early stopping. Early stopping still does not interrupt training of models that improve merely slowly from the start, but this behavior can in some cases be considered reasonable as, for example, it is often the case with small learning rate that training will proceed more slowly but the final performance of the model will be better.

Applying dynamic auto-sizing to random search is quite straightforward. Training can simply be switched from using a classical static model to using a dynamic auto-sizing model, with the hyperparameters that control the hidden layer sizes being replaced by a single hyperparameter that controls the strength of regularization [10]. This is a reasonable approach, as it can significantly decrease the number of hyperparameters (provided that dynamic auto-sizing functions correctly), and it will be demonstrated in chapter 6 that dynamic auto-sizing can often bring predictive performance benefits.

5.2 Anytime Grid Search

Presumably the largest advantage of grid search is that it is a systematic technique, meaning that it explores all of the specified regions of the hyperparameter space. In an anytime setting, this behavior is more difficult to achieve. One possible approach is presented in this thesis under the name *anytime grid search*. Anytime grid search works iteratively over increasing "levels" - on each level, a grid search is run in a more granular fashion than on the previous level, while skipping the hyperparameter combinations that have already been examined by the previous iterations. As the number of combinations on a level can grow very quickly due to the curse of dimensionality, the order of the examined hyperparameter combinations is randomized so that the algorithm usually does not get stuck in small parts of the hyperparameter space for too long. Because of this, after several levels a run of anytime grid search starts to resemble random search, with the main difference being that the allowed hyperparameter values are discretized in case of anytime grid search.

More specifically with n hyperparameters, in a single level of anytime grid search it is iterated over the n-fold Cartesian product of the progressively more granular set X of allowed relative values of a single hyperparameter, after the subtraction of the set of the values from the previous levels. One concrete way is to set the allowed relative hyperparameter values as

$$X = \bigcup_{i=1}^{2^l-1} \frac{i}{2^l}, \tag{5.1}$$

[10] Although dynamic auto-sizing introduces new hyperparameters such as the initial hidden layer sizes, it will be shown in chapter 6 that these hyperparameters do not require heavy fine-tuning as certain values work well in many training settings.

where l is the number of the level starting at 1. The concrete hyperparameter values can then be obtained analogically to linear scaling, with the relative value of 0 denoting the minimal allowed value and the relative values of 1 denoting the maximum allowed value. A visualization of the explored relative values of two hyperparameters for the first four iterations of anytime grid search can be found in figure 5.1. An alternative formula for the allowed relative hyperparameter values within a level could be designed so that the second level of the algorithm tests the lower and upper bound of the interval.

As with random search, dynamic auto-sizing can be applied to anytime grid search straightforwardly by switching from training with static models to dynamic auto-sizing models, together with dropping the hyperparameters controlling individual layer sizes and replacing them with a single hyperparameter controlling the regularization strength.

5.3 Anytime Hyperparameter Evolution

As was mentioned in section 5.1, random search carries the two limitations of not exploiting promising regions of the hyperparameter space, and not interrupting training prematurely when it does not appear to lead to a satisfactory result. In order to address these two limitations, an evolutionary algorithm was designed as part of this thesis, presented here under the name *anytime hyperparameter evolution*. This technique allows to exploit promising sections of the hyperparameter space, interrupt training of unsatisfactory models, and

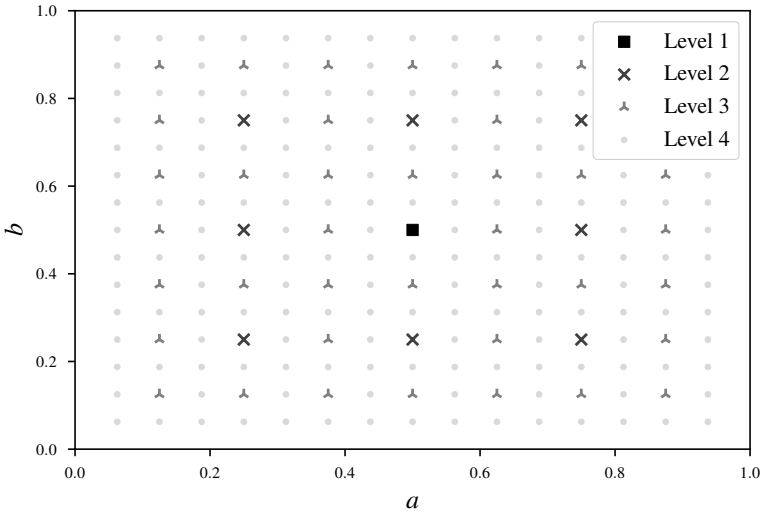

Figure 5.1: Visualization of the first four levels of iterations of anytime grid search that tests combinations of values of two hyperparameters. Iterations within the same level are tested in a random order.

Algorithm 3 Anytime hyperparameter evolution

```
1: function ANYTIME-HYPERPARAMETER-EVOLUTION(hyperparam_conf)
2:     λ ← population size
3:     λ_new ← number of new individuals added in each generation
4:     best_model ← null
5:     P ← INITIALIZE-INDIVIDUALS*(λ, hyperparam_conf)
6:     loop
7:         individual_best ← fittest individual from P
8:         P ← P ∪ RECOMBINATION*(P, hyperparam_conf)
9:         P ← P ∪ INITIALIZE-INDIVIDUALS*(λ_new, hyperparam_conf)
10:        TRAIN-MODELS*(P)
11:        P ← select λ individuals from P using tournament selection
12:        P ← P ∪ {individual_best}
13:        increase age of all individuals in P by 1
14:        update best_model using models of individuals from P
15:    end loop
16:    return best_model upon interruption
17: end function
```

moreover allows for the possibility of improving indefinitely as it periodically trains new models.

Anytime hyperparameter evolution is mostly based on evolution strategies, with several important modifications. First and foremost, an individual is represented by a combination of hyperparameters and also a complete model, but the genome is merely a vector of real numbers that defines the model's hyperparameters. Therefore, the genotype is decoupled from the phenotype significantly. As is common in evolution strategies, during every epoch, the genome of every recombinant individual is recombined from several parents that are randomly sampled from the population, and afterwards it is randomly mutated, corrected, and added to the population. Mutation is performed on the genome in the usual manner of evolution strategies, ideally with separate strategy parameters for each hyperaparameter and fixed to values provided by expert in order to speed up the process[11]. Correction is performed so that the hyperparameter values defined by the genome are placed in the valid ranges. The model of each individual stays unchanged by recombination, mutation, and correction, but is subsequently trained for a single epoch with use of the hyperparameter values defined by the genome. After this training of the individuals' models, the offspring population is created using tournament selection, with the fitness function defined as the performance on the validation set divided by an age penalty, which is greater than 1 for individuals whose

[11]During prototyping, it was also experimented with random mutation of models, which was implemented as noise being added to the model parameters. This, however, did not seem to improve the performance of the algorithm.

models have existed for longer than the specified number of epochs[12]. To serve as possible replacements for the old individuals, new randomly generated individuals are introduced in each generation after the recombination operator. Elitism is also applied, which copies the best individual from the previous generation and adds it to the current generation. This skips the current epoch's training for this model. In order to help the reader with understanding of the algorithm, high-level pseudocode is provided under algorithm 3, with pseudocode of the custom components provided under algorithm 4.

The typical behavior of anytime hyperparameter evolution is to start training multiple randomly initialized models, and progressively select a subset consisting of the best models and clone them with modified hyperparameter values. Experiments during prototyping suggested that it is important to clone the models instead of merely updating them in place with new hyperparameter values, as the modification can always have a negative effect on the performance of individual models as well as of the whole algorithm. In any case, after a number of epochs, only individuals belonging just to a handful of the original models remain in the population, but each is usually present in multiple copies with various values of hyperparameters. The algorithm thus gradually switches from exploration to exploitation. After a sufficient number of epochs, the age penalization of the existing models becomes sufficiently strong so that the models are replaced by new randomly initialized models. As with random search and grid search, the algorithm remembers the overall best result and always returns it upon interruption.

The integration of dynamic auto-sizing with anytime hyperparameter evolution works in a similar manner as with random search and anytime grid search, with the difference that the regularization strength is not constant during training, but rather gets updated dynamically according to the value of the hyperparameter in the genome. In the experiments, the initial regularization strength was set to a relatively high value so that the models started small, but the values of the regularization strength hyperparameter then progressively grew throughout the whole population as this enabled the models to grow and thus obtain higher fitness.

Overall, the largest differences of the technique from ordinary evolutionary algorithms are that the population size is typically much smaller in order to limit the computational costs, an individual's genotype only determines its phenotype very indirectly, and deliberate degeneration of the phenotypes in population is performed so that only a few best individuals are present (each individual still carries a different genome and thus a different combination of hyperparameter values). With small models and datasets, the technique could be modified so that during each generation, the model of each individual is

[12]This duration is measured since the random initialization of the model. The penalty can have the form of $1 + \alpha \cdot \beta^{age_{individual} - age_{limit}}$ if $age_{individual}$ is greater than age_{limit}, and 1 otherwise; α in this case is a small positive constant close to 0, and β is a constant greater than 1.

Algorithm 4 Components of anytime hyperparameter evolution

function Initialize-Individuals*(*count*, *hyperparam_conf*)
    $\mathbf{I} \leftarrow \{\}$
    while $|\mathbf{I}| < count$ **do**
        $individual \leftarrow$ New-Object()
        $individual.Genome \leftarrow$hyperparam_conf.Initial-Values()
        $individual.Model \leftarrow$ Initialize-Model()
        $individual.Age \leftarrow 0$
        $\mathbf{I} \leftarrow \mathbf{I} \cup \{individual\}$
    end while
    return I
end function

function Recombination*(**P**, *hyperparam_conf*)
    $\tilde{\mathbf{P}} \leftarrow \{\}$
    for each *individual* in **P do**
        $offspring \leftarrow$ Copy-Object($individual$)
        $offspring.Genome \leftarrow$ mean genome of ρ randomly sampled individuals from **P**
        $\boldsymbol{\sigma} \leftarrow hyperparam_conf.$Get-Strategy()
        mutate $offspring.Genome$ using the strategy vector $\boldsymbol{\sigma}$
        correct $offspring.Genome$ so that the values are within $hyperparam_conf.$Get-Allowed-Ranges()
        $\tilde{\mathbf{P}} \leftarrow \tilde{\mathbf{P}} \cup \{offspring\}$
    end for
    return $\tilde{\mathbf{P}}$
end function

function Train-Models*(**P**)
    for each *individual* in **P do**
        train $individual.Model$ for a single epoch using hyperparameter values from $individual.Genome$
    end for
end function

function Calculate-Fitness*(*individual*)
    $metric_{val} \leftarrow$ performance of $individual.Model$ on the validation set
    if $individual.Age \leq age\_limit$ **then**
        return $metric_{val}$
    end if
    $penalty \leftarrow 1 + \alpha \cdot \beta^{individual.Age - age\_limit}$
    return $metric_{val}/penalty$
end function

completely retrained from the start; this however is ineffective in case such training is computationally expensive, as the initial generations would train models on suboptimal hyperparameter combinations.

