HMM:

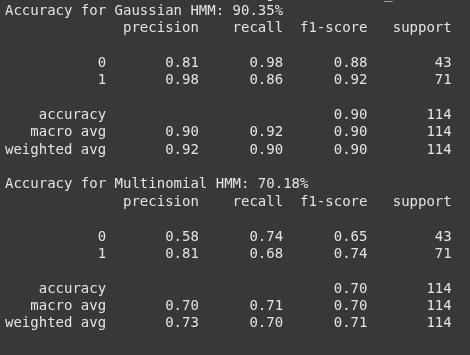
The Hidden Markov Model is trained on two datasets such as: Wisconsin Breast Cancer Classification and Ionosphere Classification. Two different models GaussianHMM and CategoricalHMM (Multinomial model).

The breast cancer classification without parameter-tuning is done by normalizing the dataset, along with reducing the dimension of the dataset to accommodate maximum variance within minimum features, and deciding to reduce the overfitting from noise present in the data. It is done as follows:

| **def** **load\_train\_breast\_cancer**():  X, y = load\_breast\_cancer(return\_X\_y=**True**)  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)    scaler = StandardScaler()  X\_train = scaler.fit\_transform(X\_train)  X\_test = scaler.transform(X\_test)    pca = PCA(n\_components= 5)  X\_train = pca.fit\_transform(X\_train)  X\_test = pca.transform(X\_test)   discretizer = KBinsDiscretizer(n\_bins=10, encode='ordinal', strategy='quantile')  cX\_train = discretizer.fit\_transform(X\_train).astype(int)  cX\_test = discretizer.transform(X\_test).astype(int)   X\_train\_0 = X\_train[y\_train==0]  X\_train\_1 = X\_train[y\_train==1]   ghmm0 = GaussianHMM(n\_components=3, covariance\_type='full', n\_iter=500).fit(X\_train\_0)  ghmm1 = GaussianHMM(n\_components=3, covariance\_type='full', n\_iter=500).fit(X\_train\_1)   ghmm0.fit(X\_train\_0)  ghmm1.fit(X\_train\_1)   n\_symbols = int(np.max([np.max(cX\_train[y\_train==0]), np.max(cX\_train[y\_train==1])]) + 1)   chmm0 = CategoricalHMM(n\_components=3, n\_iter=500, random\_state=42, n\_features=n\_symbols)  chmm1 = CategoricalHMM(n\_components=3, n\_iter=500, random\_state=42, n\_features=n\_symbols)   chmm0.fit(cX\_train[y\_train==0])  chmm1.fit(cX\_train[y\_train==1])   gpreds = []  **for** x **in** X\_test:  score0 = ghmm0.score([x])  score1 = ghmm1.score([x])  gpreds.append(1 **if** score1 > score0 **else** 0)   gacc = np.mean(gpreds == y\_test)   cpreds = []  **for** x **in** cX\_test:  score0 = chmm0.score([x])  score1 = chmm1.score([x])  cpreds.append(1 **if** score1 > score0 **else** 0)   cacc= np.mean(cpreds==y\_test)   print(f"Accuracy for Gaussian HMM: {gacc\*100:.2f}%")  print(classification\_report(y\_test, gpreds))   print(f"Accuracy for Multinomial HMM: {cacc\*100:.2f}%")  print(classification\_report(y\_test, cpreds)) |
| --- |

| load\_train\_breast\_cancer() |
| --- |

The post-training metrics are:



The training using ionosphere dataset done by first normalizing the dataset and reducing the dimension of the dataset to accommodate maximum variance in data while keeping the feature size minimum:

def load\_train\_ionosphere():

#from sklearn.datasets import fetch\_openml

iono = fetch\_openml(name="ionosphere", version=1, as\_frame=True)

X = iono.data.values

y = np.array([1 if v == 'g' else 0 for v in iono.target])

print(X.shape, y.shape)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

pca = PCA(n\_components= 5)

X\_train = pca.fit\_transform(X\_train)

X\_test = pca.transform(X\_test)

from sklearn.preprocessing import KBinsDiscretizer

discretizer = KBinsDiscretizer(n\_bins=10, encode='ordinal', strategy='quantile')

cX\_train = discretizer.fit\_transform(X\_train).astype(int)

cX\_test = discretizer.transform(X\_test).astype(int)

X\_train\_0 = X\_train[y\_train==0]

X\_train\_1 = X\_train[y\_train==1]

ghmm0 = GaussianHMM(n\_components=3, covariance\_type='full', n\_iter=500).fit(X\_train\_0)

ghmm1 = GaussianHMM(n\_components=3, covariance\_type='full', n\_iter=500).fit(X\_train\_1)

ghmm0.fit(X\_train\_0)

ghmm1.fit(X\_train\_1)

n\_symbols = int(np.max([np.max(cX\_train[y\_train==0]), np.max(cX\_train[y\_train==1])]) + 1)

chmm0 = CategoricalHMM(n\_components=3, n\_iter=500, random\_state=42, n\_features=n\_symbols)

chmm1 = CategoricalHMM(n\_components=3, n\_iter=500, random\_state=42, n\_features=n\_symbols)

chmm0.fit(cX\_train[y\_train==0])

chmm1.fit(cX\_train[y\_train==1])

gpreds = []

for x in X\_test:

score0 = ghmm0.score([x])

score1 = ghmm1.score([x])

gpreds.append(1 if score1 > score0 else 0)

gacc = np.mean(gpreds == y\_test)

cpreds = []

for x in cX\_test:

score0 = chmm0.score([x])

score1 = chmm1.score([x])

cpreds.append(1 if score1 > score0 else 0)

cacc= np.mean(cpreds==y\_test)

print(f"Accuracy for Gaussian HMM: {gacc\*100:.2f}%")

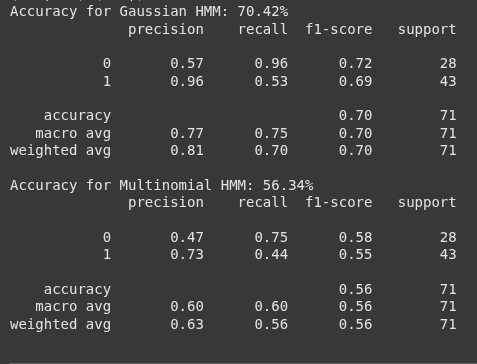
print(classification\_report(y\_test, gpreds))

print(f"Accuracy for Multinomial HMM: {cacc\*100:.2f}%")

print(classification\_report(y\_test, cpreds))

| load\_train\_ionosphere() |
| --- |

The post-training metrics are:



Using Optuna for parameter-tuning for both the datasets. Below is the approach to apply parameter-tuning.

def load\_datasets():

X\_bc, y\_bc = load\_breast\_cancer(return\_X\_y=True, as\_frame=False)

iono = fetch\_openml(name="ionosphere", version=1, as\_frame=True)

X\_iono = iono.data.values

y\_iono = np.array([1 if v == 'g' else 0 for v in iono.target])

return (X\_bc, y\_bc, "BreastCancer"), (X\_iono, y\_iono, "Ionosphere")

def train\_hmm(X\_train, X\_test, y\_train, y\_test, model\_type="gaussian", n\_components=3, pca\_dim=5):

if model\_type == "gaussian":

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

if X\_train.shape[1] > pca\_dim:

pca = PCA(n\_components=pca\_dim)

X\_train = pca.fit\_transform(X\_train)

X\_test = pca.transform(X\_test)

if model\_type == "multinomial":

discretizer = KBinsDiscretizer(n\_bins=10, encode='ordinal', strategy='quantile')

X\_train = discretizer.fit\_transform(X\_train).astype(int)

X\_test = discretizer.transform(X\_test).astype(int)

X0 = X\_train[y\_train == 0]

X1 = X\_train[y\_train == 1]

print(f"X0 shape: {X0.shape}")

print(f"X1 shape: {X1.shape}")

if model\_type == "gaussian":

optuna\_tuning= OptunaGaussianTuning(X\_train, y\_train, X\_test, y\_test)

params= optuna\_tuning.objective()

n\_components= params.best\_params["n\_components"]

covariance\_type= params.best\_params["covariance\_type"]

n\_iter= params.best\_params["n\_iter"]

model0 = GaussianHMM(n\_components=n\_components, covariance\_type=covariance\_type, n\_iter=n\_iter, random\_state=42)

model1 = GaussianHMM(n\_components=n\_components, covariance\_type=covariance\_type, n\_iter=n\_iter, random\_state=42)

else:

n\_symbols = int(np.max([np.max(X0), np.max(X1)]) + 1)

optuna\_tuning= OptunaCategoricalTuning(X\_train, y\_train, X\_test, y\_test)

params= optuna\_tuning.objective()

n\_components= params.best\_params["n\_components"]

n\_iter= params.best\_params["n\_iter"]

model0 = CategoricalHMM(n\_components=n\_components, n\_iter=n\_iter, random\_state=42, n\_features=n\_symbols)

model1 = CategoricalHMM(n\_components=n\_components, n\_iter=n\_iter, random\_state=42, n\_features=n\_symbols)

model0.fit(X0)

model1.fit(X1)

y\_pred = []

for x in X\_test:

x = np.expand\_dims(x, axis=0)

try:

score0 = model0.score(x)

score1 = model1.score(x)

except ValueError:

score0, score1 = -np.inf, -np.inf

y\_pred.append(1 if score1 > score0 else 0)

if model\_type!='gaussian':

return accuracy\_score(y\_test, y\_pred), n\_components, n\_iter, None

else:

return accuracy\_score(y\_test, y\_pred), n\_components, n\_iter, covariance\_type

def run\_experiments():

datasets = load\_datasets()

test\_sizes = [0.2, 0.1, 0.3]

models = ["gaussian", "multinomial"]

results = []

for X, y, name in datasets:

for test\_size in test\_sizes:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=test\_size, random\_state=42, stratify=y)

for model\_type in models:

acc, n\_components, n\_iter, covariance\_type = train\_hmm(X\_train, X\_test, y\_train, y\_test, model\_type, n\_components=4, pca\_dim=6)

results.append({

"Dataset": name,

"Model": model\_type.title(),

"Test\_Size": f"{int(test\_size\*100)}-{int((1-test\_size)\*100)}",

"Accuracy": round(acc\*100, 2),

"n\_components": n\_components,

"n\_iter": n\_iter,

"covariance\_type": 'NAN' if covariance\_type is None else covariance\_type

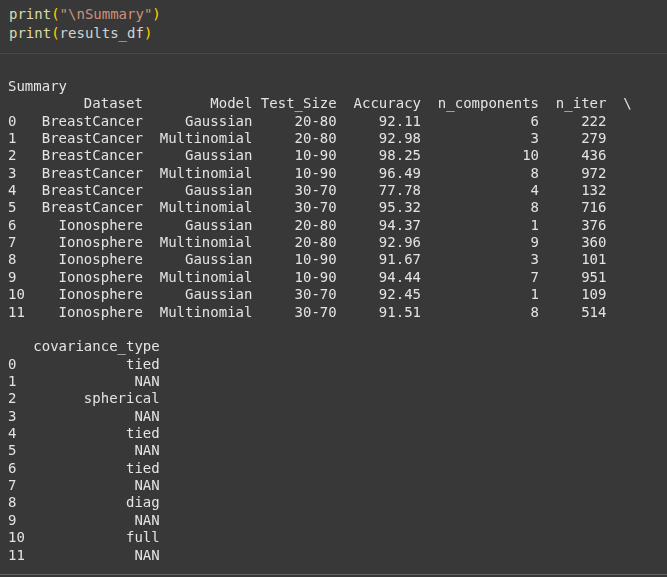
})

df = pd.DataFrame(results)

return df

results\_df = run\_experiments()

Below is the summary generated after training data using parameter-tuning. Different dataset sizes and model type- Gaussian and Multinomial are used with different parameters and the accuracy is noted.



The best of the parameters are listed below. The parameters are used to retrain the models and then the metrics are registered.

parameters= {

"BreastCancer":{

"test\_size": 0.1,

"n\_components": 10,

"n\_iter": 436,

"covariance\_type": "spherical",

"model\_type": "gaussian"

},

"Ionosphere": {

"test\_size": 0.1,

"n\_components": 7,

"n\_iter": 951,

"model\_type": "multinomial"

}

}

def train\_best\_(parameters):

datasets = load\_datasets()

for X, y, name in datasets:

params = parameters[name]

x\_train, x\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=params["test\_size"], random\_state=42, stratify=y)

if params["model\_type"] == "gaussian":

scaler = StandardScaler()

x\_train\_processed = scaler.fit\_transform(x\_train)

x\_test\_processed = scaler.transform(x\_test)

pca = PCA(n\_components=6)

x\_train\_processed = pca.fit\_transform(x\_train\_processed)

x\_test\_processed = pca.transform(x\_test\_processed)

model0 = GaussianHMM(n\_components=params["n\_components"], covariance\_type=params["covariance\_type"], n\_iter=params["n\_iter"], random\_state=42)

model1 = GaussianHMM(n\_components=params["n\_components"], covariance\_type=params["covariance\_type"], n\_iter=params["n\_iter"], random\_state=42)

x0\_train = x\_train\_processed[y\_train == 0]

x1\_train = x\_train\_processed[y\_train == 1]

model0.fit(x0\_train)

model1.fit(x1\_train)

preds = []

preds\_prob = []

for x in x\_test\_processed:

x = np.expand\_dims(x, axis=0)

score0 = model0.score(x)

score1 = model1.score(x)

preds.append(1 if score1 > score0 else 0)

# Calculate pseudo-probabilities for ROC curve

total\_score = score0 + score1

prob0 = score0 / total\_score if total\_score != 0 else 0.5

prob1 = score1 / total\_score if total\_score != 0 else 0.5

preds\_prob.append([prob0, prob1])

preds\_prob = np.array(preds\_prob)

print(preds\_prob.shape)

acc = accuracy\_score(y\_test, preds)

score\_f1 = f1\_score(y\_test, preds)

precision = precision\_score(y\_test, preds)

recall = recall\_score(y\_test, preds)

print(f"\n--- Results for {name} ({params['model\_type']} model) ---")

print(f"Accuracy: {acc\*100:.2f}%")

print(f"F1 Score: {score\_f1\*100:.2f}%")

print(f"Precision: {precision\*100:.2f}%")

print(f"Recall: {recall\*100:.2f}%")

plot\_confusion(y\_test, preds, f'{name}\_{params["test\_size"]}\_{params["model\_type"]}')

plot\_roc\_auc(y\_test, preds\_prob, f'{name}\_{params["test\_size"]}\_{params["model\_type"]}')

elif params["model\_type"] == "multinomial":

discretizer = KBinsDiscretizer(n\_bins=10, encode='ordinal', strategy='quantile')

x\_train\_processed = discretizer.fit\_transform(x\_train).astype(int)

x\_test\_processed = discretizer.transform(x\_test).astype(int)

x0\_train = x\_train\_processed[y\_train == 0]

x1\_train = x\_train\_processed[y\_train == 1]

n\_symbols = int(np.max([np.max(x0\_train), np.max(x1\_train)]) + 1)

model0 = CategoricalHMM(n\_components=params["n\_components"], n\_iter=params["n\_iter"], random\_state=42, n\_features=n\_symbols)

model1 = CategoricalHMM(n\_components=params["n\_components"], n\_iter=params["n\_iter"], random\_state=42, n\_features=n\_symbols)

model0.fit(x0\_train)

model1.fit(x1\_train)

preds = []

preds\_prob= []

for x in x\_test\_processed:

x = np.expand\_dims(x, axis=0)

score0 = model0.score(x)

score1 = model1.score(x)

preds.append(1 if score1 > score0 else 0)

# Calculate pseudo-probabilities for ROC curve

total\_score = score0 + score1

prob0 = score0 / total\_score if total\_score != 0 else 0.5

prob1 = score1 / total\_score if total\_score != 0 else 0.5

preds\_prob.append([prob0, prob1])

preds\_prob = np.array(preds\_prob)

acc = accuracy\_score(y\_test, preds)

score\_f1 = f1\_score(y\_test, preds)

precision = precision\_score(y\_test, preds)

recall = recall\_score(y\_test, preds)

print(f"\n--- Results for {name} ({params['model\_type']} model) ---")

print(f"Accuracy: {acc\*100:.2f}%")

print(f"F1 Score: {score\_f1\*100:.2f}%")

print(f"Precision: {precision\*100:.2f}%")

print(f"Recall: {recall\*100:.2f}%")

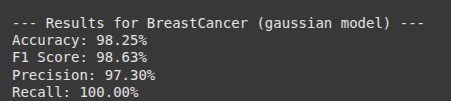
plot\_confusion(y\_test, preds, f'{name}\_{params["test\_size"]}\_{params["model\_type"]}')

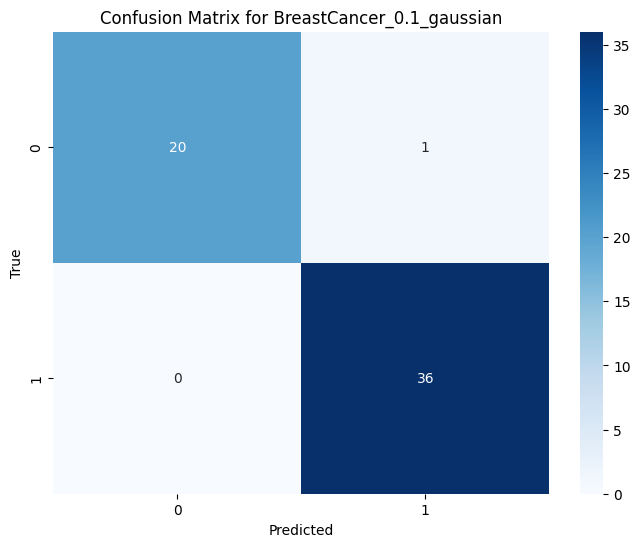
plot\_roc\_auc(y\_test, preds\_prob, f'{name}\_{params["test\_size"]}\_{params["model\_type"]}')

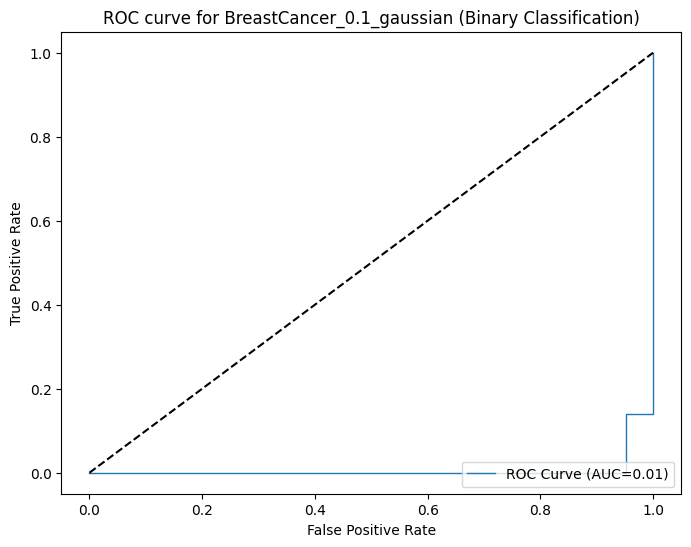
else:

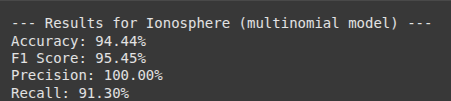
print(f"Unknown model type: {params['model\_type']}")

