CSE641 Assignment 1 : Backpropagation

Anshuman Suri (2014021)

anshuman14021@iiitd.ac.in

Abstract

The assignment requires implementing backpropagation for neural networks from scratch. The resulting framework should include support for Sigmoid and ReLU activation functions, along with Dropout. The framework has been implemented in *python*, with *numpy* for matrix related computations.

1. Data-Set

The data-set used for training is the MNIST dataset. It contains a set of 28x28 images, which are handwritten numerical digits. The data-set has been loaded using the *keras* library for easy parsing. The value pixel values are normalized to bring it down in a [0,1] range.

A total of 60,000 images are used for training, while the remaining 10,000 images are used for testing.

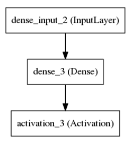
2. Network Architectures

The possible hyper-parameters are for the constructed network are: dropout, learning rate, momentum rate, batch size. Using grid-search (and testing over cross-validation), hyperparameters were selected. For the architecture for the multi-layer perceptron model was selected by hit and trial, trying out siffereng number of hidden neurons (100, 250, 500). The performance comes out to be almost the same for all three, thus one of them was picked arbitrarily. The activation functions tried out were: ReLU and SoftMax. SoftMax gives better performance than the other one.

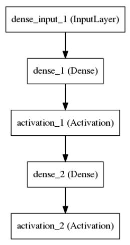
Each batch was run for a maximum of 50 epochs, and was terminated if the error between two consecutive epochs was less than 0.1%

2.1. Single Layer Perceptron

For a single layer perceptron, the network consists of a 784 node input (which is the 28x28 image flattened into a 1-d array). This is passed through a soft-max to get 10 nodes, from which this output is compared with the actual outputs. The hyperparamteres that work best for this architecture are (found using grid search):



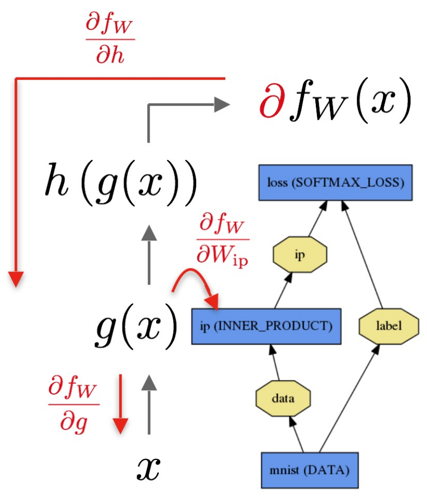
2.2. Multi Layer Perceptron


dense_input_1 (InputLayer)
dense_1 (Dense)
activation_1 (Activation)
dense_2 (Dense)
activation_2 (Activation)

For a multi layer perceptron, the network consists of a 784 node input (which is the 28x28 image flattened into a 1-d array). This is passed on to a hidden layer with 100 neutors (with a softmax activation layer), which is then passed through a soft-max layer to get 10 nodes, from which this output is compared with the actual outputs. The hyperparamteres that work best for this architecture are (found using grid search) are given in the section towards the end.

3. Backpropagation Algorithm

While training a model for a given data-set, the weight parameters need to be adjusted according to the error, so that our model may learn patterns inherent in the data.


$\frac{\partial f_W}{\partial h}$
$\partial f_W(x)$
$h(g(x))$
loss (SOFTMAX_LOSS)
$\frac{\partial f_W}{\partial W_{\text{ip}}}$
ip
$g(x)$
ip (INNER_PRODUCT)
label
$\frac{\partial f_W}{\partial g}$
data
x
mnist (DATA)

The diagram here shows in brief what happens in the backward pass of an iteration while we train a network. FOr this specific model, the input is passed throug ha composition of functios, till we finally get the predicted output from the model. Since we have the actual output associated with this input, we can calculate the error for this output (using our loss function, defined by us).

The ultimate goal while learning is to adjust the weights across all layers such that the final error is minimized. Thus, we need the derivative of each parameter with respect to the final output, so that updates to these weights can be made accordingly.

