University of Tartu

Artificial neural network for image classification

Computational neuroscience project

Author: Sten Sootla

Mentor: Tambet Matiisen

Tartu 2015

Table of Contents

Introduction

In the last few years, there has been a lot of hype about artificial neural networks and other machine learning methods, mostly due to the success of a series of applications in the industry that use this technology. This in turn has spawned the creation of relatively powerful open source libraries and services that make this technology accessible to average developers. In order to use these libraries optimally, programmers have to have a basic understanding of the underlying implementation of the algorithms.

The purpose of this paper is to do just that. First the reader is given a very broad overview of artificial neural networks and the kind of problems that can be solved with them. After that, the most important concepts of neural networks are described individually, based on an implementation of a custom neural network that is a able to learn to classify 10 different classes of images.

The simple neural network that is implemented in conjuction with writing the paper is first and foremost exepcted to classify images more accurately than random classification would. Since there are 10 classes, randomly classifying the images would produce an accuracy of 10%. As the author has no previous experience working with neural networks, it is hard to make any further predictions that are not merely speculations. Having said that, it would be wonderful if the relatively trivial network implemented in this project would be able achieve an accuracy of 50% on test data. To give the reader some perspective, the best network according to Kaggle's leaderboard is able to classify the images from the CIFAR-10 dataset with a 95% accuracy.[1]

1. Aritificial neural networks

Artificial neural networks (ANNs) are statistical learning algorithms that are inspired by properties of the biological neural networks. They are used for a wide variety of tasks, from relatively simple classification problems to speech recognition and computer vision.

ANNs are loosely based on biological neural networks in a sense that they are implemented as a system of interconnected processing elements, sometimes called nodes, which are functionally analogous to biological neurons. The connections between different nodes have numerical values, called weights, and by altering these values in a systematic way, the network is eventually able to approximate the desired function.

Each node in the network takes many inputs from other nodes and calculates a single output based on the inputs and the connection weights. This output is generally fed into another neuron, repeating the process. When equipped with the information given in the last sentence, one can easily envision the internal hierarchical structure of the artifical neural network, where neurons are organized into different layers, as depicted below. The input layer receives the inputs and the output layer produces an output. The layers that lye in between these two are called hidden layers.

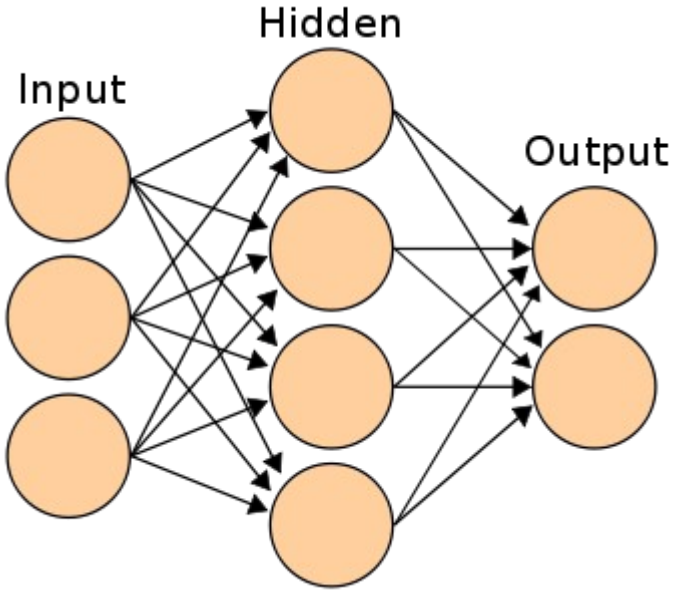

Figure 1. A simple neural network with one hidden layer.[2]

The hidden layers can be thought of as individual feature detectors, recognizing more and more complex patterns in the data as it is propagated throught the network. For example, if the network is given a task to recognize a face, the first hidden layer might act as a line detector, the second hidden takes these lines as input and puts them together to form a nose, the third hidden layer takes the nose and matches it with an eye and so on, until finally the whole face is constructed. This hierachy enables the network to eventually recognize very complex objects.[3]