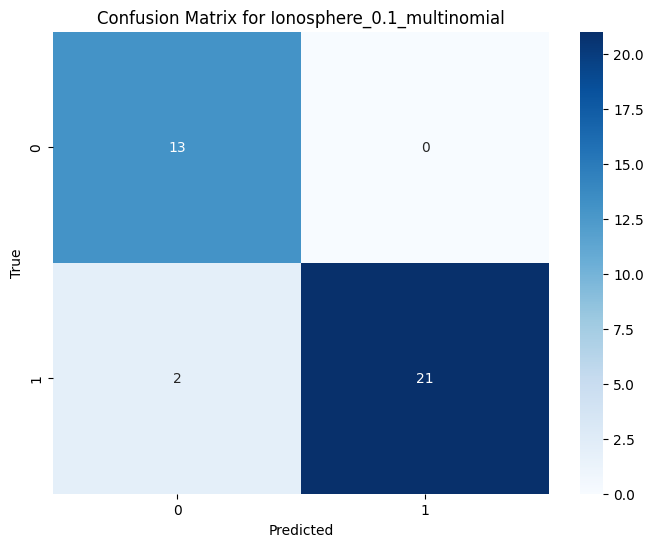
train\_best\_(parameters)

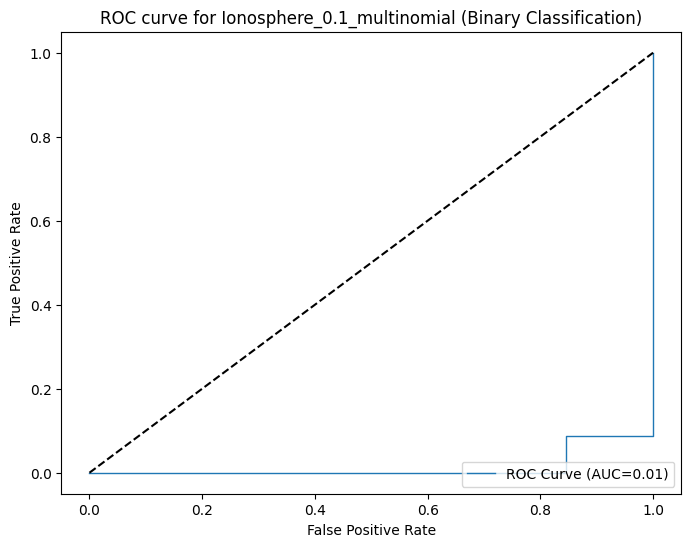












Training Cifar10 and Mnist over CNN

The Cifar10 dataset is a collection of images that are commonly used to train machine learning and computer vision algorithms. It consists of 10 classes of image groups. The Mnist on the other hand is another collection of handwritten digits having numerical images of 0 to 9.

CNN is used to classify images to 10 classes. Image preprocessing techniques such as image augmentation is applied to introduce diversity in images, which further help prevent overfitting in the model training.

(mnist\_train\_images, mnist\_train\_labels), (mnist\_test\_images, mnist\_test\_labels)= tf.keras.datasets.mnist.load\_data()

(cifar\_train\_images, cifar\_train\_labels), (cifar\_test\_images, cifar\_test\_labels)= tf.keras.datasets.cifar10.load\_data()

Below script implement image augmentation:

def get\_image\_augmentation():

train\_datagen = ImageDataGenerator(

rotation\_range=20,

shear\_range=0.2,

zoom\_range=0.2,

horizontal\_flip=True,

fill\_mode='nearest',

rescale= 1./255

)

test\_datagen= ImageDataGenerator(rescale= 1./255)

return train\_datagen, test\_datagen

The CNN model is defined below, which takes input image shape to extract image features, and learns about the image features which helps the model in classifying images into 10 classes.

def simple\_cnn(input\_shape, classes= 10):

model= tf.keras.models.Sequential(

[

tf.keras.layers.Conv2D(64, (3, 3), activation="relu",padding="same", input\_shape= input\_shape),

tf.keras.layers.BatchNormalization(),

tf.keras.layers.MaxPool2D(2, strides=2, padding="valid"),

tf.keras.layers.Conv2D(128, (3, 3), activation="relu", padding="same"),

tf.keras.layers.BatchNormalization(),

tf.keras.layers.MaxPooling2D(2, strides= 2, padding="valid"),

tf.keras.layers.Dropout(0.2),

tf.keras.layers.Conv2D(256, (3, 3), activation= "relu", padding="same"),

tf.keras.layers.BatchNormalization(),

tf.keras.layers.Conv2D(256, (3, 3), activation= "relu", padding="same"),

tf.keras.layers.BatchNormalization(),

tf.keras.layers.MaxPooling2D(2, strides= 2, padding="valid"),

tf.keras.layers.Dropout(0.2),

tf.keras.layers.Conv2D(512, (3, 3), activation= "relu", padding="same"),

tf.keras.layers.BatchNormalization(),

tf.keras.layers.Conv2D(512, (3, 3), activation= "relu", padding="same"),

tf.keras.layers.BatchNormalization(),

tf.keras.layers.MaxPooling2D(2, strides= 2, padding="valid"),

tf.keras.layers.Dropout(0.3),

tf.keras.layers.Flatten(),

tf.keras.layers.Dense(256, activation="relu"),

tf.keras.layers.Dropout(0.3),

tf.keras.layers.Dense(128, activation="relu"),

tf.keras.layers.Dense(classes, activation="softmax")

]

)

return model

Model training for Mnist is implemented below:

mnist\_train\_datagen, mnist\_test\_datagen= get\_image\_augmentation()

mnist\_train\_images = np.expand\_dims(mnist\_train\_images, axis=-1)

mnist\_test\_images = np.expand\_dims(mnist\_test\_images, axis=-1)

mnist\_train\_datagen.fit(mnist\_train\_images)

mnist\_test\_datagen.fit(mnist\_test\_images)

mnist\_model= simple\_cnn([28, 28, 1], 10)

mnist\_model.compile(optimizer="adam", loss="sparse\_categorical\_crossentropy", metrics=["accuracy"])

mnist\_history= mnist\_model.fit(

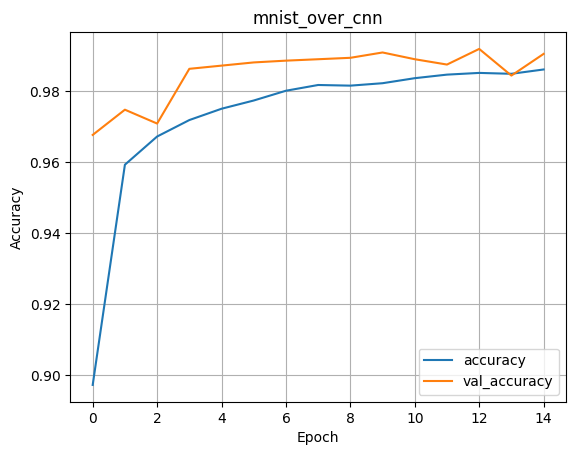
mnist\_train\_datagen.flow(mnist\_train\_images, mnist\_train\_labels),

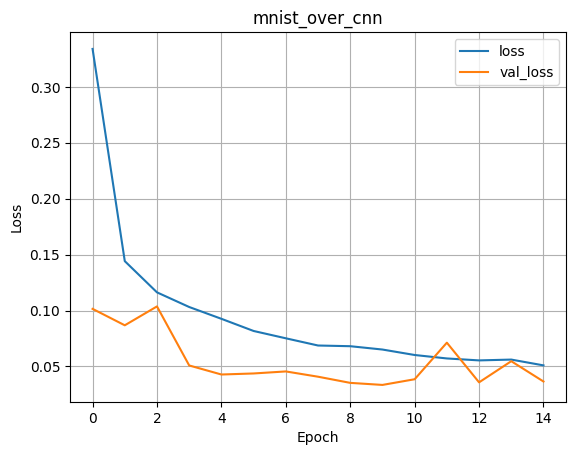
epochs=15,

validation\_data=mnist\_test\_datagen.flow(mnist\_test\_images, mnist\_test\_labels),

callbacks= [tf.keras.callbacks.EarlyStopping(patience= 7, monitor='val\_loss')])

The model training accuracy and loss curve is plotted below:





The model training for Cifar10 is implemented below:

cifar\_train\_datagen, cifar\_test\_datagen= get\_image\_augmentation()

cifar\_train\_datagen.fit(cifar\_train\_images)

cifar\_test\_datagen.fit(cifar\_test\_images)

cifar\_model= simple\_cnn([32, 32, 3], 10)

cifar\_model.compile(optimizer="adam", loss="sparse\_categorical\_crossentropy", metrics=["accuracy"])

cifar\_history= cifar\_model.fit(

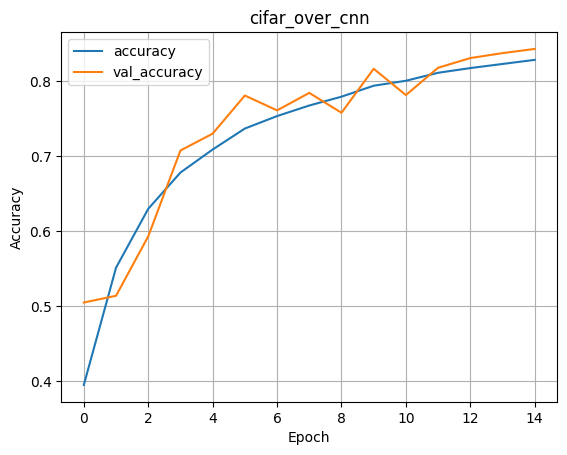
cifar\_train\_datagen.flow(cifar\_train\_images, cifar\_train\_labels),

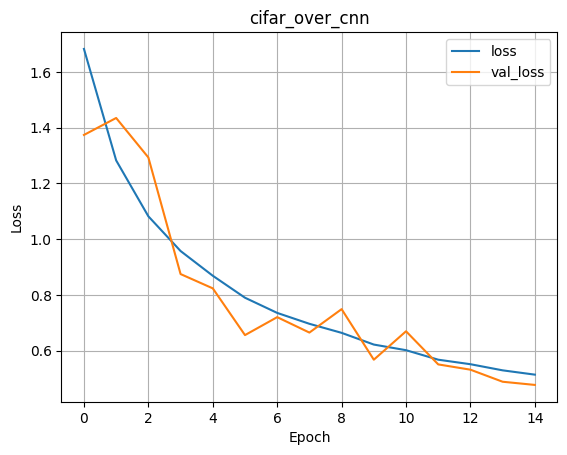
epochs=15,

validation\_data=cifar\_test\_datagen.flow(cifar\_test\_images, cifar\_test\_labels),

callbacks= [tf.keras.callbacks.EarlyStopping(patience= 7, monitor='val\_loss')])

The model training and validation accuracy and loss is plotted below:





Next we move onto another task, where we will try to train 5 models such as: VGG16, AlexNet, GoogLeNet, RNN, CNN on the Mnist dataset. The training of models will be done on two splits of datasets such as 80% of dataset and 90% of dataset.

