Jayathilaka H.A.D.T.T.

E/16/156

CO542

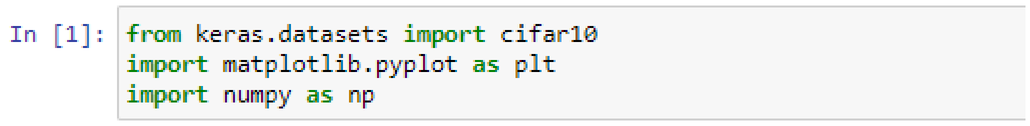
Neural Networks and Fuzzy Systems

2021

Lab 06 - CNN

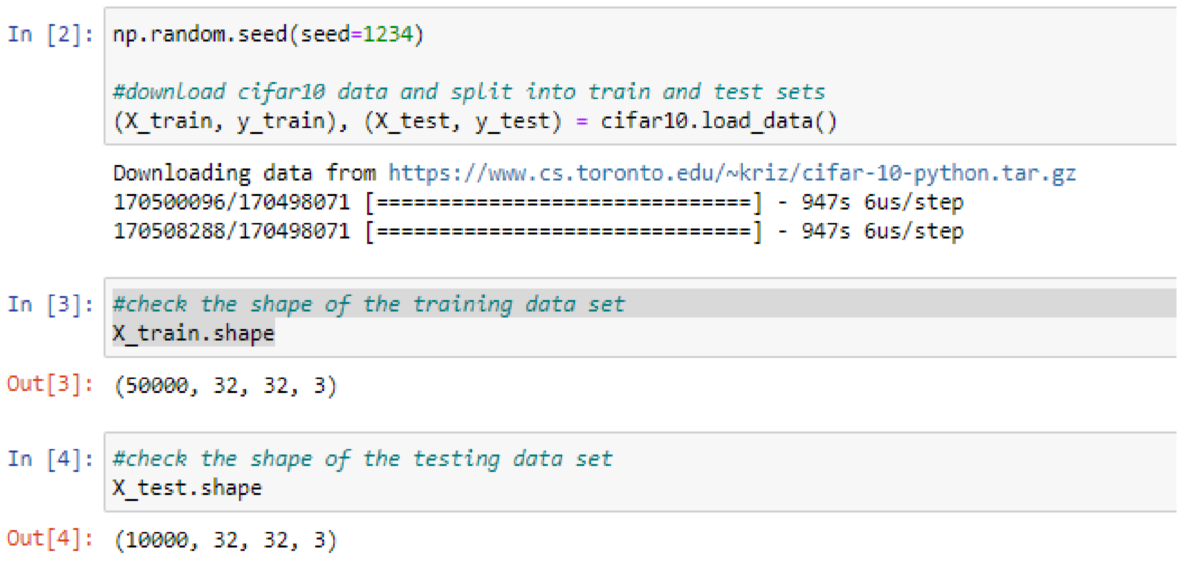
The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck) with 6000 images per class. There are 50000 training images and 10000 test images. Your task here is to build and train a CNN to detect images in each of these classes using Keras framework and other required libraries accordingly.

1. Import the CIFER-10 data set using keras.datasets.



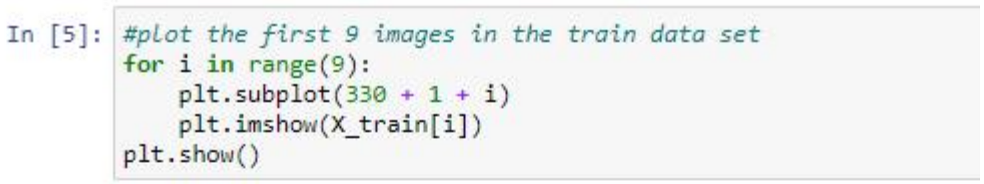
```
In [1]: from keras.datasets import cifar10
        import matplotlib.pyplot as plt
        import numpy as np
```

