Programming Assignment-3

Name: Mohaiminul Al Nahian

RIN: 662026703

Course No.: ECSE 6850

Introduction:

In this programming assignment, our goal is to implement a Convolutional Neural Network to classify CIFAR-10 dataset. The dataset is divided into 10 classes naming- airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck. For this problem, we are given a subset of the total dataset, with 50000 training images and 5000 validation images. The network architecture we use has 3 convolution layers, 2 maxpool layers, one fully connected layer and one output layer. The maxpool layers are placed after the first two convolution layers. The neural network has been trained by minimizing the cross-entropy loss function of the output nodes. We tune our parameters by minimizing the loss function over several epochs by gradient descent approach. It is computationally inefficient to calculate gradients of all training samples at a time. So, We have implemented Stochastic Gradient Descent technique. The size of the layers had been fixed by the instruction of the assignment. But through trial and error, we have tuned the hyper-parameters of the model. In each epoch, we train over all mini-batches and keep track of training & testing loss and training & testing accuracies. After finishing the training, we check the model's performance on the test data and save the learned parameters in the format specified.

The rest of the report is organized as follows: We describe the architecture of the model, then we discuss about the hyper-parameter settings, we plot the results of accuracy and loss over epochs, we do a result analysis, we visualize some of the learned weights, finally we close the report with a discussion. We have also included the code used to train the model.

Model Description:

The architecture used is a feedforward convolutional neural network. The input image to the CNN has a shape of (32,32,3). We have normalized the image pixels by dividing the intensities by 256. The first convolution layer has 16 filters. The kernel size is 5x5x3. So, we get 16 featuremaps. We have not used zero padding to the feature maps have size 28x28. The activation function is ReLu. We have used a dropout layer after this which is only active during the training phase. Then we do a 2x2 maxpooling operation giving 16 feature

maps of 14x14. Then we do another convolution with 32 number of 5x5x16 kernels without zero-padding, giving 32 featuremaps, each with a size of 10x10. We apply another dropout here which is activated only during training. Then we do maxpooling 2x2 giving 64 maps of size 5x5. Then we apply third convolution layer, having 64 number of 3x3x32 kernels. So, we get 64 featuremaps of size 3x3. We then flatten this layer and add a fully connected layer of 500 nodes. Then we apply another dropout layer here, activated during training. Then we connect this layer with the output layer having 10 nodes corresponding to the 10 output classes. The activation functions for all convolution and the fullyconnected layer is ReLu. The output node has the softmax activation function. A visualization of the CNN architecture is given below

