Deeplearning-for-beginners

July 23, 2018

1 Deeplearning for beginners

Hi, with this notebook I will getting you started with Deeplearning in Python3. We will build our own neural net from scratch and learn how to use tflearn. Also we want to cover diffrent types of neural nets and more. Prequisites: 1. Python3 knowledge (go on youtube and look for a python3 getting started or a youtuber called sendtex, he has got a starter series on his channel) 2. Algebra knowledge (if you want to learn or refresh it I reccomend you the Khan academy; https://www.khanacademy.org/)

If you find spelling/writing mistakes please report them to me so I can fix them.
Note: it's possible that these will not work when you read it because of new updates!

---

2 Usage of machine/deeplearning

So as you may know machine learning is implemented in our everyday life, so you have propably used it at least once. You use it in photo editing, in apps(Snapchat's filters for example, that makes you look like a dog or whatever else), in your Google search (the recommendations that google is showing you) or in advertisment (like Amazon:"The people who bought this product also bought..."). So as you see Machine Learning is everywere and why don't use it aswell. Or in image classification (this is a dog and this and this not...) and more.

Before we continue I am not a biologist and not a AI expert and it's possible that there is something wrong, so if you find something please inform me!

3 The general idea

So the examples above sounds nice but how do they work... Let's find it out: So as we humans often do, we looked around in nature and find a powerfull "device" to work with data. It's called "Brain"; so the brain is learning by exprience. In detail our brain is working with neurons (so were do you think the name: "Neural Network" is from?), a neuron is a nerv that taking an input and returning an output if a certain point is reached (the neuron is "fireing"). The neuron sends these output to another neurons with a nerve called "synapse" if a neuron is using one synapes more then others these will became stronger.(You have to know that not only a single neuron is for one task responsible, so they are working together) If these getting stronger you can better do things like distinguish cats and dogs or drive a car. An artifical neural network is doning the same with math. A synapse can in- or decrease an incomming signal so sometimes if the input signal is to

low to start an action a synapse can increase the signal so the neuron will fire, but it's possible that it decrease and cause the opposite effect.

4 An artificial neural net

So here is an image of an artificial neural net: As you can see there are 4 input layer 5 hidden layer and one output layer. Each layer is connected with a line, these line is the same as the synapse but in this image you can't see weights (how strong the synapse is evolved). So each "bubble" represents a neuron and each layer is (surprise) a layer ;) . They are varius types of neural nets for varius type of work outside.Let's have a short look on this to grafics to get a better understanding of the varius types. So don't be afraid we will cover some of the most importent ones and then you can understand the rest by yourselfe.

4.1 # Common types of neural networks

So let's start with the most common types of nets.

1. The most common one I guess is the feed forward neural network (I will say nn for neural network). There every data flow from the input layer to the output layer without turning backwards. They are multi or single layer ones for diffrent tasks. They will often be used for: if no feedback loop is needed (as opposite see the RNN), pattern recognition. Often used with linear functions but not only with them. If you this net got more than one (hidden)layer we can call it deep feed-forward neural network. It's a type of network not a network itself for example an CNN is an Feedforward ANN (Artificial Neural Network) 2. The next one on my list is the Convolutional Neural Network (CNN). You can see an image of an Deep Convolutional ANN (Artificial Neural Network) above. I will include another image after the text. These types of nets will often be used for object detection or image detection. In general all spacial (means that the data position matters like in texts or sound waves, because the filter will go over the data in their order and not random) data that is in two or 3D. It's working way could be descriped as this: fist we apply convelution, then we use max pooling and then we use a normalization function and finaly we squash it together via a fully connected layer. You could split the CNN in two parts: 1 Feature Learning and 2 Classification. A possible CNN 3. The Recurrent Neural Network (short: RNN) is used to predict time related stuff like: The next word in a text or the next event in a film (never seen but I think possible, maybe after you are done reading this you can try it out). Other usage ways are speech recognition, handwriting recognition. One main diffrence between a FFN and a RNN is that the RNN get's the hidden state + the noraml input(example: it not only getting let's say 4 it get's the 1,2,3 aswell so you can easiely predict the the next number in a sequence) In short such nets are often used for work with sequential information. 4. And now to the next one called LSTM (Long Short Term Memory) is a cell that can "remeber" things like predictions from before. They will be used together with other nets like a RNN (Most popular one). These cell will get every previous inputs to act like an save point. It's saving data for later usage. Thinks of it as a memory cell that is getting updated (in training phase).We go in dept this later. 5. Next we have got the Deconvelutional Network (DN). These networks are like CNNs but reverse, you can use them to visulize your CNNs so that you can get a better understanding of them. You can use them to generate propabilitay maps with them. So you can use them for calssification aswell in combination with other networks like a CNN. 6. A perceptron: This is the simplest Neural Network, it represents on Neuron and is ideal to start with, in order to understand Neural Networks and their behavior. It's used for liner porblems and could be used for binary classification.

