CHAPTER 5

Introduction to Deep Learning

Deep learning, as a sub-field of machine learning, is concerned with the study of algorithms that have the ability to learn through experience. The experience typically is gained by presenting the algorithm with data examples from which it learns a set of rules. In the special case of deep learning the models at considerations are Deep Neural Networks (DNNs).
In this chapter first Artificial Neural Networks (ANNs) are introduced before turning to the more advanced DNNs. Two different types of DNNs namely Feedforward Neural Networks (FNNs) and Recurrent Neural Networks (RNNs) as well as their training algorithms will be discussed. Furthermore, different types of activation functions, weight initializations, and optimization algorithms are reviewed. Special emphasis is put on the application to supervised classification tasks (discussed in section **??**) which will be the kind of problems at hand in the latter chapters of this thesis.

5.1 Artificial Neural Networks

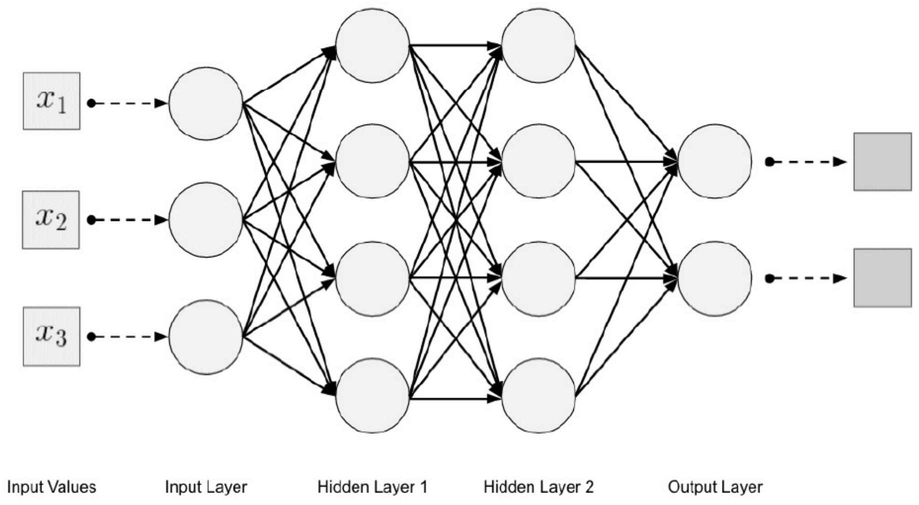

Figure 5.1: Schematic structure of a deep neural network with fully connected layers. The input values of the given task x_i are fed into the neurons (depicted as circles) of the input layer and processed in the hidden layers. The weight which connect the neurons are represented by a solid arrow in between the different layers. In the output layer, all information is combined and predictions (depicted as dark squares) about the classes are made. (Figure taken from [3])

As it is often the case for inventions in science and technology, the idea of ANNs is inspired by nature, more precisely by the human brain. The two key principles involved in the brain's decision-making process are recognizing patterns and inductive thinking [2]. These, for any machine learning model highly desirable attributes, are modeled by imitating the structure of the human brain in an ANN. The basic building blocks of the brain, neurons, process and transmit information i.e. electric and chemical signals. Depending on the input signals received by the neuron, the output transmitted to the next neuron is either 1 when an internal threshold is surpassed or 0 otherwise. The complicated processes summarize the input signals to the correct net input z are part of ongoing research in the field[???]. For the sake of ANNs, the net input z is defined as

$$z = b + \sum_i x_i w_i \tag{5.1}$$

where w_i are weights representing the assigned importance of the different features x_i. The bias term b is the artificial equivalent to the threshold in a biological neuron. Additionally, an arbitrary activation function $\Phi(z)$ is applied to the net input corresponding to the unknown degrees of freedom in the decision-making process and determining the final output of the artificial neuron.
The neurons in an ANN are structured as layers, shown in Figure 5.1. The first layer is called the *input layer*, containing all the input features. In practice, these input features are physical variables such as kinematic quantities of the final state particles. The layers following the input layer are called *hidden layers* if an ANN has more than one hidden layer it is considered a DNN and thus, by definition a deep learning model. The different layers are connected to their successors via weight vectors $\vec{w}$. Depending on the magnitude of the weight a neuron can focus on a combination of features provided by the neurons in the previous layer. The model complexity and flexibility, therefore, increases with the number of neurons n and layers ℓ it contains. In the case of fully connected layers, i.e. every neuron in a given layer is connected to all neurons in the previous layer, the number of free parameters (weights and bias) can be calculated according to

