Supervised and Unsupervised Training on CIFAR-10 dataset

Introduction

Image classification in computer vision refers supervised learning algorithm that takes taking an input image and outputs class. Advancement in deep learning techniques and hardware technology have made image classification task perform accurately at high speed. Autoencoder is an unsupervised artificial neural network that learns how to efficiently generate an encoded compressed version of data and then learns how to reconstruct to its original input a back from its encoded representation. In this report, a semi-supervised network is designed that combines supervised and unsupervised architectures in one model. The coded features of autoencoder set as input to the image classifier network to predict different classes of images in the CIFAR-10 dataset as shown in figure 1. The CIFAR-10 dataset, which is a labeled subset of the 80 million tiny images dataset is used for classification task. The dataset consists of 60000 32x32 color images in 10 mutually exclusive classes, with 6000 images per class. There are 50000 training images and 10000 test images. Some random images are shown in figure 1. However, in this report we modify the dataset slightly by using 50% of bird, deer and truck classes for training. Two types of training techniques have been used: **a.** separately training a deep-autoencoder and feeding the coded features to classifier network and **b.** end-to-end network where deep-autoencoder and classifier network are trained simultaneously. Also, we compare the accuracy of both training types with standard simple convolutional neural network (CNN). This report does not focus on image reconstruction using autoencoder networks rather the focus is on solely using autoencoders for feature extraction.

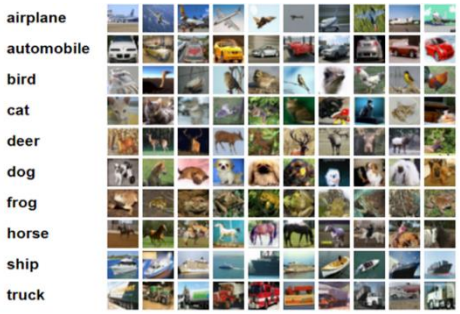

Figure 1: Random Images of CIFAR-10 dataset

Hypothesis

An autoencoder combines an encoder from the original space to a latent space – generating feature codes, and a decoder to map back to the identity of data. During training such semi-supervised network, encoder and decoder layers are stacked with hidden feature coded-layers, and it is trained in a supervised way by having every sample input image having respective class. Once this nonlinear feature extraction technique is trained, the feature space contains all the information required to represent input space and removes redundant information from the data thus performing dimensionality reduction. The efficiency of encoding is dependent on how much useful the extracted features are and different autoencoder uses different techniques to improve such efficiency. There are two ways to go about this method. One way is that once the efficient encoding map through auto-encoding is obtained , all the samples can be transformed to its feature space, and input of these encoded features are fed classifier without bothering encoding or decoding maps anymore **(separate-training model)**. This method uses multiple loss functions – one for autoencoder network and one for image classification. Another method to verify the hypothesis is that decoding layers can be remove and only consider encoding map as the first transformation in end-to-end transformation **(end-to-end model)** with one loss function

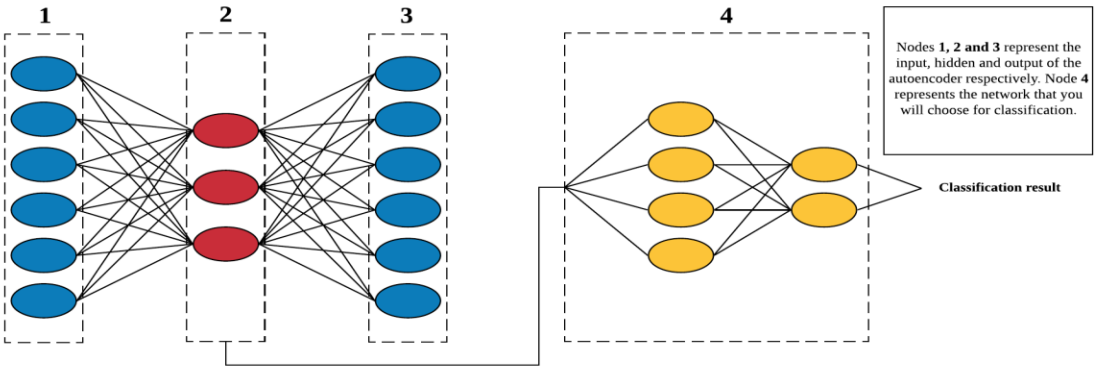

Figure 2: Overall architecture of the hypothesis (as given by Ridge-i)

System Configuration

Keras, a python based deep learning library was used to achieve this task. Also, other libraries such as scikit-learn, numpy, were also for computations. PC with Intel Core i7-960 CPU operating at 3.20~GHz clock speed, 16GB RAM with Linux operating system (ubuntu) used for interfacing Titan-V GPUs in 16x PCIe slots on motherboard.

