Transfer Learning in Autonomous Vehicles Using Convolutional Networks

Gustav Lundberg

Master's thesis
2019:E1

Lund University

Faculty of Engineering
Centre for Mathematical Sciences
Mathematics

CENTRUM SCIENTIARUM MATHEMATICARUM

Transfer Learning in Autonomous Vehicles using Convolutional Networks

Master's Dissertation by

Gustav Lundberg

Supervisor:
Alexandros Sopasakis, Department of Mathematics, Numerical Analysis

Examiner:
Mattias Ohlsson, Department of Theoretical Physics

1 Abstract

This thesis has investigated the potential benefits of using transfer learning when training convolutional neural networks for the task of autonomously driving a car. Three transfer learning networks were trained and compared with a conventionally trained convolutional neural network. The first conclusion is that training, as expected, is considerably quicker when using transfer learning. More specifically, transfer learning utilizes pretrained networks and does not require the entire network to be trained, rather just parts of it. This results in faster training since less weights have to be updated per epoch of training. Here, training was approximately twice as fast per epoch compared to training a non-transfer learning network. The pretrained network used in this thesis was in fact also trained, something that in a practical transfer learning setting would already have been done. The data used to train the so called pretrained network originates from a different, although similar, domain.

By keeping a varying number of convolutional layers from the pretrained network fixed, three networks were trained using transfer learning. This way, knowledge is said to be transferred from a previously solved problem to the current problem. The network called fine-tuning network 2 performed better than the two other transfer learning networks when tested in the simulator. After only 7 epochs of training, this network could drive around the entire track without going off-road, even at higher speeds. This is to be compared with the so called conventional network, trained under a classic machine learning setting. Not until having been trained for 18 epochs did the performance of the conventional network match the performance of the fine-tuning network 2, trained for merely 7 epochs. In other words, in less than half the number of epochs, and with each epoch training twice as fast, a network using transfer learning performed as well as a non-transfer learning network.

Contents

1 Abstract **1**

2 Introduction **4**
2.1 Problem formulation . . . 4

3 Theory **5**
3.1 Basic concepts of machine learning . . . 5
3.1.1 Task . . . 5
3.1.2 Performance . . . 5
3.1.3 Experience . . . 5
3.2 Artificial neural networks . . . 6
3.3 Training an ANN . . . 8
3.3.1 Optimization and gradient descent . . . 8
3.3.2 Non-convex optimization . . . 9
3.3.3 Other optimizers . . . 10
3.3.4 Hyperparameter-tuning . . . 10
3.3.5 Common hyperparameters related to optimization . . . 10
3.3.6 Vanishing and exploding gradient problem . . . 11
3.3.7 Activation functions . . . 11
3.4 Regularization . . . 12
3.4.1 Bias-variance trade-off . . . 12
3.4.2 Overfitting and underfitting . . . 13
3.4.3 L2 regularization . . . 14
3.4.4 Early stopping . . . 14
3.4.5 Dropout . . . 15
3.5 Convolutional neural networks . . . 15
3.5.1 The convolutional layer . . . 15
3.5.2 Hyperparameters of CNNs . . . 16
3.5.3 Advantages of CNNs . . . 16
3.5.4 Pooling . . . 17
3.5.5 Architecture of a typical CNN . . . 17
3.6 Datasets . . . 18
3.6.1 Preprocessing . . . 18
3.6.2 Normalizing the inputs . . . 18
3.6.3 Data augmentation . . . 19
3.6.4 Batch normalization . . . 19
3.7 Transfer learning . . . 19
3.7.1 Transferring knowledge with CNNs . . . 20

4 Methodology **22**
4.1 Software . . . 22
4.2 Data . . . 22
4.2.1 Lake track . . . 22
4.2.2 Jungle track . . . 22
4.2.3 Data collection . . . 22
4.2.4 Data specifications . . . 22
4.2.5 Transfer learning view . . . 23
4.3 Preprocessing and data augmentations . . . 23
4.3.1 Distribution of steering angles . . . 24

4.4 Network architecture and hyperparameters . . . 25
4.5 Tasks . . . 27
4.5.1 Task 1 - Training on the lake dataset . . . 27
4.5.2 Task 2 - Training on the jungle dataset/Transfer learning . . . 27
4.5.3 Task 3 - Training a network for comparison . . . 27
4.5.4 Evaluating the performance in the simulator . . . 27

5 Results **28**
5.1 Task 1 . . . 28
5.2 Task 2 . . . 29
5.2.1 Feature extraction . . . 29
5.2.2 Fine-tuning network 1 . . . 31
5.2.3 Fine-tuning network 2 . . . 31
5.3 Task 3 . . . 32

6 Discussion **32**
6.1 Data . . . 32
6.2 Network architecture . . . 33
6.3 Task 1 . . . 33
6.4 Task 2 . . . 33
6.4.1 Feature extraction network . . . 33
6.4.2 Fine-tuning network 1 . . . 34
6.4.3 Fine-tuning network 2 . . . 34
6.5 Task 3 . . . 35

7 Conslusions **35**

8 Further work **36**

9 References **37**

2 Introduction

The increase in computer power and access to large datasets are two important factors that have helped contribute to drive the rapid development of powerful machine learning applications [24][15]. However, in many applications the amount of data and training time is still limited. An interesting method that aims at alleviating these problems is that of transfer learning [16][21][33]. The idea is that knowledge from a similar, previously solved problem should be useful in a new but related problem. In the context of neural networks, one wishes to not having to train a network from scratch on the new problem, rather continue training the network from the previous problem, or a modified version of it.

Transfer learning is inspired by the human ability to learn new concepts quicker if we already possess knowledge about a similar concept. Consider for example the process of learning to drive a bus and how much easier that would be if you already know how to drive a car. For humans, a closer relation between the old and new concept makes the learning easier. Analogously, this property generally holds for transfer learning too, where problems only distantly related may not work well with standard transfer learning techniques [31].

The aim of this thesis is to investigate the possible benefits of using transfer learning to train convolutional networks for the task of autonomously driving a car. Knowledge is transferred from a related problem with the same task. The difference lies in the driving environments, where the old problem deals with a slightly less complex environment. To solve the old problem a convolutional neural network is trained with data from this simpler environment. This network is called a pretrained network which is then further trained on the new driving environment, thus transferring knowledge from the old to the new problem. The goal is to achieve better or equal performance with less amount of training, compared to training a network using conventional, i.e. non-transfer, machine learning techniques.

2.1 Problem formulation

The experiments conducted in this thesis are arranged into three main parts, or tasks.

Task 1 - Pretraining
The first part consists of training a network from scratch with data from an environment different from the testing environment. Such a network is referred to as a pretrained network and is further used in task 2.

Task 2 - Transfer learning
In the second part, transfer learning is applied by making use of the pretrained network from the previous task. Using data from the testing environment, the pretrained network from task 1 is further trained with the goal of being able to autonomously drive in this new environment.

Task 3 - Training a network for comparison
To evaluate the so called transfer learning networks from task 2, another network is trained. This network is trained from scratch using only data from the testing environment, i.e. no pretraining is used. It is then to be compared to the transfer learning networks, both in terms of performance on the testing environment and training time.

3 Theory

3.1 Basic concepts of machine learning

Machine learning is a field of artificial intelligence that aims at providing computer systems the ability to learn from data, without being explicitly programmed. It is a multidisciplinary field, combining ideas and results from mainly statistics and computer science. Including steps such as data processing, prediction and decision-making, machine learning is also said to be an algorithmic field.

Let us start with defining what is meant by learning. In his book *Machine Learning*, Tom M. Mitchell defines learning as follows [17],

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

3.1.1 Task

The task refers to what the computer is intended to do after having learned. For example, teaching a computer to play chess simply means that the task is *playing chess*. There are a vast number of tasks that can be learned through machine learning. Two of the most common types of tasks are classification and regression.

As the name suggests, classification deals with classes or categories. Given a dataset, i.e. a collection of data, the computer tries to learn to predict what category the input belongs to. For example, given an image as input the computer might say what object is depicted. This is known as object detection and is a common classification task.

On the other hand, regression is when the computer for a given input tries to predict a numerical value, as opposed to a category. Again, with images as inputs, this time from a forward facing camera of a car, the output could be the steering angle and/or the speed of the car.

3.1.2 Performance

To be able to tell if a computer has learned or not there must be some sort of performance measure. There are plenty of such measures and the choice of measure depends on the task. A common choice for classification tasks is called accuracy which simply is the ratio between the number of correctly classified data points and the total number of data points. For regression a common choice of performance measure is the mean squared error (MSE). This is the average of the squares of the errors, where the error is the difference between the predicted output and the actual output.

3.1.3 Experience

Broadly speaking, learning can be categorized into supervised, unsupervised and reinforcement learning [23].

In supervised learning the dataset D consists of input-output pairs, $D = (\boldsymbol{x}(n), \boldsymbol{y}(n))$ where $n = 1, ..., N$. One such pair is called an example and consists of an input $\boldsymbol{x}(n)$ and an output $\boldsymbol{y}(n)$, also known as label or target. Note that the input and output are written as vectors, but these could be scalars or of any other dimension. For a dataset containing images of cats and dogs, the input would be an image and the output target the string "cat" or "dog", or (0/1). The observed output $\boldsymbol{y}$ in the dataset comes from some unknown function $\mathbf{y} = f(\mathbf{x})$. The task for the learning algorithm is to find a function h that approximates this function f. In machine learning, h is commonly modelled with a neural network. Given an input to the network, the value the network outputs is called a prediction.

Unsupervised learning deals with datasets where the examples contain only an input $\boldsymbol{x}(n)$ but no corresponding label $\boldsymbol{y}(n)$. This kind of learning aims at finding useful properties of the dataset and includes tasks such as density estimation and clustering. In the former, the goal is to find an estimate of the underlying probability distribution $f(\mathbf{x})$, given the examples in the observed dataset. A common method for clustering tasks is the simple k-means clustering algorithm, which groups the dataset into k clusters of similar examples.

Reinforcement learning is a third type of learning that revolves around an agent and an environment. Via algorithms, the agent learns to make decisions in the environment. During training of the agent, it interacts with the environment by taking actions and then perceiving the results of those actions. Goals are set up so that there is a measure of how well the agent acts, giving it rewards or punishments. An area where reinforcement learning is common is in games, due to the fact that simulations of the game easily can be executed a large number of times, which is essential for successful learning.

3.2 Artificial neural networks

Lately, astonishing results have been achieved in the field of machine learning using a type of model called an artificial neural network (ANN), and particularly deep ANN's [24][15]. As the name suggests it is inspired from the biological neural network that makes up the human brain. Figure 1 shows a simple ANN. This network consists of only one neuron, or node, that makes mathematical computations from input x_k's to output y according to

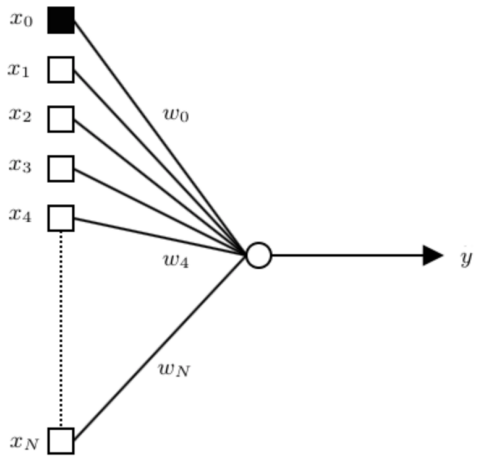


Figure 1: The simplest form of an artificial neural network containing only one neuron, represented as a circle. The squares represent the input and the lines connecting them to the neuron represent the weights. Note that only a few weights have been written out to avoid a cluttered figure. The black square represents the bias node. The output y is computed using Equation (1).

$$y = \varphi\big(\sum_{k=1}^{N} w_k x_k + w_0\big) = \varphi\big(\sum_{k=0}^{N} w_k x_k\big). \tag{1}$$

Here w_k's are called the weights or parameters of the network. Specifically, w_0 is called a bias and can be put inside the sum by letting $x_0 = 1$. φ is referred to as an activation function and could be various functions depending on the problem. They are usually non-linear and it is up to the designer of the network to specify one. More about activation functions in Section 3.3.7.

By connecting neurons to each other it is possible to build large networks containing numerous neurons. These networks are commonly structured in layers where the entire network still takes an example as input and outputs a prediction in the final layer of the network. A typical network can look something like the one in Figure 2. The first layer is the input layer and the last one the output layer. The other layers are called hidden layers. A network containing only connections going forward in the structure forms a feed-forward network. On the contrary, networks with backwards connections are called recurrent networks and have connections forming a circle. For recurrent networks, one should specify the directions of the connections with arrows. In Figure 2 the arrows have been left out, implying that there are only feed-forward connections and thus it is a feed-forward network. Only feed-forward networks will be considered in this report.

