In [1]:
```
from google.colab import drive
drive.mount('/gdrive')
%cd /gdrive
```

```
Mounted at /gdrive
/gdrive
```

In [2]:
```
import keras
from keras.datasets import cifar10
from keras.models import Model, Sequential
from keras.layers import Dense, Dropout, Flatten, Input, AveragePooling2D, merge, Activation
from keras.layers import Conv2D, MaxPooling2D, BatchNormalization
from keras.layers import Concatenate
from keras.optimizers import Adam
from tensorflow.keras import models, layers
from tensorflow.keras.models import Model
from tensorflow.keras.layers import BatchNormalization, Activation, Flatten
from tensorflow.keras.optimizers import Adam
from keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.callbacks import LearningRateScheduler, CSVLogger
from tensorflow.keras.callbacks import ReduceLROnPlateau
from tensorflow.keras.callbacks import TensorBoard
import datetime
```

CNN Clearly Explained

https://machinelearningmastery.com/how-to-develop-a-cnn-from-scratch-for-cifar-10-photo-classification/#:~:text=CIFAR%2D10
he%20field%20of%20machine%20learning.&text=Top%20performance%20on%20the%20problem,90%25%20on%20the%20test%20dataset.

Information about **CIFAR data** set

The dataset is comprised of 60,000 32×32 pixel color photographs of objects from 10 classes, such as frogs, birds, cats, s
ss 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

In [3]:
```
# this part will prevent tensorflow to allocate all the avaliable GPU Memory
# backend
import tensorflow as tf
```

In [4]:
```
# Hyperparameters
batch_size = 64
num_classes = 10
epochs = 200
l = 6
num_filter = 35
compression = 1.0
dropout_rate = 0.2
```

Loading Dataset

We know some things about the dataset.

For example, we know that the images are all pre-segmented (e.g. each image contains a single object), that the images all
are size of 32×32 pixels, and that the images are color. Therefore, we can load the images and use them for modeling almos

In [5]:
```
# Load CIFAR10 Data
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.cifar10.load_data()
img_height, img_width, channel = X_train.shape[1],X_train.shape[2],X_train.shape[3]

# convert to one hot encoing
'''
use a one hot encoding for the class element of each sample,
transforming the integer into a 10 element binary vector with a 1 for the index of the class value.
We can achieve this with the to_categorical() utility function.
'''
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz (https://www.cs.toronto.edu/~kriz/cifar-10-pytho
170500096/170498071 [==============================] - 11s 0us/step
```

Pixel Data point - All image may not be in same resolution best way to handle is pixel data point

We know that the pixel values for each image in the dataset are unsigned integers in the range between no color and full c
5.

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, e.g. rescale them to the range [0,1]. This involves first converti
rom unsigned integers to floats, then dividing the pixel values by the maximum value.

In [6]:
```
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

# normalize to range 0-1
X_train = X_train / 255.0
X_test = X_test / 255.0
```

In [7]:
```
type(X_train)
```

Out[7]:
```
numpy.ndarray
```

In [8]:
```
X_train.shape
```

Out[8]:
```
(50000, 32, 32, 3)
```

In [9]:
```
len(X_train)
```

Out[9]:
```
50000
```

In [10]:
```
X_test.shape
```

Out[10]:
```
(10000, 32, 32, 3)
```

Data Augmentation

still now we seen. Data Augmentation mostly used for imbalanced dataset to make it balanced. Here all categorical are equa ll we use Data Augumentation to represent the image in more generalizsed manner,

more over in this dataset are 32*32 means image is slow, This low resolution is likely the cause of the limited performanc line algorithms are able to achieve on the dataset.

In [11]:

```
datagen_x_train = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.125,
    height_shift_range=0.125,
    horizontal_flip=True,
    fill_mode='nearest',
    zoom_range=0.10
                              )

# prepare iterator
Data_aug_x_train = datagen_x_train.flow(X_train, y_train, batch_size=64)
```

In [12]:
```
it = datagen_x_train.flow(X_train)
for i in range(9):
    # define subplot
    plt.subplot(330 + 1 + i)
    # generate batch of images
    batch = it.next()
    # convert to unsigned integers for viewing
    image = batch[0].astype('uint8')
    # plot raw pixel data
    plt.imshow(image)
# show the figure
plt.show()
```

What is Dense layer

In DenseNet, each layer obtains additional inputs from all preceding layers and passes on its own feature-maps to all subs
catenation is used. Each layer is receiving a “collective knowledge” from all preceding layers.

see below diagram , all precceding layers are passing to next layer
Youtube video - https://www.youtube.com/watch?v=oV4YBitzXKw

