1 Machine Learning Theory

Throughout this section the theory of machine learning that will be used in this thesis is described and explained. In Section 1.1 we describe the general motivation behind using machine learning. Then, in Section 1.2 we explain the three most common paradigms in machine learning and how we will use some of them. In Section 1.3 we give a brief overview of how to evaluate a developed model. Lastly, in Section 1.4 we describe the mathematics behind feedforward- and convolutional neural networks .

1.1 Motivation

It can be difficult for humans to recognize certain patterns and trends in data. This becomes more difficult the greater the quantity of the data is, which is becomming more and more common with the rapidly growing topic of *Big Data*. For this reason, computers are often used instead of humans to recognize patterns and trends in the data by analyzing the data, which is what is called *Machine Learning*. In this thesis, we will use machine learning in section **??** to develop a model to estimate the 2D pose of a single human in an image. Later, in section **??**, we will use machine learning to improve our understanding of the model.

1.2 Machine Learning Paradigms

Machine learning is usally split into the at least two

1. *Supervised learning* where the data consists of features and labels. By analyizing the data the algorithm learns to predict the labels given the features [2]. Supervised learning is further split into *classification* and *regression*.

2. *Unsupervised learning* where the data only consists of features. The algorithm then learns properties of the data, without any provided labels [2].

In this thesis we will make use of supervised learning when developing our model for pose estimation, where we will make use of regression. Later, unsupervised learning is used when we explore our developed model.

1.3 Evaluation of Machine Learning Models

When developing a machine learning model it is important to know how trustworthy the developed model is. This is usually done by testing how good the model is at generalizing to unseen data, which is done by making use of several dataset splits and *evaluation metrics*.

1.3.1 Splitting the dataset

When developing a machine learning model, the data needs to both create the model, but also to evaluate the model. For the evaluation of the model, one of the two following techniques is usually used

1. *Cross validation* where the data is split into K non-overlapping chunks. The model is then trained for K rounds on $K-1$ chunks, where the last chunk is used for evaluating the model [9].

2. *Train-validation-test* where the data is split into 3 random non-overlapping chunks. The training dataset is then used for training the model and the validation dataset is used for evaluating the model as it is being developed - this often means, that the *hyperparameters*, the paramters that are not possible to fit from the data, are being tweaked to yield the

best validation loss. Lastly, the testing dataset is used as a final evaluation of the model to yield an unbiased evaluation of the ability of the model to generalize to unseen data. Once the testing dataset has been used it can no longer be used for evaluating the data, as this potentially introduces bias in future evaluations [3].

Throughout this thesis the train-validation-test technique will be used over cross validation for evaluating the developed models. This is done, since cross validation is better suited for smaller datasets, as the runtime is much greater than the runtime of the train-validation-test technique.

1.3.2 Evaluation Metrics for Supervised Machine Learning (Loss Functions)

When we have trained a model, we need to somehow evaluate how well the model performs on unseen data. This is usually done by making use of evaluation metrics or *loss functions*. There are many different loss functions, each with their own advantages and disadvantages. One of the most common loss functions for regression is the *Mean Squared Error (MSE)*, defined as

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where y_i is the true value of the ith observation and $\hat{y}_i$ is the estimated value of the ith observation. Thus, MSE measures the average squared difference between the true observation and the estimated observation. The aim of a model is thus to make the MSE as small as possible [4].

1.4 Neural Networks

In recent years *deep learning* and *neural networks* have revolutionized the use of machine learning. In this thesis a neural network will be used for performing the human pose estimation. Throughout subsection 1.4 the theory and mathematics behind neural networks is described and explained.

1.4.1 Feedforward Neural Networks

Algorithm 1 Gradient Descent [13]

Require: Learning rate η
Require: Starting position $\boldsymbol{\theta}$
Require: Function to minimize f
1: **while** stopping criterion not met **do**
2: Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} - \eta\nabla f(\boldsymbol{\theta})$
3: **return** $\boldsymbol{\theta}$

Overfitting and Regularization
The main goal of a machine learning model is to generalize well on unseen data. This can often be difficult, as the model simply "remembers" the training data instead of learning the patterns in the training data. In other words, the gap between the training error and the test error is too large, which is a concept called *overfitting*. Certain techniques are designed to reduce the test error - these techniques are collectively called *regurlization* [1].

