Digital Image Processing, Homework 3

Ashkan Zarkhah - 610399196

In this assignment, we use TensorFlow and Keras to download pre-trained CNN models like ResNet, VGGNet, EfficientNet, MobileNet, DenseNet, and ConvNet (2022). Additionally, we use PyTorch to build a GoogleNet model from scratch and train it.

Imports

```
#Imports
import os, re, time, json
import PIL.Image, PIL.ImageFont, PIL.ImageDraw
import numpy as np
import tensorflow as tf
from tensorflow.keras.applications.resnet50 import ResNet50
from matplotlib import pyplot as plt
import tensorflow_datasets as tfds
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.utils import to_categorical
from keras.layers import Dense,Flatten
from keras.datasets import cifar10
import cv2
from keras import backend as K
from keras.optimizers import SGD
from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score, ConfusionMatrixDisplay, confusion_matrix
import gc
from keras.utils import to_categorical
from tensorflow.keras.models import load_model

#Category names for visualization
classes = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog',
'frog', 'horse', 'ship', 'truck']
```

Data Loader

```
#Loading data and splitting it into train, test, and validation
def Data_loader():
    (X_train, y_train) , (X_test, y_test) =
tf.keras.datasets.cifar10.load_data()
    X_train, X_val, y_train, y_val = train_test_split(X_train,
y_train, test_size=0.2, random_state=42)
    return (X_train, y_train, X_val, y_val, X_test, y_test)
```

```
(X_train, y_train, X_val, y_val, X_test, y_test) = Data_loader()
X_test_for_display = X_test
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 6s 0us/step
```

Visaulization tool

```
# utility to display a row of images with their predictions
def display_images(X, y_pred, y, n):
  indexes = np.random.choice(len(y_pred), size=n)
  images = X[indexes]
  preds = y_pred[indexes]
  labels = y[indexes]

  preds = preds.reshape((n,))
  labels = labels.reshape((n,))

  fig = plt.figure(figsize=(3 * n, 4))
  plt.yticks([])
  plt.xticks([])

  for i in range(n):
    ax = fig.add_subplot(1, n, i+1)
    class_index = preds[i]

    plt.xlabel("Predicted label : " + classes[class_index] +  "\n" + "True label : " + classes[labels[i]])
    #plt.ylabel()
    plt.xticks([])
    plt.yticks([])
    plt.imshow(images[i])

#Displaying some images
display_images(X_train, y_train, y_train, 5)
```

Evaluation Tool

```
#Developing a function to evaluate models
def Model_evaluator(model, X_test, y_test):
    y_pred = model.predict(X_test, batch_size=64)
    y_pred = np.argmax(y_pred, axis = 1)

    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='weighted')
    recall = recall_score(y_test, y_pred, average='weighted')
    f1 = f1_score(y_test, y_pred, average='weighted')

    print(f"Accuracy: {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1 Score: {f1:.4f}")

    cm = confusion_matrix(y_test, y_pred)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm)
    disp.plot(cmap=plt.cm.Blues)
    plt.show()

    #Displaying some images
    display_images(X_test_for_display, y_test, y_pred, 5)
```

Pre-trained CNN Models with TensorFlow and Keras

ResNet50

For the preprocessing, we use `resnet.preprocess_input` to convert the input images from RGB to BGR, and then to zero-center each color channel with respect to the ImageNet dataset, without scaling.

For the model, we first add a resize layer to resize images into 224x224, and then we add 3 dense layers at the end of the model.

In separate parameter tunings, we found that the best parameters are:

- Optimizer: SGD
- Number of epochs: 5
- Batch size: 64

Preprocessing

```
#Preprocessing for Resnet 50
def ResNet50_preprocess(input_images):
    input_images = input_images.astype('float32')
    output_images =
tf.keras.applications.resnet50.preprocess_input(input_images)
```

```
    return output_images

X_train = ResNet50_preprocess(X_train)
X_val = ResNet50_preprocess(X_val)
X_test = ResNet50_preprocess(X_test)
```

Model Definition

```
#Developing ResNEt 50 model
def ResNet50_model():
    inputs = tf.keras.layers.Input(shape=(32,32,3))
    resized = tf.keras.layers.UpSampling2D(size=(7,7))(inputs)

    base_model = 
tf.keras.applications.resnet.ResNet50(input_shape=(224, 224, 3),
                                               include_top=False,
                                               weights='imagenet')
(resized)

    new_leyers = tf.keras.layers.GlobalAveragePooling2D()(base_model)
    new_leyers = tf.keras.layers.Flatten()(new_leyers)
    new_leyers = tf.keras.layers.Dense(512, activation="relu")
(new_leyers)
    new_leyers = tf.keras.layers.Dense(128, activation="relu")
(new_leyers)
    new_leyers = tf.keras.layers.Dense(10, activation="softmax", 
name="classification")(new_leyers)

    model = tf.keras.Model(inputs=inputs, outputs = new_leyers)
    model.compile(optimizer='SGD',
                loss='sparse_categorical_crossentropy',
                metrics = ['accuracy'])
    return model


model = ResNet50_model()
model.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-
applications/resnet/
resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5
94773248/94765736 [==============================] - 0s 0us/step
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         [(None, 32, 32, 3)]       0         
_________________________________________________________________
up_sampling2d (UpSampling2D) (None, 224, 224, 3)       0         
_________________________________________________________________
```

```
resnet50 (Functional)        (None, 7, 7, 2048)        23587712
_________________________________________________________________
global_average_pooling2d (Gl (None, 2048)              0
_________________________________________________________________
flatten (Flatten)            (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 512)               1049088
_________________________________________________________________
dense_1 (Dense)              (None, 128)               65664
_________________________________________________________________
classification (Dense)       (None, 10)                1290
=================================================================
Total params: 24,703,754
Trainable params: 24,650,634
Non-trainable params: 53,120
_________________________________________________________________
```

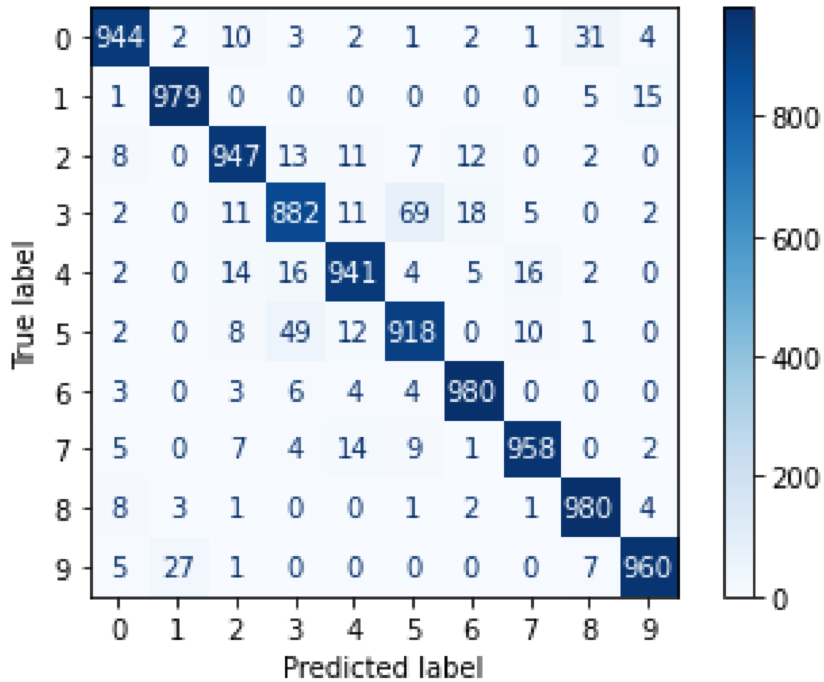
Training

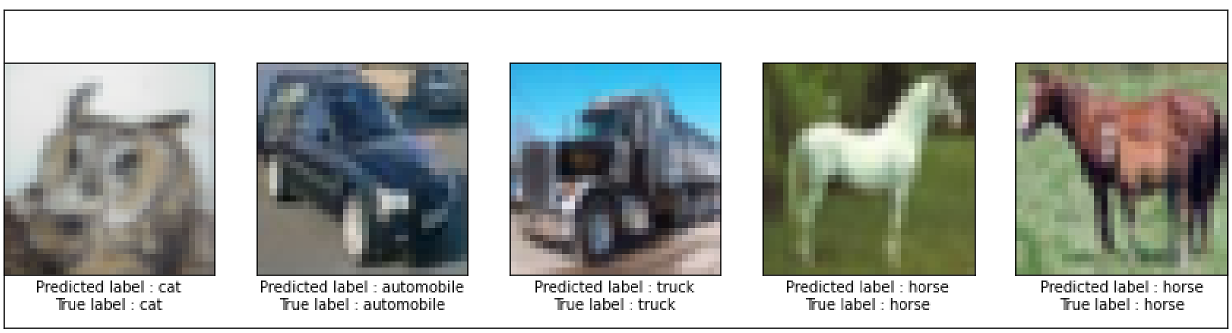
```
#Learning Resnet50 model
EPOCHS = 5
history = model.fit(X_train, y_train, epochs=EPOCHS, validation_data =
(X_val, y_val), batch_size=64)
model.save("/kaggle/working/ResNet50.h5")

Epoch 1/5
625/625 [==============================] - 204s 326ms/step - loss:
0.4530 - accuracy: 0.8498 - val_loss: 0.2478 - val_accuracy: 0.9139
Epoch 2/5
625/625 [==============================] - 203s 325ms/step - loss:
0.1138 - accuracy: 0.9623 - val_loss: 0.1746 - val_accuracy: 0.9408
Epoch 3/5
625/625 [==============================] - 203s 324ms/step - loss:
0.0367 - accuracy: 0.9899 - val_loss: 0.1820 - val_accuracy: 0.9434
Epoch 4/5
625/625 [==============================] - 203s 324ms/step - loss:
0.0145 - accuracy: 0.9971 - val_loss: 0.1761 - val_accuracy: 0.9459
Epoch 5/5
625/625 [==============================] - 203s 324ms/step - loss:
0.0074 - accuracy: 0.9988 - val_loss: 0.1764 - val_accuracy: 0.9485
```

Evaluating

```
#Evaluating resnet50 model
#model = load_model("/kaggle/working/ResNet50.h5")
Model_evaluator(model, X_test, y_test)

Accuracy: 0.9489
Precision: 0.9489
```

```
Recall: 0.9489
F1 Score: 0.9488
```






```
#Loading Data over
(X_train, y_train, X_val, y_val, X_test, y_test) = Data_loader()
```

VGGNet19

For the preprocessing, we use `vgg19.preprocess_input` to convert the input images from RGB to BGR, and then to zero-center each color channel with respect to the ImageNet dataset, without scaling.

For the model, we first add a resize layer to resize images into 48x48, and then we add 3 dense layers at the end of the model.