First try to get the datasets and merge the train and test dataset as further down, they will be split into 80% and 90% of data while training the models.

| def get\_data(dataset\_name):   if dataset\_name=='cifar10':  (cifar\_train\_images, cifar\_train\_labels), (cifar\_test\_images, cifar\_test\_labels)= tf.keras.datasets.cifar10.load\_data()    cifar\_images= np.concatenate([cifar\_train\_images, cifar\_test\_images], axis=0)  cifar\_labels= np.concatenate([cifar\_train\_labels, cifar\_test\_labels], axis=0)    print(cifar\_images.shape, cifar\_labels.shape)  return cifar\_images, cifar\_labels   elif dataset\_name=="mnist":  (mnist\_train\_images, mnist\_train\_labels), (mnist\_test\_images, mnist\_test\_labels)= tf.keras.datasets.mnist.load\_data()  mnist\_train\_images = np.expand\_dims(mnist\_train\_images, axis=-1)  mnist\_test\_images = np.expand\_dims(mnist\_test\_images, axis=-1)    mnist\_images= np.concatenate([mnist\_train\_images, mnist\_test\_images], axis=0)  mnist\_labels= np.concatenate([mnist\_train\_labels, mnist\_test\_labels], axis=0)  return mnist\_images, mnist\_labels |
| --- |

Then the data augmentation that will be applied is defined:

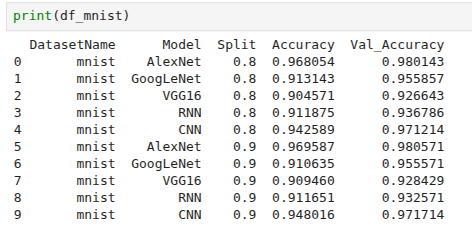
| def get\_image\_augmentation():  train\_datagen = ImageDataGenerator(  rotation\_range=20,  shear\_range=0.2,  zoom\_range=0.2,  horizontal\_flip=True,  fill\_mode='nearest',  rescale= 1./255  )   test\_datagen= ImageDataGenerator(rescale= 1./255)   return train\_datagen, test\_datagen |
| --- |

Then, the models are defined which will be further trained.

| def simple\_cnn(input\_shape, classes= 10):  model= tf.keras.models.Sequential(  [  tf.keras.layers.Conv2D(64, (3, 3), activation="relu",padding="same", input\_shape= input\_shape),  tf.keras.layers.BatchNormalization(),  tf.keras.layers.MaxPool2D(2, strides=2, padding="valid"),  tf.keras.layers.Conv2D(128, (3, 3), activation="relu", padding="same"),  tf.keras.layers.BatchNormalization(),  tf.keras.layers.MaxPooling2D(2, strides= 2, padding="valid"),  tf.keras.layers.Dropout(0.2),   tf.keras.layers.Conv2D(256, (3, 3), activation= "relu", padding="same"),  tf.keras.layers.BatchNormalization(),  tf.keras.layers.Conv2D(256, (3, 3), activation= "relu", padding="same"),  tf.keras.layers.BatchNormalization(),  tf.keras.layers.MaxPooling2D(2, strides= 2, padding="valid"),  tf.keras.layers.Dropout(0.2),   tf.keras.layers.Conv2D(512, (3, 3), activation= "relu", padding="same"),  tf.keras.layers.BatchNormalization(),  tf.keras.layers.Conv2D(512, (3, 3), activation= "relu", padding="same"),  tf.keras.layers.BatchNormalization(),  tf.keras.layers.MaxPooling2D(2, strides= 2, padding="valid"),  tf.keras.layers.Dropout(0.3),   tf.keras.layers.Flatten(),  tf.keras.layers.Dense(256, activation="relu"),  tf.keras.layers.Dropout(0.3),  tf.keras.layers.Dense(128, activation="relu"),  tf.keras.layers.Dense(classes, activation="softmax")  ]  )  return model  def build\_vgg\_cifar(input\_shape, num\_classes, freeze\_base=True):  base = VGG16(weights='imagenet', include\_top=False, input\_shape=input\_shape)   if freeze\_base:  base.trainable = False   for layers in base.layers[-10:]:  layers.trainable=True    model= tf.keras.models.Sequential(  [  tf.keras.layers.RandomFlip("horizontal"),  tf.keras.layers.RandomRotation(0.1),  tf.keras.layers.Resizing(224, 224),  tf.keras.layers.Rescaling(1./255),  #preprocess\_input,  base,  tf.keras.layers.GlobalAveragePooling2D(),  tf.keras.layers.Dense(256, activation='relu'),  tf.keras.layers.Dropout(0.5),  tf.keras.layers.Dense(num\_classes, activation='softmax')  ])  return model   # this is for mnist dataset def build\_vgg16\_transfer(input\_shape, num\_classes, freeze\_base=True):  base = VGG16(weights='imagenet', include\_top=False, input\_shape=input\_shape)  if freeze\_base:  base.trainable = False  model= tf.keras.models.Sequential(  [  tf.keras.layers.RandomFlip("horizontal"),  tf.keras.layers.RandomRotation(0.1),  tf.keras.layers.Resizing(224, 224),  tf.keras.layers.Rescaling(1./255),  #preprocess\_input,  base,  tf.keras.layers.GlobalAveragePooling2D(),  tf.keras.layers.Dense(256, activation='relu'),  tf.keras.layers.Dropout(0.5),  tf.keras.layers.Dense(num\_classes, activation='softmax')  ])  return model   def build\_alexnet(input\_shape, num\_classes):  # A Keras-style AlexNet (simplified)   inp= tf.keras.layers.Input(shape=[None, None, 3])  x = tf.keras.layers.RandomFlip("horizontal")(inp)  x = tf.keras.layers.RandomRotation(0.1)(x)  x = tf.keras.layers.Resizing(224, 224)(x)  x = tf.keras.layers.Rescaling(1./255)(x)  x = tf.keras.layers.Conv2D(96, (11,11), strides=4, activation='relu', padding='same')(x)  x = tf.keras.layers.MaxPooling2D((3,3), strides=2)(x)  x = tf.keras.layers.Conv2D(256, (5,5), activation='relu', padding='same')(x)  x = tf.keras.layers.MaxPooling2D((3,3), strides=2)(x)  x = tf.keras.layers.Conv2D(384, (3,3), activation='relu', padding='same')(x)  x = tf.keras.layers.Conv2D(384, (3,3), activation='relu', padding='same')(x)  x = tf.keras.layers.Conv2D(256, (3,3), activation='relu', padding='same')(x)  x = tf.keras.layers.MaxPooling2D((3,3), strides=2)(x)  x = tf.keras.layers.Flatten()(x)  x = tf.keras.layers.Dense(4096, activation='relu')(x)  x = tf.keras.layers.Dropout(0.5)(x)  x = tf.keras.layers.Dense(4096, activation='relu')(x)  x = tf.keras.layers.Dropout(0.5)(x)  out = tf.keras.layers.Dense(num\_classes, activation='softmax')(x)  model = models.Model(inp, out)  return model   def inception\_module(x, filters):  # filters: tuple/list (f1, f3r, f3, f5r, f5, poolproj)  f1, f3r, f3, f5r, f5, poolproj = filters  path1 = tf.keras.layers.Conv2D(f1, (1,1), padding='same', activation='relu')(x)  path2 = tf.keras.layers.Conv2D(f3r, (1,1), padding='same', activation='relu')(x)  path2 = tf.keras.layers.Conv2D(f3, (3,3), padding='same', activation='relu')(path2)  path3 = tf.keras.layers.Conv2D(f5r, (1,1), padding='same', activation='relu')(x)  path3 = tf.keras.layers.Conv2D(f5, (5,5), padding='same', activation='relu')(path3)  path4 = tf.keras.layers.MaxPooling2D((3,3), strides=1, padding='same')(x)  path4 = tf.keras.layers.Conv2D(poolproj, (1,1), padding='same', activation='relu')(path4)  return tf.keras.layers.concatenate([path1, path2, path3, path4], axis=-1)   def build\_googlenet\_like(input\_shape, num\_classes):   inp = tf.keras.layers.Input(shape=[None, None, 3])  x = tf.keras.layers.RandomFlip("horizontal")(inp)  x = tf.keras.layers.RandomRotation(0.1)(x)  x = tf.keras.layers.Resizing(224, 224)(x)  x = tf.keras.layers.Rescaling(1./255)(x)  x = tf.keras.layers.Conv2D(64, (7,7), strides=2, padding='same', activation='relu')(x)  x = tf.keras.layers.MaxPooling2D((3,3), strides=2, padding='same')(x)  x = tf.keras.layers.Conv2D(64, (1,1), activation='relu')(x)  x = tf.keras.layers.Conv2D(192, (3,3), padding='same', activation='relu')(x)  x = tf.keras.layers.MaxPooling2D((3,3), strides=2, padding='same')(x)  # a few inception modules (small)  x = inception\_module(x, (64, 96, 128, 16, 32, 32))  x = inception\_module(x, (128, 128, 192, 32, 96, 64))  x = tf.keras.layers.MaxPooling2D((3,3), strides=2, padding='same')(x)  x = tf.keras.layers.GlobalAveragePooling2D()(x)  x = tf.keras.layers.Dropout(0.5)(x)  out = tf.keras.layers.Dense(num\_classes, activation='softmax')(x)   model= models.Model(inp, out)  return model   def build\_rnn\_for\_images(input\_shape, num\_classes):  # Treat each row as a timestep sequence of pixels (flatten channels)  timesteps = input\_shape[0]  features = input\_shape[1] \* input\_shape[2]  inp = tf.keras.layers.Input(shape=input\_shape)  x = tf.keras.layers.Reshape((timesteps, features))(inp)  x = tf.keras.layers.GRU(256, return\_sequences=False)(x)  x = tf.keras.layers.Dense(128, activation='relu')(x)  out = tf.keras.layers.Dense(num\_classes, activation='softmax')(x)  model = models.Model(inp, out)  return model |
| --- |

