1 Machine Learning Theory

Throughout this section the theory of machine learning that will be used in this thesis is described and explained.

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 **MANGLER REFERENCE** to develop a model to estimate the 2D pose of a single human in an image. Later, in section **MANGLER REFERENCE**, we will use machine learning to improve our understanding of the model.

1.2 Machine Learning Paradigms

Machine learning is usally split into the following three paradigms

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 [8]. Supervised learning is further split into *classification* and *regression*. If the value of each label is limited, then the task is a classification task. If the value of each label is not limited, then the task is a regression task.

2. *Unsupervised learning* where the data only consists of features. The algorithm then learns properties of the data, without any provided labels [8].

3. *Reinforcement learning* where the algorithm learns to perform the action in a given environment that yields the highest reward [1].

In this thesis we will make use of supervised learning when developing our model for pose estimation. 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 unseen data, which is done by making use of *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 random non-overlapping chunks of equal size. The model is then trained for K rounds on $K-1$ of the chunks, where the last chunk is used for evaluating the model. After each round the parameters of the model is reset to ensure one round does not affect another round. After the K rounds the average loss of the K rounds is the loss of the model [7].

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 model. Once the testing dataset has been used it can no longer be used for evaluating the data, as this ensure an unbiased evaluation [4].

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 [2].

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 The Mathematics Behind Neural Networks

Algorithm 1 Estimates $\operatorname{argmin}_{\boldsymbol{x}}$ of function f [10]

Require: Learning rate η
Require: Starting position $\boldsymbol{x}$
1: **procedure GradientDescent**
2: **while** stopping criterion not met **do**
3: Apply update: $\boldsymbol{x} \leftarrow \boldsymbol{x} - \eta\nabla f(\boldsymbol{x})$
4: **return** $\boldsymbol{x}$

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* [3].

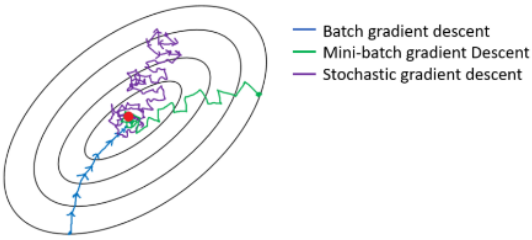

Figure 1: Comparison of batch, mini-batch and stochastic gradient descent [5]

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 $\boldsymbol{x}$ and a function f as input, where f is the function to minimize and η. It then computes the gradient of f with respect to $\boldsymbol{x}$, and subtracts the gradient times η from $\boldsymbol{x}$. This is done untill a stopping condition is met, such as when the magnitude of the gradient $|\nabla f(\boldsymbol{x})|$ is small or untill a maximum amount of iterations has been reached [10].

Online, Mini-batch and Batch methods
When gradient descent is used for in machine learning, computing $\nabla f(\boldsymbol{x})$ is usually done by averaging the gradient of each of the n observations of the trainingset, 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* (also known as *stochastic gradient descent*) 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 [10].
Choosing the right batch size can be difficult, however, there are a few guidelines which one can follow [5] [10]

1. Looking at Figure 1 we see, that 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 power of 2 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.

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.

Algorithm 2 Estimates $\operatorname{argmin}_{\boldsymbol{\theta}}$ of loss function L [3]

Require: Learning rate η
Require: Decay rate ρ
Require: Starting position $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}

procedure RMSProp
  Initialize accumulation variables: $\boldsymbol{r} \leftarrow 0$
  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: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
    Compute parameter update: $\Delta\theta = -\frac{\eta}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  return $\boldsymbol{\theta}$

The algorithm works by using an decaying average that discard 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 [3].

Batch Normalization
Batch normalization is a recent adaptive reparametrization method, which is applied to individual layers. Let $\boldsymbol{\mathcal{B}}$ be a minibatch of activations of the layer to normalize, where each row corresponds to the activations of a unique observation. To normalize $\boldsymbol{\mathcal{B}}$ we do

$$\boldsymbol{\mathcal{B}} = \frac{\boldsymbol{\mathcal{B}} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}$$

where

$$\boldsymbol{\mu} = \frac{1}{|\boldsymbol{\mathcal{B}}|} \sum_{\boldsymbol{x} \in \boldsymbol{\mathcal{B}}} \boldsymbol{x}$$

and

$$\boldsymbol{\sigma} = \sqrt{\delta + \frac{1}{|\boldsymbol{\mathcal{B}}|} \sum_{\boldsymbol{x} \in \boldsymbol{\mathcal{B}}} (\boldsymbol{\mathcal{B}} - \boldsymbol{\mu})^2}$$

where δ is positive number, close to 0, which is used to avoid division by zero when normalizing $\boldsymbol{\mathcal{B}}$.

Epoch
An *epoch* is an iteration through the whole dataset during fitting of the network. Multiple epochs are often needed to reach the minimum of the loss function [10].

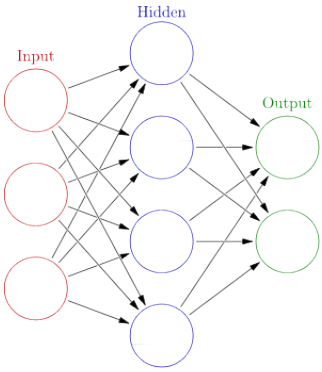

Figure 2: Visualization of a feedforward neural network with a single hidden layer [9]

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 2. The network 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 2), 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 2), 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 $h_n^{(i)}$ 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

$$\boldsymbol{h}^{(i+1)} = g^{(i+1)}\left(\boldsymbol{W}^{(i+1)}\boldsymbol{h}^{(i)} + \boldsymbol{b}^{(i+1)}\right)$$

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

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

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

$$\boldsymbol{h}^{(i)} = \begin{pmatrix} h_0^{(i)} \\ h_1^{(i)} \\ \vdots \\ h_n^{(i)} \end{pmatrix},$$

$\boldsymbol{b}^{(i+1)}$ 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 an *activation function*, that is typically applied element-wise [3] [6]. 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* that states, that a feedforward network with a linear output layer and at least one hidden layer with the ReLu-function (or another activation function from a wide class of activation functions) can approximate any continuous function on a closed and bounded subset of $\mathbb{R}^n$, as long as the network has enough hidden neurons [3].
Backpropagation

1.4.2 Convolutional Neural Networks

NN Upsampling
Maxpooling
Convolution

1.4.3 Stacked Hourglass

Reasoning behind using the Stached Hourglass
The Residual Modules
The Hourglass
The Stacked Hourglass

1.5 Principal Components Analysis and K-means Clustering

1.5.1 Principal Components Analysis (PCA)

1.5.2 K-means Clustering