In separate parameter tunings, we found that the best parameters are:

- Optimizer: adam
- Number of epochs: 50
- Batch size: 256

Preprocessing

```
#Preprocessing for VGG19
def VGG19_preprocess(input_images):
    input_images = input_images.astype('float32')
    output_images = 
tf.keras.applications.vgg19.preprocess_input(input_images)
    return output_images

X_train = VGG19_preprocess(X_train)
X_val = VGG19_preprocess(X_val)
X_test = VGG19_preprocess(X_test)

y_test_for_evaluate = y_test

y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
y_val = to_categorical(y_val, 10)
```

Model Definition

```
#Developing VGG19 model
def VGG19_model():
    inputs = tf.keras.layers.Input(shape=(32,32,3))

    base_model = tf.keras.applications.vgg19.VGG19(input_shape=(48, 
48, 3),
                                                    include_top=False,
                                                    weights='imagenet')
(inputs)

    new_leyers = tf.keras.layers.GlobalAveragePooling2D()(base_model)
    new_leyers = tf.keras.layers.Flatten()(new_leyers)
    new_leyers = tf.keras.layers.Dense(512, activation="relu")
(new_leyers)
    new_leyers = tf.keras.layers.Dense(256, activation="relu")
(new_leyers)
    new_leyers = tf.keras.layers.Dense(10, activation="softmax", 
name="classification")(new_leyers)

    model = tf.keras.Model(inputs=inputs, outputs = new_leyers)
    #SGD didn't work well
    model.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics = ['accuracy'])
    return model
```

```
model = VGG19_model()
model.summary()
```

```
Model: "functional_17"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_20 (InputLayer)        [(None, 32, 32, 3)]       0         
_________________________________________________________________
vgg19 (Functional)           (None, 1, 1, 512)         20024384  
_________________________________________________________________
global_average_pooling2d_8 ( (None, 512)               0         
_________________________________________________________________
flatten_8 (Flatten)          (None, 512)               0         
_________________________________________________________________
dense_16 (Dense)             (None, 512)               262656    
_________________________________________________________________
dense_17 (Dense)             (None, 256)               131328    
_________________________________________________________________
classification (Dense)       (None, 10)                2570      
=================================================================
Total params: 20,420,938
Trainable params: 20,420,938
Non-trainable params: 0
_________________________________________________________________
```

Training

```
#Learning VGG19 model
#Bigger batch size helped the model
EPOCHS = 50
history = model.fit(X_train, y_train, epochs=EPOCHS, validation_data =
(X_val, y_val), batch_size=256)
```

```
Epoch 1/50
157/157 [==============================] - 11s 71ms/step - loss:
2.6569 - accuracy: 0.1261 - val_loss: 2.1135 - val_accuracy: 0.1679
Epoch 2/50
157/157 [==============================] - 11s 67ms/step - loss:
1.8962 - accuracy: 0.2242 - val_loss: 1.8299 - val_accuracy: 0.2355
Epoch 3/50
157/157 [==============================] - 11s 67ms/step - loss:
1.8325 - accuracy: 0.2438 - val_loss: 1.7808 - val_accuracy: 0.2734
Epoch 4/50
157/157 [==============================] - 11s 67ms/step - loss:
1.7694 - accuracy: 0.2626 - val_loss: 1.7351 - val_accuracy: 0.2932
Epoch 5/50
157/157 [==============================] - 11s 67ms/step - loss:
```

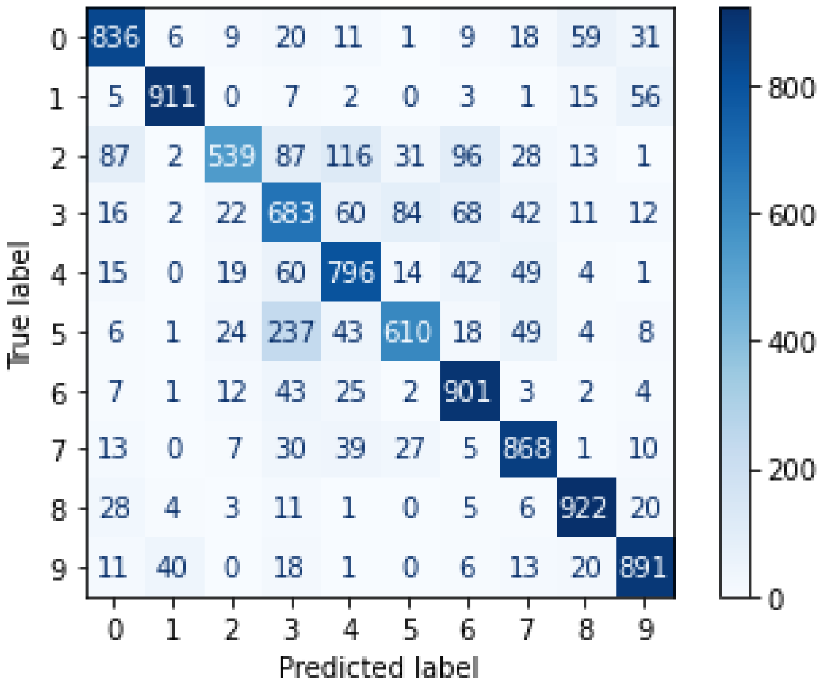
```
1.6696 - accuracy: 0.3281 - val_loss: 1.6555 - val_accuracy: 0.3065
Epoch 6/50
157/157 [==============================] - 11s 67ms/step - loss:
1.5468 - accuracy: 0.3891 - val_loss: 1.5841 - val_accuracy: 0.3695
Epoch 7/50
157/157 [==============================] - 11s 67ms/step - loss:
1.3692 - accuracy: 0.4674 - val_loss: 1.3123 - val_accuracy: 0.4869
Epoch 8/50
157/157 [==============================] - 11s 67ms/step - loss:
1.1702 - accuracy: 0.5617 - val_loss: 1.1584 - val_accuracy: 0.5813
Epoch 9/50
157/157 [==============================] - 11s 67ms/step - loss:
1.0542 - accuracy: 0.6237 - val_loss: 1.0384 - val_accuracy: 0.6280
Epoch 10/50
157/157 [==============================] - 11s 67ms/step - loss:
0.9871 - accuracy: 0.6506 - val_loss: 1.0518 - val_accuracy: 0.6339
Epoch 11/50
157/157 [==============================] - 11s 67ms/step - loss:
0.9197 - accuracy: 0.6783 - val_loss: 1.0634 - val_accuracy: 0.6385
Epoch 12/50
157/157 [==============================] - 11s 67ms/step - loss:
0.8648 - accuracy: 0.7005 - val_loss: 0.9207 - val_accuracy: 0.6898
Epoch 13/50
157/157 [==============================] - 11s 67ms/step - loss:
0.7130 - accuracy: 0.7533 - val_loss: 0.7978 - val_accuracy: 0.7280
Epoch 14/50
157/157 [==============================] - 11s 67ms/step - loss:
0.6237 - accuracy: 0.7864 - val_loss: 0.8031 - val_accuracy: 0.7304
Epoch 15/50
157/157 [==============================] - 11s 67ms/step - loss:
0.5589 - accuracy: 0.8122 - val_loss: 0.7979 - val_accuracy: 0.7414
Epoch 16/50
157/157 [==============================] - 10s 67ms/step - loss:
0.5052 - accuracy: 0.8315 - val_loss: 0.7352 - val_accuracy: 0.7657
Epoch 17/50
157/157 [==============================] - 11s 67ms/step - loss:
0.4452 - accuracy: 0.8511 - val_loss: 0.7233 - val_accuracy: 0.7737
Epoch 18/50
157/157 [==============================] - 10s 67ms/step - loss:
0.4129 - accuracy: 0.8661 - val_loss: 0.7312 - val_accuracy: 0.7784
Epoch 19/50
157/157 [==============================] - 11s 67ms/step - loss:
0.3552 - accuracy: 0.8835 - val_loss: 0.7570 - val_accuracy: 0.7732
Epoch 20/50
157/157 [==============================] - 11s 67ms/step - loss:
0.3044 - accuracy: 0.9024 - val_loss: 0.9738 - val_accuracy: 0.7144
Epoch 21/50
157/157 [==============================] - 10s 67ms/step - loss:
0.3194 - accuracy: 0.8976 - val_loss: 0.7415 - val_accuracy: 0.7897
```

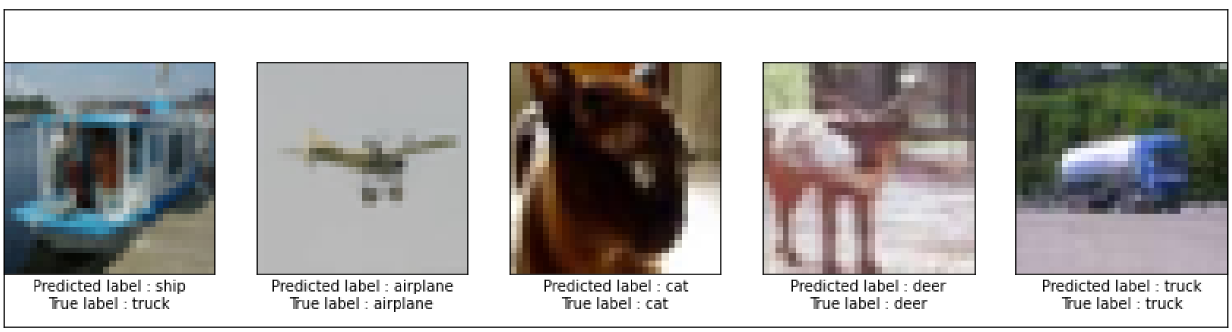
```
Epoch 22/50
157/157 [==============================] - 10s 67ms/step - loss:
0.2809 - accuracy: 0.9120 - val_loss: 0.7803 - val_accuracy: 0.7713
Epoch 23/50
157/157 [==============================] - 11s 67ms/step - loss:
0.2336 - accuracy: 0.9268 - val_loss: 0.8628 - val_accuracy: 0.7791
Epoch 24/50
157/157 [==============================] - 10s 67ms/step - loss:
0.2156 - accuracy: 0.9327 - val_loss: 0.8007 - val_accuracy: 0.7869
Epoch 25/50
157/157 [==============================] - 10s 67ms/step - loss:
0.1956 - accuracy: 0.9396 - val_loss: 0.8662 - val_accuracy: 0.7787
Epoch 26/50
157/157 [==============================] - 11s 67ms/step - loss:
0.1812 - accuracy: 0.9446 - val_loss: 0.8287 - val_accuracy: 0.7781
Epoch 27/50
157/157 [==============================] - 11s 67ms/step - loss:
0.1817 - accuracy: 0.9452 - val_loss: 0.8227 - val_accuracy: 0.7885
Epoch 28/50
157/157 [==============================] - 11s 67ms/step - loss:
0.1617 - accuracy: 0.9507 - val_loss: 0.9628 - val_accuracy: 0.7848
Epoch 29/50
157/157 [==============================] - 11s 67ms/step - loss:
0.1318 - accuracy: 0.9600 - val_loss: 0.8648 - val_accuracy: 0.7855
Epoch 30/50
157/157 [==============================] - 11s 67ms/step - loss:
0.1412 - accuracy: 0.9571 - val_loss: 0.8385 - val_accuracy: 0.7921
Epoch 31/50
157/157 [==============================] - 11s 67ms/step - loss:
0.1189 - accuracy: 0.9634 - val_loss: 0.9103 - val_accuracy: 0.7948
Epoch 32/50
157/157 [==============================] - 11s 67ms/step - loss:
0.1160 - accuracy: 0.9651 - val_loss: 0.8913 - val_accuracy: 0.7958
Epoch 33/50
157/157 [==============================] - 11s 67ms/step - loss:
0.1135 - accuracy: 0.9663 - val_loss: 0.9442 - val_accuracy: 0.7977
Epoch 34/50
157/157 [==============================] - 11s 67ms/step - loss:
0.1007 - accuracy: 0.9697 - val_loss: 0.9320 - val_accuracy: 0.7948
Epoch 35/50
157/157 [==============================] - 11s 68ms/step - loss:
0.0982 - accuracy: 0.9713 - val_loss: 0.9655 - val_accuracy: 0.7976
Epoch 36/50
157/157 [==============================] - 11s 68ms/step - loss:
0.1202 - accuracy: 0.9647 - val_loss: 0.8680 - val_accuracy: 0.7902
Epoch 37/50
157/157 [==============================] - 11s 68ms/step - loss:
0.1095 - accuracy: 0.9684 - val_loss: 0.8453 - val_accuracy: 0.7967
Epoch 38/50
```

```
157/157 [==============================] - 11s 68ms/step - loss:
0.1173 - accuracy: 0.9665 - val_loss: 0.9144 - val_accuracy: 0.8035
Epoch 39/50
157/157 [==============================] - 11s 68ms/step - loss:
0.0740 - accuracy: 0.9780 - val_loss: 1.0050 - val_accuracy: 0.7865
Epoch 40/50
157/157 [==============================] - 11s 68ms/step - loss:
0.0916 - accuracy: 0.9734 - val_loss: 0.9923 - val_accuracy: 0.8026
Epoch 41/50
157/157 [==============================] - 11s 68ms/step - loss:
0.0892 - accuracy: 0.9747 - val_loss: 0.9752 - val_accuracy: 0.7977
Epoch 42/50
157/157 [==============================] - 11s 68ms/step - loss:
0.0663 - accuracy: 0.9817 - val_loss: 1.0898 - val_accuracy: 0.7946
Epoch 43/50
157/157 [==============================] - 11s 68ms/step - loss:
0.1088 - accuracy: 0.9681 - val_loss: 1.0602 - val_accuracy: 0.7929
Epoch 44/50
157/157 [==============================] - 11s 68ms/step - loss:
0.0804 - accuracy: 0.9779 - val_loss: 1.0498 - val_accuracy: 0.7986
Epoch 45/50
157/157 [==============================] - 11s 68ms/step - loss:
0.0869 - accuracy: 0.9742 - val_loss: 1.0590 - val_accuracy: 0.7962
Epoch 46/50
157/157 [==============================] - 11s 68ms/step - loss:
0.0739 - accuracy: 0.9791 - val_loss: 1.0336 - val_accuracy: 0.7999
Epoch 47/50
157/157 [==============================] - 11s 68ms/step - loss:
0.0657 - accuracy: 0.9807 - val_loss: 0.9387 - val_accuracy: 0.7946
Epoch 48/50
157/157 [==============================] - 11s 68ms/step - loss:
0.0829 - accuracy: 0.9772 - val_loss: 1.1676 - val_accuracy: 0.8035
Epoch 49/50
157/157 [==============================] - 11s 68ms/step - loss:
0.0748 - accuracy: 0.9787 - val_loss: 1.0667 - val_accuracy: 0.7907
Epoch 50/50
157/157 [==============================] - 11s 68ms/step - loss:
0.0759 - accuracy: 0.9791 - val_loss: 1.0344 - val_accuracy: 0.7972
```