2. Study the shapes of the training and testing datasets.

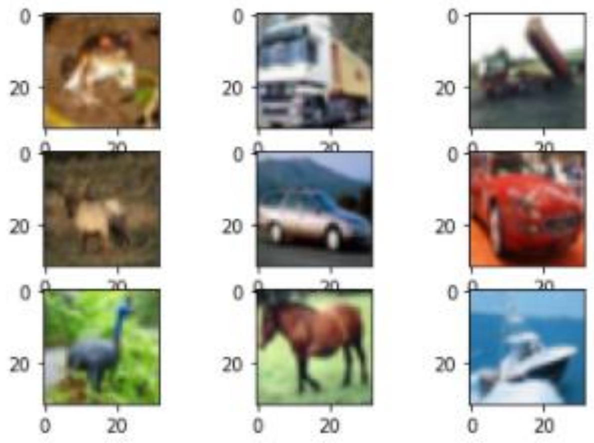


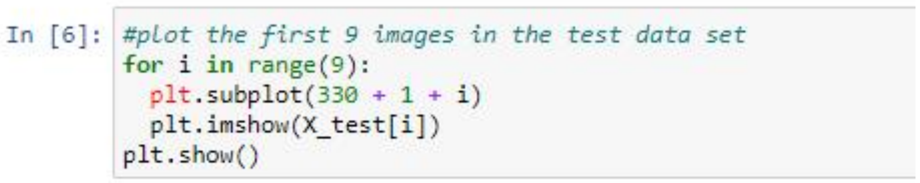
```
In [2]: np.random.seed(seed=1234)

        #download cifar10 data and split into train and test sets
        (X_train, y_train), (X_test, y_test) = cifar10.load_data()

        Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
        170500096/170498071 [==============================] - 947s 6us/step
        170508288/170498071 [==============================] - 947s 6us/step

In [3]: #check the shape of the training data set
        X_train.shape

Out[3]: (50000, 32, 32, 3)

In [4]: #check the shape of the testing data set
        X_test.shape

Out[4]: (10000, 32, 32, 3)
```

3. **Visualize some images in the train and test tests to understand the dataset. You may use mat�plotlib.pyplot.imshow to display the images in a grid.**

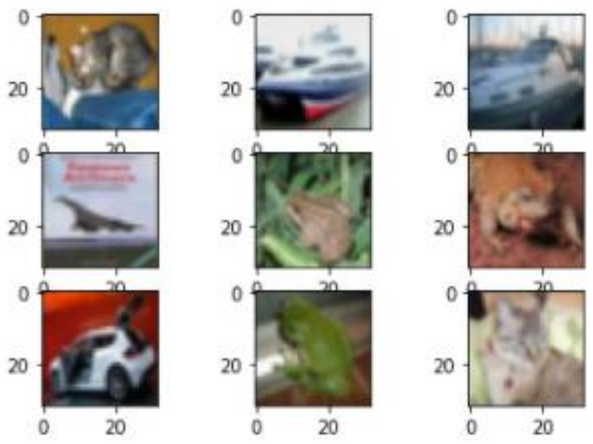


```
In [5]: #plot the first 9 images in the train data set
        for i in range(9):
            plt.subplot(330 + 1 + i)
            plt.imshow(X_train[i])
        plt.show()
```



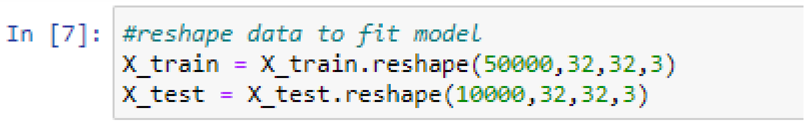


```
In [6]: #plot the first 9 images in the test data set
        for i in range(9):
          plt.subplot(330 + 1 + i)
          plt.imshow(X_test[i])
        plt.show()
```



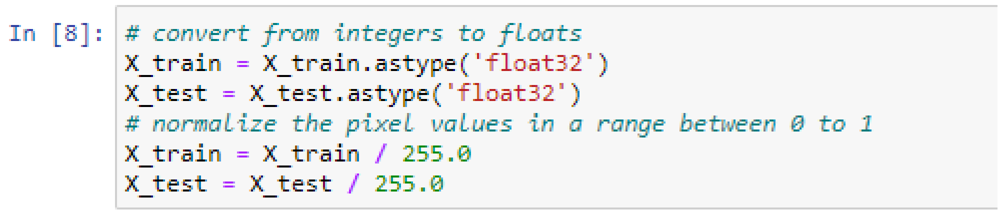
4. **Under the data pre-processing procedures,**