5.4 Progressive Model Growth

Dynamic auto-sizing enables to change size of the models over the course of training by tweaking the strength of regularization. As smaller deep learning models are in general faster to train than larger models as they need less data and the calculations are less computationally intensive, an interesting option is to start training with small models and large regularization strength, and progressively increase the size of the models by decreasing the regularization strength. Furthermore, the approach can be turned into an anytime algorithm by restarting the training with random hyperparameter values once the model ceases to improve[13]. This algorithm is presented here under the name *progressive model growth*.

The algorithm works on two levels. The high level can be perceived as random search, and simply calls the low level with a randomly sampled combination of hyperparameters. The low level randomly initializes a small dynamic auto-sizing model and sets the regularization strength to a large value. The model is then iteratively trained on a training dataset, and after each epoch is evaluated on the validation set. If the validation loss is higher than the best value achieved during this run of the low level, the training is said to be *stalled*, and the regularization strength is lowered using a constant multiplier. This allows the model to grow. In case the model improves beyond the best achieved validation loss within a specified number of epochs after stall (which can be called the *stall duration limit*), the low-level continues until the next stall, then the regularization strength is lowered again, and so on. However, in case the model fails to improve after stall, the regularization strength is not lowered further, and after the number of epochs specified by the stall duration limit, the low-level run is restarted by the high level with a new combination of hyperparameter values. The pseudocode is presented in figure 5.

Several variants of the scheduling of regularization strength are possible. The presented variant lowers the regularization strength at the point in time in which the model stops improving, which can ideally mean that the model is reaching its predictive capacity and growth could be beneficial. It was also experimented with the variant that lowered the regularization strength even sooner, at the point the rate of improvement became lower than a specified threshold, but this did not seem to be beneficial. On the other hand, another option would be to wait for several epochs after the model stops improving to see if the stall is only temporary, and only decrease the regularization strength if no improvement occurs.

[13]This can be expected to happen at some point due to overfitting.

Algorithm 5 Progressive model growth

```
1: function Progressive-Model-Growth(hyperparam_conf)
2:     best_model ← null
3:     loop
4:         model ← Initialize-Model()
5:         hyperparams ← hyperparam_conf.Random-Sample()
6:         reg_strength ← initial regularization strength
7:         reg_strength_min ← regularization strength limit
8:         val_loss_best ← ∞
9:         stall_duration ← 0
10:        stall_duration_max ← limit of duration of stall
11:        while reg_strength ≥ reg_strength_min and stall_duration ≤
   stall_duration_max do
12:            train model for a single epoch using hyperparams and
   reg_strength
13:            val_loss ← loss of model on the validation set
14:            if val_loss ≥ val_loss_best then
15:                if stall_duration == 0 then
16:                    reg_strength ← reg_strength · reg_strength_multiplier
17:                end if
18:                stall_duration ← stall_duration + 1
19:            else
20:                val_loss_best ← val_loss
21:                stall_duration ← 0
22:                update best_model if outperformed by model
23:            end if
24:        end while
25:    end loop
26:    return best_model upon interruption
27: end function
```

Chapter 6

Experiments and Results

This chapter describes the experimental evaluation of the methods described in chapters 4 and 5. The first part of the chapter, section 6.1, focuses on evaluation of dynamic auto-sizing as a standalone model training technique. Section 6.2 then focuses on evaluation of applications of dynamic auto-sizing in anytime learning algorithms.

6.1 Dynamic Auto-Sizing

6.1.1 Implementation

In order to conduct experiments, dynamic auto-sizing was implemented[14] in the TensorFlow library [86]. As the library did not allow to dynamically change the numbers of units with predefined layer types, custom fully-connected and convolutional layers were developed, with the growth and pruning of units implemented using low-level tensor operations. As it is not possible to modify the number of units in a layer without knowledge of the size of the previous layer, a custom class for sequential models was implemented which allows to successively prune or grow all layers in the model, while optionally leaving some of the layers unchanged. Moreover, as a change in the number of filters in a convolutional layer can non-trivially affect a subsequent fully-connected layer, a custom input flattening layer was implemented as well.

Although modern TensorFlow uses dynamic computational graphs by default, these can be compiled into static graphs in order to allow for reduced computational time. Therefore in the implementation, between the growth and pruning of units in each epoch, a static TensorFlow computational graph was compiled in order to speed up the fitting of the model to the data, thus

[14] The TensorFlow implementation, together with the anytime algorithms as well as all of the experiments presented in this thesis, are freely available under the MIT license at `https://github.com/vcahlik/dynamic-auto-sizing`.

limiting the need for the slower eager execution mode only for the relatively inexpensive operations of growth and pruning.

6.1.2 Setup

The method was configured as follows: the values of the hyperparameters λ_{perc} (growth percentage) and λ_{min} (minimum number of grown units in each layer) were set to 0.2 and 20, respectively. The pruning threshold τ was set to 10^{-3}. The initial parameters, as well as the parameters of the newly grown units, were initialized using LeCun normal initialization [87]. For the experiments in section 6.1, training was stopped after a predefined number of epochs, which was set so that the size of the model as well as the performance on the validation set would be stabilized. In some of the experiments, after the model was trained using the dynamic auto-sizing approach, fine-tuning in a static manner (without regularization, growth, or pruning) was performed. Unless stated otherwise, the regularization strength α was set to 2×10^{-5}, and the initial number of units in each hidden layer was set to 100.

All models in the experiments are composed of five hidden layers, usually with the first four being convolutional layers and the final one fully-connected. In case of the convolutional models, the second and fourth convolutional layer used strides of 2 to reduce the sizes of the feature maps, and these two layers were followed by a dropout layer with dropout rate of 0.2 and 0.5, respectively. No dropout was used with the fully-connected models. Instead of using batch normalization to mitigate the problem of unstable gradients, self-normalization [88] was used by standardizing the input data, using LeCun normal initialization for all layers, and using the SELU activation function for all layers except the output layer, for which either softmax or no activation was used, depending on the task. Training was performed with Adam optimizer and the categorical cross-entropy or mean squared error cost function. The experiments in section 6.1 were run on an Nvidia Tesla V100 GPU.

6.1.3 Dependency of Discovered Layer Sizes on Various Factors

The behavior of dynamic auto-sizing was initially analyzed on the CIFAR-100 dataset with a model composed of four convolutional layers and a single hidden dense layer. Random restarts of training with identical hyperparameters lead to very similar final layer sizes, as is shown in Fig. 6.1. This holds true even when training is restarted with various initial layer sizes, as can be seen in Fig. 6.2 and Fig. 6.3.

Another experiment demonstrates that training a dynamic auto-sizing model with different regularization strengths α leads to different resulting layer sizes, with a clear trend of higher amounts of regularization leading to smaller models, as can be seen in Fig. 6.4, which shows sizes of hidden layers

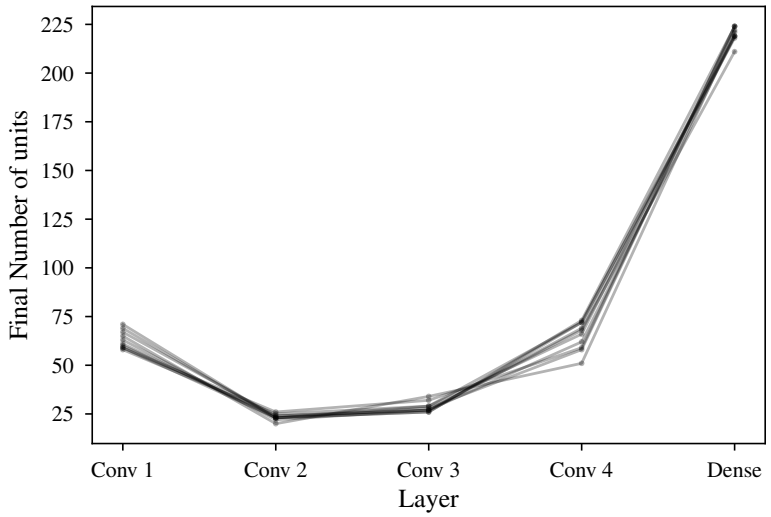

Figure 6.1: Final layer sizes of models trained using dynamic auto-sizing with identical training configuration. Each curve represents one model.

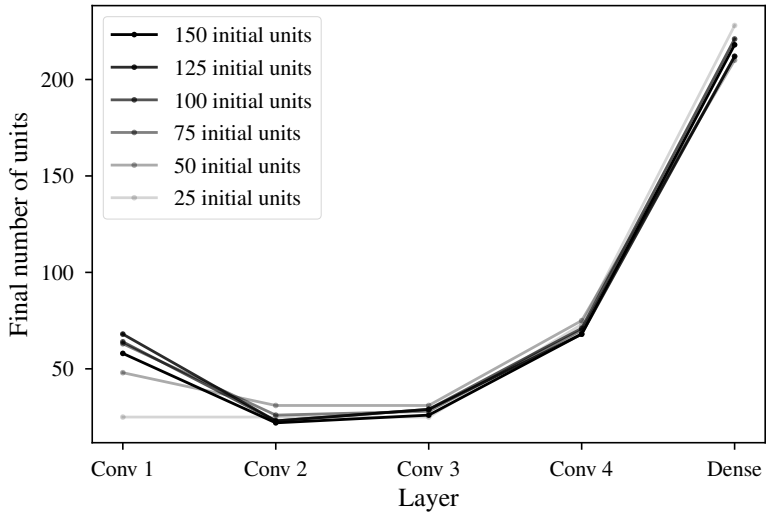

Figure 6.2: Final layer sizes of models trained using dynamic auto-sizing with various initial numbers of units, equal for each layer. Each curve represents one model.