Evaluation

```
#Evaluating VGG19 model
Model_evaluator(model, X_test, y_test_for_evaluate)

Accuracy: 0.7957
Precision: 0.8026
Recall: 0.7957
F1 Score: 0.7932
```






```python
#Loading Data over
(X_train, y_train, X_val, y_val, X_test, y_test) = Data_loader()
```

EfficientNet

For the preprocessing, we use `efficientnet.preprocess_input` to convert the input images from RGB to BGR, and then to zero-center each color channel with respect to the ImageNet dataset, without scaling.

For the model, for fine tuning, we add 3 dense layers at the end of the model.

In separate parameter tunings, we found that the best parameters are:

- Optimizer: adam
- Number of epochs: 20
- Batch size: 256

Preprocessing

```
#Preprocessing for VGG19
def EfficientNetB0_preprocess(input_images):
    input_images = input_images.astype('float32')
    output_images =
tf.keras.applications.efficientnet.preprocess_input(input_images)
    return output_images

X_train = EfficientNetB0_preprocess(X_train)
X_val = EfficientNetB0_preprocess(X_val)
X_test = EfficientNetB0_preprocess(X_test)

y_test_for_evaluate = y_test

y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
y_val = to_categorical(y_val, 10)
```

Model Definition

```
#Developing EfficientNetB0 model
def EfficientNetB0_model():
    inputs = tf.keras.layers.Input(shape=(32,32,3))

    base_model =
tf.keras.applications.efficientnet.EfficientNetB0(input_shape=(32, 32,
3),
                                                               include_top=False,
                                                               weights='imagenet')
(inputs)

    new_leyers = tf.keras.layers.GlobalAveragePooling2D()(base_model)
    new_leyers = tf.keras.layers.Flatten()(new_leyers)
    new_leyers = tf.keras.layers.Dense(512, activation="relu")
(new_leyers)
    new_leyers = tf.keras.layers.Dense(256, activation="relu")
(new_leyers)
    new_leyers = tf.keras.layers.Dense(10, activation="softmax",
name="classification")(new_leyers)

    model = tf.keras.Model(inputs=inputs, outputs = new_leyers)
    #Like VGG adam worked better
    model.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics = ['accuracy'])
    return model
```

```python
model = EfficientNetB0_model()
model.summary()
```

```
Model: "functional_29"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_32 (InputLayer)        [(None, 32, 32, 3)]       0         
_________________________________________________________________
efficientnetb0 (Functional)  (None, 1, 1, 1280)        4049571   
_________________________________________________________________
global_average_pooling2d_14  (None, 1280)              0         
_________________________________________________________________
flatten_14 (Flatten)         (None, 1280)              0         
_________________________________________________________________
dense_28 (Dense)             (None, 512)               655872    
_________________________________________________________________
dense_29 (Dense)             (None, 256)               131328    
_________________________________________________________________
classification (Dense)       (None, 10)                2570      
=================================================================
Total params: 4,839,341
Trainable params: 4,797,318
Non-trainable params: 42,023
_________________________________________________________________
```

Training

```python
#Learning EfficientNetB0 model
#Like VGG batch size helped the model
EPOCHS = 20
history = model.fit(X_train, y_train, epochs=EPOCHS, validation_data =
(X_val, y_val), batch_size=256)
```

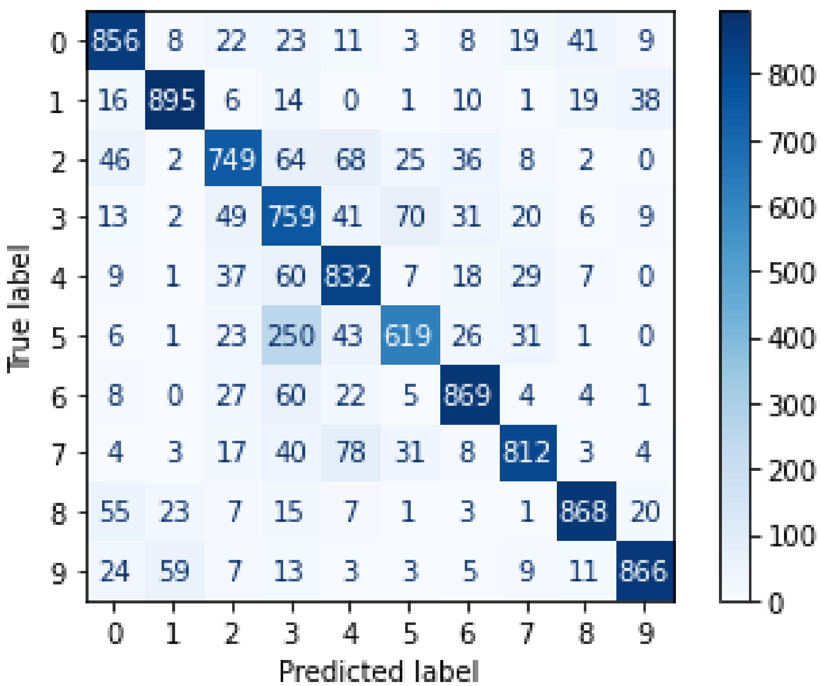
```
Epoch 1/20
157/157 [==============================] - 10s 63ms/step - loss:
1.0702 - accuracy: 0.6276 - val_loss: 0.7871 - val_accuracy: 0.7322
Epoch 2/20
157/157 [==============================] - 8s 53ms/step - loss: 0.5919
- accuracy: 0.7962 - val_loss: 0.6289 - val_accuracy: 0.7909
Epoch 3/20
157/157 [==============================] - 8s 53ms/step - loss: 0.4248
- accuracy: 0.8524 - val_loss: 0.6179 - val_accuracy: 0.7984
Epoch 4/20
157/157 [==============================] - 8s 53ms/step - loss: 0.3279
- accuracy: 0.8853 - val_loss: 0.6513 - val_accuracy: 0.8063
Epoch 5/20
157/157 [==============================] - 8s 53ms/step - loss: 0.2619
- accuracy: 0.9082 - val_loss: 0.6313 - val_accuracy: 0.8089
```