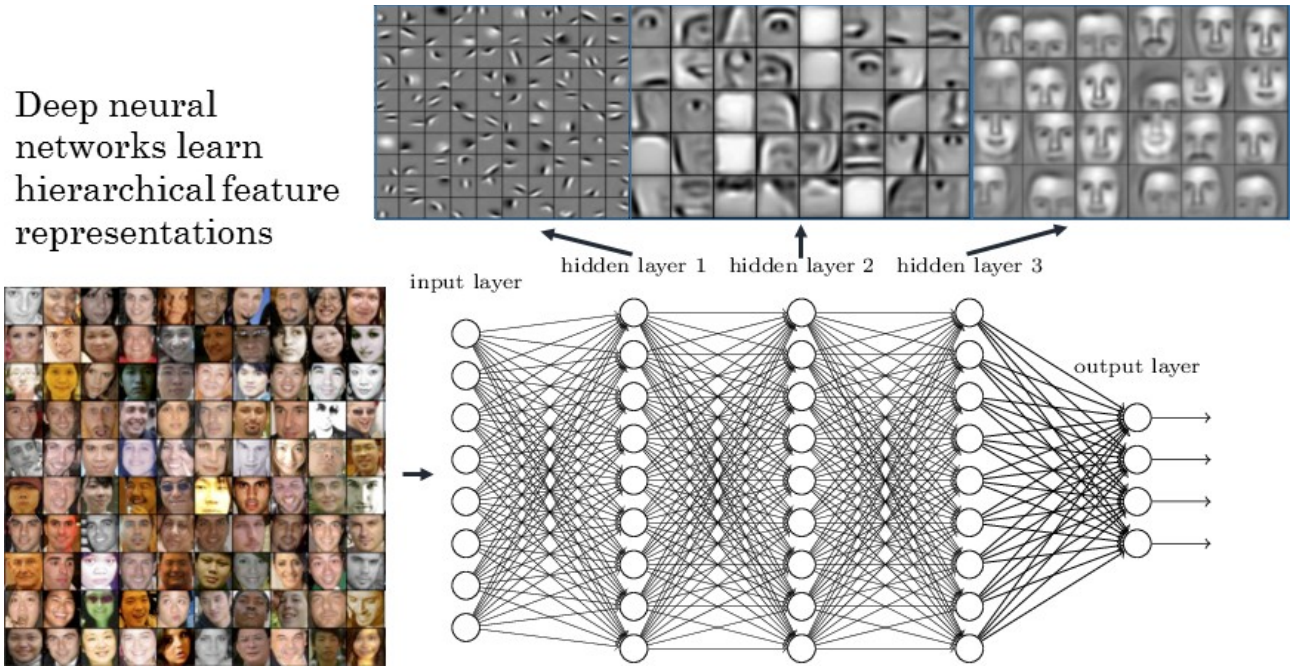

Figure 2. An example of neural network layers as feature detectors.[4]

As stated before, the network is able to accurately approximate an arbitrary function by altering its weights in a systematic way. Initially, the weights are given random values and the network must be trained in order to find the weigth parameters that produce the desired effect. In order to achieve this, we must first compare the neural network output to the desired output, calculate the error and use this to adjust the weights of the network in proportion to their contribution for the error in the output.

Because of the computationally expensive backpropagation phase that has to be done for every input set, neural networks are very slow learners and need huge amount of computational power to produce desirable results. Even more so, if we consider that today's top of the line networks contain millions of neurons and in some extreme cases, up to billion connection weights. For example,

Google's neural network that was able to recognise 20,000 object categories with 15.8% accuracy was trained on a cluster of 1 000 servers (totaling 16,000 CPU cores) for three days.[5]

2. Implementation

This section describes the implementation of a simple fully-connected feed-forward artificial neural network with one hidden layer that is able to classify 10 classes of images from the CIFAR-10 dataset. The network is wrtten in Python 2.7 and the only external libarry it is dependent on is numpy 1.8.2. The code for the artificial neural network can be found here: https://github.com/stensoootla/neural_network

2.1 CIFAR-10 dataset

CIFAR-10 is a popular computer vision dataset that is used by object recognition algorithms. It is a labeled subset of 80 million tiny images dataset that was collected by Alex Krizhevsky, Vinoid Nair and Geofrrey Hinton. The dataset is made up of 60 000 32x23 colour images that are organized in 10 classes, each of which consists of 6000 images. These 60 000 images are divided into 50 000 training images and 10 000 test images.[6]

Here are the 10 different classes in the dataset and some examples of the pictures that belong to these classes:

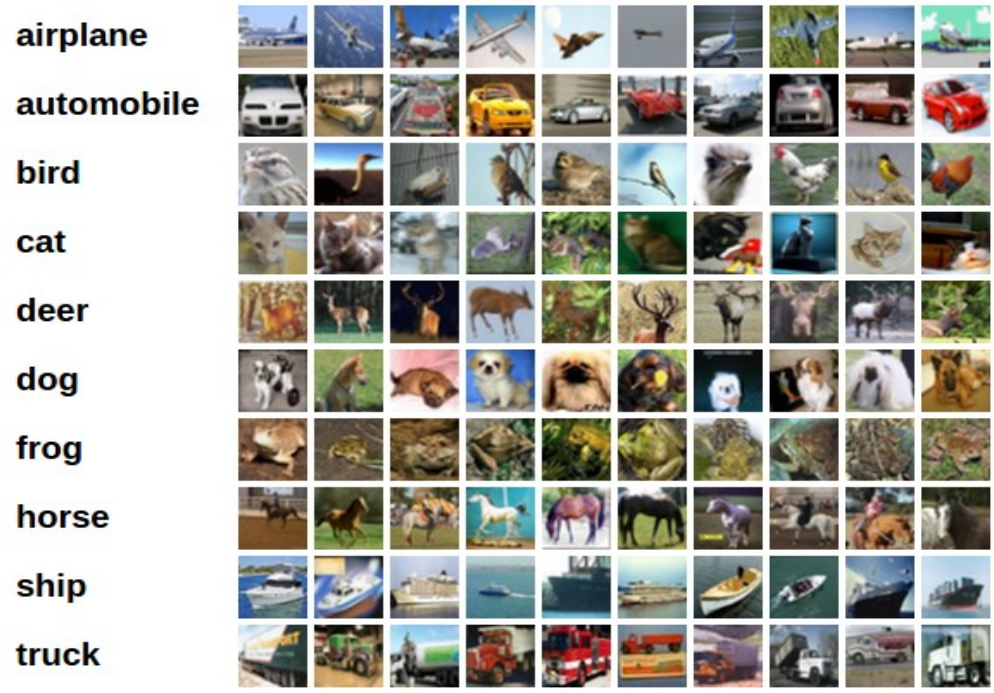

Figure 3. Example of the CIFAR-10 dataset.[7]

The pictures are stored in memory as six 10 000 x 3072 numpy arrays where each row represents a single image. The 3072 values in each row are logically divided into 3 chunks of 1024 values, each representing the red, green and blue channel values, respectively. The images are stored in row-major order, which means that every i-th logical block consisting of 32 elements in the row represent the i-th row channel value in the actual image.[6]

The labels are stored as six lists of 10 000 elements, where each element is an integer between 0 and 9. These numbers map to the class names, so that 0 maps to the airplanes class, 1 maps to automobiles etc. The i-th element in the labels list is the label for the i-th picture in the numpy array that was described in the previous paragraph.[6]

2.2 Activation function

Each node in the neural network takes many inputs and produces a single ouput based on the weighted sum of these inputs.

A relatively simple model that achieves this functionality – the perceptron - was invented already in 1957 by Frank Rosenblatt. The perceptron sums together all the inputs and their corresponding weights, compares the result to some threshold and outputs a discrete value based on this comparison. For example, if the sum is bigger than or equal to the threshold, the node should output 1, otherwise the node emits 0. Mathematically, it can be formalized as follows:

$$f(x)=\begin{cases}1 \; if \sum_i w_i x_i \geq threshold \\ 0 \; if \sum_i w_i x_i < threshold\end{cases}$$

where w and x are the weight and input vectors, respectively.[8]

By moving the threshold to the left side in both equations and replacing it with the bias term b, which is equal to the negatative threshold (b = -threshold), we get a new and more commonly used representation of the perceptron rule:

$$f(x)=\begin{cases}1 \; if \sum_i w_i x_i + b \geq 0 \\ 0 \; if \sum_i w_i x_i + b < 0\end{cases}$$

where w and x are the weight and input vectors, respectively, and b is the bias term.[8]

Although in theory, one can use perceptrons to compute any function, it is almost never used in practice, mostly due to its discrete nature - a small change in the weights might cause the perceptron to produce a drastically different output, which is not desirable for learning.[8] Accordingly, in this neural network implementation, a sigmoid activation function is used instead of the perceptron, which is actually quite similar to the former, but gets rid of its shortcoming. The formula for the sigmoid function is the following[9]:

$$\sigma(x)=\frac{1}{1+e^{x}}$$

When representing it as a node in the neural network with weights, inputs and biases, the function takes the following form:

$$\sigma(x)=\frac{1}{1+e^{-\sum_i w_i x_i + b}}$$

The intuition behind the function is that it takes a real valued input and outputs a value between 0 and 1. The bigger the input value, the closer the output is to 1, and vice versa. In that respect, the simgoid function is very similar to the perceptron, because for most of the inputs, it produces an output that is really close to either 1 or 0, as can be seen from the following graph:

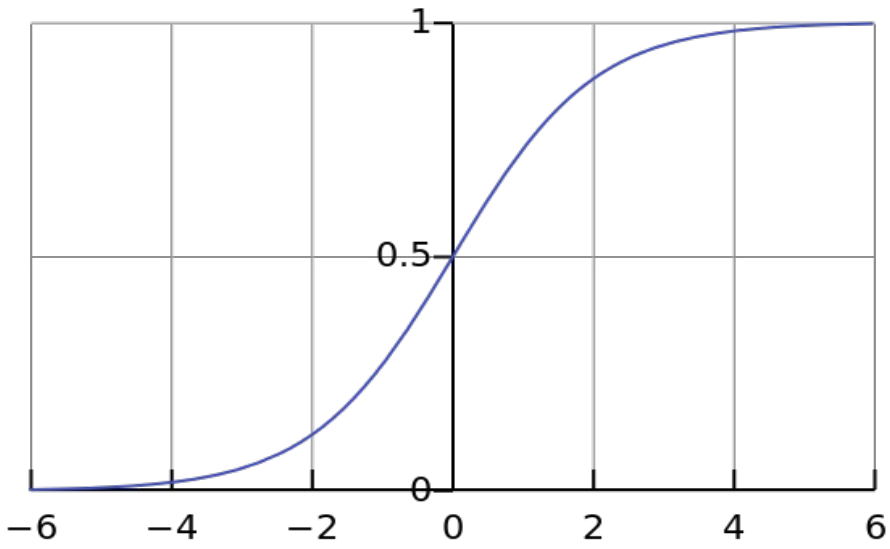

Figure 4. The sigmoid activation function.[10]

2.3 Loss function

The task of the neural network implemented in this project is to learn to classify images. In order to learn something, one must be provided with some feedback about his current performace. The job of the loss function is precisely that: to evaluate the accuracy of the neural network. As the name suggests, if the loss is low, the neural network is doing a good job at classifying the images, and the loss will be high if the network is not guessing the right classes.

In order to calculate the loss for a specific guess, the neural network's output must first be interpreted as class scores. This is the job of the score function, which takes the values from the output layer nodes and calculates the probability that a given input represents a specific class. The score function used in this project is called the softmax function and is given by the following formula:

$$p_i = \frac{e^{z_i}}{\sum_{j \in group} e^{z_j}}$$

where z is a vector of output nodes and and *group* denotes a set of indexes of every node in the output layer.[11]

To illustrate the workings of the score function, an example is probably needed. Let's say we have a neural network that has the job to classify 3 different classes of data. For this, we need 3 nodes in the output layer, each “voting” for a different class (first neuron represents the first class, second neuron represents the second class etc). The 3 nodes in the output layer could have the values 3, 5, 4, so z = (3, 4, 5). After calculating the softmax probability distribution, we get p=(0.09, 0.244, 0.665). Since the third node has the highest score, we say that the network is classifying the input to be from class 3.

After calculating the probability score for each class, the cross-entropy loss function can be used to calculate the network's total loss:

$$C = -\sum_i t_i \log p_i$$

where t is a vector representing the correct class and p is the probability distribution that was described above.[11] The target vector t has the element 0 at every position, except at the index that corresponds to the integer representing the correct class, where the value is 1. For example, if the correct class is 3, the target vector is (0, 0, 0, 1, 0, 0, 0, 0, 0, 0).

2.4 Gradient descent

In the previous section we defined a loss function, which essentially tells us how accurately the neural network is able to classify data. Now, we need a way to actually act on this information and somehow minimize the loss of the network during training by changing the weights between different nodes. That is done with the gradient descent algorithm.

Gradient descent works by finding the gradient of the loss function with respect to the weights of the network, telling us the direction where the function increases the most. In order to minimize the loss, it is required to subtract a fraction of the gradient from the corresponding weight vector. To understand the algorithm intuitively, an analogy can be used.

