IT3105 Module 5

Deep learning for image classification

This report is written for the 5th module in the course IT3105 Artificial Intelligence Programming at the Norwegian University of Science and Technology. The task was to use Theano, a numerical computation library for Python, to build deep artificial neural networks capable of classifying handwritten digits in the range 0 to 9. The dataset used is the standard MNIST database consisting of 60 000 training images and 10 000 testing images of handwritten digits. The original images are formatted as 28 * 28 matrices of pixel intensities in the range [0, 255], but we preprocess these values by scaling them such that they are in the range [0.0, 1.0].

Design choices

HiddenLayer

A HiddenLayer object takes three arguments; the number of incoming neurons to this layer, the number of neurons in this layer, and an activation function being either hyperbolic tangent, sigmoid, rectified linear units or softmax. The weights for a layer are initialized conditionally by the activation function using the method *init_weights*, which takes the activation function as an argument and produces weights which are uniformly sampled from the interval $\left[-\sqrt{\frac{6}{N_{in}}}, \sqrt{\frac{6}{N_{in}}}\right]$ if the activation function is hyperbolic tangent, $\left[-4\sqrt{\frac{6}{N_{in}}}, 4\sqrt{\frac{6}{N_{in}}}\right]$ if it is sigmoid, $[0.0, 0.1]$ if it is rectified linear units or simply zeroes if it is sigmoid. The output produced by a layer is defined by the following code snippet

```
activation(T.dot(input, self.weights))
```

where input is the output produced at layer *i – 1*.

ANN

An ANN (artificial neural network) object takes five arguments; the number of units in the input layer, a list of hidden layer sizes, a list of activation functions for each layer, the number of units in the output layer, and a learning rate. The ANN object keeps track of a list of layers, which are populated with HiddenLayer objects based on the list of hidden layer sizes and activation functions, and a final HiddenLayer object with softmax activation function to be used as the output layer. This means that no matter what the topology of an ANN object is, the final layer is a Softmax layer no matter what.

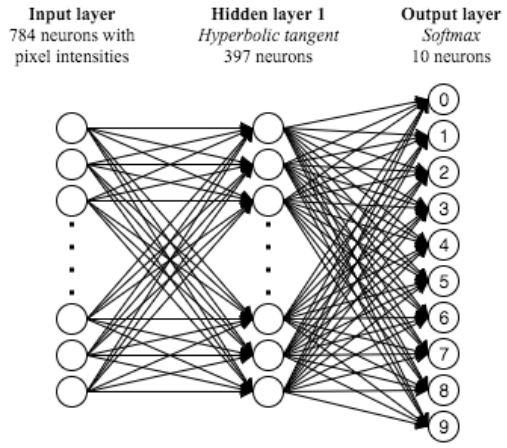
The cost function used is either the sum of squared errors or cross-entropy, depending on the architecture of the neural network that we wish to train. The cost function is fed into our stochastic gradient descent method, which calculates the gradient of the error function with respect to all the weights, and then uses this to update the network's weights in an attempt to minimize the cost function.

Many implementations of neural networks with Theano on the web, including the tutorials by Theano themselves, use optimizations such as L1+L2 or dropout regularization to combat overfitting, and early-stopping in order to obtain good test performance faster. We decided not to use any such optimizations in order to get a deeper understanding of the central aspects of neural network implementations, and we thought that the payoff would be low compared to the effort used to implement it.

Artifical Neural Networks and their architectures

Network 1: 1-Layer hyperbolic tangent NN w/ cross-entropy

This network has only one hidden layer consisting of 397 units, and uses the hyperbolic tangent activation function in this layer. The cost function used is categorical cross-entropy. We found out that a good rule of thumb for the number of hidden units in a 1-layer neural network is $\frac{(num_{in}+num_{out})}{2}$.

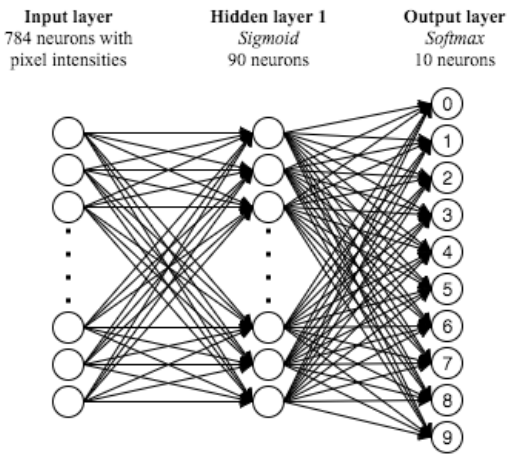


Network 2: 1-Layer hyperbolic tangent NN w/ sum of squared errors

This network is exactly the same as the one described above in *Network 1*, but used the sum of squared errors as its cost function, as opposed to categorical cross-entropy.

