Module 5 report

Håkon Gimse, Kari Skjold

General description'

Our DigitRecognizer class takes eight parameters which we use to build the neural network. The size of the training data is 60 000 (the whole set of images we were given), the number of input nodes is 28*28 = 784, and the number of output nodes is 10, because there are 10 possible output answers, that is, digits between 0 and 9. Number of hidden layers, how many hidden nodes in each layer, activation functions and the learning rate are parameters that we take as inputs because we have experimented with these to get the best possible result. These functions are described in detail later.

We start by building the neural network. Initially, we set the network's weights to a random value between -0.1 and 0.1. The size of the weight matrix between the input layer and the first layer will be the number of input nodes (60 000) **x** number of nodes in the first layer. The next weight matrixes represents weights between hidden layers. The last weight matrix represents the weight between the last hidden layer and the output layer. The size of this matrix is therefore the number of nodes in the hidden layer **x** 10. The input matrix and the first weight activates the first hidden layer. The next layers will be activated by the previous layer and the weight from the previous layer to this layer.

Our training function uses a theano function called trainer, that takes input and target as parameters and returns the sum of squared errors for that epoche. Each time trainer is called, the weights are updated with the backprop_acts function. The training function sums up the errors from all the epochs and returns the summed error. We also have a function called predictor, used for testing. The predictor does the same thing as the trainer, but does not update the weights. Testing is simply done, by running the test data through the network using the predictor function, and comparing the expected result, against the index of the output node with the highest activation function.

Case 1 - Initial network

In our first case we constructed a neural network with only one hidden layer. This layer contained 700 nodes. We used a sigmoid function as activation function for both the activated layers. We found that 700 nodes performed better than any alternative we tried. We read that "A rule of thumb is for the size of a hidden layer to be somewhere between the input layer size and the output layer size" (Blum, 1992, p. 60). After some testing we proved this wrong in this case. We will elaborate more about this experimentation in case 3.

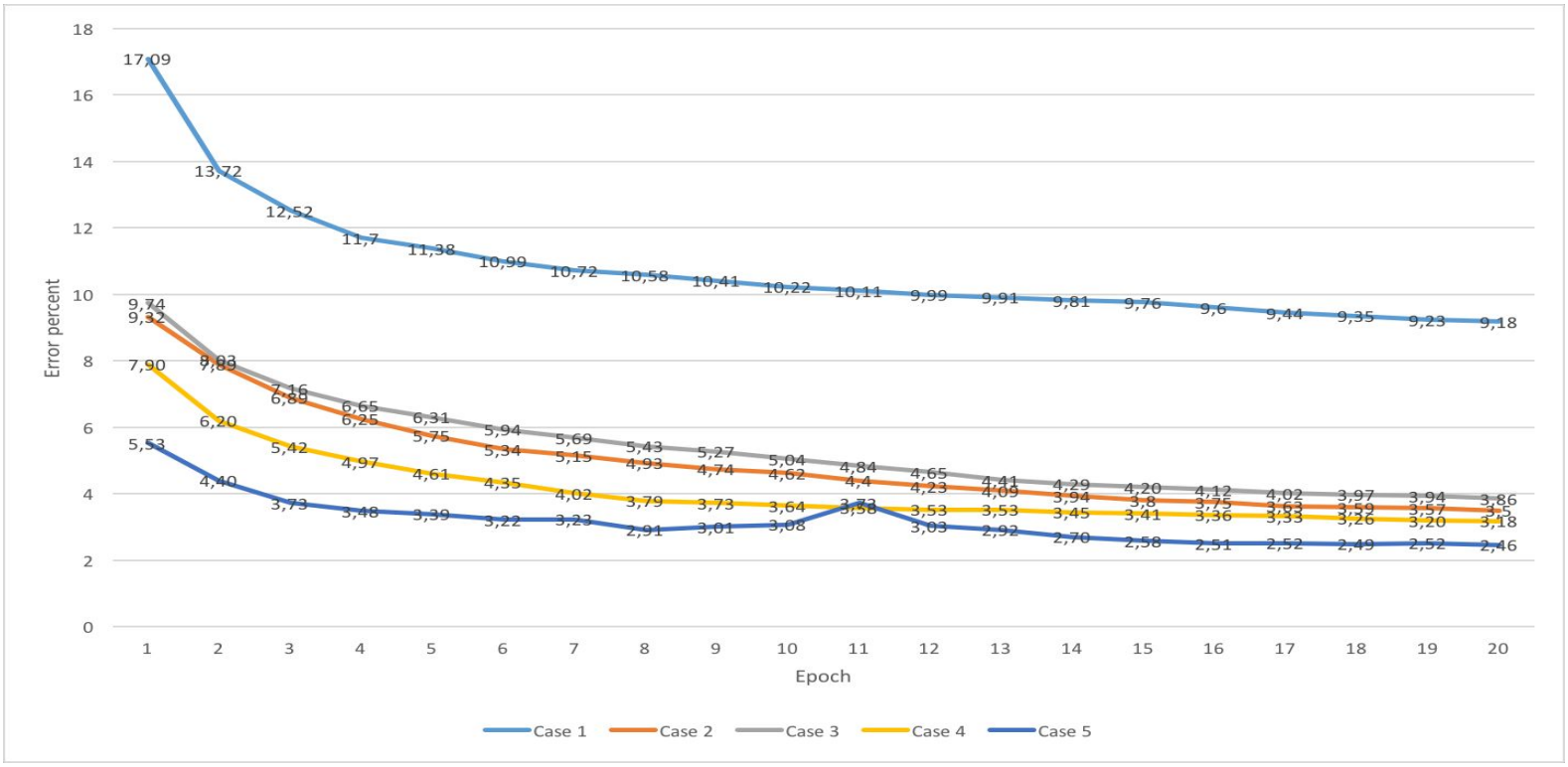
This case is the worst performing case we have taken into consideration, the reason for selecting this one, is because it was one of the first network configurations that actually worked and reached an accuracy of 96% within the time limit. It is also fairly simple.

