TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology

Serkan Ongan 156395IASM

DIGIT RECOGNITION USING NEURAL NETWORK IMPLEMENTED IN PYTHON

ISS0031 Modelling and Identification

Supervisor: Aleksei Tepljakov

Tallinn 2016

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Aim of the project

The aim of the project is to get a better understanding of neural networks and to get familiar with Python programming language and its libraries as tools for neural networks, as well as their implementations. Therefore it includes both theoretical and practical information in those areas.

For the theoretical part, in addition to information about neural network, I'll also look into Python libraries and their features. And for the practical part, more information can be found in the following section.

1.2 The problem and the approach

The practical problem to be solved for the project is *digit recognition using neural network implemented in Python programming language*.

For training and test, I have used an already modified and prepared dataset (MNIST) that comes with statistics about certain machine learning tasks and their performance on the data, which I hope will offer a good benchmark for comparing performance.

For example, single layer neural network is shown to have an %8.4 error rate on test data, and a 2-layer network with total 300 hidden nodes using mean square error achieves %4.7 error. For my project I aim to reach a %5-10 error.

For Python libraries (which are discussed in Chapter 3), I plan to use a library that will offer a good balance between ease of use and "low level" implementation. I have noticed that an opinionated library like TensorFlow requires also a considerable amount of time to get familiar with the library itself, which would make it harder to stick to my goals for the project.

Chapter 2

Neural Networks

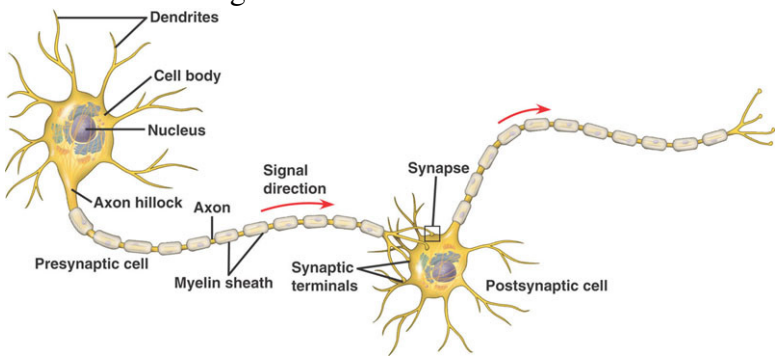
In this chapter, I take a brief look at neural networks in order to lay the foundations of the actual Python implementation of the project.

2.1 Biological model

Taking a look at the biological model that inspired artificial neural networks can help us understand their structure better. But since the biological model of how neuron and brain work is highly complex, I think it's beneficial to make it a very brief look.

A typical neuron can be thought as a signal processing device. [15] This happens through positively or negatively charged ions. Each neuron is connected to many others via synapses. Inputs to the neuron is summed up and if it's over a certain threshold the neuron is excited. In general, an neuron can be excited or inhibited.

Figure 2.1: Neuron structure.



This basic model is very close to how a TLU (threshold logic unit) works. A neural network is formed of simplified version of biological neurons and their connections, and are capable of making approximations for given data, after necessary training.

2.2 Basic theory

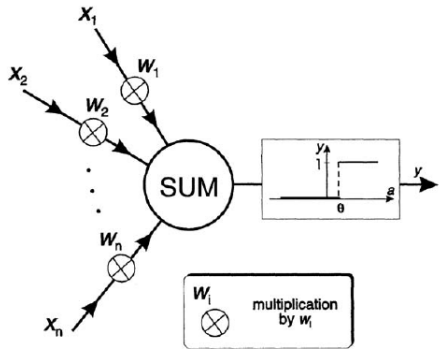
An artificial neural network is an assembly of interconnected simple processing elements (nodes), whose functionality is stored in the interconnections' strengths (weights). [15]

Smallest element of a neural network is a node. Taking a look at TLU, a basic node structure, could make understanding how neural networks work easier.

2.2.1 Threshold logic unit

A TLU unit can be explained as below.

Figure 2.2: A TLU



In the above example, the sum of weighted inputs is called **activation.** An **activation function** (a step function in this case) takes this activation and the node's bias (θ) and produces an output. So the output of a node depends on the weights, inputs, bias and the activation function type.

