Lab Course SS2024:
Data Science in Astrophysics
Session 8: Neural Networks-I

Sudeshna Boro Saikia*

May 1, 2024

*sudeshna.boro.saikia@univie.ac.at

1 Introduction

Artifical neural networks (referred to as neural networks from hereon) are a form of supervised machine learning algorithm. Neural networks approximate some function and learn the mapping between an input $\mathbf{x}$ and an output $\mathbf{y}$. The learning is carried out by introducing training data sets (input vector and output labels) to the network. The network applies this learnt mapping on new test data sets. The fundamental building block of a neural network is a **neuron** and as the name suggests, it can be loosely "compared" to biological neurons. In reality a neuron is a function and a neural network consists of neurons arranged in layers. In its simplest form a neural network consists of an input layer of neurons, an output layer of neurons, and a hidden layer of neurons [1]. This layered architecture is known as a feed-forward network, shown in Figure 1. The number of hidden layers also dictates the depth of the model, giving rise to the terminology deep learning.

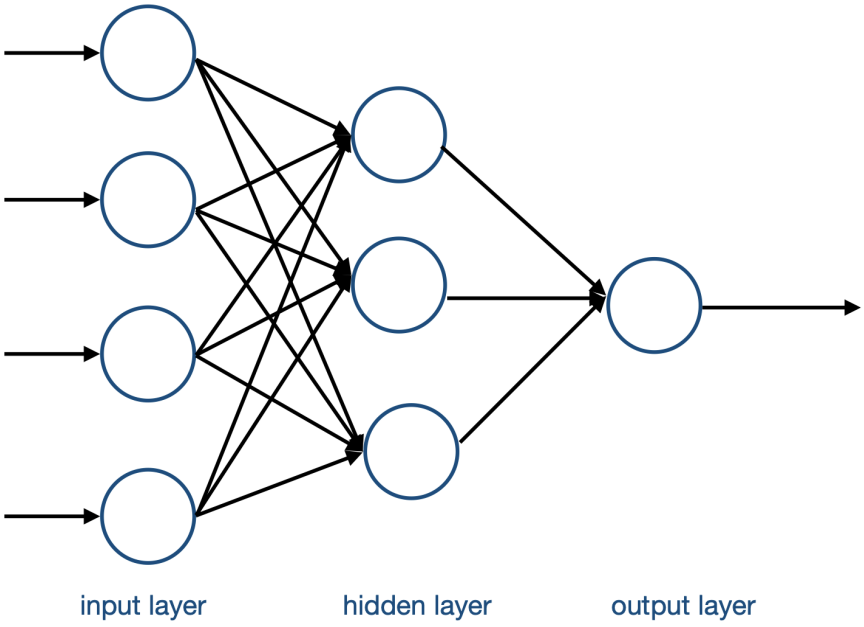

Figure 1: Schematic of a neural network.

[1] simplest when it comes to multi-layered networks. A single-layered network or a perceptron is generally considered as the simplest form of neural network

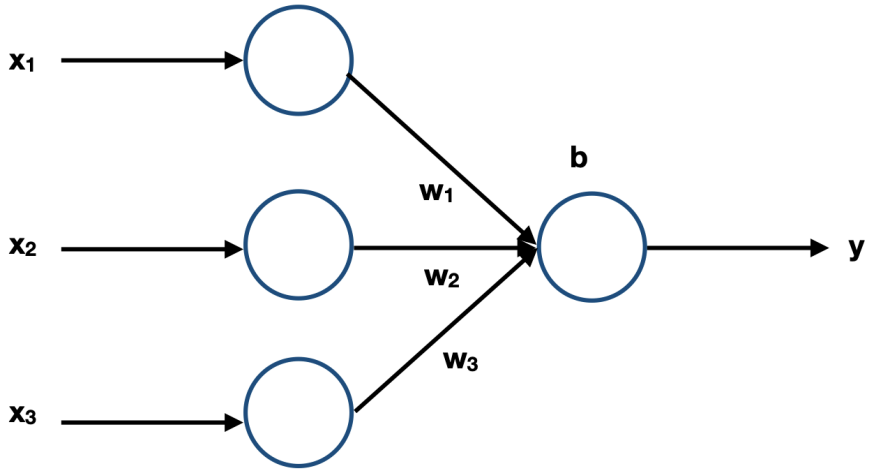

Figure 2: A schematic diagram of a perceptron with input $\mathbf{x}$ and output y, and weights $\mathbf{w}$ and a bias b.

