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).

Testing the project using Iris dataset

Until now I had tested the network only with the basic truth table dataset consisting of 4 examples to see if it learns at all, so it was quite exciting to test it with something slightly bigger. I used the Iris dataset which consists of 150 examples each of which has 4 features and is classified into 1 of 3 possible classes.

I created a couple of temporary pre-processing functions - to one-hot encode the labels and to divide the dataset into a train set and a test set. The train set had 70% of the samples and the test set 30%.

The model consisted of 3 layers

- Dense layer with 4 inputs (features) and 8 outputs using Tanh
- Dense layer with 8 outputs using ReLU
- Dense layer with 3 (possible classes) outputs using Softmax

The model used Categorical Cross-Entroy loss.

I was playing around with different learning rates and discovered that 0.01 worked the best. 500 epochs were used for training.

I trained the model using the training set and then let it predict using the test set. I did this for 5 times. Each time the train/test split was different as they were created randomly. I got the following results when predicting:

- 45/45 - 100%
- 39/45 - 86.67%
- 45/45 - 100%
- 44/45 - 97.78%

- 45/45 - 100%

I'm rather happy with the results, somehow I was expecting worse. It's a good place to move on from.
I will write the functions that I used to pre-process the data here as well.

The function for importing the Iris dataset:

```
def import_data(filename: str) -> pd.DataFrame:
    names = [
        "sepal_length",
        "sepal_width",
        "petal_length",
        "petal_width",
        "class",
    ]
    data = pd.read_csv(filename, names=names)
    data['class'] = pd.Categorical(data['class'])
    data['class'] = data['class'].cat.codes

    return data
```

The function for one-hot encoding the labels:

```
def hot_encode(y, n_classes):
    hot_encoded = np.zeros((len(y), n_classes))
    for i, c in enumerate(y):
        hot_encoded[i][c] = 1
    return hot_encoded
```

The function for separating the data into train/test splits:

```
def train_test_split(data: pd.DataFrame, split: float) -> np.
                                    ndarray:
    X = data.drop('class', axis='columns').to_numpy()
    y = data['class'].to_numpy()
    y = hot_encode(y, 3)

    shuffle = np.arange(len(data))
    random.shuffle(shuffle)
    n_split = int(len(data) * split)

    X_train = X[shuffle[:n_split]]
    y_train = y[shuffle[:n_split]]
    X_test = X[shuffle[n_split:]]
    y_test = y[shuffle[n_split:]]

    X_train = np.reshape(X_train, (X_train.shape[0], 1, X_train.
                                    shape[-1]))
    y_train = np.reshape(y_train, (y_train.shape[0], 1, y_train.
                                    shape[-1]))
    X_test = np.reshape(X_test, (X_test.shape[0], 1, X_test.shape[-
                                    1]))
    y_test = np.reshape(y_test, (y_test.shape[0], 1, y_test.shape[-
                                    1]))

    return X_train, y_train, X_test, y_test
```

The function for building the model (for some reason in this branch the Dense layers were still called Linear layers - I have already changed the name):

```
def build_model():
    inputs = Linear(input_size=4, output_size=8, activation='tanh')
    x = Linear(output_size=8, activation='relu')(inputs)
    outputs = Linear(output_size=3, activation='softmax')(x)

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

The function for calculating the accuracy:

```
def calculate_accuracy(preds, y_test, print_comparison=False):
    correct = 0
    for pred, true in zip(preds, y_test):
        y_pred = np.argmax(pred)
        y_true = np.argmax(true)
        if y_pred == y_true:
            correct += 1
        if print_comparison:
            print(y_pred, y_true)
    print()
    print(f"Predicted correctly {correct}/{len(preds)} which" +
          f"is {round(correct / len(preds) * 100, 2)}%")
```

Function calls and training-testing the model:

```
data = import_data('iris.data')
split = 0.7
X_train, y_train, X_test, y_test = train_test_split(data, split)

model = build_model()
model.compile(loss_fn='categorical_cross_entropy')
model.fit(X_train,
          y_train,
          epochs=500,
          learning_rate=0.01,
          print_loss=True)

preds = model.predict(X_test)
accuracy = calculate_accuracy(preds, y_test)
```

Writing the first tests

I used unittest framework to write the first tests. They are testing whether the loss functions, activation functions and their respective derivative functions are correct. The outputs are compared to the outputs of the corresponding functions in Tensorflow.

Input layer and Linear activation function

I created an Input layer which should always be the first layer of the network. It was worth adding it mainly because it makes the forward and backward functions in the Model class a bit cleaner. Because the first layer behaves differently

to all the layers after it, it might also be easier to implement new layers later on because I don't have to think about two different scenarios: layer as the first one in network vs layer mid-network.

Writing tests for layers

I wrote some more tests testing the initialization of Dense and Input layers. They are testing whether the arguments passed on initialization are of correct type and in the permitted range of values. I also wanted to make sure that some variables cannot be changed after the initalization or can be changed only inside the class but it proved to be more difficult than I thought in Python because of the lack of private variables and methods. At the moment I left it as is.

Adding first weight initialization techniques

I added the following options for initializing weights:

- Random normal
- Random uniform
- Zeros
- Ones
- Xavier normal
- Xavier uniform
- He normal
- He uniform

Actually Random normal technique was used before to initialize weights and Zeros was used to intialize biases, so those two are not new. The others are.

I used the function $0.3x_1^2 + 0.2x_2 + 0.5$ to create a dataset and trained a network to try to learn this function. It's not a very good (or fair) test by any means, but I wanted some confirmation that using those new initialization techniques could result in a better performance. And I got it, because when using two layers - one with either Sigmoid or Tanh activation function and another with Linear activation function - and switching between different initialization techniques then one of the Xavier techniques won almost every time.

Like I said, this test isn't very fair but it's good to see that there are some new options and they might actually help in making the model better depending on the task