BUILDING THE ARTIFICIAL NEURAL NETWORK ENVIRONMENT

Artificial Neural Networks in plane control

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Bachelor's thesis

Valkeakoski, Automation Engineering

Fall 2016

Daniil Naumetc

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Automation Engineering
Valkeakoski

Author Daniil Naumetc **Year** 2016

Subject Artificial Neural Networks in plane control

ABSTRACT

These days Artificial Neural Networks have penetrated into all digital technologies that surround us. Mostly every online service like Facebook, Google, Instagram are using Artificial Intelligence to build better service for their users.

Google Self-Driving Car Project that started several years ago already have results as driverless cars already moving on the streets of California.

Artificial Intelligence makes a breakthrough in Medicine as well. Such programs already successfully find disease reasons and make clinical decisions.

Boldly saying, we can expect AI replacing human beings in most spheres of our lives.

This Thesis idea is to make approach for developing AI for piloting, since the topic is not highly developed yet.

Keywords Artificial neural networks, autopilot, artificial intelligence, machine learning

Pages 32 pages including appendices 9 pages

CONTENTS

1 INTRODUCTION

Every program is built up on a model. The model can be full or not, some of variables may be so insignificant that won't really affect the result. But some variables that were not taken into count can modify the result in a very unexpected way, so that a hardware that works with an incomplete software can cause damage to an environment it operates with.

Along with the open model based class of problems, there is a class of problems, that has a lack of information like how does a problem can be solved. The model of a process can be so large that it is impossible to cover it all, or even only a small part.

To deal with that kind of problems where some variables are not known we can replace them with the artificial ones(nodes) and try to find their role using the recorded experience of inputs and outputs. The pre-learned model found while solving the problem can be not as full as the real world model, but using neural networks we can start solving the problem not trying to understand how it works. So, we can try to balance between time spent on development and the outcome, that can be accurate *enough*. On the other, hand the model learned by large amount of data can be much more accurate than any other algorithm that was built by a human being.

Also, it is necessary to mention that ANN can become an extremely effective black-box in topics where human doesn't know how to make machine do some operations. For instance, processing natural language is a kind of a problem, where it is not obvious how to program the voice recognition in a classic way. A neural network is the only way, lets voice recognition works in Google or Apple.

Summarising the first impression, ANN is a program that can learn possible results by the data given on the inputs. To do so it builds abstract models of relationships between variables.

Modern systems that operate planes are developed enough to be some how responsible for people who uses airline services. Autopilot systems were evolving for decades and one technology is not enough to upend the whole industry. I also believe that such systems that I want to propose are being developed right now.

Since all the programs and models were built by humans, they are quite heavy and have a strong theoretical background. It can become an interesting topic to build a plane, that can learn to fly and head the course by itself. There are two problems that can be covered by this project: hard predictability of natural processes and impossibility to work

in an unknown environment. Second problem is seen quite interesting for me because for computers unknown environment can be a place with unusual conditions, for example air pressure, or broken left wing with the different lift characteristics from the right one can be a good example of unknown environment.

2 BACKGROUND

2.1 Neuron

Brain of every being is represented by a large Neural Network, the amount of neurons and their structure as a network determines a level of intellect of that creature. But the neuron itself remains the same among most of creatures. The basic idea behind the neuron is to receive signals of other neurons, combine a transformed signal based on inputs and transfer it to other neurons. Signals are transferred as electrical signals from one neuron to another and it is always changed by the last one.

Basic natural neuron consists of three main parts: dendrite, nucleus and axon. Dendrite is a terminal, that receives electrical signals and transfer it to nucleus. Nucleus is a part that remembers previous experiences in a way, that it can give a positive output if the input satisfies it or negative on the contrary. Then the output goes to axon, which is connected to several other neurons of the network.

It is important to understand about neurons, that each of them can be represented as some mathematical (logical) function. For example, we can define a neuron that we want to give high positive signal (1) as cat and neutral (0) as “other”. Dendrites(inputs) can be programmed to receive 0 or 1 from other neurons responsible for cat wool, four paws and tail. If all three inputs have 1, that makes nucleus give 1 as well. [1]

The most attractive feature of neuron is that it can learn. The function represented a neuron can change standard variables in time while it is learning. The results with the same inputs may vary before and after learning activities. Basically, if every time neuron gives us an opinion, that a thing that has four paws and a tail is a cat, we say that it is not a cat, it can lead to extinction of the particular neuron.

2.2 Neural Network

One neuron can solve quite simple problems and doesn’t represent value

by itself. But a collection of neurons connected in a certain way can solve quite complex problems. Basic NN contains one or several layers of neurons. Layer represents a set of neurons that are not connected between each other in most cases; and stay between other layers. Some neuron layers are defined as input and output layers which depends on its position.

The simplest kind of neural network is a *single-layer perceptron* network, which consists of a single layer of output nodes. The inputs are fed directly to the outputs via a series of weights.

Between input and output neurons can be a set of other layers, which called hidden layers. They are responsible for building complex high-level abstractions.

The example with the cat in section 2.1 is quite empirical one. In real life we faced to much complex systems. If we want to define a cat, we require lots of other properties of an animal: colouring, height, weight, face and so on. Amount of properties can be counted in hundreds and thousands, but neuron can't be connected to each of them, and even if could that would keep NN very simple. In such case neurons in hidden layers can handle processing of abstractions. Neurons in the middle hidden layers recognise patterns from previous layers and the results being sent forward.

In complex neural networks neurons in hidden layers can represent abstractions that can't be understood by humans, but in such problems like image recognition, one layer can recognise patterns like circles, that could be used to recognise wheels of a car.

2.3 **Artificial neuron**

Artificial neuron represents real neuron mathematically. Real neuron has very complicated concept. It works by biological mechanics. Many features play role in it: hormones, speed of conduction, chemical composition of the neuron, presence of elements in organism. Artificial neuron works by very basic rules. Nucleus replaced with simple mathematical function called "activation function", dendrites are multiplication function, they multiply incoming signal by a weight that is always changed after learning. Axon has no transformation functions.

There are lots of activation functions used in programmed neuron, but the most basic and effective one is sigmoid function. Sigmoid can input signal of any range but outputs signal only between -1 and 1. That makes much easier to work within a network. Moreover, only after the sigmoid was used was found another characteristic that increases effectiveness of the function. Neural network learning process is very *expensive* for the

computer, because of the amount of mathematical computations. Sigmoid function simplifies the process because the calculation of derivative of sigmoid function is very easy. That would be covered furtherer. [2]

2.4 **Artificial Network**

Due to complexity of real neural network we can't make a full model of it. Creating the model that repeats all the processes influencing the neurons would be a rough problem to solve. Even though, right now, the models built to recognise voice or pictures are relatively simple comparing to brain of a smallest being and they require lots of computational power from an operating unit.

The way that lets us use neural networks is quite simple. First of all, programmers of neural networks do not think in terms of neurons as computation units. What plays a role in programming of artificial neural networks are the weights. That weights that represent dendrite in natural neuron is the most important thing because only them are always changing while neural networks learn. After learning all the things used to create a network remains the same except weights. In some cases, learning rate can be changed, but it is not necessary to know the learning rate to use the learned network.

Secondly, in most cases the network is just a program. The memory of a network, as told before, is stored as weights. In most of modern Neural Networks neurons of two particular layers are connected between each other (not inside of one layer). This type of connection is called Each-to-Each. The interesting feature of this connection type is that all weights can be easily stored as massive. For example, if we have 2 layers connected Each-to-Each: 10 neurons and 5 neurons, because each of 5 neurons in the second layer would have 10 connections from the first layer. We can easily store them as an array of two dimensions, 5 and 10 respectively, and treat them as matrixes.

2.5 **Simple perceptron**

Single layer perceptron is an example of a basic feed forward network, which was the first artificial neural network built. It has just two layers: one input and one output. Neurons in the first layer receive signals and transfer them to the output neurons. As accepted first layer doesn't apply any filter to data and transfer it directly from dendrite to axon. After that signals got multiplied by weights and continue to activation function and further outcome as output signals.

As an example of how it works we can model a single layer perceptron with AND activation function. We will have two input neurons and one output. [3]

The output neuron would have two weights w1 and w2 that come from Neuron 1 and Neuron 2 respectively. The output would be T. The output of input neurons is meant to be an input itself and equal to I1 and I2. The perceptron separates input into two categories: the one that cause fire on output and the one that does not. It does this by looking at

$$w_1 I_1 + w_2 I_2 < t,$$

If LHS is smaller than t it doesn't fire, otherwise it fires. That is, it is drawing the line:

$$w_1 I_1 + w_2 I_2 = t ,$$

and looking at where the input point lies. Points on one side of the line fall into 1 category, points on the other side fall into the other category. And because the weights and thresholds can be anything, this is just *any line* across the 2 dimensional input space.