Another useful term is fully-connected networks, referring to networks with only fully-connected layers. Such a layer has connections from each node in the previous layer to *each* node in the next layer.

The depth of a network refers to the number of layers it contains. The network in Figure 2 has a depth of two - one hidden layer and one output layer (the input layer does not count). Nowadays, with the increase of computational resources and the development of new algorithms and optimization techniques, very deep networks may contain over a hundred layers [27].

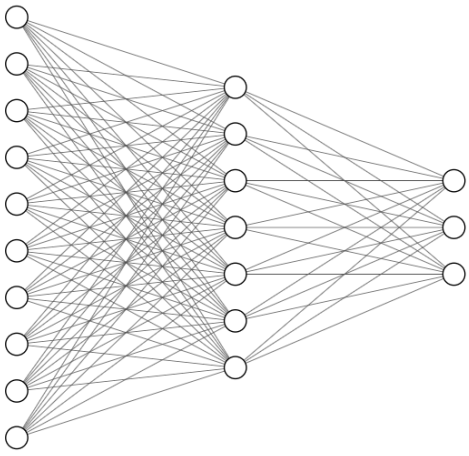


Figure 2: A fully-connected feed-forward network containing one hidden layer.

The output of the feed-forward network in Figure 2 is computed using Equation (1) for all the neurons in the network. Component i of the output is then given by

$$y_i(n) = \varphi_o\Big(\sum_{j=0}^{J} w_{ij}\varphi_h\big(\sum_{k=0}^{K}\tilde{w}_{jk}x_k(n)\big)\Big), \qquad \forall i = 1, ..., M \ \text{ and } \ n = 1, ..., N \tag{2}$$

where M is the number of neurons in the output layer, K the dimension of the input and J the number of neurons in the hidden layer. The weights from the input-to-hidden and hidden-to-output layer are denoted $\tilde{w}_{jk}$ and w_{ij} respectively. Note here that n specifies that example n is being used, and N refers to the total number of examples in the dataset. Also note that the biases have not been explicitly written out and that there may be different activation functions in different layers. φ_o and φ_h are the activation functions in the output and hidden layer respectively.

From equation 2 we notice that the ANN in figure 2 represents a function $f : \mathbb{R}^K \to \mathbb{R}^M$. For a neural network to be of any use, it is of importance that there are no fundamental limitations built into them, restricting their general applicability. It turns out that an ANN with at least one

hidden layer and a finite number of neurons is able to approximate *any* continuous function. This is true under mild assumptions on the activation functions. Details are left out here but can be found in the so called universal approximation theorem [10].

3.3 Training an ANN

So how does an ANN actually learn from data? The goal of the network is to learn patterns from the data. Having learned these patterns it should be able to generalize and make predictions about new data the network has not seen before, i.e. during training. Therefore, it is common practice to randomly split the dataset into three parts - a training, a validation and a test set. The training set is used during training, and if the split has been done randomly, the hope is that if there is good performance on the training set this will also carry over to the test set. If this is the case, the network is said to have good generalization performance. The validation set is used indirectly during training, meaning that the network will not actually see this set during training. Rather, it is used to validate the adequacy of the hyperparameter values. Hyperparameters are parameters non-trainable to the network, thus needing to be set manually. More about hyperparameters in Section 3.3.4 and 3.3.5. Finally, after training is complete, the test set is used to evaluate the generalization performance of the network. This set is thus only used in the very end, when the network will not be further trained or modified in any way. This way, it will give an unbiased measure of the true generalization performance of the network, compared to evaluating on the validation (or training) set.

To formulate the learning algorithm, an error function E has to be chosen, sometimes referred to as a loss function L. A common choice for regression tasks is the MSE. Given a dataset D, the algorithm tries to minimize the error E produced by the examples from the dataset. The minimization is done by updating the weights of the network via some optimization algorithm.

3.3.1 Optimization and gradient descent

There are several options when choosing an optimization method. Arguably the most common one is called gradient descent. In short, the procedure goes as follows. Given an input, the output is first computed by propagating the signal (input) forward through the network. This step is called the forward pass. Once the output has been computed the error for that example can be calculated, using the error function of choice. The next step is called backpropagation and involves computing the gradient of the error function with respect to the weights. An update rule for the weights is now applied. Updating the weights using only one input, as just described, is called on-line updating. Other variants of gradient descent use either all examples before updating (batch-updating) or a so called mini-batch of typically between 10 and a few hundred examples [1]. The gradient descent method using on-line updating or mini-batches is referred to as stochastic gradient descent, since a fraction of the examples are chosen at random for each update [39].

Let us look at the optimization procedure in more detail for a fully-connected feed-forward network with one hidden layer using the batch-updating rule. The error function E using MSE is given by

$$E = \frac{1}{2N} \sum_{n=1}^{N} \sum_{i} (d_i(n) - y_i(n))^2, \tag{3}$$

where $y_i(n)$ is the i :th component of the output from the network, using the n:th example as input. Similarly, $d_i(n)$ is the i :th component of the true label of the n:th input. Notice here that the error function has been normalized with $1/2$ for convenience. Gradient descent then uses the update rules

$$\Delta\tilde{w}_{jk} = -\eta\frac{\partial E}{\partial \tilde{w}_{jk}} \tag{4}$$

$$\Delta w_{ik} = -\eta\frac{\partial E}{\partial w_{ik}} \tag{5}$$

for the input-to-hidden and the hidden-to-output weights respectively. Here, η is the learning rate, a hyperparameter controlling the magnitude of the weight updates. A large learning rate means that the optimizer takes large steps in the direction of the negative gradient, intuitively increasing the speed of the training. However, too large of a learning rate may result in the training not converging because it keeps overshooting the desired minimum or even worse, the error may diverge [30]. Hence, this hyperparameter needs to be tuned to find the right balance between speed and reliability of the optimization.

Let us denote the output from node j of the hidden layer $h_j(n) = \varphi_h\big(\sum_k \tilde{w}_{jk}x_k(n)\big)$. The gradients in Equation (4) and (5) can then be calculated [20] as

$$\frac{\partial E}{\partial w_{ik}} = -\frac{1}{N}\sum_n \underbrace{(d_i(n) - y_i(n))\varphi'\bigg(\sum_j w_{ij}h_j(n)\bigg)}_{\delta_i(n)} h_j(n) = -\frac{1}{N}\sum_n \delta_i(n)h_j(n) \tag{6}$$

and

$$\frac{\partial E}{\partial \tilde{w}_{jk}} = -\frac{1}{N}\sum_{n,i}\delta_i(n)w_{ij}\varphi_h'\bigg(\sum_{k'}\tilde{w}_{jk'}x_{k'}(n)\bigg)x_k(n) = -\frac{1}{N}\sum_n \tilde{\delta}_j(n)x_k(n), \tag{7}$$

where $\tilde{\delta}_j(n)$ in Equation (7) is defined as

$$\tilde{\delta}_j(n) = \varphi_h'\bigg(\sum_{k'}\tilde{w}_{jk'}x_{k'}(n)\bigg)\sum_i \delta_i(n)w_{ij}. \tag{8}$$

3.3.2 Non-convex optimization

Minimizing the error function E for a neural network in general is a non-convex optimization problem [36, 4]. As opposed to convex functions, non-convex functions do not just have a global minimum, but rather one or more multiple local minima. In practice, the error function commonly has a large number of local minima [4]. For many problems, hoping to find the global minimum then empirically proves to be unfeasible [36]. Instead, one may only wish to find a *good* local minimum, hopefully still leading to good generalization performance.

Furthermore, non-convex optimization needs to deal with saddle points, i.e. avoid getting stuck in them. One way of dealing with both saddle points and bad local minima is to avoid using the batch-update rule with the gradient descent method. Suppose the algorithm is stuck in a stationary point. The gradient information from the "full" error function, i.e. the error function using all examples will not help to escape the stationary point, since the gradient by definition is zero in a stationary point. Using the batch-updating rule for gradient descent would correspond to using the full error function. Instead of averaging the error function over all examples, one may calculate the gradients using only a fraction of the examples. This corresponds to using stochastic gradient descent. The stationary points of this "reduced" error function, likely will not coincide with those of the full error function. Intuitively, stochastic gradient descent may thus prevent the optimization algorithm from getting stuck in stationary points and empirical studies proves that it generally finds better solutions than the batch-updating gradient descent [13].

A key consideration regarding the optimization process is the one of learning rate [13]. The relation between training time and learning rate has already been mentioned. However, the learning rate also affects what solution the algorithm will find. For example, a larger learning rate has the ability to escape local minima whereas a smaller learning rate may lead to the algorithm getting stuck [13]. Therefore, a common approach is to have a flexible learning rate, allowing it to change when needed. Usually this means having a larger learning rate initially to avoid bad local minima, and then lowering it over time when getting closer to better minima.

3.3.3 Other optimizers

There are a number of alternative optimization methods that can be used when minimizing the error function. Some of these are straight up attempts at improving the basic gradient descent method, while so called second order methods use local quadratic approximations of the error function. The latter requires computing the Hessian matrix which in turn can be used to avoid getting stuck in saddle points. According to Dauphin et al. the biggest challenge when optimizing a deep neural network is in fact how to escape saddle points, not local minima [7]. One reason for this, they argue, is that in higher-dimensional problems, the ratio between the number of saddle points and the number of local minima increase exponentially with the dimensionality. A drawback of dealing with saddle points using second order methods is that iteratively computing the Hessian is resource-intensive. A remedy for this, as suggested by Reddi et al. [22] is to use a first order method except when being close to a saddle point. Here the Hessian information would be used, thus reducing the computational cost compared to a pure second order method.

One of the most common optimizers used recently for training deep neural networks is the so called Adam optimizer [12]. Adam is a first order method. Compared to gradient descent, which has a common learning rate, Adam uses individual adaptive learning rates for the weights. This is done by keeping running averages of the past gradient and squared gradient for each weight. Empirically, Adam has been found to produce great results in a wide range of non-convex optimization problems [12] which is why it has become so popular.

3.3.4 Hyperparameter-tuning

The optimization of a neural network requires choosing good values for the hyperparameters. The process of finding good values is called hyperparameter-tuning.

The most basic method is to search manually, i.e. making qualified guesses to the extent possible. For each set of hyperparameter values, the network is trained and evaluated. The procedure is halted when satisfying results are achieved.

A more structured way of doing hyperparameter-tuning is known as grid search. Here, each hyperparameter is given a set of values to be tested, e.g. the learning rate may be given the values $\eta = \{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$. The network is then trained and evaluated for each combination of hyperparameter values. The network giving the best results is chosen as the final network.

Another method that seems to be working better or at least as good as grid search is known as random search [2]. Here, the values of the hyperparameters are drawn from some distribution over the same or a similar domain as during grid search.

Other more sophisticated methods exist, such as bayesian optimization [28], although that will not be presented here.

3.3.5 Common hyperparameters related to optimization

Common hyperparameters related to the optimization includes number of epochs, batch size, learning rate, samples per epoch and test size. A brief description of these follows below.

During optimization, when the algorithm has processed the entire training dataset once, it is said that training has progressed for one epoch. Commonly, each example has to be used multiple times during training in order to achieve satisfying network performance. The number of times the dataset is presented to the algorithm is thus a hyperparameter referred to as the number of epochs.

Batch size determines how many examples are to be included in one batch of gradient descent, see Section 3.3.1. A batch size of one means that the weights are updated using gradient information from a single example, i.e. on-line updating. On the contrary, a batch size greater than one means the gradient information is averaged over multiple examples.

Learning rate has indeed been introduced above in Section 3.3.1 and 3.3.2 and as stated it determines the magnitude of the weight updates during optimization.

Samples per epoch is a hyperparameter related to the technique of augmentation, see Section 3.6.3. This technique takes examples from the training dataset and makes small modifications to them, thereby synthetically creating more data. Samples per epoch determines how many augmented examples there should be in one epoch. Note that when no augmentation is used, samples per epoch is not relevant since it would just correspond to the total number of examples in the training dataset.

As stated in Section 3.3, the dataset is commonly split into three parts referred to as the training, validation and test set. The hyperparameter known as test size decides the way the split is done, i.e. the ratio of the sizes between the sets.