$$\sum_{\ell=1}^{L} n^{\ell-1} n^{\ell} + n^{\ell} \tag{5.2}$$

The last layer is called the *output layer* where all information is compressed and predictions are made. In this thesis, the prediction made is the affiliation of an event to a signal or background process. In terms of machine learning, such a problem is called a supervised classification task where the supervision is given by the correct assignment known from the simulation. To get the classification right, an ANN has to learn which combination of features ultimately results in the best separation between the different classes. This process is often revered to as *training* and will be discussed in the next section.

5.2 Training of Feed-Forward Neural Networks

There are several different types of Neural Network, each adjusted and optimized for their specific field of application. However, they all share a similar, gradient-based training procedure that can be best understood considering Feed-Forward Neural Networks[1] (FNNs).
The first step of any gradient-based method is the forward pass. The forward pass involves the evaluation of the net input and the activation functions for each neuron successively. The neuron outputs values $a = \Phi(z)$ are preserved for the necessary gradient calculations later in the training. The forward pass is applied for each event individually such that the predictions made in the output layer result in a prediction distribution p. This distribution is used to evaluate the current performance of the neural network based on a *loss function*.

[1] The name originates from the fact that in a FNN information is always passed forward. Different types of ANNs also involve a cyclic flow of information within one neuron (see section 5.3)

Loss function

The loss function $L(\theta)$ is a measure of how close p is to the true distribution y. It depends on the weight tensor θ which includes all trainable parameters[2]. The training of a FNN has converged if the trainable parameters are adjusted such that the loss function is at its global minimum i.e. for all events the predicted probability is as close as possible to the true class.
A commonly chosen loss function for classification task is the cross-entropy loss[???] defined for $K = 2$ classes and m events as

$$L(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{K} y_j^{(i)} \cdot \ln(p_j^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\ln(p^{(i)}) + (1 - y^{(i)})\ln(1 - p^{(i)})] \tag{5.3}$$

where $p_j^{(i)}$ is the predicted probability that the i^{th} instance belongs to class j and $y_j^{(i)}$ is either 1 if the true class of the instance is j or 0 otherwise. It can be derivate from the Kullback-Leibler Divergence [???] and thus interpreted as the likelihood that p could have been generated by y. Selecting the cross-entropy loss avoids small values in the gradient calculations which is key for fast convergance of the training algorithm discussed in the next paragraph.

Backpropagation algorithm

The most widely used training algorithm is called the Backpropagation algorithm [???]. In addition to the first forward pass, a second backward pass is performed. In the backward pass, weight updates for each weight in the ANN are calculated based on the neuron outputs obtained in the forward pass. The exact execution of the weight updates depends on the optimization algorithm chosen. Nevertheless, most optimization algorithms are variations of the simple gradient descent method. Gradient descent calculates the partial derivatives of the loss function for each weight, exploiting the chain rule. In the case of the cross-entropy loss for a single event, the derivative with respect to the weight between the j^{th} neuron in layer ℓ and the k^{th} neuron in pervious layer $\ell - 1$ is given by

$$\frac{\partial L}{\partial w_{jk}^{\ell}} = \frac{1}{n}\sum_{k=0}^{n} x_k^{\ell-1}(\phi(z_j^{\ell}) - y_j) \tag{5.4}$$