5 Structure

So after giving you some of the most important nets we are talking about some technical jargon(so you now what is what). Then we build a net from scratch (a perceptron), after that we talk about diffrent classifiers, activation functions and optimzers aswell as some other parameters. When we completed this we talk about the math for this nets and how to choose the right for your problem. And finaly we will build another net with Tflearn. So stay ready and if you haven't done already make yourself a coffe or whatever you like to drink and then let's go.

6 Technical Jargon

1. A Vector: Is a one dimensional matrix that represents features(not directly the features but it represents the data) of for example an Car (with, hidth, length). It could look like this [1,2,3]
2. Activation function: Is used to apply non-linearity to our network, so we can easiely modell diffrent problems that are not linear. Without it (see later in math part) we can't use back-propagtion (see below) too fix our weights and improve the neuronal network. It's used to modell a neuron so, if a certain threshold is reached the neuron "fires" and the data moves on. This prevent the (need example, so let's say we have got an net that detect your face and print a message)net from simply saying: "Hey thats a face, so print the message" because only when your face is their, the threshold is reached and the net "fires". There are different out their, but we will cover wich to use when and why.
3. Backpropagation: A mathematical way to reverse fix "errors" in a neuronal network so. Simple it's used for the learning process.
4. Weights: The real power of an neural net are the weights. They represents the strenght of the connection and are the "brain" of a net because the weights represents/save the knowlegde of a neural network.
5. Learning Rate: The vlue that is resonsible for the learning speed of the neuroanal net (a good one is 0.001). It's the trait between time and efficiencie. (We will cover this in the math part more in dept)
6. Loss function: see cost function
7. Optimization function:
8. Overfitting: If the net is best fitted on a specific dataset but not for varius data (data that is not in the dataset)
9. Underfitting: This modell isn't well fitted on the data.
10. Bias: Let your graph don't only start at (0/0); called the origin. It's the b by the function f(x)=m*x+b. So your graph be moved at the y-axis section. So you can solve some more (not very much, but a little) more complex problems. The bias will be applied at the activation function. The bias allows you to work with input that is eqaul to 0 so you can train much better.
11. Error/Error value: The difference between the wanted and the predicted output.
12. Cost function: Takes in all weights and biases and return how bad/good they are in one number (the cost value). Parameters: The training data.
13. Gradient descent: A algorithm to find the local minima of a function.

Some note: If we are talking about learning, we mean that we have got an cost function that we try to minimize.

7 Let's build a perceptron

So first let's cover how a perceptron works. It's very simple: 1. You give it some input values 2. Multiply them by their weights 3. Summ it up 4. Apply it to an activation function 5. (Only if you train) backpropagete the error (calculate the error backwards in respect of each weight) 5. Grab the output and use it (or not) 6. Done Then some general stuff: A percepton can only be used to linear problems like if the sum over or under one, or is it black or blue, or is it a man or a woman (ok maybe this is a little bit unrealistic, but we will see) or other thinks like that. This simple grafic below show s you a linear and a non-linear problem.