3.3.6 Vanishing and exploding gradient problem

Before delving into the particular choice of activation functions, a potential problem will first be illustrated that one may face when training deep neural networks with backpropagation. The problem is called the vanishing or exploding gradient problem and is increasingly severe the deeper the network is. The reason for this being that the gradients that are backpropagated through the network are multiplied together an increasing number of times the deeper the network is. This may result in very slow training (vanishing gradient problem) or unstable training (exploding gradient problem), depending on the norm of the gradients that are being multiplied together [9].

3.3.7 Activation functions

The activation function φ acts on a node according to Equation (1). Traditional choices [32] of activation function in the hidden layers include the logistic function and the hyperbolic tangent function, given by

$$f(x) = \frac{1}{1 + e^{-x}} \in (0, 1) \tag{9}$$

and

$$f(x) = \tanh(x) \in (-1, 1) \tag{10}$$

respectively. The logistic function has a non-zero mean and nodes with this activation function will thus act as biases for the next layer. If these biases are not treated in any way, it may cause learning to slow down [6]. Thus, the hyperbolic tangent function, having zero mean, has been suggested as a replacement. However, the problem of vanishing gradients is present with both of these functions. This is realized by simply differentiating Equation (9) and (10), which shows that their gradients are close to zero over a large region of the domain, meaning the vanishing gradient problem is indeed present.

An attempt at addressing this problem is by choosing an activation function called ELU. This is an acronym for exponential linear units and it is given by

$$f(x,\alpha) = \begin{cases} \alpha(e^x - 1), & x \leq 0 \\ x, & x > 0 \end{cases} \tag{11}$$

This activation function has a mean close to zero, thereby minimizing the bias shift experienced with the logistic and hyberbolic tangent functions. Furthermore, ELU alleviates the vanishing gradient problem by having a gradient equal to one for *any* positive value, as opposed to close to zero for the previously mentioned activation functions [6].

3.4 Regularization

3.4.1 Bias-variance trade-off

Assume that the observable d_k takes the form $d_k(\mathbf{x}) = f_k(\mathbf{x}) + \epsilon_k$, where f_k is an unknown function one seeks to learn and ϵ is intrinsic noise, commonly modelled by a Gaussian distribution. Under this setting, using the squared loss $L = L(d, y(\mathbf{x})) = (y(\mathbf{x}) - d)^2$ as the error function, the expected loss $\mathbb{E}[L]$ takes the form

$$\mathbb{E}[L] = \iint L(d, y(\mathbf{x}))p(\mathbf{x}, d)\mathrm{d}\mathbf{x}\mathrm{d}d, \tag{12}$$

where $p(\mathbf{x}, d)$ is the joint probability distribution over the input and the observable. Following the derivation done by Bishop [4], the loss can be written as

$$\begin{aligned} L(d, y(\mathbf{x})) &= (y(\mathbf{x}) - d)^2 = (y(\mathbf{x}) - \mathbb{E}[d|\mathbf{x}] + \mathbb{E}[d|\mathbf{x}] - d)^2 \\ &= (y(\mathbf{x}) - \mathbb{E}[d|\mathbf{x}])^2 + 2(y(\mathbf{x}) - \mathbb{E}[d|\mathbf{x}])(\mathbb{E}[d|\mathbf{x}] - d) + (\mathbb{E}[d|\mathbf{x}] - d)^2 \end{aligned} \tag{13}$$

Substituting this into Equation (12), the expected loss can be written as

$$\iint \left\{(y(\mathbf{x}) - \mathbb{E}[d|\mathbf{x}])^2 + 2(y(\mathbf{x}) - \mathbb{E}[d|\mathbf{x}])(\mathbb{E}[d|\mathbf{x}] - d) + (\mathbb{E}[d|\mathbf{x}] - d)^2\right\} p(\mathbf{x}, d)\mathrm{d}d\mathrm{d}\mathbf{x}. \tag{14}$$

Performing the integral over d results in

$$\mathbb{E}[L] = \int (y(\mathbf{x}) - \mathbb{E}[d|\mathbf{x}])^2 p(\mathbf{x})\mathrm{d}\mathbf{x} + \int \mathrm{var}(d|\mathbf{x})p(\mathbf{x})\mathrm{d}\mathbf{x}, \tag{15}$$

where the cross-term has vanished. The second term represents the intrinsic noise and is independent of the network y. Thus, the expected loss is minimized when the network satisfies $y(\mathbf{x}) = \mathbb{E}[d|\mathbf{x}]$, resulting in the first term being zero and thus only the intrinsic noise contributing to the expected loss. In practice, this is unattainable since with only a limited amount of data the regression function $\mathbb{E}[d|\mathbf{x}]$ cannot be known exactly and thus it is not possible to simply construct the network according to $y(\mathbf{x}) = \mathbb{E}[d|\mathbf{x}]$. Let us therefore assume that the dataset D is of limited size. The notation $y(\mathbf{x}; D)$ is now used, specifying that the network is dependent on the particular dataset D. The integrand of the first integral of Equation (15) can now be written as

$$\begin{aligned} (y(\mathbf{x}; D) - \mathbb{E}[d|\mathbf{x}])^2 &= (y(\mathbf{x}; D) - \mathbb{E}_D[y(\mathbf{x}; D)] + \mathbb{E}_D[y(\mathbf{x}; D)] - \mathbb{E}[d|\mathbf{x}])^2 \\ &= (y(\mathbf{x}; D) - \mathbb{E}_D[y(\mathbf{x}; D)])^2 + (\mathbb{E}_D[y(\mathbf{x}; D)] - \mathbb{E}[d|\mathbf{x}])^2 \\ &\quad + 2(y(\mathbf{x}; D) - \mathbb{E}_D[y(\mathbf{x}; D)])(\mathbb{E}_D[y(\mathbf{x}; D)] - \mathbb{E}[d|\mathbf{x}]). \end{aligned} \tag{16}$$

Taking the expectation of this with respect to the dataset D gives

$$\mathbb{E}_D[(y(\mathbf{x};D) - \mathbb{E}[d|\mathbf{x}])^2] = \mathbb{E}_D[(y(\mathbf{x};D) - \mathbb{E}_D[y(\mathbf{x};D)])^2] + (\mathbb{E}_D[y(\mathbf{x};D)] - \mathbb{E}[d|\mathbf{x}])^2 \tag{17}$$

since the cross-term of Equation (16) vanishes [4]. Putting this back into Equation (15) finally results in

$$\begin{aligned}\mathbb{E}[L] = &\underbrace{\int \{\mathbb{E}_D[y(\mathbf{x};D)] - \mathbb{E}[d|\mathbf{x}]\}^2 p(\mathbf{x})\mathrm{d}\mathbf{x}}_{bias^2} \\ &+ \underbrace{\int \mathbb{E}_D\left[\{[y(\mathbf{x};D)] - \mathbb{E}_D[y(\mathbf{x};D]\}^2\right] p(\mathbf{x})\mathrm{d}\mathbf{x}}_{variance} \\ &+ \underbrace{\int \mathrm{var}(d|\mathbf{x})p(\mathbf{x})\mathrm{d}\mathbf{x}}_{noise},\end{aligned} \tag{18}$$

where again, the last term represents the intrinsic noise [4]. The first term, called the bias term, measures how much the different models differ from the desired target function $\mathbb{E}[d|\mathbf{x}]$. The second term, called the variance term, measures how sensitive the function $y(\mathbf{x};D)$ is to the particular dataset D. It turns out that these two terms are in conflict, meaning that models with a small bias will have a large variance [4]. This problem is called the bias-variance trade-off. As a consequence, flexible models (networks with a lot of parameters) will in general have the ability to fit a dataset very well, resulting in low bias. However, this will inevitably also lead to large variance. On the contrary, a simple model will give a small variance but will not be able to capture all the complexity in the dataset. Thus, the aim is to find a good balance between the two terms that will lead to good generalization.

3.4.2 Overfitting and underfitting

Before describing particular regularization techniques, a more formal definition of underfitting and overfitting will be presented. Possible definitions of the terms are given below.

> **Underfitting** occurs when the model is not flexible enough, resulting in bad performance on both the training and testing data.
>
> **Overfitting** occurs when the model is too flexible, giving good results on training data but struggles on testing data, resulting in poor generalization performance.

The concept of underfitting is strongly related to that of having a large bias in the bias-variance trade-off. Similarly, overfitting relates to that of having a model tune into random noise in the training data and finding patterns that only exist in the training set, leading to a large variance. Overfitting is one of the big challenges when training a model since, especially deep neural networks, contain a large number of parameters and are therefore prone to overfitting [29]. A common way of dealing with these issues is to use what is called regularization. Common practice is to first make sure the model is flexible enough to capture important patterns in the data, i.e. avoiding underfitting. Regularization plays the part of confining overfitting.

3.4.3 L2 regularization

The first technique described here of regularizing a network is called L2 regularization, also known as weight decay. The idea is to add a second term to the error function, specifically the L2 norm of the weight vector. This term penalizes large values of the weights, essentially shrinking them towards zero. The new error function takes the form

$$E(\mathbf{w}) = E_{unregularized}(\mathbf{w}) + \lambda E_{L2}(\mathbf{w})$$

where

$$E_{L2}(\mathbf{w}) = \frac{1}{2}\sum_i w_i^2$$

and $E_{unregularized}(\mathbf{w})$ is the standard error function without any regularization. λ controls the extent to which the parameters are allowed to shrink towards zero, effectively controlling the amount of regularization. It has to be set manually and is thus a hyperparameter.

Other versions of this regularization technique exist where other norms than the L2-norm are used. Using the L1 norm for example is called lasso regression and has a tendency to shrink some parameters more than others. This commonly results in a sparse model where some parameters are shrunk all the way to zero, essentially playing no role in the network 4.

3.4.4 Early stopping

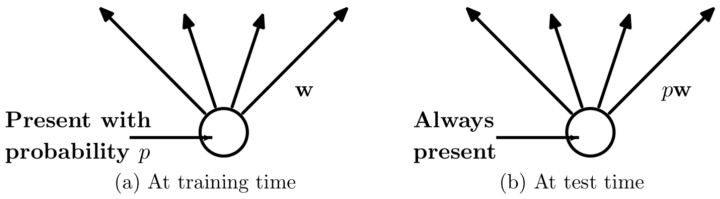
Another technique known as early stopping is easily illustrated with a figure. Figure 3 depicts the training procedure, showing how the training and validation error varies with the number of epochs the model has been trained for. A typical behaviour of overfitting is illustrated. Initially, both the training and validation error decreases, desirably. Eventually, as the training progresses, the model starts to tune in "too well" on the training data, meaning it starts to pick up on noise and patterns that only exist in the training set. This results in the validation error starting to increase, likely causing the generalization performance to deteriorate. A remedy for this is to simply stop training once the validation error starts to increase, as further training will not improve the model. Since training is interrupted prematurely this is referred to as early stopping.

Figure 3: Illustration of the regularization technique early stopping. The method suggests to stop training when the validation loss starts increasing, since this indicates that overfitting is occurring.

3.4.5 Dropout

A technique that has shown to work very well is a rather new one, called dropout. It was first introduced in 2014 and has turned out to work well in a number of different domains [29]. The idea is to avoid overfitting by randomly dropping nodes and their associated weights during training, effectively preventing the weights to adapt too well. Figure 4 shows how the technique is used in practice. During training, the nodes are dropped with probability $1 - p$, in effect creating a thinned network. During testing, the full network is used. However, to make up for training with thinned networks, the weights are scaled with the factor p. The motivation behind this is that the output during testing then is the same as the expected output during training. Note that p is a hyperparameter controlling the amount of droupout regularization.

Another way to view dropout is to see it as training a large number of networks with shared weights and then use the average as the final network. To clarify, suppose the full network has n nodes. Then there are 2^n possible thinned networks. During training, for each presentation of training examples to the network, a thinned network is sampled and trained. Thus, a large number of different thinned networks are trained. In the end, the average of all these trained networks is used as the final network. The averaging is not done explicitly, rather the technique of keeping all nodes but scale the weights with the factor p is used, described in the previous section. This is also illustrated in Figure (4).



(a) At training time (b) At test time

Figure 4: Illustration of how the nodes are treated during training and testing time using dropout. **Left:** During training the nodes are present with probability p, i.e. dropped with probability $1 - p$. **Right:** At test time, the nodes are always present but the weights are scaled with the factor p. In effect, the expected output from the node is thus the same during training and testing [29].

3.5 Convolutional neural networks

A type of network that is well suited for tasks involving images as input is the convolutional neural network (CNN) [15]. This network utilizes the property of images that nearby pixels are correlated to a greater degree than more distant pixels. This is done by the extraction of low-level local features in early layers in the network. These features are then merged to form higher-level features later in the network that are used to make predictions in the output layer.