So what the perceptron is doing is simply drawing a line across the 2-d input space. Inputs to one side of the line are classified into one category, inputs on the other side are classified into another. e.g. the OR perceptron, $w_1=1$, $w_2=1$, $t=0.5$, draws the line:

$$I_1 + I_2 = 0.5$$

across the input space, thus separating the points (0,1), (1,0), (1,1) from the point (0,0):

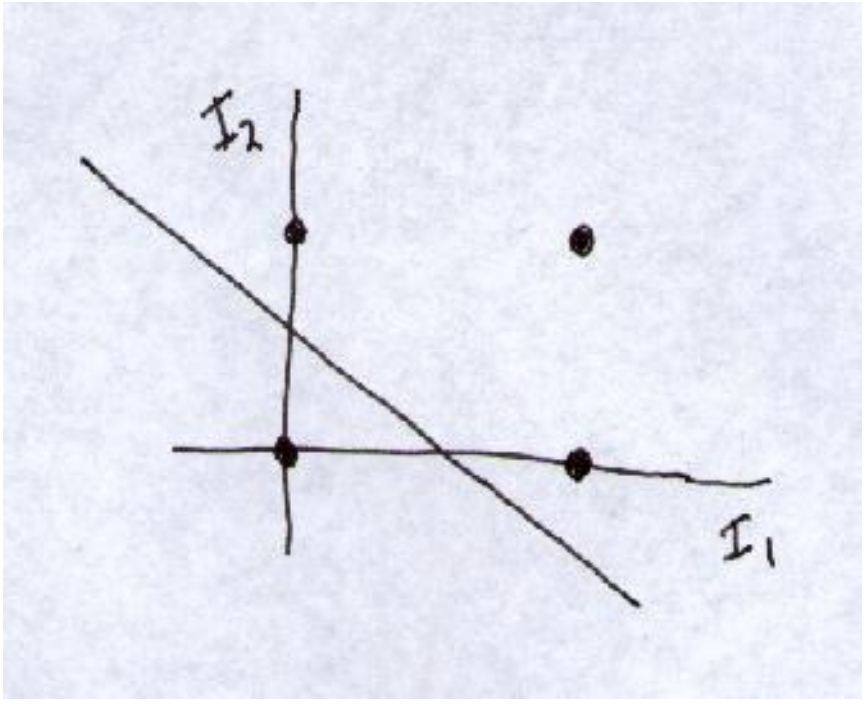

Figure 1: Input space

As you might imagine, not every set of points can be divided by a line like this. Those that can be, are called *linearly separable*.

In 2 input dimensions, we draw a 1 dimensional line. In n dimensions, we are drawing the (n-1) dimensional *hyperplane*:

$$w_1 I_1 + \ldots + w_n I_n = t$$

2.6 **Backpropagation**

Backpropagation is the most basic idea behind the Neural Networks. It lets neural networks learn.

If we consider that feedforward network is called by feeding input signals through the entire network towards its output. The backpropagation mechanism is the opposite to feeding forward – feeding backward.

To arrange learning process we need to propagate the teaching sample input-output pair to the experience-less network and get the predicted output of the network. Then what we would is to show the network which how differs its prediction to original one. For that task we would calculate the difference between original output and imperfect output done by the network. If we will just send the original output back through the network will be just seamless. For such a case was invented a way to backpropagate the error itself.

To backpropagate the error first we will apply it to the input weights of the output neurons. We will compute ΔW_i for each input weight and the signals outputted by previous layer. After we calculated the outputs of each neuron of previous layer we need to backpropagate the error through the activation function. We can't just send the error through it, so what we do is we get the derivative of the function and through the derivative we send the error. After that all the steps taken we repeat for each neuron and each layer down to the input of the network. [4]

This trick is the basic for all the most used neural networks, like feedforward, convolutional, recurrent and Long Short-Term Memory cells.

2.7 **Input and output, Normalization**

The most important and time consuming step in working with Artificial Neural Networks is preparing data. Since we are working with mathematical functions it is not possible just to input some values in random format. In machine learning and in neural networks learning, in particular, data should be in right format. The process of formatting the data is called "Normalization". Artificial Neural Networks input data depends on activation function. Usually data should be brought to interval [0,1] or [-1,1] if the input neuron's activation function is Sigmoid or Tanh, or similar. If normalization is not applied, input data would exert additional influence.

Neural network outputs data in same format as inputs, usually, between 0 and 1.

For example, if we have a problem of image recognition, popular MNIST, where the on input we get a picture and we should output a digit, that corresponds to an image. Raw images for that problem are pictures of 28x28 pixels, each pixel has a range of 256 colors. To input this picture, we make an array, that is called "Tensor" and contain a set of numbers. In our case the tensor will have 784 variables. Each of them should be between 0 and 1, where 0 is white and 1 is black. If we have a pixel that is middle gray valued as 128, to store it in required range we divide the value by 256. So we get 0.5 for middle gray. We apply this technique to all the rest 783 variables. To say further we will have an input neuron for each input variable in the tensor.

To solve the problem, we need to recognize which digit is drawn on image. For example, we can have only 10 possibilities, an image can have a digit between 0 and 9, inclusive. So, we make 10 output neurons, for each digit. After the model is trained network should output some values between 0 and 1, and the one neuron that has the highest value, is winning. For instance, if output is [0.1,0.2,0.15,0.94,0.25,0.12,0.21,0.3,0.1,0.2], 4^{th}

value has the highest number, that means, that 4^{th} value wins, which corresponds to number 3. This technique is called “One-hot”. [5]

Machine learning problems consume big amounts of data. To be sure that we learned model would work out it is usually needed to collect not less than ten thousand of working samples for simple problems. But more complex ones require even more. For example, accomplished task by DeepDrive, which aim was learning a model to drive simulated car, took a dataset of 600 thousand photos, which consumed 80 Gigabytes of memory size. To collect this amount of data researchers needed to record driven car for 42 hours. [6]

It is obvious that it is not possible to load such large datasets into RAM to process it. For such cases researchers divide data into batches. Each batch contain some amount of working samples, for example, 1000 images which are loaded to RAM. After batch is finished, next batch is loaded.

Working sample contain inputs and outputs that are fed into model to teach it. It can be image or text, or just a sequence of signals. The desired outputs have to be loaded in parallel, so that each sample should be fed and processed with the corresponding output.

2.8 **Activation Functions**

In computer systems activation functions used inside nodes. They define the output of node according to input. For example, standard computer chip can give 1 or 0 depending o input. This is very simple to linear activation function in Neural Networks. However, linear activation functions don’t let to solve non trivial problems in with small amount of neurons in a network. Nonlinear functions are the base that lets neural networks solve complex tasks.

In neural networks that designed to be biologically similar. Activation function represents potential of neuron to fire on certain conditions. Basic neuron can give its’ output as 1 or 0, but more sophisticated activation functions allow more complex outputs, like between 0 and 1, where output of 0.3 can mean that the neuron sure for 30%, that is probably not enough, but 0.9 can mean that the neuron is 90% sure. So, to get understanding on how differs functions we will review most used ones.

2.8.1 Identity

First of all, goes the simplest function which is linear, that is also called Identity. The main drawback of this kind of activation function is that if we apply two or more hidden layers, the entire network would be equivalent to a single layer model. Also this type of activator is unbounded, that creates a lack of normalisation, which is very harmful for computation process and for model itself.

To understand better, we will take a look at how one Identity neuron will separate the point in data space.

We will have a set of points marked as 1 and set of 0 marked points. After learning was applied we see the line separates sets.

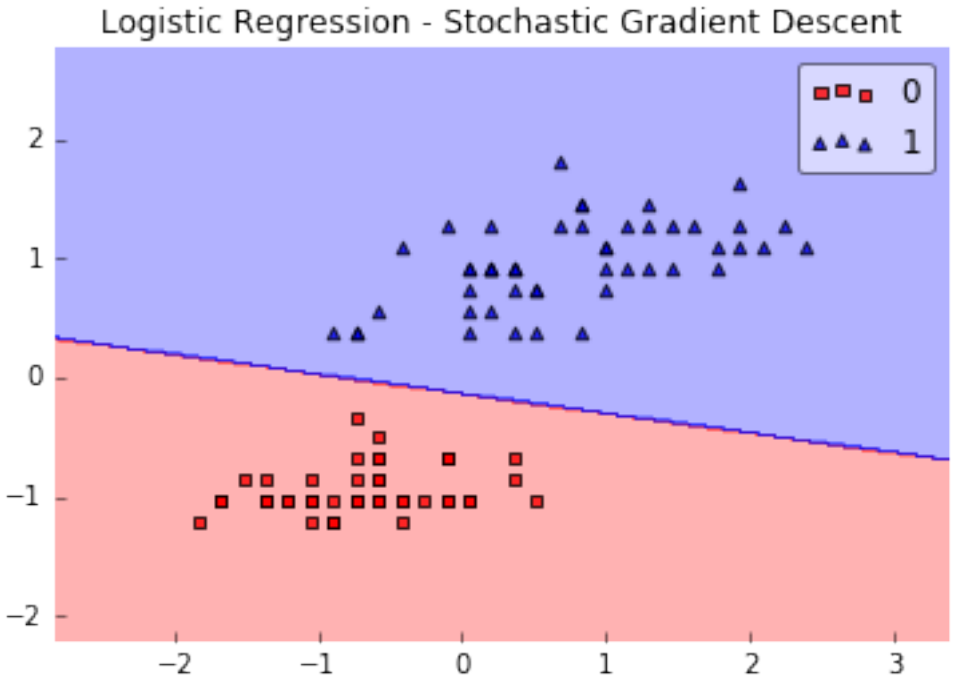

Figure 2: Logistic Regression – Stochastic Gradient Descent

The line separates points very well, unless we have a trickier task, like this:

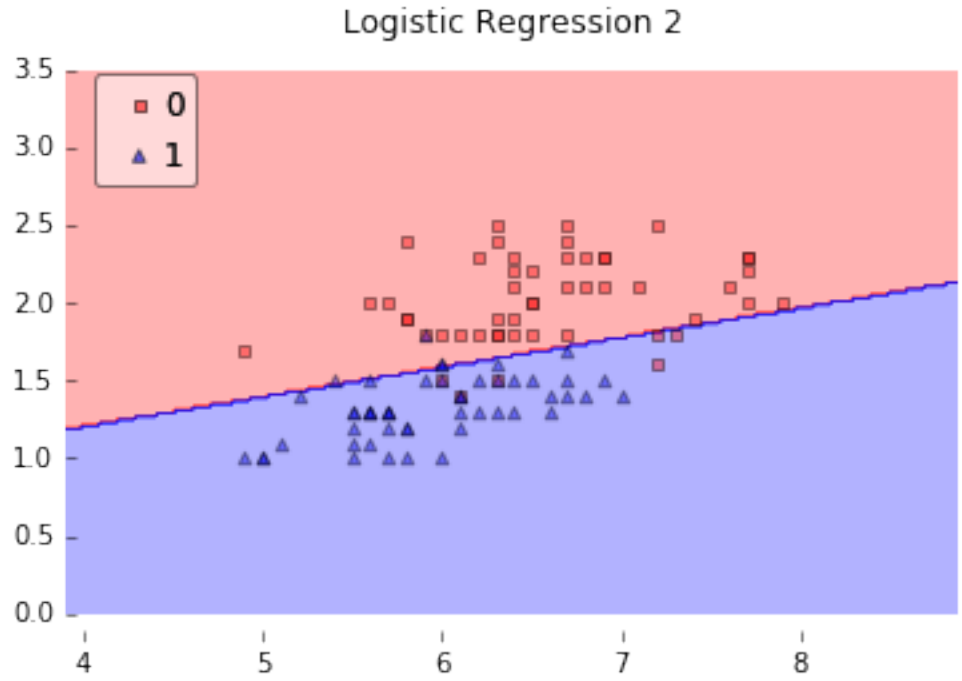

Figure 3: Logistic Regression

Here sets can't be divided by one line, so we need to apply some complex solution. Non-linear function would fit the problem. Multi-layer perceptron with one hidden layer of 200 neurons powered with sigmoid functions will solve it.