Creating an Imbalance Dataset and Data Preprocessing

CIFAR-10 dataset has 5000 images/class for training and 1000 images/class for testing 10 classes. An imbalance in dataset was created by transferring 2000 images of bird, truck and deer from training dataset to testing dataset. By developing a user defined function **data_modify()** function, the normalized imbalance in dataset was created. The snapshot of the code is shown in figure 3. Further, 20% of training data is also used for validation set using train_test_split() function from scikit-learn library. The classes were converted to one-hot encoded data using the utils.to_categorical function of the keras library.

Factors while designing the model

Number of factors were considered while designing this hypothesis. These are listed below

1. Though there are number of autoencoder layers that could be designed, in this experiment only deep CNN based autoencoder were only considered.
2. To compare results of end-to-end model, separate-training model and standard CNN based classifier, the classifier network has been maintained same for all methods.
3. For the training the networks, the epochs have been set to 100. Beyond 100 epochs model accuracy and loss did not have any significant changes.
4. The batch size is set to 512. Dropout function regularizer has been used to reduce overfitting. Stochastic Gradient Descent is considered for optimizing the objective function of the network.

End to End Model

In this model, auto-encoder and image classifier are not trained separately. Rather, the encoded feature of each input sample is passed to the image classifier network directly (removing the decoded layer) and train the complete semi-supervised network with **just one loss function** for classification. Thus, each less-dimensional coded features for each image is fed to the network every time and trained to generate the feature space. One key advantage of this method is the total learnable training parameters are very small and prediction accuracy of the test set is higher than separately training model network. The code snippet of end-to-end model is given in code snippet section in **CS1** as function **end_to_end()** . As seen in the **CS1** the output of encoder is passed to the input of the image classifer CNN. Snippet **CS2** and **CS3** shows the code for encoder and decoder network of end-to-end module. The total trainable and non-trainable parameters are 137866 and 576 respectively. The encoder-decoder model is a deep convolutional autoencoder model. The encoder model had 4-convolutional blocks of filter size 32, 32, 64 and 64 of kernel size 3x3. Batch normalization layer followed after every convolutional layer and after every two convolutional layer, a max-pool layer was added. The classifier model was initially set like VGG-16 architecture. However, having so many layers did not have any effect on the accuracy and other parameters (precision, recall, and f1-score). Thus, a smaller architecture was designed for this application. Dropout layers are added after every convolutional layer (followed by Batch Normalization and Maxpooling) as regularizer for reducing overfitting. In end-to-end model, batches of augmented image data is generated. The parameters for image augmentation are as rotation, zoom, width shift, height shift, shear and flip. From the confusion matrix (by using **confusion_matrix()** function from **sklearn** library) shown in figure 3, the model works quite good with 100 epochs for all the classes despite imbalanced data. Moreover, bird, deer and truck have high precision than other classes.

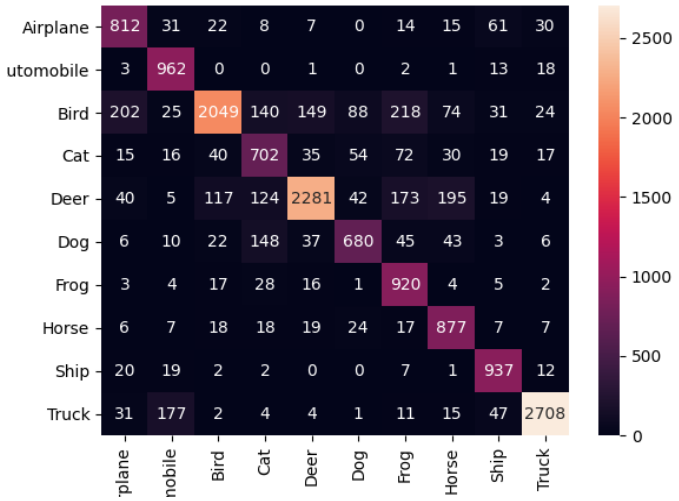

Figure 3: Confusion matrix for end-to-end model based image classification

As seen in the figure 4, for the epoch set upto 100, the model accuracy is set around 80% for the end-to-end network for both training and validation (test) set , while the model loss is reduced up to approximately 0.5. A log-loss (categorical_crossentropy) function is used to calculate loss and accuracy measure for multi-class classification.