3.5.1 The convolutional layer

As the name suggests this kind of network deals with convolutions. This is a mathematical operation where two real-valued functions are convolved to produce a third function [40]. In the context of a CNN, the input image is convolved with so called filters. Assume that the image has size $n \times m \times r$, where n is the height, m the width and r the number of color channels of the image. The filters, also referred to as kernels, have size $f_n \times f_m \times f_r$, where $f_n < n$, $f_m < m$ and $f_r = r$. They are essentially three-dimensional tensors containing the weights of the convolutional layer. Since the filters are smaller in size than the input, each filter will 'move across' the input.

For each part of the input that the filter currently hits, called a receptive field, an elementwise multiplication followed by a summation is computed. The result of convolving one such filter with the input is called a feature map and has size $(n - f_n + 1) \times (m - f_m + 1)$, assuming $r = f_r$ and no zero-padding (see Section 3.5.2) [18]. Thus, if there are k filters, there will be k feature maps per input. Similarly to ANN's, there are biases, one for each filter. There is also an activation function acting on each feature map. Figure 5 shows in detail how a convolutional layer with two filters operates.

3.5.2 Hyperparameters of CNNs

In a convolutional layer, there are a couple of hyperparameters to consider. Mentioned in the previous section, the number of filters k and the size $f_n \times f_m \times f_r$ of each filter plays a role in the size of the output of the convolutional layer. These are all hyperparameters. As previously mentioned, given a single input image, each filter produces one feature map. Stacking these k feature maps along the depth dimension thus results in the output from the convolutional layer to have size $(n - f_n + 1) \times (m - f_m + 1) \times k$.

Furthermore, stride s is a hyperparameter determining how much each filter should move across the input at each step. Having a stride of s means that the filters move s pixels at a time. It is common to have a stride of $s = 1$ or $s = 2$.

The last hyperparameter is called zero-padding p. Suppose for this example that the size of the input after the convolutional layer should remain constant. This can be done by adding padding of zeros to the input to the convolutional layer (see the first column of Figure 5). The filter can then start convolving slightly outside of the image, now incorporating the padding of zeros to work with (for the elementwise multiplication mentioned in the previous section). The final output from the convolutional layer is then of size $n_{out} \times m_{out} \times r_{out}$ where $n_{out} = n - f_n + 2p/(s - 1)$, $m_{out} = m - f_m + 2p/(s - 1)$ and $r_{out} = k$. Evidently, the hyperparameters can be chosen in such a way as to preserve the size of the input if needed. Again, Figure 5 summarizes how the convolutional layer and its hyperparameters operate on an image.

3.5.3 Advantages of CNNs

Compared to an ANN, there are two important advantages to a CNN - sparse connectivity and parameter sharing. For simplicity, consider again the network in figure 5, consisting of only one convolutional layer. The input images has size $n = m = 5$, $r = 3$ and the output of the CNN has size $n_{out} = m_{out} = 3$, $r_{out} = 2$. This gives a total of $3 \times 3 \times 3 + 1 = 28$ parameters per filter (the last parameter comes from the bias). Here there are two filters giving a total of $28 \times 2 = 56$ parameters.

On the other hand, a fully-connected ANN with one layer will have $N_i \times N_o$ trainable parameters, where N_i is the number of inputs and N_o the number of outputs. In this case, $N_i = n \times m \times r = 5 \times 5 \times 3 = 75$ and $N_o = n_{out} \times m_{out} \times r_{out} = 3 \times 3 \times 2 = 18$, giving the ANN a total number of $N_i \times N_o = 75 \times 18 = 1350$ parameters. Compare this to the CNN with merely 56 parameters!

The reason for this huge difference is that the CNN utilizes the fact that features in one part of the image, are also useful in other locations. Hence the usage of filters that slide across the entire image. This sliding action means that a single filter acts multiple times on each image, in turn leading to a CNN using extensive weight sharing compared to a fully-connected ANN with no such sharing. Furthermore, note that each element in the feature maps is computed using only a small, local part of the image, i.e. the part that the filter currently hits, see figure 5. In other words, the convolutional layer is not fully-connected, but rather has sparse connections. This way, in general a CNN ends up using far fewer parameters than the fully-connected ANN [37].

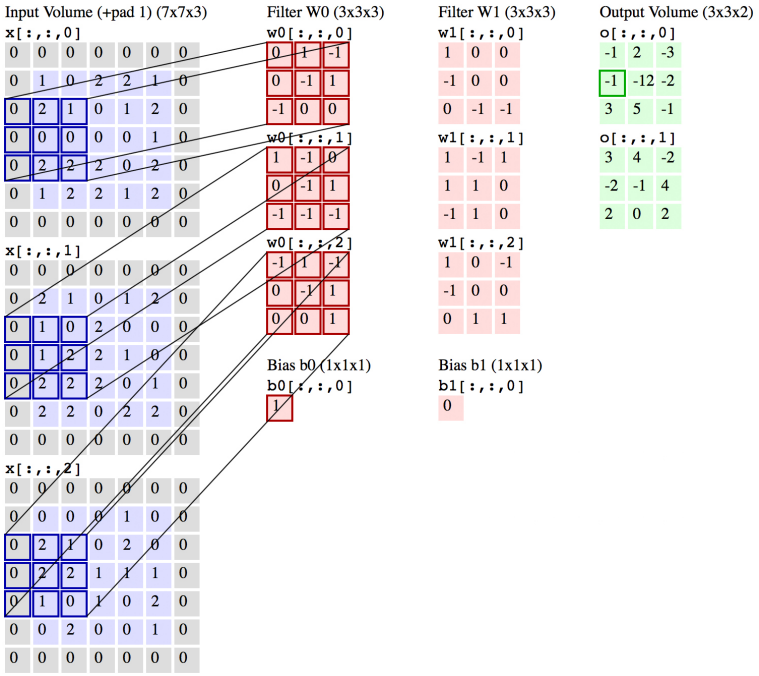


Figure 5: A convolutional layer with two filters acting on an image. The input image has size $5 \times 5 \times 3$ and is shown in the first column. Note that zero-padding of $p = 1$ is used since there are zeros, shown in gray, framing the actual image, shown in blue. The second and third columns represent the two filters and the last column shows the output. The first filter (second column) gives rise to the upper part of the output and the second filter (third column) results in the lower part of the output. In this very moment the figure shows how the first filter hits a certain location of the input image, resulting in -1 in the marked square in the output. This is calculated by elementwise multiplication of the filter with the part of the image that it currently hits (depicted with lines across the first and second column), followed by summation and the adding of the bias. A stride of 2 is used resulting in the output having size $3 \times 3 \times 2$, where the depth of 2 comes from the fact that there are two filters. The activation function has been left out for convienience. The image is taken from [37].

3.5.4 Pooling

A common strategy when using CNNs is to include what is known as pooling layers. This type of layer operates in such a way as to reduce the height and width of the input whilst keeping the depth constant. The idea is to reduce the number of parameters needed in the CNN by downsampling the size of the input [37]. This could also be viewed as feature selection, i.e. less relevant features are discarded for more relevant ones, which is known to reduce the risk of overfitting [35].

Arguably the most common pooling operation is the 2×2 max pooling. Here the height and width of the input are halved by choosing the largest value in each non-overlapping 2×2 grid. Other common operations include average pooling and stochastic pooling [35]. The former operation computes the average over the pooling region while the latter stochastically picks the activation within each pooling region according to a multinomial distribution [34].

3.5.5 Architecture of a typical CNN

A typical CNN consists of one or more convolutional layers in the early stages, followed by fully-connected layers. A pooling layer usually follows right after a convolutional layer [37]. Note that

this is a common basic architecture and that other types of layers may also be found in CNNs such as e.g. batch normalization layers, see section 3.6.4.

The early convolutional layers in a CNN act as low-level feature extractors. They detect edges and dots which are generic features of real images. The succeeding convolutional layers act on a mid-level and the final layers on an even higher level, in some sense merging the lower-level features from previous layers. This way later layers are able to detect larger shapes and patterns, commonly features that are more task-specific as opposed to the more generic features in earlier layers. The fact that CNNs work this way, turns out to be a very interesting and useful property. As a consequence, when faced with a new problem, it should be possible to reuse previously trained low-level extractors, i.e. early convolutional layers, instead of training a new network from scratch! This is what transfer learning is about, more about it in section 3.7.

3.6 Datasets

The importance of working with high-quality datasets cannot be emphasized enough. The dataset is what the algorithm learns from, meaning that a good dataset is of tremendous importance. A few problems with datasets include too little data, learning from bad labeling and missing unexpected patterns [25]. Only the problem of too little data will be discussed here, more specifically in section 3.6.3.

3.6.1 Preprocessing

Preprocessing is an important step in machine learning where the dataset is prepared for training. The particular choice of preprocessing is highly dependent on the problem at hand and it is therefore hard to give a general view of the topic. Here, only problems that deal with images as inputs will be considered.

Consider again the problem of detecting cats and dogs in images. Assume that the dataset is imbalanced, containing many images of cats but only a few of dogs. The dataset is said to have a bias. In this situation, a network that predicts all images contain cats would result in a rather small loss. It is therefore not far-fetched that training would indeed result in such a network. A method used to getting rid of biases in datasets is called under- or oversampling. Oversampling means that the class with the fewer examples is sampled from, creating more of those examples. Similarly, undersampling deals with removing examples from the class with more examples, thus decreasing the bias.

Other preprocessing steps are cropping and changing from one color space to another. For a self-driving car with a camera mounted at the front of the car, pixel information of the hood is not relevant. Cropping in such a case would be preferable. What color space to use to represent the images seem to be dependent on the problem at hand [19]. Some common choices are RGB, YUV and grayscale.

3.6.2 Normalizing the inputs

Normalizing the inputs is something that is commonly done in practice to make sure the different input values are on the same scale. For instance, consider a problem with inputs $x_1 \in [0, 1]$ and $x_2 \in [0, 1000]$. Furthermore, keep in mind that the weights of the network are commonly initialized by drawing from some distribution, i.e. they are somewhat similar in magnitude. Since the second input in general takes larger values, there is a risk that the first input will "drown" in the second, i.e. have little or negligible effect on the output [20]. Hence normalization of the input is preferable. Two common ways of normalizing the inputs are to either transform the inputs to have zero means and unit variance or to linearly transform them to a fixed interval $I = [a, b]$.

3.6.3 Data augmentation

Consider the classification task of object detection in images. In general, by doing small augmentations to the image the captured object still remains the same. It is said that the predictions from the network should stay *invariant* under these augmentations. For example, to classify handwritten digits, a network would desirably predict the same digit even if the input image has been translated, scaled or rotated slightly. Here, a few of the augmentation techniques to consider when training a convolutional neural network will be presented.

Translation means that the image is moved along either one or both of the x and y-direction. In the case of object detection, the object will be in a different position after translation. The label naturally remains unchanged since the object has not changed, only the position of it.

Flipping the image along one axis is another useful augmentation technique. In the case of training a network to drive a car with an imbalanced network, say with too many images of the car turning left, doing a horizontal flip helps remove this bias. In fact, flipping each image with a probability of one half will alone create symmetry to the dataset.

Shadowing means that regions of the image gets darkened in an attempt at creating realistic-looking shadows. This is done by lowering the intensity of pixels in random, coherent areas of the image. This technique may help the network driving at different hours of the day, since during training it will now be presented with images that try to imitate the effects of sunbeams falling from different angles.

Adjusting the brightness is the last augmentation technique discussed here. As opposed to shadowing, the entire image is now either brightened or darkened by changing the pixel values.

In summary, data augmentation is a technique used to create more data by doing transformations to the inputs. In some sense, it can be viewed as a regularization technique since an important benefit from augmenting the data is to avoid overfitting [14]. There is a well-known relation between the amount of data, the model complexity and overfitting. In general, keeping the model fixed, more data leads to less overfitting [4]. Simard et al's work explains why data augmentation is of such importance in machine learning, stating that the number one most important practice in machine learning is to have a large dataset, achievable with augmentations [26]. It should be mentioned that there is a limitation to the effectiveness of data augmentation and that more 'real' data is in general preferred over augmented data.

3.6.4 Batch normalization

A problem with training deep neural networks is that the input distribution to the layers in the network does not remain constant during training. Consider a layer F in the network, any layer but the input layer, with corresponding input $\mathbf{x}_F$. As the weights of the previous layers are updated, the distribution of $\mathbf{x}_F$ changes. This phenomena is called internal covariate shift and has negative effects such as slowing down the network training [11]. A suggested remedy for this problem is called batch normalization. In combination with gradient descent using mini-batches, the idea is to normalize the input to a layer for each of these batches. This leads to the desired speed up of training and also works as regularization [11].