- **Reshape the input datasets accordingly.**



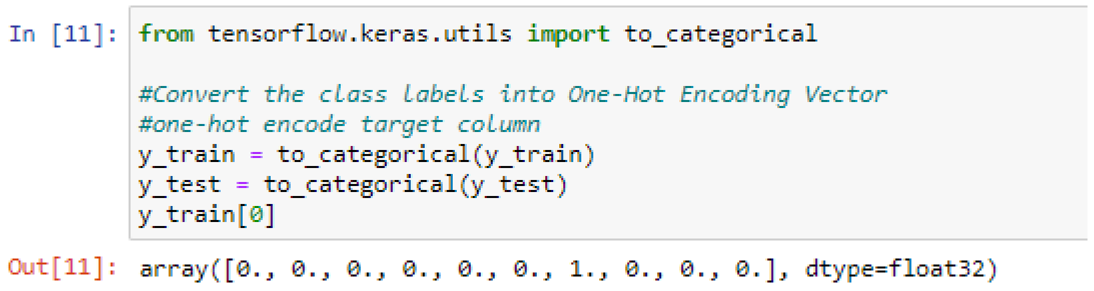
```
In [7]: #reshape data to fit model
        X_train = X_train.reshape(50000,32,32,3)
        X_test = X_test.reshape(10000,32,32,3)
```

• **Normalize the pixel values in a range between 0 to 1.**



```
In [8]: # convert from integers to floats
        X_train = X_train.astype('float32')
        X_test = X_test.astype('float32')
        # normalize the pixel values in a range between 0 to 1
        X_train = X_train / 255.0
        X_test = X_test / 255.0
```

• **Convert the class labels into One-Hot encoding vector. Clearly mention the requirement of this conversion.**

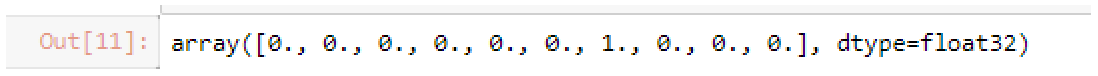


```
In [11]: from tensorflow.keras.utils import to_categorical

         #Convert the class labels into One-Hot Encoding Vector
         #one-hot encode target column
         y_train = to_categorical(y_train)
         y_test = to_categorical(y_test)
         y_train[0]

Out[11]: array([0., 0., 0., 0., 0., 0., 1., 0., 0., 0.], dtype=float32)
```

<u>**Requirement of converting the class labels into One-Hot encoding vector.**</u>

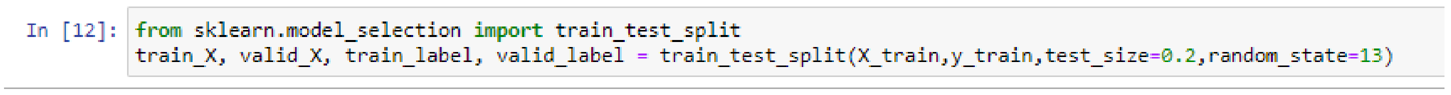
- This eliminates the hierarchy issues but does have the downside of adding more columns to the data set.
- In one-hot encoding each category value is converted into a new column and assigned 1 or 0 (similar as true/false) value to that newly inserted column.
- Each numeric value is represented as a binary vector containing all zero values except the numeric value's index, which is denoted with a 1.

<u>**The output after this conversion**</u>



```
Out[11]: array([0., 0., 0., 0., 0., 0., 1., 0., 0., 0.], dtype=float32)
```