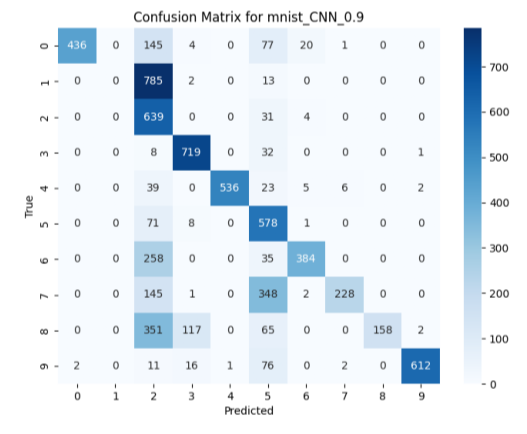
All the model definitions are used for both the Cifar10 and Mnist dataset except build\_vgg\_cifar which is used only for Cifar10 since it requires some layers of the VGG16 trainable to capture the image features.

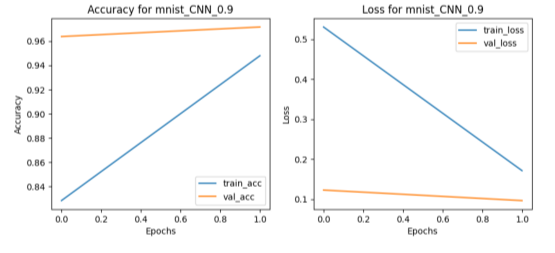
After training the models on Mnist upon different dataset split sizes, below is the results logged.

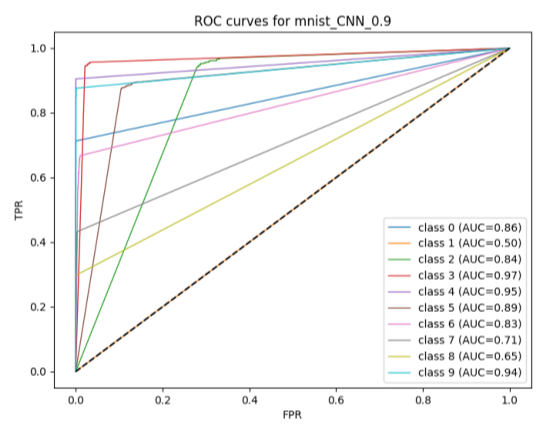


Plotting of the best case results of various models upon different data splits trained on Mnist dataset.

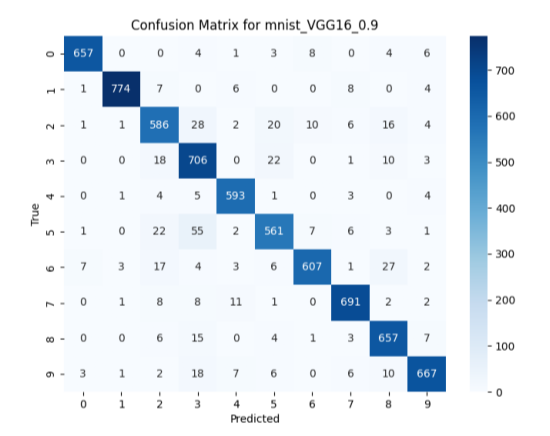
CNNs with split of 0.9:



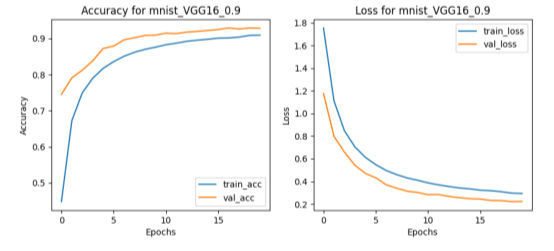


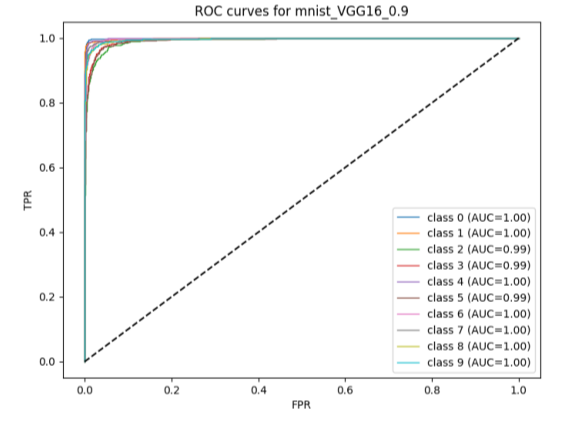


VGG16 with a split of 0.9:

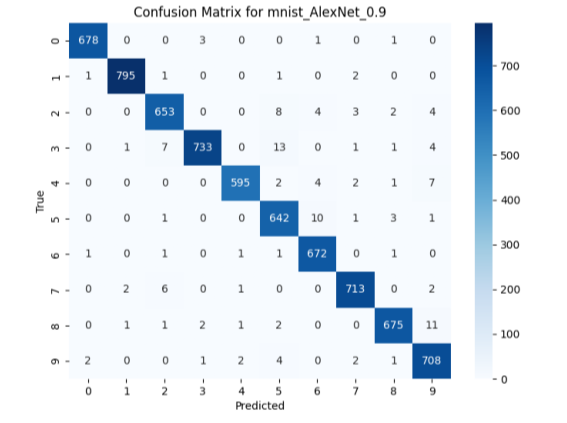


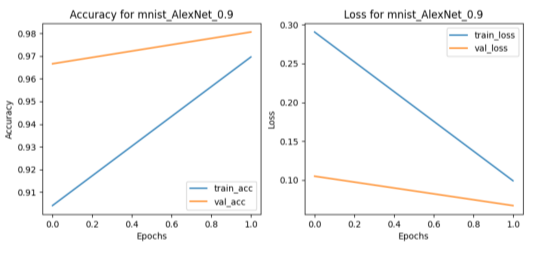
-

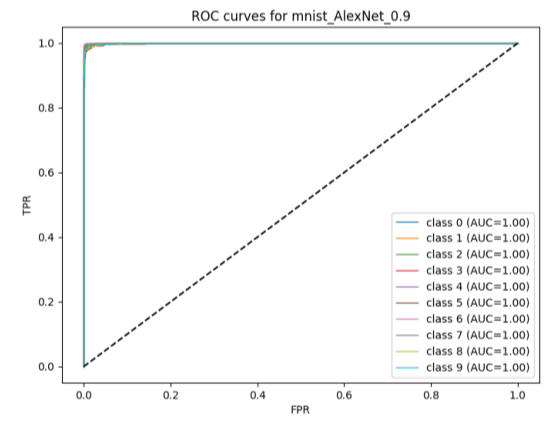




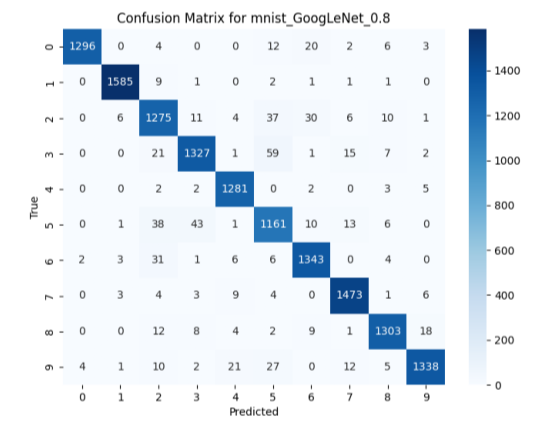
AlexNet with a split of 0.9:

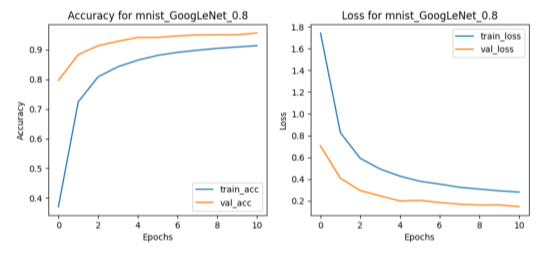


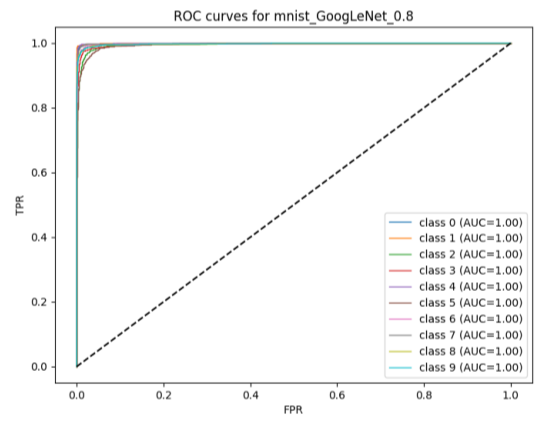




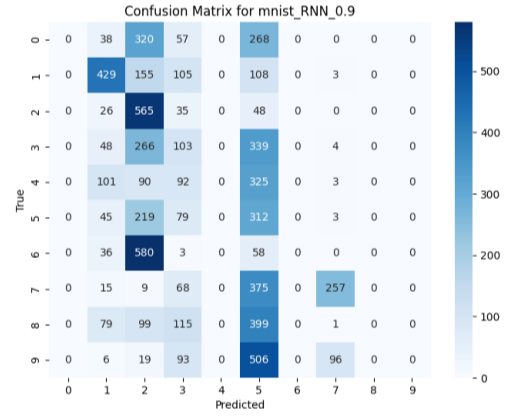
GoogLeNet with split of 0.8:

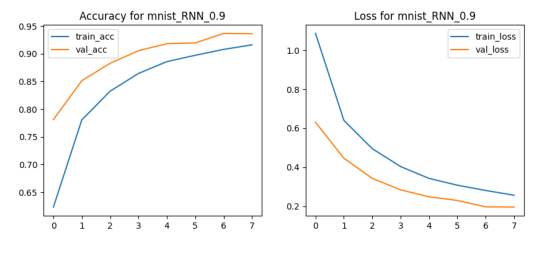


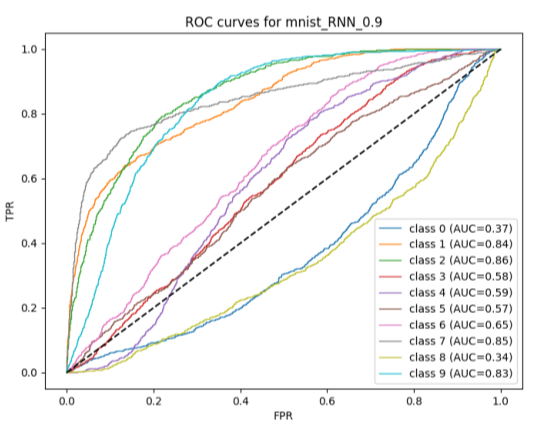




RNN with split of 0.9:







Now, let's move on to training 5 image classification models: CNN, VGG16, GoogLeNet, AlexNet and RNN. The models are trained on the Cifar10 dataset that contain images of real world entities of 10 classes. The training and test dataset obtained from the Keras Api are merged into a single image and image\_label variable. Few image augmentation steps are integrated into the models such as VGG16, AlexNet, GoogLeNet, and models such as CNN and RNN have custom functions that implement the image augmentation step. Image augmentation step is an important step while handling vision related tasks. It introduces variability and prevents models from overfitting. Let’s go through the model training step.

This function will store the image dataset and return the data on function call.

def get\_data(dataset\_name):

if dataset\_name=='cifar10':

(train\_images, train\_labels), (test\_images, test\_labels)= tf.keras.datasets.cifar10.load\_data()

images= np.concatenate([train\_images, test\_images], axis=0)

labels= np.concatenate([train\_labels, test\_labels], axis=0)

print(images.shape, labels.shape)

return images, labels

else:

(train\_images, train\_labels), (test\_images, test\_labels)= tf.keras.datasets.mnist.load\_data()

images= np.concatenate([train\_images, test\_images], axis=0)

labels= np.concatenate([train\_labels, test\_labels], axis=0)

print(images.shape, labels.shape)

return images, labels

Then let’s look into the image augmentation step implemented by CNN and RNNs:

def get\_image\_augmentation():

train\_datagen = ImageDataGenerator(

rotation\_range=20,

shear\_range=0.2,

zoom\_range=0.2,

horizontal\_flip=True,

fill\_mode='nearest',

rescale= 1./255

)

test\_datagen= ImageDataGenerator(rescale= 1./255)

return train\_datagen, test\_datagen

It performs a few tasks such as rescale the image, rotate the images, etc. to introduce variability.

Now lets go through the model definitions:

def simple\_cnn(input\_shape, classes= 10):

model= tf.keras.models.Sequential(

[

tf.keras.layers.Conv2D(64, (3, 3), activation="relu",padding="same", input\_shape= input\_shape),

tf.keras.layers.BatchNormalization(),

tf.keras.layers.MaxPool2D(2, strides=2, padding="valid"),

tf.keras.layers.Conv2D(128, (3, 3), activation="relu", padding="same"),

tf.keras.layers.BatchNormalization(),

tf.keras.layers.MaxPooling2D(2, strides= 2, padding="valid"),

tf.keras.layers.Dropout(0.2),

tf.keras.layers.Conv2D(256, (3, 3), activation= "relu", padding="same"),

tf.keras.layers.BatchNormalization(),

tf.keras.layers.Conv2D(256, (3, 3), activation= "relu", padding="same"),

tf.keras.layers.BatchNormalization(),

tf.keras.layers.MaxPooling2D(2, strides= 2, padding="valid"),

tf.keras.layers.Dropout(0.2),

tf.keras.layers.Conv2D(512, (3, 3), activation= "relu", padding="same"),

tf.keras.layers.BatchNormalization(),

tf.keras.layers.Conv2D(512, (3, 3), activation= "relu", padding="same"),

tf.keras.layers.BatchNormalization(),

tf.keras.layers.MaxPooling2D(2, strides= 2, padding="valid"),

tf.keras.layers.Dropout(0.3),

tf.keras.layers.Flatten(),

tf.keras.layers.Dense(256, activation="relu"),

tf.keras.layers.Dropout(0.3),

tf.keras.layers.Dense(128, activation="relu"),

tf.keras.layers.Dense(classes, activation="softmax")

]

)

return model

def build\_vgg\_cifar(input\_shape, num\_classes, freeze\_base=True):

base = VGG16(weights='imagenet', include\_top=False, input\_shape=input\_shape)

if freeze\_base:

base.trainable = False

for layer in base.layers[-10:]:

layer.trainable=True

model= tf.keras.models.Sequential(

[

tf.keras.layers.RandomFlip("horizontal"),

tf.keras.layers.RandomRotation(0.1),

tf.keras.layers.Resizing(224, 224),

tf.keras.layers.Rescaling(1./255),

#preprocess\_input,

base,

tf.keras.layers.GlobalAveragePooling2D(),

tf.keras.layers.Dense(256, activation='relu'),

tf.keras.layers.Dropout(0.5),

tf.keras.layers.Dense(num\_classes, activation='softmax')

])

return model

# this is for mnist dataset

def build\_vgg16\_transfer(input\_shape, num\_classes, freeze\_base=True):

base = VGG16(weights='imagenet', include\_top=False, input\_shape=input\_shape)

if freeze\_base:

base.trainable = False

model= tf.keras.models.Sequential(

[

tf.keras.layers.RandomFlip("horizontal"),

tf.keras.layers.RandomRotation(0.1),

tf.keras.layers.Resizing(224, 224),

tf.keras.layers.Rescaling(1./255),

#preprocess\_input,

base,

tf.keras.layers.GlobalAveragePooling2D(),

tf.keras.layers.Dense(256, activation='relu'),

tf.keras.layers.Dropout(0.5),

tf.keras.layers.Dense(num\_classes, activation='softmax')

])

return model

def build\_alexnet(input\_shape, num\_classes):

# A Keras-style AlexNet (simplified)

inp= tf.keras.layers.Input(shape=[None, None, 3])

x = tf.keras.layers.RandomFlip("horizontal")(inp)

x = tf.keras.layers.RandomRotation(0.1)(x)

x = tf.keras.layers.Resizing(224, 224)(x)

x = tf.keras.layers.Rescaling(1./255)(x)

x = tf.keras.layers.Conv2D(96, (11,11), strides=4, activation='relu', padding='same')(x)

x = tf.keras.layers.MaxPooling2D((3,3), strides=2)(x)

x = tf.keras.layers.Conv2D(256, (5,5), activation='relu', padding='same')(x)

x = tf.keras.layers.MaxPooling2D((3,3), strides=2)(x)

x = tf.keras.layers.Conv2D(384, (3,3), activation='relu', padding='same')(x)

x = tf.keras.layers.Conv2D(384, (3,3), activation='relu', padding='same')(x)

x = tf.keras.layers.Conv2D(256, (3,3), activation='relu', padding='same')(x)

x = tf.keras.layers.MaxPooling2D((3,3), strides=2)(x)

x = tf.keras.layers.Flatten()(x)

x = tf.keras.layers.Dense(4096, activation='relu')(x)

x = tf.keras.layers.Dropout(0.5)(x)

x = tf.keras.layers.Dense(4096, activation='relu')(x)

x = tf.keras.layers.Dropout(0.5)(x)

out = tf.keras.layers.Dense(num\_classes, activation='softmax')(x)

model = models.Model(inp, out)

return model

def inception\_module(x, filters):

# filters: tuple/list (f1, f3r, f3, f5r, f5, poolproj)

f1, f3r, f3, f5r, f5, poolproj = filters

path1 = tf.keras.layers.Conv2D(f1, (1,1), padding='same', activation='relu')(x)

path2 = tf.keras.layers.Conv2D(f3r, (1,1), padding='same', activation='relu')(x)

path2 = tf.keras.layers.Conv2D(f3, (3,3), padding='same', activation='relu')(path2)

path3 = tf.keras.layers.Conv2D(f5r, (1,1), padding='same', activation='relu')(x)

path3 = tf.keras.layers.Conv2D(f5, (5,5), padding='same', activation='relu')(path3)

path4 = tf.keras.layers.MaxPooling2D((3,3), strides=1, padding='same')(x)

path4 = tf.keras.layers.Conv2D(poolproj, (1,1), padding='same', activation='relu')(path4)

return tf.keras.layers.concatenate([path1, path2, path3, path4], axis=-1)

def build\_googlenet\_like(input\_shape, num\_classes):

inp = tf.keras.layers.Input(shape=[None, None, 3])

x = tf.keras.layers.RandomFlip("horizontal")(inp)

x = tf.keras.layers.RandomRotation(0.1)(x)

x = tf.keras.layers.Resizing(224, 224)(x)

x = tf.keras.layers.Rescaling(1./255)(x)

x = tf.keras.layers.Conv2D(64, (7,7), strides=2, padding='same', activation='relu')(x)

x = tf.keras.layers.MaxPooling2D((3,3), strides=2, padding='same')(x)

x = tf.keras.layers.Conv2D(64, (1,1), activation='relu')(x)

x = tf.keras.layers.Conv2D(192, (3,3), padding='same', activation='relu')(x)

x = tf.keras.layers.MaxPooling2D((3,3), strides=2, padding='same')(x)