Dense layer is split in to three parts

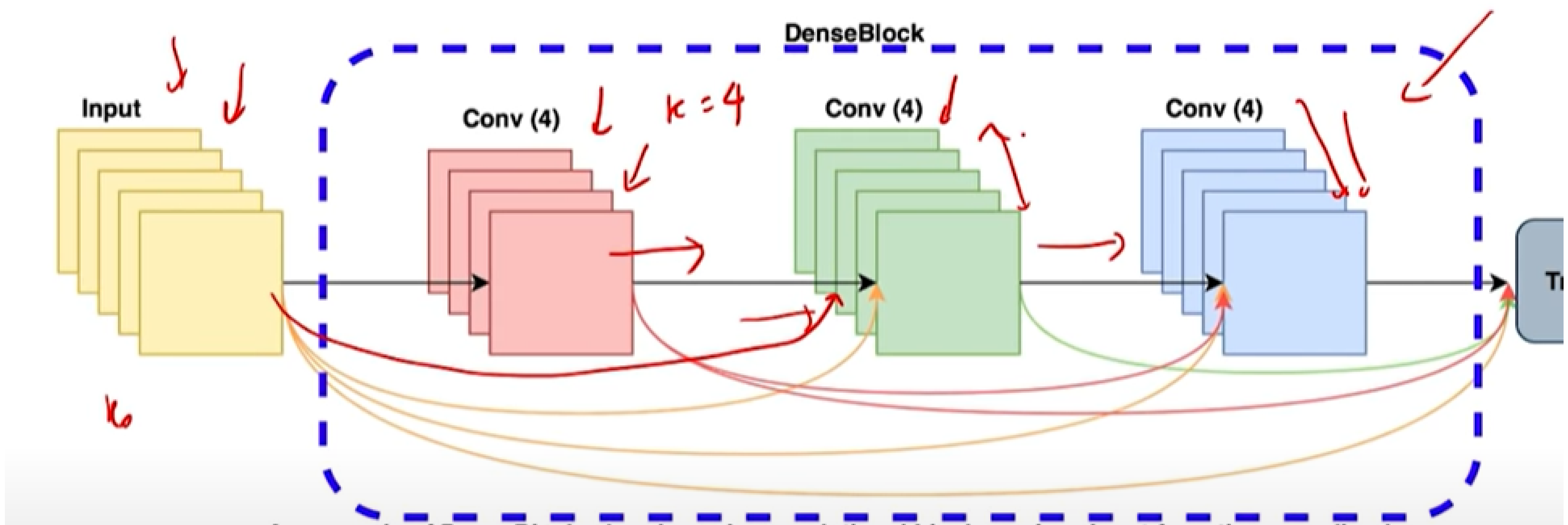
1. **DenseBlock** - All preceeding input passed to next layer, suppose we have 100 layers then memory expolation will happen
oing to use fixed length of feature learned from previous layer
2. **Transistion** - used to reduce the dimension coming from the dense block(Max pooling concept will take place)
3. **Output layer** - Average max pooling

Advantages
1. Won't Over fit easy
2. Each layer receives inputs from all the preceding layers and passes its own information to all subsequent layers, which
nal output layer has direct information from every single layer including the very first layer. This right here is suppose
roblem of redundant layers.

Refer - https://towardsdatascience.com/review-densenet-image-classification-b6631a8ef803#:~:text=DenseNet%2DBC%20(Further%
oss%20transition%20layers%20remains%20unchanged

In [3]:
```
from IPython.display import Image
Image(filename=r"C:\Users\User\Desktop\Applied_A\Program\DEEP_Learning\Callback\Callbacks\Densenet_architecture.png")
```

Out[3]:



An example of DenseBlock wherein each convolutional block receives input from the preceding layers. The memory explosion is circumvented by setting the number of feature map learnt in convolutional operation to a small value (k=4).

In [13]:
```
# Dense Block
def denseblock(input, num_filter = 12, dropout_rate = 0.2):
    global compression
    temp = input
    for _ in range(l):
        BatchNorm = layers.BatchNormalization()(temp)
        relu = layers.Activation('relu')(BatchNorm)
        Conv2D_3_3 = layers.Conv2D(int(num_filter*compression), (3,3), use_bias=False ,padding='same')(relu)
        if dropout_rate>0:
            Conv2D_3_3 = layers.Dropout(dropout_rate)(Conv2D_3_3)
        concat = layers.Concatenate(axis=-1)([temp,Conv2D_3_3])

        temp = concat

    return temp

## transition Blosck
def transition(input, num_filter = 12, dropout_rate = 0.2):
    global compression
    BatchNorm = layers.BatchNormalization()(input)
    relu = layers.Activation('relu')(BatchNorm)
    Conv2D_BottleNeck = layers.Conv2D(int(num_filter*compression), (1,1), use_bias=False ,padding='same')(relu)
    if dropout_rate>0:
         Conv2D_BottleNeck = layers.Dropout(dropout_rate)(Conv2D_BottleNeck)
    avg = layers.AveragePooling2D(pool_size=(2,2))(Conv2D_BottleNeck)
    return avg

#output layer
def output_layer(input):
    global compression
    BatchNorm = layers.BatchNormalization()(input)
    relu = layers.Activation('relu')(BatchNorm)
    AvgPooling = layers.AveragePooling2D(pool_size=(2,2))(relu)
    flat = layers.Flatten()(AvgPooling)
    output = layers.Dense(num_classes, activation='softmax')(flat)
    return output
```