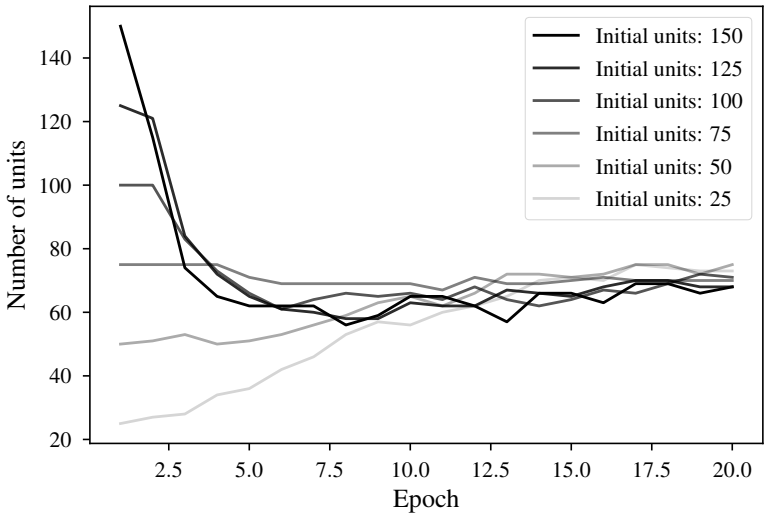

Figure 6.3: Evolution of the number of filters in the last convolutional layer over the course of training, for the same experiment as in Fig. 6.2.

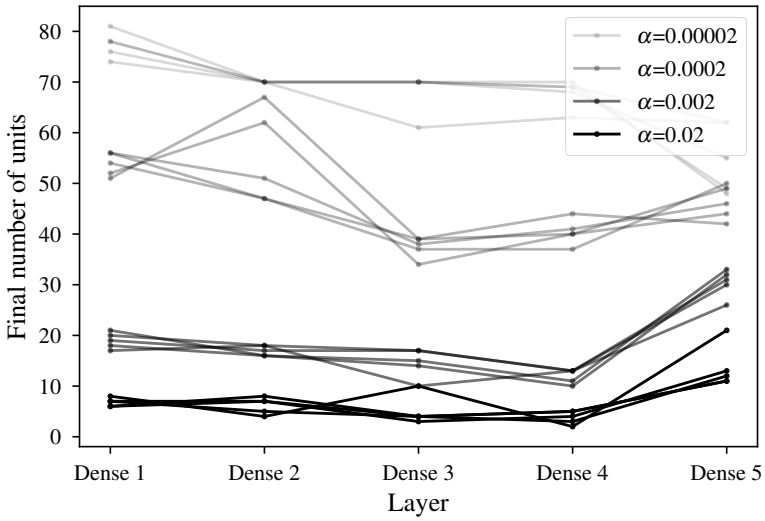

Figure 6.4: Final layer sizes of fully-connected models trained using dynamic auto-sizing with various regularization strengths α. Each curve represents one model.

of fully-connected models trained on a regression task on the 15-puzzle dataset [89] using dynamic auto-sizing.

Dynamic auto-sizing models tend to increase in size with the growing complexity of the problem domain, as can be seen in table 6.1, which lists the numbers of parameters, layer sizes, and accuracies of convolutional models trained using dynamic auto-sizing on the MNIST, Fashion MNIST, SVHN

Dataset	Parameters	Hidden layer sizes					Cross-val. accuracy
		conv1	conv2	conv3	conv4	dense	($mean \pm 2SD$)
MNIST	166K	19	14	18	34	93	99.4 % ± 0.1
Fashion MNIST	233K	22	15	21	36	124	93.3 % ± 0.5
SVHN	471K	17	16	21	45	158	93.1 % ± 0.3
CIFAR-10	665K	39	18	26	57	175	77.0 % ± 0.7
CIFAR-100	851K	66	20	30	64	194	45.1 % ± 1.1
Tiny ImageNet	5.32M	47	13	48	54	377	18.7 % ± 2.2

Table 6.1: Average sizes (total parameter counts and numbers of units in hidden layers) of convolutional models trained using dynamic auto-sizing on various computer vision datasets, together with cross-validated accuracies after fine-tuning.

(Street View House Numbers) [90], CIFAR-10, CIFAR-100, and Tiny ImageNet [91] datasets. This is not a surprising phenomenon, as training a small model on a complex task typically leads to a large training loss, which in turn leads to the regularization term not having as much significance in the cost function, allowing the model to grow.

6.1.4 Predictive Performance of Dynamic Auto-Sizing Models

In these experiments, predictive capabilities of dynamic auto-sizing models were tested against static models, as well as against dynamic auto-sizing models with the structured sparsity $l_{2,1}$ regularization, which roughly corresponds to the original auto-sizing method [15].

Training was performed on various computer vision datasets, with models composed of four convolutional layers and a final hidden dense layer. On each dataset, first a dynamic auto-sizing model utilizing weighted l_1 regularization was trained in a cross-validation setting, and the mean resulting layer sizes were used as the architecture of a static model (labeled A). In order to determine how suitable the layer sizes discovered by the dynamic model are for a static model, another static model (labeled B) was trained, this time with an equal number of filters in each convolutional layer. The number of filters per layer was set so that the number of parameters of the resulting model would be as close as possible to that of the first two models. The number of neurons in the final dense hidden layer was preserved. Also for each dataset, a dynamic model utilizing the structured sparsity $l_{2,1}$ regularization was trained, with a regularization strength α set so that the resulting model would have approximately the same number of parameters as the other models. All models were

[15] It was also experimented with the $l_{\infty,1}$ regularizer, which however did not seem to work well when coupled with the Adam optimizer. Theoretically, this could be due to the regularizer only reducing a single value of each group of parameters at every step, although this has not been experimentally confirmed.

Dataset	Cross-val. accuracy ($mean \pm 2SD$)			
	Dynamic, weighted l_1	**Static A**	**Static B**	**Dynamic, $l_{2,1}$**
MNIST	99.4 % ± 0.1	99.3 % ± 0.2	99.3 % ± 0.2	99.2 % ± 0.2
Fashion MNIST	93.3 % ± 0.5	92.4 % ± 0.3	92.8 % ± 0.4	92.1 % ± 0.3
SVHN	93.1 % ± 0.3	91.4 % ± 0.8	92.4 % ± 0.3	92.3 % ± 0.5
CIFAR-10	77.0 % ± 0.7	72.9 % ± 1.3	75.6 % ± 1.1	72.3 % ± 1.4
CIFAR-100	45.1 % ± 1.1	32.4 % ± 1.4	35.4 % ± 1.5	39.5 % ± 1.8
Tiny ImageNet	18.7 % ± 2.2	11.3 % ± 0.8	11.3 % ± 0.7	17.1 % ± 1.4

Table 6.2: Cross-validated accuracies of dynamic auto-sizing models trained with weighted l_1 regularization against the following baselines: *Static A* (static model with identical layer sizes), *Static B* (static model with similar number of parameters but uniform sizes of convolutional layers), and *Dynamic,* $l_{2,1}$ (dynamic model with comparable number of parameters, trained with $l_{2,1}$ regularization).

trained for 40 epochs, with the dynamic auto-sizing models using static fine-tuning for the last 20 epochs. For each model, the results corresponding to the best epoch were recorded. The learning rates were set independently for each model and dataset using grid search.

The cross-validated results can be found in table 6.2. For each dataset, the model trained using dynamic auto-sizing with weighted l_1 regularization surpassed the accuracies of all the baseline models, with the largest differences being present on the most difficult datasets (CIFAR-100 and Tiny ImageNet), where also dynamic auto-sizing with the $l_{2,1}$ regularization (analogous to the original auto-sizing method) outperformed both static models. It is interesting that static model B, which used a uniform number of filters for all convolutional layers, always matched or surpassed the accuracy of static model A, which utilized the layer sizes discovered by dynamic auto-sizing with weighted l_1 regularization. This shows that the layer sizes obtained using dynamic auto-sizing are not optimal for use in static models.

6.1.5 Analysis of Results

In order to explain the observed accuracy benefit of dynamic auto-sizing, further tests were performed on the CIFAR-100 dataset. First a dynamic auto-sizing model utilizing weighted l_1 regularization was trained in a cross-validation setting. The mean resulting layer sizes were then used as the architecture for analogous models with deactivated growth of new units, which further differed by the utilized regularization type, whether or not pruning of units was activated, and whether or not the last 20 epochs were run in the fine-tuning setting (with regularization, growth, and pruning deactivated). As in the previous experiments, all models were trained for 40 epochs, the best result of all the epochs was recorded for each model, and the learning rates as

Type	**Growth**	**Pruning**	**Regularization**	**Fine-Tuning**	**Cross-val. accuracy** ($mean \pm 2SD$)
Dynamic	Yes	Yes	weighted l_1	Yes	45.1 % ± 1.1
Dynamic	No	Yes	weighted l_1	Yes	45.0 % ± 1.0
Static	-	-	weighted l_1	Yes	43.8 % ± 1.5
Dynamic	No	Yes	weighted l_1	No	43.7 % ± 0.8
Static	-	-	weighted l_1	No	43.0 % ± 1.8
Static	-	-	l_1	No	43.8 % ± 1.5
Static	-	-	None	-	32.4 % ± 1.4

Table 6.3: Cross-validated accuracies of various types of models on the CIFAR-100 dataset.

well as the strength of l_1 regularization (where utilized) were set independently for each model using grid search.