Gradient Descent

The goal of a machine learning model when training is to minimize its loss. There are different methods to do so, however, the most common algorithms are variants of *gradient descent*, whose algorithm is described in Algorithm 1. The algorithm works by taking a learning rate η, a starting position $\boldsymbol{\theta}$ and a function f as input, where f is the (loss) function to be minimized. It then computes $\nabla_{\boldsymbol{\theta}} f$ and subtracts the gradient times η from $\boldsymbol{\theta}$. This is done until a stopping condition is met, such as when the magnitude of the gradient $|\nabla f(\boldsymbol{\theta})|$ is small or until a maximum amount of iterations has been reached [13].

Online, Mini-batch and Batch methods
When gradient descent is used in machine learning, computing $\nabla f(\boldsymbol{x})$ can be done by averaging the gradient of each of the n observations of the training set, which is called a *batch gradient method* and is computational inefficient, as the cost is $\mathcal{O}(n)$. It is therefore common to use variants of gradient descents, that reduces the cost of computing the gradient. In *online gradient methods* a single observation from the dataset is used to compute the gradient, which brings the cost down to $\mathcal{O}(1)$. In *mini-batch gradient methods* a subset of the dataset is used to compute the gradient, making the cost $\mathcal{O}(|\mathcal{B}|)$, where $|\mathcal{B}|$ is the mini-batch size [13].
Choosing the right batch size can be difficult, however, there are a few guidelines which one can follow [6] [13]

1. Batch gradient descent uses the fewest iterations, however, each iteration takes the longest to compute. On the other hand, in online gradient descent each iteration is the fastest to compute, however, it is also the method that uses the most iterations. Lastly, mini-batch gradient descent combines the two: it uses less iterations than online gradient descent, but more than batch gradient descent, and each iteration takes less time than in the case with batch gradient descent, but longer than in the case with online gradient descent.

2. A batch size that is of power of 2 can offer in better runtime for some hardware. A batch size that is often used for larger models is 16, however, they typically range between 32 and 256.

3. Smaller batch sizes can offer a regularizing effect, as it is difficult for the model to "remember" the complete dataset from batches that does not represent the whole dataset.

Algorithm 2 RMSProp [1]

Require: Learning rate η
Require: Decay rate ρ
Require: Starting position $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}
Require: Function to minimize f
Initialize accumulation variables $\boldsymbol{r} = \boldsymbol{0}$
while stopping criterion not met **do**
  Sample a minibatch of m observations from the training set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$
  Compute gradient: $\boldsymbol{g} = \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i f\left(\boldsymbol{x}^{(i)}\right)$
  Accumulate squared gradient: $\boldsymbol{r} = \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
  Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\eta}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$
  Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
return $\boldsymbol{\theta}$

Optimization Algorithms
Online, mini-batch and batch gradient descent are all optimization algorithms used for estimating the minimum of a function. One problem of these algorithms is, that the learning rate

can be difficult to choose. Therefore, there have been developed a range of various optimization algorithms that uses a separate learning rate for each parameter and automatically adapt these learning rates. One of which is *RMSProp*, which has been visualized in Algorithm 2. The algorithm works by using an decaying average that discards knowledge from the past, so that it can converge after finding a convex bowl. The algorithm uses a hyperparameter ρ, that controls the length scale of the moving average [1].

Algorithm 3 Stochastic Gradient Descent [1]

Require: Learning rate η
Require: Initial parameter $\boldsymbol{\theta}$
Require: Function to minimize f
1: **while** stopping criterion not met **do**
2: Sample a minibatch of m observations from the traning set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(n)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$
3: Compute gradient estimate: $\hat{\boldsymbol{g}} = \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i f\left(\boldsymbol{x}^{(i)}\right)$
4: Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} - \eta\hat{\boldsymbol{g}}$

Another important optimization algorithm is *Stochastic gradient descent* (*SGD*). Unlike RMSProp, SGD does not adapt the learning rate over time, but instead keeps it fixed. The algorithm of SGD has been visualized in Algorithm 3. SGD is very closely related to the algorithm behind gradient descent, however, instead of updating the parameters for each sample, SGD instead uses the mean of n samples. The momentum algorithm works by accumulating a decaying moving average of the past gradients and continuing to move in their direction.

