Machine Learning - MLP on MNIST Dataset and Regularization

Shivani Sharma
Faculty of Computer Science, Dalhousie University
6050 University Ave, Halifax, Nova Scotia, Canada, B3H 4R2
Sh477091@dal.ca

Abstract

In this paper, we are constructing an MLP for the MNIST dataset, describing its usage on the MNIST Handwritten digits dataset to predict the ASCII Value.Our main aim in this paper, is to discuss the approach to build fully functional MLP using basic python and Numpy.We compare different models using one hidden layer. We also trained our model for different learning rate and the most suitable learning rate is 0.1. Keras has been used just to import the MNIST dataset. No third party libraries have been used. The learning curves for this dataset are represented, giving the final test data classification accuracies. Two of the regularization techniques, Ridge and LASSO are used in order to avoid the over-fitting of the data examples. Additional noise has been added to check the robustness of the model.

1. Introduction

Multi-Layer Perceptron (MLP) is a class of feed-forward Artificial Neural Networks (ANN). MLP typically performs well at speech recognition, image recognition and machine translation applications. Multilayer perceptrons are sometimes colloquially referred to as "vanilla" neural networks, especially when they have a single hidden layer. An MLP consists of at least three layers of nodes: an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. The input layer is passed through the hidden layer to the output layer. The output values of the nodes in the hidden layer become the input to the outer layer, which are then scaled by the values of the connection strength as specified by the elements in the weight matrix wo - which is the weight between the output and the hidden layer.

I have also applied backpropagation, activation and regularization to my MLP. In machine learning, **backpropagation** is a widely used algorithm in training feedforward neural networks for supervised learning. Generalizations of backpropagation exist for other artificial neural networks (ANNs). In fitting a neural network, backpropagation computes the gradient of the loss function with respect to the weights of the network for a single input-output example. As the name suggests, the output values are passed back through the previous layers to calculate the

descent. Neural networks always face an issue of over-fitting of data, which here is resolved using the L1 and L2 regularization. After that model was trained with final parameters and predictions were made on the test set. Later, we added noise to the test data to check the variance in performance of the model.

2. MNIST dataset

The MNIST dataset consists of 70,000 images of handwritten digits, out of which 60,000 are used for training and 10,000 are used for testing purpose. Each instance is 28x28 pixel grayscale image. To reduce the computational load on the model so that it would compute in a reasonable time frame, a reduced dataset of length **N=1024** was taken for both training and test datasets. The results included in this report are thus the result of training and testing on 1024 data examples. We needed a 1-D vector for input layer. Therefore, we re-shaped each image as a single vector. Fig 1 shows the example of Raw MNIST

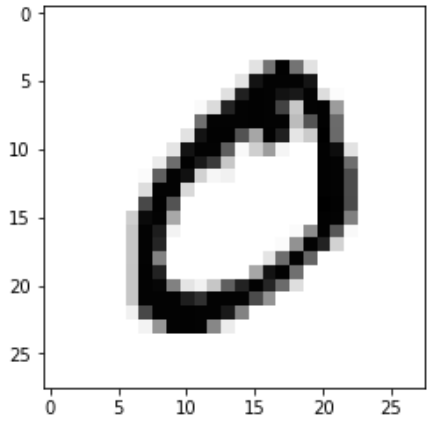


Fig 1: Raw mnist data

As in this experiment we are supposed to convert the hadnwrtitten digits their respective ASCII values, I have here created a **data-dictionary**:

dict = {0: 48, 1: 49, 2: 50, 3: 51, 4: 52, 5: 53, 6: 54, 7: 55, 8: 56, 9: 57}

This dictionary is used to convert the 0-9 binary values to ASCII. After conversion our input matrix is of shape (8x1024). Y_train and Y_test data example are converted to ASCII using this, resulting in an array and having datatype conversion - **dtype=np.uint8.**

After which I have converted the ascii character codes to binary, for which I have used unpackbits along with a incrementing counter. **numpy.unpackbits()** is an another function for doing binary operations. It is used to unpack elements of a uint8 array into a binary-valued output array. The resultant output array is then transposed.