where the predicted distribution p in Equation **??** was expressed as a function of the n output features $a_k^{\ell-1}$ provided by the neurons in the previous layers. Equation 5.4 only holds for particular choices of the activation function for instance the sigmoid function $\frac{1}{1+e^{-z}}$. This often-used activation function can be especially problematic since its derivation amounts to small numerical values. However, since the partial derivative of the cross-entropy is independent of the neuron's output derivative, this problem can be avoided. Hence, gradient descent in combination with cross-entropy ensures fast coverage of the weight updates. The weight updates themselves are performed by adjusting the current weight $w_{jk}^{\ell}(t-1)$ to the new weight $w_{jk}^{\ell}(t)$ via

$$w_{jk}^{\ell}(t) = w_{jk}^{\ell}(t-1) - \alpha \cdot \frac{\partial L}{\partial w_{jk}^{\ell}} \tag{5.5}$$

where α is a tunable hyperparameter called the *learning rate* which determines the step size of the learning updates.
One complete iteration of the Backpropagation algorithm is called an *epoch*. It comprises the forward pass, the evaluation of the loss function, and updates of the weights based on a gradient method such as gradient descent. Typically a FNN needs to be trained over several epochs to obtain a well-separating model.

[2] The bias term can be reinterpreted as the weight $w_0 = -1$ of the weight tensor [???]

5.3 Recurrent Neural Networks

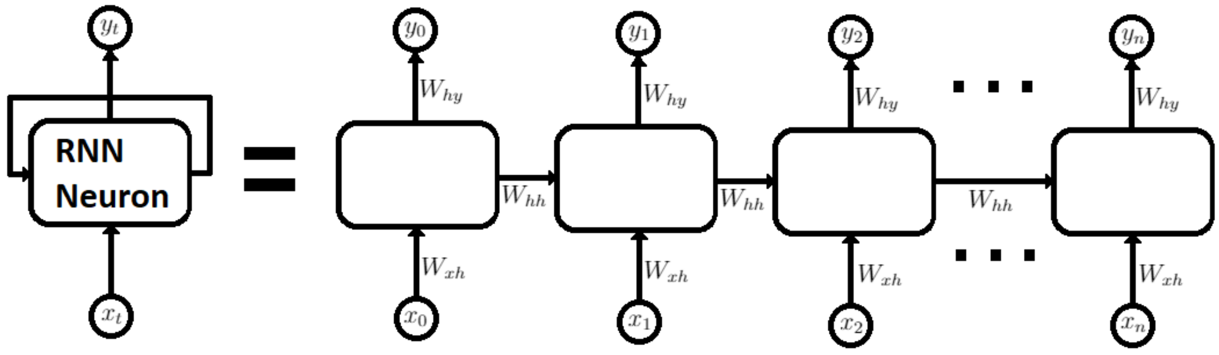

Figure 5.2: Neuron of a Recurrent Neural Network unrolled through time. The information is propagated in a feed-forward manner using the same three weight matrices at each time step. [REDO!!!]

Even though FNNs have outperformed human experts in numerous tasks[???], they still lack some fundamental capabilities of the brain. One of which is the ability to remember. To solve this problem Recurrent Neural Networks (RNNs) were introduced. The input features of RNNs are ordered data sequences of varying lengths. RNNs are most prominently capable of understanding and make up meaningful sentences. However, RNNs can also be utilized in physics applications when measurements can be organized as a sequence. For instance, by ordering final state particles of the same type according to their p_T discussed in more detail in chapter **??**.

The neurons of an RNN have at least one additional state called the *hidden state*. This state functions as a feedback loop in time connecting the data earlier in the sequences with their successors. Thus information which is important for the context is always available at all times. Figure 5.2 shows the hidden state loop unrolled in time. Each time-step is a copy of the same neuron receiving a different part of the sequence as input x_i. Every copy of the neuron is connected with its descendent by the same weight matrix w_{hh}. Consequently, the update of the hidden state vector $\vec{h}(t-1)$ to the new hidden state $\vec{h}(t)$ can be expressed as

$$\vec{h}(t) = \Phi(w_{hh}\vec{h}(t-1) + w_{xh}\vec{x}) \tag{5.6}$$

where w_{xh} is the weight matrix connecting the input vector $\vec{x}$ with the hidden state and Φ the activation function. The individual outputs y_i are connected with the current hidden state h_i via the weight matrix w_{hy} and therefore can be calculated as

$$y_i = w_{hy} \cdot h_i \tag{5.7}$$

The training of the weight matrices is performed by the Backpropagation Through Time algorithm (BTT). Similar to the standard Backpropagation algorithm, by propagating the gradient backward from the output to the input layer. Additionally, the BTT propagates the gradients also backward in time.