2.2.2 Activation functions

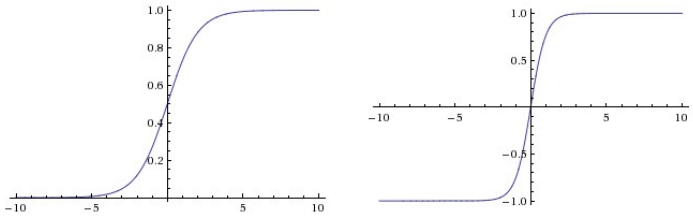
Activation functions affect the way node produces outputs, meaning they also affect the way information flows through the layers of the neural network. So we need to choose activation functions according to our goal and network structure.

For example, a sigmoid function would be a better choice if derivations are used. Two most used activation functions are hyperbolic tangent (eq. 2.1) and sigmoid functions (eq. 2.2). [16]

$$f(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{2.1}$$

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

Figure 2.3: Sigmoid (left) and hyperbolic tangent functions.



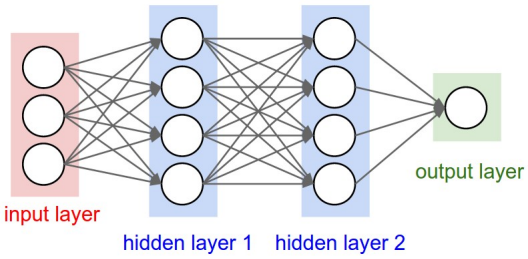
Another important activation function is called **Softmax** which is used in the output layer to give output as a probability. More precisely it produces a K sized vector of real values in the range of (0, 1) that add up to 1 when summed.

This is helpful in tasks like digit recognition where output layer size is bigger than 1.

2.2.3 Layers

After nodes, the next structure in a network is layers. A neural network has two special layers. Input and output layers, which depend on the data structure. The actual inner structure of the neural network depends on hidden layers between input and output layers, which effect the number of weights and therefore directly the training time of the network.

Figure 2.4: A neural network with 2 hidden layers.



2.3 Training

A neural network gives us a model derived from collected input and output data. But to achieve this, we need to **train** the neural network. For neural networks, training means updating weights and biases according to the error in the each iteration of the training.

Training dataset includes an input and an output for each example. During training, the training input is fed to the neural network and its output is compared to the training data output. With this comparison we acquire an error, and the *objective of training is minimizing this error.*

So in its basics, training is the optimization of global error for the neural network.

2.3.1 Cost function

Cost function can be thought as the objective function in optimization problems. It is a representation of the global error in the network and a measurement of how well a neural network fits to the entire training dataset. It also depends on weights and biases.

$$C = F(w, \theta), \quad C \to min \tag{2.3}$$

There are several differents methods for cost function. But some basic cost functions are: *sum of square errors (eq. 2.6), mean square error (eq. 2.5), root mean square (eq. 2.7).*

Although it should be noted that MSE is one of the most common cost functions used in neural networks. But there are exceptions, like the Levenberg Marquardt Algorithm requiring SSE. [16]

A **linear error** is simply the difference of ideal output (i) and actual output (a).

$$E = (i - a) \tag{2.4}$$

$$MSE = \frac{1}{n}\sum_{i=1}^{n} E^2 \tag{2.5}$$

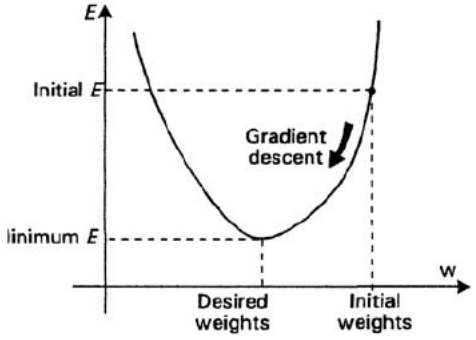
$$SSE = \frac{1}{2}\sum_{p} E^2 \tag{2.6}$$

$$RMS = \sqrt{\frac{1}{n}\sum_{i=1}^{n} E^2} \tag{2.7}$$

2.3.2 Optimization

The aim of the training is to minimize the cost function. For this, optimization methods are used. One way to optimize the cost function is to use gradient descent.

Figure 2.5: Gradient descent on error function.



Gradient descent is a means of taking a step in the direction that does the most to decrease our cost function. We use derivatives to decide which way to take a small step in order to make the cost function smaller.