3. Building the Network

- Backward Propagation:

While training a neural network, we tweak the weights of the neuron connections such that the prediction accuracy is increased. First, we calculate the error after feeding the input signals forward, and then we backpropagate that error through the layers i.e we calculate gradient descent of the error function over the weights and then use that information to update the parameters.

There are a number of ways to calculate the error functions such as cross-entropy, absolute error, and mean squared error. Error here is calculated between the expected outputs and the outputs forward propagated from the network. These errors are then propagated backward through the network from the output layer to the hidden layer, assigning blame for the error and updating weights as they go.

- Forward Propagation/ Feed Forward:

For the first layer the activations are basically just the inputs. After which - we need to loop through all the weight matrices i.e. looping through all the layers in the network, where we calculate the net activations. For activations, I am using sigmoid function here. The main reason why we use sigmoid function is because it exists between (0 to 1). Therefore, it is especially used for models where we have to predict the probability as an output.Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice. Also it is differentiable. That means we can find slope of the sigmoid curve at any two points. The activation functions are created for all the training and testing data.

- Delta Learning Rule:

The Delta Rule uses the difference between *target activation* (i.e., target output values) and obtained activation to drive learning. During forward propagation through a network, the output (activation) of a given node is a function of its inputs. The inputs to a node, which are simply the products of the output of preceding nodes with their associated weights, are summed and then passed through an activation function before being sent out from the node.

4. BASELINE MODEL

To train the MLP network, we need to update the weights with the momentum = 0.9 and learning rate as 0.1. Our initial network consists of three layers, input layers with 784 nodes, hidden layer with 10 nodes and output layer with 8 nodes. We decided on using the sigmoid activation function for hidden as well as output layer.

We then introduce and initialize some variables which now include the activation of the hidden nodes h and the weights to the hidden nodes wh, as well as the corresponding gradient dwh and delta term dh. We then iterate over the trials. In this experiments it is over 1000 trials. We used

Absolute error for computing loss metric. We also evaluated accuracy metric for evaluating the performance of the model.

5. MODEL IMPROVEMENT

Ridge and Lasso regression are powerful techniques generally used for creating parsimonious models in presence of a 'large' number of features. Ridge performs Performs L2 regularization, i.e. adds penalty equivalent to **square of the magnitude** of coefficients while LASSO Performs L1 regularization, i.e. adds penalty equivalent to **absolute value of the magnitude** of coefficients. Ridge regression is quite similar to RSS except that there is also a shirnkage parametre 'λ' that minimizes their value. 'λ' is also called as 'tuning parametre' and it is determined separately using cross-validation technique. To overcome the problem that ridge has, Lasso s an alternative that can pick relevant features that will be useful for modelling. Lasso also has the shrinkage parametre but the difference that has with Ridge is that there is no squared term of the estimated coefficient but only an absolute value.

We have also tried to perform the drop-out regularization, for which I have created a drop-out matrix which is then applied to the hidden layers, to avoid over-fitting.

6. RESULTS

5.1 MLP

After training the MLP model the following learning curves were obtained. The final achieved accuracy was approximately 98% on 1024 test data examples.

