```
# DongKyu Kim
# ECE 471 CGML Assignment 4
# CIFAR-10
# Professor Curro

# library imports
import numpy as np
from tqdm import tqdm
import keras
import tensorflow as tf
from keras import optimizers, regularizers
from keras.datasets import cifar10
from keras.models import Model
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D
from keras.layers import BatchNormalization, Activation, Input
from keras.callbacks import LearningRateScheduler

# Parameters
r = 0.9 # Ratio of training data to (training + validation)
batch_size = 128
epochs = 200
numclass = 10
learning_rate = 0.005

# Useful functions
def one_hot_encoding(data,numclass):
    targets = np.array(data).reshape(-1)
    return np.eye(numclass)[targets]

def lr_adaptive(self,epoch):
    lr = learning_rate
    if epoch > 20:
        lr = learning_rate/10
    elif epoch > 40:
        lr = learning_rate/20
    return lr

# Data Generation
class Data(object):
    def __init__(self,r,numclass):
        (xtemp,ytemp),(self.X_test,self.Y_test) = cifar10.load_data()
        mask = np.full(xtemp.shape[0],True)
        mask[np.random.choice(xtemp.shape[0],
            int(xtemp.shape[0]*(1-r)),replace=False)] = False
        self.numclass = numclass
        self.X_train = xtemp[mask].astype('float32')
        self.Y_train = ytemp[mask]
        self.X_val = xtemp[~mask].astype('float32')
        self.Y_val = ytemp[~mask]
        self.X_test = self.X_test.astype('float32')
        self.Y_test = self.Y_test
        self.X_train = self.normalize(self.X_train)
        self.X_val = self.normalize(self.X_val)
        self.X_test = self.normalize(self.X_test)
        self.Y_train = one_hot_encoding(self.Y_train,self.numclass)
        self.Y_val = one_hot_encoding(self.Y_val,self.numclass)
        self.Y_test = one_hot_encoding(self.Y_test,self.numclass)

    def normalize(self,X):
        X /= 255
        # These values are from
        # https://github.com/kuangliu/pytorch-cifar/issues/19
        mu = [0.4914,0.4822,0.4465]
```

```
        std = [0.2023,0.1994,0.2010]
        for i in range(0,2):
            X[:,:,:,i] = (X[:,:,:,i]-mu[i])/std[i]
        return X

# Model
class My_Model(object):
    def __init__(self,data,batch_size,epochs):
        self.data = data
        self.batch_size = batch_size
        self.epochs = epochs

    def convconv(self,filters,kernel_size,strides):
        return Conv2D(filters,kernel_size,strides =strides,
            padding='same',activation='relu')

    #The below block is from [1], Second branch is motivated by [3]
    #The below function is ultimately unused,
    #however I left this here for future usage.
    def residual_block(self,input,filter):
        x = BatchNormalization()(input)
        x = Activation('relu')(x)
        x = self.convconv(filter,[3,3],1)(x)
        x = BatchNormalization()(input)
        x = Activation('relu')(x)
        x = self.convconv(filter,[3,3],1)(x)

        y = BatchNormalization()(input)
        y = Activation('relu')(y)
        y = self.convconv(filter,[3,3],1)(y)
        y = BatchNormalization()(y)
        y = Activation('relu')(y)
        y = self.convconv(filter,[3,3],1)(y)
        return keras.layers.add([input,x,y])

    #This is a combination of residual block without the identity connection.