One can imagine being placed blindfolded at the top of the mounatin and given a task to find a way down. The strategy a blindfolded person would use would be to feel the ground just before him and take a small step in the direction where the ground feels to descent the steepest. Doing this iteratively, the person is guaranteed to arrive at a location where the ground just around him is not descending. Since the mountain is not always going downwards at every location, but rather has a very fluctuating terrain, it is not guaranteed that the place the person feels the ground to be horizontal in every direction is actually the steepest place on the mountain. When it is not, it is called a local minima and it can be visualized with the following picture:

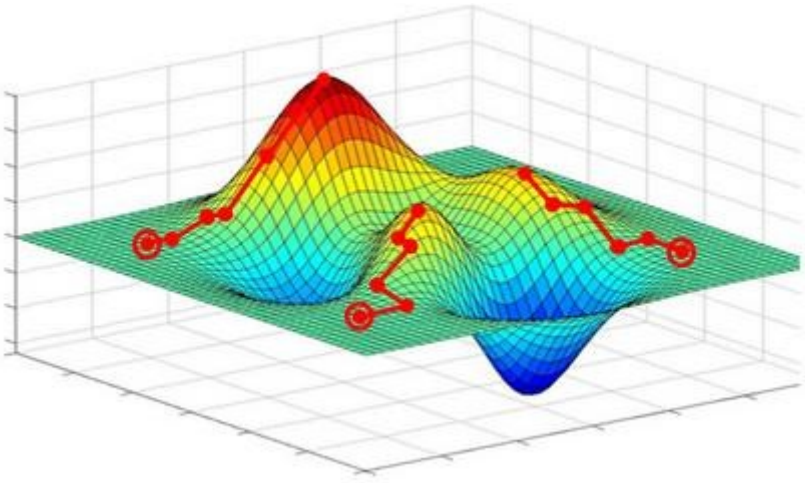

Figure 5. A 3 dimensional graph representing the gradient descent algorithm.[12]

As can be seen from the graph, the hiker's move along the paths (depicted as red lines) that continuosly take them down the mountain when it is descending, but fail to reach the absolute

bottom once the ground is stable.

In order to get out of local minima, several methods can be used, most notably by adjusting the learning rate, which is intuitively the length of the steps that the hiker takes in a specific direction. When the learning rate is low, the hiker takes very small steps and can be almost certain that he is descending, but it takes a lot of time to reach the bottom of the mountain. On the other hand, when the learning rate is high and he takes long steps, he might reach the ground faster, but with a relatively big possibility of going in the wrong direction. For example, if the hiker feels that the ground is descending just before him, but then takes a very big step in that direction, he might end up at a location that is actually higher than he was at previously.[13]

Another straigtforward method that is used to improve gradient descent and to help it get out of local minima is called the momentum update, which adds a fraction of the previous weight update values to the current one. The intuition behind the method is very simple and can be easily derived from its name: the faster the hiker is descending (i.e the steeper the hill), the more momentum he has due to inertia when he makes his next step, hence the step will be longer if he is descending fast, and shorter when the descent is not as drastic.[14]

2.5 Backpropagation

Backpropagation is a practical realization of the gradient descent algorithm in multilayered neural networks. It calculates the gradient of the loss function with respect to all the the weights in the network by iteratively applying the multivariable chain rule.

Applying the backpropagation algorithm to a neural network is a two way process: we first propagate the input values through the network and calculate the errors, and then we backpropagate the errors through the network backwards to adjust the connection weights in order to minimize the error. The algorithm calculates the gradient of the loss function with respect to the weights between the hidden layer and output layer nodes, and then it proceeds to calculate the gradient of the loss function with respect to the weights between the input layer and hidden layer nodes. After calculating the gradients, it subtracts them from the corresponding weight vectors to get the new weights for the connections. This process is repeated until the network produces the desired outputs.[15]

The whole process is better explained with an example. In the neural network implemented in this

project, the gradient of the loss function with respect to the weights between the hidden layer and output layer nodes can be computed as follows:

$$\frac{\partial loss}{\partial w_2} = \frac{\partial loss}{\partial z} \frac{\partial z}{\partial w_2}$$

where loss is the cross-entropy loss described in section 2.3 and z is a vector that holds the values of the output layer nodes.

Going further down the line, the gradient of the loss function with repsect to the weights between the input layer and hidden layer nodes is calculated with the following formula:

$$\frac{\partial loss}{\partial w_1} = \frac{\partial loss}{\partial z} \frac{\partial z}{\partial y} \frac{\partial y}{\partial w_1}$$

where y is a vector that holds the values of the hidden layer nodes.