The cross-validated results, which are presented in table 6.3, show that most of the predictive advantage of dynamic auto-sizing models comes from the mere use of weighted l_1 regularization, which alone in this experiment caused a 10.6 % accuracy boost over an unregularized model. Further, although smaller, accuracy boosts come both from utilizing fine-tuning and the use of pruning. Especially the accuracy benefit of pruning is surprising, as in each case, only a few units were pruned from the model during training. The results also show that utilizing mere l_1 regularization also results in a significant accuracy boost over an unregularized model, however l_1 regularization can not be utilized in a true dynamic auto-sizing setting as it does not induce structured sparsity.

6.2 Anytime Learning with Dynamic Auto-Sizing

In this section, performances of the various anytime learning algorithms presented in chapter 5 are analyzed and compared. As is common with anytime algorithms, this comparison is performed using both the performance traces and the estimated performance profiles, which are criteria that take into account both the predictive performance of the obtained models as well as the invested computational time. Additionally, it is attempted to give an explanation of the results by analyzing the performance metrics and durations of the individual runs.

6.2.1 Setup

Experiments were performed with convolutional and fully-connected models on various datasets. Compared to section 6.1, smaller datasets and models were used in order to save computational resources as relatively long runs of anytime learning algorithms are usually required to obtain relevant performance traces. More specifically, for each run (such as a single iteration of

random search or a single run of evolution), the train and test sets were obtained using random sampling from the datasets (with the train and test set never overlapping). This configuration has the benefit of being more flexible than classical cross-validation, as it removes the dependency between the test set size and the number of runs, which is beneficial as it does not require to set the number of runs in advance before running the experiment. Moreover, because of the small size of the models, training was in most cases faster on a CPU than with a GPU. Therefore all experiments were run in the TensorFlow eager execution mode on a system with an eight core AMD Ryzen 7 2700 CPU and 24 GB of RAM, except for the experiments with the CIFAR-100 and 15-puzzle datasets, which were run on a system with an Nvidia Tesla V100 GPU, Intel Xeon Gold 6254 CPU, and 64 GB of RAM, on compiled TensorFlow computational graphs. Regarding fine-tuning of the models (that is, training with omitted regularization), this technique was not used, as preliminary experiments surpringly showed that this is not beneficial to the approach. As for the sampling of the datasets, a 2.5 % sample was used for the Fashion MNIST, SVHN and CIFAR-10 datasets (with an 80:20 train set size to test set size ratio), a 0.5 % sample was used for the 15-puzzle dataset with a 90:10 train/test ratio, and 100 % "samples" of the CIFAR-100 dataset (with an 83:17 train/test ratio). The sample sizes were set large enough so that at least several hundred instances would be present in each test set.

The same model architectures were used as in experiments in section 6.1, more specifically, convolutional models were used for the computer vision classification tasks and fully-connected models were used for the regression tasks, with all models being composed of 5 hidden layers. Unless stated otherwise, the hyperparameters were set equally for all models and techniques. The hyperparameters that could be optimized by a specific method (such as random search and evolution) were set so that the allowed ranges would be identical, and therefore the comparison across methods would be fair in the sense that all methods would be allowed to set the hyperaparameters equally, this was not possible in some edge cases, which will be described in the appropriate sections.

6.2.2 Analysis of Individual Methods

6.2.2.1 Random Search

Random search was implemented as described in section 5.1, together with the described modification of always returning the best result upon interruption. To avoid the need for tuning a hyperparameter controlling the number of epochs, early stopping was used that halted the training if the model failed to improve on the test set for 8 consecutive epochs.

With "static" models that did not utilize dynamic auto-sizing, the tunable hyperparameters included five hyperparameters for the size of each hidden

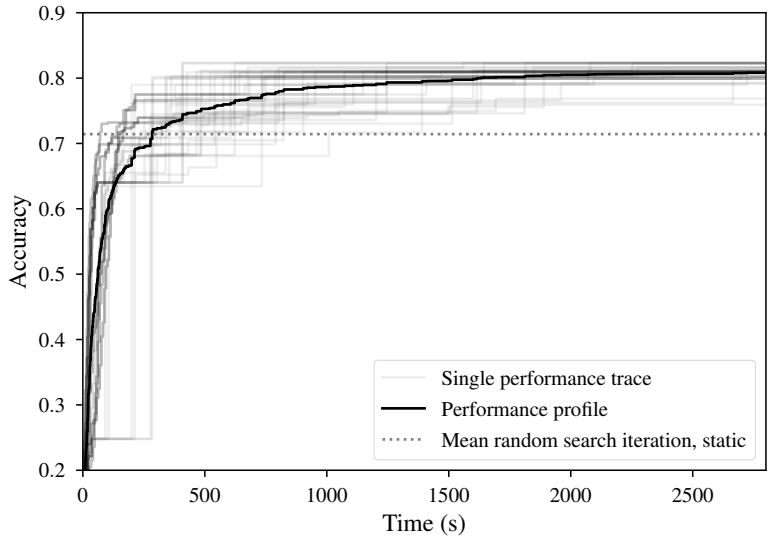

Figure 6.5: Individual test set performance traces and the corresponding performance profile for random search with static models trained on samples of the SVHN dataset. The dotted line represents the average accuracy of a single iteration.

layer (with uniform sampling, with range of 10 to 100 units), the learning rate (uniform sampling, range from $1 \cdot 10^{-4}$ to $6 \cdot 10^{-4}$), batch size (uniform sampling, range from 16 to 64), and the strength of L1 regularization, unless stated differently in the corresponding sections. The strength of L1 regularization would be allowed in the range of $1 \cdot 10^{-3}$ to approximately $3.2 \cdot 10^{-5}$, with first uniformly sampling an exponent value from the range of -4.5 to -3 and then returning the value of 10 to the power of this exponent. In some experiments, the ranges of allowed regularization strength were modified so that the resulting models would be similar in size to the static models, as will be mentioned in the corresponding sections. With dynamic auto-sizing models, the tunable ranges of all hyperparameter values were identical to the static version of random search, but the hyperparameters controlling the layer sizes were dropped (the sizes of the models were determined only by the regularization strength), with each hidden layer being initialized with 20 units. The allowed ranges of hidden layer sizes of static models were chosen so that both static and dynamic models would be allowed to grow to approximately equal sizes, and thus the allowed hidden layer sizes of static models were modified for some experiments, as will be noted in the corresponding sections.

Random search has the interesting property that the individual iterations are independent. Therefore, it is possible to increase precision of the performance profiles by sampling the performance traces from a pool of random search iterations. The number of sampled performance traces for each experiment was set to 50 in this thesis. This allowed to save some computa-

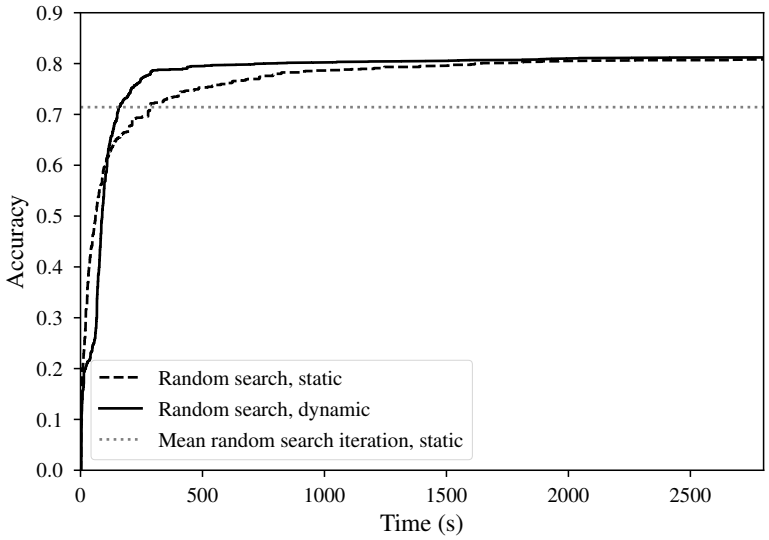

Figure 6.6: Estimated test set performance profiles for random search with static and dynamic models trained on samples of the SVHN dataset.

tional resources used for the experiments, although the experiments still were resource-heavy, as for better precision of the results it is necessary to obtain data of total duration that is longer at least by several multiples than the length of the obtained performance traces[16].

Several performance traces of random search with static models trained on samples of the SVHN dataset[17], along with the corresponding aggregated performance profile, can be seen in figure 6.5. It can be seen that individual performance traces are steppy rather than smooth; this is due to an imperfect design of the experiments, in which the best accuracy was not updated after each training epoch, but rather after each complete iteration. To help alleviate this problem and avoid these "bumps" at least for the initial minutes of the algorithm runs where differences in the accuracy metric are significant, the first iteration of all random search experiments was always sampled from a set of runs for which the best accuracy was updated after each epoch of training, resulting in much smoother performance profiles.

For static models on the SVHN dataset, the mean accuracy obtained by a random search iteration on the test set was approximately 71.4 %, with a mean iteration duration at 182 seconds. With dynamic auto-sizing models, the mean test accuracy grew to 74.9 %, but the mean iteration duration grew to 343 seconds. This increase in time per iteration, caused mostly by the dynamic auto-sizing models being trained for a significantly higher number of

[16]If the performance traces were sampled from a pool of equal total duration, each performance trace would end with an equal value of the metric.

[17]For experiments on the SVHN dataset, the allowed regularization strength was limited to the range from 10^{-5} to 10^{-4} for all algorithms.

