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. Also I will try to write down some notes that I will have found useful.

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

The first one explains how to build a (very) basic library using only fully connected layers, a Tanh activation function and a Mean Squared Error function for calculating the loss.

Fully connected layer
On initialization a fully connected layer creates a weight matrix in the size (nr of inputs, nr of outputs) and a bias matrix in the size (1, nr of outputs).
On forward pass, it takes a vector of inputs, multiplies each element with a corresponding weight, sums them together and adds bias. So basically it performs the following operation:

$$AW + B$$

where AW is a matrix multiplication between input vector A and weight matrix W, and B is the bias matrix.

Activation function To add non-linearity, the result of fully connected layer is passed through an activation function. For example Tanh function:

$$\frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Loss function

Loss function is used to estimate the correctness of the prediction (the distance between predicted values and actual values). The bigger the loss the worse the prediction. In ideal the result of the loss function would be 0.
In the tutorial Mean Squared Error function was implemented:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \bar{y}_i)^2$$

where n is the number of predicitons, y is the actual answer (label) and $\bar{y}$ is the prediction.

To learn, the model goes through three stages (many times):

- Forward pass - the model passes inputs through the network and makes a prediction
- Backward pass - the model calculates derivatives for each parameter (weight and bias) in the network in relation to the loss using chain rule.
- Update - the model updates each weight by substracting a fraction of the calculated gradient (the size of this fraction is determined by a hyperparameter called learning rate)

I followed the tutorial rather closely, but there were a couple of things I didn't really like.

One of them was that in the original implementation, a user would have to add both the activation function and the corresponding derivative function as arguments when adding an activation function to the network:

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

I thought it would be more convenient if a string argument could be used instead and the program would assign the necessary functions itself:

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

I also added the possibility to define the model by just passing a list of layers on initialization. So besides 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))
```

it's also possible to define the model in the following way:

```
model = Model(
    [
        Linear(2, 3),
        Activation('tanh'),
        Linear(3, 1),
        Activation('tanh'),
    ]
)
```

I added a couple of new activation functions and corresponding derivative functions to the project.

Changes that I didn't yet make, but might be a good idea:

- It would probably make sense for the loss function to be a required argument when creating the model instead of adding it after the model is initialized.
- Activation function doesn't have to be a separate layer, but could be a part of another layer (fully connected layer, convolutional layer and so on) as in an activation layer there are no parameters to update.
- It seems dodgy that on initialization of fully connected layers 0.5 is substracted from each weight and bias matrix element. I should check out whether this is a standard practice.
- Adding layers in a list or one after another and calling reverse(layers) in the model when backpropagating works only in a sequential model. At some point in the future it would be good to build a better way of organizing the layers so that non-sequential models could be built.

Defining layers with activation functions and creating models functionally

I managed to fix three of the points listed in the previous paragraph.

First, the activation functions are now connected to the Linear layers. So when creating a linear layer, activation function is passed as an argument.

Second, the weights are initialized as a normal distribution and biases as zeros. I will still read about weight initialization techniques later, but for now even this small change seemed to make a difference.

Third, I changed the way the models are created. Instead of adding layers one by one or passing them as a list, the layers are created functionally. Each layer created remembers the layer that precedes it. So basically, a unidirectional graph of layers is created.

Instead of defining a model 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))
```

Or like this:

```
model = Model(
    [
        Linear(2, 3),
        Activation('tanh'),
        Linear(3, 1),
        Activation('tanh'),
    ]
)
```

It is done like this:

```
def create_model():
    inputs = Linear(input_size=2, output_size=8, activation='
                                  tanh')
    x = Linear(output_size=4, activation='sigmoid')(inputs)
    outputs = Linear(output_size=1, activation='relu')(x)

    model = Model(inputs, [outputs])
    return model
```

Then the model is compiled: the layers are organized in the correct order so that the forward pass and backward pass are done correctly.

Hopefully, this approach allows for creating non-sequential models in the future. So in theory it should be possible to make something like this for example:

```
def create_model():
    # Base model
    base_in = Layer(input_dim, output_dim, activation)
    x = Layer(output_dim, activation)(base_in)
    x = Layer(output_dim, activation)(x)
    base_out = Layer(output_dim, activation)(x)

    # First branch
    b1_1 = Layer(output_dim, activation)(base_out)
    b1_2 = Layer(output_dim, activation)(b1_1)
    b1_out = Layer(output_dim, activation)(b1_2)

    # Second branch
    b2_1 = Layer(output_dim, activation)(base_out)
    b2_2 = Layer(output_dim, activation)(b2_1)
    b2_out = Layer(output_dim, activation)(b2_2)

    model = Model(base_in, [b1_out, b2_out])
```

Of course, the way forward pass, backward pass and updates work, had to be changed as well because of this way of creating models. I feel that long-term it's better to have this kind of system for creating models even though at the moment it doesn't make much difference (probably makes things slightly more difficult even?).