Assignment A04: Neural Networks

A. Thieshanthan, 180641N

April 2, 2021

Abstract

This report covers the results and methods of implementing a Linear classifier, 2 Layer neural network and a convolutional neural network for CIFAR-10 classification. Important parts of the code are provided within the report. Full code can be found at `https://github.com/thieshanthan/EN2550`.

1 Linear Classifier

For this linear classifier, the scoring function $f(x) = Wx + b$ is used with the loss function Mean sum of squared errors(MSE). Some preparation and pre processing has been done to ease the coding and improve results. In some parts of the code numpy equivalent tensorflow functions are used to run the training on GPU.

1.1 Preparing dataset and Initializing parameters

The dataset images are normalized to produce a better effect on training. Normalizing is done by dividing by 255(largest entry in an image) and subtracting the mean of entire training set. Image labels are converted to one hot encoding vectors, which eases the loss computation. As there are 50000 training samples(m) with each image of size $32 \times 32 \times 3$ the training dataset is reshaped into (3072, 50000) vector. This prepared dataset is used for Questions 1,2 and 3 of this assignment.

Weights W and b are initialized to random values as per standard normal distribution and zeros respectively. A dictionary *linear_model_history* is initialized to store the history of training.

1.2 Model

As this is a simple model adding extra features such as learning rate decay, shuffling training data and adding regularization are relatively easy processes. For this model L2 regularization has been used.

$$\hat{Y} = WX + b \tag{1}$$

$$L = \frac{1}{m}\sum(\hat{Y} - Y)^2 - \frac{1}{m}\sum W^2 \tag{2}$$

which results in following gradients,

$$\frac{dL}{dw} = \frac{2}{m}(\hat{Y} - Y)X^T - \frac{2}{m}W \qquad \frac{dL}{db} = \frac{2}{m}(\hat{Y} - Y)$$

Using this gradients, gradient descent algorithm has been implemented. The backward propagation and gradient descent code is provided in listing 1

```
    # Loss function: Mean squared error loss
    loss = tf.math.reduce_sum((shuffled_y - y_hat)**2).numpy() / batch_size
    # Backward Pass
    dw1 = tf.matmul(y_hat - shuffled_y, shuffled_x, transpose_b=True) * (2/
    batch_size)
    I = tf.ones((batch_size, 1))
    db1 = tf.matmul(y_hat - shuffled_y, I) * (2/batch_size)
    # Gradient descent
    w1 = w1 - learning_rate * dw1 - (reg)* w1
    b1 = b1 - learning_rate * db1
```

Listing 1: Backward Propagation and Gradient Descent

1.3 Training the classifier

Training has been done for 300 epochs with regularization and learning rate decay. Learning rate is reduced every 100 epochs and shuffling the training is also carried out. The hyperparameters and results are listed below in table 1. The accuracy function used is provided in listing 2

```
def get_accuracy(y_hat, y):
    """ Classification accuracy  """
    y_hat_bin = np.argmax(y_hat,axis=0)
    y_class = np.argmax(y,axis=0)
    acc = 100*np.sum(y_hat_bin==y_class)/y_class.size
    return acc
```

Listing 2: Accuracy function used in Q1 Q2 and Q3

Hyperparameters	Value
Initial Learning rate	0.0015
decay	0.001
regularization factor	0.01

Table 1: Hyperparameters for Linear Classifier

1.4 Results

epoch	Training Loss	Training accuracy	Testing loss	Testing accuracy
1	1.0050	10.49	0.9953	15.41
300	0.8702	30.42	0.8696	30.91

Table 2: Training and testing loss and accuracy for linear classifier

The trained weights are displayed as 10 images in figure 1 and graphs of losses and accuracy are presented in figure 2a.

2 Two Layer Neural Network

A two layer neural network with sigmoid function as the activation function for the hidden layer has been implemented. There is no activation at the output layer and loss is MSE.