For example, for C that $C(v_1, v_2)$, making small changes in v_1 and v_2 would cause a small changes in C.

$$\triangle C \equiv \frac{\delta C}{\delta v_1}\triangle v_1 + \frac{\delta C}{\delta v_2} v_2 \tag{2.8}$$

$$\triangle v = \begin{bmatrix} \triangle v_1 \\ \triangle v_2 \end{bmatrix} \textit{ and } \nabla C = \begin{bmatrix} \frac{\delta C}{\delta v_1} \\ \frac{\delta C}{\delta v_2} \end{bmatrix} \tag{2.9}$$

The amount of step we need to take to make C smaller can be calculated as the following formula, where η is the learning rate and $\eta > 0$.

$$\triangle v = -\eta \nabla C \tag{2.10}$$

But our initial starting point can make the optimization stop in a local minimum, and for that reason it's sometimes a good idea to train the network with different starting points.

- Some optimization methods:

1. *Backpropogation method:* A very common gradient descent based method. It requires a desired output for each input value, therefore it's more of a supervised learning tool. Gradient descent based methods often find the best solution within their starting point, so achieving a global result (finding the global minimum) is often has randomness to it.

2. *Genetic algorithm:* It is a global optimization method. Since it searches in many directions, the probability of finding a global minimum increases. And it makes it a very efficient tool for difficult non-linear functions.

3. *Simulated annealing:* SA is based on physical annealing process where physical substances go from a higher state of energy to a smaller one. A concrete example is heating metal to molten state and letting it slowly cooldown until it becomes solid again. An in application to neural networks, one distinct difference is that SA depends on user defined parameters in contrast to GA's dynamically assigned ones. Also it permits increasing "energy state", meaning accepting points on cost function with higher error values. This helps escaping local minimimums. [17]

Chapter 3

Python

This chapter includes information about certain Python libraries that offer tools to implement neural networks. It concludes with a section about the comparison of using computer vision solution instead of neural networks.

3.1 Language and libraries

Python is a high level scripting language that supports several programming paradigms, including procedural and object-oriented, as well as a partial support for functional programming. This makes it a very flexible tool to use in different application areas.

Another advantage of Python is that it has a clear and almost pseudo-code like syntax, making it easier to read and code. Below can be seen some examples of the syntax.

```
def function_name(args):
  """ Function definition."""
  return value

# List comprehensions: a fast and convenient way to create lists.
[x for x in range(1,10)]     #[1, 2, 3, 4, 5, 6, 7, 8, 9]
[x*x for x in range(1,10)]   # [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Listing 3.1: Python programming language.

Being popular and open-sourced, the language has plenty of libraries (roughly 90000 libraries [2]) and it is a widely used tool in scientific programming area.

For implementation, I have looked into following libraries.

NumPy

NumPy is one of the fundamental scientific packages in Python. [3] It offers powerful N-dimensional objects and tools for integrating with C/C++ and Fortran code, which make

it possible to solve performance issues using lower level languages.

```
import numpy as np

vector_4 = np.random.randn(4)
# array([ 0.00695554, -0.28457662,  2.36434078, -0.05432569])