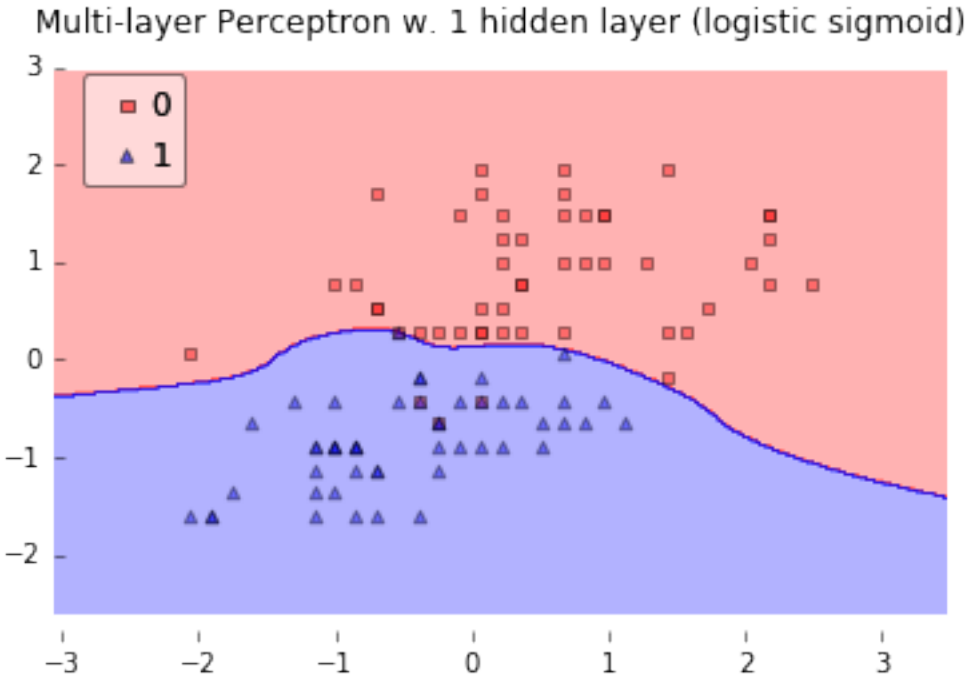

Figure 4: Multi-layer Perceptron, w. 1 hidden layer, 70 units

2.8.2 Sigmoid

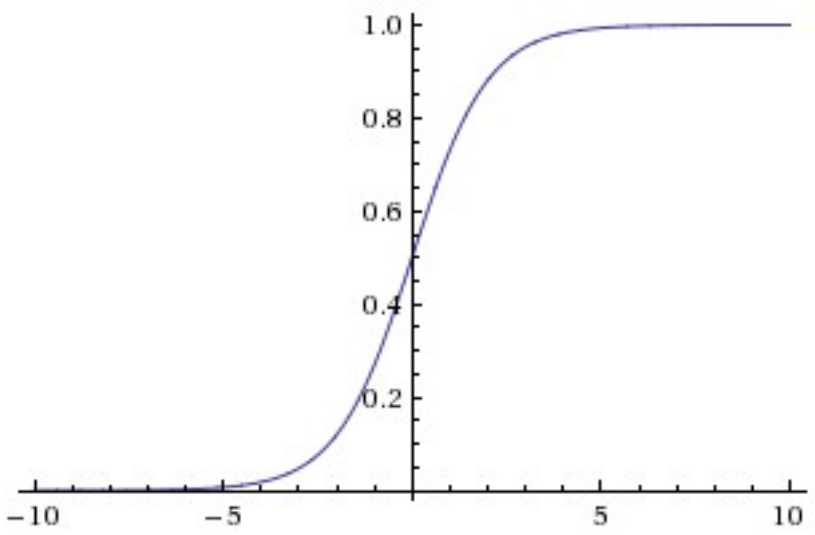

Figure 5: Sigmoid graphic

Sigmoid is a smooth monotonic non-linear function, that has “S” shape and always increase. It gained popularity in Neural Networks as an activation function due to ability increase low signals and not get over saturated from high signals. The function can input signals from -∞ to +∞ and outputs signal from 0 to 1, that makes it perfect normalising function. Derivative of sigmoid can be easily expressed through the function itself, that significantly simplifies the backpropagation process. [7]

On the other hand, saturation of sigmoid function is a drawback, because it kills gradients. Gradients that close to 0 or 1 are almost zero. Recall that during backpropagation, this (local) gradient will be multiplied to the gradient of this gate’s output for the whole objective. Therefore, if the local gradient is very small, it will effectively “kill” the gradient and almost no signal will flow through the neuron to its weights and recursively to its data. Additionally, one must pay extra caution when initializing the weights of sigmoid neurons to prevent saturation. For example, if the initial weights are too large then most neurons would become saturated and the network will barely learn. Or even it is possible that sigmoid give irrelevant results:

f’(a)=f(a) (1-f(a)), when a goes infinite large f(a) can become 1, so:

$$1*(1-1) = 0$$

Also, sigmoid outputs are not zero-centred. This is undesirable since neurons in later layers of processing in a Neural Network would be receiving data that is not zero-centered. This has implications on the dynamics during gradient descent, because if the data coming into a neuron is always positive, then the gradient on the weights will during backpropagation become either all be positive, or all negative. This could introduce undesirable zig-zagging dynamics in the gradient updates for

the weights. However, notice that once these gradients are added up across a batch of data the final update for the weights can have variable signs, somewhat mitigating this issue. Therefore, this is an inconvenience but it has less severe consequences compared to the saturated activation problem above.

2.8.3 Tanh

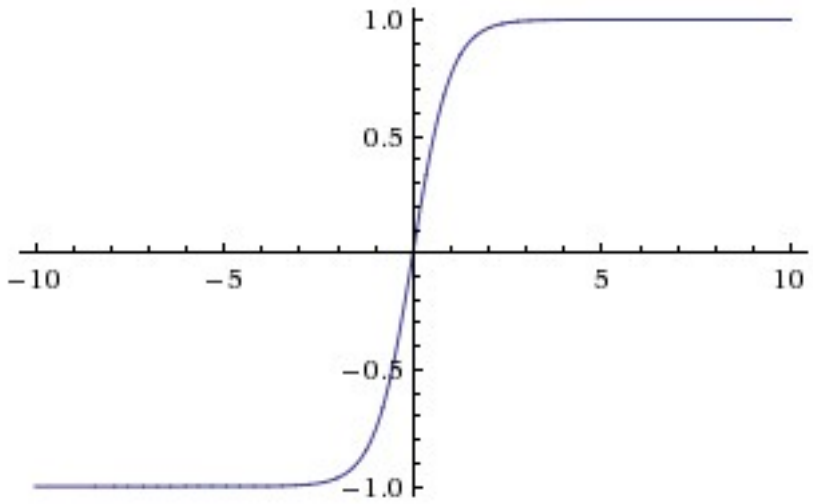

Figure 6: Tanh graphic

The tanh non-linearity is shown on the image above on the right. It squashes a real-valued number to the range [-1, 1]. Like the sigmoid neuron, its activations saturate, but unlike the sigmoid neuron its output is zero-centred.[7] Therefore, in practice the tanh non-linearity is always preferred to the sigmoid nonlinearity. Also note that the tanh neuron is simply a scaled sigmoid neuron, in particular the following holds:

$$\tanh(x) = 2\sigma(2x) - 1$$

2.8.4 ReLU

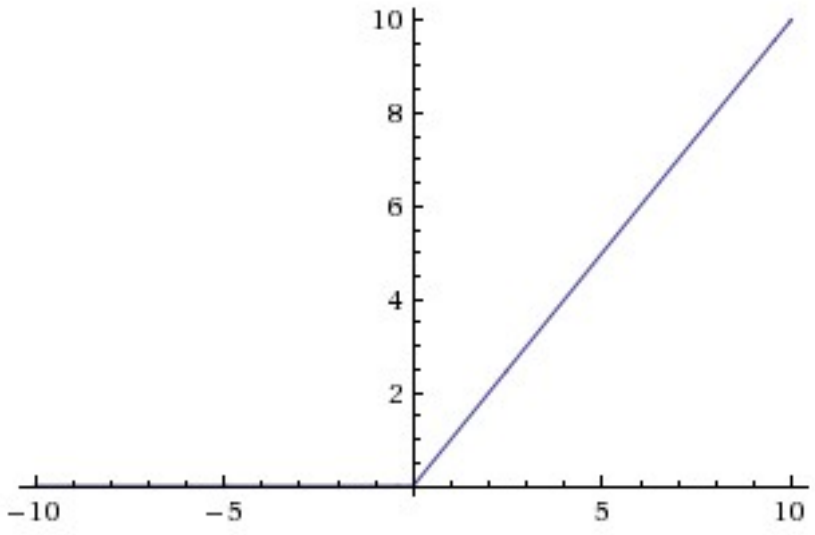

Figure 7: ReLU graphic

The Rectified Linear Unit has become very popular in the last few years. It computes the function f(x)=max(0,x). In other words, the activation is simply thresholded at zero. It was found to greatly accelerate the convergence of stochastic gradient descent compared to the sigmoid/tanh functions. It is argued that this is due to its linear, non-saturating form. Compared to tanh/sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero. [7]

On the other hand, ReLU can die. For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any data point again. If this happens, then the gradient flowing through the unit will forever be zero from that point on. That is, the ReLU units can irreversibly die during training since they can get knocked off the data manifold. For example, you may find that as much as 40% of your network can be “dead” (i.e. neurons that never activate across the entire training dataset) if the learning rate is set too high. With a proper setting of the learning rate this is less frequently an issue.