2 Fundamentals of a neural network

2.1 Perceptron

The earliest form of artifical neuron was first introduced in the 1950s and 1960s and is known as a perceptron. The perceptron takes binary inputs and returns a binary output. In this setup inputs are directly mapped to an output by a linear function. The basic architecture of a perceptron is shown in Figure 2 where a input layer is followed by a single output perceptron. Let's take an example training data set $\mathcal{D}$=($\mathbf{x}$,$\mathbf{y}$), where $\mathbf{x}$ is the input vector and $\mathbf{y}$ is the label or the output. In Figure 2 x has three inputs or features x_1, x_2, x_3. Both input and output are binary in this case (0 or 1). To compute the output $\mathbf{y}$ (0 or 1) **weights** are introduced, where $\mathbf{w}$ expresses the importance of each feature. The final output is then dependent on whether the weighted sum is less than or greater than a **threshold** value.

$$\text{output} = \begin{cases} 0 & \text{if } \sum_i w_i x_i \leq \text{threshold} \\ 1 & \text{if } \sum_i w_i x_i > \text{threshold} \end{cases} \tag{1}$$

The threshold is related to the more widely known term **bias**, where bias $\equiv -$ threshold. The bias determines how easy it is for the perceptron to return a 1. Re-writing equation 1 gives us

$$\text{output} = \begin{cases} 0 & \text{if } \mathbf{w}\ \mathbf{x} + \mathbf{b} \leq 0 \\ 1 & \text{if } \mathbf{w}\ \mathbf{x} + \mathbf{b} > 0 \end{cases} \tag{2}$$

This shows that the final output depends not only on the input but also on the weights and the bias. By tuning the weights and biases we can turn a perceptron or a network

of perceptrons into a machine learning algorithm.

During the training phase, a neural network consisting of perceptrons (or any other neuron) changes the weights and biases to get changes in the output. To be more specific ability to obtain small changes in the output due to small changes made in the weights and biases is what makes learning possible. Unfortunately, a neural network made up of perceptrons is very bad at performing this task. Since perceptrons are binary functions their output is either 0 or 1. A small change in one of the parameters (weights and biases) could cause the perceptron to flip from 0 to 1. This behaviour is very undesirable when a network involves multiple perceptrons and could lead to untraceable complications. A solution to this problem is the sigmoid neuron.

2.2 Sigmoid neuron

Sigmoid neurons are modified perceptrons, where the output is no longer 0 and 1 but instead it is $\sigma(\mathbf{wx} + \mathbf{b})$, where σ is the sigmoid function (also referred to as logistic function).

$$\sigma(\mathbf{z}) \equiv \frac{1}{1 + e^{-\mathbf{z}}} \tag{3}$$

In equation 3, $\mathbf{z} \equiv \mathbf{wx} + \mathbf{b}$. The behaviour of a sigmoid neuron is similar to a perceptron for very large and small values of $\mathbf{z}$. If $\mathbf{z}$ is very large and positive then $\sigma(z) \approx 1$ (equation 3), and for large negative values of $\mathbf{z}$ the output approaches 0 (see Figure 3). For intermediate values the sigmoid neuron deviates from the perceptron.

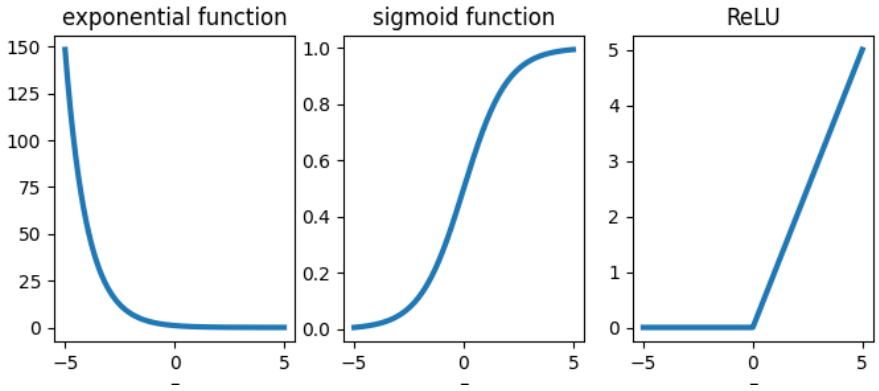

Figure 3: *Left to right*:Exponential,sigmoid and ReLU functions.