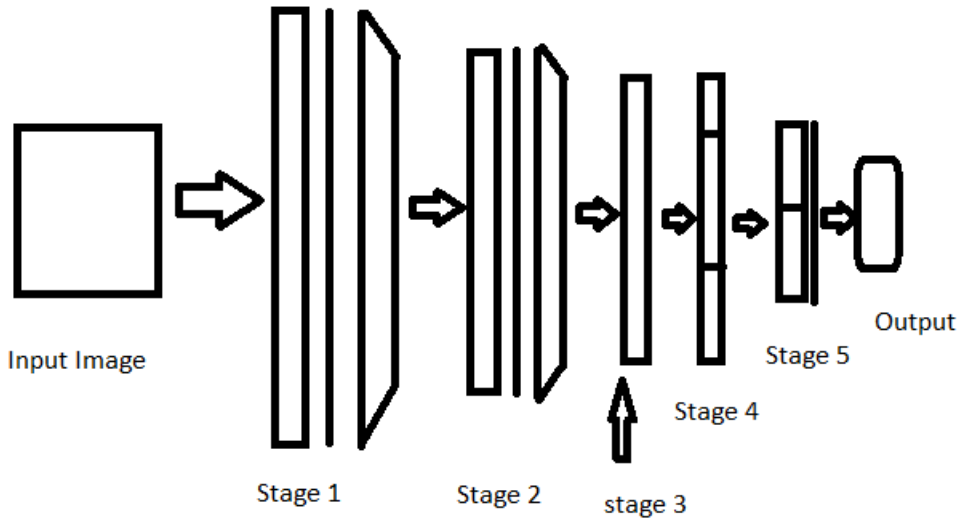


Figure: CNN Architecture

Here, the convolution layers have been shown by a rectangles, dropout layers are represented by vertical line, Maxpool layer is shown by a parallelogram, vectorized conv layer and fully connected layers are shown by rectangles with horizontal stripes. The architecture is divided into 5 stages along with one input and one output stage. The input image goes to the first stage where there is a convolution layer with ReLu activation, a dropout layer and a maxpool layer. Then the stage 2 has similar convolution with ReLu, dropout and maxpool layers although of different size. The third stage has a convolution

layer. Stage 4 represents the vectorized flattened conv3 layer. Stage-5 represents the fully connected layer with 500 nodes and a dropout layer and then finally we have the output layer with 10 nodes.

Loss Function:

The negative conditional log likelihood loss function as the loss for a single training sample (x[m], y[m]), is defined as follows:

$$
\begin{aligned}
l(y[m], \hat{y}[m]) &= -\log p(y[m] \mid x[m]) \\
&= -\log \prod_{k=1}^{K} p(y[m][k] = 1 \mid x[m])^{y[m][k]} \\
&= -\sum_{k=1}^{K} y[m][k] * \log(p(y[m][k] = 1 \mid x[m])) \\
&= -\sum_{k=1}^{K} y[m][k] * \log(\hat{y}[m][k])
\end{aligned}
$$

We calculate the loss for all samples in an iteration and then sum them to get the total loss.

Optimizer:

To optimize the loss function, we have used Adam Optimizer. The hyper-parameter setting description is given in a following section.

Hyper-parameter Setting:

After trial and error, we have used a learning rate for the Adam optimizer to be 0.005 with beta1=0.9 and beta2=0.999 and all other values to be default. We have also carefully chosen a batch size of 100 after playing with different batch sizes and running the code for several epochs. In order to regularize the model, we have used three dropout layers as described before. The first two dropout layers after the conv layers had a dropout probability of 0.25 and the third dropout layer after the fully connected layer had a dropout probability of 0.5 during the training phase. The testing phase ignores the dropout layer.

Plots of Loss and Accuracy vs Epochs:

The training has been done for a total of 100 epochs. After that, the test loss no longer decreases, rather starts an upward trend, indicating overfitting. The loss and accuracy curves are given below:

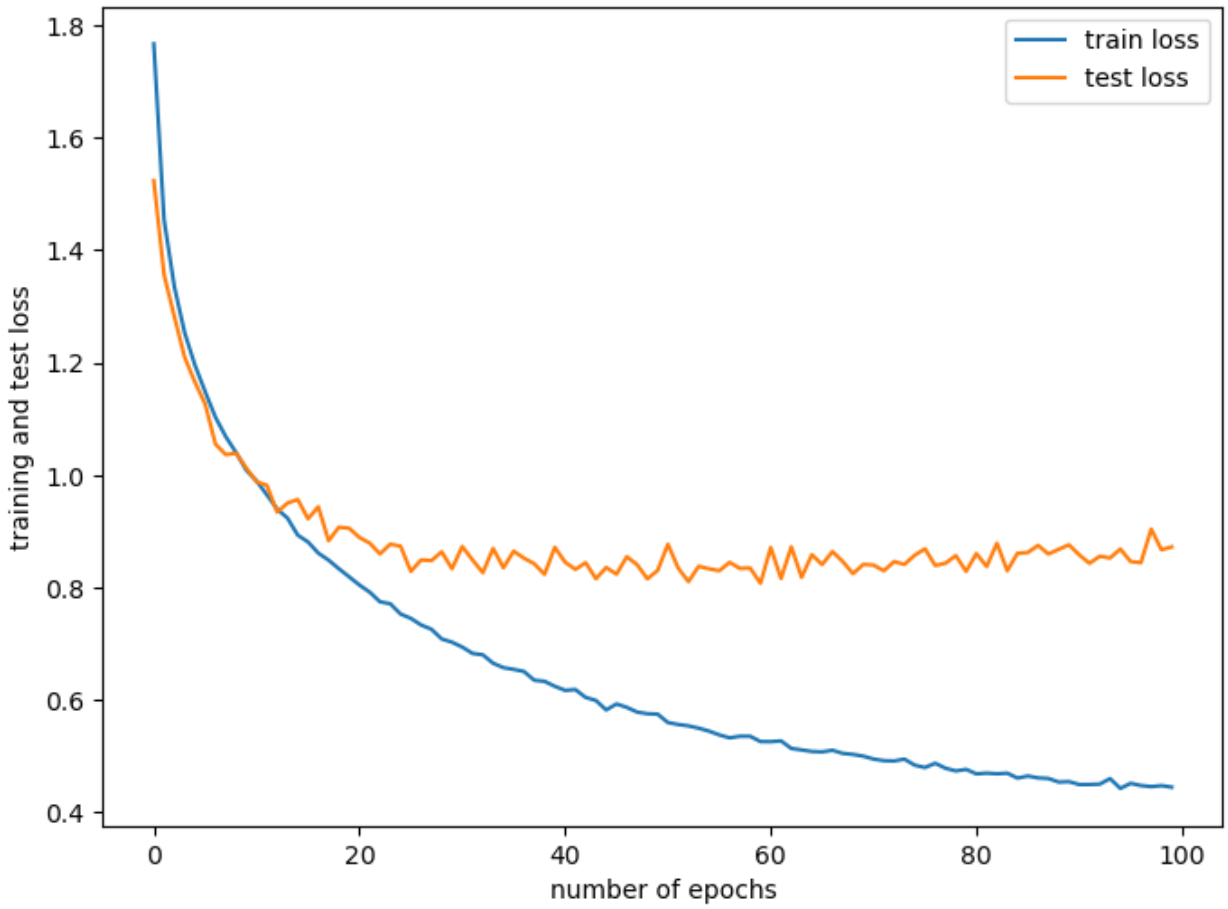


Figure: Train and Test Loss vs Epoch

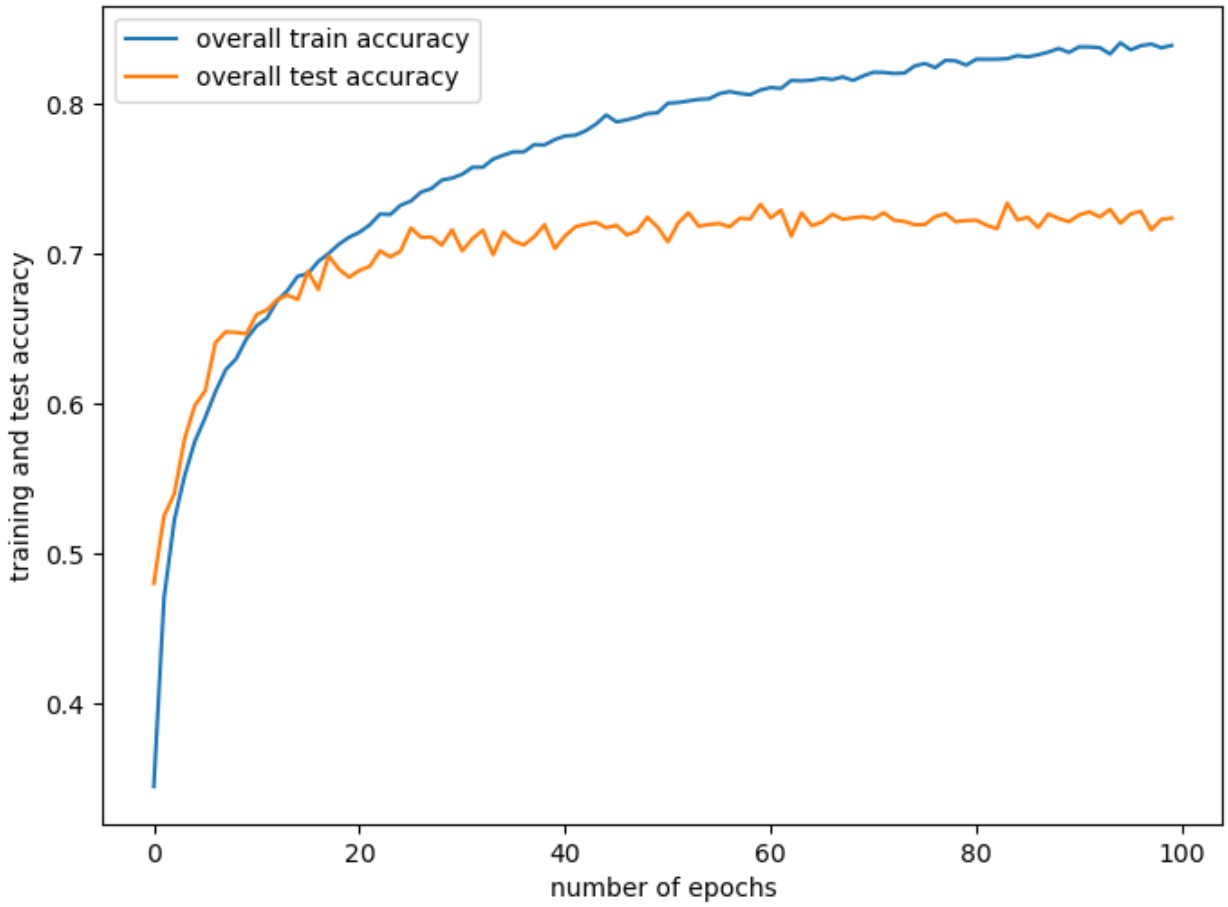


Figure: Train and Test Accuracy over Epoch

Result Analysis (Classification Error and Accuracies):