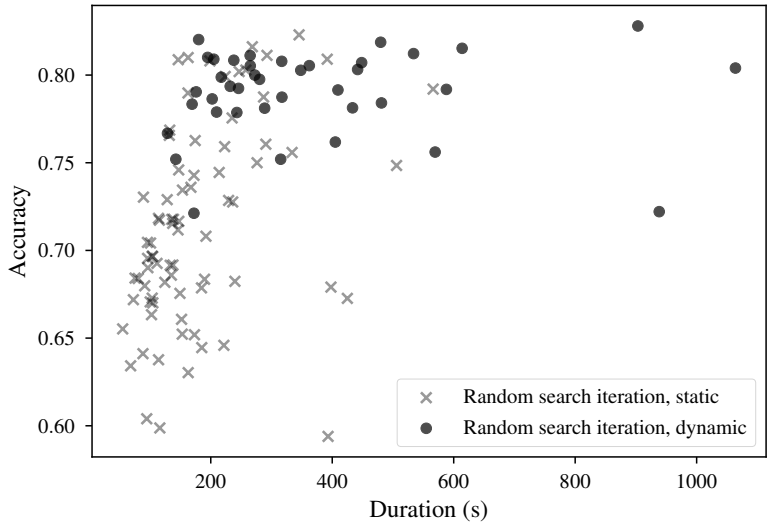

Figure 6.7: Test set accuracies and durations of individual iterations of random search with static and dynamic models trained on samples of the SVHN dataset. Two short and low-scoring iterations of random search with dynamic models have been omitted from the figure.

epochs before halted by early stopping[18], negatively affected the performance of dynamic models in the anytime setting. This can be seen in figure 6.6, which compares the estimated performance profiles of random search with static and dynamic models trained on samples of the SVHN dataset. It can be seen that the dynamic auto-sizing variant of random search soon takes lead over the static variant, but with the accuracies being very close most of the time. This can be explained by a scatter plot of the durations and accuracies of individual iterations presented in figure 6.7, which shows that the best accuracies achieved by iterations of random search with static and dynamic models are comparable.

Experiments were also run on the complete CIFAR-100 dataset[19]. As can be seen in figure 6.8, using dynamic auto-sizing also seems to provide a slight benefit over static models in this case, however at the end of the measured period, the static version of random search ties the performance of the dynamic version and it is unknown whether this trend would continue if the algorithms were allowed to run for a longer time.

[18]On average, an iteration of static random search consisted of roughly 40 epochs of training, while an iteration of dynamic random search consisted of about 80 epochs.

[19]For this dataset, the strength of L1 regularization was allowed in the range of $1 \cdot 10^{-5}$ to $1 \cdot 10^{-4}$, and for the static models, the size of the final hidden layer was allowed in the range from 50 to 400 units to match the sizes of the dynamic auto-sizing models.

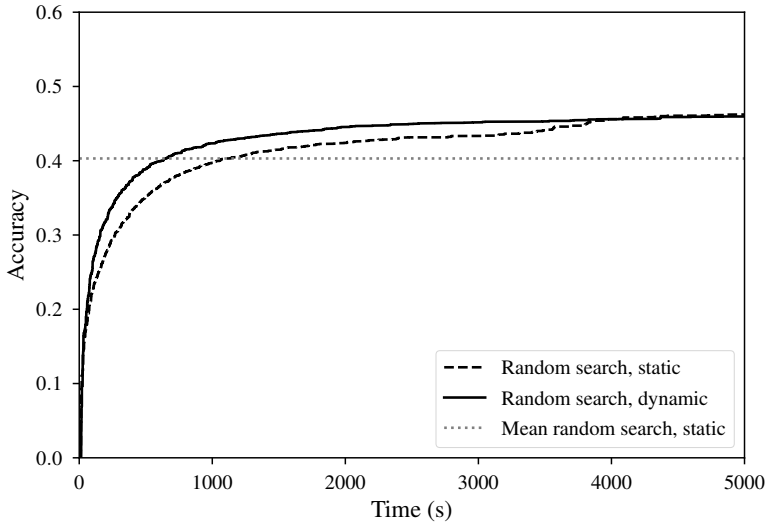

Figure 6.8: Estimated test set performance profiles for random search with static and dynamic models trained on the CIFAR-100 dataset.

6.2.2.2 Anytime Grid Search

Experiments with anytime grid search were performed similarly to random search. The algorithm was implemented as described in section 5.2. As with random search, the best result was always returned upon interruption, and early stopping was used to halt training once the model failed to improve on the test set for 8 consecutive epochs. However, as the individual iterations of grid search are not independent, complete performance traces were measured instead of generating them by sampling from a set of iterations. The number of measured performance traces was set to 6 for all experiments, and these performance traces were used to estimate each resulting performance profile. The configuration of the technique in terms of the used model architectures, as well as in terms of the allowed ranges of tunable hyperparameters, was identical to the configuration of random search, which was described in the previous section.

Performance traces along with the aggregated performance profile of anytime grid search with static models on the SVHN dataset can be seen in figure 6.9. The mean accuracy of individual iterations on the test set was approximately 70.9 %, with a mean iteration duration at 136 seconds. With dynamic auto-sizing models, the mean test accuracy grew to 76.2 %, and the mean iteration duration grew to 303 seconds. The performance profiles of both variants can be seen in figure 6.10. This time, utilization of dynamic auto-sizing shows a significant improvement in terms of the tradeoff of time against accuracy.

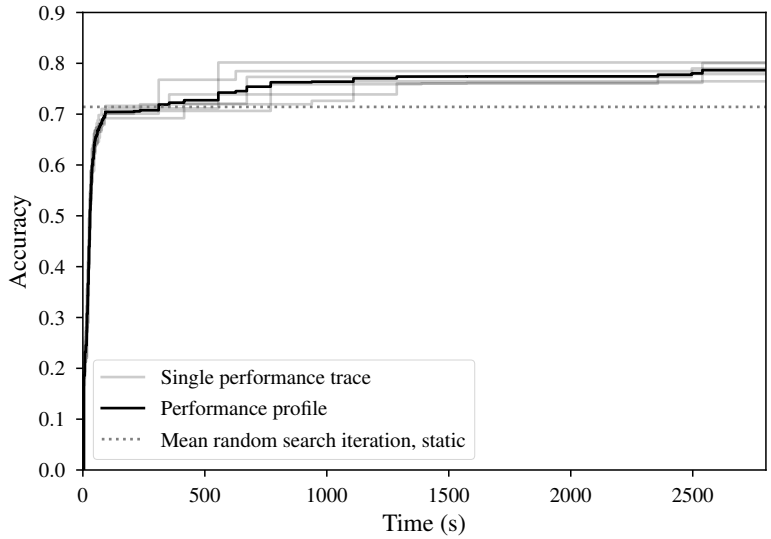

Figure 6.9: Individual test set performance traces and the corresponding performance profile for anytime grid search with static models trained on samples of the SVHN dataset.

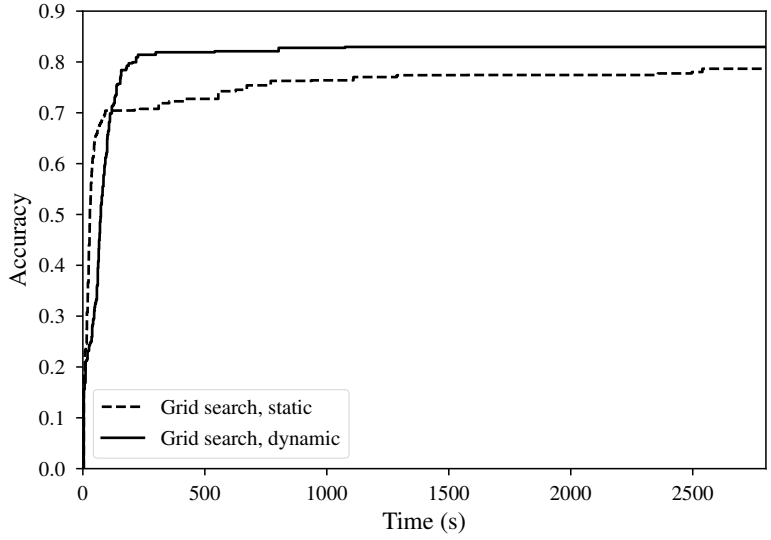

Figure 6.10: Estimated test set performance profiles for anytime grid search with static and dynamic models trained on samples of the SVHN dataset.

6.2.2.3 Anytime Hyperparameter Evolution

Similarly to the other methods, anytime hyperparameter evolution was implemented in Python, with the algorithm behaving as described in section 5.3. The computational expenses were overwhelmingly dominated by the training of the models of the individuals. The method was configured so that there are 10 individuals in the parent population, the tournament size is 3, offspring individuals are recombined from 5 parents, and 2 randomly initialized individuals are introduced in each epoch; the length of the period for which new individuals are not penalized was set to 10. Experiments were run only with dynamic auto-sizing models, which used similar training configuration to the models used with random search and anytime grid search, and equal hyperparameters and hyperparameter ranges to the ones stated stated in section 6.2.2.1. The initial values of the hyperparameters were set to 32 in case of batch size, $4 \cdot 10^{-4}$ for learning rate, and $1 \cdot 10^{-3}$ for regularization strength. The initial regularization strength is the highest allowed value of the hyperparameter; this was set in order to ensure that the models would start at their smallest possible size and gradually grow over time, in the case that higher regularization strengths would be beneficial.

During the run of anytime hyperparameter evolution, the tracked value of the best accuracy on the test set is updated after every generation. This is imperfect design, as it would be slightly beneficial for the performance of the algorithm to update the best accuracy after the training of each individual. Regarding the sampling of datasets, this was always performed only once at the start of the algorithm run. As with grid search, complete performance traces were always measured, with the final performance profiles estimated from 6 performance traces in all experiments. The performance traces of runs of the algorithm on samples of the Fashion MNIST dataset, as well as the resulting performance profile, are presented in figure 6.11.

6.2.2.4 Progressive Model Growth

Progressive model growth was implemented according to the description in section 5.4. The high-level of the algorithm, that is random search of hyperparameters, was configured identically to the experiments with random search described in section 6.2.2.1. In the experiments, the initial regularization strength was set to $1 \cdot 10^{-3}$ and the miminum regularization strength was set to $3.2 \cdot 10^{-5}$, as in the experiments with anytime hyperparameter evolution. The maximum duration of stall was set to 10 epochs.