# a few inception modules (small)

x = inception\_module(x, (64, 96, 128, 16, 32, 32))

x = inception\_module(x, (128, 128, 192, 32, 96, 64))

x = tf.keras.layers.MaxPooling2D((3,3), strides=2, padding='same')(x)

x = tf.keras.layers.GlobalAveragePooling2D()(x)

x = tf.keras.layers.Dropout(0.5)(x)

out = tf.keras.layers.Dense(num\_classes, activation='softmax')(x)

model= models.Model(inp, out)

return model

def build\_rnn\_for\_images(input\_shape, num\_classes):

# Treat each row as a timestep sequence of pixels (flatten channels)

timesteps = input\_shape[0]

features = input\_shape[1] \* input\_shape[2]

inp = tf.keras.layers.Input(shape=input\_shape)

x = tf.keras.layers.Reshape((timesteps, features))(inp)

x = tf.keras.layers.GRU(256, return\_sequences=False)(x)

x = tf.keras.layers.Dense(128, activation='relu')(x)

out = tf.keras.layers.Dense(num\_classes, activation='softmax')(x)

model = models.Model(inp, out)

return model

Now let's go through the for loop that employs the model training step over the 80% and 90% split of dataset and the 5 different models.

def run\_experiment(dataset\_name):

splits= [0.8, 0.9]

results= []

images, labels= get\_data(dataset\_name)

for split in splits:

train\_images, test\_images, train\_labels, test\_labels= train\_test\_split(images, labels, train\_size= split, random\_state= 42)

for m in model\_type:

plot\_name\_prefix= f'{dataset\_name}\_{m}\_{split}'

if m in ["CNN", 'RNN']:

train\_datagen, test\_datagen= get\_image augmentation()

train\_datagen.fit(train\_images)

test\_datagen.fit(test\_images)

tf.keras.backend.clear\_session()

if dataset\_name=='mnist':

model= model\_type[m]((28, 28, 1), 10)

else:

model= model\_type[m]((32, 32, 3), 10)

model.compile(

optimizer= tf.keras.optimizers.Adam(1e-4),

loss='sparse\_categorical\_crossentropy',

metrics= ['accuracy']

)

print(f'Training starting for {m} with split- {split}')

history= model.fit(

train\_datagen.flow(train\_images, train\_labels),

epochs= 15,

validation\_data= test\_datagen.flow(test\_images, test\_labels),

callbacks= [

ReduceLROnPlateau(monitor='val\_loss', factor=0.5, patience=3, verbose=1),

EarlyStopping(monitor='val\_loss', patience=8, verbose=1),

StopAtAccuracy(0.91)

]

)

plot\_history(history, plot\_name\_prefix)

plot\_confusion(test\_labels, np.argmax(model.predict(test\_images), axis=1), plot\_name\_prefix)

plot\_roc\_auc(test\_labels, model.predict(test\_images), 10, plot\_name\_prefix)

elif m=='VGG16':

if dataset\_name=='mnist':

train\_images\_rgb = np.repeat(train\_images, 3, axis=-1)

test\_images\_rgb = np.repeat(test\_images, 3, axis=-1)

else:

train\_images\_rgb= train\_images

test\_images\_rgb= test\_images

train\_images\_processed= preprocess\_input(train\_images\_rgb)

test\_images\_processed= preprocess\_input(test\_images\_rgb)

tf.keras.backend.clear\_session()

if dataset\_name=='mnist':

model= model\_type[m]((224, 224, 3), 10)

else:

model= build\_vgg\_cifar((224, 224, 3), 10)

model.compile(

optimizer= tf.keras.optimizers.Adam(1e-4),

loss= 'sparse\_categorical\_crossentropy',

metrics= ['accuracy']

)

print(f'Training starting for {m} with split- {split}')

history= model.fit(

train\_images\_processed, train\_labels,

epochs= 20,

validation\_data= (test\_images\_processed, test\_labels),

callbacks= [

ReduceLROnPlateau(monitor='val\_loss', factor=0.5, patience=3, verbose=1),

EarlyStopping(monitor='val\_loss', patience=8, verbose=1),

StopAtAccuracy(0.91)

]

)

plot\_history(history, plot\_name\_prefix)

plot\_confusion(test\_labels, np.argmax(model.predict(test\_images\_processed), axis=1), plot\_name\_prefix)

plot\_roc\_auc(test\_labels, model.predict(test\_images\_processed), 10, plot\_name\_prefix)

else:

if dataset\_name=='mnist':

train\_images\_rgb = np.repeat(train\_images, 3, axis=-1)

test\_images\_rgb = np.repeat(test\_images, 3, axis=-1)

else:

train\_imagess\_rgb= train\_images

test\_images\_rgb= test\_images

tf.keras.backend.clear\_session()

model= model\_type[m]((224, 224, 3), 10)

model.compile(

optimizer= tf.keras.optimizers.Adam(1e-4),

loss= 'sparse\_categorical\_crossentropy',

metrics= ['accuracy']

)

print(f'Training starting for {m} with split- {split}')

history= model.fit(

train\_images\_rgb, train\_labels,

epochs= 20,

validation\_data= (test\_images\_rgb, test\_labels),

callbacks= [

ReduceLROnPlateau(monitor='val\_loss', factor=0.5, patience=3, verbose=1),

EarlyStopping(monitor='val\_loss', patience=8, verbose=1),

StopAtAccuracy(0.91)

])

plot\_history(history, plot\_name\_prefix)

plot\_confusion(test\_labels, np.argmax(model.predict(test\_images\_rgb), axis=1), plot\_name\_prefix)

plot\_roc\_auc(test\_labels, model.predict(test\_images\_rgb), 10, plot\_name\_prefix)

print(f'Logged the training metrics...')

results.append({

"DatasetName": dataset\_name,

"Model": m,

"Split": split,

"Accuracy": history.history['accuracy'][-1],

"Val\_Accuracy": history.history['val\_accuracy'][-1],

})

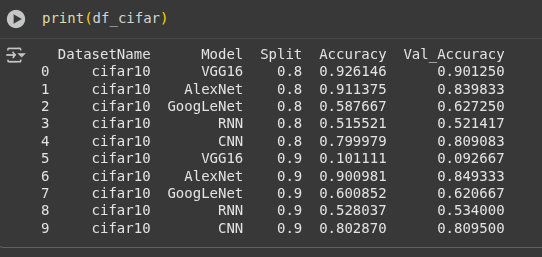
results= pd.DataFrame(results)

results.to\_csv(f'results/{dataset\_name}\_results.csv', index=False)

return results

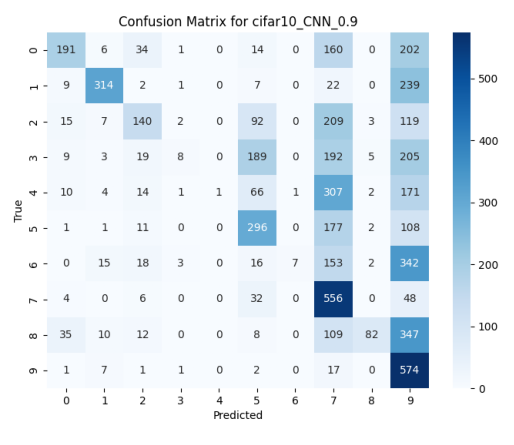
The model training loop goes over to split the dataset on 80% and 90% of the dataset and trains all the models on that data iteratively. Few measures are taken to manage the efficient training and prevent the model from learning from noise. Learning rate scheduler, Earlystopping along with StopAtAccuracy are used here. The learning rate scheduler reduces the learning rate when the validation loss does not decrease to make the model able to absorb the variance. EarlyStopping is used to stop the training when validation loss does not reduce after a few epochs and StopAtAccuracy is employed to stop the training when a target accuracy is reached. This all helps keep check the model training along with reducing the unnecessary waste of compute resources.