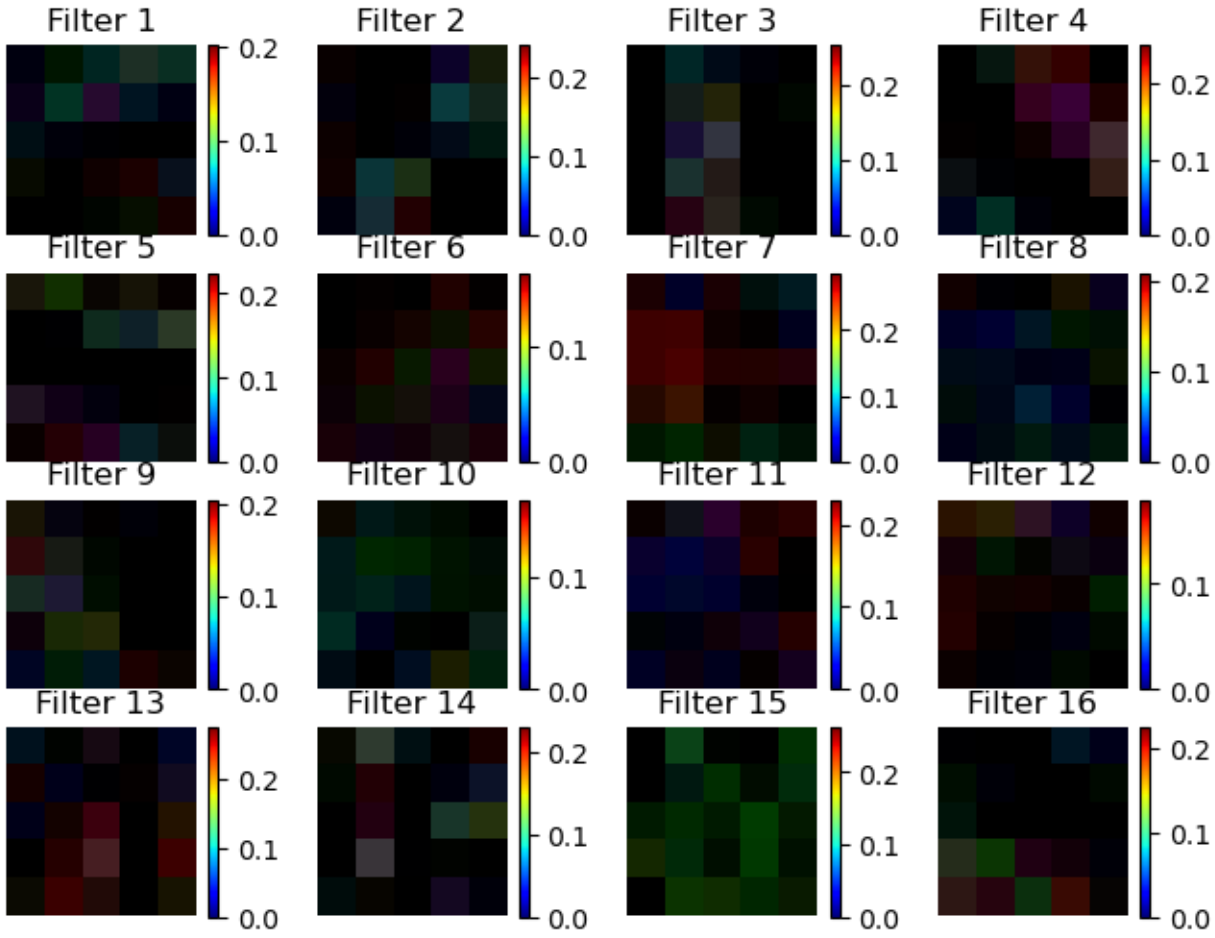
Using the learned parameters we calculate the training errors, test error, training accuracy and test accuracy. The results are shown below:

Item Name	Class 0	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6	Class 7	Class 8	Class 9	**Average**
Training Accuracy	0.9622	0.9668	0.9358	0.9136	0.9772	0.8774	0.9868	0.9508	0.9798	0.9512	**0.9502**
Training Error	0.0378	0.0332	0.0642	0.0864	0.0228	0.1226	0.0132	0.0492	0.0202	0.0488	**0.0498**
Test Accuracy	0.7233	0.796	0.625	0.525	0.794	0.573	0.855	0.735	0.841	0.766	**0.724**
Test Error	0.277	0.204	0.375	0.474	0.205	0.426	0.144	0.264	0.158	0.234	**0.276**

So, model model shows around 72.4% average accuracy on Test data.

Visualization of Learned Filters for First Convolution Layer:

The visualization of the learned filters for the first convolution layer is given below:



Discussions:

In this problem, we have employed a simple Convolutional Neural Network to perform the task of CIFAR-10 dataset classification. From the results, we see that, the model performs fairly well in training data with almost 95% accuracy, but despite several approaches tuning of hyper-parameters, the test accuracy is 72.4% for all classes. If we look at carefully on the per class results, we see that class 3 and class 5, corresponding to cats and dogs classes respectively have the lowest performance. The reason for the model's relatively low

performance on test data could have several reasons. First, the architecture chosen might not have the capacity to grasp the actual generalizability of the dataset. So, changing the architecture could increase performance. For example, at the beginning, I accidentally used 32 filters instead of 16 on the first convolutional layer, which was showing relatively better results in terms of test loss and test accuracy. In case of hyper-parameter selection, I have tried different learning rates to see the trend. A learning rate of 0.001 was making the training performance fluctuating a lot and a learning rate of 1e-4 was making the training very slow. After several attempts, learning rate of 0.005 showed reasonably good performance. Batch size was also affecting the result. Smaller batch sizes was making the training process very fluctuating so I chose a batch size of 100. In order to regularize the training, I have applied dropout layers during training. I used dropout probability of 0.25 for the first two convolution layers, whereas used a dropout probability of 0.5 on the fully connected layer. I did not use a dropout layer after conv3 layer, because the feature map size was already very small. Overall, implementing different knowledge gained from the course lectures we could get a moderately good performance from this small convolutional neural network for classification of CIFAR-10 dataset.