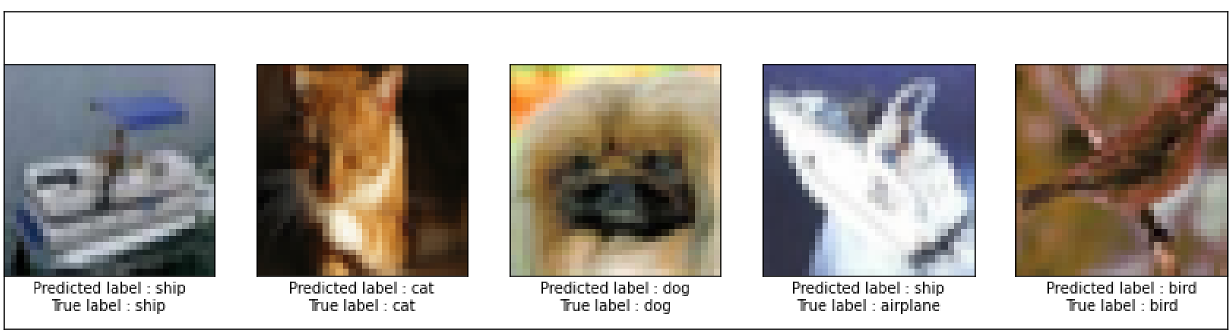
```
Epoch 6/20
157/157 [==============================] - 8s 53ms/step - loss: 0.2134
- accuracy: 0.9269 - val_loss: 0.7366 - val_accuracy: 0.7964
Epoch 7/20
157/157 [==============================] - 8s 54ms/step - loss: 0.1777
- accuracy: 0.9391 - val_loss: 0.7291 - val_accuracy: 0.8073
Epoch 8/20
157/157 [==============================] - 8s 53ms/step - loss: 0.1661
- accuracy: 0.9421 - val_loss: 0.7948 - val_accuracy: 0.7997
Epoch 9/20
157/157 [==============================] - 8s 53ms/step - loss: 0.1392
- accuracy: 0.9517 - val_loss: 0.7833 - val_accuracy: 0.8147
Epoch 10/20
157/157 [==============================] - 8s 53ms/step - loss: 0.1355
- accuracy: 0.9528 - val_loss: 0.7216 - val_accuracy: 0.8250
Epoch 11/20
157/157 [==============================] - 8s 53ms/step - loss: 0.1070
- accuracy: 0.9629 - val_loss: 0.8272 - val_accuracy: 0.8185
Epoch 12/20
157/157 [==============================] - 8s 53ms/step - loss: 0.0950
- accuracy: 0.9676 - val_loss: 0.7882 - val_accuracy: 0.8175
Epoch 13/20
157/157 [==============================] - 8s 53ms/step - loss: 0.0864
- accuracy: 0.9709 - val_loss: 0.8525 - val_accuracy: 0.8213
Epoch 14/20
157/157 [==============================] - 8s 53ms/step - loss: 0.0817
- accuracy: 0.9721 - val_loss: 0.8530 - val_accuracy: 0.8223
Epoch 15/20
157/157 [==============================] - 8s 53ms/step - loss: 0.0838
- accuracy: 0.9732 - val_loss: 0.7954 - val_accuracy: 0.8203
Epoch 16/20
157/157 [==============================] - 8s 53ms/step - loss: 0.0814
- accuracy: 0.9724 - val_loss: 0.8283 - val_accuracy: 0.8212
Epoch 17/20
157/157 [==============================] - 8s 53ms/step - loss: 0.0792
- accuracy: 0.9739 - val_loss: 0.7825 - val_accuracy: 0.8311
Epoch 18/20
157/157 [==============================] - 8s 53ms/step - loss: 0.0757
- accuracy: 0.9759 - val_loss: 0.8120 - val_accuracy: 0.8222
Epoch 19/20
157/157 [==============================] - 8s 53ms/step - loss: 0.0615
- accuracy: 0.9792 - val_loss: 0.8685 - val_accuracy: 0.8153
Epoch 20/20
157/157 [==============================] - 8s 53ms/step - loss: 0.0580
- accuracy: 0.9809 - val_loss: 0.8902 - val_accuracy: 0.8192
```

Evaluation

```
#Evaluating EfficientNetB0 model
Model_evaluator(model, X_test, y_test_for_evaluate)
```

```
Accuracy: 0.8125
Precision: 0.8209
Recall: 0.8125
F1 Score: 0.8138
```






```
#Loading Data over
(X_train, y_train, X_val, y_val, X_test, y_test) = Data_loader()
```

MobileNet

For the preprocessing, we use `mobilenet.preprocess_input` to convert the input images from RGB to BGR, and then to zero-center each color channel with respect to the ImageNet dataset, without scaling.

For the model, to stop overfitting, we only add one leyer at the end of the model. additionaly, to stop overfitting even more, we use l2 regularization and 0.5 drop out on the last layer, too.

In separate parameter tunings, we found that the best parameters are:

- Optimizer: adam
- Number of epochs: 50
- Batch size: 512

Preprocessing

```
#Preprocessing for MobileNetV2
def MobileNetV2_preprocess(input_images):
    input_images = input_images.astype('float32')
    output_images =
tf.keras.applications.mobilenet_v2.preprocess_input(input_images)
    return output_images

X_train = MobileNetV2_preprocess(X_train)
X_val = MobileNetV2_preprocess(X_val)
X_test = MobileNetV2_preprocess(X_test)

y_test_for_evaluate = y_test

y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
y_val = to_categorical(y_val, 10)
```

Model Definition

```
#Developing MobileNetV2 model
def MobileNetV2_model():
    inputs = tf.keras.layers.Input(shape=(32,32,3))

    base_model =
tf.keras.applications.mobilenet_v2.MobileNetV2(input_shape=(32, 32,
3),
                                                include_top=False,
                                                weights='imagenet')
(inputs)

    new_leyers = tf.keras.layers.GlobalAveragePooling2D()(base_model)
    new_leyers = tf.keras.layers.Flatten()(new_leyers)
    #Regularization to stop overfitting
```

```
    new_leyers = tf.keras.layers.Dense(512, activation="relu",
kernel_regularizer=tf.keras.regularizers.l2(0.01))(new_leyers)
    #Drop out to stop overfitting
    new_leyers = tf.keras.layers.Dropout(0.5)(new_leyers)
    new_leyers = tf.keras.layers.Dense(10, activation="softmax",
name="classification")(new_leyers)

    model = tf.keras.Model(inputs=inputs, outputs = new_leyers)
    model.compile(optimizer='adam',
                loss='categorical_crossentropy',
                metrics = ['accuracy'])
    return model


model = MobileNetV2_model()
model.summary()
```

```
Model: "functional_51"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_61 (InputLayer)        [(None, 32, 32, 3)]       0
_________________________________________________________________
mobilenetv2_1.00_224 (Functi (None, 1, 1, 1280)        2257984
_________________________________________________________________
global_average_pooling2d_28  (None, 1280)              0
_________________________________________________________________
flatten_28 (Flatten)         (None, 1280)              0
_________________________________________________________________
dense_54 (Dense)             (None, 512)               655872
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
classification (Dense)       (None, 10)                5130
=================================================================
Total params: 2,918,986
Trainable params: 2,884,874
Non-trainable params: 34,112
_________________________________________________________________
```

Training

```
#Learning MobileNetV2 model
#Like VGG batch size helped the model
EPOCHS = 50
history = model.fit(X_train, y_train, epochs=EPOCHS, validation_data =
(X_val, y_val), batch_size=512)
```

```
Epoch 1/50
79/79 [==============================] - 6s 71ms/step - loss: 5.4101 -
```

```
accuracy: 0.4857 - val_loss: 4.6127 - val_accuracy: 0.2512
Epoch 2/50
79/79 [==============================] - 5s 58ms/step - loss: 1.4620 -
accuracy: 0.6988 - val_loss: 6.4582 - val_accuracy: 0.2225
Epoch 3/50
79/79 [==============================] - 4s 57ms/step - loss: 0.8103 -
accuracy: 0.7706 - val_loss: 5.6539 - val_accuracy: 0.2414
Epoch 4/50
79/79 [==============================] - 4s 57ms/step - loss: 0.6132 -
accuracy: 0.8174 - val_loss: 7.5627 - val_accuracy: 0.1829
Epoch 5/50
79/79 [==============================] - 4s 57ms/step - loss: 0.5188 -
accuracy: 0.8489 - val_loss: 5.4429 - val_accuracy: 0.2084
Epoch 6/50
79/79 [==============================] - 4s 57ms/step - loss: 0.4883 -
accuracy: 0.8588 - val_loss: 5.3809 - val_accuracy: 0.2416
Epoch 7/50
79/79 [==============================] - 4s 57ms/step - loss: 0.4131 -
accuracy: 0.8816 - val_loss: 3.3687 - val_accuracy: 0.3817
Epoch 8/50
79/79 [==============================] - 4s 57ms/step - loss: 0.3604 -
accuracy: 0.8980 - val_loss: 4.6307 - val_accuracy: 0.3285
Epoch 9/50
79/79 [==============================] - 5s 57ms/step - loss: 0.3499 -
accuracy: 0.9057 - val_loss: 5.6521 - val_accuracy: 0.2420
Epoch 10/50
79/79 [==============================] - 4s 57ms/step - loss: 0.2895 -
accuracy: 0.9200 - val_loss: 3.5670 - val_accuracy: 0.3537
Epoch 11/50
79/79 [==============================] - 4s 57ms/step - loss: 0.2864 -
accuracy: 0.9219 - val_loss: 4.1258 - val_accuracy: 0.3513
Epoch 12/50
79/79 [==============================] - 4s 57ms/step - loss: 0.2709 -
accuracy: 0.9257 - val_loss: 4.0196 - val_accuracy: 0.3648
Epoch 13/50
79/79 [==============================] - 5s 57ms/step - loss: 0.2232 -
accuracy: 0.9416 - val_loss: 2.5414 - val_accuracy: 0.5003
Epoch 14/50
79/79 [==============================] - 4s 57ms/step - loss: 0.2167 -
accuracy: 0.9422 - val_loss: 4.0927 - val_accuracy: 0.3470
Epoch 15/50
79/79 [==============================] - 5s 57ms/step - loss: 0.2664 -
accuracy: 0.9330 - val_loss: 2.5343 - val_accuracy: 0.4696
Epoch 16/50
79/79 [==============================] - 5s 57ms/step - loss: 0.2329 -
accuracy: 0.9396 - val_loss: 4.7361 - val_accuracy: 0.3260
Epoch 17/50
79/79 [==============================] - 4s 57ms/step - loss: 0.1716 -
accuracy: 0.9561 - val_loss: 3.3006 - val_accuracy: 0.4211
```

```
Epoch 18/50
79/79 [==============================] - 4s 57ms/step - loss: 0.1872 -
accuracy: 0.9525 - val_loss: 2.6072 - val_accuracy: 0.4938
Epoch 19/50
79/79 [==============================] - 4s 57ms/step - loss: 0.1659 -
accuracy: 0.9582 - val_loss: 4.4506 - val_accuracy: 0.3460
Epoch 20/50
79/79 [==============================] - 4s 57ms/step - loss: 0.1760 -
accuracy: 0.9563 - val_loss: 2.6353 - val_accuracy: 0.4993
Epoch 21/50
79/79 [==============================] - 4s 57ms/step - loss: 0.2021 -
accuracy: 0.9505 - val_loss: 2.1479 - val_accuracy: 0.5895
Epoch 22/50
79/79 [==============================] - 4s 57ms/step - loss: 0.1656 -
accuracy: 0.9592 - val_loss: 2.0448 - val_accuracy: 0.6285
Epoch 23/50
79/79 [==============================] - 5s 57ms/step - loss: 0.1733 -
accuracy: 0.9581 - val_loss: 2.3271 - val_accuracy: 0.6083
Epoch 24/50
79/79 [==============================] - 4s 56ms/step - loss: 0.1514 -
accuracy: 0.9646 - val_loss: 2.4254 - val_accuracy: 0.6075
Epoch 25/50
79/79 [==============================] - 4s 57ms/step - loss: 0.1660 -
accuracy: 0.9619 - val_loss: 3.0645 - val_accuracy: 0.5513
Epoch 26/50
79/79 [==============================] - 4s 57ms/step - loss: 0.1213 -
accuracy: 0.9708 - val_loss: 2.3634 - val_accuracy: 0.5959
Epoch 27/50
79/79 [==============================] - 5s 57ms/step - loss: 0.1306 -
accuracy: 0.9688 - val_loss: 2.3982 - val_accuracy: 0.6209
Epoch 28/50
79/79 [==============================] - 4s 57ms/step - loss: 0.1070 -
accuracy: 0.9749 - val_loss: 2.4013 - val_accuracy: 0.6125
Epoch 29/50
79/79 [==============================] - 4s 57ms/step - loss: 0.1374 -
accuracy: 0.9675 - val_loss: 2.7224 - val_accuracy: 0.5962
Epoch 30/50
79/79 [==============================] - 4s 57ms/step - loss: 0.1305 -
accuracy: 0.9698 - val_loss: 2.1058 - val_accuracy: 0.6446
Epoch 31/50
79/79 [==============================] - 4s 57ms/step - loss: 0.1110 -
accuracy: 0.9747 - val_loss: 2.6764 - val_accuracy: 0.5830
Epoch 32/50
79/79 [==============================] - 4s 57ms/step - loss: 0.1114 -
accuracy: 0.9748 - val_loss: 2.4300 - val_accuracy: 0.6382
Epoch 33/50
79/79 [==============================] - 4s 57ms/step - loss: 0.2315 -
accuracy: 0.9455 - val_loss: 2.5927 - val_accuracy: 0.6129
Epoch 34/50
```