While the above-described RNNs can in principle retain information over a long sequence, in practice the calculated gradient quickly vanishes, making the training impossible. The solution to this problem comes with the Long Term Short Memory (LSTM) neuron architecture which introduces one more intermediate state called the *cell state*. A LSTM neuron performs four basic steps at each time step. Firstly the irrelevant parts in the current cell state are forgotten based on the new information and the old hidden state. Then, new information that is relevant for processing the sequence is stored, followed by updating the cell state based on the results obtained. Lastly, the output to the hidden state is computed based on the updated cell state and the new information.

The cell state has the advantage that it allows for an uninterrupted flow of the gradients and therefore making it possible to process long sequential data.

5.4 Optimization of Neural Networks

At the base of each optimization of Neural Networks is the "no free lunch theorem". It states that there is no one single model that works for every problem[???]. The only way to build ANNs with high performance is to try out different optimization algorithms, activation functions, weight initialization, and so forth. Hence, after introducing the performance measure of choice, the most commonly used options will be discussed in this section.

Performance measures

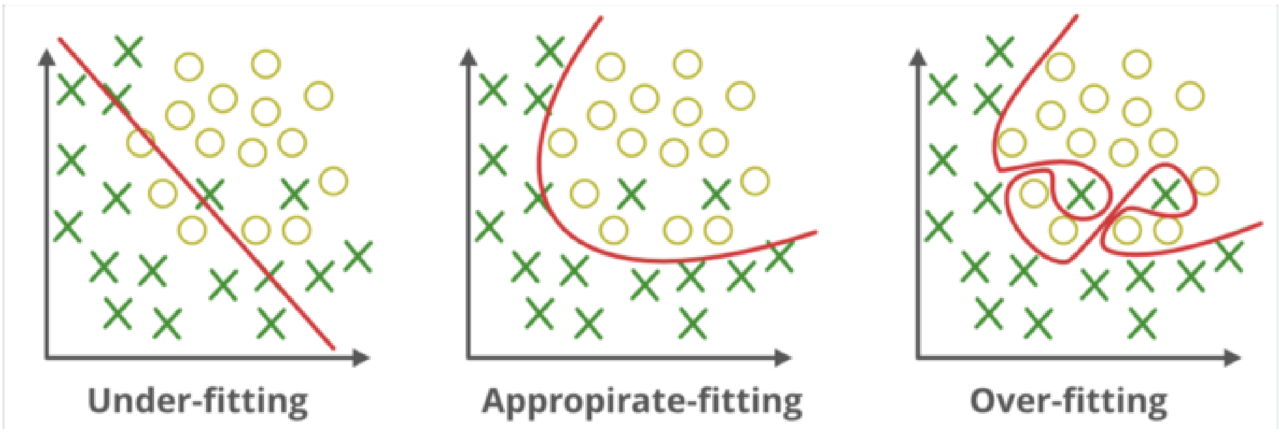

Figure 5.3: Three different models (red) to distinguish the two classes of circles and crosses. The rightmost model is too simple (underfitting) whereas the leftmost is too complex (underfitting). Only the model applied in the middle will generalize well while having good separation power. (Figure taken from [4]).

