Adaptive Intelligence Assignment 1: Supervised learning of handwritten letters

Wenjie Zhang
The University of Sheffield,Sheffield,United Kingdom

***Abstract*—This report focuses on the implementation of a supervised learning algorithm – an artificial neural network (ANN) based multi-layer perceptron with Mean Square Error, which is used to classify images. Regularization is done by adding penalty term L1. The data set used in this report is EMNIST, but samples per class restrict to 1000 for training and 250 for testing. For validation, we further split the train set into train set and development set with portion 8:2.**

***Index Terms*—EMNIST, MLP, ANN network, MSE, L1,**

I. Introduction

Supervised learning is a task of learning a function that maps inputs into output based on previously seen examples. It deals with two types of problems – classification and regression problems. This report mainly focuses on the classification problem, our objective is to find the pattern in the data that correlate with desired output and hence predict correct labels for unseen input data. In order to do it, there are some key ingredients: desired output(label) need to be provided along with the input, there has to be some feedback for the predicted output (i.e. how well the results against desired output), offline analysis is used rather than real-time analysis.

The process of training a classifier using supervised learning usually contains 4 stages: 1. Input raw data with corresponding desired output(label). 2. Using the algorithm to characterize input and according to features to predict outputs. 3. Using provided desired output to measure how well the algorithm is and generate errors(feedback). 4. According to the errors(feedback), tune parameters in the algorithm to minimize the error and repeat the process to reach the optimum results.

II. Methods

A. Data Normalisation

As most commonly in machine learning, the features of data are measured at different scales and do not contribute equally to model fitting, model learning might end up creating a bias and take long time to learn and convergence. Thus, normalization is an important step for machine learning, which is especially useful for MLP. As we are doing back-propagation in MLP, normalization offer more stability and faster speed during the process. In this report, we choose Min-Max normalization, comparing to Standard deviation it converges more quickly and the process is faster. The reason for it is Min-Max has better distribution, as the data is pixel, value from 0-255, most value is 0. The difference between max and min is large and mean usually small.

B. Weight Initialisation

Besides normalization, weight initialization also affects the process of learning to a large extent. The objective of weight initialization is to prevent gradient exploding or vanishing. Gradient exploding means changes in nth layer quicker than n-1th layer, which leads to greater weight and gradient exploding, usually appears in multi-layer network and large initial weight. Gradient vanishing means latter layer (i.e. input layer to first hidden layer) times lots of ¡ 1 numbers, which leads the results to 0, thus weight does not update, usually appears in multi-layer network and unsuitable activation function, such as sigmoid. Both due to the process of backpropagation. Desired weight initialization should have a mean value equal to 0. Xavier initialization initializes weight from a standard normal distribution and then scale by square root of 1/n. The most important factor here is square root 1/n, which adjusts variance and leads to faster convergence. Comparing to other methods of weight initialization, such as uniform distribution (0,1) only considering variance of input layer, Xavier considering variance of input and output layer and try to keep the same for each layer. If we initialize weight to zero or fixed value, the model may fail (no convergence). Thus, in this report, we choose Xavier initialization. But He initialization may be a better way to initialize for this report since we are using RELU activation function, x¡0 x=0.

C. Activation Function

The activation function enables the network to solve non-linear problems, it's a crucial element for neuron networks. Here we take sigmoid, tanh, and RELU as examples of activation functions to comparing. Sigmoid function has advantages, outputs are sensitive to input, and parameters updates more stable. But it has drawbacks, 1. no matter the value of inputs, the derivative also less than 0.5, and for large inputs the derivatives approach to 0 which may lead to gradient vanish. 2. The parameters of the layer are all updated in the positive or negative direction, which leads to slow convergence. 3. Long runtime since there is an exponential function. Another activation function tanh, it's quite similar to Sigmoid but it avoids the second drawback of Sigmoid. The activation function used in this report is RELU function, since the calculation is much simpler, faster convergence. Also, for positive numbers, the derivative of RELU is always 1 which avoids gradient explode. However, it also has some drawbacks, such as the derivatives always 0 for negative numbers, which means it unable to pass gradient for negative values. The

neuron with negative values won't update and propagate to n-1th layer, which is known as the dead RELU problem. We can avoid this problem by decreasing learning rates so that fewer dead neurons, a small amount of dead neurons won't affect too much.