Leaky ReLU can solve this problem by changing the threshold from 0 to 0.01.

2.9 **Types**

2.9.1 Feed forward Network

Since the first single layer perceptron was created, scientists were trying to solve different types of problems with it. Unfortunately, not all the problems can be solved with feedforward networks. There were created several types of Neural Networks with different structures that solve problems of different kinds.

To say more about types of tasks that can be solved with neural networks mostly its problems with predictions. If we need to predict some kind of event in future if we have some information about the past, we will use Neural Networks. But it is not the only case we use them. Using Neural Networks, we can classify given information, forecast, compress data, make associative memory, etc.

There are three main types of Networks: Feedforward Networks, Recurrent Networks and self-organising maps. Also there is a network called Radial Basis Function Network, but it didn't get popularity.

ANNs allow signals to travel one way only: from input to output. There are no feedback (loops); *i.e.*, the output of any layer does not affect that same layer. Feed-forward ANNs tend to be straightforward networks that associate inputs with outputs. They are extensively used in pattern recognition. This type of organisation is also referred to as bottom-up or top-down.

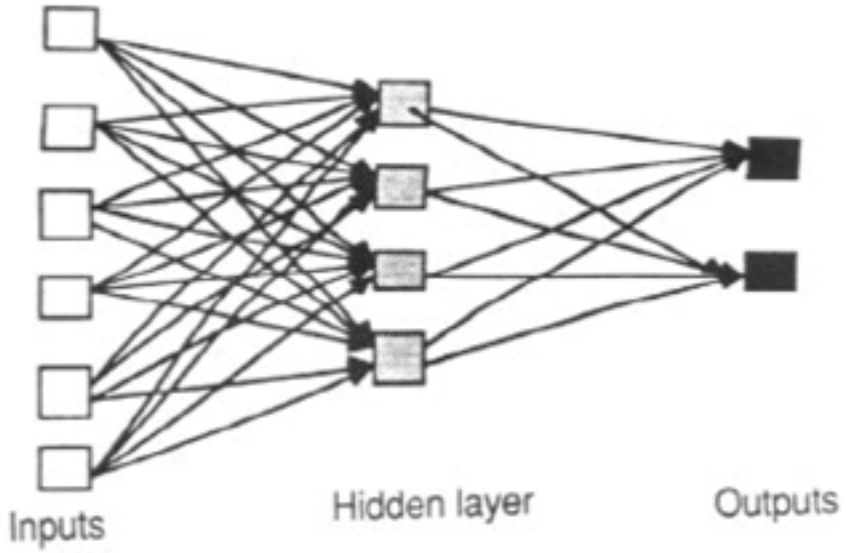

Figure 8: Multi-layer perceptron

The layers' structure can vary to be extremely large and complicated. For example, to a problem of image recognition was invented a structure called convolutional neural network, where each image gets divided into blocks of pixels for many times. If picture has resolution 4 x 4 pixels it can

be split in 9 blocks of 2 x 2 pixels for further processing.

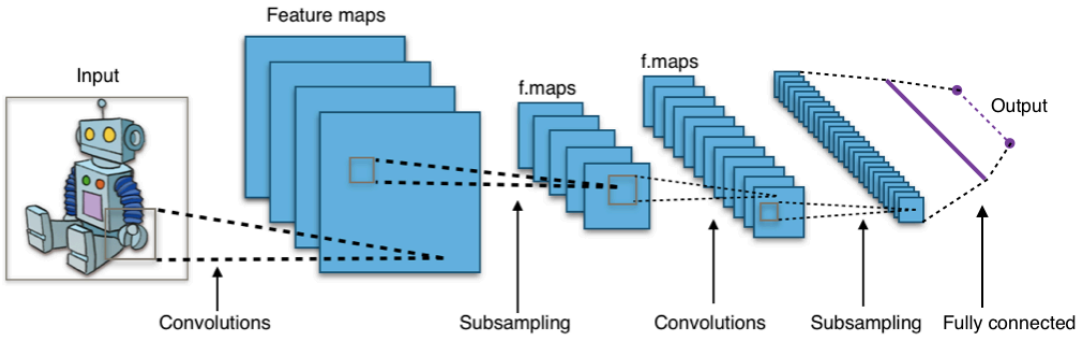

Figure 9: Convolutional Neural Nework

2.9.2 Recurrent Network

Recurrent networks can have signals traveling in both directions by introducing loops in the network. Feedback networks are powerful and can get extremely complicated. Computations derived from earlier input are fed back into the network, which gives them a kind of memory. Feedback networks are dynamic; their 'state' is changing continuously until they reach an equilibrium point. They remain at the equilibrium point until the input changes and a new equilibrium needs to be found.

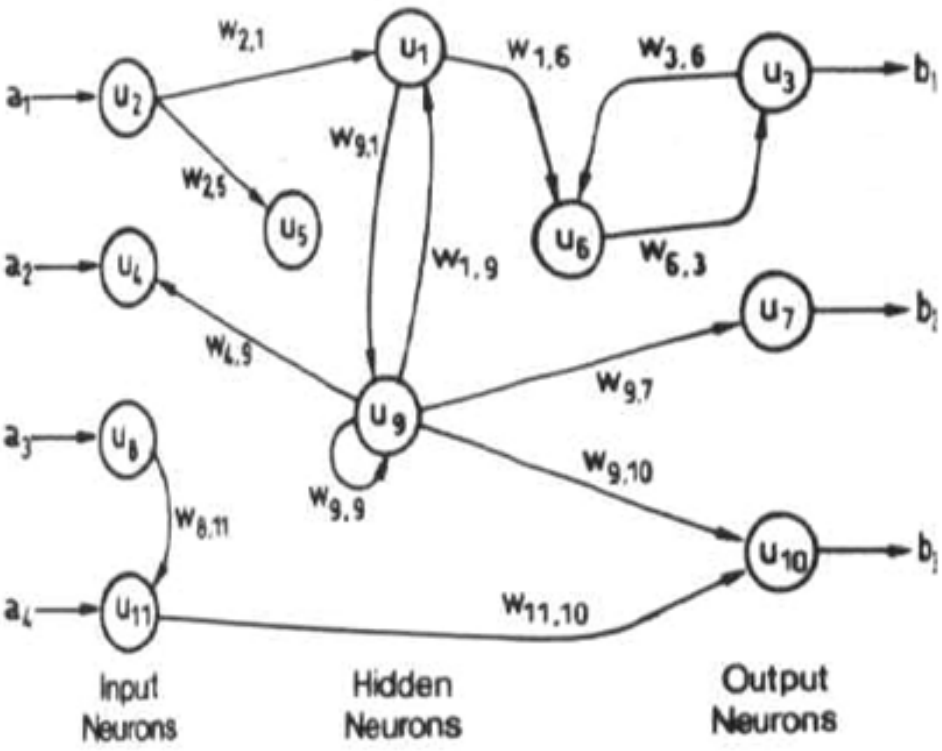

Figure 10.1: Recurrent Network

Feedforward neural networks are ideally suitable for modelling relationships between a set of predictor or input variables and one or

more response or output variables. In other words, they are appropriate for any functional mapping problem where we want to know how a number of input variables affect the output variable. The multilayer feedforward neural networks, also called multi-layer perceptron (MLP), are the most widely studied and used neural network model in practice.

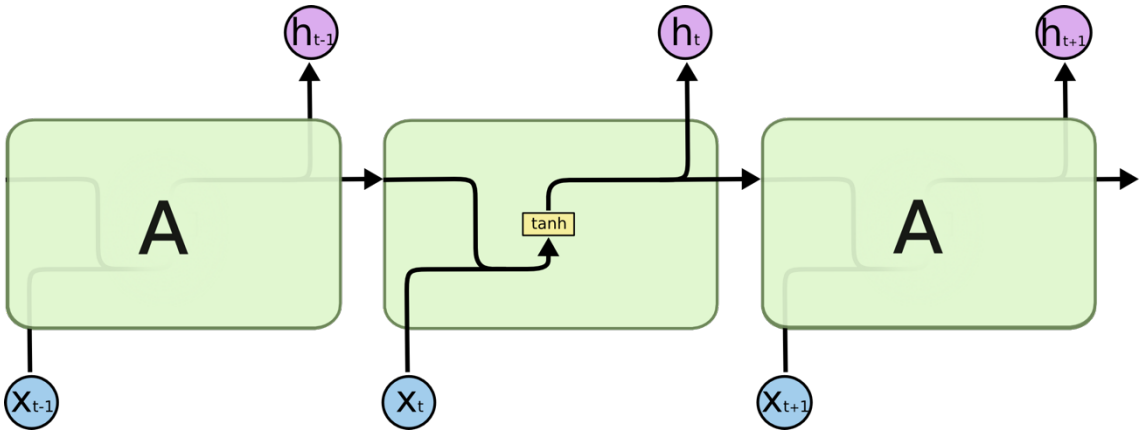

Figure 10.2 : Recurrent network

As an example of feedback network, I can recall Hopfield's network. The main use of Hopfield's network is as associative memory. An associative memory is a device which accepts an input pattern and generates an output as the stored pattern which is most closely associated with the input. The function of the associate memory is to recall the corresponding stored pattern, and then produce a clear version of the pattern at the output. Hopfield networks are typically used for those problems with binary pattern vectors and the input pattern may be a noisy version of one of the stored patterns. In the Hopfield network, the stored patterns are encoded as the weights of the network.