Algorithm 4 Stochastic Gradient Descent with Momentum [1]

Require: Learning rate η
Require: Momentum parameter α
Require: Initial parameter $\boldsymbol{\theta}$
Require: Initial velocity $\boldsymbol{v}$
Require: Function to minimize
1: **while** Stopping criterion not met **do**
2: Sample a minibatch of m observations from the traning set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(n)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$
3: Compute gradient estimate: $\boldsymbol{g} = \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i f\left(\boldsymbol{x}^{(i)}\right)$
4: Compute velocity update: $\boldsymbol{v} = \alpha\boldsymbol{v} - \eta\boldsymbol{g}$
5: $\boldsymbol{\theta} = \boldsymbol{\theta} + \boldsymbol{v}$

Momentum

One problem with SGD is how slow it often can be. For this reason *momentum* is often used to accelerate learning. The algorithm behind SGD with momentum has been visualized in Algorithm 4. The algorithm works by introducing two new variable; $\boldsymbol{v}$, which is the direction and speed of which the parameters move through the parameter space, and $\alpha \in [0, 1)$, which describes how quickly the contribution of previous gradients decay. Common values of α are $0.5, 0.9$ and 0.99 [1].

Batch Normalization

Batch normalization is a reparametrization method, which is applied to individual layers in a neural network. If $\boldsymbol{x} \in \mathcal{B}$ is an input to the batch normalization, BN, then batch normalization

is done by the following

$$\text{BN}(\boldsymbol{x}) = \boldsymbol{\gamma} \odot \frac{\boldsymbol{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}}}{\hat{\boldsymbol{\sigma}}_{\mathcal{B}}} + \boldsymbol{\beta}$$

where

$$\hat{\boldsymbol{\mu}} = \frac{1}{|\mathcal{B}|} \sum_i \mathcal{B}_i$$

and

$$\hat{\boldsymbol{\sigma}} = \sqrt{\epsilon + \frac{1}{|\mathcal{B}|} \sum_i (\mathcal{B} - \boldsymbol{\mu})_i^2},$$

which makes the minibatch have 0 mean and unit variance. $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are then used to make the mini-batch have an arbitrary mean and standard deviation and are two parameters that needs to be learned when the network is being fitted. This helps the network to converge, as the batch normalization keeps centering the mean and standard deviation of the mini-batches [13].

Epoch

An *epoch* is an iteration through the complete dataset during fitting of the network. Multiple epochs are often needed to reach the minimum of the loss function [13].

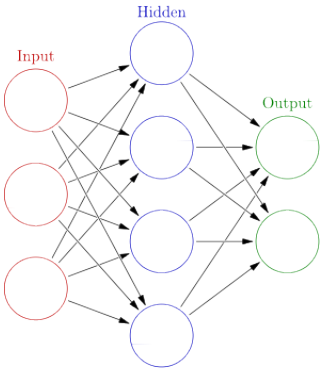


Figure 1: Visualization of a feedforward neural network with a single hidden layer [12]

The Architecture and Forwardpropagation

One of the most common types of neural networks are *feedforward neural networks*, where the data flows unidirectionally through the network. Such a network is visualized in Figure 1. The network is a directed acyclic graph and is built up of three types of components: the *input layer*, the *hidden layers* and the *output layer*. Each layer is built up of *units*, also called *neurons* (which are visualized as circles in Figure 1), where each neuron has a *bias* assigned to it, and is connected to one or two other layers through *edges* (which are visualized as arrows in Figure 1), where each edge has a *weight* assigned to it. Hidden layers are connected to two other layers - one before the hidden layer and one after the hidden layer - where the input layer is only connected to the next layer in the network and the output layer is only connected to the previous layer in the network.

We can define the network mathemaically by letting $a_n^{(i)}$ denote the value of the nth node in the ith layer, $w_{m,n}$ denote the value of the weight of the edge connecting the nth node in the ith layer to the mth node in layer $i+1$ and $b_n^{(i)}$ denote the bias corresponding to the nth node in the ith layer.