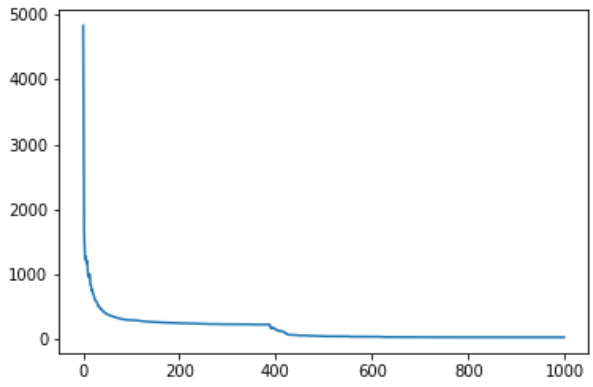
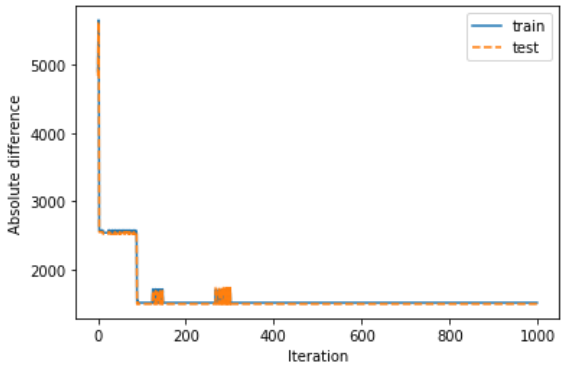
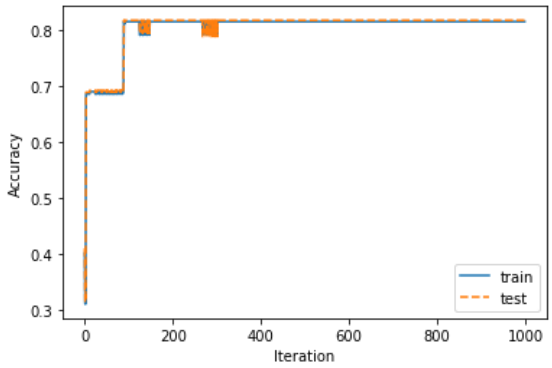


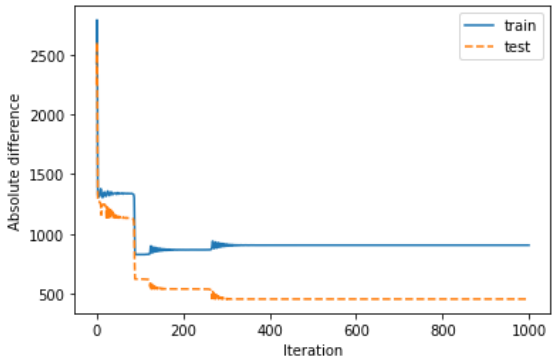
Fig 2: MLP learning curve

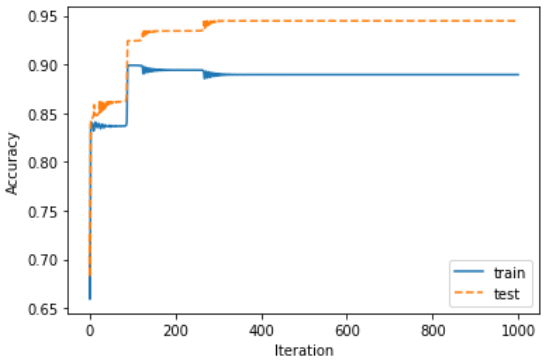
5.2 MLP with L2 Regularization





5.3 MLP with L1 Regularization





References:

Trappenberg, T. (2020). Fundamentals Of Machine Learning. Oxford: Oxford University Press.

LeCun, Y., Cortes, C. & Burges, CJ. (2010). Mnist handwritten digit database. AT&T Labs [Online]. Available at: http://yann.lecun.com/exdb/mnist [accessed 15th October 2020]

Gupta, P. (2020). Regularization in Machine Learning. Medium. Retrieved 19 October 2020, from https://towardsdatascience.com/regularization-in-machine-learning-76441ddcf99a. [accessed 19th October 2020]

Only Numpy: Implementing Different combination of L1 /L2 norm/regularization to Deep Neural Network…. Medium. (2020). Retrieved 23 October 2020, from https://towardsdatascience.com/only-numpy-implementing-different-combination-of-l1-norm-l2-norm-l1-regularization-and-14b01a9773b. [accessed on 20th October 2020]

J, D. (2020). Lasso and Ridge Regularization. Medium. Retrieved 23 October 2020, from https://medium.com/all-about-ml/lasso-and-ridge-regularization-a0df473386d5. [accesed on 22nd October 2020]