Case 2 - Activation function
From the first case to the second we experiment with different activation functions. Our dynamic ANN builder has four parameters for selecting different activation functions. The options are Tanh, Sigmoid, Rectify and Softmax. These are described below. We decided on using the same configuration as in case 1, but with Rectify as activation function in the hidden layer, and Softmax in the output layer. This preformed ca 8% better than the first case, and was within 96% in just a few epochs. This was the best combination of activation functions we found, using one hidden layer.

Case 3 - Number of hidden nodes
Now we that we had found something that performed reasonably well, we wanted to experiment with the number of hidden nodes again. This time with better activation functions. We worked our way from 100 nodes tio 1000 with steps of 100. This experiment yielded the same results we had gotten initially. From 3 and up, there were not much difference, but 700 performed a little better than the rest. We tried with the number suggested by the rule of thumb, 350, and stuck with this as the case. It performed ca 0.5% worse than case 2.

Case 4 - Several hidden layers
In case 4 we wanted to experiment with the number of hidden layers. An increase in number of layers makes the network more complex, and can become more fine-tuned. We had written code for dynamically building the network with several hidden layers, so testing was made easy. We tried with up to 7 hidden layers, each containing a different amount of nodes, but we found that more than three layers performed worse than our previous case. Training a network with several hidden layers also takes considerably longer than with fewer layers. We chose case 4 to be a network with two hidden layers with 700 nodes in each of them, for better comparison with the previous cases. We found that two hidden layers performed best, and had an increase in accuracy on about 0.4% from case 2. The activation functions used here was Rectify, Rectify, Softmax.

Case 5 - Learning rate
In the final case we experimented with the learning rate. We figured that this should be the last fine tuning parameter to be experimented extensively with, as it would affect the comparison between previously tested networks the least. The rest of the network is configured as in case 4. We found that our networks only performed well using a learning rate in a small interval, between 0.013 and 0.001. This was found during extensive testing this of this parameter. If we use learning rates close to the top of this range, the results jumps between getting smarter and dumber, but it learns faster to a certain point. We ended up with using a learning rate of 0.01 as this was close enough to the top of the range, to learn fast, but not close enough to give the effect of learning to much each run, and corrupt the network. This network performed very well and achieves 97.5% after 20 epochs.



Tanh: A hyperbolic tangent activation function

Sigmoid: S-shaped function above 0. Given by the formula:

$$S(x) = \frac{1}{1 + e^{-x}}$$

Rectify: A simple ramp function, pulling the input in a neuron up to above 0. Given by the function: $S(x) = Max(0, x)$

Softmax: Softmax function, given by the function:

$$softmax_{ij}(x) = \frac{\exp x_{ij}}{\sum_k \exp(x_{ik})}$$

Usually used to activate the output layer.

Error function: We are simply using the sum of squared errors in all our cases.

Preprocessing: Before running the images through the network we normalize all the pixel values to a float in the range [0, 1].

Bulk processing: We pass several images at the time through the network, before backpropagating. This way, training is faster.

Automated ANN Construction

We construct our ANN dynamically using user input from the command prompt in the beginning of each run. The user gets to specify the number of hidden layers, how many nodes each individual hidden layer should consist of, which activation function that should be used in each layer, the learning rate and the size of how many images should be processed in each bulk.

References

Blum, A. (1992), Neural Networks in C++, NY: Wiley.