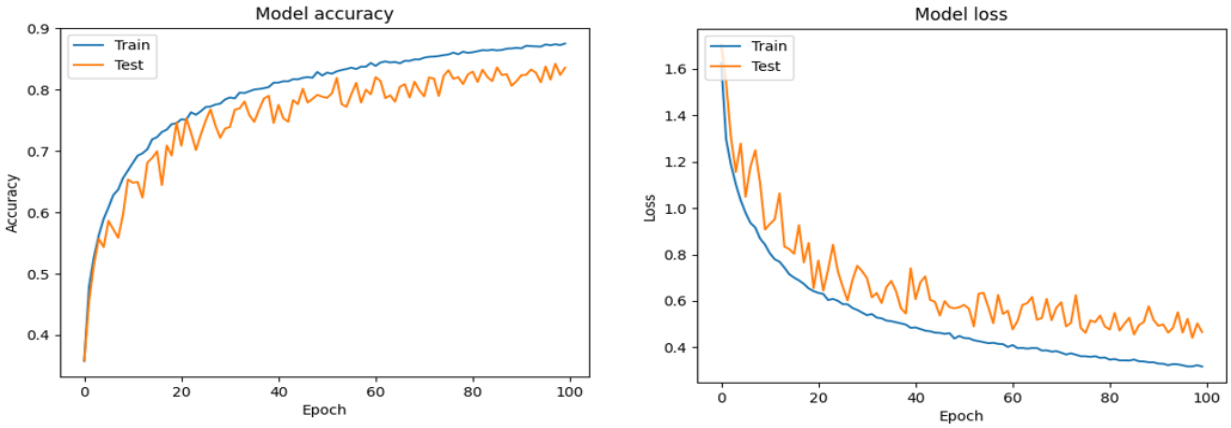

Figure 4: Model accuracy and loss for end-to-end model

Figure 5 shows the sample actual and predicted images with its probability values for end-to-end network. The code snippet for displaying images and predicted class is shown in **CS8**

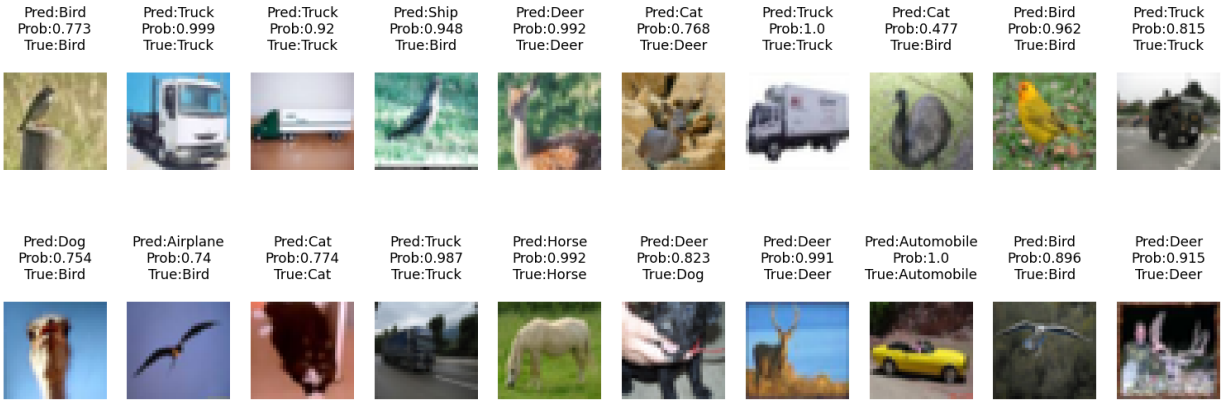

Figure 5: Actual Image and Predicted Class from end-to-end model features

Table1 is the classification report for CIFAR-10 dataset for end-to-end model based prediction. **classification_report()** from **sklearn** library is used for generating the report.

	precision	recall	f1-score	support
Airplane	0.71	0.81	0.76	1000
Automobile	0.77	0.96	0.85	1000
Bird	0.9	0.68	0.77	3000
Cat	0.6	0.7	0.65	1000
Deer	0.89	0.76	0.82	3000
Dog	0.76	0.68	0.72	1000
Frog	0.63	0.92	0.74	1000
Horse	0.7	0.88	0.78	1000
Ship	0.82	0.94	0.87	1000
Truck	0.96	0.9	0.93	3000
accuracy			0.81	16000
macro avg	0.77	0.82	0.79	16000
weighted avg	0.83	0.8	0.81	16000

Table 1: Classification Report generated by prediction from end-to-end model.

Separate-Training Model

In this model, autoencoder and image classifier networks are trained separately. Thus, two different loss functions are required. Mean Square Error loss function is used for calculating the loss from auto encoder model and log-loss function is used for multi-class classification. No data augmentation technique could be used for classifier network as the input to it was the encoded data from autoencoder network. The number of layers and parameters for the autoencoder model is same as used for end-to-end model. As seen in code snippet **CS5**, autoencoder() function, the encoder has 4-convolutional blocks of filter size 32,32,64 and 64 of kernel size 3x3. Batch normalization layer followed after every convolutional layer and after every two convolutional layer, a max-pool layer was added. Similarly, in the decoder model along with up-sampling 4-blocks of filter size 64,64 32 and 32 of kernel size 3x3 is added. Finally, a (3x1) layer is added along with sigmoid activation function to reconstruct the input. The classifier network is same as end-to-end model as shown in snipped **CS6**. The total learnable parameters are also significantly larger than end-to-end model as decoded layer of autoencoder block cannot be ignored. For autoencoder network, the total trainable and non-trainable parameters are 168131 and 896 respectively. For image classifier network, the trainable and non-trainable parameters are 71338 and 192 respectively. Thus, despite using similar architecture as end-to-end model, the use of decoder layers in autoencoder network increases the number of parameters. From the confusion matrix shown in figure 6, the model works quite good for all the classes despite imbalanced data. However, compared to end-to-end model, the precision is less for all classes. One key advantage over the end-to-end model is that once the autoencoder network is trained separately, this network could be used for transfer learning. However, for the current dataset and the network parameters, the measurement factors like accuracy, precision is lower than end-to-end model.

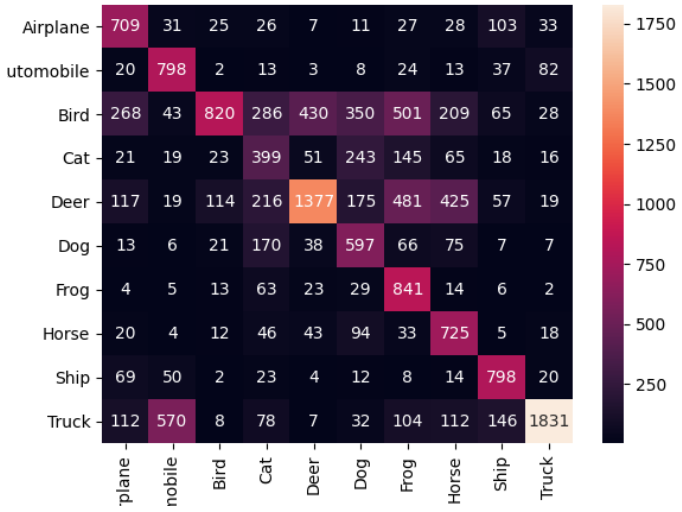

Figure 6: Confusion matrix for separate-training model based image classification

As seen in the figure 7, for the epoch set upto 100, the model accuracy is set around 65% for the separate-training network for both training and validation (test) set, while the final model loss is reduced up to approximately 1.0.

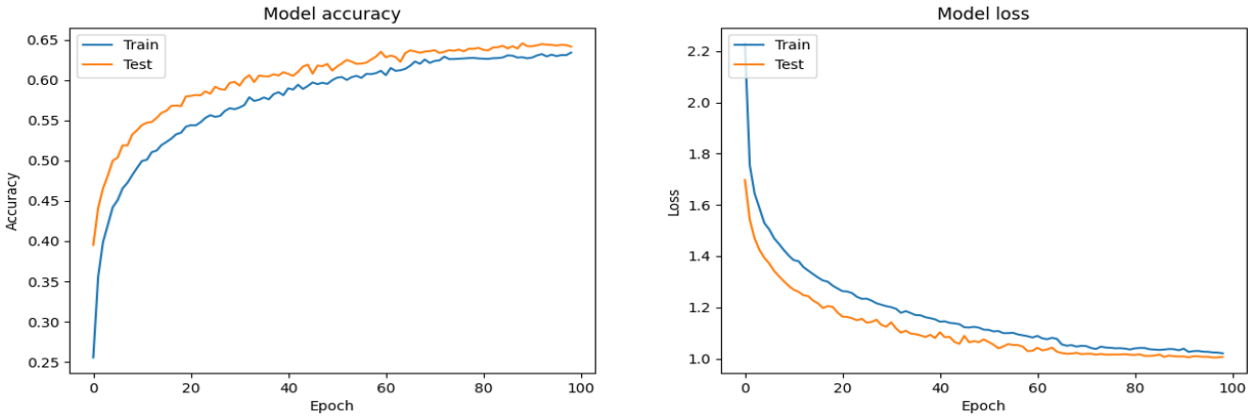

Figure 7: Model accuracy and loss for separate-training model

Figure 8 shows the sample actual and predicted images with its probability values for separate-training network.

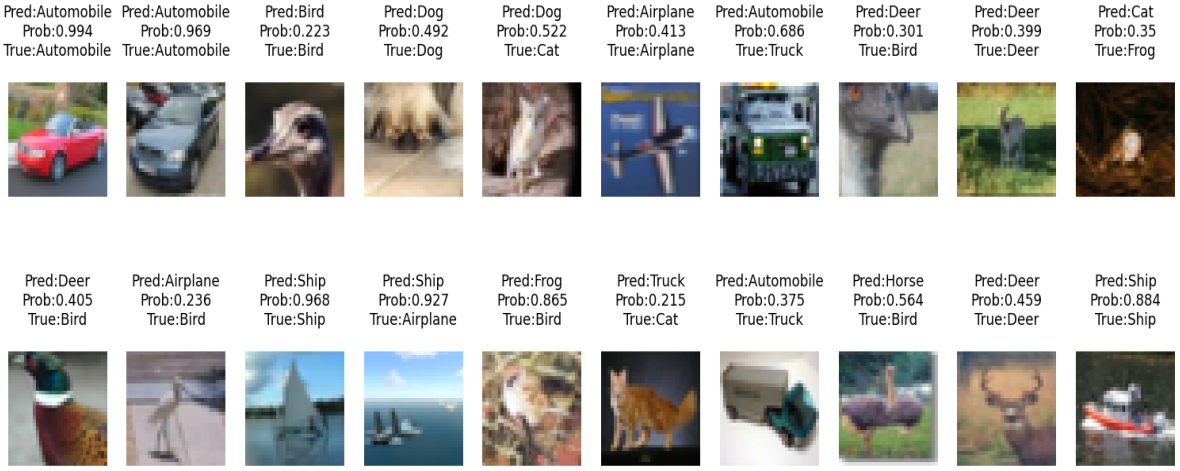

Figure 8: Model accuracy and loss for separate-training model

Table1 is the classification report for CIFAR-10 dataset for end-to-end model based prediction. **classification_report()** from **sklearn** library is used for generating the report.

	precision	recall	f1-score	support
Airplane	0.51	0.74	0.6	1000
Automobile	0.55	0.8	0.65	1000
Bird	0.74	0.42	0.54	3000
Cat	0.34	0.49	0.4	1000
Deer	0.74	0.51	0.61	3000
Dog	0.41	0.62	0.5	1000
Frog	0.5	0.79	0.61	1000
Horse	0.51	0.74	0.61	1000
Ship	0.65	0.78	0.71	1000
Truck	0.89	0.63	0.74	3000
accuracy			0.6	16000
macro avg	0.58	0.65	0.61	16000
weighted avg	0.66	0.59	0.6	16000

Table 2: Classification Report generated by prediction from separate-training model.

Simple CNN Model

A simple CNN model for image classification is developed to compare with end-to-end and separate-training model. In this model, autoencoder networks are not used. Rather, the images are fed directly to the classifier model. This is done to verify if coded representation to the classifier model gives better result when compared to image input to the network. Just like the classifier network of other models, the architecture has 2-convolutional blocks of filter size 32 and 64 with kernel size 3x3 as shown in code snippet **CS7**. Batch normalization and max-pooling and dropout layer are followed after every convolutional layer. The total trainable and non-trainable parameters are 545,290 and 192 respectively which is clearly higher than both end-to-end and separate-training model. From the confusion matrix shown in figure 9, the model works quite good for all the classes despite imbalanced data. However, compared to end-to-end model, the results are not inspiring. As seen in the figure 10, for the epoch set up to 100, the model accuracy is set around 65% for the separate-training network for both training and validation (test) set, while the final model loss is reduced up to approximately 1.2. However, the accuracy and loss have high variance at almost every epoch in the end.

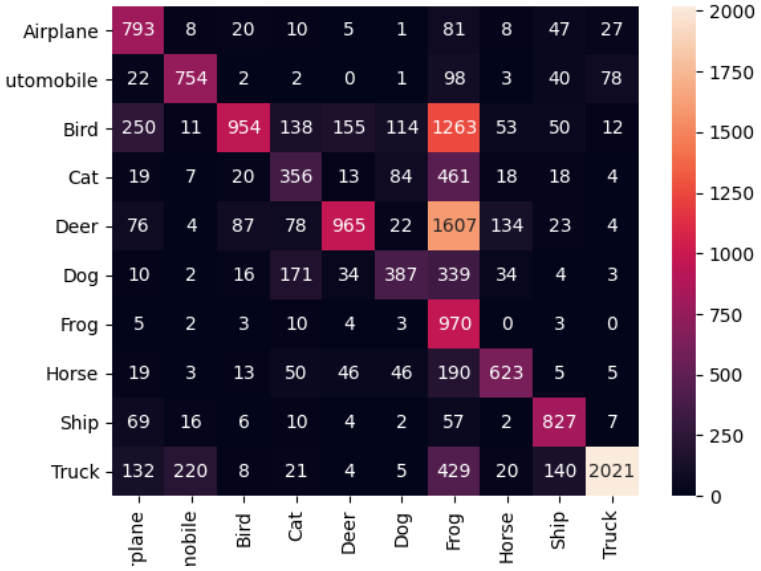

Figure 9: Confusion matrix of simple-CNN model on test data

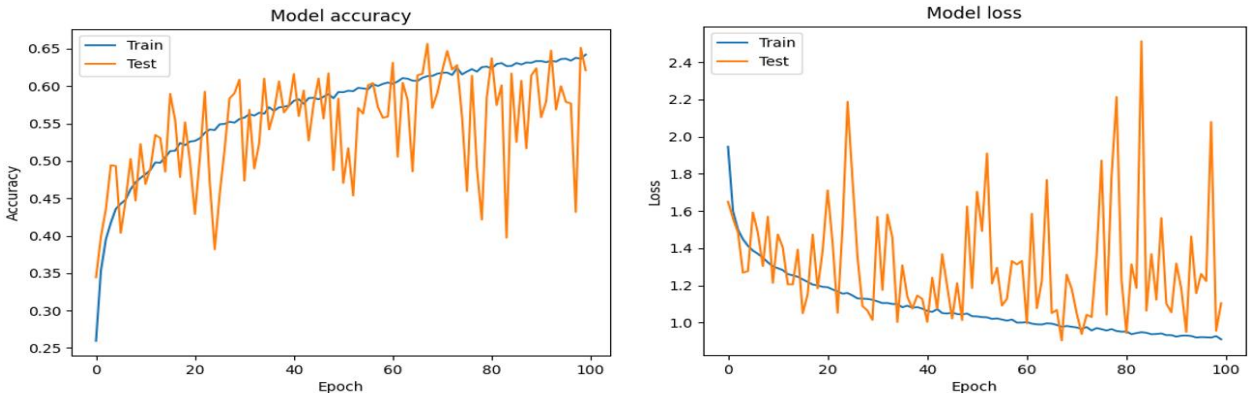

Figure 10: Model accuracy and loss of simple-CNN network.

Figure 11 shows sample actual and predicted images with its probability values for simple-CNN model. Table 2 shows classification report of prediction of this model.

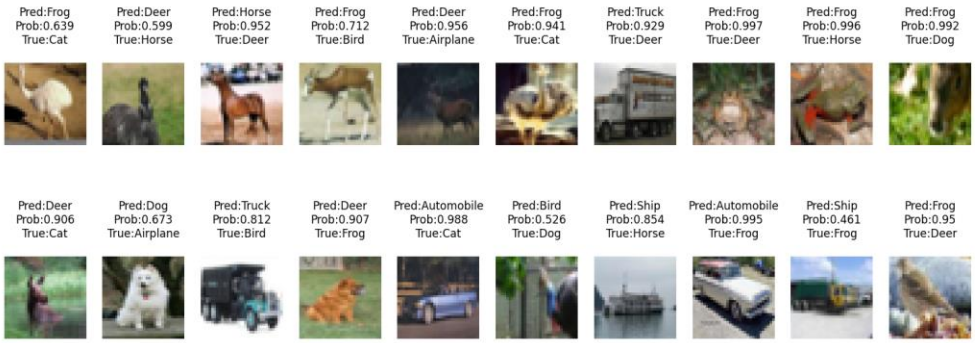

Figure 11: Sample Images of actual data with prediction class with its probabilities for simple-CNN model

	precision	recall	f1-score	support
Airplane	0.57	0.79	0.66	1000
Automobile	0.73	0.75	0.74	1000
Bird	0.84	0.32	0.46	3000
Cat	0.42	0.36	0.39	1000
Deer	0.78	0.32	0.46	3000
Dog	0.58	0.39	0.46	1000
Frog	0.18	0.97	0.3	1000
Horse	0.7	0.62	0.66	1000
Ship	0.71	0.83	0.77	1000
Truck	0.94	0.67	0.78	3000
accuracy			0.54	16000
macro avg	0.65	0.6	0.57	16000
weighted avg	0.72	0.54	0.56	16000

Table 3: Classification Report generated by prediction from simple-CNN model.

Conclusions and Future Work

As the hypothesis suggested, the end-to-end model has higher accuracy than separate-training and classic convolutional models. It can be seen the scatter plots in the figure 11 that of all the developed models that end-to-end encoder has comparatively thinner classification space with less overlap between classes. Moreover, end-to-end model require lesser parameters to generate the desired result of greater accuracy, precision. It must be considered for the CIFAR-10 dataset, that classifier network in both end-to-end and separate-training models are kept small and not deep. Keeping a deeper classifier network with max-pooling erases the network as the non-redundant encoded data is of lesser dimensions (8x8x64). Also, it is found from the results of all three models that classes that had more test-dataset (bird, deer and truck) generated better precision compared to other classes. In future, other encoder-decoder architectures like U-Net can be considered. Also, a hybrid model that can encode and extract features simultaneously could also be developed.

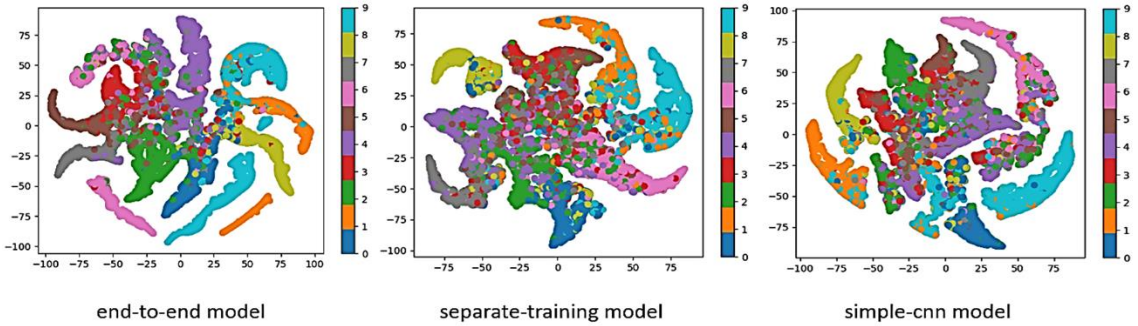

Figure 12 : Scatter plot of different models.

	Parameters	mAP	recall	f1-score
End-to-End Model	138442	0.77	0.82	0.79
Separate-Training Model	169027	0.58	0.65	0.61
Simple CNN. Model	545482	0.65	0.6	0.57

Table 4: Comparison of different models. End-to-end model has lesser parameters with better mAP scores.

Code Snippets

```
def data_modify(xtrain,ytrain,xtest,ytest):

    dict = {0: 'Airplane', 1:'Automobile', 2:'Bird', 3:'Cat', 4:'Deer', 5:'Dog', 6:'Frog', 7:'Horse', 8:'Ship', 9:'Truck'}

    testXFinal = []
    testYFinal = []
    trainXFinal = []
    trainYFinal = []
    countLabel = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    for i, j in zip(xtrain, ytrain):
        if ((j==2)or(j==4)or(j==9)):
            if(countLabel[int(j)]<2000):
                testXFinal.append(i)
                testY.append(j)
                countLabel[int(j)]+=1
            else:
                trainXFinal.append(i)
                trainYFinal.append(j)
        else:
            trainXFinal.append(i)
            trainYFinal.append(j)

    testXFinal  = np.array(testXFinal)
    testYFinal  = np.array(testYFinal)

    trainXFinal = np.array(trainXFinal)
    trainYFinal = np.array(trainYFinal)
    testXFinal = np.append(testXFinal, xtest, axis=0)
    testYFinal = np.append(testY, ytest, axis=0)



    trainXFinal = trainXFinal.astype('float32')
    testXFinal = testXFinal.astype('float32')
    trainXFinal = trainXFinal / 255
    testXFinal = testXFinal / 255
    return(trainXFinal,trainYFinal,testXFinal,testYFinal)
```

CS1: Code for generating imbalance dataset

```
def encoder(inputImage):
    c1 = Conv2D(32, (3, 3), activation='relu', padding='same')(inputImage)
    c1 = BatchNormalization()(c1)
    c1 = Conv2D(32, (3, 3), activation='relu', padding='same')(c1)
    c1 = BatchNormalization()(c1)
    p1 = MaxPooling2D(pool_size=(2, 2))(c1) #16 x 16 x 32
    c2 = Conv2D(64, (3, 3), activation='relu', padding='same')(p1)
    c2 = BatchNormalization()(c2)
    c2 = Conv2D(64, (3, 3), activation='relu', padding='same')(c2)
    c2 = BatchNormalization()(c2)
    p2 = MaxPooling2D(pool_size=(2, 2))(c2) #8 x 8 x 64
    encodedData=p2
    return(encodedData)
```

CS2: Code for generating encoder network for end-to-end model

```
def decoder(encodedData):
    c5 = Conv2D(128, (3, 3), activation='relu', padding='same')(encodedData)
    c5 = BatchNormalization()(c5)
    c6 = Conv2D(64, (3, 3), activation='relu', padding='same')(c5)
    c6 = BatchNormalization()(c6)
    up1 = UpSampling2D((2,2))(c6)
    c7 = Conv2D(32, (3, 3), activation='relu', padding='same')(up1)
    c7 = BatchNormalization()(c7)
    up2 = UpSampling2D((2,2))(c7)
    decoded = Conv2D(3, (3, 3), activation='sigmoid', padding='same')(up2)
    return(decoded)
```

CS3: Code for generating decoder network for end-to-end model

```
def end_to_end():
    input = Input((32,32,3))

    encoderOutput=encoder(input)
    decoderOutput=decoder(encoderOutput)

    conv1 = Conv2D(32, (3, 3), activation='relu', padding='same')(encoderOutput)
    conv1 = BatchNormalization()(conv1)
    #conv1 = Conv2D(32, (3, 3), activation='relu', padding='same')(conv1)
    #conv1 = BatchNormalization()(conv1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1) #4 x 4 x 32
    pool1 = Dropout(0.5)(pool1)

    conv2 = Conv2D(64, (3, 3), activation='relu', padding='same')(pool1)
    conv2 = BatchNormalization()(conv2)
    #conv2 = Conv2D(64, (3, 3), activation='relu', padding='same')(conv2)
    #conv2 = BatchNormalization()(conv2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
    pool2 = Dropout(0.5)(pool2)
    #conv3 = Conv2D(128, (3, 3), activation='relu', padding='same')(pool2)
    #conv3 = BatchNormalization()(conv3)
    #conv3 = Conv2D(128, (3, 3), activation='relu', padding='same')(conv3)
    #conv3 = BatchNormalization()(conv3)
    #conv3 = Conv2D(256, (3, 3), activation='relu', padding='same')(conv3)
    #conv3 = BatchNormalization()(conv3)
    #pool3 = MaxPooling2D(pool_size=(2, 2))(conv3) #1 x 1 x 256


    #conv4 = Conv2D(512, (3, 3), activation='relu', padding='same')(pool3)
    #conv4 = BatchNormalization()(conv2)
    flat = Flatten()(pool2)
    den = Dense(128, activation='relu')(flat)
    classify = Dense(10, activation='softmax',name='classification')(den)
    outputs = [classify]

    return Model(input,outputs)
```