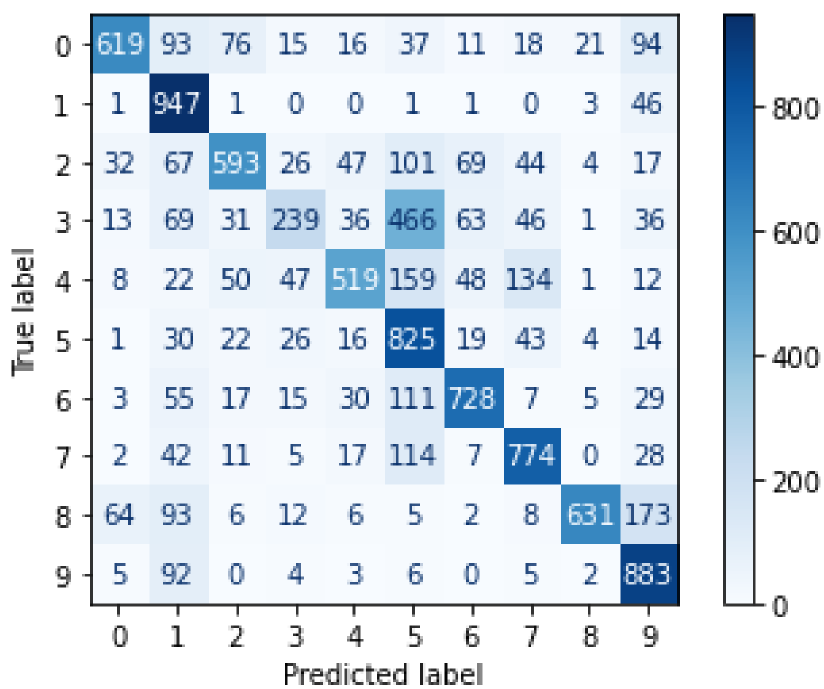
```
79/79 [==============================] - 4s 57ms/step - loss: 0.0938 -
accuracy: 0.9803 - val_loss: 2.6339 - val_accuracy: 0.6057
Epoch 35/50
79/79 [==============================] - 4s 57ms/step - loss: 0.2243 -
accuracy: 0.9632 - val_loss: 2.4458 - val_accuracy: 0.6593
Epoch 36/50
79/79 [==============================] - 4s 57ms/step - loss: 0.1188 -
accuracy: 0.9740 - val_loss: 1.9529 - val_accuracy: 0.7120
Epoch 37/50
79/79 [==============================] - 4s 57ms/step - loss: 0.1003 -
accuracy: 0.9805 - val_loss: 2.0048 - val_accuracy: 0.7205
Epoch 38/50
79/79 [==============================] - 4s 57ms/step - loss: 0.1257 -
accuracy: 0.9707 - val_loss: 2.0086 - val_accuracy: 0.6945
Epoch 39/50
79/79 [==============================] - 4s 57ms/step - loss: 0.0930 -
accuracy: 0.9807 - val_loss: 2.2424 - val_accuracy: 0.6771
Epoch 40/50
79/79 [==============================] - 4s 57ms/step - loss: 0.1447 -
accuracy: 0.9704 - val_loss: 2.4848 - val_accuracy: 0.6707
Epoch 41/50
79/79 [==============================] - 4s 57ms/step - loss: 0.1113 -
accuracy: 0.9740 - val_loss: 3.7992 - val_accuracy: 0.5663
Epoch 42/50
79/79 [==============================] - 4s 57ms/step - loss: 0.1110 -
accuracy: 0.9775 - val_loss: 2.8997 - val_accuracy: 0.6178
Epoch 43/50
79/79 [==============================] - 4s 57ms/step - loss: 0.1274 -
accuracy: 0.9727 - val_loss: 2.7979 - val_accuracy: 0.6512
Epoch 44/50
79/79 [==============================] - 5s 57ms/step - loss: 0.1588 -
accuracy: 0.9638 - val_loss: 3.0105 - val_accuracy: 0.5987
Epoch 45/50
79/79 [==============================] - 4s 57ms/step - loss: 0.0991 -
accuracy: 0.9785 - val_loss: 3.3308 - val_accuracy: 0.6144
Epoch 46/50
79/79 [==============================] - 4s 57ms/step - loss: 0.0925 -
accuracy: 0.9796 - val_loss: 3.1410 - val_accuracy: 0.5971
Epoch 47/50
79/79 [==============================] - 4s 57ms/step - loss: 0.1329 -
accuracy: 0.9743 - val_loss: 2.4663 - val_accuracy: 0.6594
Epoch 48/50
79/79 [==============================] - 4s 57ms/step - loss: 0.0811 -
accuracy: 0.9836 - val_loss: 2.5988 - val_accuracy: 0.6621
Epoch 49/50
79/79 [==============================] - 4s 57ms/step - loss: 0.1043 -
accuracy: 0.9786 - val_loss: 2.6676 - val_accuracy: 0.6545
Epoch 50/50
```

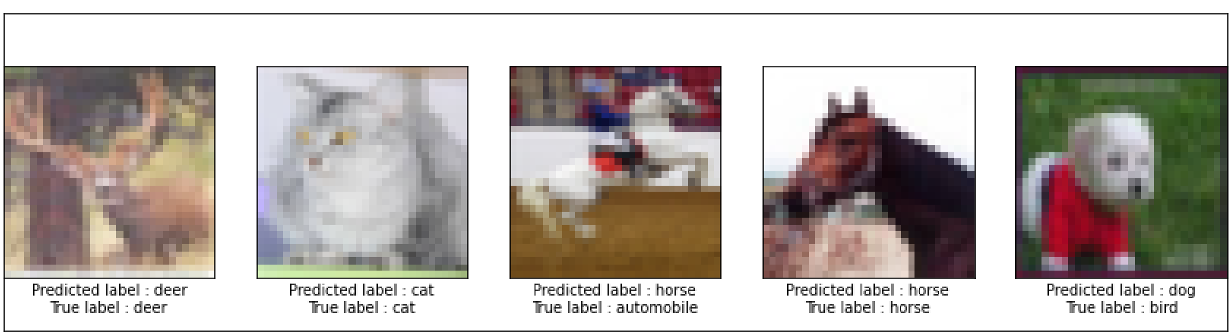
```
79/79 [==============================] - 4s 56ms/step - loss: 0.1052 -
accuracy: 0.9777 - val_loss: 2.2046 - val_accuracy: 0.6912
```

Evaluating

```
#Evaluating MobileNetV2 model
Model_evaluator(model, X_test, y_test_for_evaluate)
```

```
Accuracy: 0.6758
Precision: 0.7095
Recall: 0.6758
F1 Score: 0.6666
```






```
#Loading Data over
(X_train, y_train, X_val, y_val, X_test, y_test) = Data_loader()
```

DenseNet

For the preprocessing, we use `densenet.preprocess_input` to convert the input images from RGB to BGR, and then to zero-center each color channel with respect to the ImageNet dataset, without scaling.

For the model, to stop overfitting, we add two leyers instead of three, at the end of the model. additionaly, to stop overfitting even more, we use a 0.5 drop out on the last layer.

In separate parameter tunings, we found that the best parameters are:

- Optimizer: SGD
- Number of epochs: 30
- Batch size: 128

Preprocessing

```
#Preprocessing for DenseNet169
def DenseNet169_preprocess(input_images):
    input_images = input_images.astype('float32')
    output_images = 
tf.keras.applications.densenet.preprocess_input(input_images)
    return output_images

X_train = DenseNet169_preprocess(X_train)
X_val = DenseNet169_preprocess(X_val)
X_test = DenseNet169_preprocess(X_test)

y_test_for_evaluate = y_test

y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
y_val = to_categorical(y_val, 10)
```

Model Definition

```
#Developing DenseNet169 model
def DenseNet169_model():
    inputs = tf.keras.layers.Input(shape=(32,32,3))

    base_model = 
tf.keras.applications.densenet.DenseNet169(input_shape=(32, 32, 3),
                                                include_top=False,
                                                weights='imagenet')
(inputs)

    new_leyers = tf.keras.layers.GlobalAveragePooling2D()(base_model)
    new_leyers = tf.keras.layers.Flatten()(new_leyers)
    new_leyers = tf.keras.layers.Dense(1024, activation="relu")
(new_leyers)
```

```
    new_leyers = tf.keras.layers.Dense(256, activation="relu")
(new_leyers)
    new_leyers = tf.keras.layers.Dropout(0.5)(new_leyers)
    new_leyers = tf.keras.layers.Dense(10, activation="softmax",
name="classification")(new_leyers)

    model = tf.keras.Model(inputs=inputs, outputs = new_leyers)
    model.compile(optimizer='SGD',
                loss='categorical_crossentropy',
                metrics = ['accuracy'])
    return model


model = DenseNet169_model()
model.summary()
```

```
Model: "functional_61"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_71 (InputLayer)        [(None, 32, 32, 3)]       0
_________________________________________________________________
densenet169 (Functional)     (None, 1, 1, 1664)        12642880
_________________________________________________________________
global_average_pooling2d_33  (None, 1664)              0
_________________________________________________________________
flatten_33 (Flatten)         (None, 1664)              0
_________________________________________________________________
dense_62 (Dense)             (None, 1024)              1704960
_________________________________________________________________
dense_63 (Dense)             (None, 256)               262400
_________________________________________________________________
dropout_3 (Dropout)          (None, 256)               0
_________________________________________________________________
classification (Dense)       (None, 10)                2570
=================================================================
Total params: 14,612,810
Trainable params: 14,454,410
Non-trainable params: 158,400
_________________________________________________________________
```