In this thesis, most of the time binary classification problems are considered where an event either belongs to the signal class or the background class. Therefore when evaluating the outcome of the classification four cases are possible

True positive (TP)	The true and the predicted class is signal
False negative (FN)	The true class is signal but the predicted class is background
False positive (FP)	The true class is background but the predicted class is signal
True negative (TN)	The true and the predicted class is background

These quantities can be summarized using the Receiver-Operator-Characteristic (ROC) curve where the signal efficiency ($x = \frac{\text{TP}}{\text{TP+FN}}$) is plotted against the background rejection ($y = 1 - \frac{\text{FP}}{\text{TN+FP}}$). The separation power of the classifier is then given by Γ the Area Under the Curve (AUC). A model that is capable of distinguishing all signal events from all background events has an AUC of 1. However, in practice, models with such high AUC are very likely to overfit the dataset used during training. Overfitting, shown in Figure 5.3, occurs when the model is too complex and not only detects subtle patterns of the data but also of the noise. One way of reducing overfitting also called overtraining is to decrease the number of parameters used in the neural network. This is achieved by simplifying its architecture i.e. the number of layers and neurons. On the other hand, a random classifier, which on average gets 50% of the event classes correctly, achieves an AUC of 0.5. A model that is too simple to parametrize the underlying structure of the data underfits the problem, as shown in Figure 5.3. In the case of Neural Networks there are various ways of avoiding underfitting such as selecting a more complex architecture or adjusting the *learning rate* of the optimization algorithm.

Optimization algorithms

Gradient Descent discussed in section ??? is just one choice of the optimization algorithm which is rarely used in practice due to its high computational cost. A very popular variation is Mini-batch Gradient Descent. In Mini-batch Gradient Descent[???], the gradient for the weight updates (equation 5.5) is computed based on a small randomly selected subset called the *batch*. To still utilize the full statistics available, weight updates are performed m times per epoch where m is given by the ratio of the number of instances in the batch divided by the number of instances in the total sample. The usage of Mini-batch Gradient Descent can decrease the computational cost significantly. Furthermore, due to the random nature of the batch selection, the training algorithm can escape local minima more easily[???].
Two problems that similarly can slow down the training process are saddle points and plateaus where the gradient and therefore the weight updates can become very small or very large. An optimization algorithm designed to increase the performance of Neural Networks in such situations is the RMSprop[???]. This algorithm introduces a constant step size where the direction of the weight update depends on the sign of the gradient. The constant step size is only changed if the previous and the current gradient have the same sign. In this case the step size v_t for the next weight updated w_t is given by the previous step size v_{t-1} multiplied with a constant factor $beta$

$$v_t = \beta v_{t-1} + (1-\beta)(\frac{\partial L}{\partial w})^2 \tag{5.8}$$

$$w_t = w_{t-1} - \frac{\alpha}{\sqrt{v_t}}\frac{\partial L}{\partial w} \tag{5.9}$$

where the name giving one over square root function (RMS) is needed to incorporate mini-batches for the gradient calculation[???].
An optimization algorithm that has been shown to work well for a broad range of applications is the Adaptive Moment Estimation (ADAM) algorithm. Like RMSprop it uses v_t term while introducing an additional term m_t called the momentum term. This term ensures a faster convergence by increasing the learning rate for those weights that had a large contribution in the previous step

$$m_t = \beta_2 m_{t-1} + (1-\beta_2)\frac{\partial L}{\partial w} \tag{5.10}$$

where β_2 is a constant scaling factor. The weight update for ADAM is than given as

$$w_t = w_{t-1} - \frac{\alpha}{\sqrt{\bar{v}_t}}\bar{m}_t \tag{5.11}$$

where $\bar{v_t} = \frac{v_t}{1-\beta}$ and $\bar{m}_t = \frac{m_t}{1-\beta_2}$ which are bias corrections needed to counteract the initialization to 0.
As always in machine learning the best choice of the hyperparameters such as learn rate, scaling factors, and batch size can only be found by testing.

Activation functions and Weight initialization

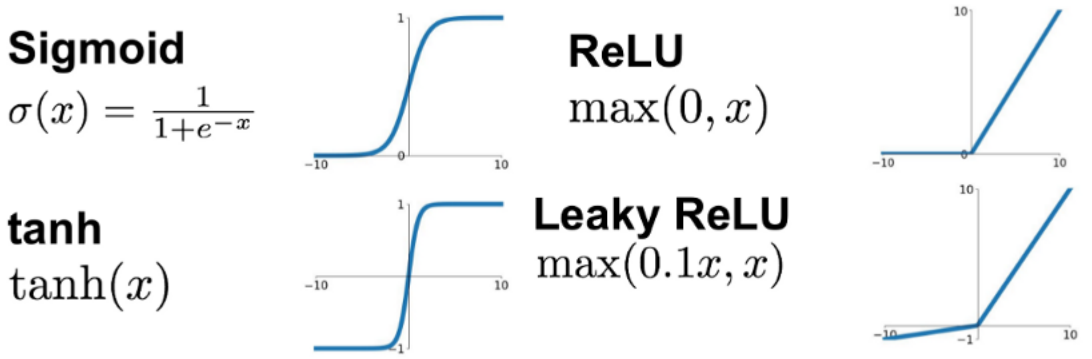

Figure 5.4: Four different activation functions. Sigmoid and Tanh are mostly used for the output layer whereas RELU and Leaky RELU are common choices for hidden layers. (Taken from [5])

Another component of a neural network that can have a big influence on its performance is the activation function. Four of the most commonly used activation functions are shown in Figure 5.4. The rectified linear (ReLU) activation function inherits the net input for positive values and evaluates to 0 otherwise. Consequently, the weight updates are easy to compute or cancel completely. This can be an advantage when it comes to computation cost and a disadvantage if the obtained performance is poor. The leaky ReLU activation function is similar to ReLU but for negative values, it returns the by α scaled net input. Thus, all weights are updated in each iteration which for some cases improves the performance. An activation function specifically designed for FNNs in the SELU activation function

$$\text{selu}(z) = \lambda \begin{cases} z & \text{if} z < 0 \\ \alpha e^{-z} - \alpha & \text{if} z \leq 0 \end{cases} \tag{5.12}$$

λ and α are two fixed parameters. According to the original paper[???], they should take the values of $\alpha = 1.6732$ and $\lambda = 1.0507$ for standard scaled inputs. FNNs with SELU activation function frequently outperform FNNs with other activation functions [???]. The sigmoid activation was introduced beforehand in section **??**. It can take values between 0 and 1 and hence is the preferred choice for the output layer where probabilities for the different classes are obtained. Moreover, it's the activation function that is closest to the initial idea of the biological neuron whose output is either 0 or 1. The biggest drawback of the sigmoid activation function is its computation related to it even in combination with the cross-entropy loss.

Closely related to the selection of the activation function is the choice of the weight initialization. For instance activation function requires the initialized weights to have zero mean and a standard deviation of 1 over the vector of the input features. The simplest choice would be to initialize all weights and biases to a constant value. However, this leads to the same partial derivative for all weights and thus only one feature could be learned. Therefore, it's not surprising that weight initialization can have a considerable influence on the performance of a neural network. The most popular methods are LeCun, Glorot and, He. These methods draw random values for the initial weights from uniform or normal distribution. They only differ in upper and lower bounds or standard deviation and mean, respectively. The biases, on the other hand, are typically initialized separately to either 0, a small constant value, or 1 in the case of LSTMs.

Regularization

A neural network can not only be improved by its performance but also by decreasing overtraining. This process is referred to as regularization. The straightforward way of decreasing the overtraining is to reduce the number of parameters. However, removing neurons or entire layers might lead to completely different training behavior. Two other regularization strategies that avoid changes in the architecture are the Ridge Regression[???] and Lasso Regression[???] also called l1 and l2 regularization. Both introduce additional terms to the loss function. In the case of Lasso Regression, this term is given by the l1 norm $\alpha_1 \sum_{i=1}^{n} |w_i|$ where α_1 is a new hyperparameter and w_i is the i^{th} weight. A useful property of this regularization is that it tends to eliminate the weights for the least important features. The term of the Ridge Regression is defined as the l2 norm $\alpha_2 \sum_{i=1}^{n} |w_i|^2$. For large α_2 all weights are close to zero, thus reducing the degrees of freedom.

Another approach to avoid overtraining is the method of Dropout[???]. At each training step, every neuron has a probability t to be ignored in the following calculations. Hence, Dropout prevents the neural network to rely too much on specific neurons.