As shown in Figure 3 (right) the sigmoid function is a smoothed out version of a step function, which indeed approaches 0 and 1 at the edges but at intermediate values it deviates from the step function. This smoothness allows for small changes in the output for small changes in the parameters, and the changes in the output is a linear function of the changes in the parameters. The sigmoid and the step functions are also known as

an **activation** function. There are a wide range of activation functions available today and one such widely used activation function is the ReLU (right panel, Figure 3).

Figure 3 can be easily visualised using PYTHON'S NUMPY and MATPLOTLIB, as shown by the code snippet below.

```
import numpy as np
import matplotib.pyplot as plt

def exp(x):
    #exponential function
    return np.exp(-x)

def sigmoid(x):
    #sigmoid function
    return 1/(1+np.exp(-x))

def relu(x):
    #ReLU
    return np.maximum(0,x)

xarray = np.linspace(-5,5,100)

fig,ax = plt.subplots(1,2, figsize=(8,6))
ax[0].plot(xarray,exp(xarray),lw=3)
ax[1].plot(xarray,sigmoid(xarray),lw=3)
```

3 The feed-forward network and backpropagation

In previous sections we learnt about the basic architecture of a neural network and how a single neuron makes a decision. How do all of these help a network to perform a given task, whether it is a classification using the famous handwritten digits of the MNIST dataset [2] or exoplanet detection or atmospheric characterization? Or in other words, how does a network decide which parameters (weights and biases) to use and tweak for predictions on a test data set?

If we take the same network architecture as Figure 1 then we have three input neurons at the input layer (x_1, x_2, x_3, x_4) and three neurons at the hidden layer, and one neuron at the output layer. The network training or learning consists of two phases, the forward and the backward phase. During the forward phase (for the training dataset $\mathcal{D} = (\mathbf{x}, \mathbf{y})$) the network output for a given input is obtained by assigning random weights and biases. The goal is to find values of weights and biases so that the output approximates the true value of $\mathbf{y}$ for the input $\mathbf{x}$. This is achieved in the backward phase and is quantified by the introduction of a **cost function**. A commonly used cost function is the mean squared error (MSE) function ,

$$C(\mathbf{w}, \mathbf{b}) = \frac{1}{n} \sum_n ||\mathbf{y} - \mathbf{a}||^2 \tag{4}$$

[2] http://yann.lecun.com/exdb/mnist/

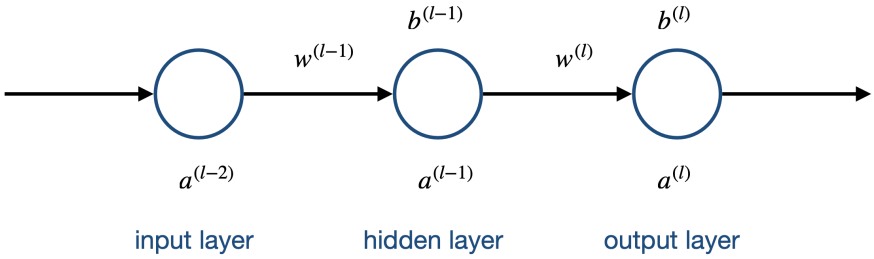

Figure 4: A network consisting of single neurons in each of the three layers.

where $C(\mathbf{w}, \mathbf{b})$ is the cost function, $\mathbf{w}$ and $\mathbf{b}$ are the weights and biases associated with the entire network, $\mathbf{y}$ is the training data set label, $\mathbf{a}$ is the output of the neural network, and n is the number of examples in the training data set. The MSE cost function is always positive, and approaches 0 when the model can well approximate the labels for all training inputs. The neural network learns by tweaking the weights and biases to minimize this cost function. *How does the network tweak the weights and biases? Which direction to go?* The negative gradient of the cost function tells us how to change the weights and biases, and backpropagation is the algorithm that computes this gradient.

In order to understand how a feed-forward network learns let's use a simple architecture, as shown in Figure 4, where the input, hidden, and output layers consist of only one neuron each.