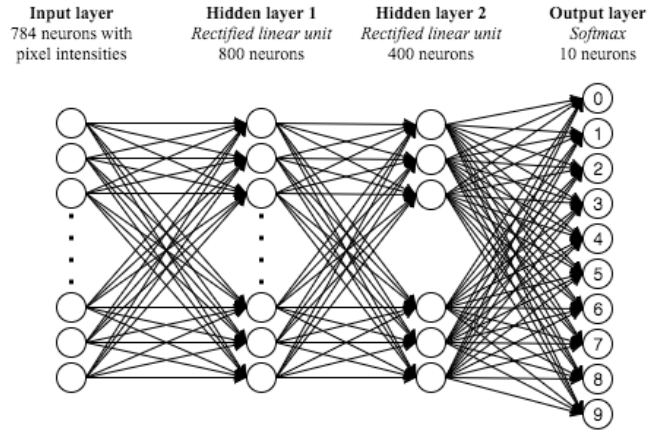
Network 3: 1-Layer sigmoid NN w/ sum of squared errors

This is one of the earliest successful neural networks that we built, and it consists of only one hidden layer with 90 units using the sigmoid activation function. The cost function used is the sum of squared errors.



Network 4: 2-Layer rectified linear unit NN w/ cross-entropy

This network has two hidden layers, consisting of 800 units and 400 units respectively. Both of the hidden layers use rectified linear units as their activation functions. The cost function used is categorical cross-entropy.



Network 5: 2-Layer hyperbolic tangent & sigmoid NN w/ cross-entropy

This network also has two hidden layers, consisting of 500 units and 300 units respectively. The first hidden layer uses hyperbolic tangent as its activation function, while the second uses sigmoid. The cost function used is categorical cross-entropy.

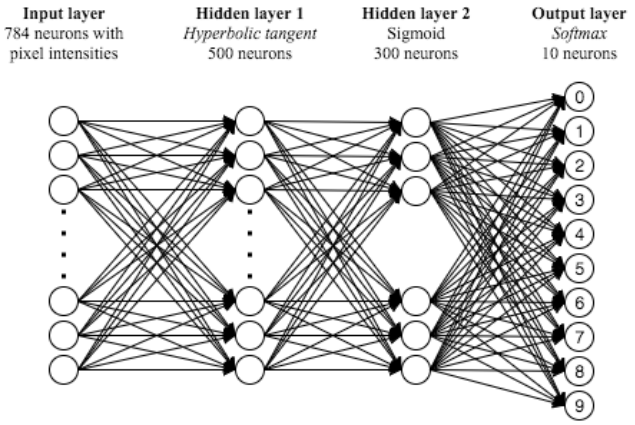


Table of comparison

All training and testing runs are performed on an early 2015 MacBook Pro with a 2.7GHz Intel Core i5 processor and 16GB 1867MHz DDR3-RAM, and we did not use the GPU. The number of epochs iterated and the minibatch size vary for all networks, but were selected carefully so that the networks would train until near-convergence, and so that it did not take a substantial amount of time to train them. In addition, all the networks ran 20 times each on the complete MNIST training and test sets in order to rule out any statistical invariance.

Neural net	**Minibatch size**	**Epochs**	**Learning rate**	**Training set average**	**Test set average %**	**Average time trained**
Network 1	10	30	10^{-2}	99.16%	97.71%	263.57 s
Network 2	100	25	10^{-2}	99.74%	97.95%	131.48 s
Network 3	15	30	$5 * 10^{-2}$	99.73%	97.64%	**95.91 s**
Network 4	20	30	10^{-2}	**99.81%**	**98.10%**	463.20 s
Network 5	15	20	10^{-2}	98.31%	97.34%	244.96 s

In conclusion, all of our five neural networks seem to perform very well on the MNIST dataset. The 2-Layer rectified linear unit neural network with cross-entropy performed best out of all five, reaching an average 99.81% correctly classified training images, and 98.10% correctly classified test images – however, this is also the network that requires the most time to train. Our 1-Layer hyperbolic tangent neural network with sum of squared errors reached nearly the same results, but also required a substantially smaller amount of time to be trained.