2.6 Results

After implementing all the previously discussed functionality in Python, the resulting artificial neural network has an accuracy of 38% on previously unseen data. To achieve these results, the network was trained for 10 epochs (the number of times the full dataset is passed through the network) with a batch size of 20 (the number of inputs that are passed through the network before updating the weights) and a learning rate of 0.001 and a momentum of 0.5. It took approximately 223 minutes for the network to finish the computation.

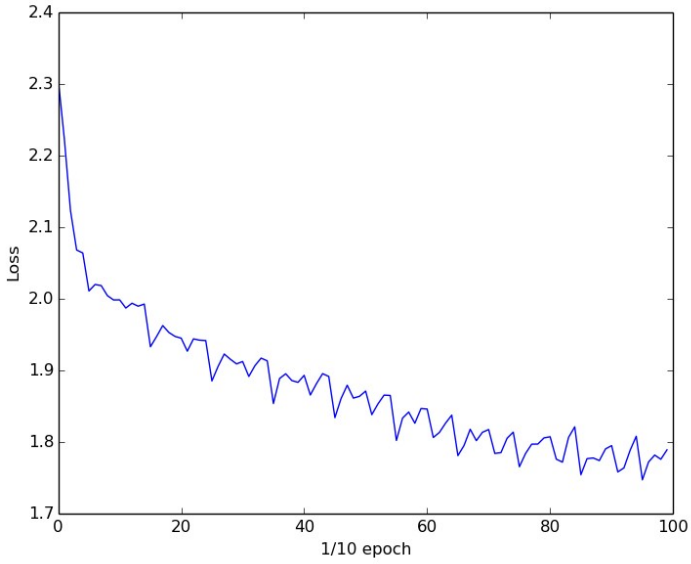

Figure 6. The loss function over time.

Conclusion

The aim of this project was to implement a simple feed-forward neural network and to give an overview of the various concepts essential to neural networks based on this implementation. Both of these goals have been achieved.

In the first chapter, the reader was given a broad overview of neural networks in general. In the second chapter, the individual components that are essential to almost all neural networks were described in full detail, starting with describing the perceptron and sigmoid activation function, then moving along to define the cross-entropy loss function and finally introducing the reader to gradient descent and backpropagation, which are used to train the network.

The implementation of the network can be considered satisfactory, but it is by no means successful, as the initially desired accuracy of 50% on test data was not achieved. Surprisingly, many of the major setbacks during development were due to the implementation subtleties of neural networks, and not due to the faulty misunderstandings of the main concepts. For example, it took the a long time to figure out that the vanilla implementation of the sigmoid activation function is not used in practice, as it is not numerically stable and results in overflowing to infinity. Secondly, working with matrcies proved to be more challenging than expected, because of the author's relative unfamiliarity to numpy.

The neural network can be improved a number of ways. First, the code can be refactored to make the whole implementation more understandable and efficient. Secondly, more hidden layers could be added in order to provide the network with the tools to learn more complex relationships in the data. Lastly, the whole implementation could be rewritten as a convulational neural network, which generally produce better results than fully-connected networks at classifying images.

References

1. https://www.kaggle.com/c/cifar-10/leaderboard

2. http://upload.wikimedia.org/wikipedia/commons/thumb/e/e4/Artificial_neural_network.svg/350px-Artificial_neural_network.svg.png

3. http://neuralnetworksanddeeplearning.com/chap1.html#toward_deep_learning

4. http://www.rsipvision.com/wp-content/uploads/2015/04/Slide6.png

5. http://arxiv.org/abs/1112.6209

6. http://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf, Alex Krizhevsky, 2009.

7. http://www.cs.toronto.edu/~kriz/cifar.html

8. http://neuralnetworksanddeeplearning.com/chap1.html#perceptrons

9. http://cs231n.github.io/neural-networks-1/

10. http://upload.wikimedia.org/wikipedia/commons/8/88/Logistic-curve.svg

11. https://d396qusza40orc.cloudfront.net/neuralnets/lecture_slides/lec4.pdf

12. http://www.forexartilect.com/img/momo.jpg

13. http://cs231n.github.io/optimization-1/#opt3

14. http://cs231n.github.io/neural-networks-3/#sgd

15. http://en.wikipedia.org/wiki/Backpropagation#Summary