Forward calculation: The input x_1 is fed into the hidden layer neuron, which is represented as $a^{(l-2)}$ in Figure 4. The activation[3] of the hidden layer is $a^{(l-1)}$ which is the input for the output layer. The final activation is a^l, which is also the output of the entire network. The subscripts represent the three layers in our network. The activations for the output and the hidden neuron is determined as follows.

$$a^{(l)} = \sigma(w^{(l)} a^{(l-1)} + b^{(l)}) \tag{5}$$

$$a^{(l-1)} = \sigma(w^{(l-1)} a^{(l-2)} + b^{(l-1)}) \tag{6}$$

This shows that output of the network is determined by its weight, bias, and the hidden layer activation. The hidden layer activation is itself triggered by its weight, bias, and the input layer activation which is also the input vector. In this simple example the number of parameters is 4; two weights and two biases.

Backpropagation: Once we have a model output ($a^{(l)}$) the goal is to minimize the difference between $a^{(l)}$ and y using the cost function. As shown in equation 4 the cost is a function of all the weights and biases associated with the network (4 in this example but could be hundred of thousands in a real network). Figure 5 shows examples of two

[3] the output of a neuron is also called an activation.

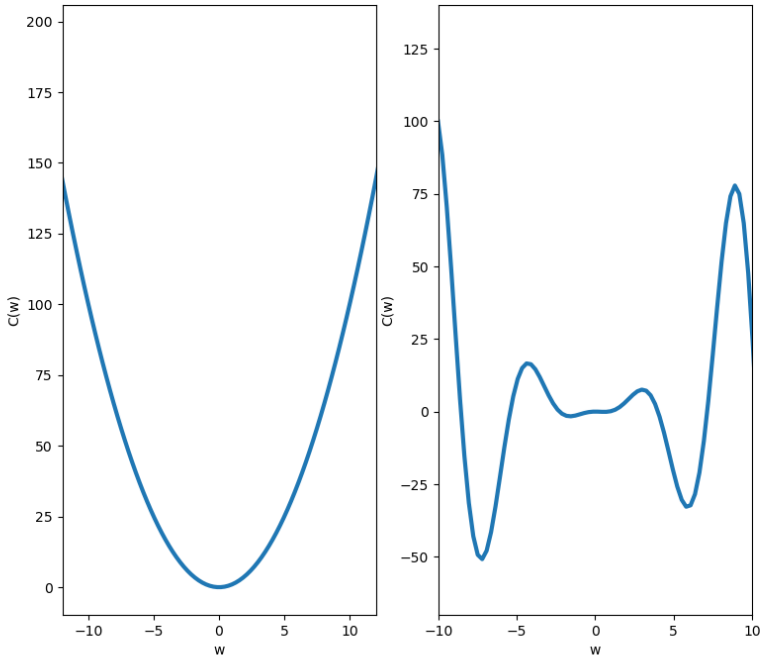

Figure 5: Two different hypothetical cost functions.

hypothetical cost functions with single input and output. The function on the left has one global minima and the one on the right has multiple local minima.

Finding the minima in both cases using our naked eye is very easy, but how can we obtain the minimum numerically? To do that we can take advantage of the negative gradient. The negative gradient of a function gives the direction of the steepest descent, i.e., which direction to move to reach the minima. A neural network calculates the negative gradient of the cost function (the downhill direction), steps in the downward direction, and repeats the process until the global minima is reached. Figure 5 shows that the step is also important so that the network is not stuck at a local minima, and this step is controlled using a hyper-parameter known as a **learning rate**.

Backpropagation is the algorithm that computes the gradient of the cost function. Going back to the network in Figure 4, the cost function for a single example would be $C_0(z) = (y - a^{(l)})^2$. Equations 5 and 8 tell us that $a^{(l)}$ itself depends on the weights and biases of the network and on the input layer. The network can't (and shouldn't) change the input but can change the weights and biases.