When data flows through the model it follows the following formula

$$\boldsymbol{a}^{(i+1)} = g^{(i+1)}\left(\boldsymbol{z}^{(i+1)}\right)$$

where

$$\boldsymbol{z}^{(i+1)} = \boldsymbol{W}^{(i+1)}\boldsymbol{a}^{(i)} + \boldsymbol{b}^{(i+1)},$$

$\boldsymbol{W}^{(i+1)}$ is the weights between layer i and layer $i+1$ defined by

$$\boldsymbol{W}^{(i+1)} = \begin{pmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,0} & w_{m,1} & \cdots & w_{m,n} \end{pmatrix},$$

$\boldsymbol{a}^{(i)}$ is the values of the nodes in the ith layer defined by

$$\boldsymbol{a}^{(i)} = \begin{pmatrix} a_0^{(i)} \\ a_1^{(i)} \\ \vdots \\ a_n^{(i)} \end{pmatrix},$$

$\boldsymbol{b}^{(i+1)}$ is the values of the biases of layer $i+1$ defined by

$$\boldsymbol{b}^{(i+1)} = \begin{pmatrix} b_0^{(i+1)} \\ b_1^{(i+1)} \\ \vdots \\ b_m^{(i+1)} \end{pmatrix}$$

and g is a non-linear *activation function*, that is typically applied element-wise [1] [11]. The purpose behind using non-linear activation functions is to make it possible for the output of the network to be non-linear. One often used activation function is the *rectified linear activation function* (or *ReLU* for short) defined by

$$g(x) = \max\{0, x\}.$$

The ReLU-function is very close to being linear, making the function keep many of the properties of linear functions that make them easy to optimize and generalizing, which are two great advantages of using the ReLU-function. Another great advantage of using the ReLU-function is stated by the *universal approximation theorem* which states, that a feedforward network with a linear output layer and at least one hidden layer with the ReLU-function (or another non-linear activation function from a wide class of activation functions) can approximate any continuous function on a closed and bounded subset of $\mathbb{R}^n$ (and actually some functions outside of this class), as long as the network has enough hidden neurons [1].

Backpropagation

Backpropagation is an algorithm used to compute the gradient of the network. It is used together with an optimization algorithm, such as RMSProp, to train the model by minimizing the training loss of the model. Backpropagation happens after data has flowed through the model from the input to the output, and works by computing the gradient of each parameter sequentially from the output to the input of the model. The procedure makes heavily use of the *chain rule* from calculus, which states, that if we let $\boldsymbol{x} \in \mathbb{R}^m$, $\boldsymbol{y} \in \mathbb{R}^n$, g be a function that maps from $\mathbb{R}^m$ to $\mathbb{R}^n$ and f be a function that maps from $\mathbb{R}^n$ to $\mathbb{R}$, then, if we let $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$, we can then compute $\frac{\partial z}{\partial x_i}$ by

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{y_j}{x_i}$$

[1]. If we use this to find the gradient of each parameter, we will find, that the partial derivative for each weight is

$$\frac{\partial L}{\partial w_{jk}^{(i)}} = \frac{\partial z_j^{(i)}}{\partial w_{jk}^{(i)}} \frac{\partial a_j^{(i)}}{\partial z_j^{(i)}} \frac{\partial L}{\partial a_j^{(i)}} = a_k^{(i-1)} g'^{(i)}\left(z_j^{(i)}\right) \frac{\partial L}{\partial a_j^{(i)}}$$

and the partial derivative of each bias is

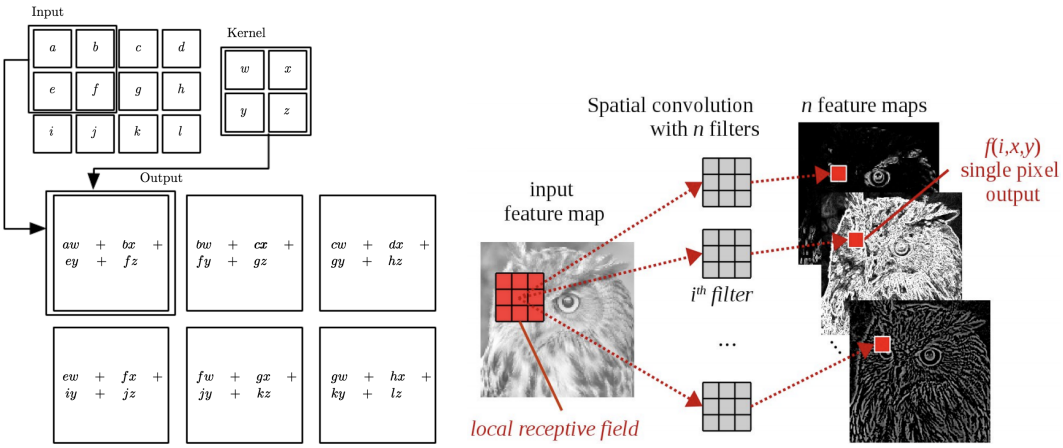
$$\frac{\partial L}{\partial b_j^{(i)}} = \frac{\partial z_j^{(i)}}{\partial b_j^{(i)}} \frac{\partial a_j^{(i)}}{\partial z_j^{(i)}} \frac{\partial L}{\partial a_j^{(i)}} = g'^{(i)}\left(z_j^{(i)}\right) \frac{\partial L}{\partial a_j^{(i)}}$$