D. Brief description of algorithm

Classification problems can be solved with numerous amounts of algorithms, such as Linear Classifiers, Support Vector Machines, Decision Trees, K-Nearest Neighbour and Random Forest, etc, choices of algorithms depend on the data and situation. In this report, as we are going to classify images of handwritten letters, the algorithm chosen is an Artificial neural network (ANN) based multi-layer perceptron. To build a more complex ANN, we need to build a basic single-layer network (perceptron). Perceptron includes input layer (pre-synaptic neurons), output layer (post-synaptic neurons), activation function, and weight from each pre-synaptic neuron to each post-synaptic neuron. Each input layer neuron match one pixel in data (sample data shape 1*784, each sample has 784 pixels), each output layer neurons match to one letter class (26 in total), weight between each input neuron, and each output neuron is initialized by Xavier initialization, activation function used is RELU. For training model, first feed-forward input (input x -¿ h= (w,x) -¿ y = RELU(h)) and calculate the error between output and desired output by MSE. Then do backpropagation use the error to adjust our weight. During this process, we need to know how weight changes affect error, so dE/dW. Since error related to RELU(h), h related to w and x, by chain rule we get dE/dW = dE/dY * dY/dH * dH/dW. Since we are using Gradient descent, dW = -learning rate * derivative of MSE * derivative of RELU * x, the negative sign absorbs in derivative of MSE, we can define derivatives of MSE * derivative of RELU as our local gradient. Thus, dW simplifies to learning rate * local gradient * input, then uses dW to adjust our weight. Since dY/dH is derivative of RELU equal to x¡0 x=0 x¿0 x=1, but x=0 is undefined in RELU derivative. We define derivative x=0 as equal to 1. In this report, we implement 2 hidden layer networks, using matrix instead of for loop to access data and update weights so that run time is decreased significantly.

III. Results and Discussion

A. Question 3,4

Single layer network model is our baseline for comparing performance between different number of hidden layers. As the Fig.1 shows, the weight update after 400 epochs is con verged close to 0. In [1] paper, the author achieves 55.78% accuracy on the EMNIST Letter data-set by linear classifier by contrast, our model shows accuracy around 63% +- 10% This may due to Min-Max is used and runs 400 epochs.

B. Question 5,6

Overfitting usually occurs when lots of noises in training data, during training the model will learn these noises to decrease the error. The reason for this is model is too complex

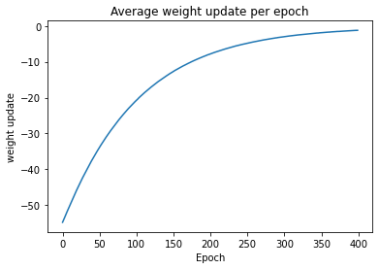

Fig. 1. weight update convergence.

with parameters more than the number of input patterns. This leads to decrease performance on unseen data, but our objective is a generalized model. Regularisation can prevent overfitting (see Fig.2) since it reduces variance problems which is essential in machine learning. Thus, regularisation methods need to be used.

There are plenty of methods of regularisation that stop overfitting, such as Early Stopping, Data Augmentation, Dropout, and L1, L2 which we used in this report. Dropout prevents overfitting by randomly ignore half of the hidden units during training, simply change the network during training. Data Augmentation preventing overfitting from more data samples, it changes original data to get more data, such as rotation, add noise, crop part of data (i.e. 784 pixels to 300 pixels). Early stopping is stopping the learning process before it reaches optimum. There is a point with the lowest error on training and unseen data, which is usually before the lowest error on training data. Early stopping is by stopping at that point to prevent overfitting. L1, L2 is used to reduce the complexity of our model by adding a penalty term to the MSE. Both are preventing overfitting by decreasing dW. There are differences between L1(Fig.3) and L2(Fig.4), we can see it though differentiate them1. dW decrease by a constant in L1 while dW decrease as proportional to W. 2. When there is large —w— L1 decrease smaller than L2, small —w— L1 decrease larger than L2. In this report, L1 is used according to the data and difference between L1 and L2 mentioned.