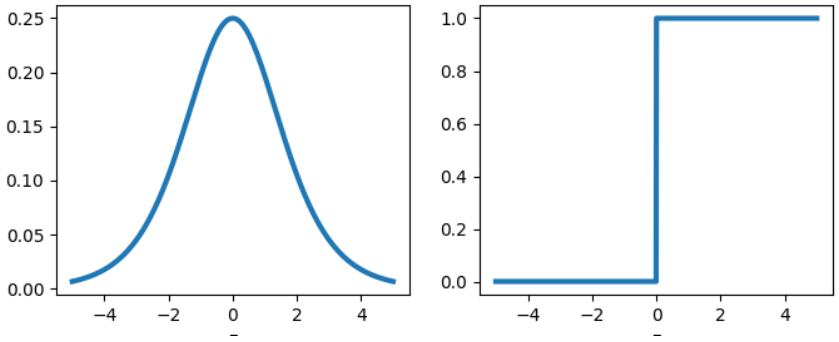

Figure 6: Derivatives of sigmoid and ReLU

The derivative of the cost function w.r.t weight $w^{(l)}$ can be written as,

$$\frac{\partial C_0}{\partial w^{(l)}} = \frac{\partial z^{(l)}}{\partial w^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial C_0}{\partial a^{(l)}} \tag{7}$$

where $z^{(l)} = w^{(l)} a^{(l-1)} + b^{(l)}$. The right hand side of the above equation represents the sensitivity of the cost function to small changes in the weight associated with the output neuron. Computing the derivatives in the above equation gives us,

$$\frac{\partial C_0}{\partial w^{(l)}} = 2a^{(l-1)} \sigma'(z^{(l)})(y - a^{(l)}) \tag{8}$$

Similarly the derivative of the cost function w.r.t bias $b^{(l)}$ is

$$\frac{\partial C_0}{\partial b^{(l)}} = 2\sigma'(z^{(l)})(y - a^{(l)}) \tag{9}$$

This shows that the cost of the network is determined in the backward direction, starting with the last layer first, hence the term backpropagation. Figure 6 shows the derivative of the sigmoid and the ReLU activation functions.

If we now move to a more complex network (e.g., Figure 1) then the terminology changes to include the multi-neuron layers. We introduce the subscripts j and k to index the output and hidden layers respectively. As an example the output layer is indexed as $a_j^{(l)}$ and the hidden layer is indexed as $a_k^{(l)}$. The weights are also written differently as there are more weights connected to each layer. The weights associated with the output layer are referred to as $w_{jk}^{(l)}$. This also influences the output layer activation,

$$a_j^{(l)} = \sigma\left(\sum_k w_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)}\right) \tag{10}$$

The cost function can be re-written as,

$$C_0 = \sum_j (y_j - a_j^{(l)})^2 \tag{11}$$

We can now do the same exercise as we did for our single neuron network and determine the gradient of the cost function. *To do at home: do equations 8 and 9 change? What does it mean for the cost function?*

Until now we discussed derivatives for a single training example. A neural network averages these derivatives over all training examples. Furthermore, in a multi-layered network the derivatives w.r.t all the weights and biases in a network makes up the gradient of the cost function. Hence backpropagation is often paired with a gradient descent algorithm to minimize the cost function.

4 Neural network classifiers: exoplanet detection

In our lab course we often came across machine learning examples for regression. In this lecture we will introduce a machine learning classifier. To be specific we will discuss ways to classify exoplanet transit lightcurves from the Kepler satellite using TENSORFLOW AND KERAS (follow the instructions on the tensorflow website for an installation).

In the last decade transit observations have been one of the most successful way of detecting exoplanets. This exoplanet detection method takes advantage of the fact that,there is a dimming in the light coming from the system when an exoplanet transits its host star. Due to the periodic nature of a planetary transit this dimming is also periodic in nature, allowing us to detect thousands of planets in this fashion.

The goal of this example is to apply a neural network to detect exoplanets. In this example we will use data from the Kepler satellite to train a simple 1D neural network that classifies the data into two classes: exoplanet host or no exoplanets.

```
21 #import pandas, tensorflow, matplotlib and any other packages
22 import pandas as pd
23 import matplotlib.pyplot as plt
24 import tensorflow as tf
25
26 #load the data
27 train_data = ..."exoTrain.csv"
28 test_data = ..."exoTest.csv"
29
30 #split the data into train and test set
31 # the label column is the output layer of the NN
32 # the rest of the columns mark the features of the input data, flux in this case
33
34 train_input = ..
35 test_input = ..
36
```