Appendix: Code For Training:

```
# %%
import tensorflow as tf
print("TensorFlow version:", tf.__version__)

from tensorflow.keras.layers import Dense, Flatten, Conv2D, Dropout, Activation, MaxPooling2D
from tensorflow.keras import Model
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import random
import pickle
from tqdm import tqdm
from sklearn.metrics import classification_report, confusion_matrix

# %%
train_data = np.load('training_data.npy')
tr_labels = np.load('training_label.npy')
test_data = np.load('testing_data.npy')
te_labels = np.load('testing_label.npy')

train_data = np.float32(train_data/255)
```

```
test_data = np.float32(test_data/255)

# %%
print(train_data.shape, tr_labels.shape, test_data.shape, te_labels.shape)
print(int(max(tr_labels)), int(max(te_labels)))

# %%
#function for 1 hot label encoding
def label_encoding(label):

    y = np.zeros([int(max(label))+1,len(label)])

    for i in range(len(label)):
        y[int(label[i]),i] = 1

    y=y.T
    return y

#function for computing classification performance
def performance_metrics(y_true,y_pred):
    matrix=confusion_matrix(y_true, np.argmax(y_pred,axis=1))
    accuracies=matrix.diagonal()/matrix.sum(axis=1)
    errors=1-accuracies
    return accuracies,errors

# %%
train_label = label_encoding(tr_labels)
train_label = np.float32(train_label)
test_label = label_encoding(te_labels)
test_label = np.float32(test_label)

print(train_label.shape, test_label.shape)

# %%
train_ds = tf.data.Dataset.from_tensor_slices((train_data, tr_labels)).shuffle(10000).batch(100)

test_ds = tf.data.Dataset.from_tensor_slices((test_data, te_labels)).batch(100)

# %%


# %%
class MyModel(Model):
  def __init__(self):
```

```
    super(MyModel, self).__init__()
    self.conv1=Conv2D(filters = 16,kernel_size = 5, padding='valid', strides = (1,1), activation =
'relu',input_shape=(32,32,3))
    self.dropout1 = Dropout(0.25)
    self.mp1=MaxPooling2D(pool_size=(2, 2), strides = (2,2))
    self.conv2=Conv2D(filters = 32,kernel_size = 5, padding='valid', strides = (1,1), activation = 'relu')
    self.mp2=MaxPooling2D(pool_size=(2, 2), strides = (2,2))
    self.conv3=Conv2D(filters = 64,kernel_size = 3, padding='valid', strides = (1,1), activation = 'relu')
    self.dropout2 = Dropout(0.25)
    self.flatten=Flatten()
    self.d1=Dense(units = 500, activation='relu')
    self.dropout3=Dropout(0.5)
    self.d2=Dense(units = 10, activation = 'softmax')
    #return model

  def call(self, x,training=False):
    x = self.conv1(x)
    if training:
      x = self.dropout1(x)
    x = self.mp1(x)
    x = self.conv2(x)
    x = self.mp2(x)
    x = self.conv3(x)
    if training:
        x = self.dropout2(x)
    x = self.flatten(x)
    x = self.d1(x)
    if training:
        x = self.dropout3(x)
    x = self.d2(x)
    return x

# Create an instance of the model
model = MyModel()

# %%
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False)

optimizer = tf.keras.optimizers.Adam(learning_rate=0.0005)

# %%
train_loss = tf.keras.metrics.Mean(name='train_loss')
train_acc = tf.keras.metrics.SparseCategoricalAccuracy(name='train_accuracy')

test_loss = tf.keras.metrics.Mean(name='test_loss')
```

```
test_acc = tf.keras.metrics.SparseCategoricalAccuracy(name='test_accuracy')

# %%
@tf.function