Since L1 can be seen as square root of square of w, the derivative is Fig.5 . So, basically we are adding a - learning rate * lambda * sign(w) to dW, which is a constant. This penalty term does aid our learning decreasing validation error as Fig.6 and Fig.7. The optimum value of lambda found is 0.00003

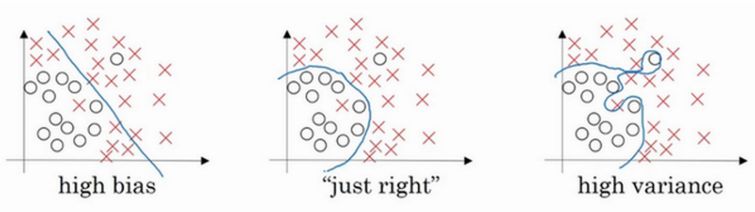

Fig. 2. variance effect.

$$L(x,y) \equiv \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2 + \lambda \sum_{i=1}^{n} |\theta_i|$$

Fig. 3. adding L1.

$$L(x,y) \equiv \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2 + \lambda \sum_{i=1}^{n} \theta_i^2$$

Fig. 4. adding L2.

$$|x|' = \frac{1}{2\sqrt{x^2}} \cdot 2x = \frac{x}{\sqrt{x^2}} = \frac{x}{|x|}$$

Fig. 5. adding L2.

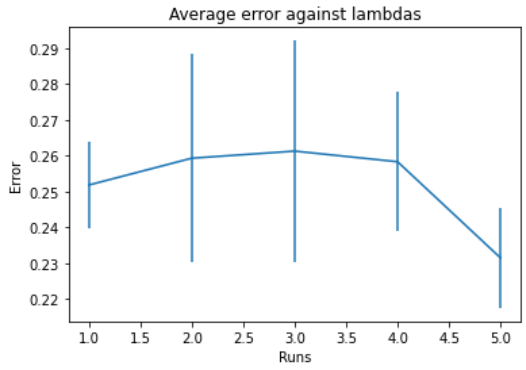

Fig. 6. without L1.

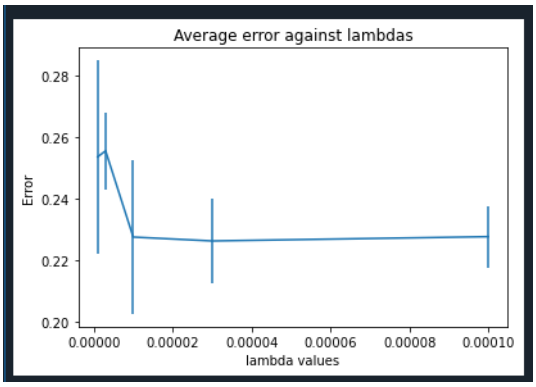

Fig. 7. with L1.

C. Question 7

In this report, we've tested single hidden layer neuron with number 50,100,200,400, the results is Fig.8. The error bar used is mean and standard deviation. This result may slightly different from running code, since i forgot to change data normalization method to Min-Max, n/255 is used in this result. Also, due to long runtime, epoches is 100 in these runs although matrix operation is performed to cancel inner for loop. Expect higher accuracy for longer time(converge).

As we can see from graph, the result is increased as number of neurons increasing in hidden layer. The reason for this might be, more neurons in hidden layer means it can remember more patterns between input and desired output In general, the result of adding a hidden layer is better than perceptron.

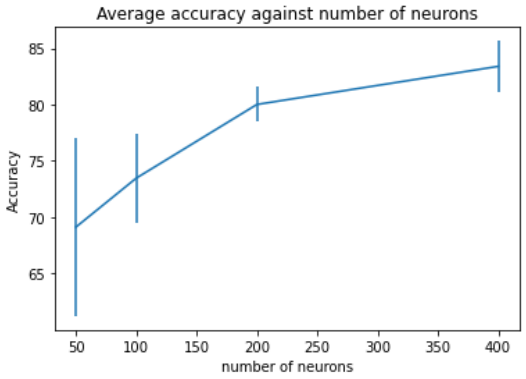

Fig. 8. question7.