3.7 Transfer learning

Transfer learning is a type of learning that uses previously attained knowledge from one problem and applies it to another similar problem. It is inspired by the fact that humans more easily learn a new concept if they are already familiar with a related one. For example, learning to play the guitar is easier if you already know how to play the bass. In a practical setting, the amount of data is not seldom limited. Transfer learning tries to transfer knowledge from a related problem, aiming

at making a relatively small amount of data sufficient for the new problem [33]. Another aim is to improve the speed of the training, compared to conventional non-transfer learning methods.

Formally, a few definitions related to transfer learning are presented below [16].

> *Definition*: **Domain**
> A domain, which is denoted by $D = \{\chi, P(X)\}$, consists of two components:
> (1) Feature space χ; and
> (2) Marginal probability distribution $P(X)$, where $X = \{x_1, ..., x_n\} \in \chi$.
>
> *Definition*: **Task**
> A task, which is denoted by $T = \{Y, f(\cdot)\}$, consists of two components:
> (1) A label space $Y = \{y_1, ..., y_m\}$; and
> (2) An objective predictive function $f(\cdot)$ which is not observed and is to be learned by pairs $\{x_i, y_i\}$.
>
> *Definition*: **Transfer learning**
> Given a source domain D_s and learning task T_s, a target domain D_t and learning task T_t, transfer learning aims to improve the learning of the target predictive function $f(\cdot)$ in D_t using the knowledge in D_s and T_s where $D_s \neq D_t$ or $T_s \neq T_t$.

The definition of transfer learning introduces the terminology of source domain and task, as well as target domain and task. The source refers to the previously solved problem, and the target to the problem at hand. In the classic context of transfer learning, it is further assumed that the domains are *somewhat* similar. Although there are work on transferring knowledge from more distant domains [31], here the premise of close relation between domains will be assumed.

Using these definitions, transfer learning can be categorized into two main types. When $T_s \neq T_t$, i.e. the source and the target tasks are different, the learning is sometimes called inductive transfer learning. On the contrary, when the tasks are the same but the domains differ, i.e. $T_s = T_t$ and $D_s \neq D_t$, it is called transductive transfer learning, or domain adaption. Note that in the classic setting of machine learning, it is assumed that $T_s = T_t$ and $D_s = D_t$.

Another possible scenario in transfer learning deals with both the domain and tasks being different, i.e. $T_s \neq T_t$ and $D_s \neq D_t$. This could for example be the source domain being images of churches and the target domain being images of cars. With a source task being a binary classification task with labels "protestantic/catholic" and the target task classifying the car brand, both $T_s \neq T_t$ and $D_s \neq D_t$ hold. Even in this setting there has been great success transferring knowledge [21].

3.7.1 Transferring knowledge with CNNs

An interesting area for transfer learning is that of using it in conjunction with convolutional neural networks. Since these networks tend to house a large number of parameters (deep CNNs), transfer learning is a good option, especially when dealing with datasets of limited size. There are various approaches when doing transfer learning depending mainly on the size of the new dataset and the similarity between the source and target domain. Four scenarios can be defined [38] that can be seen as rules of thumb. They are shown in figure 6 and below it is assumed that a CNN with a typical architecture (see section 3.5.5) has been trained on the original problem, i.e. the source domain and task. This CNN is referred to as the pretrained network. The four scenarios are presented below.

(**a**) *The new dataset is small and the domains are similar.* A small dataset means there should be concerns about overfitting. The similarity between the datasets indicates that not only low-level but also higher-level features in the CNN are relevant to the new problem. Hence, it is

recommended to remove the last part of the pretrained network, i.e. the last or last couple of fully-connected layers. The convolutional layers remain the way they are. On top of this reduced network, one or more fully-connected layers are stacked. These newly added layers are then trained, while the weights of all other layers are kept fixed. This way of doing transfer learning is sometimes referred to as letting the pretrained network act as a *feature extractor*.

(**b**) *The new dataset is large and the domains are similar.* There is now more data compared to the previous scenario, meaning that overfitting is not as imminent. Hence, a transfer learning method called fine-tuning is recommended. Fine-tuning, like feature extraction from scenario (a), also involves first removing the last part of the pretrained network and then adding new fully-connected layers. The network, like before, now consists of newly added fully-connected layers stacked on convolutional layers from the pretrained network. During fine-tuning however, the weights of all or some of the convolutional layers are also allowed to be updated and thus to adapt to the new dataset. The pretraining therefore helps with the initialization of the weights of the convolutional layers. The number of convolutional layers that are trained may depend on the similarity between the source and target domain and task. More similar means that higher-level features are relevant to both problems to a larger degree, and thus less convolutional layers needs to be trained. Since a fine-tuning has comparatively more trainable weights than a feature extractor network, training lasts longer.

(**c**) *The new dataset is small and the domains are dissimilar.* Again, a small dataset raises the concern for overfitting. Since the domains are also dissimilar it may not be a good idea to stack and train fully-connected layers on top of the convolutional layers of the pretrained network. This is due to the fact that higher-level features of the pretrained network are likely not relevant to a problem with a domain being only distantly related to the domain of the original problem. Hence, it is recommended to stack the fully-connected layers on top of *only the first few* convolutional layers of the pretrained network. Thus, only low-level features are transferred. This is also a sort of feature extraction where only low-level features are extracted.

(**d**) *The new dataset is large and the domains are dissimilar.* In this case one might see no point in doing transfer learning - dissimilar domains is not the preferred setting in transfer learning and a large new dataset means it should be possible to train a network completely from scratch. However, fine-tuning may give better results than training from scratch, especially if training time is restricted.

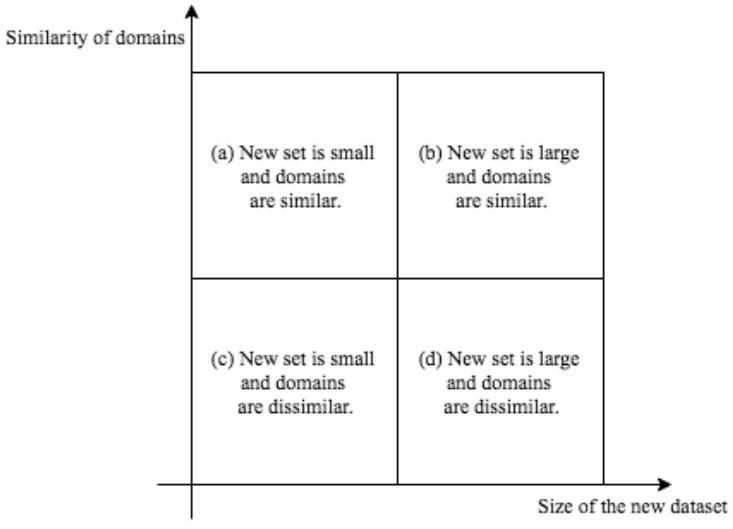


Figure 6

4 Methodology

4.1 Software

The code in this thesis is implemented using the *Python* programming language. The high-level neural networks API *Keras* running on top of the machine learning framework *TensorFlow* is used extensively. The code is not run locally but submitted to LUNARC which is the center for scientific and technical computing at Lund University. To collect data and test the models Udacity's self-driving car simulator is used.

4.2 Data

The data in this thesis consists of two separate datasets, both originating from Udacity's car simulator. The simulator contains multiple driving environments, where this project includes two of those environments.

4.2.1 Lake track

The first dataset comes from the environment referred to as the lake track. As the name suggests this environment consists of a road winding up alongside a lake. The track is fairly simple if the difficulty is measured in terms of the amount of sharp turns. In other words, the track consists of mainly driving straight.

4.2.2 Jungle track

This track is situated in the jungle and compared to the lake track it is much more difficult. The majority of the time is spent turning on this winding road. The track also includes a fair amount of elevation, sometimes causing problems when a peak is prohibiting a clear visual understanding of the path.

4.2.3 Data collection

Since no dataset of desirable quality is found online, data collection is included in the project. This is done by manually driving the car in the simulator while constantly collecting the necessary data. The driving is done in a very strict manner and the datasets can be said to consist of mainly two parts. The first and major bulk consists of "normal" driving, trying to stay centered in the lane. The remaining part is referred to as recovery data. Here, data is collected from situations where the car is just about to go off-road, and how it then recovers to the center of the lane again. The purpose of the recovery data is to teach the car to safely return to the roadway if, or when, it drifts away from the center of the lane.

The lake dataset consists of roughly 13.000 examples while the jungle dataset was considerably smaller, containing merely slightly less than 4.000 images. Note that this is no coincidence. The lake dataset is to be used for pretraining a network with a larger amount of data. This pretrained network will then be used in a transfer learning setting on the new and smaller dataset, namely the jungle dataset. To be clear, in a normal transfer learning setting, one would ideally have a pretrained network ready to use. Since no such network is available in this project, the pretraining of a network has to be done as well, explaining the need for the lake dataset.

4.2.4 Data specifications

The datasets take the form $D = \{\mathbf{x}(n), \mathbf{d}(n)\}_{n=1,..,N}$ where an input example $\mathbf{x}(n)$ consists of three images corresponding to the left, center and right cameras mounted at the front of the car,

see figure 7 and 8. The images are represented with their pixel values. The size of an image is 320×160 pixels and the color space is the classic RGB. During training, when an example is treated, only one of the three images is chosen at random. One example will therefore be referred to as *one* image, even though it really contains three images.

Each example has a corresponding label $\mathbf{d}(n)$, containing two numeric values. These represent the steering angle and speed of the car and are the quantities that are to be predicted by the network.

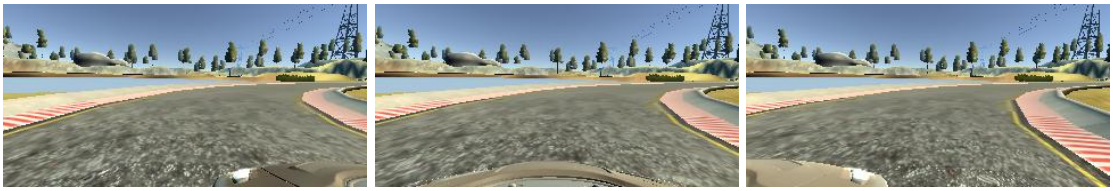


Figure 7: Data recorded during the collection phase from the lake track. The three images represent the left, center and right camera and make up one example.

Figure 8: Data recorded during the collection phase from the jungle track. The three images represent the left, center and right camera and make up one example.

4.2.5 Transfer learning view

A short transfer learning view will here be given. Mentioned above, the lake track will only be used to pretrain a network, making it the source domain. The jungle track is the target domain since the goal is to train a network to perform well here. A domain consists of a feature space and a marginal probability distribution (see section 3.7). The lake and jungle datasets both consists of images of the same size, represented in the same way using pixel values. Thus, the feature spaces of the domains are identical. However, the images originate from two different driving environments, meaning that the pixel values have different distributions. The domains are thus said to have different marginal probability distributions, leading to the domains being different, i.e. $D_s \neq D_t$.

Similarly, a task consists of two components - a label space and an objective predictive function. The label spaces for the source and target tasks are the same, since they both consist of the numeric values representing the steering angle and speed. Whether the objective predictive functions are the same remains unclear. One could probably argue that they are the same or at least similar. Since the feature spaces are identical, one could imagine that the desired output from feeding an image $\mathbf{x}$ into the objective predictive functions $f_s(.)$ and $f_t(.)$ then ideally should be the same. However, if in practice this actually holds remain unclear since the marginal probability distributions practically are non-overlapping, i.e. the exact same image will never be drawn from them.

4.3 Preprocessing and data augmentations

Before training, the input is preprocessed and augmented. The preprocessing includes cropping and converting the image to another color space. Since the sky and the hood of the car does not

provide any important information, they are subject to cropping. The conversion of color space from RGB to YUV is done motivated by the fact that it has been done on previous work with good results [5].

Augmentation is done to enlargen the datasets and includes translation, horizontal flipping, shadowing and adjusting the brightness. The augmentation of the images is done on the fly to save storage. This means that only images currently being processed by the algorithm are augmented. When they have been trained on, they are discarded. This way no augmented images are stored permanently leading to less memory being used.