Training

```
#Learning DenseNet169 model
#More complexity, less batch size
EPOCHS = 30
history = model.fit(X_train, y_train, epochs=EPOCHS, validation_data =
(X_val, y_val), batch_size=128)
```

```
Epoch 1/30
313/313 [==============================] - 25s 78ms/step - loss:
1.3596 - accuracy: 0.5317 - val_loss: 0.8221 - val_accuracy: 0.7261
Epoch 2/30
313/313 [==============================] - 21s 68ms/step - loss:
0.7182 - accuracy: 0.7597 - val_loss: 0.6437 - val_accuracy: 0.7832
Epoch 3/30
313/313 [==============================] - 21s 67ms/step - loss:
0.5143 - accuracy: 0.8285 - val_loss: 0.5660 - val_accuracy: 0.8123
Epoch 4/30
313/313 [==============================] - 21s 68ms/step - loss:
0.3758 - accuracy: 0.8771 - val_loss: 0.5756 - val_accuracy: 0.8161
Epoch 5/30
313/313 [==============================] - 21s 68ms/step - loss:
0.2768 - accuracy: 0.9089 - val_loss: 0.5716 - val_accuracy: 0.8280
Epoch 6/30
313/313 [==============================] - 21s 68ms/step - loss:
0.2015 - accuracy: 0.9348 - val_loss: 0.6025 - val_accuracy: 0.8257
Epoch 7/30
313/313 [==============================] - 21s 68ms/step - loss:
0.1490 - accuracy: 0.9503 - val_loss: 0.6552 - val_accuracy: 0.8279
Epoch 8/30
313/313 [==============================] - 21s 68ms/step - loss:
0.1113 - accuracy: 0.9645 - val_loss: 0.6774 - val_accuracy: 0.8267
Epoch 9/30
313/313 [==============================] - 21s 68ms/step - loss:
0.0858 - accuracy: 0.9726 - val_loss: 0.7204 - val_accuracy: 0.8339
Epoch 10/30
313/313 [==============================] - 21s 68ms/step - loss:
0.0708 - accuracy: 0.9773 - val_loss: 0.7613 - val_accuracy: 0.8312
Epoch 11/30
313/313 [==============================] - 21s 68ms/step - loss:
0.0556 - accuracy: 0.9819 - val_loss: 0.7985 - val_accuracy: 0.8265
Epoch 12/30
313/313 [==============================] - 21s 68ms/step - loss:
0.0540 - accuracy: 0.9829 - val_loss: 0.8063 - val_accuracy: 0.8327
Epoch 13/30
313/313 [==============================] - 21s 68ms/step - loss:
0.0451 - accuracy: 0.9860 - val_loss: 0.8189 - val_accuracy: 0.8316
Epoch 14/30
313/313 [==============================] - 21s 67ms/step - loss:
0.0389 - accuracy: 0.9880 - val_loss: 0.8190 - val_accuracy: 0.8344
Epoch 15/30
313/313 [==============================] - 21s 67ms/step - loss:
0.0331 - accuracy: 0.9891 - val_loss: 0.8249 - val_accuracy: 0.8381
Epoch 16/30
313/313 [==============================] - 21s 68ms/step - loss:
0.0338 - accuracy: 0.9893 - val_loss: 0.8324 - val_accuracy: 0.8369
Epoch 17/30
313/313 [==============================] - 21s 67ms/step - loss:
```

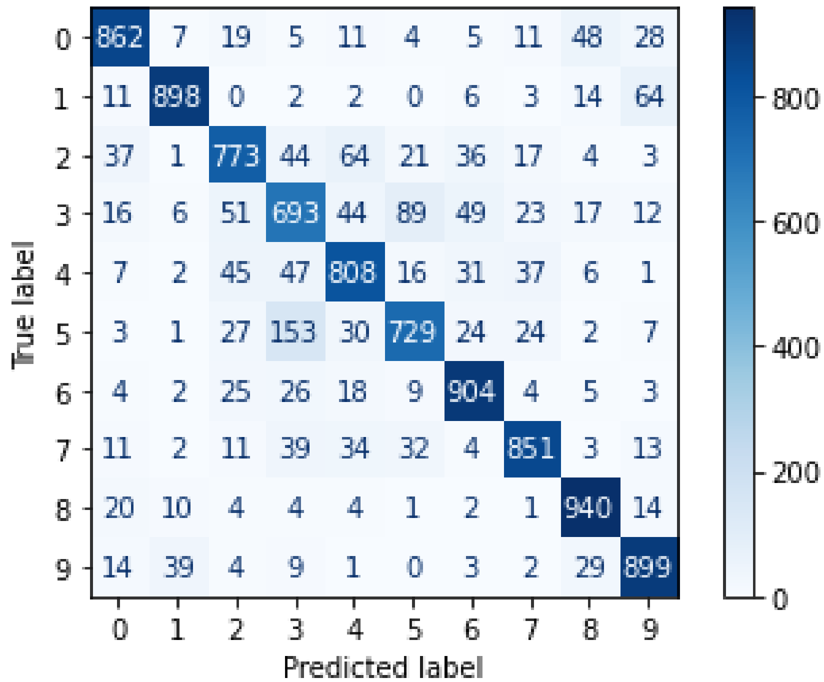
```
0.0321 - accuracy: 0.9897 - val_loss: 0.8477 - val_accuracy: 0.8354
Epoch 18/30
313/313 [==============================] - 21s 67ms/step - loss: 
0.0285 - accuracy: 0.9906 - val_loss: 0.8836 - val_accuracy: 0.8307
Epoch 19/30
313/313 [==============================] - 21s 67ms/step - loss: 
0.0234 - accuracy: 0.9930 - val_loss: 0.8641 - val_accuracy: 0.8371
Epoch 20/30
313/313 [==============================] - 21s 67ms/step - loss: 
0.0220 - accuracy: 0.9933 - val_loss: 0.8739 - val_accuracy: 0.8371
Epoch 21/30
313/313 [==============================] - 21s 68ms/step - loss: 
0.0267 - accuracy: 0.9921 - val_loss: 0.9036 - val_accuracy: 0.8263
Epoch 22/30
313/313 [==============================] - 21s 67ms/step - loss: 
0.0215 - accuracy: 0.9934 - val_loss: 0.8685 - val_accuracy: 0.8375
Epoch 23/30
313/313 [==============================] - 21s 68ms/step - loss: 
0.0181 - accuracy: 0.9942 - val_loss: 0.9097 - val_accuracy: 0.8344
Epoch 24/30
313/313 [==============================] - 21s 68ms/step - loss: 
0.0161 - accuracy: 0.9951 - val_loss: 0.9021 - val_accuracy: 0.8392
Epoch 25/30
313/313 [==============================] - 21s 67ms/step - loss: 
0.0162 - accuracy: 0.9948 - val_loss: 0.9043 - val_accuracy: 0.8387
Epoch 26/30
313/313 [==============================] - 21s 67ms/step - loss: 
0.0160 - accuracy: 0.9952 - val_loss: 0.9241 - val_accuracy: 0.8427
Epoch 27/30
313/313 [==============================] - 21s 68ms/step - loss: 
0.0183 - accuracy: 0.9944 - val_loss: 1.4835 - val_accuracy: 0.7770
Epoch 28/30
313/313 [==============================] - 21s 67ms/step - loss: 
0.0306 - accuracy: 0.9901 - val_loss: 0.8620 - val_accuracy: 0.8428
Epoch 29/30
313/313 [==============================] - 21s 68ms/step - loss: 
0.0168 - accuracy: 0.9948 - val_loss: 0.9244 - val_accuracy: 0.8355
Epoch 30/30
313/313 [==============================] - 21s 68ms/step - loss: 
0.0157 - accuracy: 0.9952 - val_loss: 0.9035 - val_accuracy: 0.8411
```

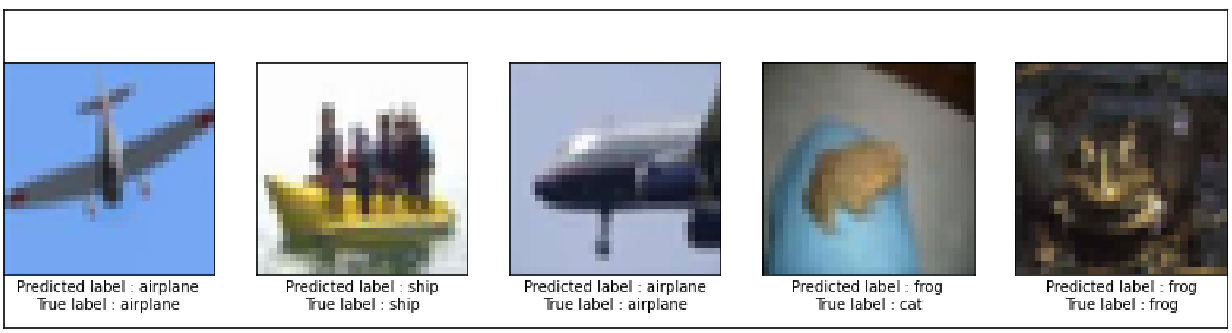
Evaluating

```python
#Evaluating DenseNet169 model
Model_evaluator(model, X_test, y_test_for_evaluate)
```

```
Accuracy: 0.8357
Precision: 0.8357
```

```
Recall: 0.8357
F1 Score: 0.8352
```






```python
#Loading Data over
(X_train, y_train, X_val, y_val, X_test, y_test) = Data_loader()
```

ConvNet (2022)

To be able to load the pretrained weights, we first upgrade our tnesorflow. Then, for the preprocessing, we use `convnet.preprocess_input` to convert the input images from RGB to BGR, and then to zero-center each color channel with respect to the ImageNet dataset, without scaling.

For the model, to stop overfitting, we add two leyers instead of three, at the end of the model. additionaly, to stop overfitting even more, we use a 0.5 drop out on the last layer.

In separate parameter tunings, we found that the best parameters are:

- Optimizer: SGD
- Number of epochs: 10
- Batch size: 64

Preprocessing

```
#Preprocessing for ConvNeXtBase
def ConvNeXtBase_preprocess(input_images):
    input_images = input_images.astype('float32')
    output_images =
tf.keras.applications.convnext.preprocess_input(input_images)
    return output_images

X_train = ConvNeXtBase_preprocess(X_train)
X_val = ConvNeXtBase_preprocess(X_val)
X_test = ConvNeXtBase_preprocess(X_test)

y_test_for_evaluate = y_test

y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
y_val = to_categorical(y_val, 10)
```

Model Definition

```
#Developing ConvNeXtBase model
def ConvNeXtBase_model():
    inputs = tf.keras.layers.Input(shape=(32,32,3))

    base_model =
tf.keras.applications.convnext.ConvNeXtBase(input_shape=(32, 32, 3),
                                                               include_top=False,
                                                               weights='imagenet')
(inputs)

    new_leyers = tf.keras.layers.GlobalAveragePooling2D()(base_model)
    new_leyers = tf.keras.layers.Flatten()(new_leyers)
    new_leyers = tf.keras.layers.Dense(512, activation="relu")
(new_leyers)
    new_leyers = tf.keras.layers.Dense(128, activation="relu")
(new_leyers)
    new_leyers = tf.keras.layers.Dropout(0.5)(new_leyers)
    new_leyers = tf.keras.layers.Dense(10, activation="softmax",
name="classification")(new_leyers)

    model = tf.keras.Model(inputs=inputs, outputs = new_leyers)
    model.compile(optimizer='SGD',
                loss='categorical_crossentropy',
                metrics = ['accuracy'])
    return model
```

```
model = ConvNeXtBase_model()
model.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/convnext/convnext_base_notop.h5
350926856/350926856 ━━━━━━━━━━━━━━━━━━━━ 3s 0us/step

Model: "functional_5"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━
━━━━━┓
┃ Layer (type)                     ┃ Output Shape           ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━
━━━━━┩
│ input_layer (InputLayer)         │ (None, 32, 32, 3)      │
0 │
├──────────────────────────────────┼────────────────────────┼───────
─────┤
│ convnext_base (Functional)       │ (None, 1, 1, 1024)     │
87,566,464 │
├──────────────────────────────────┼────────────────────────┼───────
─────┤
│ global_average_pooling2d         │ (None, 1024)           │
0 │
│ (GlobalAveragePooling2D)         │                        │
│
├──────────────────────────────────┼────────────────────────┼───────
─────┤
│ flatten (Flatten)                │ (None, 1024)           │
0 │
├──────────────────────────────────┼────────────────────────┼───────
─────┤
│ dense (Dense)                    │ (None, 512)            │
524,800 │
├──────────────────────────────────┼────────────────────────┼───────
─────┤
│ dense_1 (Dense)                  │ (None, 128)            │
65,664 │
├──────────────────────────────────┼────────────────────────┼───────
─────┤
│ dropout (Dropout)                │ (None, 128)            │
0 │
├──────────────────────────────────┼────────────────────────┼───────
─────┤
│ classification (Dense)           │ (None, 10)             │
1,290 │
```

```
└─────────────────────────────────────┴───────────────────────────┴────────────
──────┘

 Total params: 88,158,218 (336.30 MB)

 Trainable params: 88,158,218 (336.30 MB)

 Non-trainable params: 0 (0.00 B)
```

