Deep Learning Assignment 2 Report

Cifar10 Analysis using Convolutional Neural Network (CNN)

Taylan Tatlı

Physics PhD student

taylan.tatl@gmail.com

CMPE 597 Assignment 2

Department of Computer Engineering,
Boğaziçi University, 01/05/2023

Contents

1 Introduction

In this project, the subject is to train a convolutional neural network using Tensorflow. In this assignment, CIFAR10 dataset is used. First section contains the work of data augmentation and data preprocess (loading data, train and test split, normalization..). Second section focuses on the design of 3-layer CNN architecture for the object classification task and its ways to get high performance as much as possible. This section will give information about the main.py (contains all training procedure, optimizers, training loop, tsne plotting,early stopping etc.), model.py (implementation of the model architecture), eval.py (evalutaion of model, test data). Also it will give the the reasoning behind choices on the kernel size, activation functions, number of feature maps, number of fully connected layers, and the size of the fully connected layers and techniques to improve the performance, such as adding dropout, batch normalization. Third section focuses one optimizer and training network. It will contain information of training and test loss and accuracy's results and early stopping procedure. The next section will contain the works of trying different optimizers after all hyperparameters are chosen. and the final section will contain visualizing the latent space of the model for the training samples (output of the feature extraction layers) at the beginning of the training, in the middle of the training, and at the end of the training.

2 Data Augmentation Techniques

2.1 Data Load and PreProcess:

Data is loaded from the reference [1]. The dataset is divided into batches to prevent memory issues [2]. When the data is loaded using tensorflow attributes, the training is cut by running out of memory errors. The CIFAR-10 dataset consists of 5 batches, named data_batch_1, data_batch_2, etc. Each file packs the data using pickle module in python. The dataset consists of 60000 32x32 colour images in 10 classes, and for each class approximately 6000 images. There are 50000 training and 5000 test images. Train images are divided into training data containing 45000 images and validation data containing 5000 image. Inputs are normalized and the classes are one-hot encoded to train. The figure of train test and validation data shapes are given in Figure 1.

```
[ ] x_train.shape,y_train.shape,x_test.shape,y_test.shape,x_val.shape,y_val.shape

    ((45000, 32, 32, 3),
     (45000, 10),
     (10000, 32, 32, 3),
     (10000, 10),
     (5000, 32, 32, 3),
     (5000, 10))
```

Figure 1: Data shapes is given.

2.2 Data Augmentation

Data augmentation is the technique that increases the amount of data with some specific methods. For the image data augmentatıon there are 5 main methods, which are geometric transformation, containing random flip, rotate etc. and should be careful as it is used, since using all transformation methods at the same time may reduce model performance. Another one colour transformations, and kernel filters changing randomly image sharpness and blurring. The last other two are random erasing that erase some part of ımages randomly and mixing images mixing multiple images [3]. In the assignment, the set augmented techniques are given Figure 2. It is inspired from the reference [4].

```
data_augment = ImageDataGenerator(
        featurewise_center=False,  # set input mean to 0 over the dataset
        samplewise_center=False,  # set each sample mean to 0
        featurewise_std_normalization=False,  # divide inputs by std of the dataset
        samplewise_std_normalization=False,  # divide each input by its std
        zca_whitening=False,  # apply ZCA whitening
        zca_epsilon=1e-06,  # epsilon for ZCA whitening
        rotation_range=0,  # randomly rotate images in the range (degrees, 0 to 180)
        # randomly shift images horizontally (fraction of total width)
        width_shift_range=0.1,
        # randomly shift images vertically (fraction of total height)
        height_shift_range=0.1,
        shear_range=0.,  # set range for random shear
        zoom_range=0.,  # set range for random zoom
        channel_shift_range=0.,  # set range for random channel shifts
        # set mode for filling points outside the input boundaries
        fill_mode='nearest',
        cval=0.,  # value used for fill_mode = "constant"
        horizontal_flip=True,  # randomly flip images
        vertical_flip=False,  # randomly flip images
        # set rescaling factor (applied before any other transformation)
        rescale=None,
        # set function that will be applied on each input
        preprocessing_function=None,
        # image data format, either "channels_first" or "channels_last"
        data_format=None,
        # fraction of images reserved for validation (strictly between 0 and 1)
        validation_split=0.0)
```

Figure 2: Data shapes is given.

3 CNN Architecture

3.1 The Reasoning behind the Model Implementation Choices

This section will describe the reasoning behind the the kernel size, activation functions, number of feature maps, number of fully connected layers, and the size of the fully connected layers. The chosen parameters can be seen in Figure 3 and also model.py in the project link.