As the experiments were performed with very small datasets due to limited computational resources, the potential performance benefit of progressive model growth could not be utilized, as the duration of a single run of the low level is negligible. In this way, the results are mainly influenced by the resulting accuracies of the runs of the high level. On the other hand, preliminary

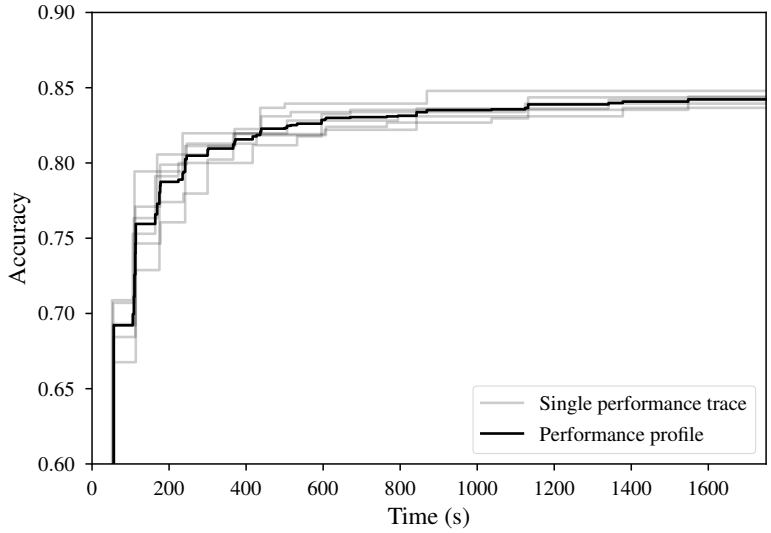

Figure 6.11: Individual test set performance traces and the corresponding performance profile for anytime hyperparameter evolution, trained on samples of the Fashion MNIST dataset.

experiments with larger datasets suggested that models obtained by progressive growth usually converge to worse predictive performance than models that are trained with low regularization strength right from the start of training, even if these models start with equally small layer sizes and the size of resulting models is equal to the size of models trained using progressive model growth. This could be explained by inherently worse performance of the optimizer in cases where growth of the model happens only after the smaller model is almost fully trained, although more experiments are necessary to support this hypothesis.

As with grid search and anytime hyperparameter evolution, the performance profiles of all experiments with progressive model growth were estimated using 6 continuously measured performance traces. The results on samples of the Fashion MNIST dataset are presented in figure 6.12.

6.2.3 Collective Evaluation

This section compares the estimated performance profiles of the presented anytime learning methods on selected datasets. Initial experiments were performed on samples of the Fashion MNIST dataset, with the results shown in figure 6.13. On this dataset, anytime hyperparameter evolution lead to the worst results of all the methods, although the baseline of an average iteration of random search was surpassed after some time. The performance of progressive model growth and the dynamic version of random search was comparable. The best results were obtained by grid search and the static version of ran-

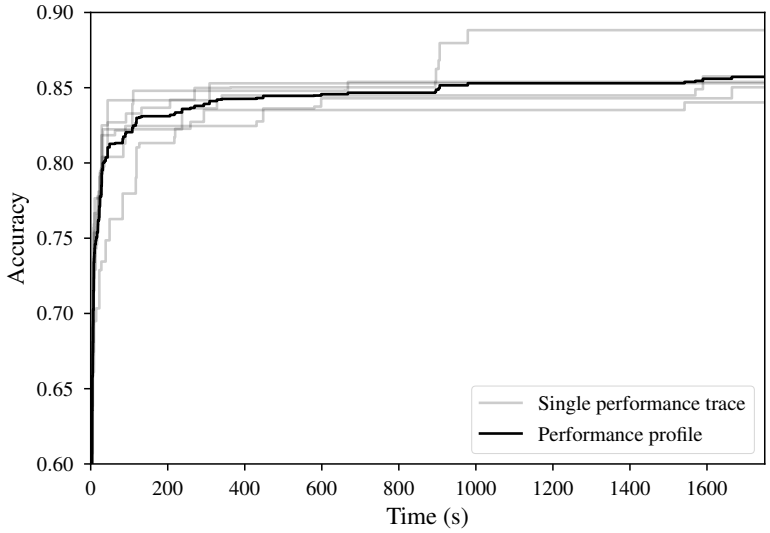

Figure 6.12: Individual test set performance traces and the corresponding performance profile for progressive model growth, trained on samples of the Fashion MNIST dataset.

dom search, with the static version of grid search obtaining slightly better results than the other two methods, although this could be due to statistical error. The observation of dynamic grid search surpassing the performance of dynamic random search is slightly surprising given the theoretical advantages of random search described in section 2.4.2.2. Nevertheless, this result could be due to chance, as it is possible that the choice of the hyperparameter value ranges resulted in favorable sequences of the hyperparameter values sampled by grid search.

The estimated performance profiles of the anytime algorithms on samples of the CIFAR-10 dataset can be found in figure 6.14. Dynamic auto-sizing shows a significant advantage over static variants of random search and anytime grid search, as the decision whether dynamic auto-sizing is used seems to have higher impact on the performance than the choice whether to use random search or anytime grid search. The performance of anytime hyperparameter evolution is comparable to the performance of the static variants of random search and anytime grid search, with the estimated performance profile of evolution slightly outperforming the two techniques at the end of the measured period. The results of progressive model growth are very good throughout the whole experiment and only at the end are surpassed by the dynamic versions of random search and grid search.

As for the 15-puzzle dataset, slightly larger samples of the dataset were used than in the experiments with Fashion MNIST and CIFAR-10. The strength of L1 regularization was allowed in the range of $1 \cdot 10^{-5}$ to approximately $3.2 \cdot 10^{-4}$. The experiments were run on fully-connected models with

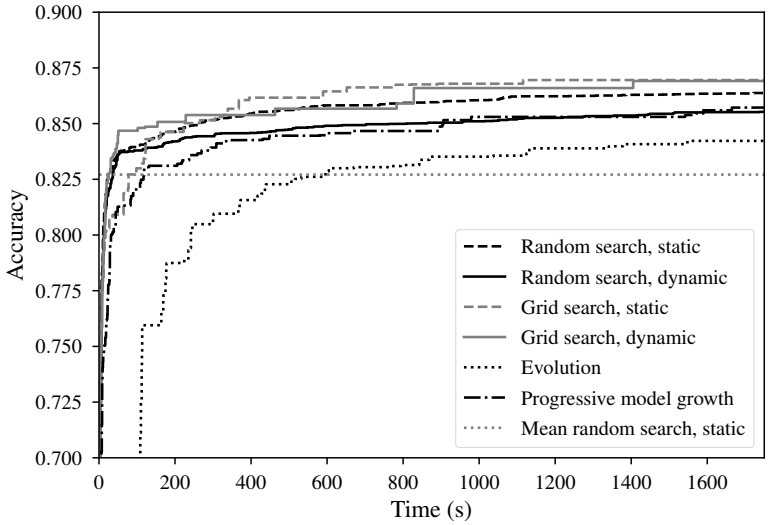

Figure 6.13: Estimated test set performance profiles for various techniques with samples of the Fashion MNIST dataset.

random search and grid search, with dynamic auto-sizing significantly outperforming the static methods. The results are shown in figure 6.15. On this dataset, the performances of random search and grid search are comparable, but the variants utilizing dynamic auto-sizing have a significant advantage over the static variants. Due to low variance in the results of individual algorithms, the results never improve much after the first iteration. The mean RMSE of models obtained by a single iteration of static random search is roughly 9.3, but the mean RMSE of models produced by single iterations of the dynamic version of random search is around 6.9, thus explaining the results.

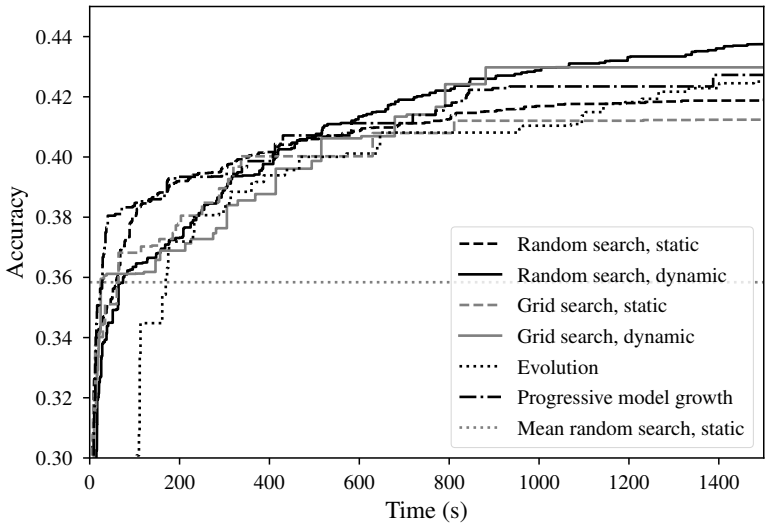

Figure 6.14: Estimated test set performance profiles for various techniques with samples of the CIFAR-10 dataset.

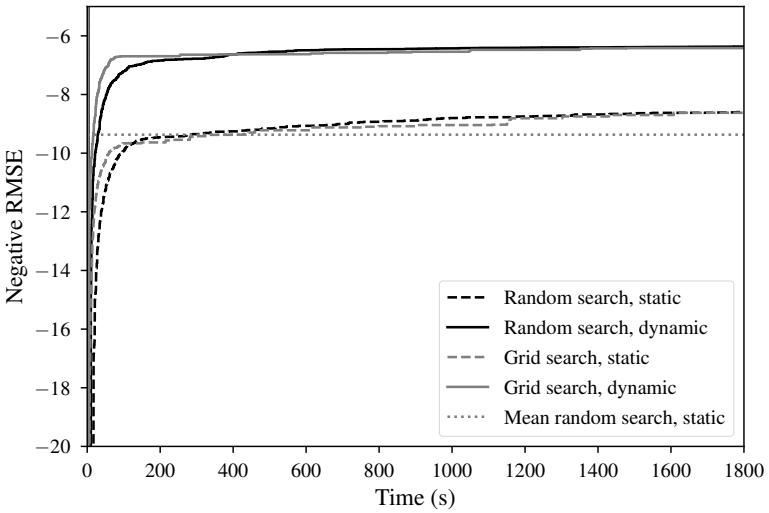

Figure 6.15: Estimated test set performance profiles for various techniques with samples of the 15-puzzle dataset.