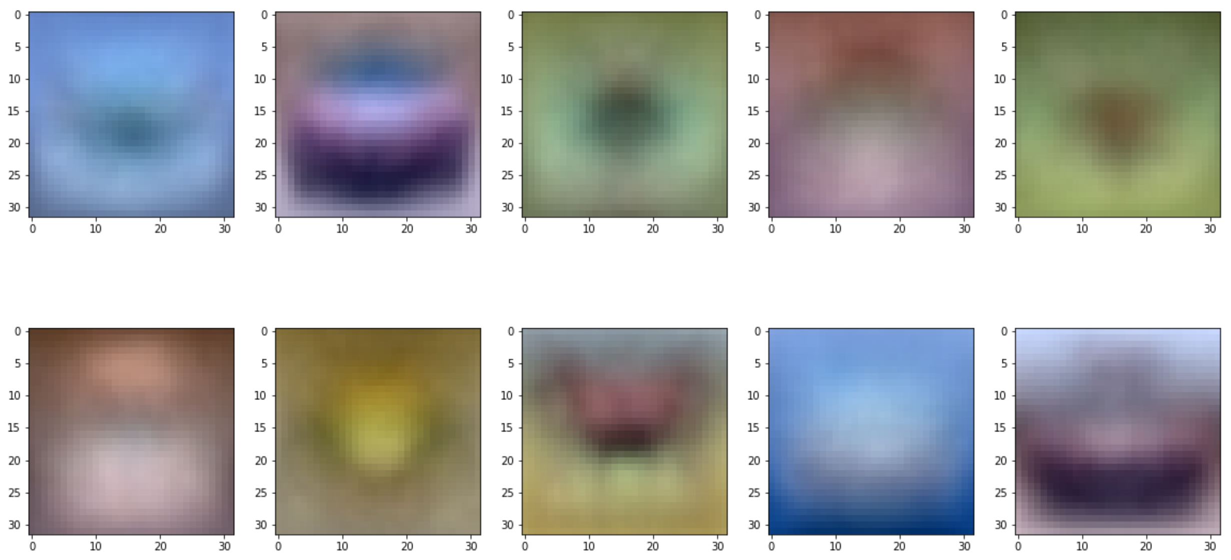
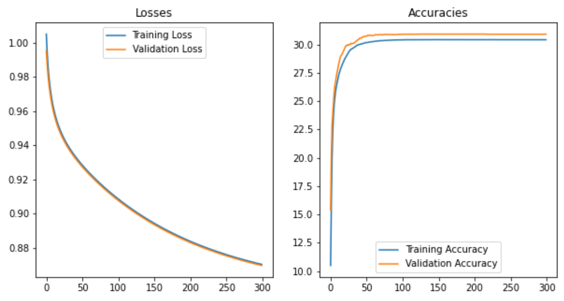
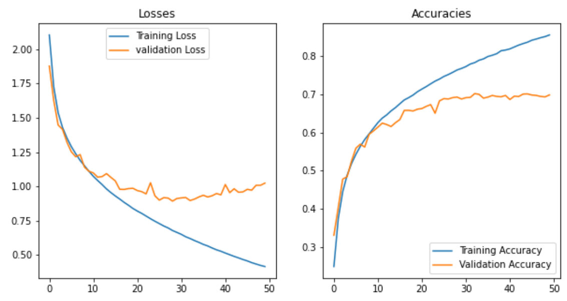


Figure 1: Trained weights of linear classifier



(a) Training of linear classifier



(b) Training of Convolutional Neural network

Figure 2: Training of linear classifier and CNN

2.1 Model

As the hidden layer has 200 units, the weights have been initialized with the shapes of [(200, 3072), (200,1)] and [(10,200), (10,1)] for each layer 1 and 2 respectively. The code is similar to the linear classifier in listing 1.

$$z_1 = w_1 X + b_1 \qquad \hat{Y} = w_2 A_1 + b_2$$

$$A_1 = sigmoid(z1) \qquad L = \frac{1}{m}\sum(\hat{Y} - Y)^2$$

the derivatives related to each weight can be represented as follows using chain rule,

$$\frac{dL}{d\hat{Y}} = \frac{2}{m}(\hat{Y} - Y) \qquad \frac{dL}{dz_1} = \frac{dL}{dA_1} \times (A_1 * (1 - A_1))$$

$$\frac{dL}{db_2} = \frac{dL}{d\hat{Y}} \qquad \frac{dL}{dw_1} = \frac{dL}{dz_1} X^T$$

$$\frac{dL}{dA_1} = w_2^T \frac{dL}{d\hat{Y}} \qquad \frac{dL}{db_1} = \frac{dL}{dz_1}$$

Gradient descent algorithm has been implemented with these gradients.

Hyperparameters	Value
Initial Learning rate	0.015
decay	0.005

Table 3: Hyperparameters for two layer network in question 2 and 3

epoch	Training Loss	Training accuracy	Testing loss	Testing accuracy
1	0.99	10.00	0.92	8.56
300	0.8583	26.67	0.8573	27.04

Table 4: Training and testing loss and accuracy for 2 layer network

2.2 Results

This 2 layer model should perform better than the linear classifier in question 1. But since we are only training the network for 300 epochs the training and testing losses tends to be higher than the linear classifier. Training for more epochs will result in a better results.