3.1.1 Kernel Size

In CNN architecture, filters are the number of output channels after convolution performed, and kernel tells the size of a convolution filter used ın the architecture. The chosen kernel size is (3x3) in this assignment. The kernel size should be as small as possible so that we can extract information as much possible. If it is too big, it would lose the information and so decrease the model performance. Also, it is better to chose odd numbered kernel size. Convolutional operation needs some symmetry, meaning that all the previous layers pixel should be symmetrically around the output pixel, if not, the model has to consider the distortions across the layers, and it increases the complexity too [5]. That's why odd numbered kernel size is used.

3.1.2 Activation Functions

The used activation function is Relu. It is for getting non-linear transformation of the data. Why Relu activation function? It is simple, fast, and don't suffer from vanishing gradients, like other functions. In more detail, The gradient computation is very simple, it is 0 or 1 depending on the sign of input. The computational step of a ReLU is easy: any negative elements are set to 0.0, so no exponentials, no multiplication or division operations and Gradients of the positive portion of the ReLU is larger that the others like logistic or tnah functions, as so the positive portion is updated more rapidly, and so all these speeds up training [6, 7].

3.1.3 Number of Feature Maps:

Each different filtering results in a distinct feature map. For each convolution layer different filter numbers used and so creating different number of feature maps. It is summarized in the Table 1.

Table 1: Number of filters and feature maps layer by layer.

Convolution Layers	**Filter Numbers**	**Fetaure Map**
Conv_1	32	32
Conv_2	64	64
Conv_1	128	128

Using more filters can create a more powerful model, but it also brings the risk of overfitting, since it can learn every small details of input data by filter and filter again. So it is sensible to use small number of filters but it can be increased as the layers progress because for the last layers extracting information can be difficult, so using more filters are reasonable here. Also filter number should be multiple of 2 [8].

3.1.4 Number and Size of the Fully Connected Layer

There are two fully connected layer. We have input in 2D but, for predictions we need 1D vector so first flatten layer bring this 1D vector and first fully connected layer take this and with size 512 and named as fully_connected_1 in the model. And we have ten classes so we need probabilistic output for each class and it should be sized as 10. In the model it is done by the layer named as y_pred and it is also final layer of model. Fully connected layer should not be too big, since it contains most of model parameters and so consumes most memory. But also should be big enough to preserve information so 512 is seen by me enough, and tested with training the model.

All implementation of these parameter are given in Figure 3, in model.py subsection.

3.2 Model.py File

It is named Model.py and it contains architecture of the model, code is shown in Figure 3. It has 3 Convolutional layer, after each convolution batch normalization is applied which will be explained later. After batch normalization, relu activation functions are applied for each convolution and after that max pooling applied on the activation function output. After all, since fully connected layer should be 1 dimensional, the output is first flatten by reshape function and then model goes with a fully connected layer with shape(,512), activation function is again relu and dropout is applied on this application function result. At the end a fully conected layer named as y_pred with shape(,10) and it contains the probabilistic values of each 10 class, classification is done depend on these values. Shape of each procedure is given in Figure 4. It is inspired from the works in reference [9] [10]

```
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

def model():
    x = tf.placeholder(tf.float32, shape=[None, 32, 32, 3])
    y_true = tf.placeholder(tf.float32, shape=[None, 10])
    train_mode = tf.placeholder(tf.bool)

    conv1 = tf.layers.conv2d(x, filters=32, kernel_size=[3, 3], padding='same')
    conv1_bn = tf.layers.batch_normalization(conv1,momentum=0.9, training=train_mode)
    conv1_act = tf.nn.relu(conv1_bn)
    conv1_pool = tf.layers.max_pooling2d(conv1_act, pool_size=[2, 2], strides=2, padding='same')
    #conv1_bn = tf.layers.batch_normalization(conv1_pool)
    print("Convolution 1: ", conv1.shape)
    print("Activation Shape: ", conv1_act.shape)
    print("Pooling Shape: ", conv1_pool.shape)
    print("Batch Normalization Shape: ", conv1_bn.shape)
    #
    conv2 = tf.layers.conv2d(conv1_pool,filters=64,kernel_size=[3, 3], padding='same')
    conv2_bn = tf.layers.batch_normalization(conv2, momentum=0.9, training=train_mode)
    conv2_act = tf.nn.relu(conv2_bn)
    conv2_pool = tf.layers.max_pooling2d(conv2_act, pool_size=[2, 2], strides=2, padding='same')
    #conv2_bn = tf.layers.batch_normalization(conv2_pool)
    print("Convolution 2: ", conv2.shape)
    print("Activation Shape: ", conv2_act.shape)
    print("Pooling Shape: ", conv2_pool.shape)
    print("Batch Normalization Shape: ", conv2_bn.shape)
    #
    conv3 = tf.layers.conv2d(conv2_pool, filters=128, kernel_size=[3, 3], padding='same')
    conv3_bn = tf.layers.batch_normalization(conv3, momentum=0.9, training=train_mode)
    conv3_act = tf.nn.relu(conv3_bn)
    conv3_pool = tf.layers.max_pooling2d(conv3_act, pool_size=[2, 2], strides=2, padding='same')
    #conv3_bn = tf.layers.batch_normalization(conv3_pool)
    print("Convolution 3: ", conv3.shape)
    print("Activation Shape: ", conv3_act.shape)
    print("Pooling Shape: ", conv3_pool.shape)
    print("Batch Normalization Shape: ", conv3_bn.shape)
    #
    flat = tf.reshape(conv3_pool, [-1, 4 * 4 * 128])
    print("flatten: ", flat.shape)

    full_connected_1 = tf.layers.dense(flat, units=512)
    activation_4 = tf.nn.relu(full_connected_1)
    dropout_4 = tf.layers.dropout(activation_4, rate=0.5, training=train_mode)
    print("full_connected_1: ", full_connected_1.shape)
    print("activation_4: ", activation_4.shape)
    print("dropout_4: ", dropout_4.shape)

    y_pred = tf.layers.dense(dropout_4, units=10)
    print("y_pred: ", y_pred.shape)
    return x, y_true, y_pred, train_mode, dropout_4, flat
```