Chapter 7

Discussion

The experiments performed in the previous chapter show that dynamic auto-sizing with weighted l_1 regularization offers an interesting AutoML technique that can replace the hyperparameters controlling the regularization strength and hidden layer sizes with a single value. Moreover, dynamic auto-sizing can be regarded as a framework for training deep learning models that is flexible regarding the possible use-cases, as it allows to dynamically change the sizes of models during training by changing the value of the regularization strength hyperparameter. Additionally, it was shown that dynamic auto-sizing models tend to grow with increasing complexity of the problem domain. Most importantly, it was demonstrated that the performance of dynamic auto-sizing models utilizing weighted l_1 regularization is competitive against the performance of models trained using conventional methods as well as dynamic auto-sizing models with underlying $l_{2,1}$ regularization. This result can be somewhat surprising given the unusual nature of weighted l_1 regularization, and the overall results with the novel dynamic auto-sizing technique can be regarded as a success.

As for the performance of the anytime algorithms, it is somewhat disappointing that the results were usually dominated by random search and anytime grid search. Although anytime grid search is a novel algorithm, it is heavily inspired by grid search, a classical hyperparameter optimization technique. On the other hand, anytime grid search shares its stochastic nature with the relatively powerful random search algorithm, thus mitigating the limitations stemming from the curse of dimensionality.

Regarding the anytime hyperparameter evolution and progressive model growth algorithms, although the obtained models were shown to outperform single average iterations of random search, the measured performance profiles were in most cases still worse than with the simpler random search and anytime grid search algorithms. This is unfortunately related to the high computational costs of the experiments, which allowed only limited tuning of the

newly introduced methods[20]. Nonetheless, at least two possible improvements of anytime hyperparameter evolution can be immediately suggested, although neither has yet been experimentally evaluated. The first potential improvement is to avoid using the fixed penalty period specifying at what time since initialization should the fitness of the individuals start being penalized. This could be done using an approach similar to early stopping, which would start penalizing individuals that have not improved for a pre-determined number of epochs. The second potential improvement draws from the behavior of the presented configuration of the algorithm, in which the models of all individuals are trained jointly epoch by epoch, thus proceeding to powerful models only very slowly. In the anytime learning setting, it would be more effective if a single powerful individual was fully trained first. On the other hand, maintaining a small population in which some individuals are significantly older and thus more powerful is problematic and it is thus possible that the anytime hyperparameter evolution technique would have to be altered significantly.

The experiments with anytime learning algorithms were performed mostly on very small datasets. It would thus be interesting to see results with larger models and datasets, especially with the progressive model growth algorithm. However, such experiments would be very computationally intensive and are thus left for another study. Higher computational budgets could further be utilized for sampling the performance profiles from a higher number of performance traces. Aditionally, the anytime learning experiments were run only for moderate durations, and experiments with longer time frames would thus give insights into the long-term behavior of the algorithms.

Another possible improvement could focus on making the dynamic auto-sizing technique less computationally intensive. Dynamic auto-sizing is inherently slower than standard model training approaches in that the additional operations of growth and pruning must be performed. However, for each epoch, newly grown units are grown, and often these are all pruned at the end of the epoch during the pruning step. Therefore, it could be experimented with the values of the hyperparameters λ_{perc} and λ_{min} that control the number of the grown units.

Moreover, weighted l_1 regularization is imperfect as well. One particular issue is that in most cases, the hidden units are stuck with the initial regularization strength for the whole duration of training. More specifically, the units with a smaller index in a given hidden layer are regularized significantly less than the units with a higher index, and there is no possibility of them being assigned a higher amount of regularization during the training. It can be hypothesized that some of these units are less powerful than some of the units with a higher index, and would be naturally pruned if given a higher

[20]Even with the smallest dataset samples, a single generation of anytime hyperparameter evolution took minutes to complete, thus limiting the potential for fine-tuning of the technique.

index. Therefore, it was also experimented with a regularization technique that functioned similarly to weighted l_1 regularization in that newly grown units were given corresponding amounts of regularization, but the regularization strengths for the existing units were shuffled for each epoch so that every unit could be assigned a large regularization strength at some point, giving it the potential to be pruned if ineffective. However, this modified regularization lead to worse performance metrics than the basic version of weighted l_1 regularization, possibly due to the regularization strengths changing too rapidly for the individual units, and the search for improvement thus still remains open.

As stated in chapter 6, fine-tuning of the models (that is, training with omitted regularization) was not used in the anytime learning experiments, as preliminary tests had shown that fine-tuning was not beneficial for the performance of the anytime algorithms. This is a relatively surprising phenomenon, as fine-tuning was shown to provide significant benefits to the performances of both dynamic and static models in section 6.1. On the other hand, fine-tuning can prolong the duration of the training, which would negatively impact the anytime algorithm performance. In any case, this topic is also not concluded and remains for future research.

Chapter 8

Conclusion

The thesis introduced dynamic auto-sizing, an approach to training deep learning models that improves upon original auto-sizing by making it more computationally effective and by enabling the model to grow. In the thesis, it was shown that when coupled with the novel weighted l_1 regularization, dynamic auto-sizing can produce models that surpass the predictive performance of several baseline models. It is plausible that dynamic auto-sizing could also serve as a tool for other engineering tasks besides casual model training, as successful experiments regarding the growth of small models have been presented in the thesis. Moreover, it can be hypothesized that dynamic auto-sizing gives the potential for pruning of deep learning models of unsuitable size, or for adaptation of size of the models to concept drift in on-line or reinforcement learning tasks. In order to support the potential future applications and research, an implementation of dynamic auto-sizing has been released under an open-source license as part of this thesis.

In the anytime learning setting, techniques that utilized dynamic auto-sizing showed performance benefits over classical model training techniques in multiple experiments. Three novel anytime learning algorithms have been designed and evaluated, namely anytime grid search, anytime hyperparameter evolution, and progressive model growth, and it was demonstrated that classical random search can be perceived as an anytime learning algorithm as well. Moreover, although the results of dynamic auto-sizing in the anytime learning setting have been shown to be affected by longer training times, areas for research have been envisioned that could help improve the results in future works.

Bibliography

[1] Murray, K.; Chiang, D. Auto-sizing neural networks: With applications to n-gram language models. *arXiv preprint arXiv:1508.05051*, 2015.

[2] Alvarez, J. M.; Salzmann, M. Learning the number of neurons in deep networks. *Advances in Neural Information Processing Systems*, volume 29, 2016.

[3] Zhou, H.; Alvarez, J. M.; et al. Less is more: Towards compact cnns. In *European conference on computer vision*, Springer, 2016, pp. 662–677.

[4] Murray, K.; Kinnison, J.; et al. Auto-sizing the transformer network: Improving speed, efficiency, and performance for low-resource machine translation. *arXiv preprint arXiv:1910.06717*, 2019.

[5] Gordon, A.; Eban, E.; et al. Morphnet: Fast & simple resource-constrained structure learning of deep networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 1586–1595.

[6] Yuan, X.; Savarese, P.; et al. Growing efficient deep networks by structured continuous sparsification. *arXiv preprint arXiv:2007.15353*, 2020.

[7] Zilberstein, S. Using anytime algorithms in intelligent systems. *AI magazine*, volume 17, no. 3, 1996: pp. 73–73.

[8] Dean, T. L.; Boddy, M. S. An Analysis of Time-Dependent Planning. In *AAAI*, volume 88, 1988, pp. 49–54.

[9] Wellman, M. P.; Liu, C.-L. State-space abstraction for anytime evaluation of probabilistic networks. In *Uncertainty Proceedings 1994*, Elsevier, 1994, pp. 567–574.

[10] Wallace, R. J.; Freuder, E. C. Anytime algorithms for constraint satisfaction and SAT problems. *ACM SIGART Bulletin*, volume 7, no. 2, 1996: pp. 7–10.

[11] Hansen, E. A.; Zhou, R. Anytime heuristic search. *Journal of Artificial Intelligence Research*, volume 28, 2007: pp. 267–297.

[12] Zilberstein, S.; Russell, S. J. Anytime sensing, planning and action: A practical model for robot control. In *IJCAI*, volume 93, 1993, pp. 1402–1407.

[13] Esmeir, S.; Markovitch, S. Anytime Learning of Decision Trees. *Journal of Machine Learning Research*, volume 8, no. 5, 2007.

[14] Svegliato, J.; Sharma, P.; et al. A model-free approach to meta-level control of anytime algorithms. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2020, pp. 11436–11442.

[15] Jesus, A. D.; Liefooghe, A.; et al. Algorithm selection of anytime algorithms. In *Proceedings of the 2020 genetic and evolutionary computation conference*, 2020, pp. 850–858.

[16] Grefenstette, J. J.; Ramsey, C. L. An approach to anytime learning. In *Machine Learning Proceedings 1992*, Elsevier, 1992, pp. 189–195.

[17] Bonarini, A. Anytime learning and adaptation of structured fuzzy behaviors. *Adaptive Behavior*, volume 5, no. 3-4, 1997: pp. 281–315.

[18] Russell, S.; Norvig, P. Artificial intelligence: A modern approach, global edition 4th. 2021.

[19] Deng, L.; Yu, D. Deep learning: methods and applications. *Foundations and trends in signal processing*, volume 7, no. 3–4, 2014: pp. 197–387.

[20] McCulloch, W. S.; Pitts, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, volume 5, no. 4, 1943: pp. 115–133.

[21] Rosenblatt, F. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

[22] Rumelhart, D. E.; Hinton, G. E.; et al. Learning representations by back-propagating errors. *nature*, volume 323, no. 6088, 1986: pp. 533–536.

[23] LeNail, A. NN-SVG: Publication-Ready Neural Network Architecture Schematics. *Journal of Open Source Software*, volume 4, no. 33, 2019: p. 747, doi:10.21105/joss.00747. Available from: `https://doi.org/10.21105/joss.00747`

[24] Géron, A. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems.* " O'Reilly Media, Inc.", 2019.

[25] Cybenko, G. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, volume 2, no. 4, 1989: pp. 303–314.

[26] Hornik, K. Approximation capabilities of multilayer feedforward networks. *Neural networks*, volume 4, no. 2, 1991: pp. 251–257.

[27] Leshno, M.; Lin, V. Y.; et al. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, volume 6, no. 6, 1993: pp. 861–867.