7.1 Here you have got an image for an perceptron:

You can see the steps obove in this schematic overview. This is a single Layer perceptron, but for more dimensional problems you can use a multilayer perceptron. So now let's code (For you information we don't use the unit step function):

```
In [1]: import numpy as np #import numpy for math operations
        from numpy import array #needed for array operations
        #define all our needed values

        weight=np.array([0,0.999901,5.28376,2]).T#setup our first weight
        input_data=np.array([[1,0,1,0],[0,0,0,0],[1,0,0,0],[0,1,1,1]])# our input
        output_data=np.array([1,0,1,0]).T# our targetted output

        #multiply our input data with the weights
        multiply=np.dot(input_data,weight)# np.dot is a command for matrix multiplication
        print("Our multiplied data:",str(multiply))#print the result
```

```
Our multiplied data: [5.28376  0.       0.       8.283661]
```

```
In [2]: #let's normalise the data with the sigmoid function; 1/1+e^-x
        for x in range(10000):
            # because adjustement needs time we repeat the process 10k times.
            sigmoid_apply=1/(1+np.exp(-multiply))# aply the sigmoid function
            error=output_data-multiply# we calculte the error value between the output value a
            #let's adjust our weights with this value
        #but first we have to figure out how confident the net is about his prediction using\n
        #this function (we will cover why to do all this later)
            confidence=multiply*(1-multiply)
            #np.dot(inputs, self.synaptic_weights
            weight+=np.dot(input_data.T,error*confidence)#reworking this stuff
        print("Confidence/Deriviative:",confidence)
        print("Weight after training:",weight)
```

```
Confidence/Deriviative: [-22.63435974   0.           0.         -60.33537856]
Weight after training: [ 969601.64869552 4997979.2231194  5967585.15567574 4997980.2232184 ]
```

So let's code this in high-level code, were you can make predictions.

```
In [3]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        class perceptron():#build perceptron class
            def __init__(self,training_data,labels):# setup all needed parameters
                self.training_data=training_data
                self.labels=labels.T
                self.weights=2*np.random.random((3,1))-1#build random weights
                print("Weights before training:")
                print(self.weights)
                print("")


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

            def confidence(self,x):# build a sigmoid_deriviative function
                return (1-x)
        #I removed the x* infront of the brackets you can play around with it,
        #but I get better accuracy when removing it

            def predict(self,x,weight):# build out predict function
                return self.sigmoid(np.dot(x,weight))

            def train(self):# setupt our training function
                for d in range(10000):
                    prediction=self.predict(self.training_data,self.weights)#first predict
                    error=self.labels-prediction# figure out the error


                    adjust=np.dot(self.training_data.T,error*self.confidence(prediction))
                    #calculate the rate for adjustment
                    self.weights+=adjust# apply it on the weights
                print("Weight after training:",self.weights)#print the new weights

            def test(self,testing_data):
                    print("Testing output:")
                    print(self.predict(testing_data,self.weights))# predict with actuall weigh

        if __name__=="__main__":
            p=perceptron(np.array([[0, 0, 1], [1, 1, 1], [1, 0, 1], [0, 1, 1]]),np.array([[0,
            p.train()
            p.test(np.array([0,0,1]))

Weights before training:
[[ 0.97848631]
```

```
 [ 0.65068214]
 [-0.80294638]]

Weight after training: [[13.77123312]
 [-0.37607629]
 [-8.81170288]]
Testing output:
[0.00014896]
```

7.1.1 The math behinde the perceptron

So you can see we predict close to one (this exmaple is taken and modified from https://medium.com/technology-invention-and-more/how-to-build-a-simple-neural-network-in-9-lines-of-python-code-cc8f23647ca1) So let's cover in detail what we are doing here and why in respect of the math. Our perecptron is build like this formula: We simply multiply the input with the weight, sum them up and then pass it into the sigmoid function to get out predictions, htis we are doing for all our weights and intputs.(I should have add an * for muliplication inside the sum symbol). So why are we doing this? Our weights are our memorys or the ability to make predictions together with the sigmoid function, but the function is everytime the same and we have to adjust the values of our brain/memories; whatever...). This process we have take from our brain. So why we use the activation function? So the activation function is used to figure out how much activated a neuron is (in our case because the sigmoid returns values between 0-1, if the output is closer to 1 the neuron is more fiering and is it closer to 0 it's fiering less, but they are diffrent activation functions for diffrent cases and some return only 1 or 0). For the people that never seen a Sigmoid function before, here is an image: So now you now the formula for our prediction and how our activation function looks like. This function would work with training weights. (Be aware if you are working with more then on neuron, this formula is only for one neuron and without bias or other learning parameters like the learning rate. It's possible that the activation functions in another layer is diffrent from before) But let's talk about the learning process, the net learns if we adjust the weights in respect to our error. So we start to calculate the error in order to know how much we missed our target. We are doing this by subtracting the expected output by the predictet output to get the difference. In the next step we using the 'Error Weightet Derriviative" formula to calculate out adjustment. First we take the error and multiply it by the predictet value. The predictet value is between 0 and 1 (only in this formula, because the sigmoid function generalise the data), so if the predicted value is 0 the weights will not change because if you multiply numbers with 0 the output is still 0. If the output is higher we adjust the weights proportional to the error, by multiplying the gradient of the sigmoid function with the error and the prediction. The gradient of the function says us how much we have to adjust the weights (if it's high we have to adjust more and if it's lower we have to adjust less). The formula is: The 1-output is the gradient of the sigmoid function, but againe we solved this in a other way. (The real deriviative of a Sigmoid function is x*(1-x) or in our case output*(1-output). But we have apllied the ouput onece and why we should do it two times? Without the first multiplication we can increase our accuracy near to 1.