Figure 3: code implementation of CNN model.

```
Convolution 1:  (?, 32, 32, 32)
Batch Normalization Shape:  (?, 32, 32, 32)
Activation Shape:  (?, 32, 32, 32)
Pooling Shape:  (?, 16, 16, 32)
Convolution 2:  (?, 16, 16, 64)
Batch Normalization Shape:  (?, 16, 16, 64)
Activation Shape:  (?, 16, 16, 64)
Pooling Shape:  (?, 8, 8, 64)
Convolution 3:  (?, 8, 8, 128)
Batch Normalization Shape:  (?, 8, 8, 128)
Activation Shape:  (?, 8, 8, 128)
Pooling Shape:  (?, 4, 4, 128)
flatten:  (?, 2048)
full_connected_1:  (?, 512)
activation_4:  (?, 512)
dropout_4:  (?, 512)
y_pred:  (?, 10)
```

Figure 4: Shape of layers are given. First element changes according to given input, batch sizes.

3.3 Main.py File

Main file contains the optimizer and the implementation of the training loop. It is inspired from the works in references [9] [10]. The chosen optimizer is Adam optimizer, loss function is softmax cross entropy. The chosen epoch number is 30 (to see result faster, higher epoch number is better until stuck in some point), learning rate is 0.01 and batch size is 32. Also it contains, data preprocess parts like data augmentation, preparing data as train, test and validation and functions to plot both model performance (loss and accuracy) and the latent representations into 2-D space. Training loop contains both tsne plotting and early stopping procedure. At the end, model is saved and session is closed. Training loop is given in Figure 6 and used optimizer and loss functions are given in Figure 5., in the project link, all is named as Main.py.

```
max_epochs = 30
Batch_size = 32
learning_rate = 0.01
tsne_plot = False
early_stoping = False


# In[ ]:


# model
x, y_true, y_pred, train_mode, fully_connected_1, latent_space_flat = model()

# loss function
loss = tf.losses.softmax_cross_entropy(y_true, y_pred)

# accuracy calculation
accuracy = tf.reduce_mean(tf.cast(tf.equal(tf.argmax(y_pred, 1), tf.argmax(y_true, 1)), tf.float32))

update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS) # for batchnorm

# optimization functions
train_optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(loss)

train_optimizer = tf.group([train_optimizer, update_ops]) # for batchnorm
```

