Artificial Intelligence: Neural Networks

Marián Kurčina

ZS 2024

xkurcinam@stuba.sk
marian.kurcina2003@gmail.com

Table of Contents

MNIST Classifier

Task

The goal is to create a neural network for classifying handwritten digits from the MNIST dataset. A feedforward neural network (multilayer perceptron) should be used and trained with the following optimization algorithms: SGD, SGD with momentum, and ADAM. In addition to measuring the training and test error, the model's accuracy should also be evaluated. The final model should achieve an accuracy of more than 97%. The PyTorch library should be used for implementation.

Solution Description

First, I load the dataset into the program and transform it so that the values range between 0 and 1. Then, I split the data into two parts: a training set and a test set. After obtaining and preprocessing the dataset, I train and test the model separately for each optimization algorithm.

The training process runs for several epochs. In each epoch, batches of digit images are fed into the model, generating predictions. The difference between the predictions and the actual labels is used to compute the gradient, which updates the linear layers of the model. The way these layers are updated depends on the training algorithm.

At the end of each epoch, I print the average training error and evaluate the model on the test set. I store the test results and proceed to the next epoch. After completing a predefined number of epochs, I conduct a final test on the model, which represents the final trained version. Finally, I generate graphs displaying the training and test errors along with the model's accuracy.

Code Description

At the beginning, I load and transform the MNIST dataset into train_dataset and test_dataset, where the first part is used for training and the second for testing.
Next, I divide the dataset into batches using train_loader and test_loader. These loaders facilitate training and testing.
The training and testing process follows. I first test using SGD, which requires a learning_rate parameter. Then, I test using SGD with momentum, requiring both learning_rate and momentum. Finally, I use ADAM, which requires only learning_rate.
The training process starts with defining the model and its layers in the code.
For each epoch:

- I reset the gradients.
- I compute the model output and the loss.
- I perform backpropagation using the loss function.
- I update the linear layer weights using the selected optimizer (SGD/SGD with momentum/ADAM).

The model output is always given as probabilities for each digit.
For every epoch, the program stores the training loss. After processing all images in the dataset, it evaluates the model using the test set, storing the accuracy of correct predictions and the test

error. These results are visualized in graphs at the end, and the model's accuracy is further analyzed using a confusion matrix.

Optimizing Training Parameters

As a first step, I decided to find the best parameters for each training algorithm, including the learning rate and momentum over 30 epochs.

	SGD	ADAM
0.1	**95.61%**	86.89%
0.01	78.48%	97.31%
0.001	11.35%	**97.70%**

Table 1: Results of testing the optimal learning rate for the training algorithms SGD and ADAM

	0.1	0.01	0.001
0.95	**98.18%**	96.89%	88.59%
0.9	97.79%	95.77%	77.98%
0.85	97.95%	94.29%	61.09%

Table 2: Results of testing the optimal learning rate and momentum for the training algorithm SGD with momentum

From the testing, I found that the best learning rate is 0.1 for SGD and SGD with momentum, while for ADAM, it is 0.001. The optimal momentum for SGD with momentum is 0.95.

Finding the Optimal Model Hyperparameters

First, I focused on selecting the right activation functions. Since my network consists of two layers, I needed to choose two activation functions. I decided to test three different activation functions: ReLU, Tanh, and Sigmoid, and I evaluated all possible combinations.

	ReLu	Tanh	Sigmoid
ReLu	97.23%	97.56%	97.26%
Tanh	97.73%	97.70%	97.63%
Sigmoid	**97.80%**	97.76%	97.02%

Table 3: Results of testing all combinations of activation functions. The first activation function in the network is listed in the columns, while the second is in the rows. The table presents the averages for all training algorithms (learning rate = 0.1, momentum = 0.9, number of epochs = 30, first layer = 100 neurons, second layer = 50 neurons).

From the test results, I found that the best activation functions to use are ReLU and Sigmoid. Next, I searched for the parameters of the linear layers, specifically the number of neurons in each layer, to maintain the highest possible accuracy.

	150	100	50
80	97.97%	97.88%	97.30%
50	97.94%	**97.80%**	97.29%
20	97.87%	97.59%	97.04%

Table 4: Results of testing combinations of neuron counts in layers

From the testing, I determined that the best choice is to use 100 and 50 neurons.

