Mathematics for Machine Learning

(MA4029)

7.5 HP

Project Report

Neural Network

Multilayer Perceptron (MLP)

Dennis Mitzéus – 70002183

Halmstad University

2024-05-16

Introduction:

This project aims to improve the understanding of practical aspects for creating Neural Networks and how they work. The task was to create a Multilayer Perceptron (MLP) and train it on three functions: f(x) = exp(x), f(x) = sin(x)/x and f(x, y) = sin(x) + cos(y). Two of these three functions are two-dimensional, meaning we have a single input and a single output, whereas the last one is a multivariate function, taking two inputs and creating an output, making that one three-dimensional (if you would plot it).

When implementing an MLP, one must understand that it consists of several deep-connected layers, also called dense layers (which I also call it in my code), which connects every neuron in one layer to every neuron in the second layer, creating *i* x *j* number of weights between two layers (excluding biases, which is added on separately size *i*).

Neural Network Structure:

My initial philosophy when structuring my logic for forward- and backpropagation was to utilize bigger functions with more logic in them to be able to better track the data as it moved forward or backward, but I quickly stumbled upon several issues with this approach as correctly sized matrixes for matrix multiplication became very hard to debug along with its non-expandable approach.

I then decided to scrap that whole mindset and start again from the very beginning now instead trying to make as small and divided components as possible, which worked much better already from the beginning. I divided most logical parts into their own classes and subclasses to then be able to reference these simple I/O components completely disregarding all complicated matrix calculations as that was handled inside the actual component itself only requiring one specific input (of some size) spits out one specific output (also of some size).

What is a layer?:

I found an interesting approach whereas dense layers can work like I described before; activation functions can also work the same way. This means my Neural Network technically has double the layers by logic, making it very easy to handle everything the same way, neuron gives an output from an input, now becomes input for an activation function which also gives an output for the next layer after that… and so on.

Inside a Neuron (or activation function):

Every neuron and activation function has three things: an input attribute (later used for calculating gradient), an input method (forward pass) and an output method (backward pass). The neuron also has two attributes for weights and bias which automatically calculates to be the size of the input to that neuron.

The forward pass does some very simple calculations: $x \cdot w + b$ using input x, and weights w and bias b to get an output which is given to the next layer.

The backward pass is a little more complicated as we need to calculate three parts: $\nabla w, \nabla b \; and \; \nabla x$. First two is used to adjust w and b with gradient descent and the third one is merely a simple output to the next layer (which of course is needed to calculate the gradient for all variables used in the previous layer), as simple as that.

Is it really efficient to do calculations on every individual neuron?:

Simply, no. That's why I lied in the previous block. Everything is done in the layer, meaning instead of one bias and a vector of w, we have a vector B and a matrix X making all calculations between the two layers much more cost effective (in terms of time and recourses).

So, the actual forward pass looks more like this: $Output = X \cdot W + B$

And backward calculations look like this:

$$\nabla X = \frac{\partial E}{\partial X}$$

$$\nabla W = \frac{\partial E}{\partial W}$$

$$\nabla B = \frac{\partial E}{\partial B}$$

This also means that the activation function gets size j input, does the activation and outputs the same size j again for the next layer.

Data creation, splitting and preprocessing:

For creating the data for all the functions, I simply got an np.linspace() of some range, and inputted all x into the function and got y.

For splitting the data, I used Scikit-Learn's train_test_split, splitting it 60/40 train and test, also shuffling it (keep in mind, I did not use the provided data as validation as I only used validation for testing that the NN trains at all).

Now, to decrease exploding gradients and a bunch of other benefits I normalized the data by using a simple min-max scaler. Especially as the functions such as exp(x) quickly gets very large.

Lastly, I reshape all data to have at least one dimension which is of length one to enable arithmetic matrix computations. So, for the first function with 200 data points (200,) becomes (200, 1, 1), while the last one is example 200 data points (200, 2) becomes (200, 2, 1).

Now it's good to go to fit, train and test on.

Neural Network Validation:

Before I train and test on the actual data I want to use, I just used an example XOR validation dataset. As it is only for actually testing that the NN trains and minimizes the error in a satisfactory manner we do not necessarily need to use a subset of our testing or training data. So, I opted with a classic XOR problem.

Loss reports:

The general range of the losses are about between 0.5 and 0.001 which is exactly what I wanted. Although I tested a lot of different setups for size, depth, learning rate and number of epochs, I came to a sacrifice where it was not realistically possible in the limited span of time I had to test to find the most optimal variables totally possible, meaning in some functions I went a little overkill taking much longer to train than possibly needed by making it both deeper and broader. But I got a good result that I am happy with regardless, so I won't continue trying to find faster configurations at this moment in time.

Train and test results:

This is the result of all training and testing errors:

	Train MSE	Train MAE	Test MSE	Test MAE
$f(x) = e^x$	0.000934	0.010989	0.000790	0.012503
$f(x) = \frac{\sin(x)}{x}$	0.000314	0.013622	0.000256	0.015673
$f(x, y) = \sin(x) + \cos(y)$	0.000714	0.020356	0.000523	0.021145

Results reflection and discussion:

There has been a lot of issues regarding this project in my experience, both before and after I started from the beginning, especially regarding having the right dimensions of all input and output data (and because most calculations were matrix arithmetics when doing calculations per layer and not per neuron). But as things started working and I found a good approach, everything slowly got better and I got the chance to expand and create a little library with built in plotting, stats and error printing, only requiring train and test inputs.

There were a lot of things I would have wanted to test, including an alternative to NumPy using PyTorch and CUDA, which I experimented a little with but did not work as expected using my first try, so that is something I want to test to implement in this current version. Also training features like a gradient descent optimization algorithm, and an overfitting prevention algorithm (comparing train and test error over a set number of epochs, seeing when training error goes down but testing error goes up) is something I would have wanted to test to implement.

But as the results of the testing is good over all three functions that was trained, I am generally satisfied with the outcomes of this project and my contribution.