D. Question 8

In this report, we've tested second hidden layer neuron with number 50,100,200,400(fixed first hidden layer neuron number = 100), the results is Fig.9. The error bar used is mean and standard deviation.

Again we can see a increasing trend as number of neurons increasing, same reason for question7. In general, more hidden layer and number of neurons both produce higher accuracy. But there is a upper bound, since there is noise in data. Also, hand written letters is a hard task to classify.

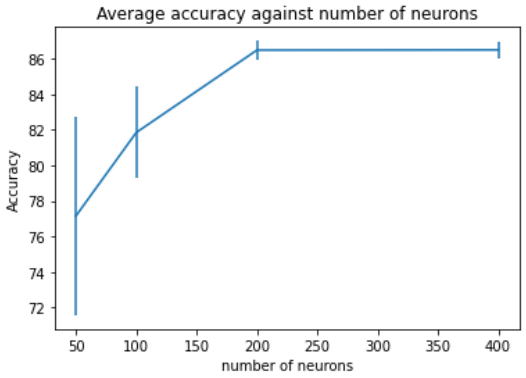

Fig. 9. question8.

ACKNOWLEDGMENT

This report used part of code from lab2.

There is 5 code file in total for different question

APPENDIX: EXAMPLE CODE FOR QUESTION8

normalize data by min-max Normalization $x_train = np.array([(samples - min(samples))/(max(samples) - min(samples))for samples in x_train])normalised_train = x_trainx_test = np.array([(samples - min(samples))/(max(samples) - min(samples))for samples in x_test])normalised_test = x_testrandomisedata with corresponding label randomised_train, randomised_train labels = shuffle(normalised_train, train labels, random_state = 0)randomised_test, randomised_test labels = shuffle(normalised_test, test labels, random_state = 0)$

split train data into dev set 8:2 $train_dev_percent = int(0.2 * randomised_train.shape[0])randomised_dev = randomised_train[: train_dev_percent]randomised_dev_labels = randomised_train labels[: train_dev_percent]randomised_train_set = randomised_train[train_dev_percent :]randomised_train_set_labels = randomised_train labels[train_dev_percent :]$

$n_samples, img_size = randomised_train.shape$

The EMNIST contains A-Z so we will set the number of labels as 26 nlabels = 26

transfer labels into array, index of eliment equal to 1 is the label of that sample $y_train = np.zeros((randomised_train labels.shape[0], nlabels))y_test = np.zeros((randomised_test labels.shape[0], nlabels))$

for i in range(0,randomised$_train labels.shape[0]) : y_train[i, randomised_train labels[i].astype(int)] = 1$

for i in range(0,randomised$_test labels.shape[0]) : y_test[i, randomised_test labels[i].astype(int)] = 1$

The number of epochs is a hyperparameter that defines the number times that the learning algorithm will work through the entire training dataset.

The batch size is a hyperparameter that defines the number of samples to work through before updating the internal model parameters.

ref: https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/

$n_epoch = 250 batch_size = 50100 n_batches = int(math.ceil(n_samples/batch_size))$

define the size of each of the layers in the network $n_input_layer = img_sizen_hidden_layer = 100 number of neurons of the hidden layer.0 delete this layer n_output_layer = nlabels$

Add another hidden layer $n_hidden_layer2 = 0100 number of neurons of the hidden layer.0 delete this layer$

eta is the learning rate eta = 0.05

set lambda value lambdalist = [0.000001, 0.000003, 0.00001, 0.00003, 0.0001] lamda = 0.00003 how many runs $different_run = 5 record validation error validation_error = np.zeros((len(lambdalist), different_run))$

number of neuron $num_neuron = [50, 100, 200, 400] record accuracy accuracy_list = np.zeros((len(num_neuron), different_run))$

for question6 for $num_neurons in range(0, len(num_neuron)) : for num_tries in range(0, different_run) : Initialize a simple network For W1 and W2 columns are the input and the Number of columns(input) needs to be equal to the number of features of Number of columns(input) needs to be equal to the number of neurons of t$