where for both cases

$$\frac{\partial L}{\partial a_j^{(i)}} = \sum_{j=0}^{n_i - 1} w_{jk}^{(i+1)} g'^{(i+1)}\left(z_j^{(i+1)}\right) \frac{\partial L}{\partial a_j^{(i+1)}}.$$

if $\boldsymbol{a}^{(i)}$ is not the output-layer. Once the partial derivative of all weights and biases has been found, the gradient vector can be formed and an optimization method can be used to optimize the parameters of the model [10].

1.4.2 Convolutional Neural Networks



(a) Example of a 2D convolution [1]

(b) Example of a convolution with multiple filters [7]

Figure 2: Convolutions visualized

Feedforward neural networks introduced in 1.4.1 can be used for pattern recognition within images, however, they are usually not used for this task. Consider a colored input image with 3 channels of dimension 64×64. If we were to use a feedforward neural network on this image, each neuron in the first hidden layer would be connected to the input layer through 12.288 weights. Not only would this use a lot of computational power and time to train, however, a network of this size would also be prone to overfitting. Instead, *convolutional neural networks* (also known as CNN's) are usually used. A CNN usually consists of *convolutional layers*, *pooling layers* and *fully-connected layers*, where the fully-connected layers are analogous to the layers in a feedforward neural network [5].

Convolutional layers

A convolutional layer is composed of a set of *kernels* (also known as *filters*), which are matrices of weights of dimension $k \times k$, where k usually is 5, 3 or 1, and each weight is a parameter

for the model to be learned [7]. Each kernel is used on the input to produce a *feature map*. The kernels are applied to the input by "sliding" over the input (where the step size is called *stride* and is usually by default equal to 1). Each $k \times k$ grid of the input (called the *local receptive field*) is then used to compute the dot-product between the grid and each kernel, which is then placed in the corresponding feature map of each kernel, as visualized in Figure 2. When all of the feature maps have been computed, the feature maps are stacked together, forming a tensor, which is then returned by the layer and a activation function can be applied.
As described previously, by using a convolutional layer we can dramatically decrease the amount of weights used by the layer. If we were to use a 3×3 kernel on a colored image, we would reduce the amount of weights on each neuron in the following layer from 12.288 down to just 27, reducing both the training time and making the network less prone to overfitting [5].

Pooling layers
Pooling layers are layers used to reduce the dimension of the input. The most common pooling layer is the *maxpooling*-operation. The operation works by considering each $k \times k$ grid, like in the case with the convolutional layer, in which the maximum entry in that grid is being inserted into the output [5].

Algorithm 5 Nearest Neighbour Upsampling [8]

Require: Input image X of size $m \times n$
Require: Wanted output size $xm \times yn$, where $x, y \in \mathbb{Z}^+$
1: Create empty image O of size $xm \times yn$
2: **for each** pixel, p, in X **do**
3: $\quad i, j =$ index of p in X
4: $\quad$ Insert p at index (xi, yj) in O
5: **for each** empty pixel $p \in O$ **do**
6: $\quad$ Let p be the value of the nearest neighbour
7: **return** O

Nearest Neighbour Upsampling
Sometimes we want to increase the size of an image. This can be done by making use of *upsampling* (also known as *interpolation*) techniques. One of the most common upsampling techniques is *nearest neighbour upsampling*, whose pseudocode has been written in Algorithm 5. The algorithm starts off by taking an image, X, of size $m \times n$, as input, which we wish to upsample to size $xm \times yn$, where x and y are positive whole numbers. The algorithm then loops over each pixel, p, in X, finds the corresponding index, i, j, of p in X, and places p at index (xi, yj) in the output image O of size $xm \times yn$. When this is done, it assigns each of the empty pixel in O the value of their nearest neighbour, making each pixel in O have a value, and then returns O [8].