ITI

(COMPUTER VISION)

Project: Image classification using cifar-10 dataset.

The dataset is comprised of 60,000 32×32-pixel color photographs of objects from 10 classes, such as frogs, birds, cats, ships, etc. The class labels and their standard associated integer values are listed below.

- 0: airplane
- 1: automobile
- 2: bird
- 3: cat
- 4: deer
- 5: dog
- 6: frog
- 7: horse
- 8: ship
- 9: truck

First, I import libraries that we need.

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import tensorflow as tf
import tensorflow.keras as K
from tensorflow.keras.layers import Input, Conv2D, Dense, Flatten, Dropout
from tensorflow.keras.layers import GlobalMaxPooling2D, MaxPooling2D,UpSampling2D
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.models import Model
```

Next step I load the data from tensorflow library and show shape (train, test) from it.

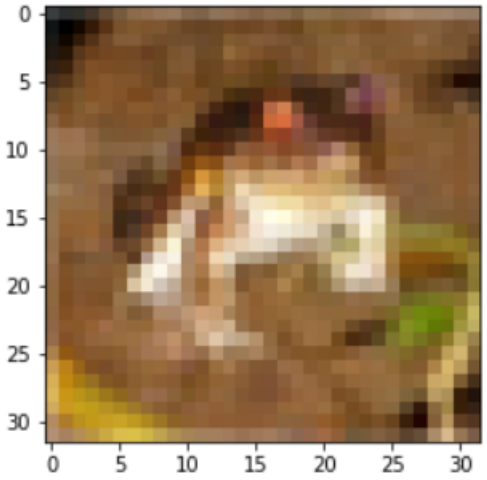
```
[ ]  # Load in the data
     cifar10 = tf.keras.datasets.cifar10

     # Distribute it to train and test set
     (x_train, y_train), (x_test, y_test) = cifar10.load_data()
     print(x_train.shape, y_train.shape, x_test.shape, y_test.shape)
```

```
(50000, 32, 32, 3) (50000, 1) (10000, 32, 32, 3) (10000, 1)
```

After I load data, I show image in data using matplotlib.

```
plt.imshow(x_train[0])
```

```
<matplotlib.image.AxesImage at 0x7fcc65ee2750>
```



I know that the pixel values for each image in the dataset are unsigned integers in the range between no color and full color, or 0 and 255.

We do not know the best way to scale the pixel values for modeling, but we know that some scaling will be required.

A good starting point is to normalize the pixel values, rescale them to the range [0,1]. This involves first converting the data type from unsigned integers to floats, then dividing the pixel values by the maximum value.

```
[ ]  x_train = x_train.astype('float32')
     x_test = x_test.astype('float32')
     x_train = x_train / 255.0
     x_test = x_test / 255.0
```

Now, let's go to doing dummy to the output.

```
[ ]  #X_train = K.applications.vgg16.preprocess_input(x_train)
     Y_train= K.utils.to_categorical(y_train, 10)
```

```
[ ]  #X_test= K.applications.vgg16.preprocess_input(x_test)
     Y_test= K.utils.to_categorical(y_test, 10)
```

It's time to create my model.

After I doing many models to a achieve a good accuracy, I found that transfer learning is a good model and not waste time.

I use vgg16 it's given me a good accuracy. I add flatten to convert the convolution part of CNN into 1D feature vector.

And we adding Dropout to reduce the overfitting.

I add batch normalization in an effort to stabilize the learning and perhaps accelerate the learning process. To offset this acceleration

```
cnn = K.applications.vgg16.VGG16(include_top=False,
                                 weights='imagenet',
                                 pooling='avg',
                                 input_shape=(32, 32, 3)
                                 )
```

```
[ ]  CNN= K.Sequential()
     CNN.add(cnn)
     CNN.add(Flatten())
     CNN.add(Dense(512, activation=('relu')))
     #CNN.add(Dropout(0.5))
     #CNN.add(Dense(256, activation=('relu')))
     CNN.add(BatchNormalization())
     CNN.add(Dropout(0.5))
     CNN.add(Dense(10, activation=('sigmoid')))
```

Now, I compile the model

The model will be optimized using stochastic gradient descent.

We will use a modest learning rate of 0.001 and a large momentum of 0.9, both of which are good general starting points. The model will optimize the categorical cross entropy loss function required for multi-class classification and will monitor classification accuracy.

