Tutorial 03: Fully Connected Neural Nets - Forward Propagation

The goal of this tutorial is to introduce the concept of Artificial Neural Networks and to provide detailed mathematical relations for a Fully Connected Neural Network in order to ease implementations in a programming language.

1 Deep Learning and Machine Learning

As a Machine Learning approach Deep Learning has the goal of solving complex problems, like image recognition, language translation, etc. The general idea in order to achieve this is to define a specific task, which the Deep Learning algorithm is supposed to solve with the best possible performance or accuracy. In the case of Supervised Learning these kinds of tasks consist of a specific data set $\boldsymbol{X}$ and according labels $\boldsymbol{Y}$. A Deep Learning algorithm's goal it to reproduce the labels based on the data. For a fair evaluation the algorithm is trained on one subset of the data and tested on a different subset, which the algorithm has never seen before.

In order to evaluate, how well a Machine Learning algorithm performs on a supervised learning task, we compare the estimation of the algorithm $\hat{\boldsymbol{Y}}$, which are based on the data $\boldsymbol{X}$, with the labels $\boldsymbol{Y}$ using a so-called **loss function** $L(\hat{\boldsymbol{Y}}, \boldsymbol{Y})$, The loss function should provide a scalar measure, e.g. the mean squared difference between the elements in $\boldsymbol{Y}$ and $\hat{\boldsymbol{Y}}$, which can be minimised by tuning parameters of the machine learning algorithm (model). In Deep Learning, we will (almost always) use the gradient descent algorithm for minimisation. Therefore, the loss function should be differentiable with respect to the model's parameters.

2 Artificial Neural Networks

The concept of Artificial Neural Networks (ANNs or NNs for short) is what distinguishes Deep Learning from most other Machine Learning algorithms. Inspired by neurons in animal brains NNs anticipate emergent/intelligent behaviour from a large number of neurons (units) and connections between them. A visualisation of a NN is given in Figure 1.

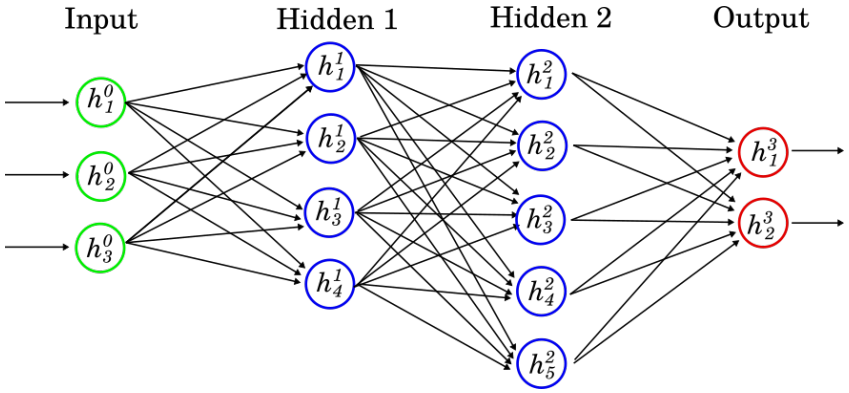

Figure 1: Visualisation of an Artificial Neural Network

2.1 Principles of Neural Networks

Each NN consists of an **input layer**, an **output** layer and at least one **hidden layer**. The input layer $\boldsymbol{i}$ has a number of neurons equal to the dimension (number of features) of one data point, e.g. if the data consists of images with 28×28 grey-scale pixels, the input layer has 784 neurons to represent the values of those pixels for a given data point. The output layer $\boldsymbol{o}$ represents our estimation of the label for a given data point, e.g., in a classification problem each neuron of the output layer represents the estimated probability of one classes. The hidden layers $\boldsymbol{h}^i$ are connecting the input layer to the output layer thereby processing the data in multiple steps.

Each neuron (except the neurons in the input layer) is connected to the neurons in the previous layer, as indicated in Figure 1. The strength of the connection between the jth neuron of the ith layer and the kth neuron of the $(i-1)$th layer is expressed by a specific **weight** w_{jk}^i . The value (activation) of the jth neuron in the ith layer h_j^i can be calculated by a weighted sum of the neurons in the previous layer, as well as an additional **bias** b_j^i, specific to this neuron, and finally a so-called **activiation function**:

$$h_j^i = \mathrm{act}^i \left(b_j^i + \sum_{k=1}^{N_{i-1}} w_{kj}^i h_n^{i-1} \right), \tag{1}$$

The activation function act^i of the ith layer is a non-linear function, which is supposed to add complexity to the model, i.e., the activation function makes it possible for neurons to have a complex non-linear dependency on previous neurons. Common examples for an activation function are the sigmoid function, the tangens hyperbolicus and the rectified linear unit.

2.2 Formalism for Neural Networks

The description of a single neuron (1) implies that the activations of neurons in each layer depend on the activations of the previous layer. In order to obtain the prediction

of the output layer, we therefore need to evaluate the neurons layer by layer, starting with the input layer. This procedure is called **forward propagation**.