Optimal Model Architecture

My network will have 2 hidden layers, the first with 100 neurons and the second with 50 neurons, using ReLU and Sigmoid as activation functions.

- Accuracy with SGD: **97.46%**
- Accuracy with SGD with momentum: **98.04%**
- Accuracy with ADAM: **97.93%**

For each run of the program, a confusion matrix is displayed.

```
Predicted->      0      1      2      3      4      5      6      7      8      9
-------------------------------------------------------------------------------
         0   |  969      0      2      0      0      0      4      2      2      1
         1   |    0   1130      1      1      0      0      0      0      3      0
         2   |    5      1   1010      3      3      0      1      6      3      0
         3   |    0      0      2    992      0      6      0      4      2      4
         4   |    1      0      3      0    962      1      2      3      1      9
         5   |    3      0      0      7      1    870      5      0      3      3
         6   |    2      2      3      0      6      4    940      0      1      0
         7   |    1      4      7      1      0      0      0   1003      3      9
         8   |    2      0      3      5      4      2      1      2    950      5
         9   |    2      3      1      5      9      2      1      4      4    978
```

Image 1: Confusion matrix

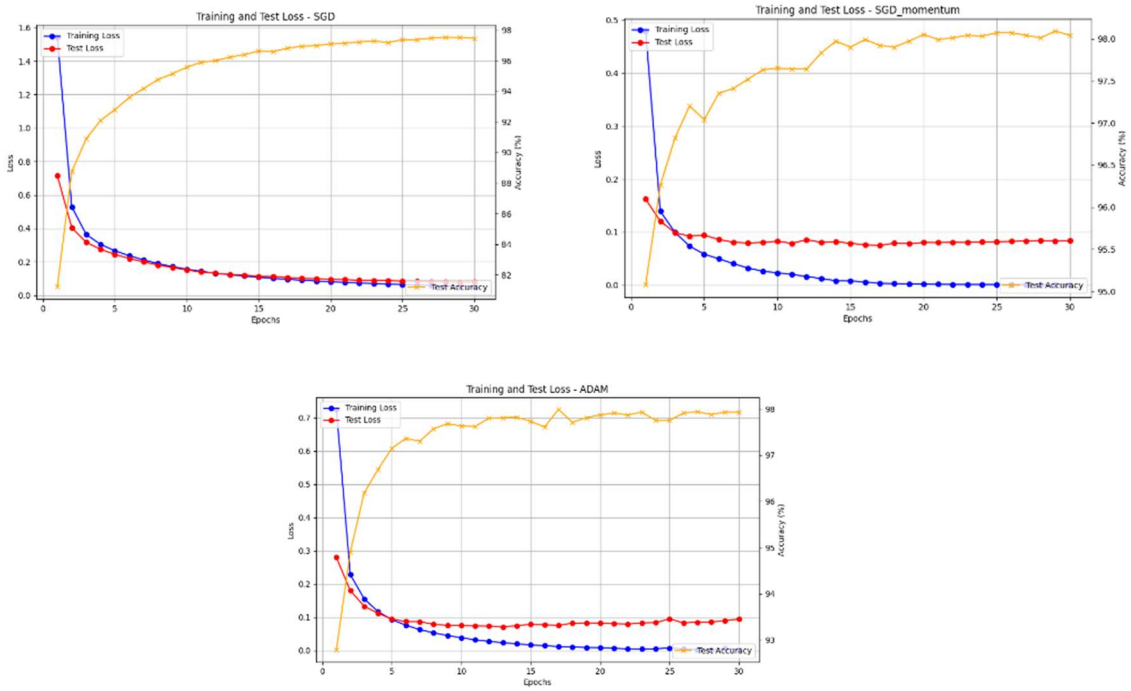

Image 2: Graphs of training and test error progression and model accuracy

Conclusion

I successfully learned to work with the PyTorch library and created a program for digit classification using the MNIST dataset and a neural network. I was able to identify the optimal model architecture and the best parameters for the training algorithms SGD, SGD with momentum, and ADAM.