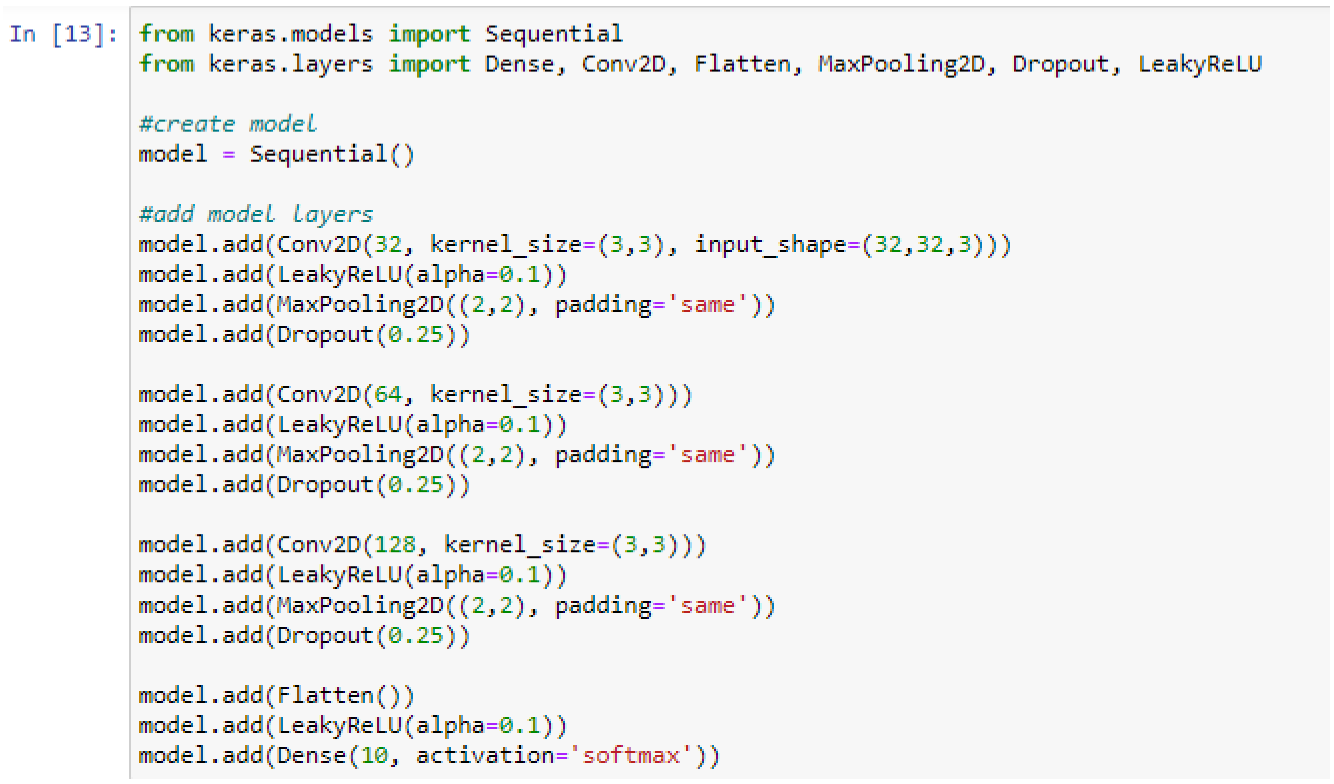
• **Use sklearn.model selection.train test split to further split the training dataset into validation and training data (e.g. allocate 0.2 of the training set as validation data).**



```
In [12]: from sklearn.model_selection import train_test_split
         train_X, valid_X, train_label, valid_label = train_test_split(X_train,y_train,test_size=0.2,random_state=13)
```

5. **Build the CNN model with three convolutional layers followed by a dense layer and an output layer accordingly. In this case,**

• Select 3 X 3 as the kernal size of each filter.
• Use different number of filters in each convolutional layer (e.g. first layer 32 filters, second layer 64 filters, third layer 128 filters).
• Use LeakyReLU as the activation function. Mention the advantage of using LeakyReLU over ReLU activation function.



```
In [13]: from keras.models import Sequential
         from keras.layers import Dense, Conv2D, Flatten, MaxPooling2D, Dropout, LeakyReLU

         #create model
         model = Sequential()

         #add model layers
         model.add(Conv2D(32, kernel_size=(3,3), input_shape=(32,32,3)))
         model.add(LeakyReLU(alpha=0.1))
         model.add(MaxPooling2D((2,2), padding='same'))
         model.add(Dropout(0.25))

         model.add(Conv2D(64, kernel_size=(3,3)))
         model.add(LeakyReLU(alpha=0.1))
         model.add(MaxPooling2D((2,2), padding='same'))
         model.add(Dropout(0.25))

         model.add(Conv2D(128, kernel_size=(3,3)))
         model.add(LeakyReLU(alpha=0.1))
         model.add(MaxPooling2D((2,2), padding='same'))
         model.add(Dropout(0.25))

         model.add(Flatten())
         model.add(LeakyReLU(alpha=0.1))
         model.add(Dense(10, activation='softmax'))
```

Advantage of using LeakyReLU over ReLU activation function

- The gradient of the ReLU activation function is 0 for all input values less than zero, which would deactivate the neurons in that region and perhaps create the dying ReLU problem.
- The term "leaky ReLU" was coined to describe a solution to this issue.

- The derivative of the LeakyReLU is 1 in the positive part, and is a small fraction in the negative part while the derivative of the ReLU is 1 in the positive part, and 0 in the negative part.

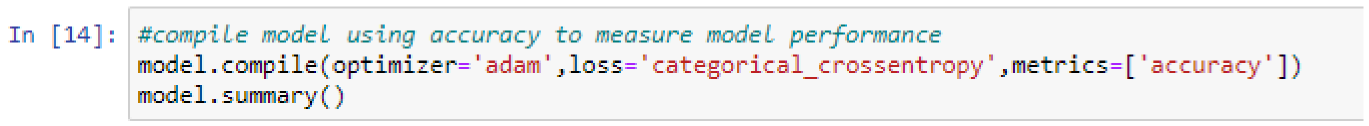
- When using LeakyReLU we can worry less about the initialization of your neural network but when using ReLU we can end up with a neural network that never learns if the neurons are not activated at the start.

• Use 2 X 2 MaxPooling layers, and Dropout layers according to the requirements and mention the purpose behind the usage of Dropout Layers.

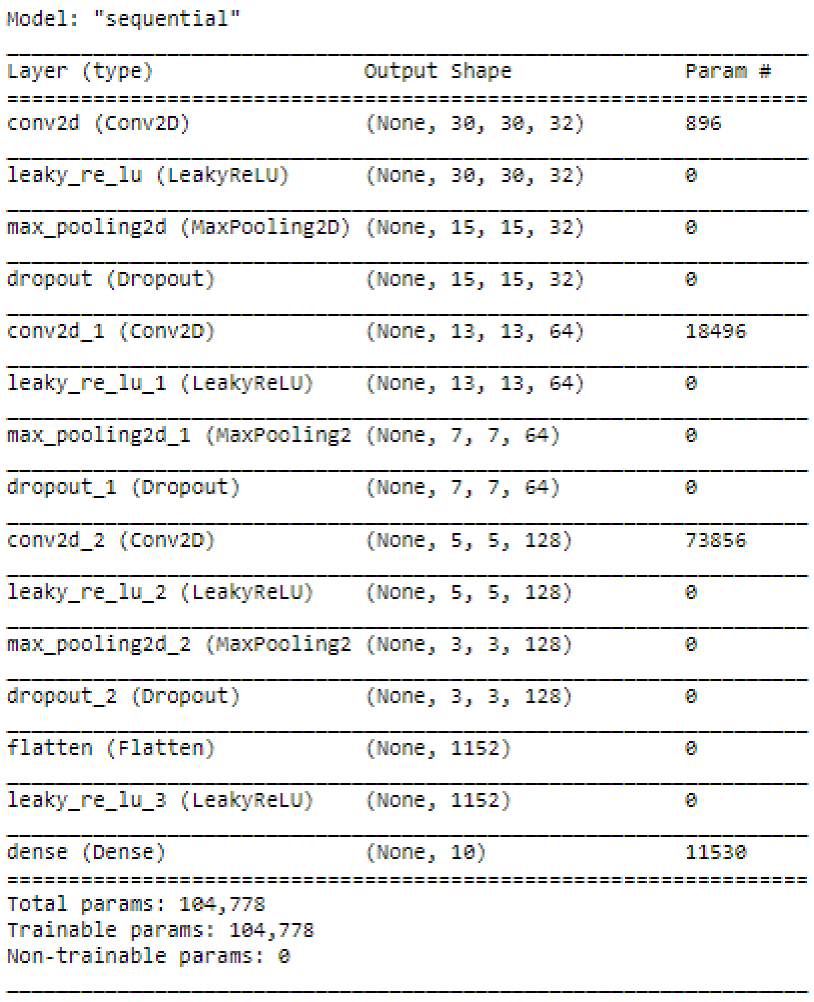
<u>**Purpose behind the usage of Dropout Layers**</u>

- It gives the effect of reducing the capacity or thinning the network during training. So that it prevent overfitting of the CNN.
- Inputs are not set to 0 are scaled up by 1/ (1 - rate). Therefore, the sum over all inputs is unchanged.