3 Implementing Stochastic Gradient Descent

The disadvantage of gradient descent algorithm is it waits until all the training data is used for one gradient calculation and then makes a progress. Stochastic Gradient Descent(SGD) update gradients for every sample processed. Which makes use of all the computations carried out and makes process towards reducing loss from step 1. But this introduces fluctuation in training. Mini batches are used to solve this and helps to arrive at a converging point much faster than regular gradient descent. This can be observed by comparing losses and accuracies provided in table 4 and table 5.

3.1 Results

It can be observed that SGD with minibatch outperforms regular gradient descent by a huge margin with given number of epochs. But using SGD somewhat increases the runtime of training but provides better results. SGD is useful when there is high number of training samples are available.

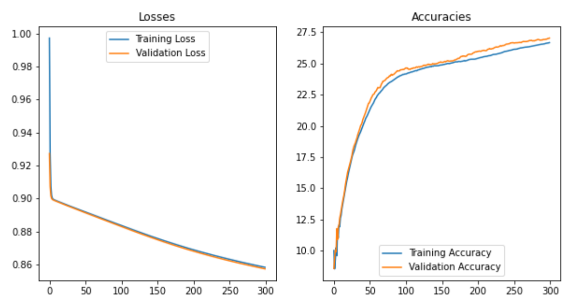
After implementing SGD 2 layer network is trained better than the linear classifier on 300th epoch which regular gradient descent algorithm would have required more epochs or a higher learning rate. But SGD have optimized the model better with the same epochs and same learning rate.

4 Convolutional Neural Network with Keras

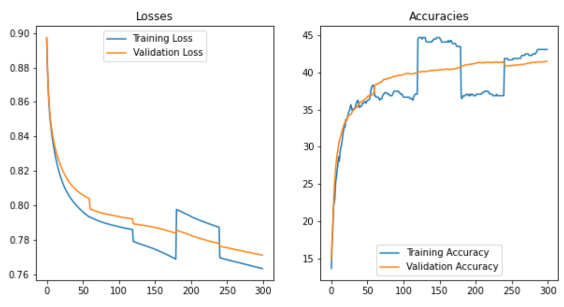
A Convolutional Neural network(CNN) has been implemented with the given architecture. Relu is used as activation for the convolutional layers and penultimate fully connected layer. Softmax

epoch	Training Loss	Training accuracy	Testing loss	Testing accuracy
1	0.8967	13.62	0.8973	14.95
300	0.7635	43.08	0.7713	41.50

Table 5: Training and testing loss and accuracy for 2 layer network with SGD optimizer



(a) Training of 2 layer neural network with gradient descent algorithm



(b) Training of 2 layer neural network with Stochastic gradient descent algorithm

Figure 3: Training with gradient descent and SGD

function is used as the activation function of the last layer. SGD has been implemented with learning rate **0.01** and momentum **0.01**.

As the convolutional layers extract features from the image, CNN outperform every network implemented from question 1 to 3. The results are provided in table 6. The implemented CNN passes the training loss and accuracy of 2 layer network with a small number of epochs. But it overfits the training data hence the testing loss is increasing after several number of epochs.

epoch	Training Loss	Training accuracy	Testing loss	Testing accuracy
1	2.22	0.20	1.87	0.33
50	0.41	0.85	1.02	0.69

Table 6: Training and testing loss and accuracy for CNN model

4.1 Number of learnable parameters

Maxpooling layers and flatten layer has no parameters. The number of parameters in a convolutional layer can be calculated as shown in equation 3

$$n_i = [(h \times w \times f_{i-1}) + 1] \times f_i \tag{3}$$

here h, w are the height and width of the filter. f_i is the number of filters in layer i. 1 is added for the bias term of each filter. Using equation 3 the calculated numbers are shown in table 7.

Layer	Parameter calculation	Number of parameters
Layer 1: C32	$[(3 \times 3 \times 3 \times) + 1] \times 32$	896
Layer 2: C64	$[(3 \times 3 \times 32 \times) + 1] \times 64$	18496
Layer 3: C64	$[(3 \times 3 \times 64 \times) + 1] \times 64$	36928
Layer 4: F64	$256 \times 64 + 64$	16448
Layer 5: F10	$64 \times 10 + 10$	650
Total parameters		73418

Table 7: Number of parameters in the CNN model