    def all_conv_block(self,filters,dropout,input):
        x = self.convconv(filters,[3,3],1)(input)
        x = BatchNormalization()(x)
        x = self.convconv(filters,[3,3],1)(x)
        x = BatchNormalization()(x)
        x = MaxPooling2D(2,strides = 2)(x)
        x = Dropout(dropout)(x)
        return x

    def build_model(self):
        input = Input(shape = self.data.X_train.shape[1:])
        x = self.all_conv_block(32,0.25,input)
        x = self.all_conv_block(64,0.35,x)
        x = self.all_conv_block(128,0.45,x)
        x = self.all_conv_block(256,0.35,x)
        x = Flatten()(x)
        x = Dense(1024,activation = 'relu')(x)
        y = Dense(self.data.numclass,activation='softmax')(x)
        self.model = Model(inputs = input, outputs = y)

    def train(self):
        self.optim = optimizers.Adam(learning_rate)
        self.model.compile(self.optim,'categorical_crossentropy',['accuracy'])
        self.datagen = keras.preprocessing.image.ImageDataGenerator(
            horizontal_flip = True,fill_mode = 'constant',
            width_shift_range = 4, height_shift_range = 4
            )
```

```
        self.datagen.fit(self.data.X_train)
        self.model.fit_generator(self.datagen.flow(self.data.X_train,
            self.data.Y_train,batch_size=self.batch_size),
            steps_per_epoch=len(self.data.X_train)/self.batch_size,
            epochs=self.epochs,
            validation_data=(self.data.X_val,self.data.Y_val),
            callbacks=[LearningRateScheduler(lr_adaptive)],verbose=2)

print('This is my cifar_10 case')
data = Data(r,numclass)
My_Model = My_Model(data,batch_size,epochs)
My_Model.build_model()
My_Model.train()
scores = My_Model.model.evaluate(data.X_test,data.Y_test,verbose=2)
print('Test loss:', scores[0])
print('Test accuracy:',scores[1])

# The state of the art cifar10 is 98.52% from Wikipedia achieved by
# AutoAugment: Learning Augmentation Policies from Data
# I started with my MNIST model, it didn't work well. Then I experimented with
# a deeper version of the MNIST model, but it just achieved 50% top-1 accuracy.
# I moved into residual neural network based on [1], and [2]. After 32 epochs,
# it converges at validation accuracy of 0.6479, and train set accuracy of
# 0.9967, with best validation accuracy of 0.6513. This model overfits, so I
# added a simple data augmentation scheme. With this augmentation method, I
# reach a validation accuracy of 0.6821, which is slightly better, but not that
# good. Then I did the data normalization off of a already known values because
# I didn't want to waste computational time, and result improved a lot to
# 0.8042 validation error.
# I tried elu, rotation_range in data augmentation, added another branch in
# residual network played around with learning rate, but all of these methods
# did not really help. Then I discussed with classmates and heard that simple
# convolutional neural network might work better. This made sense to me because
# even in the residual network paper, they mention for the deep versions of the
# model, they started with a simple conv net layers, and then add the residual
# network, and all the state of the art papers ran the model for 6 weeks, so
# maybe with my device and time, residual network wasn't effective. So I
# changed back to a deep 12 layer network with conv-net blocks, with adaptive
# learning rate, I achieved validation accuracy of 0.9120 and test accuracy of
# 0.9029.


# [1] Identity Mappings in Deep Residual Networks
#   https://arxiv.org/pdf/1603.05027.pdf
# [2] Deep Residual Learning for Image Recognition
#   https://arxiv.org/pdf/1512.03385.pdf
# [3] Shake-Shake regularization
#   https://arxiv.org/pdf/1705.07485.pdf
```

```
This is a modified output because the original had all 200 epochs.
I decided to show first and every 10th epoch.