Some note: "A loss function is a part of a cost function which is a type of an objective function". (Quoted from:https://medium.com/@lachlanmiller_52885/machine-learning-week-1-cost-function-gradient-descent-and-univariate-linear-regression-8f5fe69815fd)

8 A "real" Neural Network

(Sounds a bit redicoulus because our perceptron is one as well) So now let's build an advanced neural network with more layers. This network will be used to run a "Hello World" programm for ANNs. So we use the mnist dataset to predict numbers given an image of a number. For our problem we have to construct a CNN, we will construct it using the tflearn library.

Some note: A binary classifiction problem is when you only have to data groups and you the net have to find out wich of these two is it. (0 or 1).

In [4]:
```
%matplotlib inline
import tflearn.datasets.mnist as mnist
x,y,X,Y=mnist.load_data(one_hot=True)
x=x.reshape([-1,28,28,1])
X=X.reshape([-1,28,28,1])
#print(Y[1].shape)# have a look at this in order of the shape
```

```
C:\Users\Flajt\Anaconda3\lib\site-packages\h5py\__init__.py:36: FutureWarning: Conversion of t
  from ._conv import register_converters as _register_converters


WARNING:tensorflow:From C:\Users\Flajt\Anaconda3\lib\site-packages\tensorflow\contrib\learn\py
Instructions for updating:
Use the retry module or similar alternatives.
curses is not supported on this machine (please install/reinstall curses for an optimal experi
Scipy not supported!
Extracting mnist/train-images-idx3-ubyte.gz
Extracting mnist/train-labels-idx1-ubyte.gz
Extracting mnist/t10k-images-idx3-ubyte.gz
Extracting mnist/t10k-labels-idx1-ubyte.gz
```