def training_function(images, labels):
  with tf.GradientTape() as tape:
    # training=True is only needed if there are layers with different
    # behavior during training versus inference (e.g. Dropout).
    predictions = model(images, training=True)
    loss = loss_object(labels, predictions)
  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))

  train_loss(loss)
  train_acc(labels, predictions)


@tf.function
def testing_function(images, labels):
  # training=False is only needed if there are layers with different
  # behavior during training versus inference (e.g. Dropout).
  predictions = model(images, training=False)
  t_loss = loss_object(labels, predictions)

  test_loss(t_loss)
  test_acc(labels, predictions)

# %%
EPOCHS = 100


train_errors=[]
train_accuracy=[]
test_errors=[]
test_accuracy=[]
train_losses=[]
test_losses=[]

for epoch in range(EPOCHS):
  # Reset the metrics at the start of the next epoch
  train_loss.reset_states()
  train_acc.reset_states()
  test_loss.reset_states()
  test_acc.reset_states()
```

```
  for images, labels in train_ds:
    training_function(images, labels)

  for test_images, test_labels in test_ds:
    testing_function(test_images, test_labels)

  a=train_acc.result()
  b=test_acc.result()
  c=train_loss.result()
  d=test_loss.result()
  print(
    f'Epoch {epoch + 1}, '
    f'Loss: {c}, '
    f'Accuracy: {a * 100}, '
    f'Test Loss: {d}, '
    f'Test Accuracy: {b * 100}'
  )
  train_losses.append(c.numpy())
  test_losses.append(d.numpy())
  train_accuracy.append(a.numpy())
  test_accuracy.append(b.numpy())

# %%
print(train_data.shape,test_data.shape)

y_train_pred = []
#Performance calculation
for i in range(50):
    x=train_data[i*1000:(i+1)*1000]
    y=model(x)
    y_train_pred.append(y)
y_train_pred=np.concatenate(y_train_pred,axis=0)

print(y_train_pred.shape)


y_test_pred = []
for i in range(5):
    x=test_data[i*1000:(i+1)*1000]
    y=model(x)
    y_test_pred.append(y)
y_test_pred=np.concatenate(y_test_pred,axis=0)

print(y_test_pred.shape)
```

```
# %%
acc,err=performance_metrics(tr_labels,y_train_pred)
print("Train Accuracy:",acc)
print("Train Error:",err)

acctest,errtest=performance_metrics(te_labels,y_test_pred)
print("Test Accuracy:",acctest)
print("Test Error:",errtest)



# %%
print("Average train accuracy:",np.mean(acc))
print("Average test accuracy:",np.mean(acctest))
print("Average train error:",np.mean(err))
print("Average test error:",np.mean(errtest))

# %%
plt.figure(figsize=(8, 6), dpi=100)
plt.plot(train_accuracy, label = "overall train accuracy")
plt.plot(test_accuracy, label ="overall test accuracy")
plt.legend()
plt.xlabel('number of epochs')
plt.ylabel('training and test accuracy')
plt.show()

# %%
plt.figure(figsize=(8, 6), dpi=100)
plt.plot(train_losses, label = "train loss")
plt.plot(test_losses, label = "test loss")
plt.xlabel('number of epochs')
plt.ylabel('training and test loss')
plt.legend()
plt.show()

# %%
weights_conv =  np.array(model.conv1.get_weights()[0])
print(weights_conv.shape)
plt.figure(figsize=(8, 6), dpi=100)
for i in range(16):
    image = weights_conv[:,:,:,i]
    plt.subplot(4,4,i+1)
    plt.imshow(image,cmap='jet')
    plt.colorbar()
    plt.axis('off')
```

```
    plt.title('Filter {}'.format(i+1))
plt.show()

# %%
model.save_weights("model_weights")

# %%
```