$n_hidden_layer2 = num_neuron[num_neurons] Xavier_init = True$

if Xavier$_init : if n_hidden_layer > 0 : W1 = np.random.randn(n_hidden_layer, n_input_layer) * np.sqrt(1/(n_input_layer)) if n_hidden_layer2 > 0 : W2 = np.random.randn(n_hidden_layer2, n_hidden_layer) * np.sqrt(1/(n_hidden_layer)) W3 = np.random.randn(n_output_layer, n_hidden_layer2) * np.sqrt(1/(n_hidden_layer2)) else : W2 = np.random.randn(n_output_layer, n_hidden_layer) * np.sqrt(1/(n_hidden_layer)) else : W1 = np.random.randn(n_output_layer, n_input_layer) * np.sqrt(1/(n_input_layer)) else : if n_hidden_layer > 0 : W1 = np.random.uniform(0, 1, (n_hidden_layer, n_input_layer)) W2 = np.random.uniform(0, 1, (n_output_layer, n_hidden_layer))$

The following normalises the random weights so that the sum of each row =1 W1 = np.divide(W1,np.matlib.repmat(np.sum(W1,1)[:,None],1,n$_input_layer)) W$

$np.divide(W2, np.matlib.repmat(np.sum(W2, 1)[:, None], 1, n_hidden_layer))$

if n$_hidden_layer2 > 0 : W3 = np.random.uniform(0, 1, (n_output_layer, n_hidden_layer2)) W3 = np.divide(W3, np.matlib.repmat(np.sum(W3, 1)[:, None], 1, n_hidden_layer2))$

W2=np.random.uniform(0,1,(n$_hidden_layer2, n_hidden_layer)) W2 = np.divide(W2, np.matlib.repmat(np.sum(W2, 1)[:, None], 1, n_hidden_layer)) else : W1 = np.random.uniform(0, 1, (n_output_layer, n_input_layer)) W1 = np.divide(W1, np.matlib.repmat(np.sum(W1, 1)[:, None], 1, n_input_layer))$

Initialize the biases bias$_W1 = np.zeros((n_output_layer,)) if n_hidden_layer > 0 : bias_W1 = np.zeros((n_hidden_layer,)) bias_W2 = np.zeros((n_output_layer,)) if n_hidden_layer2 > 0 : bias_W3 = np.zeros((n_output_layer,)) bias_W2 = np.zeros((n_hidden_layer2,))$

Train the network for i in range(0,n$_epoch) :$

Initialise the gradients for each batch dW1 = np.zeros(W1.shape) if n$_hidden_layer > 0 : dW2 = np.zeros(W2.shape)$

We will shuffle the order of the samples each epoch $shuffled_idxs = np.random.permutation(n_samples) shuffled_data_set = randomised_train[shuffled_idxs] shuffled_label_set = y_train[shuffled_idxs]$

for batch in range(0,n$_batches) : Initialise the gradients for each batch dW1 =$

$np.zeros(W1.shape)dbias_W1 = np.zeros(bias_W1.shape)if n_hidden_layer > 0 : dW2 = np.zeros(W2.shape)dbias_W2 = np.zeros(bias_W2.shape)if n_hidden_layer2 > 0 : dW3 = np.zeros(W3.shape)dbias_W3 = np.zeros(bias_W3.shape)$

shuffled input idx = shuffled$_idxs[batch * batch_size]$

x0 = shuffled$_data_set[batch*batch_size : batch*batch_size+ batch_size]$

Form the desired output, the correct neuron should have 1 the rest 0

desired$_output = shuffled_label_set[batch * batch_size : batch * batch_size + batch_size]$

h1 = np.matmul(W1,x0.T)+np.tile(bias$_W1, (batch_size, 1)).T Apply the RELU function $x1 = np.maximum(h1, 0)$

if $n_hidden_layer > 0 : Neural activation : hidden layer -> output layer h2 = np.matmul(W2, x1) + np.tile(bias_W2, (batch_size, 1)).T$

Apply the RELU function x2 = np.maximum(h2,0)

if $n_hidden_layer2 > 0 : Neural activation : hidden layer1 -> hidden layer2 h3 = np.dot(W3, x2) + np.tile(bias_W3, (batch_size, 1)).T$