Backpropagation algorithm

Task

In this task, you need to implement a fully functional backpropagation algorithm, which includes both the forward and backward passes for operators and functions, as well as the update of the network parameters. You will validate the algorithm by training a feedforward neural network. Use the NumPy library to solve the task (PyTorch and TensorFlow libraries are prohibited). Implement a modular architecture where modules can be chained. The implementation must include a linear layer, activation functions sigmoid, tanh, and ReLU, and a mean squared error (MSE) loss function. For validation, use the AND, OR, and XOR problems, and test networks with one and two hidden layers.

Solution Description

Since the model must be capable of chaining, the individual layers and the entire model will be implemented using object-oriented programming. Each layer will be its own object with functions. The model can be defined using layers, which allows easy modifications to the neural network.

Training will involve repeatedly inserting input combinations of 1s and 0s into the network, obtaining the network's prediction, and adjusting the weights of the linear layers using the difference and its gradient.
Training will take place in epochs, where in each epoch, all combinations of 0 and 1 pairs will be inserted into the network as input. The model will predict, and from this prediction, the MSE loss will be calculated to determine the error. This error will be used to calculate the gradient. This gradient will then be sent backward, and in each layer, the weights and biases will be updated using matrices of gradients or momentum.
After the gradient passes through all layers, the process will restart with a new input or a new epoch. After the given number of epochs, the predictions for all inputs will be output, and the user can check if the predictions are correct.

Code Description

The model, the MSE loss function, and all layers will be defined as objects with forward and backward functions. The model can sequentially call the functions of its layers. In the forward pass, it calls the forward functions, and in the backward pass, it calls the backward functions.
The training process works as follows: the model receives some input, which is fed into the model, and the model sequentially calls the forward function for each of its layers, passing the output of the previous layer as the input to the next layer. After passing through all layers, a prediction is made. The difference between the prediction and the true label is calculated using MSELoss.forward, and from this error, the gradient is computed. This gradient is then sent backward through the model. The model sequentially calls the backward functions for all layers, with the output of the previous layer becoming the input for the next layer. When the gradient passes through a linear layer, the weight and bias matrices are updated.
After the backward pass, the process continues for the next input.

Model

The model is defined by its layers. The forward method sequentially calls the forward methods for all its layers, passing the output of each layer as the input for the next layer. The backward method sequentially calls the backward methods for all layers, passing the output of each layer as the input to the next layer.

MSELoss

This object is used for calculating the model's error and computing the gradient.

Forward Method: $\frac{\sum(predicted-targe\ )^2}{output\ size}$ – Average difference squared

Backward Method: $\frac{\sum 2*(predicted-targ\ \ )}{output\ size}$ – Mean for all components of the output

Linear Layer

The linear layer is the only layer that updates its parameters during backpropagation. This layer must remember the weight matrix (W) and bias matrix (b) for the forward method. Additionally, the

layer must remember the last input, and for calculating momentum, it must store the velocity matrices (vW) for weights and (vb) for biases.

Forward Method: $input * W + b$

In the forward method, the input is multiplied by the weight matrix, and the bias vector is added to it.

Backward Method: $dW = grad * input^T$

db = grad

With momentum:

$$vW = momentum * vW - learning\ rate * dW$$

$$vb = momentum * vb - learning\ rate * db$$

$$W = W + vW$$

$$b = b + vb$$

Without momentum:

$$W = W - learning\ rate * dW$$

$$b = b - learning\ rate * db$$

In the backward method, the gradient matrices (dW, db) are first calculated. These will be used to update the weight matrix and bias matrix. If momentum is used, velocity matrices are calculated using the momentum and the gradient matrices. These matrices are used for updating the weight matrices.

Sigmoid

The sigmoid activation function must remember the value of the last output, as this value is used during the backward method calculation.

Forward Method: $\frac{1}{1+e^{-x}}$

Backward Method: $grad * output * (1 - output)$

Relu

This activation function returns 0 if the input is negative, or x if the input is positive. This object must remember the last input from the forward method.

Forward Method: $\max(0, x)$

Backward Method: $grad * (input > 0)$

Tanh

The activation function **tanh** must remember the value of the last output, which is used in the calculation of the backward method.