# reshaping into a 2x2 matrix.
vector_2x2 = vector_4.reshape(2,2)
# array([[ 0.00695554, -0.28457662],
#        [ 2.36434078, -0.05432569]])
```

Listing 3.2: Very brief look into NumPy

When compared to **Matlab**, there are fundamental differences.

1. Basic type in NumPy are multidimensional arrays, and operations on these arrays are element-wise. If matrix-like operations are needed, a sub-type called **matrix** type should be used.

2. Python arrays are 0-indexed, so the first element of an array A can be reached by A[0].

3. Better support for GUI applications via Python's general purpose programming approach.

4. In contrast to Matlab, arrays have pass-by-reference semantic.

It should also be noted that official advice of the library is to use **arrays** instead of the **matrix** sub-type. While matrices behave more-or-less like Matlab matrices, they have disadvantages like having maximum 2 dimensions. [4]

Matplotlib

Matplotlib is an open-source 2-D plotting library that is widely used by Python applications.

TensorFlow

TensorFlow is an open-source library developed within Google's Machine Intelligence research organization, as a tool to make research in machine learning quicker and easier to transition from prototype to production. [6]

TensorFlow is a system where you represent computations as graphs. Each operation (node) on the flow takes **tensors**. (In TensorFlow, a tensor is mainly a multi-dimensional array with static type and dynamic dimensions.) And each operation produces tensors.

- Some features:

1. Backed by Google.
2. Good documentation, but complex.
3. Multi GPU support.
4. Distributed training.
5. Model checkpoints allows for pausing the training, evaluating and continuing from the checkpoint.

PyBrain

PyBrain is short for *Python-Based Reinforcement Learning, Artificial Intelligence and Neural Network Library*, and it's a modular tool that offers easy to use algoritms for machine learning operations. They aim to be a library that can be used both entry-level students and researchers, so providing a less dense alternative to libraries like TensorFlow.

- Features:

1. Support for standard and advanced algorithms in many areas of machine intelligence. (Supervised and unsupervised learning, reinforcement learning, optimizations, neural networks, plotting etc.)
2. Allows complex architectures.

Theano

Theano is developed in University of Montreal's Institute of Learning Algorithms. It's used in the university's research as well as its machine learning classes. In general, it's a library used to define, optimize and evaluate expression related to multi-dimensional arrays. [9]

- Features:

1. NumPy integration
2. Faster operations with GPU usage.
3. Efficient derivations
4. Extensive testing suite.

If offers speeds closer to compiled C code implementations, and it comes with several optimizations for symbolic expressions. ($x*y/y->x$)

ScikitLearn

Scikit-Learn is a machine learning library built upon NumPy, SciPy and matplotlib libraries. It can handle both supervised and unsupervised learning, and supports following tasks: *Classification, regression, clustering, dimensionality reduction, model selection, data preprocessing.*

Its advantage lies in the fact that it has extensive documentation and a lot of contribution from experts in machine learning field. It also provides a consisten interface to most machine learning tasks in a single library. [11]

Keras

Keras is a library dedicated to neural network tasks. It can run on TensorFlow or Theano libraries, therefore providing their advantages like speed, GPU usage and so on. Its focus is on being a library that enables fast experimentations. [12]

- Features:

1. Supports both convolutional and recurrent neural networks, as well as combination of both.
2. GPU support.
3. Fast prototyping.
4. Minimalist.

Libraries with Python bindings

In addition to the above libraries, there are libraries that were written in other languages but offers Python bindings: *FANN: Fast Artificial Neural Network Library, Caffe, mxnet.*

3.2 Computer vision alternative (OpenCV)

Another approach to digit recognition can be done using a computer vision library like OpenCV. OpenCV is a cross-platform computer vision library that offers interface for Java, C++, Python and C. It's possible to use classifiers like k-nearest neighbor, support vector machine and random forest.

MNIST database offers a list of machine learning tools and their performances on the MNIST datasets. This can give us an idea about the comparison between neural network and some methods used in computer vision.

Table 3.1: Method comparison.

Methods	**Test Error Range**
Neural Networks	% 0.35 - 4.7
Convolutional Networks	% 0.23 - 1.7
K-Nearest Neighbor	% 0.52 - 5.0
Support Vector Machines	% 0.56 - 1.4

Chapter 4

Implementation

4.1 Data

The data used in the project was acquired at the Modified NIST database [1], which is a subset of a larger study done at the National Institute of Standards and Technology of United States of America. It is preprocessed and formatted, therefore offering a good learning tool.

Data preparation and format

Each digit image was centered in 28x28 pixel format and later transformed into a vector. Each vector element represents a pixel in the image in the form of **blackness** of the said pixel. Also each image was labeled in the form of vector with a size of 10.

Input

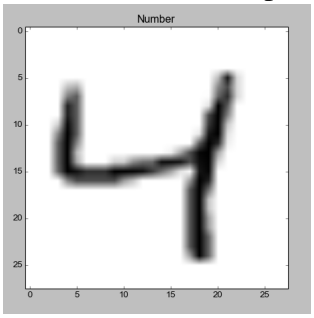
Each input to the neural network will be in the form a vector (784x1). A sample input if represented as a 28x28 matrix could look as follows:

Table 4.1: Number matrix representation

R/C	**1**	**2**	**...**	**28**
1	0	0	...	0
2	0	0	...	0
.	0.9	0.7	...	0.2
28	0.2	0.1	...	0

And if plotted according to each pixel's blackness value, such a number could look as follows:

Figure 4.1: Number visual representation



Output

Each number in the set is also labelled by a vector of (10x1) size. So, for example label for the above number would be:

```
number_4 = [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
```

Listing 4.1: Vector for number 4 in Python script

Datasets

The dataset comes with a training set of 70000 examples. For the project training set was partioned into 50000 examples for training and 20000 examples for testing and validation.

Table 4.2: Datasets

Dataset	Inputs	Labels
training dataset	50000	50000
validation dataset	10000	10000
testing dataset	10000	10000

4.2 The neural network

Layers have been choosen in the following way: 784, 30, 10. I found that a single 30-node hidden layer was a good balance, especially when considering the large input size.

Several **cost functions** were tried for different implementations, but to be consistent I tried to use MSE whenever I could.

For the **optimization method** variations of gradient descent method were used.

4.3 Numpy based implementation

For the codes used in this implementation, *Neural Networks and Deep Learning* reference was my main reference [18], especially for **backpropogation** and **stochastic gradient descent** algorithms.

4.3.1 Usage

The FeedForward class implements a neural network with stochastic gradient descent optimization and backpropogation. Its only dependency is NumPy, which is used for matrix multiplications and similar actions.

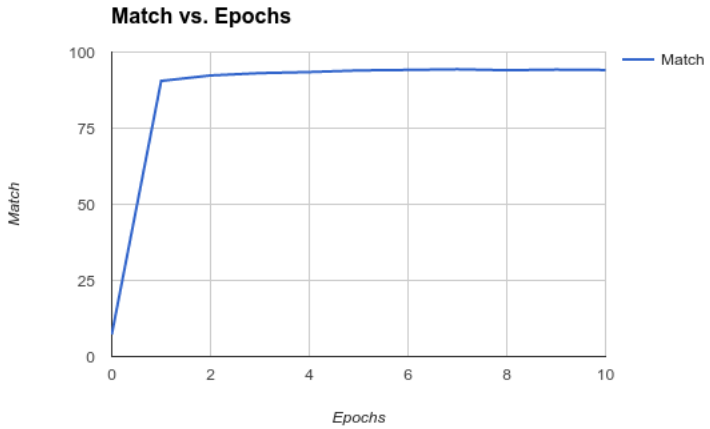
```
# Creating a feed forward neural network with
net = FeedForward([784, 30, 10])       # [Input, Hidden, Output]

# Training for 10 epochs with 3.0 learning rate.
net.train(training_data, 10, 3.0, test_data)
net.save_state()
```

Listing 4.2: Numpy based implementation.

4.3.2 Result

This network can get good match results with relatively small amount of epochs. After 10 epochs, it reached a %94.12 match rate.

Figure 4.2: Epoch vs matches



4.4 PyBrain implementation

4.4.1 Usage

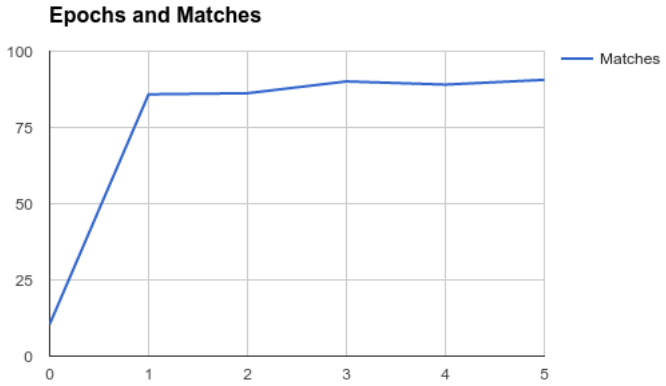
```python
from feed_forward_pybrain import FeedForwardPB