4.3.1 Distribution of steering angles

Figure 9 shows the distribution of steering angles from the jungle dataset. It is clear that there is an unproportionately large number of examples with steering angles close to zero. Similarly, there are too many examples with steering angles corresponding to sharp left and right turns. Note that the steering angle has been normalized to lie in the interval $[-1, 1]$ where 0 represents straight driving and a steering close to -1 or 1 means sharp turning left or right respectively.

To address these issues a sort of oversampling method in conjunction with augmentation is used. This is achieved in a slightly unconventional manner. Instead of focusing on examples with underrepresented outputs, examples are chosen at random. For each of these examples, the image from the left, center or right camera is chosen at random with equal probability. This means that the left or right camera is chosen with probability two thirds. If the left or right camera was picked, a shift of the steering angle is added or subtracted to make up for the fact that the steering angle otherwise corresponds to the center camera. Since examples with overrepresented steering angles are more likely to be chosen, together with the fact that the left or right camera is chosen with probability two thirds, this procedure effectively helps even out the imbalances of the distribution.

Similarly, the augmentation technique of translation also works in favor of evening out the imbalances in the dataset. The result of these techniques is shown in figure 10. Clearly, the distribution is much more well-balanced, with a non-negligible representation of steering angles ranging all the way from sharp left to sharp right turns. There is a slight emphasis on angles close to zero.

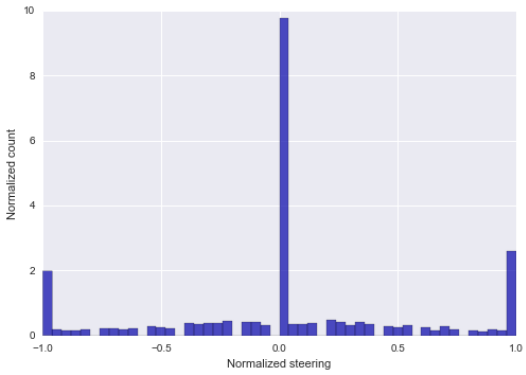


Figure 9: Histogram showing the distribution of steering angle of the jungle dataset *before* any augmentation or sampling. Note that the steering is not measured in degrees but rather has been normalized to lie in the interval $[-1, 1]$. The count has been normalized to form a probability distribution.

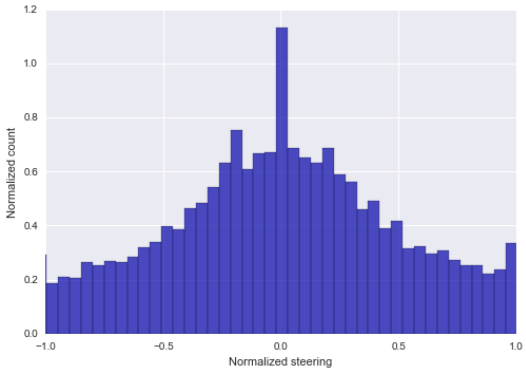


Figure 10: Histogram showing the distribution of steering angle of the jungle dataset *after* augmentation and sampling. Note that the steering is not measured in degrees but rather has been normalized to lie in the interval $[-1, 1]$. The count has been normalized to form a probability distribution.

4.4 Network architecture and hyperparameters

The network architecture used is shown in figure 11. This network design is based on NVIDIA's architecture used in their CNN to train a self-driving car [5]. The normalization layer makes sure the inputs are in the range [-1, 1]. The number of trainable parameters in the network with no layers freezed exceeds 250 thousand by a small margin.

Aiming to use many layers with small filter sizes, the network consists of five convolutional layers with kernel sizes of 3×3 and 5×5. Compared to using fewer layers with larger kernel sizes, the former approach is claimed to work better [37]. Below follows an overview of the network design and the training procedure.

Activation function Motivated by their ability to alleviate the vanishing gradient problem [6], ELU is chosen as activation function throughout the network. The slightly simpler function ReLU (Rectified Linear Units) is not considered due to its lack of zero means. Other choices are possible but not tried due to time limitations.

Regularization The network is regularized using dropout. Other methods, such as L2 regularization are tested although no improvement in performance is detected compared to dropout. As mentioned earlier, augmentation in some sense also works as regularization by increasing the size of the dataset. The batch normalization layers also has a regularization effect.

Optimizer The choice of optimizer falls on Adam. This first-order gradient-based optimizer is a common choice when training large neural networks and has been shown to achieve good results [12]. Due to time limitations no further testing is done with other optimizers.

Choice of hyperparameters Due to the rather large number of trainable parameters in the network, training is time-consuming, lasting multiple hours. This is unfortunate, leading to the hyperparameters being tuned via the basic method of manual search. Hyperparameters related to the optimization includes number of epochs, batch size, learning rate, samples per epoch, dropout probability and test size. The final values are shown in the results.

The ratios of the sizes of the sets, determined by the hyperparameter `test_size` are set according to

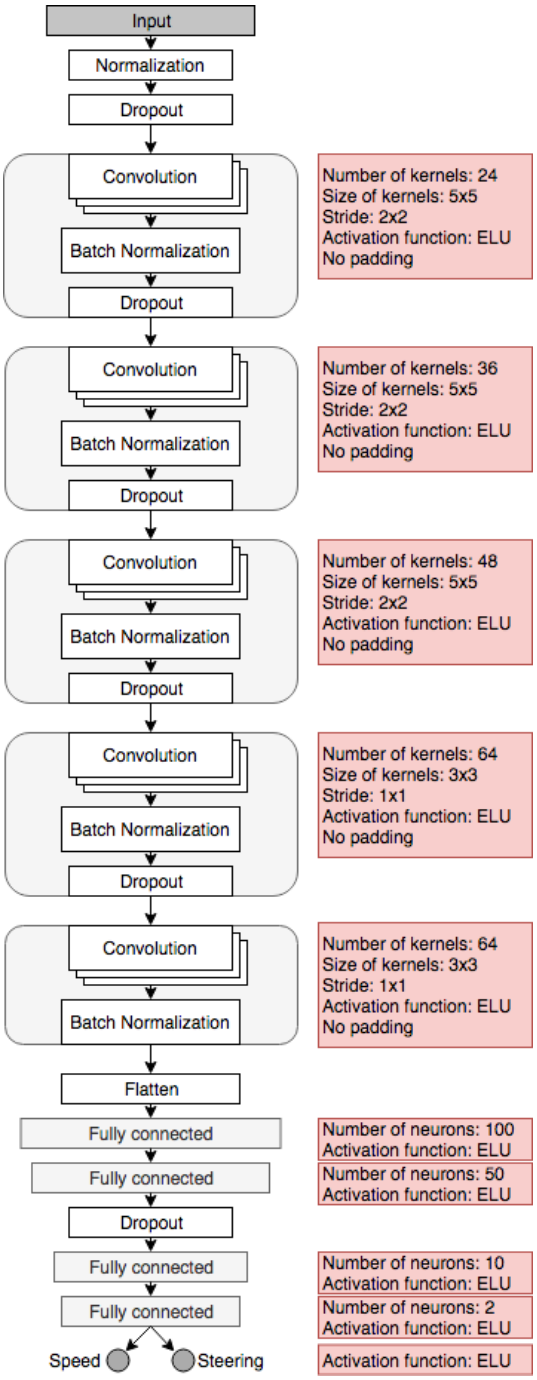


Figure 11: Convolutional neural network based on the NVIDIA design. There are five convolutional layers mixed with batch normalization layers. After flattening there are four fully-connected layers followed by the output layer consisting of only two nodes. Extensive dropout is used throughout the network.

$$(\texttt{training set} : \texttt{validation set} : \texttt{test set}) = 1 - \texttt{test\_size} : \frac{\texttt{test\_size}}{2} : \frac{\texttt{test\_size}}{2}$$

.

The test size hyperparameter is treated slightly different from the rest of the hyperparameter. The other hyperparameters are in general tuned with the main goal of minimizing the loss on validation data. That same goal cannot be used when tuning the test size hyperparameter, since having a very small test size may yield low validation loss, even though the generalization performance is poor. Therefore the test size hyperparameter is set constant in this thesis, i.e. `test_size` $= 0.3$ for all experiments.

4.5 Tasks

Below follows a short description of the how the experiments are conducted for the different tasks. The networks used in all tasks have the same architecture, shown in figure 11.

4.5.1 Task 1 - Training on the lake dataset

The first step is to train a network on the lake dataset. This is to be used as the pretrained network for task 2. The architecture shown in figure 11 is used together with the preprocessing, augmentations and other hyperparameters described in section 4.3 and 4.4.

4.5.2 Task 2 - Training on the jungle dataset/Transfer learning

The second part uses the network from task one as a starting point for further training. More specifically, the weights of the pretrained network serves as weight initialization in accordance with standard transfer learning methodology. Task two consists of training three networks using different transfer learning methods. The first network uses the pretrained network as a feature extractor while the second and third networks use fine-tuning.

During the training of the first network the weights of all layers except the fully-connected layers are set non-trainable. These weights will thus not be updated during training. Training of the fully-connected layers is then carried out in a normal fashion, again using the augmentations of section 4.3 and hyperparameters of section 4.4. The resulting network is referred to as the feature extractor network.

The second and third networks are referred to as the fine-tuning network 1 and 2. The pretrained network of task one is again used as a starting point. The idea of fine-tuning is to not only train the last part of the network, but also to let the weights of the convolutional layers be updated. To make sure the emphasis of the training still is on the last fully-connected layers however, the network is first trained for only a few epochs with the weights of all the convolutional layers set non-trainable, i.e. only the fully-connected layers are trained. Then, a couple of the later convolutional layers are set trainable. They are then trained together with the fully-connected layers, i.e. the network is being fine-tuned. The fine-tuning network 1 and 2 differ in how many of these convolutional layers are being fine-tuned.

4.5.3 Task 3 - Training a network for comparison

This task involves the training of a network using only data from the jungle track. Now, the network is trained from scratch without any pretraining. Again, the architecture and hyperparameters are the same as in the previous two tasks. The performance of this network is evaluated against the other networks to see how well transfer learning is performing.

4.5.4 Evaluating the performance in the simulator

The ultimate goal is for the networks to perform well on the jungle track in the simulator. Thus, a lot of time is spent closely evaluating the networks driving in autonomous mode in the simulator.

Hyperparameter	Value
learning_rate	10^{-4}
number_of_epochs	30
batch_size	40
keep_probability	0.5
test_size	0.3
samples_per_epoch	50 000

Table 1: Choices of hyperparameters used for training the networks. Only small modifications were made to these for some of the networks.

The most important aspect studied is the ability to stay in the lane for as long as possible. A well-trained network can finish an entire lap of the track. To be able to determine which of two such networks is better, the speed of the car is increased until one network eventually cannot finish the entire track, leading to the conclusion that the other network performed better.

Just like a normal car, the speed of the car in the simulator is controlled via the throttle. The throttle in turn is set according to

$$\texttt{throttle} = \texttt{speed\_constant} - \left(\frac{\texttt{speed}}{\texttt{speed\_predicted}}\right)^2, \tag{19}$$

Here `speed` is a measurement from a sensor in the car itself, representing the actual speed of the car. `speed_predicted` is the computed output from the network, representing what the network believes to be the correct speed at the moment. The second term of Equation 19 is thus large if `speed` is large and `speed_predicted` small.

The first term of Equation 19 is a constant set manually that in a broader sense controls the speed of the car. If `speed_constant` is set small, the average speed will be kept low. On the contrary, a large `speed_constant` can be used to challenge the network to drive at generally higher speeds. In summary, Equation 19 states that if the actual speed of the car is higher than the predicted speed, the value of the throttle will be small or even negative, resulting in the car decelerating. Similarly, if the actual speed of the car is lower than the predicted speed, the car will accelerate.

5 Results

The optimization hyperparameter values are given in Table 1. They are the result of manually tuning the hyperparameters.

5.1 Task 1

This task was only a preparatory step to succeeding tasks. It involved training a network to act as a pretrained network for transfer learning networks in task 2. It can be stated that the training was successful, resulting in a car that could easily drive itself around the lake track. Any further results will be omitted.

5.2 Task 2

5.2.1 Feature extraction

The losses from the feature extractor network can be seen in Figure 12. The hyperparameters used are shown in Table 1. Training the network for 30 epochs took 26 330 seconds ≈ 7 hours, 19 minutes. The losses in the figure indicate that the network is indeed learning as both the training and validation loss are decreasing over time. Note that the validation loss is constantly smaller than the training error. The reason for this seemingly strange behaviour can be explained by the way the losses are computed in Keras. The training loss is averaged over all batches of training data. If the network is learning, this means that the first batch of an epoch tends to produce a larger loss than later batches. Since the validation error uses the network as it is at the end of an epoch, this usually means that the validation loss is smaller than the training loss [3].