2.9.3 LSTM

LSTM networks are one of the most interesting types of networks and they are quite new comparing to other types. They were invented as a replacement of normal recurrent networks due to some reasonable advantages. Normal recurrent networks were a kind of breakthrough. However, they also had drawbacks. One of the most significant disadvantages was the problem of vanishing (or exploding) gradients. The main idea is that the recurrent networks work less efficiently with bigger amounts of steps remembered by network.

As a solution for this problem was invented Long Short-Term Memory unit. In this model normal neurons are replaced with special units that have quite complicated structure comparing with normal neural networks. LSTM units help to keep an error that can be backpropagated through layers and time. Unlike normal recurrent networks, that would

be hard to calculate at 100 step, LSTM can remember over 1000 steps.

LSTMs contain information outside the normal flow of the recurrent network in a gated cell. To data could be applied all types of operations, like, writing or reading. Inside the cell there is a mechanism, that when receives the information can decide what to do with it. It decides whether it is necessary to store the information or it can be deleted. To make such decisions cell gates learn on practice, which operations in which conditions brought more precise predictions in the past. This learning process also covered with the same mechanism of backpropagation, mainly this is possible due to universality of the backpropagation algorithm. [8]

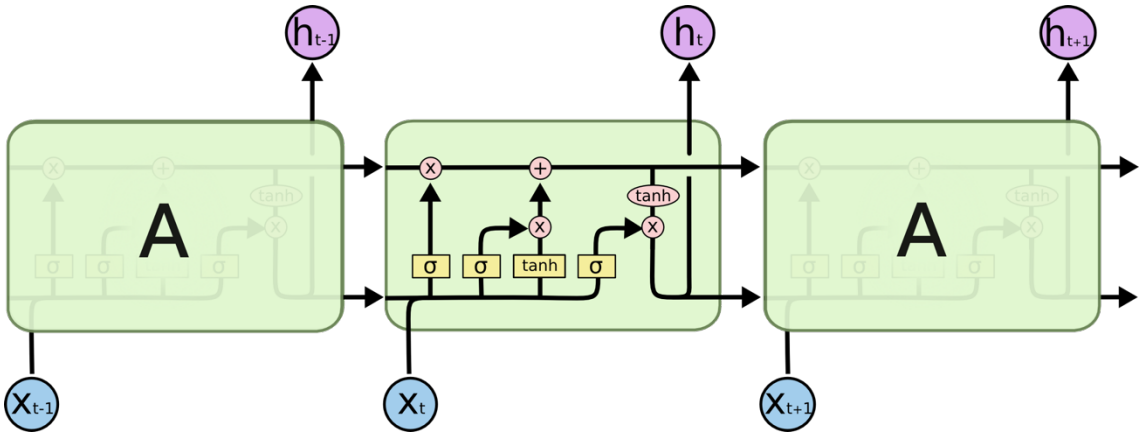

Figure 11: LSTM cell

2.9.4 Kohonen's self-organizing maps

Kohonen's self-organizing maps (SOM) represent another neural network type that is markedly different from the feedforward multilayer networks. Unlike training in the feedforward MLP, the SOM training or learning is often called unsupervised because there are no known target outputs associated with each input pattern in SOM and during the training process, the SOM processes the input patterns and learns to cluster or segment the data through adjustment of weights (that makes it an important neural network model for dimension reduction and data clustering). A two-dimensional map is typically created in such a way that the orders of the interrelationships among inputs are preserved. The number and composition of clusters can be visually determined based on the output distribution generated by the training process. With only input variables in the training sample, SOM aims to learn or discover the underlying structure of the data.

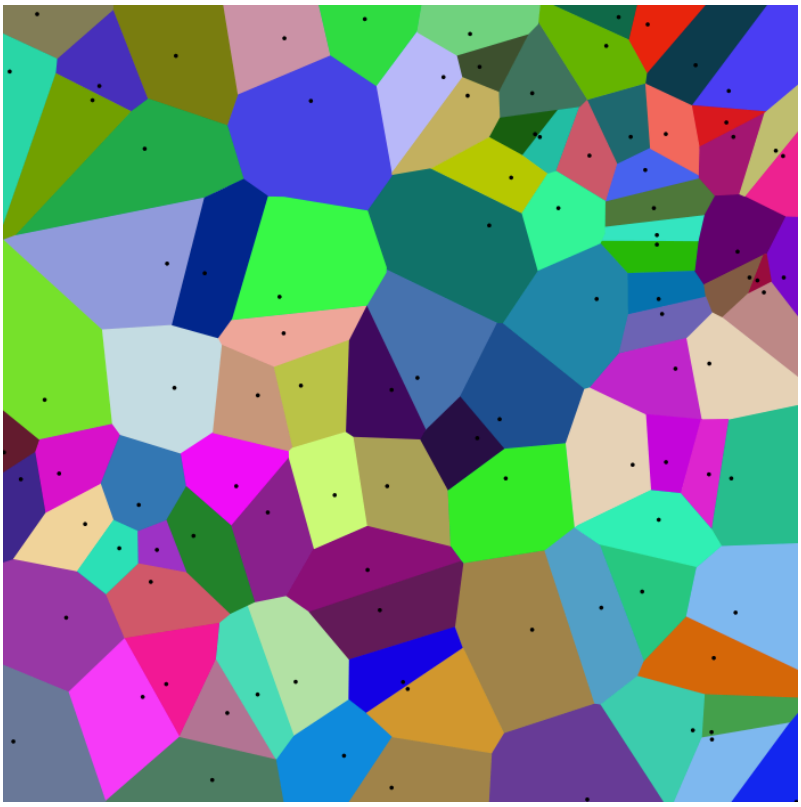

Figure 12: Kohonen's SOM

These Neural network are the basic ones. On the top of them people construct more sophisticated networks for more tight type of problems. To deal with the topic we don't need to know some sort of specific network, we need to succeed working with one of those are mentioned. At some point we may understand that we may need some sort of specific structure, but it would be a matter of an experiment, because most of the networks are somehow unique. Moreover, everyday we people invent new structures working on their goals.

2.10 **Learning procedure of feed-forward network**

To understand how neural networks learn we will take an example of Andrew Trask. The goal of the experiment would be teaching computer statistical to recognise inputs and predict outcome. To do so we will feed to the network several combinations of 3 input digits where each can be 1 or 0, and one output digit 1 or 0. Reviewing the simplest process of learning of Neural Networks we are going to write a code in Python programming language, which is quite simple to understand and an effective tool for prototyping. The model we want to build is two layer-perceptron. To multiply and process matrixes we will use NumPy library for Python. We will use sigmoid as activation function. Back-propagation will cover the learning process. [9]

First of all, we need to define Sigmoid function. Sigmoid is defined as formula

$$S(t)= 1/(1+e^{-t})$$

For back propagation algorithm we need to define the derivative of sigmoid function. As it was said already simple sigmoid derivative is one of main reasons of sigmoid popularity. The derivative can be written as

$$S'(t)=t*(1-t)$$

It is obvious that derivative of sigmoid is much easier to calculate than the original function.

Now we ant to write the sigmoid as a Python function that can be used in program. We will do it in a way that one function will be used to calculate sigmoid and derivative of it. To calculate sigmoid will be used such form: nonlin(x), where x is a variable needed to pass through sigmoid. If we need to know the outcome of derivative we request

```
nonlin(x, deriv=True)
def nonlin(x,deriv=False):
        if(deriv==True):
                return x*(1-x)
        return 1/(1+np.exp(-x))
```

Next we are going to define inputs and outputs. As it was mentioned before the input data should be normalised. But, our task does not require digits more than one and less than zero.

For our task we need 3 input nodes, 1 output node and 4 hidden neurons. We are going to feed too small batch of information, but we aim to feed it a couple of thousands times. The input batch will look like a massive of 3 columns and 4 rows. 3 columns mean that in first row we have 3 digits that are 3 inputs. We will have 4 teaching examples, so that we have 4 rows. Output will look like an array of 1 column and 4 rows: one output for each of four examples. X would be the inputs and y outputs.

```
X = np.array([[0,0,1],
                [0,1,1],
                [1,0,1],
                [1,1,1]])

y = np.array([[0],
                [1],
                [1],
                [0]])
```

Between 3 layers we will have 2 sets of weight connections. First it is necessary to initialise them with random values to break the symmetry and prevent equal weights after learning. The weights matrixes we will call synapses. To create two synapses, we will run the following code:

```
syn0 = 2*np.random.random((3,4)) - 1
syn1 = 2*np.random.random((4,1)) - 1
```

After that preparations are ready, now we start a loop of 60 thousand repetitions, all the code after that will be inside of a loop and will run along the program works

```
for j in xrange(60000):
```

First of all we need to feed forward our network with given sequences. Layer 0 will input the feed batch matrix 3 by 4. The second layer marked as l1 will be a result of matrix multiplication of layer 0 with weights of the first synapse and application of sigmoid function to that. The process of calculation of the last layer is similar but works with the second layer and the second synapse.

```
l0 = X
l1 = nonlin(np.dot(l0,syn0))
l2 = nonlin(np.dot(l1,syn1))
```

Since the output layer values now changed and have 1;4 dimensions, we want to check how much our results are different from what we really want. In Python language it is very easy to work with matrixes, so we just subtract one from another one.

```
l2_error = y - l2
```