CS4: Code for end-to-end model with encoder, decoder and image classification network

```
def autoencoder():
    input = Input((32,32,3))


    encoderLayer = Conv2D(32, (3, 3), activation='relu', padding='same')(input)
    encoderLayer = BatchNormalization()(encoderLayer)
    encoderLayer = Conv2D(32, (3, 3), activation='relu', padding='same')(encoderLayer)
    encoderLayer = BatchNormalization()(encoderLayer)

    encoderLayer = MaxPool2D(2)(encoderLayer)
    encoderLayer = Conv2D(64, (3, 3), activation='relu', padding='same')(encoderLayer)
    encoderLayer = BatchNormalization()(encoderLayer)
    encoderLayer = Conv2D(64, (3, 3), activation='relu', padding='same')(encoderLayer)
    encoderLayer = BatchNormalization()(encoderLayer)

    encoderLayer = MaxPool2D(2)(encoderLayer)
    code = BatchNormalization()(encoderLayer)

    decoderLayer = UpSampling2D((2,2))(code)
    decoderLayer = Conv2D(64, (3, 3), activation='relu', padding='same')(decoderLayer)
    decoderLayer = BatchNormalization()(decoderLayer)
    decoderLayer = Conv2D(64, (3, 3), activation='relu', padding='same')(decoderLayer)
    decoderLayer = BatchNormalization()(decoderLayer)
    decoderLayer = UpSampling2D((2,2))(decoderLayer)
    decoderLayer = Conv2D(32, (3, 3), activation='relu', padding='same')(decoderLayer)
    decoderLayer = BatchNormalization()(decoderLayer)
    decoderLayer = Conv2D(32, (3, 3), activation='relu', padding='same')(decoderLayer)
    decoderLayer = BatchNormalization()(decoderLayer)
    decoderLayer = Conv2D(3, 1)(decoderLayer)
    decoderLayer = Activation("sigmoid")(decoderLayer)
    return Model(input,code), Model(input,decoderLayer)
```

CS5: Code of autoencoder network for separate-training model

```
def classifier_conv(inp):
    input = Input((inp.shape[1], inp.shape[2], inp.shape[3]))
    conv1 = Conv2D(32, (3, 3), activation='relu', padding='same')(input)
    conv1 = BatchNormalization()(conv1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
    pool1 = Dropout(0.5)(pool1)

    conv2 = Conv2D(64, (3, 3), activation='relu', padding='same')(pool1)
    conv2 = BatchNormalization()(conv2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
    pool2 = Dropout(0.5)(pool2)
    flat = Flatten()(pool2)
    den = Dense(128, activation='relu')(flat)
    classify = Dense(10, activation='softmax',name='classification')(den)
    outputs = [classify]

    return Model(input,outputs)
```

CS6: Code for generating classifier network for separate-training model

```
def classic_conv():

    input = Input((32,32,3))
    conv1 = Conv2D(32, (3, 3), activation='relu', padding='same')(input)
    conv1 = BatchNormalization()(conv1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
    pool1 = Dropout(0.5)(pool1)

    conv2 = Conv2D(64, (3, 3), activation='relu', padding='same')(pool1)
    conv2 = BatchNormalization()(conv2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
    pool2 = Dropout(0.5)(pool2)
    flat = Flatten()(pool2)
    den = Dense(128, activation='relu')(flat)
    classify = Dense(10, activation='softmax',name='classification')(den)
    outputs = [classify]

    return Model(input,outputs)
```

CS7: Code of convolutional network for simple convolutional model

```
def show_test_end_to_end(m, testData, classY):
    cols = 10
    rows = 2
    fig4 = plt.figure(figsize=(2 * cols - 1, 4 * rows - 1))
    print(m)
    for i in range(cols):
        for j in range(rows):
            random_index = np.random.randint(0, len(classY))
            ax = fig4.add_subplot(rows, cols, i * rows + j + 1)
            ax.grid('off')
            ax.axis('off')
            testImage = np.expand_dims(testData[random_index], axis=0)
            testResult = m.predict(testImage)

            plt.imshow(testData[random_index])
            index = np.argsort(testResult[0,:])
            ax.set_title("Pred:{}\nProb:{:.3}\nTrue:{}\n".format(dict[index[9]],testResult[0,index[9]], dict[classY[random_index][0]]))
    fig4.show()
    fig4.savefig('sample_output_end_to_end.png')
```

CS8: Code of convolutional network for display image and predicted class