The performance in the simulator is the ultimate goal and therefore thorough testing was done. The results for the transfer learning network can be viewed in Table 2. The table shows how far the car could drive without driving off the road or crashing. Testing in the simulator was done for several different values of the speed constant in Equation (19) and for several different versions of the network, differing in terms of the number of epochs they were trained for. The performance is measured in percent of the total track completed before driving off the road, 100% meaning it finished the entire track. Notice that there are some values that occur more frequently than others. This is due to some turns being much harder than others, separating the wheat from the chaff.

Feature extractor network

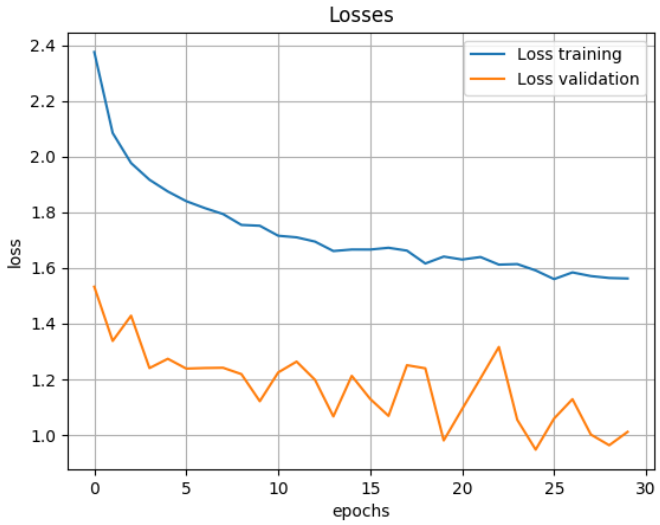


Figure 12: Losses from the feature extractor network. The pretrained network comes from task 1 and all layers but the fully-connected layers are set non-trainable during training. The losses indicate that the network is indeed learning as both the training and validation loss are decreasing.

(a) **Feature extractor network**

speed_constant \ number_epochs	6	12	18	24	30
0.7	19 %	100 %	54 %	54 %	100 %
0.9	19 %	100 %	54 %	19 %	100 %
1.1	19 %	19 %	54 %	4 %	19 %
1.3	19 %	19 %	32 %	4 %	19 %

(b) **Fine-tuning network 1**

speed_constant \ number_epochs	6	7	12	18	24	30
0.7	54 %	100 %	100 %	100 %	100 %	100 %
0.9	50 %	100 %	100 %	100 %	100 %	100 %
1.1	54 %	100 %	100 %	19 %	100 %	29 %
1.3	19 %	72 %	19 %	100 %	19 %	4 %

(c) **Fine-tuning network 2**

speed_constant \ number_epochs	6	7	12	18	24	30
0.7	54 %	100 %	100 %	100 %	100 %	100 %
0.9	54 %	100 %	100 %	46 %	100 %	100 %
1.1	54 %	100 %	100 %	46 %	100 %	100 %
1.3	54 %	100 %	100 %	19 %	100 %	100 %

(d) **Conventional network**

speed_constant \ number_epochs	6	12	16	18	24	30
0.7	19 %	19 %	4 %	100 %	100 %	100 %
0.9	19 %	4 %	4 %	100 %	100 %	100 %
1.1	4 %	4 %	4 %	100 %	100 %	100 %
1.3	19 %	4 %	4 %	100 %	19 %	100 %

Table 2: This table shows the results from testing the networks from task 2 and 3 in the simulator. Testing is done for varying values of the `speed_constant` and `number_epochs`. The percentage values shows how far the car can drive before going off-road, given as percent of one lap of the jungle track. Note that in (b), (c) and (d) there is one extra column highlighted in gray. The purpose of these columns is to help showing how many epochs of training is needed before the networks could finish the entire track for a majority of values of the `speed_constant`. **(a)** The feature extractor networks' overall performance leaves a great deal to be desired. In the initial phase of training, after 6 and 12 epochs, it performs decently. With more training however, it struggles to keep up with the other networks. **(b)** The fine-tuning network 1 overall performs very well. After just 6 epochs for slow to medium speeds it manages to drive half the track before going off-road. With just one more epoch of training however, i.e. 7 epochs, it finishes the entire track except with high speeds. **(c)** This network overall performs really well. After 7 epochs and with more training it can clear the entire track at all speeds. The exception to this is at 18 epochs, for which the network performs much worse. **(d)** The network trained from scratch using only the jungle data performs decently when trained for 6 epochs, and poorly for 12 epochs. However, with more training this network performs really well. Note that training for this network took considerably longer than training the transfer learning networks, see Table 3.

Network	Training time
Feature extractor network	7 hours, 19 minutes
Fine-tuning network	7 hours, 31 minutes
Fine-tuning network 2	7 hours, 27 minutes
Conventional network	13 hours, 57 minutes

Table 3: This table summarizes the training times for the three transfer learning networks and the conventional network for easier comparison. Training lasted for 30 epochs for all networks.

5.2.2 Fine-tuning network 1

The results in terms of losses from the fine-tuning network are seen in Figure 13. The losses show a similar behaviour to the losses from the feature extractor network, indicating successful training. The training lasted for 27045 seconds ≈ 7 hours, 31 minutes. The evaluation of the performance in the simulator can again be viewed in Table 2.

Fine-tuning network 1

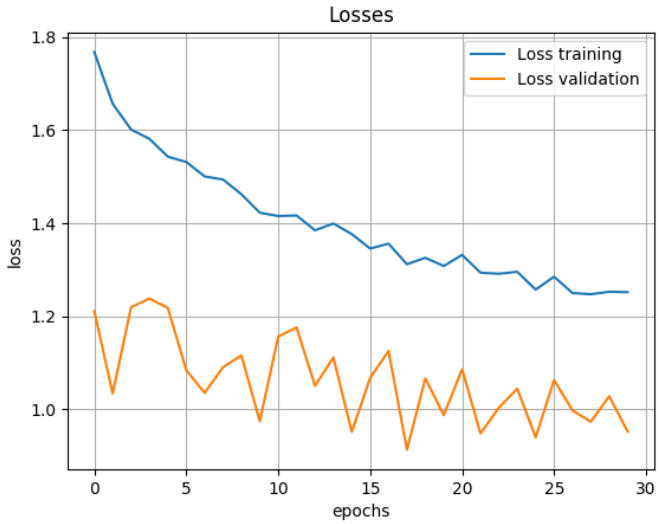


Figure 13: Losses from the fine-tuned network 1. The pretrained network comes from task 1. The fully-connected layers of the network are trained for 5 epochs before fine-tuning all but the first three convolutional layers is carried out. The losses indicate that the network is indeed learning as both the training and validation loss are decreasing.

5.2.3 Fine-tuning network 2

The losses from the fine-tuning network 2 can be seen in Figure 14. They show a similar behaviour to the feature extractor network and fine-tuning network 1, again indicating successful training. Training lasted for 27045 seconds ≈ 7 hours, 27 minutes and the performance in the simulator is shown in Table 2.

Fine-tuning network 2

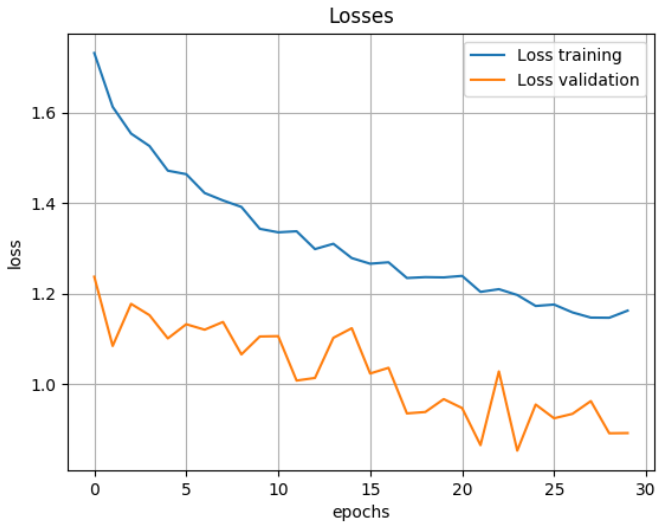


Figure 14: Losses from the fine-tuned network 2. The pretrained network comes from task 1. The fully-connected layers of the network are trained for 5 epochs before fine-tuning all but the first two convolutional layers is carried out. The losses indicate that the network is indeed learning as both the training and validation loss are decreasing.

5.3 Task 3

Task 3 consists of training a conventional network from scratch, using only the jungle data. The losses are shown in figure 15, and again they show a similar behaviour to the two transfer learning networks from task 2. Note that the y-axis uses a different scale than figure 12 and 13, being about one order of magnitude greater. Training lasted for 50209 seconds $\approx$ 13 hours, 57 minutes and the tests done in the simulator are again summarized in table 2.

6 Discussion

6.1 Data

At first glance, it might be tempting during the data collection to simply drive the car staying centered in the lane. Training a network with such data though, it was quickly discovered that the car would inevitably drift off-center of the lane. When this happens, the car enters territory it has never seen during training and thus does not know what to do. It simply cannot generalize to such an extent.

Adding data where the car recovers from such situations turns out to work surprisingly well. Training a network with this new dataset, the car can handle these situations smoothly. This clearly demonstrates the importance of a good dataset.

Another great discovery during this thesis is that the size of the dataset does not have to be huge. Data collection resulted in about 17.000 images being collected, corresponding to roughly half an hour of driving. Compare this to other networks trained for the task of self-driving a car, using considerably more data. NVIDIA for example used a dataset consisting of 72 hours of driving [5].

Conventional network

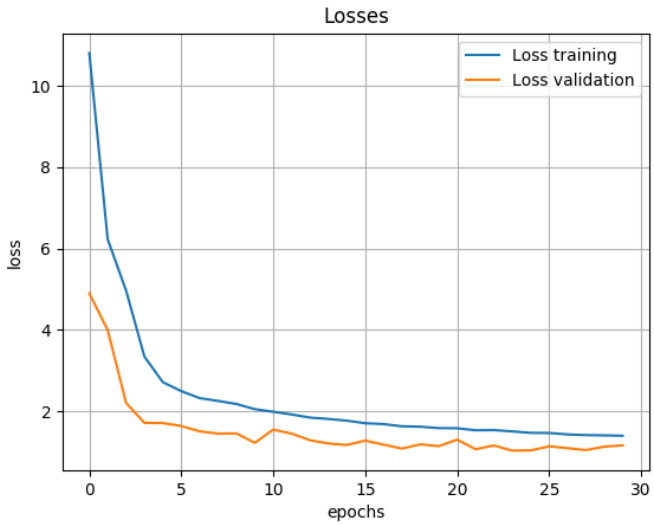


Figure 15: Losses from the conventional network which is trained from scratch using only the jungle dataset. Although the scale of the y-axis is of a different order of magnitude than figure 12, 13 and 14, the losses clearly indicate that the network is learning.

Augmentations play an important role in making it possible to successfully train a network to be able to autonomously drive in the simulator with such a small dataset. They not only help even out imbalances in the dataset but also has the effect of increasing the size of the dataset. Of course, one should keep in mind that NVIDIA trained a network to drive a car in a real-life situation, which is much more complex environment than that of the simulator.

6.2 Network architecture

The network architecture used in task 1-3 is based on the NVIDIA design [5]. Some experimentation was done before the final architecture was decided on. This included for example experimentation with the number of convolutional layers and their filter sizes, the number of batch normalization layers, the number of nodes in the fully-connected layers and the amount of regularization. Due to the fact that the network training takes many hours, it is unfortunately not possible to test as many architectures as one would like.

6.3 Task 1

No discussion for this task is presented.

6.4 Task 2

6.4.1 Feature extraction network

Based on the performance in the simulator (see Table 2), the feature extractor network does at least as good or better than the conventional network for `number_epochs` = 6 and 12. With more training though, the conventional network clearly performs better than the feature extractor

network. The latter network even has a decline in performance as training progresses and there seems to be overfitting. This is contradictory to what is shown in Figure 12, suggesting that there is no overtraining. This raises the question as to how well the decrease in validation loss actually translates to better performance in the simulator.

Overfitting or not, a possible explanation as to why the performance in the simulator diminishes as training progresses has to do with the way transfer learning was applied in this case. Remember that this network originates from a pretrained network trained on the lake dataset. All layers but the fully-connected layers were then set non-trainable. This means that all the convolutional layers are fixed, effectively being trained for the lake track. Armed with the knowledge that features in early convolutional layers are more generic than features in later layers, it might not be optimal to freeze *all* convolutional layers. If the source and target domains are dissimilar it may be preferable to only freeze the early convolutional layers while letting the weights of later ones remain trainable. This suggests that the procedure described in Section 3.7.1 under scenario (a), which was evaluated with the feature extractor network, might not be the best option. It is thus also plausible that the notion of scenario (a) that "the new dataset is small and the domains are similar" might not be valid.