In [5]:
```
import tflearn


class Neural_Network():
    def __init__(self,x,y):
        self.x=x
        self.y=y
        self.epochs=10

    def main(self):
        cnn=tflearn.layers.core.input_data(shape=[None,28,28,1],name="input_layer")
        #reate an input layer
        cnn=tflearn.layers.conv.conv_2d(cnn,32,2, activation="relu")
        #first convelutional layer
        cnn=tflearn.layers.conv.max_pool_2d(cnn,2)# pooling layer
        cnn=tflearn.layers.conv.conv_2d(cnn,32,2, activation="relu")#and repeat
        cnn=tflearn.layers.conv.max_pool_2d(cnn,2)
        cnn=tflearn.layers.core.flatten(cnn)
```

```
            cnn=tflearn.layers.core.fully_connected(cnn,1000,activation="relu")
            cnn=tflearn.layers.core.dropout(cnn,0.85)# droput to improve net
            cnn=tflearn.layers.core.fully_connected(cnn,10,activation="softmax")
            #fully connected layer
            cnn=tflearn.layers.estimator.regression(cnn,learning_rate=0.001)
            #error backpropagation
            modell=tflearn.DNN(cnn,checkpoint_path="C:\\Users\\Flajt\\Documents\\GitHub\\De
            modell.fit(self.x,self.y,n_epoch=self.epochs,validation_set=(X,Y))# train net
            modell.save("C:\\Users\\Flajt\\Documents\\GitHub\\Deeplearning_for_starters\\Mn

        def predict(self,x):# need to rebuild net from above fist
            cnn=tflearn.layers.core.input_data(shape=[None,28,28,1],name="input_layer")
            #the name is usefull if you want to plot the network later
            cnn=tflearn.layers.conv.conv_2d(cnn,32,2, activation="relu")
            cnn=tflearn.layers.conv.max_pool_2d(cnn,2)
            cnn=tflearn.layers.conv.conv_2d(cnn,32,2, activation="relu")
            cnn=tflearn.layers.conv.max_pool_2d(cnn,2)
            cnn=tflearn.layers.core.flatten(cnn)
            cnn=tflearn.layers.core.fully_connected(cnn,1000,activation="relu")
            cnn=tflearn.layers.core.dropout(cnn,0.85)
            cnn=tflearn.layers.core.fully_connected(cnn,10,activation="softmax")
            cnn=tflearn.layers.estimator.regression(cnn,learning_rate=0.001)
            modell=tflearn.DNN(cnn)
            modell.load("C:\\Users\\Flajt\\Documents\\GitHub\\Deeplearning_for_starters\\Mn
                        weights_only=True)
            print(modell.predict(x))# let it predict


    nn=Neural_Network(x,y)
    nn.main()
    print("")
    print("")
    #nn.predict([X[1]])
    print("Label for prediction:"+str(Y[1]))#print wich number it should be (count from le
```

```
Training Step: 12899  | total loss: 0.38863 | time: 24.127s
| Adam | epoch: 015 | loss: 0.38863 -- iter: 54976/55000
Training Step: 12900  | total loss: 0.34988 | time: 25.643s
| Adam | epoch: 015 | loss: 0.34988 | val_loss: 0.05748 -- iter: 55000/55000
--
INFO:tensorflow:C:\Users\Flajt\Documents\GitHub\Deeplearning_for_starters\Mnist_modells\checkpo
INFO:tensorflow:C:\Users\Flajt\Documents\GitHub\Deeplearning_for_starters\Mnist_modells\mnist.t


Label for prediction:[0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
```

9 Explanation

So what you can see here is a CNN, as you can see with tflearn it's much easier to build such a network as if you do it from scratch. After this we build one net in Tensorflow, so you can decide wich one you like more. But now let's cover what we have done. A CNN usually consists of a 2 ore more `convolutional layers`, 2 or more `pooling-layers`, one or more `flatten layers`, and finaly one or more `fully connected layer/s`. We used first an `input layer` to tell our network how our data is shaped, in our case is it `(28,28,1)`. The `28,28` is the size of the image in pixel and the 1 is how much features (in our case numbers that represent the shadeing of grey) if you enter 2 you got 2 in each row. The next step was to setup a `convolutional layer`, these layer works so that we have got a field of a given size that will move over our image and within this field we multiply our weights. So we will get a smaller matrix as output. (An important note is that not every neuron in a CNN is connected with every other neuron because it needs to much computing power.). Our `pooling layer` is a `max pooling` one, so that means he uses a window and searchs for the biggest number in that window to add it in a new matrix. So we can find relevant features in our data. We repeat this process one more time and then add a `flatten layer`, this layer simply flattens our data into a a 2D matrix so we can work better with it. The first `fully connected` layer is responsible for the `feature detection`, and as the name suggests, in this layer every neuron is connected each other neuron. The `dropout layer` set the return values to 0, with that we try to generalise our net as good as possible. The last `fully connected layer` is used to make predictions wich number it can be. So that was it not as complicated as you can see. And the `modell.predict`function is the same, but instead of using the `modell.fit` function to make predictions.

So I hope this was usefull and helped you to better understand ANN (Artificial Neural Networks). I would suggest you a break and try to understand what we have done and after you got it (or not, you can try it againe later)I recommend you to do something else for this day or just one to two hours (or more, have a little walk or something like this).

Note: ANNs can be devided into 3 types. 1. `Suppervised Learning`: Learn the mapping between input and output. 2. `Unsupervised Learning`: Let the network find the labels by for example clustering. 3. Reinforced Learning: The modell uses an "`agent`" that tries to find the best way for solving a problem via `trial and error` to maximize his `reward`.

10 Math

Here we cover the following topics: 1. Gradient Descent & Backpropagation 2. Activation Functions

10.1 Gradient Descent & Backpropagation

Note: Backpropagation is the same as Gradient Descent, it's only the name in they use in ML. So you have already learned how a network make predictions, by passing the inputs through the trained weights, sum them up and then pass it inside an activation function. This is called `Feed forward` or `Forward Propagation`. Now we will talk about `Backpropagation`, this is the most important part in a ANN. So let's get started! The idea of `Gradient Descent` is to find the local minima of our error and our weights (see a grafic below). Imagen a coordinate system with the error on the y-axis and the weights on the x-axis. So if we would draw the function in it we can see that these function is a square function (this is only a two-dimensional modell, but it can be three-

dimensional or multidimensional aswell). So we want to find the best weights with the lowest error value (the local minima of the function). We use gradient descent to find the best weights and this we are doning in small steps (the size of these steps is controlled by the learning rate) so you don't miss them. So here you can see what we have done: Note: we adjust our weights by this delta w. So I would recommend you to read this article againe and have a look at my sources and read a little bit aroud. After this make a pause, before moving on. If something sounds weird please inform me so I can fix it. If we want to adjust the weights we have to use`Backpropagation`, this will allow use to change each weight in realation to the error. (So each weight contribioutes to the error, one more than others). Now let's have a look on the formula: So to find out how sensitive a weight to the error is, we have to use the chain rule and solve this equation of particial derivatives. The error could be calculated with the `MSE` (Mean squarred error.) The fuction can be found here: https://en.wikipedia.org/wiki/Mean_squared_error#predictor Note that this is for only one weight!. You adjust your weight by add the *negativ* output of this equation to the original weight and this process will be repeated for 1000 and more times, so the weights adjust slowly (we move down the gradient with each iteration). You also have to do it for the bias. Luckly all ML frameworks that support Deep Learning does this stuff for you.

Let's have a short look on the formula: To find out how much we should change the weight in relation to our error, we have to figure out first how we have to calculate the partital deriviative of z (were we multiply everything with the weights and add the bias) and the weight. The we have to do the same for the output and z and then for the Error and the output. I highly recommend you to watch this video:https://www.youtube.com/watch?v=tIeHLnjs5U8 and have a look at this website: https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/, because I find they explain it very well, better then I could so please check them out. I hope I could give you a small peak to this stuff.

10.1.1 Activation functions:

The activation function is used to transform the inputs (input values times weight + bias) for this neuron in a single value, depending wich function you choose. It's very important wich function you use for your ouput layer, to find out wich one you need you have to know wich type of problem your problem is. For example: Binary classification: were I would use the unit step function (https://en.wikipedia.org/wiki/Heaviside_step_function) wich returns either 0 or 1, or for Classification the softmax function wich returns probabilities (https://en.wikipedia.org/wiki/Softmax_function). For your hidden layers, never use linear functions **never**, because you can only solve linear problems but not non linear ones. If you need linear regression, use the linear function in the output layer. For hidden layer functions wich should solve non linear problems, use `RELU` (https://en.wikipedia.org/wiki/Rectifier_(neural_networks)) or `Tanh` (https://en.wikipedia.org/w/index.php?title=Tanh&redirect=no). Don't use the simgoid function, because it can causes a vaneshing gradient and that's not good! The best one I think is Relu, because it's computational better (other arguments can you find here:https://www.youtube.com/watch?v=-7scQpJT7uo.

10.1.2 Personal note:

So that was my short intro to Deeplearning, and I hope it was usefull for you. In the file called sources.txt are the sources for this samll Notebook. I highly recommend you to have a look inside if you haven't undestood something or if you have but you want to learn more (I add small de-

scriptions for the links so you know what it is about). I am not a ML expert in any way, so just keep it in mind. If you find spelling mistakes, wrong explanations or something else, just inform me (please be kind :)) and I will try fix it (You can add the answer or whatever if you like). If you want to go deeper into Deeplearning (haha...) I recommend you to suscribe to Sirajs channel:https://www.youtube.com/channel/UCWN3xxRkmTPmbKwht9FuE5A . He covers a lot of Deeplearning and ML stuff and explain it very good. So I would be glad about some kind of feedback and now I want to thank you that you have read untill the end, so thanks and have a nice day. (If you are curios about Deep Q Networks, I have one read for you included, originally I wanted to add this, but then I thought it would be too much. The code is documented and you could use it for your own projects, if you want to test it use modell 6.2 or 6)