UMEÅ UNIVERSITY **October 12, 2020**
Department of Computing Science
Project Report

5DV124 - LP1 2020

Fundamentals of Artificial Intelligence

ANN - MNIST

Participants
Oliver Larsson c18ols@cs.umu.se
Filip Henningsson c18fhn@cs.umu.se

Teacher
Ola Ringdahl

Graders
Lois Vanhee
Adam Dahlgren
Timotheus Kampik
Andrea Aler Tubella
Lennart Steinvall
Carl-Anton Anserud

Contents

1 Problem Specification

This assignment is about creating an artificial neural network (ANN) that is able to classify handwritten digits, taking ASCII images as input. The images used are a subset of the MNIST dataset. The network is only trained to identify the numbers 4, 7, 8 and 9 to reduce training time requirement.

2 Usage and user's guide

The program can be run by navigating to the program root folder, where `digits.py` is located, and executing:

```
python3 digits.py training-images.txt training-labels.txt
↪ validation-images.txt
```

Where `training-images.txt` is a file containing images to train the network on, `training-labels.txt` is a file containing the correct labels for the already supplied images and `validation-images.txt` contains data to classify when the network is trained.

Output from this program will be in the ASCII based file format given by the assignment specification and written to `stdout`.

3 Algorithm Description

The main ANN algorithm can be divided into two parts: The forward propagation phase, where inputs are passed into the network and each layer propagates it's activation into the input of the next layer. And backward propagation where the calculated error of the output is passed to the layers in reverse order and the network parameters are updated. A detailed description of these two phases are presented in the following sections.

3.1 Forward Propagation

To understand the forward propagation through the network it is vital to understand the structure of the network. Each layer contains a set of weights and biases as well as a non-linearity called the activation function of the layer. To calculate the activation of a layer, the inputs are multiplied by the corresponding weights and the biases are then added to the result. This yields so called logits which are then passed through the activation function to yield the final activation of the layer. Mathematically this can be represented with matrix

operations.

$$z_l = I_{l-1} \times W_l + B_l$$
$$a_l = f(z_l)$$

Where subscripts represents the layer, z is the resulting logits, I is the input, W is the weights, B is the biases, f is the activation function and a is the activation of the layer.

For most of the layers, the activation function used was the *Leaky ReLU* function:

$$f(z) = \begin{cases} z & z >= 0 \\ 0.1z & z < 0 \end{cases}$$

The exception to this was the output layer where the hyperbolic tangent was used instead, This was to limit the activation to not exceed 1.

3.2 Backward Propagation

When training the network, the perceived cost of a prediction can be calculated using a loss or cost function. Since we want to minimize the cost of the output we can calculated the derivative of the cost with respect to each parameter in the network. We can then use these derivatives to update the parameters as to minimize the cost. When this is done over each training example, the network should also get better at classifying completely new data.

Since the derivative of each layer depends on that of the next layer, it is more efficient to calculate them in reverse, starting from the last layer, and propagating the derivatives backward. This is the core of the backpropagation algorithm.

Like forward propagation, backpropagation can be expressed using matrix operations.

$$\delta_l = \frac{dC}{da_l} \times f'(z_l)$$
$$\frac{dC}{dB_l} = \delta_l$$
$$\frac{dC}{dW_l} = a_l \times \delta_l$$
$$\frac{dC}{da_l} = \delta_l \times w_l^T$$

Here the last equation propagates the derivative, and the rest calculates the gradient for the network parameters.

The parameters are then updated using a learning rate γ, the error ϵ and the

calculated gradient

$$W_{l_{e+1}} = W_{l_e} - \gamma\epsilon\frac{dC}{dW_l}$$
$$B_{l_{e+1}} = B_{l_e} - \gamma\epsilon\frac{dC}{dB_l}$$

3.3 Other systems

In addition to these two core parts there are a couple of lesser features which aim to improve the learning process.

L2 regularization is used to limit the complexity of the network. L2 regularization works by adding a fraction of the squared L2 norm of the weights to the cost of the output. This limits the complexity of the network by placing a cost on weight outliers which might cause overfitting to the training data.