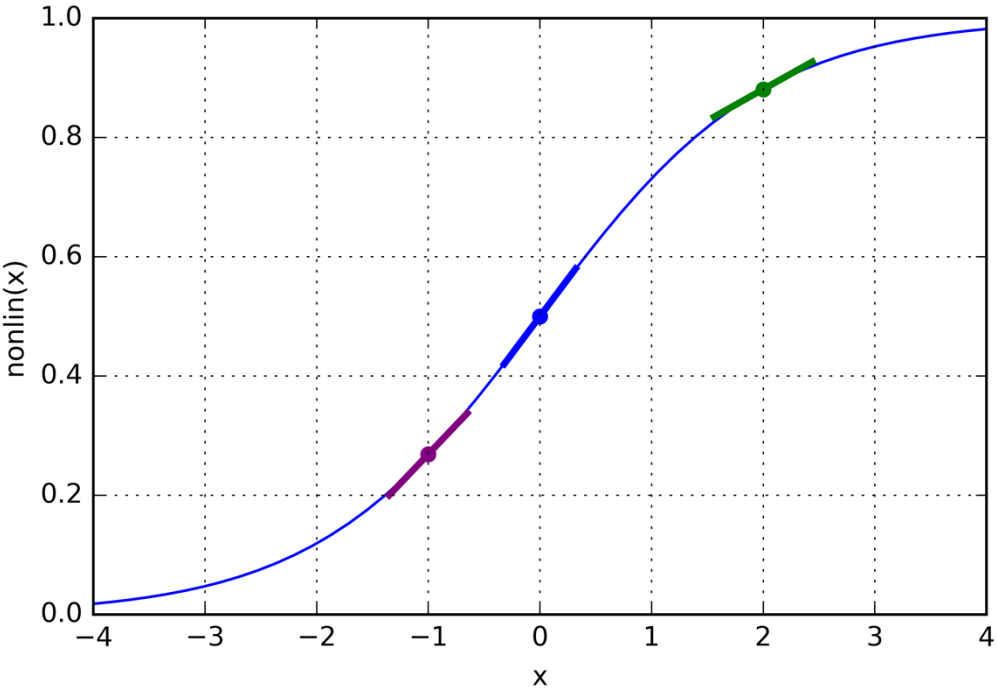

Figure 13: Derivatives for sigmoid function

By the end of the loop cycle we need to edit weights by some value, so the next time they will make more accurate predictions. But first of all we need to find a value to change the existing weights. For that we will use a method called The Error Weighted Derivative. We will apply this method to the first and the second layers. The idea of this technique is very simple. If we get a derivative of a value, we get an angle of a slope. After that, multiplying “slope” by the angle, we can reduce the error of highly confidential predictions.

For example, if the the slope is parallel to abscissa, the network either have very high value or very low. This means that the network is quite sure that in positive or negative way. On the other hand, if the angle is quite steep, the value is near to x=0, y= 0.5, for instance, this means that the network is not confident at all. So, we need to force it to make a choice to more confident decisions.

```
l2_delta = l2_error*nonlin(l2,deriv=True)
```

After that we calculate the error for the first layer. To do so we use "confidence weighted error" from l2 to establish an error for l1. It simply sends the error across the weights from l2 to l1. This gives what you could call a "contribution weighted error" because we learn how much each node value in l1 "contributed" to the error in l2.

```
l1_error = l2_delta.dot(syn1.T)
```

Next we are getting delta for layer 1 as we did for the layer 2.

```
l1_delta = l1_error * nonlin(l1,deriv=True)
```

In the end we add the values we calculated to synapses 0 and 1 just multiplying layer and corresponding delta.

```
syn1 += l1.T.dot(l2_delta)
syn0 += l0.T.dot(l1_delta)
```

This is a good example of simple feed forward perceptron with backpropagation learning algorithm.

2.11 **Commonly used frameworks**

Neural networks are quite complicated tools to use. They require understanding lots of mathematics and logics. Also, it is necessary to know programming to maintain in computing. Basic high level languages like Python or Pascal can be appropriate only in studying stage, if the user of neural networks need to build his own network and program it, it is very important to know fast low-level language, like C, C++ or GO at least. Also, some knowledge about Unix-like systems and possibilities to parallelize computing tasks would definitely help the researcher to solve his tasks in deadline.

The list of requirements to user is quite big, and that can keep out lots of researchers from tasks of machine learning and deep learning in particular. Also, writing new code to experiment with neural networks would take lots of time. Researchers started to build frameworks and open-source them to populate this field of studies and also, to get support of researchers from all over the world.

Basically most of the frameworks are build in high-level languages as front-end to give more flexibility and support access to GPU parallelism, which is the most efficient way calculating matrixes these days and give from 10 to 40 times increase in learning rate of neural network. [10] The front-end language is used for prototyping and it is usually Python, due to its popularity, flexibility and simplicity. C++ is the back-end language in the most popular frameworks. The compilation process takes very short time and the program build with it runs very fast.

3 **MATERIALS AND METHODS**

The task of auto piloting with neural networks is quite popular these days, mostly it is used only in terms of driver-less cars. Building pilot-less plane is much more complicated and responsible task. Car autopilot depends more on picture recognition tasks, while plane task requires more data from sensors and not deal with image processing at all, except tasks related to landing and taking off, where pilot actively observe current state of environment. Between landing and taking off pilot is guided only by signals of sensors like GPS, heading, altitude indicators.

Since plane automation development is quite time and resource consuming task, the best idea for developing auto pilot is to use plane simulator.

3.1 **Plane stabling task, basic concept**

Stable flight is one of the basic routine of a flight. We can divide the simplified flight algorithm into 5 parts: taking off, alignment to the course, keeping the course, preparing to landing, landing itself. To keep the course it is needed to stay stable during the flight. This can be achieved by controlling simple set of variables. To set our task more obvious to solve, lets define its goal as a stable flight, opposite to crashing. We will not hardly focus on the way the plane goes, but at the same time we do not need the plane to fly by rounds.

To keep the plane flying we request such variables as: 3-axis acceleration, altitude, pitch, roll, turn rate, airspeed. This set of variables can be assumed as an environmental feedback on our actions. Also, it would be a good practice to monitor the current states of flight control systems: throttling, aileron, elevator and rudder.

The first set of variables we can call outputs and the second in this case will be inputs. Mainly, we are considering the environment as a dynamic system that responds us with its state change according to our actions. If we change some output like we will turn right with elevator, the system will respond us with decreasing height and increasing turn rate. This concept can be called Proactive: we do something and environment respond us with something else. This concept could be helpful if we have a few options and we can predict the future with them, and choose the best option for us. But, on the other hand if we have too many options calculating each one in bulk will consume time and resources greater than allowed.

In this case we have to change input and output in places. Now it will be a Reactive approach. We will input current environment states and produce output as a set of changes we need to apply to aileron, elevator or rudder.

So, now, the input would be environmental states and the output will be the change needed to apply to control systems.

3.2 **Plane stabling task in details**

As it is stated already we are going to take several variables from Flight Gear to develop our piloting system. Stepping forward we will take all somehow relevant variables from Flight Gear. It is not necessary, but neural networks tend to pull out all information from inputs and independently decide which information is relevant and which is not. If the input information not affects the outputs, it will not screw up our results, furthermore it will be possible to define which data is irrelevant on the later steps and it can be retrained without useless inputs. So the full list of variables we need is:

1. 3-axis acceleration
2. Altitude
3. Pitch
4. Roll
5. Aircraft speed relatively to the ground
6. Turn rate
7. Vertical speed indicator
8. Airspeed
9. Throttle
10. Aileron
11. Elevator
12. Rudder

To keep stable fly it is needed to keep all the variables from 1 to 8 stay the same changing variables from 9 to 12.

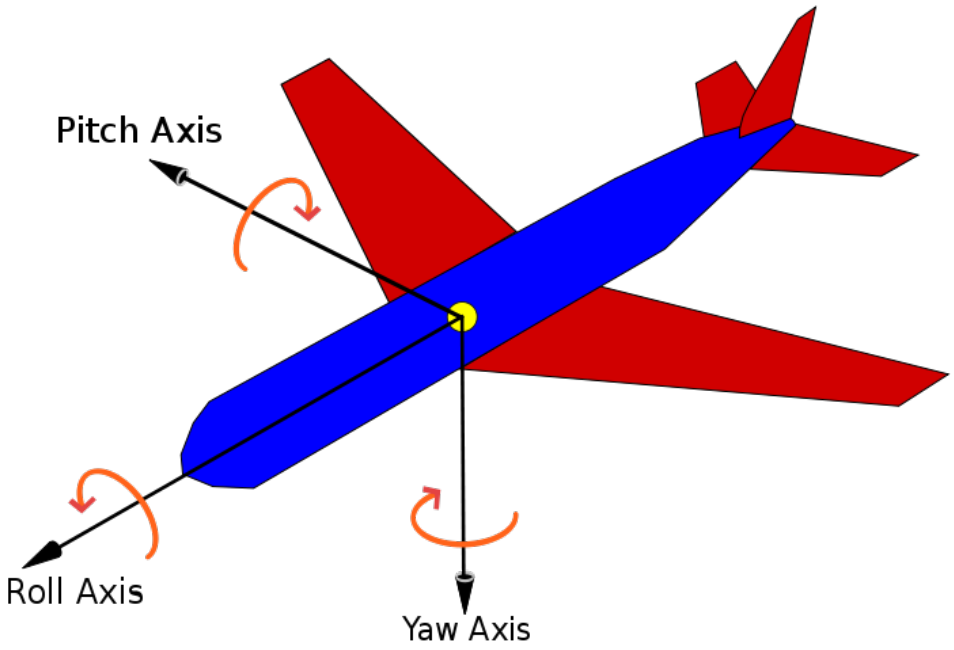

Figure 14: Pitch, roll and yaw Axis

The main variables here are Pitch and Roll they control that plane is parallel to the ground. If the plane is stable in Pitch, Roll, Height and Turn rate, that means that it moves straight. Airspeed determines the speed of aircraft relatively to the air around. Wind presence also affect this indicator. Since wind affects moving aircraft very high, pitch and roll may be different with the same throttle, aileron, elevator and rudder, for such a reason we need to have to know as much as possible about wind affect on the aircraft. For such reason we need additional variables like 3-axis acceleration, ground and vertical speed. These additional information gives enough representation of what is going on around the plane.