Training

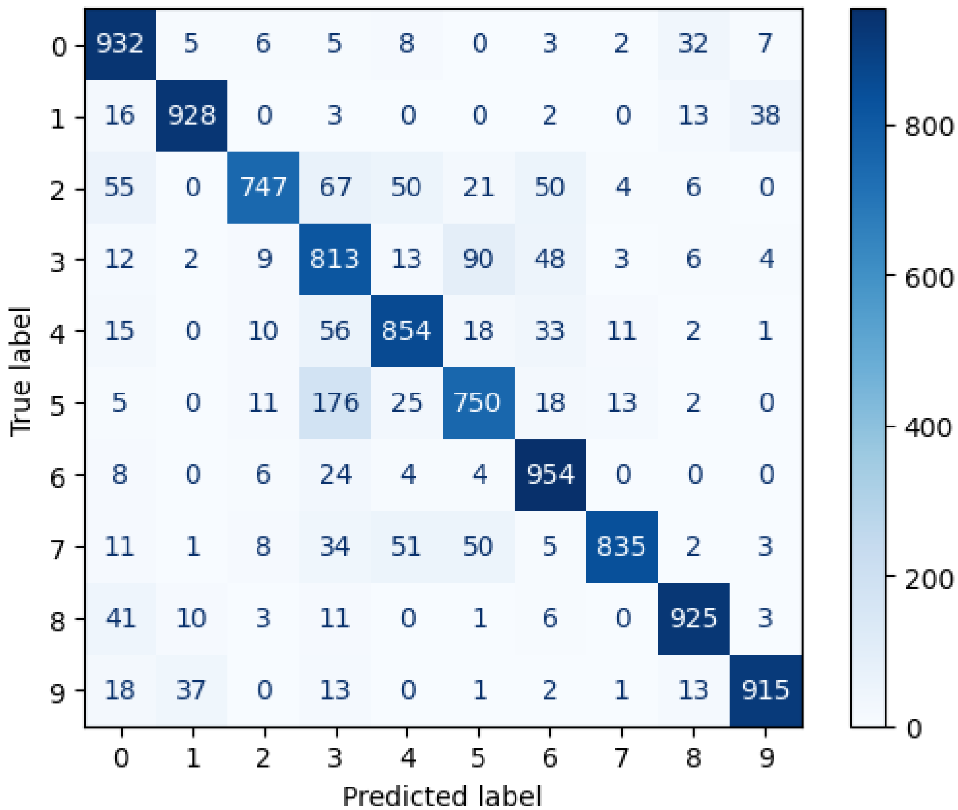
```
#Learning ConvNeXtBase model
#More complexity, less batch size
import logging
logging.getLogger('tensorflow').setLevel(logging.ERROR)

EPOCHS = 10
history = model.fit(X_train, y_train, epochs=EPOCHS, validation_data =
(X_val, y_val), batch_size=64)

Epoch 1/10
  2/625 ━━━━━━━━━━━━━━━━━━━━ 37s 61ms/step - accuracy: 0.1133 - loss:
2.9732

WARNING: All log messages before absl::InitializeLog() is called are
written to STDERR
I0000 00:00:1718141237.035139     108 device_compiler.h:186] Compiled
cluster using XLA!  This line is logged at most once for the lifetime
of the process.
W0000 00:00:1718141237.115770     108 graph_launch.cc:671] Fallback to
op-by-op mode because memset node breaks graph update
W0000 00:00:1718141237.116256     108 graph_launch.cc:671] Fallback to
op-by-op mode because memset node breaks graph update
W0000 00:00:1718141237.116834     108 graph_launch.cc:671] Fallback to
op-by-op mode because memset node breaks graph update
W0000 00:00:1718141237.117658     108 graph_launch.cc:671] Fallback to
op-by-op mode because memset node breaks graph update
W0000 00:00:1718141237.118196     108 graph_launch.cc:671] Fallback to
op-by-op mode because memset node breaks graph update
W0000 00:00:1718141237.130277     108 graph_launch.cc:671] Fallback to
op-by-op mode because memset node breaks graph update

625/625 ━━━━━━━━━━━━━━━━━━━━ 0s 52ms/step - accuracy: 0.3638 - loss:
1.8047

W0000 00:00:1718141277.300228     105 graph_launch.cc:671] Fallback to
op-by-op mode because memset node breaks graph update
W0000 00:00:1718141277.300758     105 graph_launch.cc:671] Fallback to
op-by-op mode because memset node breaks graph update
W0000 00:00:1718141277.301295     105 graph_launch.cc:671] Fallback to
op-by-op mode because memset node breaks graph update
```

```
625/625 ━━━━━━━━━━━━━━━━━━━━ 93s 76ms/step - accuracy: 0.3640 - loss: 1.8040 - val_accuracy: 0.7448 - val_loss: 0.7353
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 35s 56ms/step - accuracy: 0.7462 - loss: 0.7637 - val_accuracy: 0.8129 - val_loss: 0.5453
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 35s 56ms/step - accuracy: 0.8304 - loss: 0.5236 - val_accuracy: 0.8228 - val_loss: 0.5281
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 35s 56ms/step - accuracy: 0.8738 - loss: 0.3875 - val_accuracy: 0.8493 - val_loss: 0.4563
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 35s 56ms/step - accuracy: 0.9027 - loss: 0.2952 - val_accuracy: 0.8650 - val_loss: 0.4261
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 35s 56ms/step - accuracy: 0.9267 - loss: 0.2283 - val_accuracy: 0.8720 - val_loss: 0.4144
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 35s 56ms/step - accuracy: 0.9438 - loss: 0.1733 - val_accuracy: 0.8719 - val_loss: 0.4373
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 35s 56ms/step - accuracy: 0.9567 - loss: 0.1385 - val_accuracy: 0.8726 - val_loss: 0.4721
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 35s 56ms/step - accuracy: 0.9661 - loss: 0.1071 - val_accuracy: 0.8773 - val_loss: 0.4990
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 35s 56ms/step - accuracy: 0.9749 - loss: 0.0785 - val_accuracy: 0.8700 - val_loss: 0.5480
```

Evaluating

```
#Evaluating ConvNeXtBase model
Model_evaluator(model, X_test, y_test_for_evaluate)
```

```
 13/157 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step

W0000 00:00:1718141606.179734     105 graph_launch.cc:671] Fallback to op-by-op mode because memset node breaks graph update
W0000 00:00:1718141606.180112     105 graph_launch.cc:671] Fallback to op-by-op mode because memset node breaks graph update
W0000 00:00:1718141606.180677     105 graph_launch.cc:671] Fallback to op-by-op mode because memset node breaks graph update

157/157 ━━━━━━━━━━━━━━━━━━━━ 17s 60ms/step
Accuracy: 0.8653
Precision: 0.8722
Recall: 0.8653
F1 Score: 0.8660
```



Building and Training GoogleNet with PyTorch

We first import newly neccessary libraries, and then load out data into transformers.

After having data, we first develop Inception blocks and then we build our model based on the blocks.

For the learning, we learn for 50 epochs and we use AdamW optimizer.

Imports

```
#Import necessary libraries for googleNet model
import torch
import torch.utils.data as data
```

```
import torch.nn as nn
import torch.optim as optim
import torchvision
from torchvision.datasets import CIFAR10
from torchvision import transforms
import matplotlib.pyplot as plt
import random
import numpy as np
```

Loading Data

```
DATA_PATH = '../data/'
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
device = torch.device("cuda:0") if torch.cuda.is_available() else
torch.device("cpu")

#Loading Data
train_set = CIFAR10(root=DATA_PATH, train=True, download=True)
DATA_MEAN = (train_set.data / 255.0).mean((0,1,2))
DATA_STD = (train_set.data / 255.0).std((0,1,2))
DATA_MEAN, DATA_STD
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
../data/cifar-10-python.tar.gz
```

{"model_id":"9e6473045c5e4f2986575096756d5784","version_major":2,"vers
ion_minor":0}

```
Extracting ../data/cifar-10-python.tar.gz to ../data/
```

```
(array([0.49139968, 0.48215841, 0.44653091]),
 array([0.24703223, 0.24348513, 0.26158784]))
```

```
#Deviding the data
test_transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize(DATA_MEAN,
DATA_STD)])

train_transform = transforms.Compose([transforms.ToTensor(),
                                     transforms.Normalize(DATA_MEAN,
DATA_STD),

transforms.RandomHorizontalFlip(),

transforms.RandomResizedCrop((32,32),scale=(0.8,1.0),ratio=(0.9,1.1))]
)

train_dataset = CIFAR10(root=DATA_PATH, train=True,
transform=train_transform, download=True)
val_dataset = CIFAR10(root=DATA_PATH, train=True,
```

```
transform=test_transform, download=True)
test_set = CIFAR10(root=DATA_PATH, train=False,
transform=test_transform, download=True)

train_set, _ = data.random_split(train_dataset, [45000, 5000])
_, val_set = data.random_split(val_dataset, [45000, 5000])

batch_size = 128
train_loader = data.DataLoader(train_set, batch_size=batch_size,
shuffle=True, drop_last=True, pin_memory=True, num_workers=8)
val_loader = data.DataLoader(val_set, batch_size=batch_size,
shuffle=True, drop_last=False, num_workers=8)
test_loader = data.DataLoader(test_set, batch_size=batch_size,
shuffle=True, drop_last=False, num_workers=8)
```

```
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
```