In [14]:

```
input = layers.Input(shape=(img_height, img_width, channel,))
First_Conv2D = layers.Conv2D(num_filter, (3,3), use_bias=False ,padding='same')(input)

First_Block = denseblock(First_Conv2D, num_filter, dropout_rate)
First_Transition = transition(First_Block, num_filter, dropout_rate)

Second_Block = denseblock(First_Transition, num_filter, dropout_rate)
Second_Transition = transition(Second_Block, num_filter, dropout_rate)

Third_Block = denseblock(Second_Transition, num_filter, dropout_rate)
Third_Transition = transition(Third_Block, num_filter, dropout_rate)

Last_Block = denseblock(Third_Transition,  num_filter, dropout_rate)
output = output_layer(Last_Block)
```

In [15]:

```
#https://arxiv.org/pdf/1608.06993.pdf
from IPython.display import IFrame, YouTubeVideo
YouTubeVideo(id='-W6y8xnd--U', width=600)
```

Out[15]:

In [16]:
```
model = Model(inputs=[input], outputs=[output])
model.summary()
```

```
________________________________________________________________________________
batch_normalization_10 (BatchNo (None, 16, 16, 140)  560         concatenate_8[0][0]
________________________________________________________________________________
activation_10 (Activation)      (None, 16, 16, 140)  0           batch_normalization_10[0][0]
________________________________________________________________________________
conv2d_11 (Conv2D)              (None, 16, 16, 35)   44100       activation_10[0][0]
________________________________________________________________________________
dropout_10 (Dropout)            (None, 16, 16, 35)   0           conv2d_11[0][0]
________________________________________________________________________________
concatenate_9 (Concatenate)     (None, 16, 16, 175)  0           concatenate_8[0][0]
                                                                 dropout_10[0][0]
________________________________________________________________________________
batch_normalization_11 (BatchNo (None, 16, 16, 175)  700         concatenate_9[0][0]
________________________________________________________________________________
activation_11 (Activation)      (None, 16, 16, 175)  0           batch_normalization_11[0][0]
________________________________________________________________________________
conv2d_12 (Conv2D)              (None, 16, 16, 35)   55125       activation_11[0][0]
________________________________________________________________________________
dropout_11 (Dropout)            (None, 16, 16, 35)   0           conv2d_12[0][0]
```

In [17]:
```
print(len(model.layers))
```

```
142
```

In [18]:
```
# Early Stopping - after val_accuracy reached 90
# refer - https://towardsdatascience.com/neural-network-with-tensorflow-how-to-stop-training-using-callback-5c8d575c18a9
ACCURACY_THRESHOLD = 0.90
class myCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs={}):

      if(logs.get('val_accuracy') > ACCURACY_THRESHOLD):
        print("\nReached %2.2f%% accuracy, so stopping training!!" %(ACCURACY_THRESHOLD*100))
        self.model.stop_training = True

early_stop = myCallback()
```

In [19]:
```
def decay_fn(epoch, lr):
    if epoch < 50:
        print('Learning rate 0.001')
        return 0.001
    elif epoch >= 50 and epoch < 75:
        print('Learning rate 0.0001')
        return 0.0001
    else:
        print('Learning rate 0.00001')
        return 0.00001

lr_scheduler = LearningRateScheduler(decay_fn)
```

In [20]:
```

csv_logger = CSVLogger('/gdrive/My Drive/CNN_CIFR/CSVlogs/trainings_new.log')


filepath="/gdrive/My Drive/CNN_CIFR/new_model_save_test/best_model-{epoch:02d}.h5"
checkpoint = ModelCheckpoint(filepath=filepath, monitor='val_accuracy',  verbose=1, save_best_only=True, mode='auto')
```

In [21]:
```
# determine Loss function and Optimizer
opt = keras.optimizers.Adam()
model.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])
```