6. **Compile the model using appropriate parameters and generate the model summery using model.summary() function (In this case make sure to specify the metrics as accuracy).**



```
In [14]: #compile model using accuracy to measure model performance
         model.compile(optimizer='adam',loss='categorical_crossentropy',metrics=['accuracy'])
         model.summary()
```



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 30, 30, 32)        896
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 30, 30, 32)        0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 15, 15, 32)        0
_________________________________________________________________
dropout (Dropout)            (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 13, 13, 64)        18496
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 13, 13, 64)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
dropout_1 (Dropout)          (None, 7, 7, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 5, 5, 128)         73856
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 5, 5, 128)         0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 3, 3, 128)         0
_________________________________________________________________
dropout_2 (Dropout)          (None, 3, 3, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 1152)              0
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 1152)              0
_________________________________________________________________
dense (Dense)                (None, 10)                11530
=================================================================
Total params: 104,778
Trainable params: 104,778
Non-trainable params: 0
_________________________________________________________________
```

7. **Train the compiled model using model.fit function and observe the train and validation set per▯formances. In this case, you may have to select an appropriate number of epochs (e.g. 25) and batch size (e.g. 64, 128 or 256).**

```
In [15]: #train the model
         model.fit(train_X, train_label, validation_data=(valid_X, valid_label),epochs=25)
```

```
Epoch 1/25
1250/1250 [==============================] - 53s 40ms/step - loss: 1.5961 - accuracy: 0.4158 - val_loss: 1.2696 - val_accuracy:
0.5411
Epoch 2/25
1250/1250 [==============================] - 49s 40ms/step - loss: 1.2379 - accuracy: 0.5598 - val_loss: 1.0519 - val_accuracy:
0.6330
Epoch 3/25
1250/1250 [==============================] - 48s 39ms/step - loss: 1.0848 - accuracy: 0.6172 - val_loss: 0.9860 - val_accuracy:
0.6620
Epoch 4/25
1250/1250 [==============================] - 48s 38ms/step - loss: 0.9835 - accuracy: 0.6540 - val_loss: 0.9230 - val_accuracy:
0.6807
Epoch 5/25
1250/1250 [==============================] - 50s 40ms/step - loss: 0.9137 - accuracy: 0.6842 - val_loss: 0.8451 - val_accuracy:
0.7097
Epoch 6/25
1250/1250 [==============================] - 50s 40ms/step - loss: 0.8690 - accuracy: 0.6955 - val_loss: 0.8007 - val_accuracy:
0.7218
Epoch 7/25
1250/1250 [==============================] - 49s 40ms/step - loss: 0.8312 - accuracy: 0.7106 - val_loss: 0.7960 - val_accuracy:
0.7264
Epoch 8/25
1250/1250 [==============================] - 49s 40ms/step - loss: 0.8030 - accuracy: 0.7179 - val_loss: 0.7569 - val_accuracy:
0.7427
Epoch 9/25
1250/1250 [==============================] - 50s 40ms/step - loss: 0.7764 - accuracy: 0.7287 - val_loss: 0.8132 - val_accuracy:
0.7178
Epoch 10/25
1250/1250 [==============================] - 51s 41ms/step - loss: 0.7559 - accuracy: 0.7362 - val_loss: 0.7768 - val_accuracy:
0.7346
Epoch 11/25
1250/1250 [==============================] - 50s 40ms/step - loss: 0.7426 - accuracy: 0.7417 - val_loss: 0.7358 - val_accuracy:
0.7481
Epoch 12/25
1250/1250 [==============================] - 51s 41ms/step - loss: 0.7324 - accuracy: 0.7467 - val_loss: 0.7432 - val_accuracy:
0.7421
Epoch 13/25
1250/1250 [==============================] - 50s 40ms/step - loss: 0.7155 - accuracy: 0.7493 - val_loss: 0.7774 - val_accuracy:
0.7390
Epoch 14/25
1250/1250 [==============================] - 50s 40ms/step - loss: 0.6989 - accuracy: 0.7559 - val_loss: 0.7225 - val_accuracy:
0.7579
Epoch 15/25
1250/1250 [==============================] - 52s 42ms/step - loss: 0.6882 - accuracy: 0.7581 - val_loss: 0.7298 - val_accuracy:
0.7511
Epoch 16/25
1250/1250 [==============================] - 53s 42ms/step - loss: 0.6818 - accuracy: 0.7599 - val_loss: 0.7341 - val_accuracy:
0.7584
Epoch 17/25
1250/1250 [==============================] - 53s 42ms/step - loss: 0.6680 - accuracy: 0.7657 - val_loss: 0.7073 - val_accuracy:
0.7629
Epoch 18/25
1250/1250 [==============================] - 52s 42ms/step - loss: 0.6627 - accuracy: 0.7695 - val_loss: 0.7005 - val_accuracy:
0.7654
Epoch 19/25
1250/1250 [==============================] - 51s 41ms/step - loss: 0.6482 - accuracy: 0.7743 - val_loss: 0.8046 - val_accuracy:
0.7503
Epoch 20/25
1250/1250 [==============================] - 51s 41ms/step - loss: 0.6550 - accuracy: 0.7712 - val_loss: 0.7373 - val_accuracy:
0.7536
Epoch 21/25
1250/1250 [==============================] - 51s 41ms/step - loss: 0.6389 - accuracy: 0.7740 - val_loss: 0.7148 - val_accuracy:
0.7625
Epoch 22/25
1250/1250 [==============================] - 52s 42ms/step - loss: 0.6391 - accuracy: 0.7762 - val_loss: 0.6941 - val_accuracy:
0.7636
Epoch 23/25
1250/1250 [==============================] - 51s 41ms/step - loss: 0.6387 - accuracy: 0.7787 - val_loss: 0.7285 - val_accuracy:
0.7643
Epoch 24/25
1250/1250 [==============================] - 49s 39ms/step - loss: 0.6241 - accuracy: 0.7793 - val_loss: 0.6866 - val_accuracy:
0.7654
Epoch 25/25
1250/1250 [==============================] - 46s 37ms/step - loss: 0.6252 - accuracy: 0.7821 - val_loss: 0.6867 - val_accuracy:
0.7672
```

```
Out[15]: <keras.callbacks.History at 0x182235e9190>
```

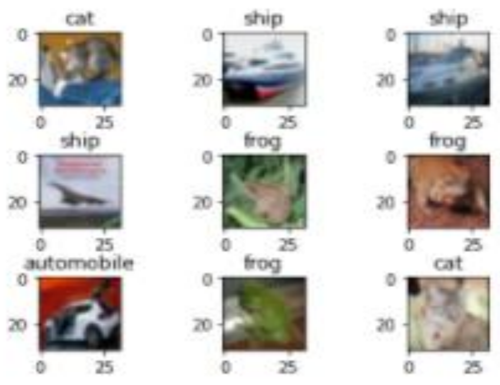
8. **Evaluate the model performance using test set. Identify the test loss and test accuracy.**

```
In [17]: #model performance using a test set
         test_eval = model.evaluate(X_test, y_test, verbose=0)
         print("Test loss:", test_eval[0])
         print("Test accuracy:", test_eval[1])

         Test loss: 0.7009048461914062
         Test accuracy: 0.7648000121116638
```

9. **Use the trained model to make predictions for the test data and visualize the model performance under each class using sklearn.metrics.classification report.**

```
In [18]: import numpy as np

         # predictions for the test data
         predictions = model.predict(X_test)
         predictions = np.argmax(np.round(predictions),axis=1)
         predictions

Out[18]: array([3, 8, 8, ..., 5, 1, 7], dtype=int64)
```

```
In [20]: from sklearn.metrics import classification_report

         # setting class names
         classes=['airplane','automobile','bird','cat','deer','dog','frog','horse','ship','truck']

         for i in range(9):
             plt.subplot(330 + 1 + i)
             plt.imshow(X_test[i])
             plt.title(classes[predictions[i]])

         plt.subplots_adjust(hspace=0.7, wspace=0)
         plt.show()

         #Classification report
         y_test_original = np.argmax(y_test,axis=1)
         print(classification_report(y_true=y_test_original, y_pred=predictions))
```



```
              precision    recall  f1-score   support

           0       0.38      0.84      0.53      1000
           1       0.95      0.81      0.87      1000
           2       0.80      0.56      0.66      1000
           3       0.70      0.47      0.56      1000
           4       0.72      0.77      0.74      1000
           5       0.77      0.60      0.67      1000
           6       0.82      0.83      0.83      1000
           7       0.90      0.70      0.79      1000
           8       0.87      0.85      0.86      1000
           9       0.88      0.83      0.85      1000

    accuracy                           0.73     10000
   macro avg       0.78      0.73      0.74     10000
weighted avg       0.78      0.73      0.74     10000
```