3.3 Plane simulation

Flight Gear simulator was chosen for development. It is open-sourced and has quite large developer community.

For the task of stabling the plane we need several properties of the current state of a plane, that come directly from plane sensor indicators. All the variables described in 3.2 can be extracted from the program.

Flight Gear have convenient mechanism of extracting and importing variables, that lets developers make their control systems outside of the program on the fly. All the variables that can be extracted and changed are

stored in a module called Property-tree. The access to the property tree can be obtained through “native” and “generic” protocols. First one gives access to the property tree from memory, that can be captured from sided C++ program. The second one opens access through UDP protocol or over Serial port.

Figure 15: Property-tree in Flight Gear

Using UDP we can run a server on the host machine and the controlling machine can capture the information if it is in local network, or even outside of it. Serial port can be used to connect low-level devices like Arduino, RaspberryPi or other devices that have GPIO extension slot on the board.

To use these options, it is necessary to create a XML file with a very simple structure.

```
<?xml version="1.0"?>

<PropertyList>

 <generic>

  <output>
   <line_separator>newline</line_separator>
   <var_separator>,</var_separator>
```

```
  <chunk>
   <name>pilot-gdamped</name>
   <type>float</type>
   <node>/accelerations/pilot-gdamped</node>
   <format>%f</format>
  </chunk>

                …

  <chunk>
   <name>airspeed-kt</name>
   <type>float</type>
   <node>/velocities/airspeed-kt</node>
   <format>%f</format>
  </chunk>

</output>
<input>
  <chunk>
   <name>throttle</name>
   <type>float</type>
   <node/controls/engines/engine[0]/throttle</node>
   <format>%f</format>
  </chunk>
</input>

 </generic>

</PropertyList>
```

It is easy to understand a meaning of each line in the code. The code is divided into two parts: input and output. Input section defines the input protocol and the output defines the output rules. The most interesting part is the “chunk” statement. Amount of chunks is correlated with amount of properties we are going to work with. Inside a chunk name can be random and doesn’t play role in the result. Type means the type of representation of a value. It can be float, integer or bool, depends on how we will use it. Inside the node is stored the path to a property inside of the Property tree. If the value is needed to be multiplied or converted the rule is defined in this section.

To run generic protocol, it is needed to run the Flight Gear program with flags

```
--generic=socket,out,10,127.0.0.1,49001,udp,outputprotocol
        --generic=socket,in,10,,49000,udp,inputprotocol
```

First flag is for exporting data from the program. First parameter is always “socket”. Second parameter defines whether the flag is for input or for output. On the 3rd place digit defines amount of packages with data would be sent or received in a second. Next goes IP address and port number of the machine that is going to receive the packages. Input protocol doesn’t need to have IP address. On the last place stands the name of the XML file containing the protocol instructions without “.xml” ending. The XML file should be placed into the /data/Protocol folder under the root of the Flight Gear.

The output of the file in Appendix 1 is looking like that:

0.910101,1.000,0.000,0.150,0.050,1148.926025,3.260596,-0.272000,28.521667,0.043057,4.652739,114.134407,56.893894

0.908808,1.000,0.000,0.150,0.050,1152.446167,3.008736,0.216146,28.492014,0.051938,4.457239,114.064537,56.915268

The Python program that received packages is under Appendix 2.

3.4 **Data processing for the ANN**

To process the data, we need to define the algorithm of the network.

For the task of stabling the plane in the air we are going to export some data from the simulation program, learn the Artificial Neural Network and after that deploy the learned model under extensive computer to control the plane by the learned algorithm.

To learn the model, we need to define the roles of each variables and normalize them. Some of the values, like turn rate and acceleration are already fine values to put in a neural network, except they are needed to put under -1;1 or 0;1 frame, so we will define the frames they are already exist and divide by the biggest value existing for that variable, of course without “-“. Values like Pitch, Roll, Aircraft speeds we are going to compare with values on the previous timestamp, calculate the delta for the values and, finally, normalize it, like we did before.

As soon as our elevator, aileron and rudder should be the reaction on the previously happened changes in other properties, we are going to calculate the delta for them comparing current and the next timestamp. So, as a result we have such dataset map.

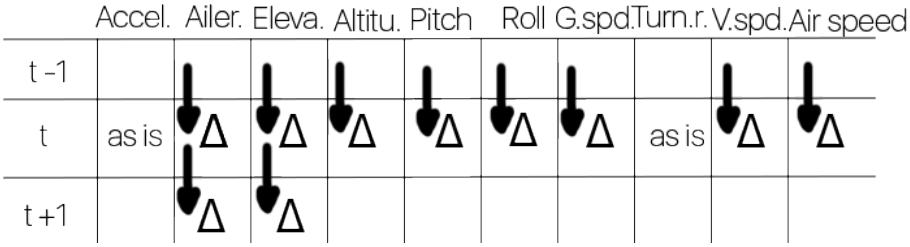

Figure 16: Timestamp dataset development

The timestamp dataset sample is represented as acceleration and turn rate as they are at current timestamp. Aileron and elevator can be represented as the difference between previous timestamp or also as they are at current. Altitude, pitch, roll, ground speed, vertical speed and air speed can be represented as difference between two timestamps, current and previous. To simplify learning process, we can get rid of throttle and rudder, so the difference between next time step and current for aileron and elevator can represent label samples that can be reactions for the situation in current timestamp.

The Python program-library that content functions for converting raw dataset into new dataset with data described above for Caffe framework can be found in Appendix 3.

3.5 **Jetson TX1**

Nvidia Jetson TX1 will fit the requirement for controlling computer. It has 256 GPU cores that are perfect for processing matrix and computing Artificial Neural Networks in particular. It runs special Linux distributive, modified Ubuntu. It supports CUDA and Cudnn libraries from the box. It has GPIO, that can be used for serial port connection to host computer. To connect host to Jetson needed to have USB-TTL cable. Arduino Nano can be used as a USB-TTL connector. It is needed to connect RST with GND pins on the Arduino, GND pin of Arduino should be connected to ground of Jetson, TX pin of the Arduino should be connected to RX of Jetson and RX of the Arduino to TX of Jetson. After that serial port will be open on 115200 baud rate.

3.6 **Artificial Neural Network**

By the moment of writing the thesis the most advanced, community supported and well documented ANN library is TensorFlow. Unfortunately, it is not working well on Jetson due to complexity of installation for that platform. Alternatively, Caffe can be used.

The network contains 3 files: dataset for learning, prototxt file describing the ANN model, prototxt file describing working process of learning, testing and deploying.

Because the flight control is not a single event process the best choice would be to use LSTM network, since it can remember long sequences of events. Unfortunately, by November Caffe framework community doesn't have documentation for using LSTM cells.

Alternatively, simple deep feed-forward network can be used. Appendix 4 and 5 contains the prototxt files for building deep feedforward network.

3.7 **Learning process**

After capturing the data started the process of learning can take up to several days. Gathering 4 samples a second will give 14.400 samples an hour. To make sure that the neural network will remember better it is necessary to take at least 100.000 samples, which will require more time. Moreover, the learning process will request maximum attention from a teacher along the way. In case LSTM cells were used, the network will copy all the manner the teacher piloting the plane.

4 **CONCLUSION**

The topic of artificial neural networks is around aircrafts for a while, but how already was mentioned it takes huge amount of work and lots of responsibilities to build up a project from scratch to real-life implementation on working prototypes.

It is absolutely necessary to continue the project in future since artificial neural networks are coming into our lives quite deeply, with increasing computing powers and development of Artificial Intelligence. Such systems but in more complex implementations will definitely conquer the market, not only the Aircrafts but also other controlling systems. The proof is Google implementations in car control systems, that already drive without driver on streets of California.

5 **FUTURE WORK**

The current implementation with LSTM cells seems to be very promising. But even now it requires some sort of teacher presented at all learning time. The possible and clever way would be automation of the process with Reinforcement learning technology, where program receiving some feedback from controlled system, so that it can learn not only what actions are good, but at the same time it can learn what is harmful for the system.

On the other way such way of learning can not be implemented in more expensive planes. It will require at least some people present on a plane to control the learning process from destroying facilities.

The other good way would be to start implementing only the learning systems with different specifications, for example, if every plane that performs flies these days will be equipped with such a system, in future we will have lots of data that can be used to teach such systems. After that implementation LSTM with reinforcement modules to pilot-less planes may force such systems to evolve.