Apply the RELU function x3 = np.maximum(h3,0)

Compute the error signal $e_n = desired_output - x3.T$

Backpropagation: output layer -¿ hidden layer 2 delta3 = $e_n.T * np.where(x3 <= 0, 0, 1)$

dW3 += np.matmul(delta3,x2.T) $dbias_W3 += np.sum(delta3)$

Backpropagation: hidden layer -¿ input layer delta2 = np.where(x2 ¡= 0, 0, 1) * np.matmul(W3.T, delta3) else: Compute the error signal $e_n = desired_output - x2.T Backpropagation : output layer -> hidden layer delta2 = e_n.T * np.where(x2 <= 0, 0, 1)$

dW2 += np.matmul(delta2, x1.T)

$dbias_W2 = np.sum(delta2)$

Backpropagation: hidden layer -¿ input layer delta1 = np.where(x1 ¡= 0, 0, 1) * np.dot(W2.T, delta2) else:

$e_n = np.subtract(desired_output, x1.T)$

delta1 = np.where(x1 ¡= 0, 0, 1) * e_n

dW1 += np.matmul(delta1,x0)

$dbias_W1 += np.sum(delta1)$

After each batch update the weights using accumulated gradients as we can't let 0 undefined, so simply set when x = 0, sign(x)=1 this part code is for no L1 W1 += eta*dW1/batch$_size bias_W1 += eta * dbias_W1/batch_size if n_hidden_layer > 0 : W2 += eta * dW2/batch_size bias_W2 += eta * dbias_W2/batch_size if n_hidden_layer2 > 0 : W3 += eta * dW3/batch_size bias_W3 += eta * dbias_W3/batch_size$

W1 += eta*dW1/batch$_size - eta * lamda * np.where(W1 < 0, -1, 1) bias_W1 += eta * dbias_W1/batch_size$

if $n_hidden_layer > 0 : W2 += eta * dW2/batch_size - eta * lamda * np.where(W2 < 0, -1, 1) bias_W2 += eta * dbias_W2/batch_size if n_hidden_layer2 > 0 : W3 += eta * dW3/batch_size - eta * lamda * np.where(W3 < 0, -1, 1) bias_W3 += eta * dbias_W3/batch_size$

TODO: use the test set to compute the network's accuracy n = randomised$_test.shape[0]$

$p_ra = 0 correct_value = np.zeros((n,)) predicted_value = np.zeros((n,))$

for num in range(0, n): x0 = randomised$_test[num] y = y_test[num]$

correct$_value[num] = np.argmax(y)$

h1 = np.dot(W1, x0) + bias$_W1 x1 = np.maximum(h1, 0) if n_hidden_layer > 0 : h2 = np.dot(W2, x1) + bias_W2 x2 = np.maximum(h2, 0) if n_hidden_layer2 > 0 : h3 = np.dot(W3, x2) + bias_W3 x3 = np.maximum(h3, 0)$

predicted_value[num] $Apply the RELU function x1 = np.argmax(x3) else : predicted_value[num] = np.argmax(x2) else : predicted_value[num] = np.argmax(x1)$

if predicted$_value[num] == correct_value[num] : p_ra = (p_ra + 1)$

accuracy = 100*p$_ra/nprint("number of neuron = ", num_neuron[num_neurons], "number of tries = ", num_tries, "Accuracy = ", accuracy) accuracy_list[num_neurons][num_tries] = accuracy yerr = np.zeros(len(num_neuron)) mean_error = np.zeros(len(num_neuron)) for v in range(0, len(num_neuron)) : mean_error[v] = np.mean(accuracy_list[v]) yerr[v] = np.std(accuracy_list[v])$

plt.xlabel('number of neurons') plt.ylabel('Accuracy') plt.errorbar(num$_neuron, mean_error, yerr = yerr) plt.title('Average accuracy against number of neurons') plt.show($

REFERENCES

[1] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik, "EMNIST: an extension of MNIST to handwritten letters." Retrieved from: http://arxiv.org/abs/1702. 05373, (2017).

[2] L. Prechelt, Early Stopping — But When?, pp. 53–67. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.