This is my cifar_10 case
Epoch 1/200
 - 24s - loss: 2.6348 - acc: 0.2571 - val_loss: 2.0944 - val_acc: 0.2887
Epoch 10/200
 - 20s - loss: 0.7270 - acc: 0.7484 - val_loss: 0.6200 - val_acc: 0.7810
Epoch 20/200
 - 20s - loss: 0.5163 - acc: 0.8234 - val_loss: 0.4570 - val_acc: 0.8418
Epoch 30/200
 - 20s - loss: 0.4340 - acc: 0.8513 - val_loss: 0.4037 - val_acc: 0.8638
Epoch 40/200
 - 20s - loss: 0.3753 - acc: 0.8717 - val_loss: 0.3786 - val_acc: 0.8712
Epoch 50/200
 - 20s - loss: 0.3382 - acc: 0.8833 - val_loss: 0.3326 - val_acc: 0.8866
Epoch 60/200
 - 19s - loss: 0.3120 - acc: 0.8918 - val_loss: 0.3309 - val_acc: 0.8906
Epoch 70/200
 - 19s - loss: 0.2928 - acc: 0.8981 - val_loss: 0.3199 - val_acc: 0.8998
Epoch 80/200
 - 19s - loss: 0.2770 - acc: 0.9058 - val_loss: 0.3061 - val_acc: 0.8990
Epoch 90/200
 - 19s - loss: 0.2617 - acc: 0.9100 - val_loss: 0.2890 - val_acc: 0.9090
Epoch 100/200
 - 19s - loss: 0.2429 - acc: 0.9163 - val_loss: 0.3172 - val_acc: 0.9032
Epoch 110/200
 - 19s - loss: 0.2328 - acc: 0.9198 - val_loss: 0.3092 - val_acc: 0.9014
Epoch 120/200
 - 19s - loss: 0.2257 - acc: 0.9214 - val_loss: 0.3021 - val_acc: 0.9076
Epoch 130/200
 - 19s - loss: 0.2123 - acc: 0.9263 - val_loss: 0.3055 - val_acc: 0.9068
Epoch 140/200
 - 19s - loss: 0.2098 - acc: 0.9271 - val_loss: 0.2886 - val_acc: 0.9132
Epoch 150/200
 - 19s - loss: 0.2001 - acc: 0.9314 - val_loss: 0.2978 - val_acc: 0.9140
Epoch 160/200
 - 19s - loss: 0.1914 - acc: 0.9336 - val_loss: 0.2907 - val_acc: 0.9150
Epoch 170/200
 - 19s - loss: 0.1928 - acc: 0.9326 - val_loss: 0.3013 - val_acc: 0.9090
Epoch 180/200
 - 19s - loss: 0.1885 - acc: 0.9340 - val_loss: 0.2818 - val_acc: 0.9114
Epoch 190/200
 - 19s - loss: 0.1792 - acc: 0.9368 - val_loss: 0.2858 - val_acc: 0.9190
Epoch 200/200
 - 19s - loss: 0.1730 - acc: 0.9377 - val_loss: 0.3113 - val_acc: 0.9120
Test loss:  0.35426006702445445
Test accuracy: 0.9029
```

```
# DongKyu Kim
# ECE 471 CGML Assignment 4
# CIFAR-100
# Professor Curro

# library imports
import numpy as np
from tqdm import tqdm
import keras
import tensorflow as tf
from keras import optimizers, regularizers
from keras.datasets import cifar100
from keras.models import Model
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D
from keras.layers import BatchNormalization, Activation, Input
from keras.callbacks import LearningRateScheduler

# Parameters
r = 0.9
batch_size = 128
epochs = 200
numclass = 100
learning_rate = 0.005

# Useful functions
def one_hot_encoding(data,numclass):
    targets = np.array(data).reshape(-1)
    return np.eye(numclass)[targets]

def lr_adaptive(self,epoch):
    lr = learning_rate
    if epoch > 20:
        lr = learning_rate/10
    elif epoch > 40:
        lr = learning_rate/20
    return lr

# Data Generation
class Data(object):
    def __init__(self,r,numclass):
        (xtemp,ytemp),(self.X_test,self.Y_test) = cifar100.load_data()
        mask = np.full(xtemp.shape[0],True)
        mask[np.random.choice(xtemp.shape[0],
            int(xtemp.shape[0]*(1-r)),replace=False)] = False
        self.numclass = numclass
        self.X_train = xtemp[mask].astype('float32')
        self.Y_train = ytemp[mask]
        self.X_val = xtemp[~mask].astype('float32')
        self.Y_val = ytemp[~mask]
        self.X_test = self.X_test.astype('float32')
        self.Y_test = self.Y_test
        self.X_train = self.normalize(self.X_train)
        self.X_val = self.normalize(self.X_val)
        self.X_test = self.normalize(self.X_test)
        self.Y_train = one_hot_encoding(self.Y_train,self.numclass)
        self.Y_val = one_hot_encoding(self.Y_val,self.numclass)
        self.Y_test = one_hot_encoding(self.Y_test,self.numclass)

    def normalize(self,X):
        X /= 255
        mu = [0.4914,0.4822,0.4465]
        std = [0.2023,0.1994,0.2010]
        for i in range(0,2):
```

```
            X[:,:,:,i] = (X[:,:,:,i]-mu[i])/std[i]
        return X
# Model
class My_Model(object):
    def __init__(self,data,batch_size,epochs):
        self.data = data
        self.batch_size = batch_size
        self.epochs = epochs

    def convconv(self,filters,kernel_size,strides):
        return Conv2D(filters,kernel_size,strides =strides,
            padding='same',activation='relu')

    def all_conv_block(self,filters,dropout,input):
        x = self.convconv(filters,[3,3],1)(input)
        x = BatchNormalization()(x)
        x = self.convconv(filters,[3,3],1)(x)
        x = BatchNormalization()(x)
        x = MaxPooling2D(2,strides = 2)(x)
        x = Dropout(dropout)(x)
        return x

    def build_model(self):
        input = Input(shape = self.data.X_train.shape[1:])
        x = self.all_conv_block(32,0.25,input)
        x = self.all_conv_block(64,0.35,x)
        x = self.all_conv_block(128,0.45,x)
        x = self.all_conv_block(256,0.35,x)
        x = Flatten()(x)
        x = Dense(1024,activation = 'relu')(x)
        y = Dense(self.data.numclass,activation='softmax')(x)
        self.model = Model(inputs = input, outputs = y)

    def train(self):
        self.optim = optimizers.Adam(learning_rate)
        self.model.compile(self.optim,'categorical_crossentropy',
            ['accuracy','top_k_categorical_accuracy'])
        self.datagen = keras.preprocessing.image.ImageDataGenerator(
            horizontal_flip = True,fill_mode = 'constant',
            width_shift_range = 4, height_shift_range = 4
            )
        self.datagen.fit(self.data.X_train)
        self.model.fit_generator(self.datagen.flow(self.data.X_train,
            self.data.Y_train,batch_size=self.batch_size),
            steps_per_epoch=len(self.data.X_train)/self.batch_size,
            epochs=self.epochs,
            validation_data=(self.data.X_val,self.data.Y_val),
            callbacks=[LearningRateScheduler(lr_adaptive)],verbose=2)


print('This is my cifar_100 case')
data = Data(r,numclass)
My_Model = My_Model(data,batch_size,epochs)
My_Model.build_model()
My_Model.train()
scores = My_Model.model.evaluate(data.X_test,data.Y_test,verbose=2)
print('Test loss: ', scores[0])
print('Test accuracy:',scores[1])
print('Test top 5 accuracy:',scores[2])

# I omitted all the comments that I already made in the cifar-10.
# I stopped and checked this code on cifar-100 when cifar-10 reached 0.8042
# validation accuracy. The top5 accuracy for validation is 0.7435 and test
```

```
# accuracy is 0.7465. This is over the minimum requirement. The model was run
# for 20 epochs. It looks like if I did more epochs I will get a better
# accuracy, but instead of running it, I decided to make a better model.
# I indeed constructed a model that works better with cifar-10.
# Using the updated model top-5 validation accuracy was 0.8418 and
# top-5 test accuracy was 0.8374.
```

```
This is a modified output because the original had all 200 epochs.
I decided to show first and every 10th epoch.
I also added new line after train top_k_categorical_accuracy for aesthetics.
This is my cifar_100 case
Epoch 1/200
 - 23s - loss: 4.5853 - acc: 0.0395 - top_k_categorical_accuracy: 0.1648
 - val_loss: 4.2882 - val_acc: 0.0566 - val_top_k_categorical_accuracy: 0.2212
Epoch 10/200
 - 19s - loss: 2.7005 - acc: 0.3017 - top_k_categorical_accuracy: 0.6256
 - val_loss: 2.5983 - val_acc: 0.3295 - val_top_k_categorical_accuracy: 0.6479
Epoch 20/200
 - 19s - loss: 2.2987 - acc: 0.3890 - top_k_categorical_accuracy: 0.7123
 - val_loss: 2.1179 - val_acc: 0.4281 - val_top_k_categorical_accuracy: 0.7544
Epoch 30/200
 - 19s - loss: 2.1717 - acc: 0.4196 - top_k_categorical_accuracy: 0.7422
 - val_loss: 2.0597 - val_acc: 0.4579 - val_top_k_categorical_accuracy: 0.7620
Epoch 40/200
 - 19s - loss: 1.9454 - acc: 0.4700 - top_k_categorical_accuracy: 0.7826 -
  val_loss: 1.9816 - val_acc: 0.4687 - val_top_k_categorical_accuracy: 0.7816
Epoch 50/200
 - 19s - loss: 1.8638 - acc: 0.4876 - top_k_categorical_accuracy: 0.7963
 - val_loss: 1.8246 - val_acc: 0.5141 - val_top_k_categorical_accuracy: 0.8162
Epoch 60/200
 - 19s - loss: 1.6748 - acc: 0.5325 - top_k_categorical_accuracy: 0.8314
 - val_loss: 1.7033 - val_acc: 0.5431 - val_top_k_categorical_accuracy: 0.8296
Epoch 70/200
 - 19s - loss: 1.7501 - acc: 0.5127 - top_k_categorical_accuracy: 0.8175
 - val_loss: 1.7578 - val_acc: 0.5255 - val_top_k_categorical_accuracy: 0.8194
Epoch 80/200
 - 19s - loss: 1.6369 - acc: 0.5421 - top_k_categorical_accuracy: 0.8380
 - val_loss: 1.7103 - val_acc: 0.5449 - val_top_k_categorical_accuracy: 0.8276
Epoch 90/200
 - 19s - loss: 1.6480 - acc: 0.5398 - top_k_categorical_accuracy: 0.8381
 - val_loss: 1.7076 - val_acc: 0.5521 - val_top_k_categorical_accuracy: 0.8286
Epoch 100/200
 - 19s - loss: 1.5067 - acc: 0.5706 - top_k_categorical_accuracy: 0.8604
 - val_loss: 1.6648 - val_acc: 0.5603 - val_top_k_categorical_accuracy: 0.8390
Epoch 110/200
 - 19s - loss: 1.5767 - acc: 0.5562 - top_k_categorical_accuracy: 0.8496
 - val_loss: 1.7459 - val_acc: 0.5453 - val_top_k_categorical_accuracy: 0.8298
Epoch 120/200
 - 19s - loss: 1.4045 - acc: 0.5976 - top_k_categorical_accuracy: 0.8795
 - val_loss: 1.7573 - val_acc: 0.5541 - val_top_k_categorical_accuracy: 0.8338
Epoch 130/200
 - 19s - loss: 1.3542 - acc: 0.6087 - top_k_categorical_accuracy: 0.8859
 - val_loss: 1.7166 - val_acc: 0.5637 - val_top_k_categorical_accuracy: 0.8412
Epoch 140/200
 - 19s - loss: 1.4385 - acc: 0.5940 - top_k_categorical_accuracy: 0.8706
 - val_loss: 1.7157 - val_acc: 0.5663 - val_top_k_categorical_accuracy: 0.8306
Epoch 150/200
 - 19s - loss: 1.4130 - acc: 0.6006 - top_k_categorical_accuracy: 0.8799
 - val_loss: 1.8080 - val_acc: 0.5381 - val_top_k_categorical_accuracy: 0.8194
Epoch 160/200
 - 19s - loss: 1.5302 - acc: 0.5681 - top_k_categorical_accuracy: 0.8564
 - val_loss: 1.7696 - val_acc: 0.5629 - val_top_k_categorical_accuracy: 0.8374
Epoch 170/200
 - 19s - loss: 1.4339 - acc: 0.5958 - top_k_categorical_accuracy: 0.8728
 - val_loss: 1.7339 - val_acc: 0.5555 - val_top_k_categorical_accuracy: 0.8326
Epoch 180/200
 - 19s - loss: 1.4378 - acc: 0.5932 - top_k_categorical_accuracy: 0.8708
 - val_loss: 1.7565 - val_acc: 0.5663 - val_top_k_categorical_accuracy: 0.8372
Epoch 190/200
 - 19s - loss: 1.4014 - acc: 0.6106 - top_k_categorical_accuracy: 0.8776
```

```
 - val_loss: 1.6688 - val_acc: 0.5647 - val_top_k_categorical_accuracy: 0.8304
Epoch 200/200
 - 19s - loss: 1.2619 - acc: 0.6379 - top_k_categorical_accuracy: 0.8998
 - val_loss: 1.7847 - val_acc: 0.5755 - val_top_k_categorical_accuracy: 0.8418
Test loss:  1.7742234039306641
Test accuracy: 0.5744
Test top 5 accuracy: 0.8374
```