```
[ ] from tensorflow.keras import optimizers

    opt = optimizers.SGD(lr=0.001, momentum=0.9)
    CNN.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
```

```
/usr/local/lib/python3.7/dist-packages/keras/optimizer_v2/optimizer_v2.py:356: UserWarning: The `lr` argument is deprecated, use `learning_rate` instead.
  "The `lr` argument is deprecated, use `learning_rate` instead.")
```

Now fitting the model with 15 epochs.

```
his = CNN.fit(x=x_train, y=Y_train,
          batch_size=64,
          validation_data=(x_test, Y_test),
          epochs=15, shuffle=True,
          verbose=1
          )
```

```
Epoch 1/15
782/782 [==============================] - 69s 84ms/step - loss: 0.9414 - accuracy: 0.6809 - val_loss: 0.7156 - val_accuracy: 0.7567
Epoch 2/15
782/782 [==============================] - 64s 82ms/step - loss: 0.5565 - accuracy: 0.8092 - val_loss: 0.6781 - val_accuracy: 0.7759
Epoch 3/15
782/782 [==============================] - 65s 83ms/step - loss: 0.4220 - accuracy: 0.8547 - val_loss: 0.7584 - val_accuracy: 0.7668
Epoch 4/15
782/782 [==============================] - 64s 82ms/step - loss: 0.3262 - accuracy: 0.8884 - val_loss: 0.5468 - val_accuracy: 0.8234
Epoch 5/15
782/782 [==============================] - 64s 82ms/step - loss: 0.2531 - accuracy: 0.9123 - val_loss: 0.5463 - val_accuracy: 0.8306
Epoch 6/15
782/782 [==============================] - 64s 82ms/step - loss: 0.1831 - accuracy: 0.9367 - val_loss: 0.5644 - val_accuracy: 0.8322
Epoch 7/15
782/782 [==============================] - 64s 82ms/step - loss: 0.1232 - accuracy: 0.9593 - val_loss: 0.7244 - val_accuracy: 0.8018
Epoch 8/15
782/782 [==============================] - 64s 82ms/step - loss: 0.0847 - accuracy: 0.9734 - val_loss: 0.5944 - val_accuracy: 0.8447
Epoch 9/15
782/782 [==============================] - 64s 82ms/step - loss: 0.0543 - accuracy: 0.9835 - val_loss: 0.5567 - val_accuracy: 0.8547
Epoch 10/15
782/782 [==============================] - 64s 82ms/step - loss: 0.0346 - accuracy: 0.9901 - val_loss: 0.5500 - val_accuracy: 0.8645
Epoch 11/15
782/782 [==============================] - 65s 83ms/step - loss: 0.0175 - accuracy: 0.9962 - val_loss: 0.6019 - val_accuracy: 0.8599
Epoch 12/15
782/782 [==============================] - 65s 83ms/step - loss: 0.0152 - accuracy: 0.9962 - val_loss: 0.6047 - val_accuracy: 0.8608
Epoch 13/15
782/782 [==============================] - 65s 83ms/step - loss: 0.0065 - accuracy: 0.9992 - val_loss: 0.6245 - val_accuracy: 0.8615
Epoch 14/15
782/782 [==============================] - 65s 83ms/step - loss: 0.0024 - accuracy: 1.0000 - val_loss: 0.5660 - val_accuracy: 0.8739
Epoch 15/15
782/782 [==============================] - 65s 83ms/step - loss: 0.0021 - accuracy: 0.9998 - val_loss: 0.5679 - val_accuracy: 0.8744
```

Now evaluate the mode .

```
CNN.evaluate(x_test,Y_test)
```

```
313/313 [==============================] - 7s 20ms/step - loss: 0.5679 - accuracy: 0.8744
[0.5679095387458801, 0.8744000196456909]
```

The accuracy is 87% is good.

Plot the trainning loss and trainning accuracy.

```
plt.subplot(211)
plt.plot(his.history['loss'],color = 'blue',label = 'train')
plt.plot(his.history['val_loss'],color = 'red',label = 'test')
plt.title('Cross Entropy loss')
plt.xlabel('Epoch number')
plt.ylabel('loss')

plt.subplot(212)
plt.plot(his.history['accuracy'],color = 'blue',label = 'train')
plt.plot(his.history['val_accuracy'],color = 'red',label = 'test')
plt.title('Classification Accuracy')
plt.xlabel('Epoch number')
plt.ylabel('Accuracy')
```

```
Text(0, 0.5, 'Accuracy')
```