Figure 5: Optimizer and loss function.

```
tf_session = tf.Session()
tf_session.run(tf.global_variables_initializer())
print("------Session Started------")
train_acc,train_loss,val_acc,val_loss = [],[],[],[] # To plot loss and accurcies, train and validation
#Two parameters defined for early stopping
key_stop=0
key_loss=0
stop_training=False
# Training start here
for epoch in range(max_epochs):

    if stop_training == True:
        print("-----Early stopping is active.-------")
        print("-----Training Stopped-----")
        break
     # for each epochs, use batch calculate epoch loss and accuracies to plot.
    number_of_batch = 0
    epoch_acc, epoch_loss, epoch_val_acc, epoch_val_loss = 0, 0, 0, 0

    for batch_x, batch_y in data_augment.flow(x_train, y_train, batch_size=Batch_size):

        _, batch_loss = tf_session.run([train_optimizer, loss],
                                       feed_dict={x: batch_x, y_true:batch_y, train_mode: True})

        batch_acc = tf_session.run(accuracy, feed_dict={x: batch_x, y_true:batch_y, train_mode: False})
        epoch_loss += batch_loss
        epoch_acc += batch_acc
        number_of_batch += 1

        if number_of_batch >= len(x_train)/Batch_size: # stop training as reach the last batch of data
            break
    epoch_loss /= number_of_batch #loss for epoch
    epoch_acc /= number_of_batch # accuracy for epoch

    epoch_val_loss, epoch_val_acc = tf_session.run([loss, accuracy],
                                                   {x: x_val, y_true: y_val,
                                                    train_mode: False})

    epoch_acc *= 100
    epoch_val_acc *= 100
    val_loss.append(epoch_val_loss)
    val_acc.append(epoch_val_acc)
    train_loss.append(epoch_loss)
    train_acc.append(epoch_acc)
    print(
        "Epoch: {} Train Loss: {:.2f} - Validation Loss: {:.2f} | Train Accuracy: {:.2f}% - Validation Accuracy: {:.2f}% ".format(
            epoch + 1, epoch_loss, epoch_val_loss, epoch_acc, epoch_val_acc))

    # Early stopping:
    # After 5 epoch if, epoch decrease 3 times then stop, check validation losses
    if early_stoping ==True:
      if len(train_loss)>5:
        if (val_loss[len(train_loss)-1]>val_loss[len(train_loss)-2] and train_loss[len(train_loss)-1]<train_loss[len(train_loss)-2]):
          print("deneme")
          key_stop +=1
          print("key_stop",key_stop)
        else:
          key_stop=0
          print("key_stop",key_stop)


      if key_stop ==3:
        print("key_stop",key_stop)
        stop_training=True

        print("Early stopping is True. Validation losses start to increse and Training loss keeping decrease.")
    if tsne_plot ==True and (epoch == 3 or epoch == 13 or epoch == 27):
      latent_fc1, latent_flatten = tf_session.run([fully_connected_1, latent_space_flat],
                                                  {x: x_test, y_true: y_test, train_mode: False})
      plot_tsne(np.array([latent_fc1]), np.array([np.where(r == 1)[0][0] for r in np.array(y_test)]),
                "fc1_test_tsne_" + str(epoch))
      plot_tsne(np.array([latent_flatten]), np.array([np.where(r == 1)[0][0] for r in np.array(y_test)]),
                "flatten_test_tsne_" + str(epoch))
```

Figure 6: Training Loop.

3.4 Eval.py

Eval.py file is implemented to load the saved model and evaluate the test performance. It is given in Figure 7. In the Figure, data preparation, loading saved model and testing the model parts are given.

```
x_train, y_train, x_test, y_test=data_train_test() # create data
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=5000, random_state=42)
y_train, y_val, y_test = np.eye(10)[y_train], np.eye(10)[y_val],np.eye(10)[np.array(y_test)]

#x_train /= 255
#x_val /= 255
x_test /= 255

x, y_true, y_pred, train_mode, fully_connected_1, latent_space_flat= model()
saver = tf.train.Saver()
tf_session = tf.Session()
saver.restore(tf_session, 'sample_data/model_save/my-model')
print("model is loaded")


# In[ ]:


# accuracy calculation
accuracy = tf.reduce_mean(tf.cast(tf.equal(tf.argmax(y_pred, 1), tf.argmax(y_true, 1)), tf.float32))

test_accuracy = tf_session.run(accuracy, {x: x_test, y_true: y_test, train_mode: False})

print("Test set accuracy {:.2f}%".format(test_accuracy*100))
tf_session.close()
```

Figure 7: Eval.py.

3.5 Techniques to Improve the Performance

3.5.1 Batch Normalization

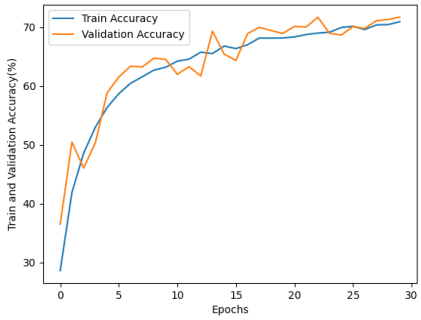
The idea behind the batch normalization is that normalizes the output of the previous layer by subtracting the batch mean and dividing by the batch standard deviation. [11] [12].

- Each layer a little more independent from other, and so the model is less sensitive to noisy images, and variations in
- After the normalization, there are no output of the layer that are really high or low, and that's why the learning rate may be increased without worrying gradient divergence.

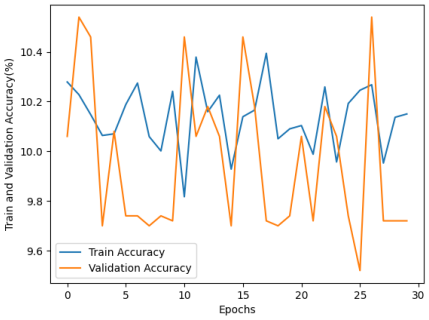
Without Batch normalization, Train and Validation accuracy is given Table 2 and The accuracy plot is given Figure 8 .

3.5.2 Dropout

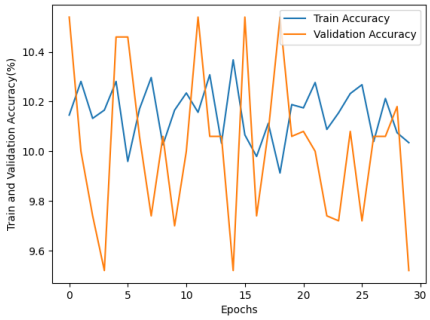
It is the technique for reducing overfitting in neural networks, it prevents complex co-adaptations on training data by randomlydropping out nodes. It is computationally cheap and effective [13]. The train and validation plots is shown in Figure 8 and the final accuracy values are given in Table 2



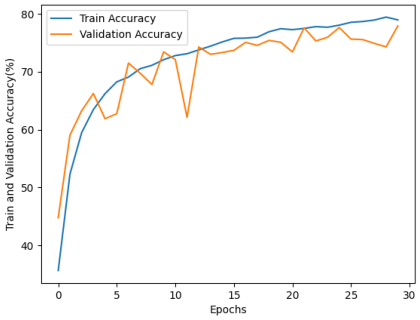
(a) With Batch Norm and Dropout.



(b) No Batch Norm and Dropout



(c) Without Batch Norm. with Dropout



(d) With Batch Norm. without Dropout

Figure 8: Accuracy resutlts for 30 epoch, no early stopping.

Table 2: Train and Validation Accuracy for epoch 30. Effects of Batch Normalization and Dropout regularization teqniques.

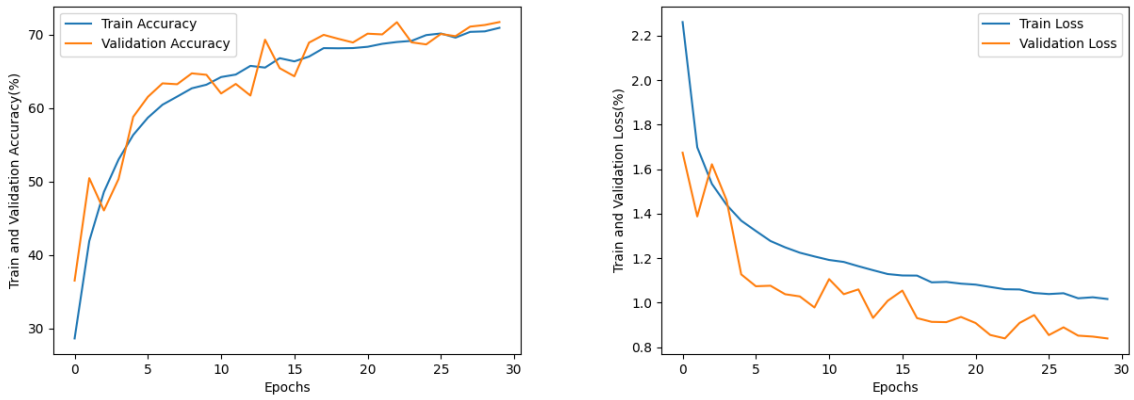
Technique	**Train Accuracy %**	**Validation Accuracy %**	**Epochs**
With Batch Norm. and Dropout	70.94	70.87	30
Without Batch Norm,Dropout	10.15	9.72	30
Without Batch Norm. with Dropout	10.03	9.52	30
With Batch Norm. without Dropout	78.97	77.92	30

It can be easily seen that without batch normalization model performance highly reduced. On the other hand, with batch normalization and without dropout it seems like that model performs well with little bit overfitting. In the model that can chosen too.

4 Optimizer, Train and Validation Performance, Early Stopping

4.1 Train Test Accuracy, Optimizer

The chosen optimizer is Adam optimizer. The batch size is 32. The train and validation accuracy and loss plot is given in Figure 8-9. Train and test accuracy's are given in Table 2.



(a) Train and Validation Accuracy.

(b) Train and Validation Loss.

Figure 9: Loss and Acuracy resutlts for 30 epoch, no early stopping..

Table 3: Train and Test Accuracy.

	Train	**Test**
Accuracy %	70.94	70.87

4.2 Early Stopping

It is an regularization being used to avoid overfitting when training a learner with an iterative method. In the assignment, the chosen Early Stopping procedure is such follows. If three times the validation loss increases consecutively and train loss keep decreases, then

stop. It tells us a point that the validation loss start to increase and may be overfitting. Code ımplementation is given Figure 10.

```
# Early stopping:
# After 5 epoch if, epoch decrease 3 times then stop, check validation losses
if early_stoping ==True:
  if len(train_loss)>5:
    if (val_loss[len(train_loss)-1]>val_loss[len(train_loss)-2] and train_loss[len(train_loss)-1]<train_loss[len(train_loss)-2]):
      print("deneme")
      key_stop +=1
      print("key_stop",key_stop)
    else:
      key_stop=0
      print("key_stop",key_stop)


  if key_stop ==3:
    print("key_stop",key_stop)
    stop_training=True

    print("Early stopping is True. Validation losses start to increse and Training loss keeping decrease.")
```

Figure 10: Early Stopping code implementation on the Main.py at tha part of training loop.

For our model with 30 epoch, early stopping didnt give any output. If we slack the procedure it can give, or increase the epoch number can enable early stopping to stop training. Also, as we check the loss or accuracy plots no need to stop for training, ıt can be easily seen that there is no overfitting problem, the expected overffitng problem for early stopping is given Figure 11 and no pattern like in my original plots. But again, if we increase the epoch number or input data etc. there can be overfitting problem and my activate early stopping.

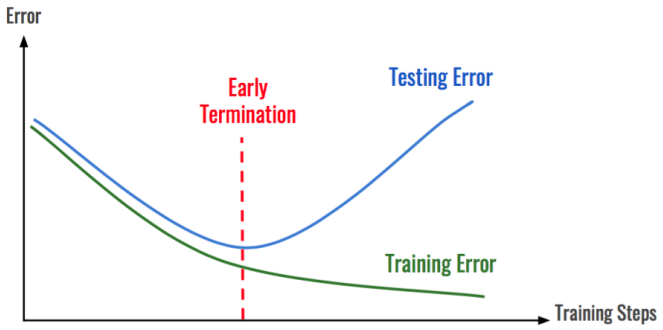


Figure 11: Expected plot for early stopping. No early stopping in my studies.

If we change early stopping procedure in a way that training loss decrease and validation loss increase, then stop, the code gives output but this procedure is not suitable for early stopping logic, it is done just for showing code works very well.

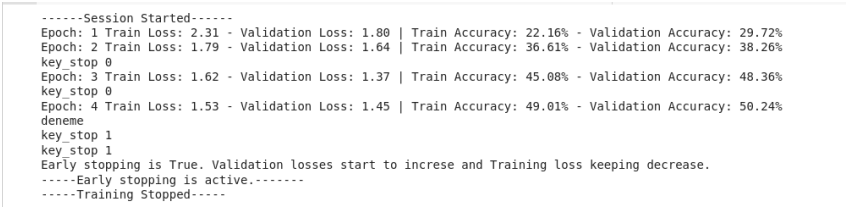


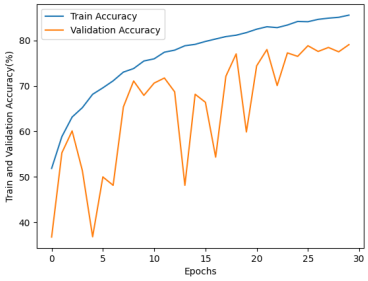
Figure 12: Early Stopping output when we push very basic early stopping.

5 Trying Different Optimizers

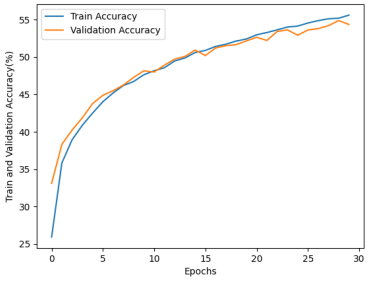
The tried optimizers are Gradient Descent(GD), ProximalAdaGrad and AdaDelta Optimizer. The values of loss and accuracy is given in Table 4 and train and validation accuracy plots for 3 optmizers are given in Figure 13.

Table 4: Differen Optimizers train anda Vvalidation loss and accuracy results..

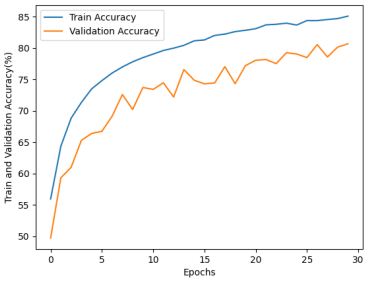
Optimizers	**Train Accuracy**	**Validation Accuracy**	**Train Loss**	**Validation Loss**
GD	85.97%	70.87%	0.62	0.60
AdaDelta	55.61%	54.36%	1.39	1.26
ProximalAdaGrad	85.07%	80.66%	0.63	0.57



(a) GradientDescent.



(b) AdaDelta.



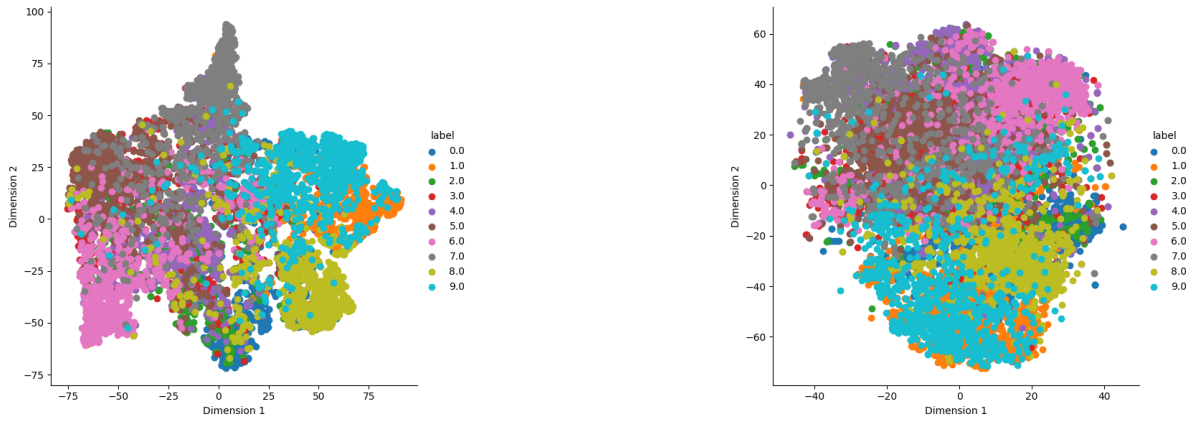
(c) ProximalAdaGrad.

Figure 13: Accuracy curves for each optimizers.

If we check accuracy curves, for GD and ProximalAdaGrad optimizer, it looks like an overfitting problem, on the other hand for the AdaDelta, accuracy result is not promising. So We cannot say that one of them better than Adam Optimizer.

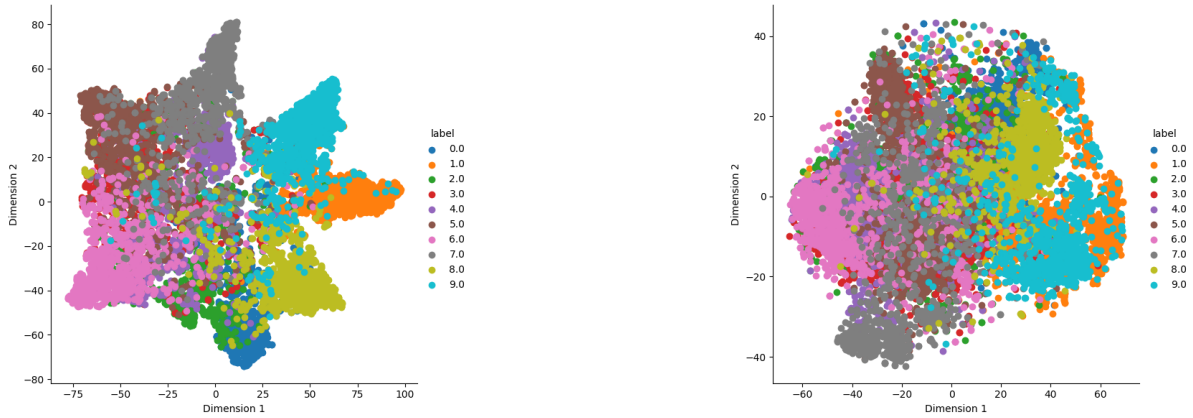
6 Visualizing the Latent Space, TSNE Plots

The chosen epochs are 4, 14 and 28 for the 30 epoch model. The plots for each are given below Figures 14-16. Fully connected Layer 1 and flatten layer is considered.



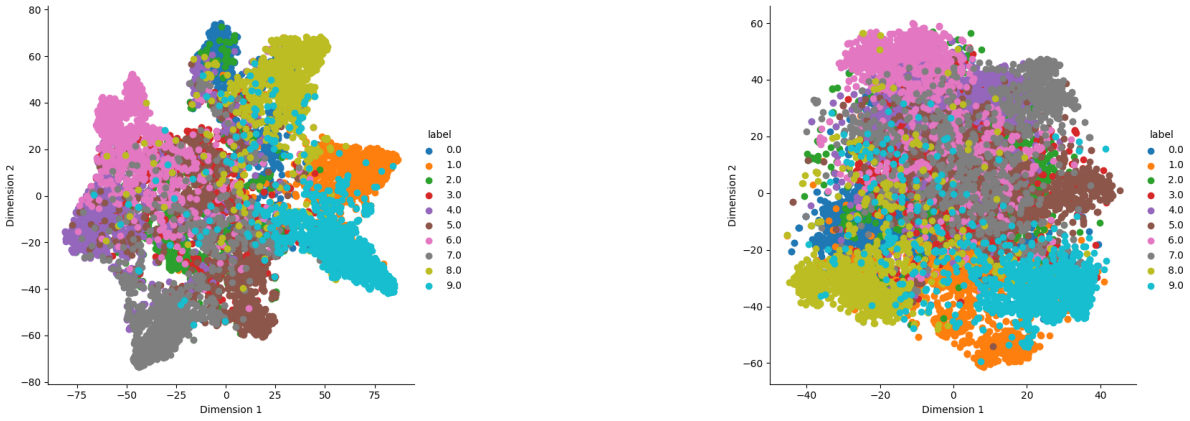
(a) Fully Connected Layer 1 (b) Flatten Layer

Figure 14: Latent Space visiulization for epoch 4.



(a) Fully Connected Layer 1 (b) Flatten Layer

Figure 15: Latent Space visualization for epoch 14.



(a) Fully Connected Layer 1

(b) Flatten Layer

Figure 16: Latent Space visualization for epoch 28.

If we consider the fully connected layer visualization, for epoch 4, features are not very much separable but for epoch 14 and 28 it is considerably separable and since the accuracy results for these epochs are so close, separability on that plots are close to each other. Also we can observe the same thing for flatten space visualization.

References

[1] T. Uni., "The cifar-10 dataset." https://www.cs.toronto.edu/~kriz/cifar.html.

[2] P. Chansung, "Cifar-10 image classification in tensorflow." https://towardsdatascience.com/cifar-10-image-classification-in-tensorflow-5b501f7dc77c.

[3] Datacamp, "A complete guide to data augmentation." https://www.geeksforgeeks.org/python-data-augmentation/.

[4] R. Nana, "cifar10 with cnn for beginer." https://www.kaggle.com/code/roblexnana/cifar10-with-cnn-for-beginer.

[5] S. Sahoo, "Deciding optimal kernel size for cnn." https://towardsdatascience.com/deciding-optimal-filter-size-for-cnns-d6f7b56f9363.

[6] stats.stackexchange, "Why do we use relu?." https://stats.stackexchange.com/questions/226923/why-do-we-use-relu-in-neural-networks-and-how-do-we-use-it.

[7] J. Brownlee, "A gentle introduction to the rectified linear unit." https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/.

[8] A. Dertat, "Applied deep learning - part 4: Convolutional neural networks." https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2.

[9] M. Lecanu, "Tp3 deep learning with pytorch: Cifar10 object classification." https://github.com/LMaxence/cifar10-classification.

[10] C. Park, "Cifar10-img-classification-tensorflow." https://github.com/deep-diver/CIFAR10-img-classificationtensorflow/blob/master/CIFAR10_image_classification.ipynb.

[11] M. Lecanu, "Tp3 deep learning with pytorch: Cifar10 object classification." https://github.com/LMaxence/cifar10-classification.

[12] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015.

[13] J. Brownlee, "A gentle introduction to dropout for regularizing deep neural networks." https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/.