[28] Pinkus, A. Approximation theory of the MLP model in neural networks. *Acta numerica*, volume 8, 1999: pp. 143–195.

[29] Goodfellow, I.; Bengio, Y.; et al. *Deep learning*. MIT press, 2016.

[30] Clevert, D.-A.; Unterthiner, T.; et al. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[31] Klambauer, G.; Unterthiner, T.; et al. Self-normalizing neural networks. *Advances in neural information processing systems*, volume 30, 2017.

[32] LeCun, Y.; Bottou, L.; et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, volume 86, no. 11, 1998: pp. 2278–2324.

[33] Krizhevsky, A.; Sutskever, I.; et al. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, volume 25, 2012.

[34] Simonyan, K.; Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[35] Zeiler, M. D.; Fergus, R. Visualizing and understanding convolutional networks. In *European conference on computer vision*, Springer, 2014, pp. 818–833.

[36] Szegedy, C.; Liu, W.; et al. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[37] He, K.; Zhang, X.; et al. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[38] Long, J.; Shelhamer, E.; et al. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431–3440.

[39] Polyak, B. T. Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics*, volume 4, no. 5, 1964: pp. 1–17.

[40] Duchi, J.; Hazan, E.; et al. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, volume 12, no. 7, 2011.

[41] Dyson, F.; et al. A meeting with Enrico Fermi. *Nature*, volume 427, no. 6972, 2004: pp. 297–297.

[42] Yuan, M.; Lin, Y. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, volume 68, no. 1, 2006: pp. 49–67.

[43] Huang, J.; Zhang, T.; et al. Learning with Structured Sparsity. *Journal of Machine Learning Research*, volume 12, no. 11, 2011.

[44] Srivastava, N.; Hinton, G.; et al. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, volume 15, no. 1, 2014: pp. 1929–1958.

[45] Bergstra, J.; Bengio, Y. Random search for hyper-parameter optimization. *Journal of machine learning research*, volume 13, no. 2, 2012.

[46] Back, T. Selective pressure in evolutionary algorithms: A characterization of selection mechanisms. In *Proceedings of the first IEEE conference on evolutionary computation. IEEE World Congress on Computational Intelligence*, IEEE, 1994, pp. 57–62.

[47] Yu, X.; Gen, M. *Introduction to evolutionary algorithms*. Springer Science & Business Media, 2010.

[48] Bäck, T.; Schwefel, H.-P. An overview of evolutionary algorithms for parameter optimization. *Evolutionary computation*, volume 1, no. 1, 1993: pp. 1–23.

[49] Holland, J. H. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[50] Turing, A. M. Computing machinery and intelligence. In *Parsing the turing test*, Springer, 2009, pp. 23–65.

[51] Barricelli, N. A. *Symbiogenetic evolution processes realized by artificial methods*. 1957.

[52] Fraser, A. S. Simulation of genetic systems by automatic digital computers I. Introduction. *Australian journal of biological sciences*, volume 10, no. 4, 1957: pp. 484–491.

[53] Whitley, L. D.; et al. *The GENITOR algorithm and selection pressure: why rank-based allocation of reproductive trials is best.* Citeseer, 1989.

[54] Yang, J.; Soh, C. K. Structural optimization by genetic algorithms with tournament selection. *Journal of computing in civil engineering*, volume 11, no. 3, 1997: pp. 195–200.

[55] Beyer, H. Evolution strategies. *Scholarpedia*, volume 2, no. 8, 2007: p. 1965, doi:10.4249/scholarpedia.1965, revision #193589.

[56] Ramsey, C. L.; Grefenstette, J. J. Case-based anytime learning. In *Case Based Reasoning: Papers from the 1994 Workshop*, AAAI Press Menlo Park, California, 1994, pp. 91–95.

[57] Parker, G. B.; Mills, J. W. Adaptive hexapod gait control using anytime learning with fitness biasing. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 1*, Citeseer, 1999, pp. 519–524.

[58] Parker, G. B. Co-evolving model parameters for anytime learning in evolutionary robotics. *Robotics and Autonomous Systems*, volume 33, no. 1, 2000: pp. 13–30.

[59] Parker, G. B. Punctuated anytime learning for hexapod gait generation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, IEEE, 2002, pp. 2664–2671.

[60] Esmeir, S.; Markovitch, S. Anytime learning of anycost classifiers. *Machine Learning*, volume 82, no. 3, 2011: pp. 445–473.

[61] Seidl, T.; Assent, I.; et al. Indexing density models for incremental learning and anytime classification on data streams. In *Proceedings of the 12th international conference on extending database technology: advances in database technology*, 2009, pp. 311–322.

[62] Xu, Z.; Kusner, M.; et al. Anytime representation learning. In *International Conference on Machine Learning*, PMLR, 2013, pp. 1076–1084.

[63] LeCun, Y.; Denker, J.; et al. Optimal brain damage. *Advances in neural information processing systems*, volume 2, 1989.

[64] Tibshirani, R. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, volume 58, no. 1, 1996: pp. 267–288.

[65] Han, S.; Pool, J.; et al. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, volume 28, 2015.

[66] Han, S.; Mao, H.; et al. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[67] Yang, J.; Ma, J. Feed-forward neural network training using sparse representation. *Expert Systems with Applications*, volume 116, 2019: pp. 255–264.

[68] Wen, W.; Wu, C.; et al. Learning structured sparsity in deep neural networks. *Advances in neural information processing systems*, volume 29, 2016.

[69] Lin, J.; Rao, Y.; et al. Runtime neural pruning. *Advances in neural information processing systems*, volume 30, 2017.

[70] Ioffe, S.; Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, PMLR, 2015, pp. 448–456.

[71] Kordík, P.; Koutník, J.; et al. Meta-learning approach to neural network optimization. *Neural Networks*, volume 23, no. 4, 2010: pp. 568–582.

[72] Stanley, K. O.; Miikkulainen, R. Evolving neural networks through augmenting topologies. *Evolutionary computation*, volume 10, no. 2, 2002: pp. 99–127.

[73] Stanley, K. O.; D'Ambrosio, D. B.; et al. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, volume 15, no. 2, 2009: pp. 185–212.

[74] Miikkulainen, R.; Liang, J.; et al. Evolving deep neural networks. In *Artificial intelligence in the age of neural networks and brain computing*, Elsevier, 2019, pp. 293–312.

[75] Gomez, F. J.; Miikkulainen, R.; et al. Solving non-Markovian control tasks with neuroevolution. In *IJCAI*, volume 99, 1999, pp. 1356–1361.

[76] Kordík, P.; Černỳ, J.; et al. Discovering predictive ensembles for transfer learning and meta-learning. *Machine learning*, volume 107, no. 1, 2018: pp. 177–207.

[77] Yao, Q.; Wang, M.; et al. Taking human out of learning applications: A survey on automated machine learning. *arXiv preprint arXiv:1810.13306*, 2018.

[78] He, X.; Zhao, K.; et al. AutoML: A survey of the state-of-the-art. *Knowledge-Based Systems*, volume 212, 2021: p. 106622.

[79] Olson, R. S.; Bartley, N.; et al. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the genetic and evolutionary computation conference 2016*, 2016, pp. 485–492.

[80] Feurer, M.; Klein, A.; et al. Efficient and Robust Automated Machine Learning. In *Advances in Neural Information Processing Systems 28 (2015)*, 2015, pp. 2962–2970.

[81] Jin, H.; Song, Q.; et al. Auto-Keras: An Efficient Neural Architecture Search System. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ACM, 2019, pp. 1946–1956.

[82] Microsoft. Neural Network Intelligence. 1 2021. Available from: `https://github.com/microsoft/nni`

[83] Liu, H.; Simonyan, K.; et al. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[84] Hutter, F.; Hoos, H. H.; et al. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, Springer, 2011, pp. 507–523.

[85] Li, L.; Jamieson, K.; et al. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, volume 18, no. 1, 2017: pp. 6765–6816.

[86] Abadi, M.; Agarwal, A.; et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[87] LeCun, Y. A.; Bottou, L.; et al. Efficient backprop. In *Neural networks: Tricks of the trade*, Springer, 2012, pp. 9–48.

[88] Klambauer, G.; Unterthiner, T.; et al. Self-normalizing neural networks. *Advances in neural information processing systems*, volume 30, 2017.

[89] Cahlik, V.; Surynek, P. Near Optimal Solving of the (N2-1)-puzzle Using Heuristics Based on Artificial Neural Networks. In *International Joint Conference on Computational Intelligence*, Springer, 2019, pp. 291–312.

[90] Netzer, Y.; Wang, T.; et al. Reading digits in natural images with unsupervised feature learning. 2011.

[91] Le, Y.; Yang, X. Tiny imagenet visual recognition challenge. *CS 231N*, volume 7, no. 7, 2015: p. 3.

Appendix A

Acronyms

AdaGrad Adaptive Gradient algorithm

Adam Adaptive moment estimation

AutoML Automated Machine Learning

CIFAR Canadian Institute For Advanced Research

CPU Central Processing Unit

DARTS Differentiable ARchiTecture Search

ELU Exponential Linear Unit

ES Evolution Strategy

ESP Enforced Sub-Populations

FLOPS FLoating point OPerations per Second

GPU Graphics Processing Unit

KL divergence Kullback-Leibler divergence

MNIST Modified National Institute of Standards and Technology

MSE Mean Squared Error

NAS Neural Architecture Search

NEAT NeuroEvolution of Augmenting Topologies

NNI Neural Network Intelligence

RAM Random Access Memory

ReLU Rectified Linear Unit

RGB Red-Green-Blue

RMSE Root Mean Squared Error

RMSProp Root Mean Square Propagation

SANE Symbiotic, Adaptive Neuro-Evolution

SELU Scaled Exponential Linear Unit

SVHN Street View House Number

TPOT Tree-based Pipeline Optimization Tool

TWEANN Topology and Weight Evolving Artificial Neural Networks

Appendix B

Contents of enclosed CD

```
README.md ..... description of contents
LICENSE ..... project license
requirements.txt ..... Python requirements
nets ..... Python package containing implementations
notebooks ..... directory with Jupyter notebooks with experiments
thesis.pdf ..... the thesis in the PDF format
```