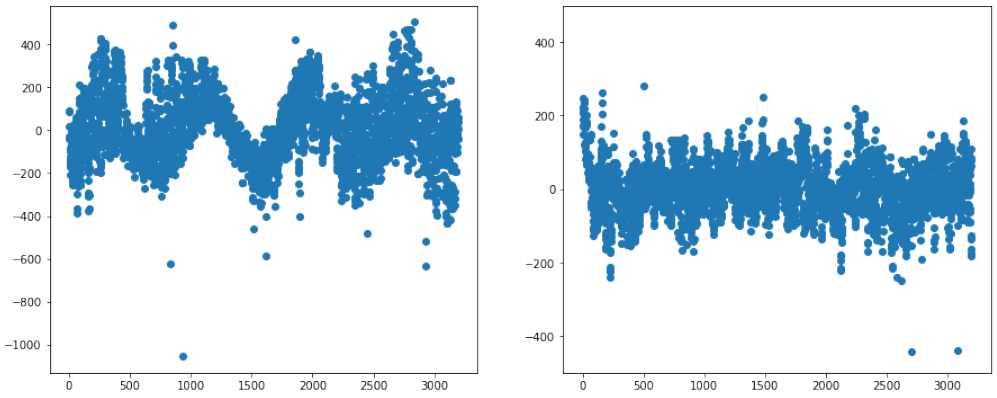

Figure 7: Lightcurve of two classes of stars, with and without exoplanets. Can you tell which one is the exoplanet host here?

```
train_output = ..
test_output = ..
```

Once we have our data sorted into appropriate arrays, dataframes or tensors, we can move on to the next stage, which is building the model. But let's have a look at the data first. Figure 7 shows an example of the two classes of data: with and without exoplanets.

To build the neural network model we need to decide the number of layers and the number of neurons for each layer. The input layer neurons are determined by the size of the input data. Since we are considering a binary classifier I suggest using two neurons for the output layers. The activation function is kept to the default. There is no strict rule on the number of layers and neurons.

```
#building the model
#use one or two hidden layers
#Fill the dots. Instead of the sigmoid function we will use a ReLU here.
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(.....),
    tf.keras.layers.Dense(.., activation='relu'),
    tf.keras.layers.Dense(..)
])
#print the model summary
```

Once the model is built we need to compile the model and perform backpropagation.

```
#the optimizer, loss function (cost function) is initialized
model.compile(optimizer="...",
              loss="...",
              metrics=['accuracy']))
```

```
#train the model
model.fit(input_data,output_data, validation_split = 0.3, epochs=10,
            shuffle=True, batch_size = 10)
```

In the model fitting phase we come ac cross a few terms that we haven't encountered until now, validation split, epochs, shuffle, batchsize. As the name suggests shuffle shuffles the dataset, but what are the rest for?

To make classification of exoplanets our neural network model passes the entire training dataset (input and output data) through the model and calculates the loss using a loss function. The network then applies an optimization algorithm to reduce the loss. This whole process of feeding the entire dataset and optimizing is known as one epoch. Often times we do this whole process multiple times. Why?

What is happening in the backpropagation phase? We are using an optimizing algorithm to update the parameters (weights and biases). Optimizing algorithms such as gradient descent are "walking" through the parameter space to find the global minima. The more epochs we have the more the parameters get updated. Of course there is a chance of over fitting if the number of epochs is too high.

To perform optimization on a large training dataset even for one epoch is a computationally expensive task. Hence the epoch is split into smaller batches. The size of the batches is also a hyper-parameter of the model.

Finally, the training dataset can be further split into training and validation dataset to validate our model. The loss and accuracy are determined for both sets. This is used to check the model performance during training.

After tweaking our hyperparameters and the model training we are ready for predictions.

```
#evaluation of test data
test_loss, test_acc = model.evaluate(test_input,  test_output, verbose=2)
```

The above model has an accuracy of more than 90%. However, despite that our model is not good at identifying exoplanet host stars. Why is that? and how can we do better? More on that in the next lecture.

References

[1] Ian Goodfellow and Yoshua Bengio and Aaron Courville (2016) *Deep learning*, MIT Press.

[2] Charu C Aggarwal (2018) *Neural Networks and Deep learning*, Springer.

[3] Michael Nielsen's online book http://neuralnetworksanddeeplearning.com/index.html