At any layer, all e have is the input to that layer and the output generated by that layer. To get the gradient of the final loss with respect to this layer, we use the chain rule for derivatives to calculate the derivatives we want. This is called *back propagation*, as we are effectively back propogating the error for a sample poit using the chain rule. This can be formulated mathematically as:

A given node which take sinput from other nodes and gives output to other nodes as well. Consider a specifc input edge, with a weight associated with it as w Let i be the input value computed considering all the input edges, and o be the output after applying an appropriate activation function. We have partial gradients being back propogated form ahead, and we need gradient with respect to the weight w. Then:

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial o} * \frac{\partial o}{\partial i} * \frac{\partial i}{\partial w} \tag{1}$$

Here, the first term on RHS is backpropagated from the layer in front of it. The second term is nothing but the derivative of the activation function at that input node. Finally the

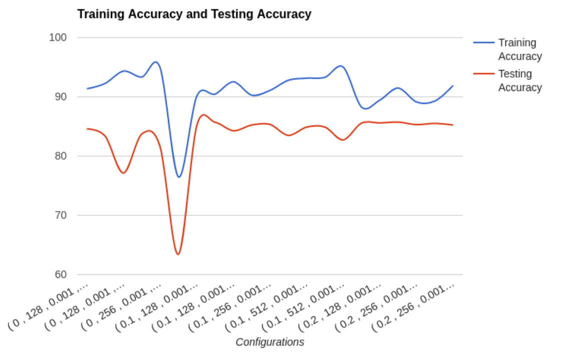
third term is the gradient if the input with respect to the weights at this node.
Using this formula and using the rule of partial derivative tree rule (if required, not needed in our archtecture), we can obtain the partial gradients at each node, and then perform updates accordingly to trin our network.

4. Analysis

4.1. Hypermarameter Selection

A grid search was performed over the following values:

- Dropout: [0.0, 0.1, 0.2, 0.4]
- Batch Size: [128, 256]
- Momentum Gamma: [0.125, 0.25, 0.5]
- Learning rate: [1e-2, 1e-3, 1e-4, 1e-5]

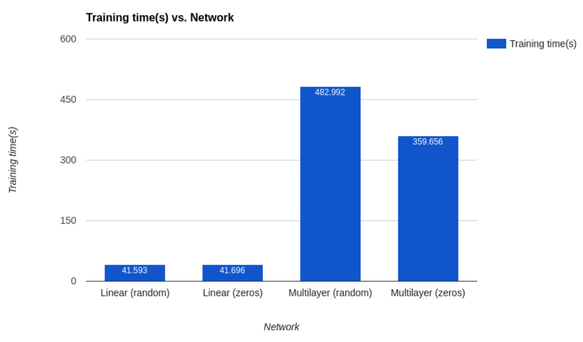


Performing grid search over these values took nearly 40 hours on a 40 core machine to run to completion. Some of the configurations (the ones which yielded a testing accuracy of greater than 80%) are show in the plot here on the left.

Other values from all of the possible grid permutations are not plotted here, as their accuracies were too low, close to random-prediction accuracy. As it is visible from the graph here, the best configuration out of all the hyperparameters (taking into account testing accuracy) for the multi-layer model is:

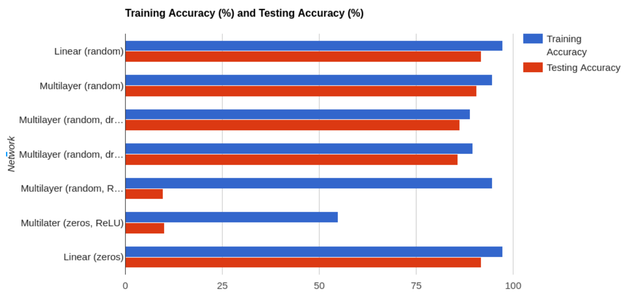
Dropout: 0.1, Batch size: 256, Learning rate: 0.001, Gamma 0.25

4.2. Accuracy and Time Analysis



The time taken to train the model is almost the same in the case of a lienar model, whether we use all zeros as the beginning weigths, or a randomly initialized matrix. However, the time taken to train the multilayer model is significantly diffrent for the two different initialization methods. A multilayer network initialized with all zeros in this case trains faster than a network initialized with random weights. This could be because the actual weights that best apprximate the input distribution are closer to zero (though intuitively one would expect random initialization to perform better).

The training and testing accuracy accuracies are plotted on the right. Over all the tried models, linear architectures seem to perform as good as multilayer ones (better mostly, in fact). The sigmoid activation function seems to be the better option in comparison to ReLU, which performs horribly on the test data. Adding dropout should increase the test error in case of overfitting, but doesn't seem to do so. Thus, the multilayer network is not overfitting.



Overall, a single layer network initialized with zero weights seems to perform best over all of these, giving a training accuracy of 97.25% with a test accuracy of 91.75%.

5. References

- Back propagation diagram from Caffe's documentation
- Karpathy's tutorial to better understand back propagation and it's implementation.