Forward Method: $\frac{e^x - e^{-x}}{e^x + e^{-x}}$

Backward Method: $grad * (1 - output^2)$

Searching for optimal model hyperparameters

I am given that the network should have 1 hidden layer with 4 neurons. However, I also need to test networks with 2 hidden layers. To do this, I decided to have 2 layers with 2 neurons each, so I can compare the performance of two networks with the same number of neurons but different numbers of layers. I will find the optimal parameters for the first network and use these parameters to compare the performance with and without momentum.
The parameters I need to find are the learning rate and which activation functions are the most advantageous for solving the problem. I searched for activation functions separately for each network. The value of momentum will be the same as in the first task, so 0.95.

	0.1	0.05	0.01
Sigmoid	1.4237	1.8026	1.9729
Tanh	**0.0511**	0.0776	0.2674
ReLU	0.9904	0.9990	1.0001

Table 5: Testing results for a network with one hidden layer for 500 epochs for the XOR problem without momentum. The success is compared based on the total deviation of the final model

From the test results, I found that the best activation functions for the network with one hidden layer and 4 neurons are Tanh, and the best learning rate is 0.1. Following this, I found the best activation functions for the network with 2 hidden layers along with 4 neurons.

	Sigmoid	Tanh	Relu
0.1	1.9773	0.4794	2

Table 6: Testing results for 2 hidden layers for 500 epochs for the XOR problem without momentum

In the case of 2 hidden layers, I observed that Tanh maintains the same loss and only occasionally improves, while ReLU is unable to learn, and its values remain unchanged. Therefore, for this problem, it is better to use only 1 hidden layer.

Architecture of the optimal model

The ideal model for solving logical operations without momentum is a model with one hidden layer and the Tanh activation function.
My model will have 1 hidden layer with 4 neurons and 2 activation functions, both Tanh.

	XOR	OR	AND
Bez momentu	0.0663	0.0413	0.039
S momentom	1.0002	1.0051	3

Table 7: Final accuracy based on loss for individual logical problems

For my ideal model, I also tested training with momentum. If I used a momentum of 0.95, the network could not be trained with the Tanh activation function, but when I tried the Sigmoid activation function, the training was successful.

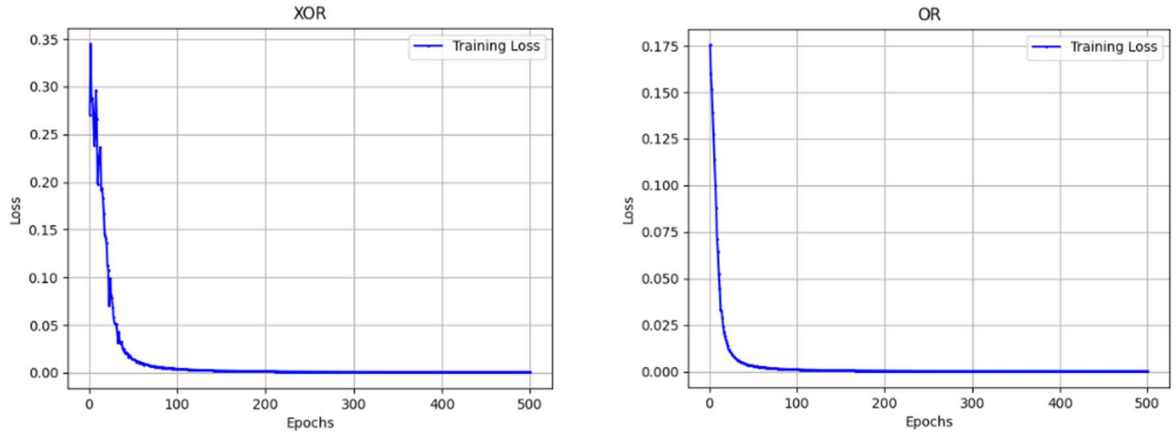

Image 3: Graphs showing the development of the error across epochs

Conclusion

I successfully implemented the backpropagation algorithm, implementing all the required parts and programming both the forward and backward passes. My model is capable of chaining, and its layers can be modified. I managed to find the optimal model for all XOR, OR, and AND problems using both one-layer and two-layer models (hidden layers), with and without momentum.

Sources

Building a neural network FROM SCRATCH (no Tensorflow/Pytorch, just numpy & math)
https://eli.thegreenplace.net/2018/backpropagation-through-a-fully-connected-layer/
https://www.youtube.com/watch?v=Ilg3gGewQ5U