In [22]:
```
(len(X_train) / batch_size)
```

Out[22]:
```
781.25
```

In [23]:
```
model.fit(Data_aug_x_train,
          steps_per_epoch=(len(X_train) / batch_size),
                    epochs=100,
                    verbose=1,
                    validation_data=(X_test, y_test),
          callbacks = [csv_logger, early_stop,checkpoint,lr_scheduler])
```

```
Learning rate 0.001
Epoch 49/100
782/781 [==============================] - ETA: 0s - loss: 0.2753 - accuracy: 0.9038
Epoch 00049: val_accuracy did not improve from 0.88560
782/781 [==============================] - 154s 197ms/step - loss: 0.2753 - accuracy: 0.9038 - val_loss: 0.4135 - val_accuracy
Learning rate 0.001
Epoch 50/100
782/781 [==============================] - ETA: 0s - loss: 0.2724 - accuracy: 0.9047
Epoch 00050: val_accuracy did not improve from 0.88560
782/781 [==============================] - 154s 197ms/step - loss: 0.2724 - accuracy: 0.9047 - val_loss: 0.4954 - val_accuracy
Learning rate 0.0001

Epoch 51/100
782/781 [==============================] - ETA: 0s - loss: 0.2274 - accuracy: 0.9205
Reached 90.00% accuracy, so stopping training!!

Epoch 00051: val_accuracy improved from 0.88560 to 0.90930, saving model to /gdrive/My Drive/CNN_CIFR/new_model_save_test/best
782/781 [==============================] - 154s 197ms/step - loss: 0.2274 - accuracy: 0.9205 - val_loss: 0.3022 - val_accuracy
```

Out[23]:
```
<tensorflow.python.keras.callbacks.History at 0x7faebd0a7a58>
```

CSV logger

In [1]:
```
import pandas as pd

train_log = pd.read_csv(r"C:\Users\User\Desktop\Applied_A\Assignment_All_in_one\DeepLearning\Dense_net_CIFAR\trainings_new.txt

train_log
```

Out[1]:

	epoch	accuracy	loss	val_accuracy	val_loss
0	0	0.44198	1.527068	0.4213	2.045505
1	1	0.58738	1.144373	0.5288	1.706873
2	2	0.65320	0.980196	0.6163	1.322578
3	3	0.69602	0.864334	0.6317	1.192576
4	4	0.72106	0.791180	0.6948	0.988763
5	5	0.74304	0.729754	0.6934	1.011601
6	6	0.76164	0.687559	0.6954	1.059975
7	7	0.77492	0.648694	0.7605	0.818448
8	8	0.78610	0.615635	0.7620	0.809635
9	9	0.79324	0.591438	0.7516	0.873774
10	10	0.80068	0.570036	0.7594	0.853101
11	11	0.81154	0.542579	0.7777	0.774972
12	12	0.81614	0.529099	0.8126	0.597148
13	13	0.82298	0.512623	0.8326	0.526444
14	14	0.82890	0.496443	0.8303	0.539239
15	15	0.83258	0.480106	0.7971	0.725843
16	16	0.84024	0.464075	0.8211	0.594409
17	17	0.84514	0.448892	0.8189	0.619969
18	18	0.84622	0.443715	0.8143	0.590964
19	19	0.85120	0.432289	0.8174	0.630162
20	20	0.85372	0.419298	0.8516	0.489171
21	21	0.85966	0.407654	0.8599	0.450736

	epoch	accuracy	loss	val_accuracy	val_loss
22	22	0.86044	0.399597	0.8604	0.470860
23	23	0.86218	0.396935	0.7562	0.940724
24	24	0.86614	0.382996	0.8466	0.509866
25	25	0.87002	0.375450	0.8299	0.591100
26	26	0.87096	0.370497	0.7843	0.857648
27	27	0.87254	0.364019	0.8416	0.530918
28	28	0.87640	0.355581	0.8401	0.563610
29	29	0.87880	0.351894	0.8389	0.574400
30	30	0.88010	0.342952	0.8701	0.447049
31	31	0.88186	0.341771	0.8484	0.523645
32	32	0.88488	0.333263	0.8578	0.502421
33	33	0.88738	0.325183	0.8617	0.482203
34	34	0.88672	0.325733	0.8336	0.607303
35	35	0.89154	0.317828	0.8583	0.497901
36	36	0.89086	0.314376	0.8519	0.510494
37	37	0.89194	0.314032	0.8658	0.489733
38	38	0.89256	0.308741	0.8856	0.369536
39	39	0.89456	0.301537	0.8551	0.494733
40	40	0.89734	0.296842	0.8616	0.490008
41	41	0.89682	0.295763	0.8734	0.418190
42	42	0.89846	0.290511	0.8800	0.422501
43	43	0.89956	0.287738	0.8715	0.449149
44	44	0.89818	0.289174	0.8712	0.435780
45	45	0.90184	0.282060	0.8671	0.453489
46	46	0.90128	0.280366	0.8829	0.400324
47	47	0.90338	0.274910	0.8685	0.488103
48	48	0.90378	0.275261	0.8791	0.413528

	epoch	accuracy	loss	val_accuracy	val_loss
49	49	0.90472	0.272393	0.8624	0.495408
50	50	0.92052	0.227371	0.9093	0.302220

In [ ]:

In [ ]: