Introduction

After the 'Applied Machine Learning' course in Taltech, I became interested in neural networks and how they work. In the course we were building a few basic networks in Keras so that we would get some kind of introduction to machine learning without diving into details.

Even though the course was very interesting, I felt that I wanted to know more about how things work 'under the hood' and decided to build (or at least try to build) a neural network library using only Python and Numpy with the intention of learning about the math and algorithms behind the networks in the progress.

The idea is to build this library without setting any specific targets except for learning something new and form an understanding about how those networks work. This document is intended to be a kind of blog in which I will write about the progress I have made.

Building the base for the project

To start out, I found a very nice tutorial:
Simple neural networks tutorial
For understanding backpropagation, Andrej Karpathy's tutorials were incredibly helpful:
Andrej Karpathy's series on neural networks

Following the first tutorial, I built a (very very) basic library that implements a fully connected layer, Tanh function and Mean Squared Error function. The plan is to modify this base project and add new things on top of it.

First modifications

I decided to modify the prject right away.

In the original implementation the user would have to add both the activation function and the corresponding derivative function as arguments when creating a new Activation layer. I thought that it would be better if the user could just add the activation function as a string and the program would define the necessary functions itself.

Instead of this...:

```
net.add(ActivationLayer(tanh, tanh_prime))
```

...an activation layer can be added like this:

```
net.add(Activation('tanh'))
```

I also added a Sigmoid activation function and a ReLU activation function.

Attaching activation functions to the layers

Activation functions are now attached to the layers. When defining a layer, you have to pass an activation function as an argument. At the moment it's not possible to create a layer without an activation function but it will be at some point.

Changes in weight initialization

The weights for fully connected layers are initialized as a normal distribution and biases as zeros. I would have to read about weight initialization techniques in the future but for now even this small change made a difference.

Creating models functionally

Instead of adding layers one by one like this...:

```
net = Network()
net.add(FCLayer(2, 3))
net.add(ActivationLayer(tanh, tanh_prime))
net.add(FCLayer(3, 1))
net.add(ActivationLayer(tanh, tanh_prime))
```

...I liked the idea of adding them functionally, like this:

```
inputs = Dense(input_size=2, output_size=32, activation='tanh')
x = Dense(output_size=16, activation='relu')(inputs)
x = Dense(output_size=8, activation='relu')(x)
x = Dense(output_size=4, activation='relu')(x)
outputs = Dense(output_size=1, activation='sigmoid')(x)

model = Model(inputs, outputs)
```

After the layers are defined, the model is compiled and a layer-graph is created from inputs to outputs. I thought that maybe this approach opens up the possibility to create non-sequential graphs in the future.

Adding loss functions on model compilation

Instead of adding loss functions like this...:

```
model.use_loss_function('mse')
```

...I changed it so that the loss function is defined when the model is compiled:

```
model.compile(loss_fn='mse')
```

I also made a separate class for a loss function and created forward and backward methods for it. I thought that this way it would be more consistent with the rest of the code.

Besides changing loss functions, I organized files into their folders so that in the future it would be easier to find what's needed.

Categorical Cross-Entropy loss and Softmax activation function

I implemented Categorical Cross-Entropy and a Softmax activation function. So far, it was the most difficult thing to do. I was trying to figure out the respective derivative functions for ages and in the end, failed. Then I discovered that it's a common practice to calculate the derivatives of those functions in tandem, in which case the derivative function is very simple and fast.

Because those functions are mostly used together anyway, I decided to make it possible to use them only in combination with each other, write a function that calculates their combined derivative and just move on for now. I might (should?) come back to it at some point in the future (when my math has improved, hopefully).