In the following we will introduce a notation based on matrices to express operations in the context of neural network in a clear manner, which will especially be beneficial for neural network implementations with python. For this purpose we denote any layer of the neural network as a row vector $\boldsymbol{h}^i \in \mathbb{R}^{1\times N_i}$ with the jth entry $\boldsymbol{h}^i_j$ representing the activation of the jth neuron. The input and output layers are included in this notation ($i = 0$ for the input layer), though if we refer to them explicitly we call them $\boldsymbol{i}$ and $\boldsymbol{o}$ respectively.

The biases of layer i are also represented by a row vector $\boldsymbol{b}^i \in \mathbb{R}^{1\times N_i}$ with $\boldsymbol{b}^i_j$ being the bias of the jth neuron in the ith layer. We further introduce weight matrices $\boldsymbol{W}^i \in \mathbb{R}^{N_{i-1}\times N_i}$, whose entries $\boldsymbol{W}^i_{nj} = w^i_{nj}$, as introduced in (1), are connecting neurons from layer $(i-1)$ and layer i.

Taking into consideration the definition of a matrix multiplication we can write the activations of a whole layer as

$$\boldsymbol{h}^i = \mathrm{act}(\boldsymbol{h}^{i-1}\boldsymbol{W}^i + \boldsymbol{b}^i). \tag{2}$$

In our notation, the activation function acts element-wise on a matrix $\boldsymbol{X}$:

$$(\mathrm{act}(\boldsymbol{X}))_{ij} = \mathrm{act}(\boldsymbol{X}_{ij}). \tag{3}$$

For a specific data point ($\boldsymbol{h}^0 = \boldsymbol{x}$) the prediction $\boldsymbol{o}(\boldsymbol{x}) = \hat{\boldsymbol{y}}(\boldsymbol{x})$ of the network can therefore be expressed as

$$\hat{\boldsymbol{y}}(\boldsymbol{x}) = \mathrm{act}(\mathrm{act}(\ldots \mathrm{act}(\boldsymbol{x}\boldsymbol{W}^1 + \boldsymbol{b}^1)\ldots \boldsymbol{W}^{n-1} + \boldsymbol{b}^{n-1})\boldsymbol{W}^n + \boldsymbol{b}^n). \tag{4}$$

If we now consider the case where we have M data points with N_0 features each, e.g. M pictures with N_0 greyscale pixels, we can write the whole set of data as a matrix $\boldsymbol{X} \in \mathbb{R}^{M\times N_0}$, where $\boldsymbol{X}_{mj}$ denotes the value of the jth feature of the mth data point. Obviously our NN is now supposed to produce M predictions $\hat{\boldsymbol{y}}$, one for each data point. We can use a notation, very similar to (2) in order to describe the forward propagation in this scenario, which will again be very helpful for a NN implemenation with python:

$$\boldsymbol{H}^i = \mathrm{act}(\boldsymbol{H}^{i-1}\boldsymbol{W}^i + \boldsymbol{B}^i). \tag{5}$$

The weight matrices stay exactly the same as in (2), the hidden layers are now of the form $\boldsymbol{H}^i \in \mathbb{R}^{M\times N_i}$ with $\boldsymbol{H}^0 = \boldsymbol{X}$ and the bias is **broadcasted** to become a matrix, meaning the row vector $\boldsymbol{b}^i$ is „copied“ M times to build up $\boldsymbol{B}^i$. More formally, we write $\boldsymbol{B}^i \in \mathbb{R}^{M\times N_i}$, with

$$\boldsymbol{B}^i_{mj} = \boldsymbol{b}^i_j \quad \forall m. \tag{6}$$

Note that the matrix multiplication and activation function still work the same way as in (2).

3 Classification Problems with Neural Networks

From the variety of problems NNs can be used for, the most common ones are regression problems, for which we try to predict one single continuous value for every data point, and classification problems, for which we try to predict the correct discrete class for every data point. In the case of a classification problem each output neuron of a NN represents one of the classes. For a concrete prediction, we usually choose the class according to the neuron with the highest activation.

However, in order to better interpret results from the activation function and in order to be able to use a differentiable probability-based loss function (namely the **cross-entropy**) it is common to use the **softmax function** as the activation function of the output layer. The softmax function is a special kind of activation functions that not only depends on one single neuron of the output layer, but on all output neurons, as it normalises the activations of the output layer to probability-interpratable values. Using our previous notation the softmax function can defined for a matrix $\boldsymbol{X} \in \mathbb{R}^{M \times N_i}$ as

$$(\text{softmax}(\boldsymbol{X}))_{mj} = \frac{\exp(\boldsymbol{X}_{mj})}{\sum_{k=1}^{N_i} \exp(\boldsymbol{X}_{mk})}, \tag{7}$$

i.e., it uses the exponential function to normalise every row of $\boldsymbol{X}$. It is easy to see that two important properties of probabilities are fullfilled in this scenario; the first one being that all values should be ≥ 0, given by the positive value set of the exponential function, and the second one being that the sum over all values should be equal to 1 (for each row/data point), given that

$$\sum_{j=1}^{N_i} \frac{\exp(\boldsymbol{X}_{mj})}{\sum_{n=1}^{N_i} \exp(\boldsymbol{X}_{mn})} = 1. \tag{8}$$