Batching splits the training data into smaller datasets and performs backpropagation on them individually each epoch which speeds up training.

Last but not least, learning rate decay. Learning rate decay works by decreasing the learning rate of the network. This makes the network less likely to "overshoot" a minimum as it takes smaller and smaller steps the further along the training process it gets.

4 Systems Description

The network is represented by the class `ANN` which can be found in `ann.py`. By invoking functions in this class the network is configured and used. When instantiated, the network is created with only a single layer. This is however, simply expanded upon by calling `add_layer()` and supplying an amount of neurons that layer should contain, as well as an activation function for that layer. This makes the network very modular and easy to test different configurations with. Each of these layers is represented by an instance of the `Layer` class.

To train the network, `train()` is called. This function trains on supplied learning data and corresponding labels using back-propagation over a supplied number of epochs. To then classify inputs with the network, `eval()` is called.

4.1 Network layout

The final network is configured with 4 layers. The input layer contains 784 (28^2) neurons, one for each pixel in the images. The subsequent 3 layers all contain 10 neurons each. The 10 neurons in the last layer each correspond to an output digit, 0 through 9.

As the network trains itself it is hard to know what each of the hidden layers will represent. Theoretically, the second layer could for example detect edges in different parts of the images, while the third detects patterns among these edges. However, as this is up to the network to decide, it is very likely that each of these layers represent different things each run.

5 Limitations on the Solution

The speed, or lack thereof, is probably the biggest limitation on the solution. On CS' `itchy` server, training the network with the supplied dataset takes approximately 6 seconds. This is within the 30 second time limit, but points towards some inefficiency along the way.

The network currently trains over a set amount of epochs. If a stop condition were to be implemented, it would be able to minimize the overfitting currently done by the network, see figure 2.

6 Test runs

When testing our solution, we plotted our loss and accuracy over the course of training to see how our network evolves. The experienced loss for both training and testing closely follows each other. It was also found that loss decreases quickly until roughly epoch 15, after which the curve mellows out. See figure 1.

The training accuracy observed hits a global maximum around epoch 13 and then starts to decline slightly. After about epoch 25, a bigger decline can be seen. Where this overfitting occurs varies a little from run to run and is why a stop condition could prove beneficial. See figure 2.

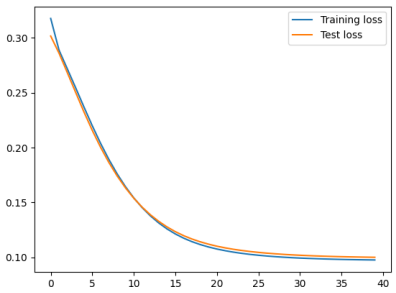

Figure 1: Loss over epochs.

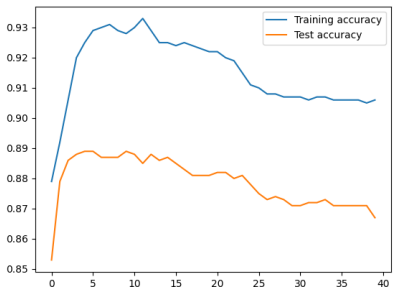

Figure 2: Accuracy over epochs.

The resulting number classification network has, when testing through labres, an accuracy of 93.8%. This is well above the required accuracy of 85%.

7 Problems and Reflections

The biggest problem we faced while developing the program was that we initially intended on using categorical crossentropy as our loss function in combination with softmax as our last layer activation function. We never got this combination working and we suspect that this is because we never managed to find the correct derivative of softmax. We spent the majority of our development time on debugging this issue but ended up falling back to our backup, mean squeared error as loss function and tanh as our last layer activation function.

When trying different layouts for the network, we found that while increasing the amount of neurons in each layer to, say, 20 each increased the predictive power of the network, it also slowed down the execution and training of the network. As such we decided that the increase in time was not worth the marginally better accuracy. Similar observations were made for the amount of hidden layers used.

Filip started by setting up the python boilerplate code, we then sat down together and wrote the code using Visual Studio Code's Live Share feature. After finishing most of the code, Filip went on to comment and make sure the code was to standard and followed the specification while Oliver wrote the initial draft of the report. We then finalized everything together.