Neural Network Assignment 6 Report

Sreeraj Rimmalapudi

Neural network architecture - Neural network with 3 layers of 1000, 250, 10 neurons achieved best performance on both the validation and test with test accuracy reaching 97.79% on average. Increasing the number of layers to 4 - 1000, 500, 100, 10 neurons, gave 97.25% accuracy on the test set. On reducing the network to 1 hidden layer with 1000 neurons and 10 output neurons, the performance dropped to 97.53%.

Inputs - The inputs were normalized as suggested in the paper[4]. The network was more stable with these inputs and the accuracy on all sets increased by 0.1%.

Labels - The output labels were encoded as vectors of size 10 with [1 0 0 0 0 0 0 0 0 0] representing 0, [0 1 0 0 0 0 0 0 0 0] representing 1 and so on.

Initial Weights - The weights were uniformly chosen at random between [-0.001, 0.001]. Weights 10 times larger than this caused overflow in exponential term and weights 10 times smaller caused gradient to be very small and hence no update.

Activation Function - The activation function for each neuron was ReLU, defined as f(x) = max(0,x). It gave nearly 2% improvement on accuracy and faster convergence than hyperbolic tangent function. Softmax function given by $\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$ was used in the final layer to treat the outputs as probabilities where z is the weighted input to the neuron.

Loss Function - Cross entropy or log loss was used as the loss function. The loss function is given as $\frac{1}{n} \sum_{i=1}^{n} -\mathrm{d}_i \log(y_i)$ - (1-d_i) ($\log(1 - y_i)$) where d is the desired output and y is the output of the network.

Mini batch update - Mini batches were used[3]. The batch size was set to 200.

Learning rate - Learning rate was set to 0.001. It was constant for each of the weights throughout the entire training process. Increasing the learning rate to 0.01 caused the network to sometimes diverge with no increase in accuracy. There was a plan initially to decay learning rate after certain epochs[3] but it performed well even without the decay.

Adaptive Momentum Estimation (Adam) Update - Adam update was used. The parameters for the update were set as recommended in the paper[1].

Epochs - Epochs were initialy set to 100. On closer inspection of the loss function, increasing the epoch would have had no effect. It was decreased to 25 the point after which no drop in error rate was observed.

Dropout - Dropout[2] was used with drop probability set to 0.5 for each neuron.It was not used in the input and output layers

Regularization - L2 regularization or weight decay was used to limit the weights from growing large. The regularization parameter was set to 0.001. Lower values caused problems with logarithm.

References

[1] Adam: a Method for Stochastic Optimization. Kingma, D. P., Ba, J. L. (2015). International Conference on Learning Representations, 1–13.
[2] Dropout: A simple way to prevent neural networks from overfitting. N. Srivastava, G.E. Hinton, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. The Journal of Machine Learning Research, 15(1):1929–1958, 2014.
[3] Practical Recommendations for Gradient-Based Training of Deep Architectures Yoshua Bengio. In Neural Networks: Tricks of the Trade, pg. 437–478. Springer, 2012
[4] Efficient BackProp, LeCun, Y., Bottou, L., Orr, G. B., and Muller, K., Tricks of the Trade (1998a).

Description of Algorithm - The algorithm takes a modular approach and performs the following operation -

1) Perform Forward Propagation and store values for backprop
2) Get loss
3) Perform Back Propagation and get the gradients
4) Update the weights using the gradients

The forward and backward propagation is performed layer by layer in forward and backward step function respectively.

The algorithm mainly uses mini batch learning with fixed batch size. It utilizes regularization, dropout, early stopping and weights updating using Adam method techniques to improve performance.

Avg. Test accuracy - 97.79%
Best Test accuracy - 97.9%
Time to train - Approximately 20 mins

Algorithm 1 Neural Network Mnist Classification

function MAIN
    Get the inputs and divide the data into training, validation and testing set
    Initialize weights, biases and set the hyper parameters
    set patience = 0
    for *epoch* = 1 to 25 **do**
        Divide the training data into batches each of size 200 ▷ Mini batches
        for each batch b_i **do** ▷ Training
            Get batch output of b_i by performing forward propagation
            Calculate cross entropy loss
            Back propagate the gradients
            Update the weights using Adam update
        Forward propagate the entire training sample and validation sample and get output **y** and **v**
        Calculate training, validation loss and accuracy ▷ Validation
        if validation accuracy increases **then**
            Store the parameters
        else
            Decrease the patience
        if patience = 0 **then**
            End training ▷ Early stopping

function FORWARD PROPAGATION
    for each layer l **do**
        Let x be the input of size NXD, where N is the batch size and D is the number of neurons in previous layer
        Let w be the weights of size DXH (H is the number of hidden neurons of layer l)
        Let b be the bias of size 1XH
        if l is not the last layer **then**
            Perform Forward step with x,w and b
        else
            Perform softmax transformation with x,w and b
        Store values for backward step

function FORWARD STEP(x,w,b)
    Perform affine transformation, out = xw + b
    Use ReLU transformation, reluout = max(0,x)
    With probability of 0.5 drop the output,ie, make it zero
    Store x,w,out

function SOFTMAX TRANSFORMATION(x,w,b)
    Perform affine transformation, z = xw + b
    for j = 1..10 **do**
        $out_j = \frac{e^{z_j}}{\sum_{k=1}^{10} e^{z_k}}$
    out is the network output of size NX10 giving probabilities of each class

Algorithm 1 Neural Network Mnist Classification (continued)

function BACKWARD PROPAGATION
    for each layer l in reversed order **do**
        if l is the last layer **then**
            Perform backprop of softmax function
        else
            Perform backstep to calculate gradient

function SOFTMAX BACKPROP
    Let t be the desired label with one hot encoding
    Let *out* be the network output
    $delta = out_i$ - t_i ▷ Softmax derivative
    $dw = x^T\ delta$ ▷ Derivative of loss wrt weight w
    for each hidden neuron k **do**
        $db_k = \sum_{i=1}^{n} delta_{i,k}$ ▷ Derivative of loss wrt bias k
    delta for previous layer = $delta\ w^T$

function BACKWARD STEP
    Get the stored values from forward propagation
    if neuron was dropped **then**
        Gradients of the weights connected to that neuron is 0
    else
        If the input value before applying ReLU in forward prop was less than zero, kill the gradient at that location ▷ ReLU derivative
        $dw = x^T\ delta$ ▷ Derivative of loss wrt weight w
        for each hidden neuron k **do**
            $db_k = \sum_{i=1}^{n} delta_{i,k}$ ▷ Derivative of loss wrt bias k
        delta for previous layer = $delta\ w^T$

function CROSS ENTROPY LOSS
    Let *out* of size NX10 denote the network output representing probability of each class
    Let t_i denoted the desired label for each input of the batch
    Let W denote the network weights
    loss = - $\sum_{i=1}^{N} log(out_{i\,t_i}) + \frac{\lambda}{2} * \|\mathrm{W}\|^2$