Defining Inception blocks

```
#Developing Inception blocks
class InceptionBlock(nn.Module):
    def __init__(self, c_in, c_red: dict, c_out: dict):
        super().__init__()

        # 1x1 branch
        self.conv_1x1 = nn.Sequential(nn.Conv2d(c_in, c_out["1x1"],
kernel_size=1),
                                     nn.BatchNorm2d(c_out["1x1"]),
                                     nn.ReLU())

        # 3x3 branch, we padding 1 in the 3x3 convolution layer to
keep same size of image
        self.conv_3x3 = nn.Sequential(nn.Conv2d(c_in, c_red["3x3"],
kernel_size=1),
                                     nn.BatchNorm2d(c_red["3x3"]),
                                     nn.ReLU(),
                                     nn.Conv2d(c_red["3x3"],
c_out["3x3"], kernel_size=3, padding=1),
                                     nn.BatchNorm2d(c_out["3x3"]),
                                     nn.ReLU())

        # 5x5 branch, we padding 2 in the 5x5 convolution layer to
keep same size of image
        self.conv_5x5 = nn.Sequential(nn.Conv2d(c_in, c_red["5x5"],
kernel_size=1),
                                     nn.BatchNorm2d(c_red["5x5"]),
                                     nn.ReLU(),
```

```
                                     nn.Conv2d(c_red["5x5"],
c_out["5x5"], kernel_size=5, padding=2),
                                     nn.BatchNorm2d(c_out["5x5"]),
                                     nn.ReLU())
        # Max pooling branch
        self.max_pool = nn.Sequential(nn.MaxPool2d(kernel_size=3,
padding=1, stride=1),
                                     nn.Conv2d(c_in, c_out["max"],
kernel_size=1),
                                     nn.BatchNorm2d(c_out["max"]),
                                     nn.ReLU())

    def forward(self, x):
        x_1x1 = self.conv_1x1(x)
        x_3x3 = self.conv_3x3(x)
        x_5x5 = self.conv_5x5(x)
        x_max = self.max_pool(x)
        output = torch.cat([x_1x1, x_3x3, x_5x5, x_max], dim=1)
        return output
```

Model Definition

```
#Developing main GoogleNet model
class GoogleNet(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()
        self.input = nn.Sequential(nn.Conv2d(3, 64, kernel_size=3,
padding=1),
                                  nn.BatchNorm2d(64),
                                  nn.ReLU())

        # Stacking inception blocks
        self.inception = nn.Sequential(
            InceptionBlock(64, c_red={"3x3": 32, "5x5": 16},
c_out={"1x1": 16, "3x3": 32, "5x5": 8, "max": 8}),
            InceptionBlock(64, c_red={"3x3": 32, "5x5": 16},
c_out={"1x1": 24, "3x3": 48, "5x5": 12, "max": 12}),
            nn.MaxPool2d(3, stride=2, padding=1),  # 32x32 => 16x16
            InceptionBlock(96, c_red={"3x3": 32, "5x5": 16},
c_out={"1x1": 24, "3x3": 48, "5x5": 12, "max": 12}),
            InceptionBlock(96, c_red={"3x3": 32, "5x5": 16},
c_out={"1x1": 16, "3x3": 48, "5x5": 16, "max": 16}),
            InceptionBlock(96, c_red={"3x3": 32, "5x5": 16},
c_out={"1x1": 16, "3x3": 48, "5x5": 16, "max": 16}),
            InceptionBlock(96, c_red={"3x3": 32, "5x5": 16},
c_out={"1x1": 32, "3x3": 48, "5x5": 24, "max": 24}),
            nn.MaxPool2d(3, stride=2, padding=1),  # 16x16 => 8x8
            InceptionBlock(128, c_red={"3x3": 48, "5x5": 16},
c_out={"1x1": 32, "3x3": 64, "5x5": 16, "max": 16}),
```

```
            InceptionBlock(128, c_red={"3x3": 48, "5x5": 16},
c_out={"1x1": 32, "3x3": 64, "5x5": 16, "max": 16})
        )

        self.output = nn.Sequential(
            nn.AdaptiveAvgPool2d((1, 1)),
            nn.Flatten(),
            nn.Linear(128, num_classes)
        )

    def forward(self, x):
        x = self.input(x)
        x = self.inception(x)
        x = self.output(x)
        return x
googleNet = GoogleNet()
```

Training

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(googleNet.parameters(),
                        lr=1e-3,
                        weight_decay=1e-4)
scheduler = optim.lr_scheduler.MultiStepLR(optimizer,
                                           milestones=[100, 150],
                                           gamma=0.1)

n_epochs = 50
for epoch in range(n_epochs + 1):
    running_loss = 0.0
    total = 0
    correct = 0
    for data in train_loader:
        inputs, labels = data[0].to(device), data[1].to(device)
        optimizer.zero_grad()
        outputs = googleNet(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    scheduler.step()
    with torch.no_grad():
        for data in val_loader:
            inputs, labels = data[0].to(device), data[1].to(device)
            outputs = googleNet(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
```

```
    print(f"Epoch: {epoch +1:3d}, loss: {running_loss/len(train_set)},
val_acc: {100 * correct / total:.2f}%")
    running_loss = 0.0

print("Training completed!")
```

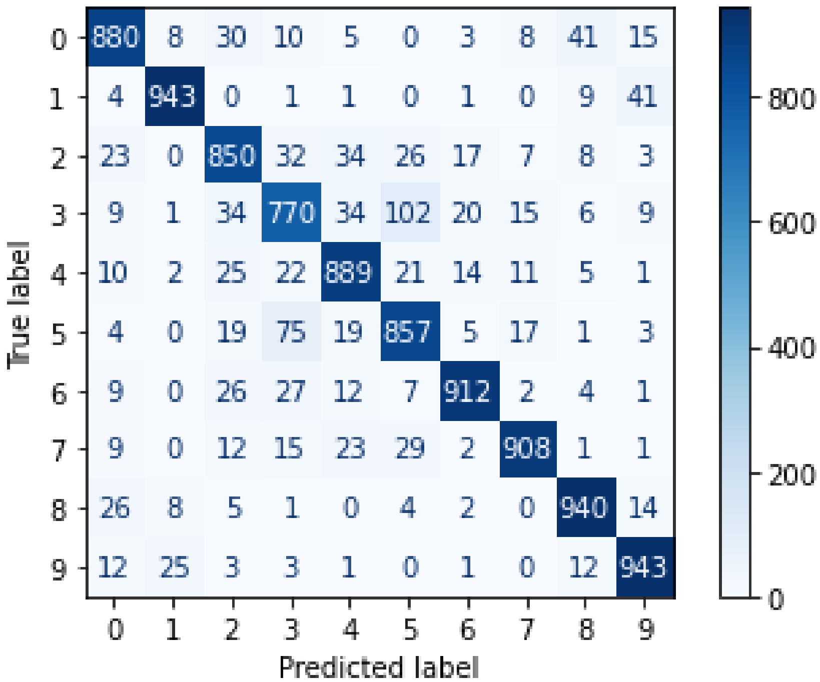
```
Epoch:   1, loss: 0.010205939573711818, val_acc: 65.76%
Epoch:   2, loss: 0.006712542927265168, val_acc: 74.28%
Epoch:   3, loss: 0.005397243330213759, val_acc: 77.98%
Epoch:   4, loss: 0.004663274614678489, val_acc: 80.02%
Epoch:   5, loss: 0.00415978581044409, val_acc: 81.12%
Epoch:   6, loss: 0.0037832225720087686, val_acc: 83.32%
Epoch:   7, loss: 0.003558736488554213, val_acc: 83.58%
Epoch:   8, loss: 0.0033155616339710023, val_acc: 84.68%
Epoch:   9, loss: 0.003115383454495006, val_acc: 85.10%
Epoch:  10, loss: 0.002932573574119144, val_acc: 85.08%
Epoch:  11, loss: 0.0027935502310593922, val_acc: 85.58%
Epoch:  12, loss: 0.0026520036111275353, val_acc: 85.76%
Epoch:  13, loss: 0.002527625013391177, val_acc: 85.92%
Epoch:  14, loss: 0.0024264772570795484, val_acc: 86.50%
Epoch:  15, loss: 0.002335746763149897, val_acc: 86.98%
Epoch:  16, loss: 0.0022226002261042594, val_acc: 86.72%
Epoch:  17, loss: 0.002136235643261009, val_acc: 87.36%
Epoch:  18, loss: 0.002034467871652709, val_acc: 87.50%
Epoch:  19, loss: 0.00200205281343725, val_acc: 87.74%
Epoch:  20, loss: 0.0019088405574361484, val_acc: 86.98%
Epoch:  21, loss: 0.0018563557712568178, val_acc: 87.78%
Epoch:  22, loss: 0.0017727391731407907, val_acc: 87.66%
Epoch:  23, loss: 0.0017328001477652127, val_acc: 87.92%
Epoch:  24, loss: 0.0016686665938960182, val_acc: 88.20%
Epoch:  25, loss: 0.00166120476474364, val_acc: 87.68%
Epoch:  26, loss: 0.0015323940641350216, val_acc: 87.56%
Epoch:  27, loss: 0.0015347886610362264, val_acc: 87.40%
Epoch:  28, loss: 0.0014850063779287869, val_acc: 87.44%
Epoch:  29, loss: 0.0014174489960902268, val_acc: 88.84%
Epoch:  30, loss: 0.0013800597647825876, val_acc: 88.40%
Epoch:  31, loss: 0.0013898838123513593, val_acc: 88.22%
Epoch:  32, loss: 0.0013419655772546927, val_acc: 88.04%
Epoch:  33, loss: 0.0013005609800418219, val_acc: 88.22%
Epoch:  34, loss: 0.0012821315591533978, val_acc: 88.52%
Epoch:  35, loss: 0.00122705048173666, val_acc: 88.80%
Epoch:  36, loss: 0.001229011454515987, val_acc: 88.90%
Epoch:  37, loss: 0.0011816696477433045, val_acc: 88.34%
Epoch:  38, loss: 0.0011252132759326035, val_acc: 89.00%
Epoch:  39, loss: 0.0011208142115010156, val_acc: 89.04%
Epoch:  40, loss: 0.001115599280430211, val_acc: 88.74%
Epoch:  41, loss: 0.0010670398283335897, val_acc: 88.72%
Epoch:  42, loss: 0.0010556180910517771, val_acc: 88.26%
Epoch:  43, loss: 0.0010065211152036984, val_acc: 89.16%
```

```
Epoch:  44, loss: 0.0010189626633293099, val_acc: 89.02%
Epoch:  45, loss: 0.0009561569395992491, val_acc: 89.44%
Epoch:  46, loss: 0.0009231896334224277, val_acc: 89.38%
Epoch:  47, loss: 0.0009578435990545485, val_acc: 89.18%
Epoch:  48, loss: 0.0009339408687419361, val_acc: 88.44%
Epoch:  49, loss: 0.0009173323584099611, val_acc: 88.60%
Epoch:  50, loss: 0.0008924635922743214, val_acc: 89.04%
Epoch:  51, loss: 0.0008746992973403798, val_acc: 89.48%
Training completed!
```

Evaluation

```
#Testing the model on test set
correct = 0
total = 0
y_pred = []
y_test_temp = []
X_test_temp = []

with torch.no_grad():
    for data in test_loader:
        inputs, labels = data[0].to(device), data[1].to(device)
        outputs = googleNet(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
        y_pred.extend(predicted.cpu().numpy())
        y_test_temp.extend(labels.cpu().numpy())
        X_test_temp.extend(inputs.cpu().numpy())


accuracy = accuracy_score(y_test_temp, y_pred)
precision = precision_score(y_test_temp, y_pred, average='weighted')
recall = recall_score(y_test_temp, y_pred, average='weighted')
f1 = f1_score(y_test_temp, y_pred, average='weighted')

print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1 Score: {f1:.4f}")

cm = confusion_matrix(y_test_temp, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm)
disp.plot(cmap=plt.cm.Blues)
plt.show()
```

```
Accuracy: 0.8892
Precision: 0.8894
Recall: 0.8892
F1 Score: 0.8891
```



True label
Predicted label