REFERENCES

1. Stanford - Artificial Neuron. (n.d.). Retrieved November 27, 2016, from https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/Neuron/index.html
2. Karpathy, A., PhD. (n.d.). Hacker's guide to Neural Networks. Retrieved November 27, 2016, from http://karpathy.github.io/neuralnets/
3. Single-layer Neural Networks (Perceptrons). (n.d.). Retrieved November 27, 2016, from http://computing.dcu.ie/~humphrys/Notes/Neural/single.neural.html
4. How the backpropagation algorithm works, chapter 2. (n.d.). Retrieved November 27, 2016, from http://neuralnetworksanddeeplearning.com/chap2.html
5. MNIST For ML Beginners. (n.d.). Retrieved November 27, 2016, from https://www.tensorflow.org/versions/r0.7/tutorials/mnist/beginners/index.html
6. DeepDrive - self-driving car AI. (n.d.). Retrieved November 27, 2016, from http://deepdrive.io/
7. Convolutional Neural Networks for Visual Recognition. (n.d.). Retrieved November 27, 2016, from http://cs231n.github.io/neural-networks-1/
8. Convolutional Neural Networks for Visual Recognition. (n.d.). Retrieved November 27, 2016, from http://cs231n.github.io/neural-networks-1/
9. A Neural Network in 11 lines of Python (Part 1). (n.d.). Retrieved November 27, 2016, from http://iamtrask.github.io/2015/07/12/basic-python-network/
10. CUDA Spotlight: GPU-Accelerated Deep Neural Networks. (2015). Retrieved November 27, 2016, from https://devblogs.nvidia.com/parallelforall/cuda-spotlight-gpu-accelerated-deep-neural-networks/

Appendix 1

Contents of XML input/output file

```
<?xml version="1.0"?>

<PropertyList>

 <generic>

  <output>
   <line_separator>newline</line_separator>
   <var_separator>,</var_separator>

   <chunk>
    <name>pilot-gdamped</name>
    <type>float</type>
    <node>/accelerations/pilot-gdamped</node>
    <format>%f</format>
   </chunk>

   <chunk>
    <name>throttle</name>
    <type>float</type>
    <node>/controls/engines/engine[0]/throttle</node>
    <format>%f</format>
   </chunk>

   <chunk>
    <name>aileron</name>
    <type>float</type>
    <node>/controls/flight/aileron</node>
    <format>%f</format>
   </chunk>

   <chunk>
    <name>elevator</name>
    <type>float</type>
    <node>/controls/flight/elevator</node>
    <format>%f</format>
   </chunk>
```

```
<chunk>
 <name>rudder</name>
 <type>float</type>
 <node>/controls/flight/rudder</node>
 <format>%f</format>
</chunk>

<chunk>
 <name>indicated-altitude-ft</name>
 <type>float</type>
 <node>/instrumentation/altimeter/indicated-altitude-ft</node>
 <format>%f</format>
</chunk>

<chunk>
 <name>indicated-pitch-deg</name>
 <type>float</type>
 <node>/instrumentation/attitude-indicator/indicated-pitch-deg</node>
 <format>%f</format>
</chunk>

<chunk>
 <name>indicated-roll-deg</name>
 <type>float</type>
 <node>/instrumentation/attitude-indicator/indicated-roll-deg</node>
 <format>%f</format>
</chunk>

<chunk>
 <name>ns-velosity-msec</name>
 <type>float</type>
 <node>/instrumentation/gps/ns-velocity-msec</node>
 <format>%f</format>
</chunk>

<chunk>
 <name>indicated-turn-rate</name>
 <type>float</type>
 <node>/instrumentation/turn-indicator/indicated-turn-rate</node>
 <format>%f</format>
</chunk>

<chunk>
 <name>vertical-speed-indicator</name>
 <type>float</type>
 <node>/instrumentation/vertical-speed-indicator/indicated-speed-mps</node>
 <format>%f</format>
```

```
 </chunk>

 <chunk>
  <name>airspeed-kt</name>
  <type>float</type>
  <node>/velocities/airspeed-kt</node>
  <format>%f</format>
 </chunk>

 <chunk>
  <name>heading</name>
  <type>float</type>
  <node>/instrumentation/heading-indicator/indicated-heading-deg</node>
  <format>%f</format>
 </chunk>
</output>
<input>

 <chunk>
  <name>Inthrottle</name>
  <type>float</type>
  <node>/controls/engines/engine[0]/throttle</node>
  <format>%f</format>
 </chunk>

 <chunk>
  <name>Inaileron</name>
  <type>float</type>
  <node>/controls/flight/aileron</node>
  <format>%f</format>
 </chunk>

 <chunk>
  <name>Inelevator</name>
  <type>float</type>
  <node>/controls/flight/elevator</node>
  <format>%f</format>
 </chunk>

 <chunk>
  <name>Inrudder</name>
  <type>float</type>
  <node>/controls/flight/rudder</node>
  <format>%f</format>
 </chunk>

</input>
```

```
 </generic>

</PropertyList>
```

Appendix 2

Python program for receiving Flight Gear data

```
import socket

UDP_IP = "127.0.0.1"
UDP_PORT = 49001

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # UDP
sock.bind((UDP_IP, UDP_PORT))
counter=0
dataset=85
data=[]
dataToFlush=''
while True:
        dataInput, addr = sock.recvfrom(1024)
        dataRawList=str(dataInput).split(',')
        dataRawList[1]=dataRawList[1][:5]
        dataRawList[2]=dataRawList[2][:5]
        dataRawList[3]=dataRawList[3][:5]
        dataRawList[4]=dataRawList[4][:5]
        print('|ail:        ',dataRawList[2],'|el:        ',dataRawList[3],'|rud:
',dataRawList[4],'|hgt: ',dataRawList[5],'|head: ',dataRawList[12],'\n')
        dataToFlush+=','.join(dataRawList)+'\n'
        counter+=1
        if counter==240:
                dataset+=1
                counter=0
                with open('dataset'+str(dataset)+".txt",'w') as file:
                        file.write(dataToFlush)

                dataToFlush=''
```

Appendix 3

Contents of Python library for preparing data for Caffe

```
import numpy as np
import deepdish as dd


#lolnp=np.array(lol,dtype='float64')

def deleteCols(array,liOColl):
                liOColl.reverse()
                for i in liOColl:
                                array=np.delete(array,np.s_[i:i+1],1)
                return array


def delta(di,nu):
   prev=float(di[0][nu])
   max1=min1=float(di[1][nu])-float(di[0][nu])
   for i in di:
         cdel=float(i[nu])-prev
         if cdel>max1:
               max1=cdel
         if cdel<min1:
               min1=cdel
         prev=float(i[nu])
   return([min1,max1])

def minmax(di,nu):
   min1=float(di[0][nu])
   max1=float(di[0][nu])
   for i in di:
         if float(i[nu])<min1:
               min1=float(i[nu])
         if float(i[nu])>max1:
               max1=float(i[nu])
   return([min1,max1])



def getparams(indi):
                #out=np.zeros(len(indi)-2,len(indi[0]))
                out1data=np.zeros([1,10],dtype=float)
                out1label=np.zeros([1,2],dtype=float)
                for i in range(len(indi)-1):
                                if i == 0:
                                                continue
                                #tm1=indi[i-1]
```

```
                        #t=indi[i]
                        #tp1=indi[i+1]
                        curr=indi[i]
                        d1=indi[i]-indi[i-1]
                        d2=indi[i+1]-indi[i]

        single=np.array([[curr[0],curr[1],curr[2],d1[3],d1[4],d1[5],d1[6],curr[7],d1
[8],d1[9]]])
                        singlelabel=np.array([[d2[1],d2[2]]])
                        out1data=np.append(out1data,single,axis=0)
                        out1label=np.append(out1label,singlelabel,axis=0)

        return([np.delete(out1data,np.s_[0:1],0),np.delete(out1label,np.s_[0:1],0
)])

def divideAlongAxis(array,column,num):
        for i in array:
                i[column]=i[column]/num
        return array

def minMaxColumn(array,column):
        return ([np.min(array[:,column]),np.max(array[:,column])])

def multiDivideAlongAxis(array,column,num):
        newarray=array
        for n in range(len(column)):
                newarray=divideAlongAxis(newarray,column[n],num[n])
        return newarray

def makeh5(data,labels):
        if len(data)==len(labels):
                X=np.zeros((len(data),1,1,len(data[0])))
                y=np.zeros((len(labels),len(labels[0])))
                for i in range(len(data)):
                        for l in range(len(data[0])):
                                X[i][0][0][l]=data[i][l]
                        for l in range(len(labels[0])):
                                y[i][l]=labels[i][l]
        print('Done creating')
        dd.io.save('test.h5', {'data': X, 'label': y}, compression=None)
        print('Done writing')
        return([X,y])

def makeh5_2d(data,labels):
        X=data
        y=labels
        dd.io.save('2ddata.h5', {'data': X, 'label': y}, compression=None)
        return 0
```

Appendix 4

Prototxt file containing description of network for Caffe

```
name: "DeepNet"
layer {
  name: "input"
  type: "HDF5Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  hdf5_data_param {
    source: "way/to/dataset.h5"
    batch_size: 64
  }
}


layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "data"
  top: "ip"
  inner_product_param {
    num_output: 10
  }
}

layer {
  name: "Sigmoid1"
  type: "Sigmoid"
  bottom: "ip1"
  top: "Sigmoid1"
}



layer {
  name: "ip2"
  type: "InnerProduct"
  bottom: "Sigmoid1"
  top: "ip2"
  inner_product_param {
    num_output: 20
  }
```

```
}

layer {
  name: "Sigmoid2"
  type: "Sigmoid"
  bottom: "ip2"
  top: "Sigmoid1"
}

layer {
  name: "ip3"
  type: "InnerProduct"
  bottom: "Sigmoid2"
  top: "ip3"
  inner_product_param {
    num_output: 10
  }
}

layer {
  name: "Sigmoid3"
  type: "Sigmoid"
  bottom: "ip3"
  top: "Sigmoid1"
}

layer {
  name: "loss"
  type: "SigmoidCrossEntropyLoss" # "SoftmaxWithLoss"
  bottom: "Sigmoid3"
  bottom: "label"
  top: "loss"
}
```

Appendix 5

Prototxt file containing description of solver for Caffe

```
# reduce learning rate after 120 epochs (60000 iters) by factor 0f 10
# then another factor of 10 after 10 more epochs (5000 iters)

# The train/test net protocol buffer definition
net: "way/to/model.prototxt"
# test_iter specifies how many forward passes the test should carry out.
test_iter: 10
# Carry out testing every 1000 training iterations.
test_interval: 1000
# The base learning rate, momentum and the weight decay of the network.
```

```
base_lr: 0.001
momentum: 0.9
#weight_decay: 0.004
# The learning rate policy
lr_policy: "step"
gamma: 1
stepsize: 5000
# Display every 100 iterations
display: 100
# The maximum number of iterations
max_iter: 60000
# snapshot intermediate results
snapshot: 10000
snapshot_prefix: "examples/example_sigmoid"
# solver mode: CPU or GPU
solver_mode: GPU
```