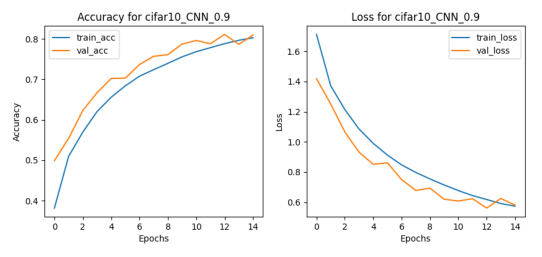
The training metrics thus obtained:

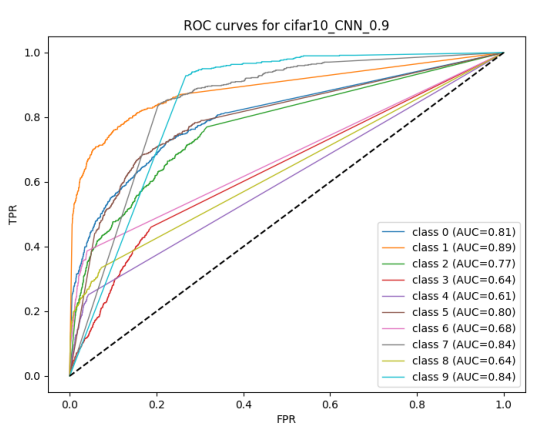


Now let's go through the plots:

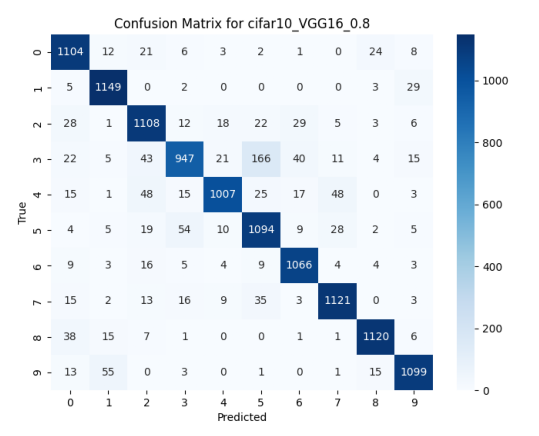
CNN with split of 0.9:

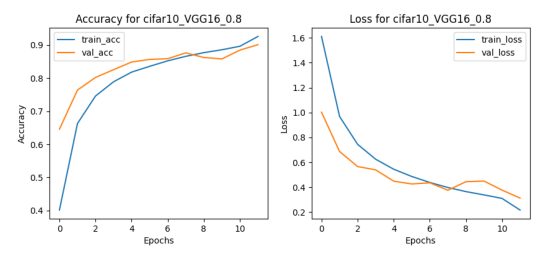


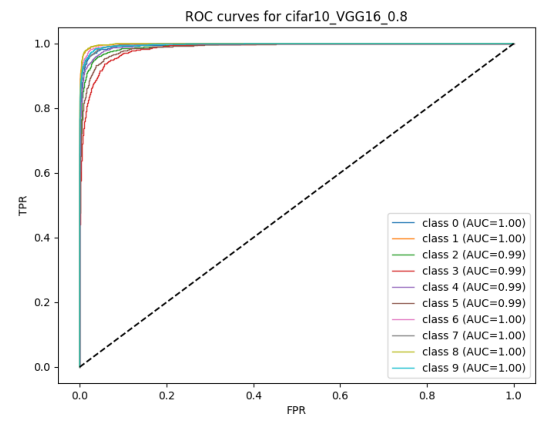




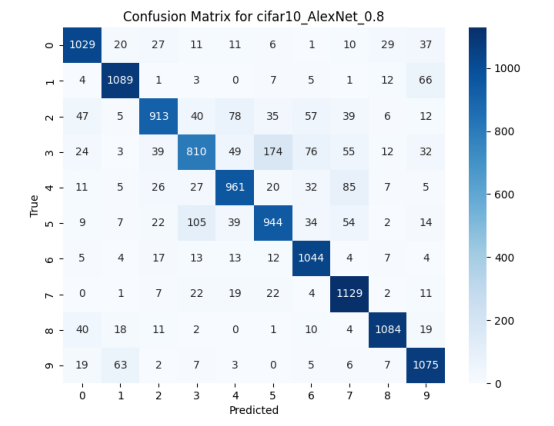
VGG16:

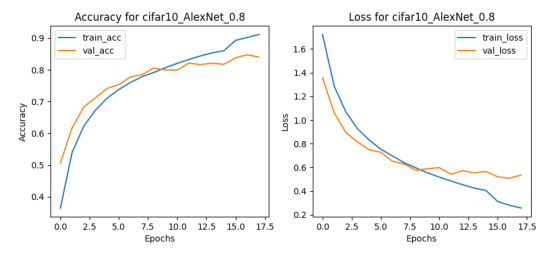


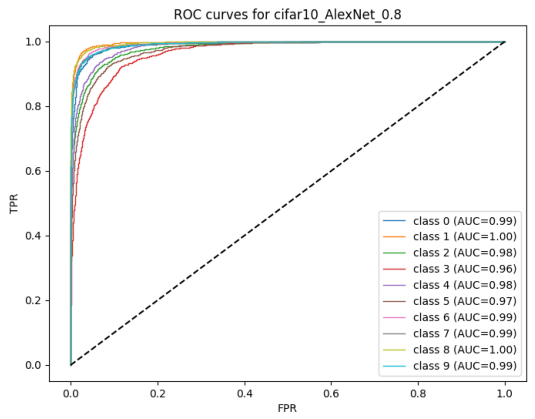




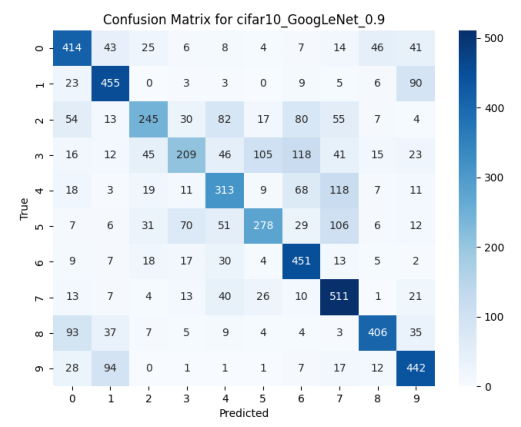
AlexNet:



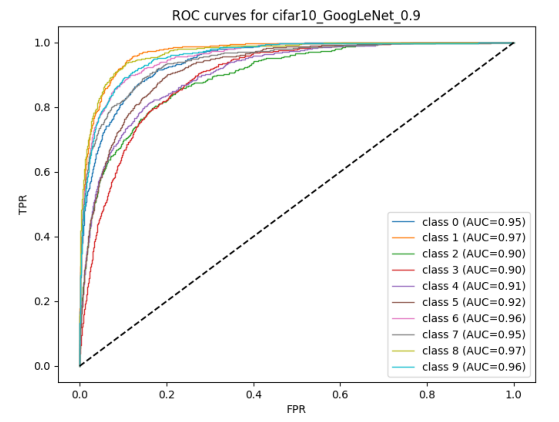




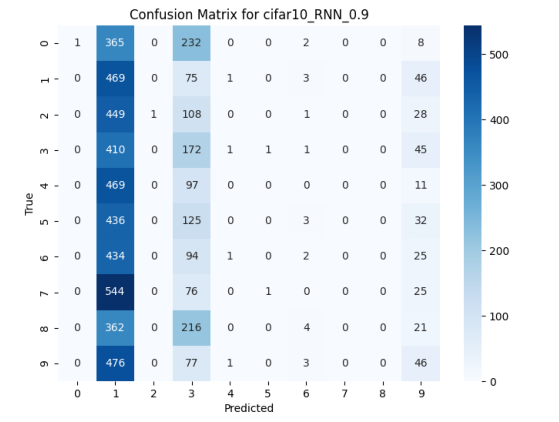
GoogLeNet:

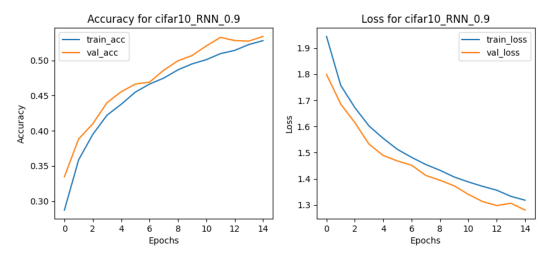


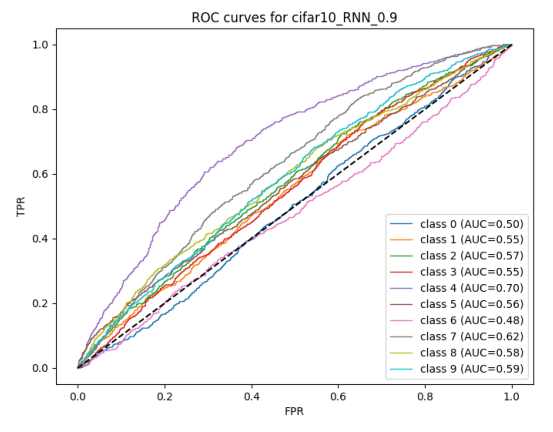




RNN:







Conclusion:

The HMM model classifies data into binary sets and for the Wisconsin Breast Cancer dataset and Ionosphere dataset, the accuracy achieved are as follows:

Pre-parameter tuned metrics:

| Model Name | Dataset | Accuracy Score |
| --- | --- | --- |
| GaussianHMM | Wisconsin Breast Cancer | 90.35 |
| MultinomialHMM | Wisconsin Breast Cancer | 70.18 |
| GaussianHMM | Ionosphere Dataset | 70.42 |
| MultinomialHMM | Ionosphere Datasete | 56.34 |

The models after parameter tuning perform exceptionally well on both the Wisconsin Breast Cancer dataset and Ionosphere dataset.

The post parameter tuning accuracy on model training on the BreastCancer dataset is achieved using the Gaussian model achieving an accuracy of 98.25%. On the other hand, the multinomial model performs better on the Ionosphere dataset and achieves an accuracy of 94.44%.

Now, let’s talk about the deep learning models trained upon the Mnist and Cifar10 datasets.

The models trained upon the Cifar10 dataset gave the best accuracy metrics:

| Model Name | Dataset split | Accuracy |
| --- | --- | --- |
| CNN | 90% | 80% |
| RNN | 90% | 52 |
| VGG16 | 80% | 92% |
| AlexNet | 90% | 91% |
| GoogLeNet | 90% | 60% |

We can conclude that RNN and GoogLeNet couldn't capture the image features necessary for it to classify the images. CNN and VGG16 again proved their capability in capturing the image features and were able to classify the images to respective classes.

The models trained upon the Mnist dataset gave the best accuracy metrics:

| Model Name | Dataset Split | Accuracy |
| --- | --- | --- |
| CNN | 90% | 94% |
| RNN | 90% | 91% |
| VGG16 | 90% | 96% |
| AlexNet | 90% | 96% |
| GoogLeNet | 80% | 91% |

This shows that each of the models performed extremely well, each achieving >=90% accuracy with a split of 0.9 except GoogLeNet which achieved it with a split of 0.8.