# Creating the neural network with default values.
nn = FeedForwardPB()            # [784, 30, 10] as layers.
nn.update_learningrate(0.5)     # Updating learning rate for training.
nn.train(5)                     # Training for 5 times.
```

Listing 4.3: PyBrain implementation

4.4.2 Result

Maximum match reached with this implementation was **%90.62**.

Figure 4.3: PyBrain implementation



4.5 Keras implementation

Keras implementation used TensorFlow backend, so it offers easy to use wrappers around TensorFlow and also good speed improvements over the other two options.

4.5.1 Usage

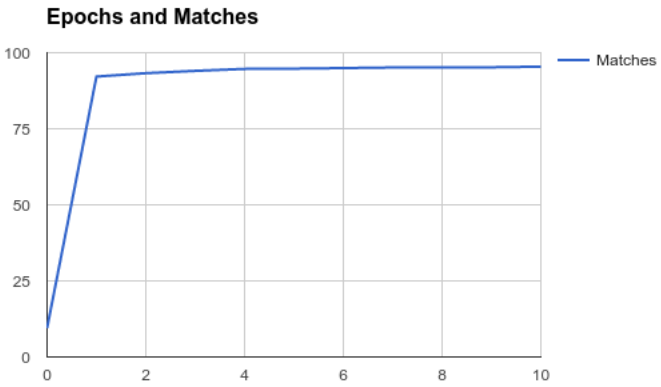
```python
from feed_forward_keras import FeedForwardKeras

# Create a default neural network.
# Layers       : 784, 30, 10
# Optimization : RMSProp
# Cost Func.   : MSE
nn = FeedForwardKeras()
nn.train(5, verbose=1)      # Train for 5 epochs.
nn.evaluate()               # Evaluate matches.
```

Listing 4.4: Keras implementation

4.5.2 Result

With this implementation, **%95.43** match could be reached after 10 epochs.

Figure 4.4: Keras implementation



Chapter 5

Conclusion

The first approach, using just **Numpy**, was a very good way to get a better understanding of how neural networks work. I was able to see how different algorithms could be implemented and to have easier access to neural network parameters and so on.

However, for experimenting and prototyping a library would be a better option. So for next option, I checked two libraries: *PyBrain* and *Keras*.

PyBrain implementation was the slowest and least performant of all three implementations. Epochs took around 2 minutes, making configuring network parameters require a little more patience.

PyBrain, I had initially assumed, was the most "low level" library among the options, therefore would be a good next step after Numpy implementation. But I found working with Keras much more pleasant.

Keras can use TensorFlow as its backend, so it provides an easy to use interface to TensorFlow and more importantly it takes advantage of its speed. It was quite easy to try new optimization algorithms and cost functions while using this library.

Next time, I would mostly likely start directly with Keras and later move on to TensorFlow.

In summary, I have found Python and its libraries very good to work with data and neural networks. Libraries like Numpy, SciPy and Matplotlib are very mature and capable, and they provide stable foundations for other libraries to built-upon.

One thing I missed from Matlab was the training GUI of neural network module, where you can get easy access to graphs for performance and other parameters. Although it's theoretically possible to do the same thing with Python, it's often a more CLI-driven development, and it's up to the user to create the visuals and outputs they need.

Bibliography

[1] LeCun, Y. *Mnist database of handwritten digits*, http://yann.lecun.com/exdb/mnist/

[2] *PyPi, Python Library*, https://pypi.python.org/pypi

[3] *Numpy*, http://www.numpy.org/

[4] *Numpy documentation*, https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html

[5] *Matplotlib*, http://matplotlib.org/

[6] *TensorFlow*, https://www.tensorflow.org/

[7] Kuster, D., *The good, bad and ugly of TensorFlow*, *KD Nuggets*, 2015

[8] *PyBrain*, http://pybrain.org/

[9] *Theano*, http://deeplearning.net/software/theano/

[10] *Scikit-learn*, http://scikit-learn.org/stable/

[11] Lorica, B., *Six reasons why I recommend scikit-learn*, https://www.oreilly.com/

[12] *Keras*, https://keras.io/

[13] OpenCV, http://opencv.org/

[14] Neiger, V. *Handwritten digits recognition using openCV*, Western University, (2015)

[15] Gurney, K. An introduction to neural networks, UCL-Press (1997)

[16] Heaton, J. Introduction to math of neural networks, Heaton Research, (2012)

[17] Rere, L.M., Fanany, M.I, Arymurthy, A.M., Simulated annealing algorithm for deep learning, Procedia Computer Science, Volume 72, (2015) Pages 137-144

[18] Nielsen, M, Neural Networks and Deep Learning, 2016, http://neuralnetworksanddeeplearning.com/