6.4.2 Fine-tuning network 1

The fine-tuning network performs very well with just a small amount of training time. After 6 epochs it clearly performs better than both the feature extractor and conventional network. With some more training its performance remains rather stable, in some cases performing worse than the conventional network.

6.4.3 Fine-tuning network 2

This network performs very well overall. With just 7 epochs of training it manages to finish the entire track at all speeds tested. It can thus be concluded that fine-tuning seems to be an appropriate method of choice in the setting studied in this thesis. There is however a decline in performance in the simulator at epoch 18. This behaviour cannot be explained with overfitting, since with more training the network performs well again in the simulator. It could not have been predicted from looking at Figure 14, since there is no sign of an increase of validation loss around that epoch.

This again raises the question to what degree the training and validation losses can directly be translated to generalization performance. Let us look at this behaviour in more depth.

The validation set does not use augmentation, as opposed to the training set. The validation data is thus just pure data (preprocessed) from the data collection phase. This means that it should be similar to the images that are predicted during the testing in the simulator. However, since the validation dataset is rather small, there will inevitably be situations during the testing where the car happens to be in situations which are not represented in the validation dataset, meaning that the validation loss cannot fully represent the true generalization performance. It therefore seems plausible that the optimization can find itself in some point where the training and validation losses are small, but where generalization is not necessarily good. It can thus be concluded that the training and validation losses cannot be the final way of evaluating the performance, rather it has to be done in the true testing environment, i.e. the simulator. This is not a new finding, but it is still an interesting and very important one and can explain the sudden decline of generalization performance at epoch 18.

6.5 Task 3

The conventional network takes about twice the amount of time to train compared to the transfer learning networks. With only a small number of training epochs, it also performs worse than the other networks, especially the fine-tuning networks. However, with more training time it actually outperforms the feature extractor network and the fine-tuning network 1, see Table 2.

With the conventional network trained, we are now in a position to evaluate the method of transfer learning compared to classic machine learning. Let us here only consider the case where there is no tight constraint on training time. For now, assume that the networks have been trained for at least 24 epochs. The conventional network then performs better than the feature extractor and fine-tuning network 1. Its performance is similar to the fine-tuning network 2, performing worse in just one out of eight cases, where `speed_constant` = 1.3 and `number_epochs` = 24, see Table 2.

From the performance of the feature extractor and fine-tuning network 1, compared to the conventional network, it can be concluded that the source and target domains are not similar enough to afford keeping three or more of the first convolutional layers fixed. However, keeping only two convolutional layers fixed seems to work really well, judging by the perfomance of the fine-tuning network 2. This indicates that the procedure described in Section 3.7.1 under scenario (b) seems to work really well. By extension, this means that the notion that "the new datasset is large and the domains are similar" seems to be somewhat valid. It may sound contradictory to previous statements in the report, where it is argued that the new dataset is in fact relatively small. However, with augmentations the dataset is enlargened, arguably making its 'effective' size not small anymore. Furthermore, the size of the dataset should probably be viewed keeping the number of parameters in the network in mind. In other words, what is meant by a small and a large dataset is somewhat arbitrary, though it can be made sensible by taking into consideration the number of parameters of the network. For example, if few parameters can bring sufficient complexity to the problem at hand, a dataset with relatively few examples might be considered a medium or even large dataset. Comparatively, with that same dataset and a model with a large number of parameters, it may be considered a small dataset. For the network used in this thesis, which contains about 250 thousand parameters, the training dataset consisting of about 4 thousand images and then add augmentations on top of this, we may reject the hypothesis that the effective size of the dataset is small. This can be compared to the ratio of 138 million parameters trained on a dataset with 1.3 million examples which was used when training the VGG16 network, achieving state-of-the-art results on image recognition tasks a couple of years ago [27].

7 Conslusions

This thesis has investigated the potential benefits of using transfer learning when training convolutional neural networks for the task of autonomously driving a car. Three transfer learning networks were trained and compared with a conventionally trained convolutional neural network. The first conclusion is that training is, as expected, considerably quicker when using transfer learning. More specifically, transfer learning utilizes pretrained networks. This means that the entire network needs not be trained, rather just parts of it. Since less weights have to be updated per epoch of training, this results in faster training. Here training was approximately twice as fast per epoch compared to training a non-transfer learning network.

By keeping a varying number of convolutional layers from the pretrained network fixed, three networks were trained using transfer learning. This way, knowledge is said to be transferred from a previously solved problem to the current problem. The network called fine-tuning network 2 performed better than the other two transfer learning networks when tested in the simulator. After only 7 epochs of training, this network could drive around the entire track without going

off-road, even at higher speeds. This network was to be compared with the so called conventional network, trained under a classic machine learning setting. Not until having been trained for 18 epochs did the performance of the conventional network match the performance of the fine-tuning network 2, trained for merely 7 epochs. In other words, in less than half the number of epochs, and with each epoch training twice as fast, a network using transfer learning performed as well as a non-transfer learning network.

8 Further work

Further work may include investigating how little data is sufficient to successfully train a transfer learning network compared to a conventional network. It could also be interesting to train networks using transfer learning in more complex driving environments. For example, CARLA is another simulator with more complex environments [8] compared to Udacity's simulator. It is a city environment that also simulates other cars and pedestrians, meaning that some more advanced techniques would have to be used to handle the increase in environment complexity. The CARLA simulator could also be used to investigate the possibilities of transfer knowledge from more distant domains, i.e. train with data from Udacity's simulator and test in the CARLA simulator. This could also be investigated using publicly available CNNs that have been pretrained on large datasets for tasks such as image recognition. One could for instance imagine that it would be possible to fine-tune pretrained networks such as for example VGG16 for the task of autonomously driving a car.

9 References

[1] Yoshua Bengio, *Practical Recommendations for Gradient-Based Training of Deep Architectures*, arXiv:1206.5533 [cs.LG]

[2] James Bergstra, Yoshua Bengio, *Random Search for Hyper-Parameter Optimization*, Journal of Machine Learning Research, 2013

[3] Chollet, François et al., *Keras*, 2015, https://keras.io

[4] Christopher M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006

[5] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, Karol Zieba, *End to End Learning for Self-Driving Cars*, 2016, arXiv:1604.07316 [cs.CV]

[6] Djork-Arné Clevert, Thomas Unterthiner, Sepp Hochreiter, *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*, 2015, arXiv:1511.07289 [cs.LG]

[7] Yann Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, Yoshua Bengio, *Identifying and attacking the saddle point problem in high-dimensional non-convex optimization*, 2014, arXiv:1406.2572 [cs.LG]

[8] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, Vladlen Koltun, *CARLA: An Open Urban Driving Simulator*, 2017, arXiv:1711.03938 [cs.LG]

[9] Ian Goodfellow, Yoshua Bengio, Aaron Courville, *Deep Learning*, MIT Press, 2016

[10] Kurt Hornik, *Multilayer Feed-Forward Networks are Universal Approximators*, https://www.cs.cmu.edu/~epxing/Class/10715/reading/Kornick_et_al.pdf, collected 051218

[11] Sergey Ioffe, Christian Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, 2015, arXiv:1502.03167 [cs.LG]

[12] Diederik P. Kingma, Jimmy Ba, *Adam: A Method for Stochastic Optimization*, 2014, arXiv:1412.6980 [cs.LG]

[13] Robert Kleinberg, Yuanzhi Li, Yang Yuan, *An Alternative View: When Does SGD Escape Local Minima?*, 2018, arXiv:1802.06175 [cs.LG]

[14] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, *ImageNet Classification with Deep Convolutional Neural Networks*, 2012, http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[15] Yann LeCun, Yoshua Bengio Geoffrey Hinton, *Deep Learning*, 2015, https://creativecoding.soe.ucsc.edu/courses/cs523/slides/week3/DeepLearning_LeCun.pdf

[16] Jie Lu, Vahid Behbood, Peng Hao, Hua Zuo, Shan Xue, Guangquan Zhang, *Transfer learning using computational intelligence: A survey*, 2015, https://www.sciencedirect.com/science/article/abs/pii/S0950705115000179

[17] Tom M. Mitchell, *Machine Learning*, McGraw-Hill Education, 1997

[18] Andrew Ng, Jiquan Ngiam, Chuan Yu Foo, Yifan Mai, Caroline Suen, Adam Coates, Andrew Maas, Awni Hannun, Brody Huval, Tao Wang, Sameep Tandon http://ufldl.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/, Stanford, collected 051218

[19] Choon-Boon Ng, Yong-Haur Tay, Bok-Min Goi, *Comparing Image Representations for Training a Convolutional Neural Network to Classify Gender*, 2013, http://ijssst.info/Vol-15/No-2/data/3251a024.pdf

[20] Mattias Ohlsson, Lecture notes for the class "Introduction to Artificial Neural Networks and Deep Learning" given at Lund Institute of Technology during 2017

[21] Maxime Oquab, Leon Bottou, Ivan Laptev, Josef Sivic, *Learning and Transferring Mid-Level Image Representations using Convolutional Neural Networks*, 2014, http://openaccess.thecvf.com/content_cvpr_2014/papers/Oquab_Learning_and_Transferring_2014_CVPR_paper.pdf

[22] Sashank J Reddi, Manzil Zaheer, Suvrit Sra, Barnabas Poczos, Francis Bach, Ruslan Salakhutdinov, Alexander J Smola, *A Generic Approach for Escaping Saddle points*, 2017, arXiv:1709.01434 [cs.LG]

[23] Stuart Russel, Peter Norvig, *Artificial Intelligence, A Modern Approach*, Pearson, 1994

[24] Jürgen Schmidhuber, *Deep Learning in Neural Networks: An Overview*, arXiv:1404.7828 v4 [cs.NE]

[25] Daniel Shapiro, 2017 https://towardsdatascience.com/artificial-intelligence-and-bad-data-fbf2564c541a, Towards Data Science, collected 051218

[26] Patrice Y. Simard, Dave Steinkraus, John C. Platt, *Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis*, 2003, http://cognitivemedium.com/assets/rmnist/Simard.pdf

[27] Karen Simonyan, Andrew Zisserman, *Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2015, arXiv:1409.1556 [cs.CV]

[28] Jasper Snoek, Hugo Larochelle, Ryan P. Adams, *Practical Bayesian Optimization of Machine Learning Algorithms*, 2012, arXiv:1206.2944 [stat.ML]

[29] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov, *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, Journal of Machine Learning Research, 2014

[30] Pavel Surmenok, 2017, https://towardsdatascience.com/estimating-optimal-learning-rate-for-a-deep-neural-network-ce32f2556ce0, Towards Data Science, collected 051218

[31] Ben Tan, Yu Zhang, Sinno Jialin Pan, Qiang Yang, *Distant Domain Transfer Learning*, 2017, http://www.ntu.edu.sg/home/sinnopan/publications/[AAAI17]Distant%20Domain%20Transfer%20Learning.pdf

[32] Anish Singh Walia, 2017 https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f, Towards Data Science, collected 051218

[33] Liu Yang, Steve Hanneke, Jaime Carbonell, *A Theory of Transfer Learning with Applications to Active Learning*, 2013, http://www.cs.cmu.edu/~liuy/atl.pdf

[34] Matthew D. Zeiler, Rob Fergus, *Stochastic Pooling for Regularization of Deep Convolutional Neural Networks*, 2013, arXiv:1301.3557 [cs.LG]

[35] Zhang, J.; Lee, T.-Y.; Feng, C.; Li, X.; Zhang, Z., *Robust Attentional Pooling via Feature Selection*, 2018, https://www.merl.com/publications/docs/TR2018-124.pdf

[36] http://www.cs.cornell.edu/courses/cs6787/2017fa/Lecture7.pdf, Cornell University, Computer Science, collected 051218

[37] http://cs231n.github.io/convolutional-networks/#conv, Stanford, collected 051218

[38] http://cs231n.github.io/transfer-learning/, Stanford, collected 051218

[39] https://en.wikipedia.org/wiki/Stochastic_gradient_descent, Wikipedia, collected 051218

[40] https://en.wikipedia.org/wiki/Convolution, Wikipedia, collected 051218

Master's Theses in Mathematical Sciences 2019:E1
ISSN 1404-6342

LUTFMA-3373-2019

Mathematics
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/