```
# Ravindra Bisram
# Deep Learning Homework 4
import pickle
import sys
import os
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Conv2D, Dropout, MaxPool2D,
Flatten, Add, Dense, Activation, BatchNormalization, Lambda, ReLU,
PReLU
from tensorflow.keras.layers import Conv2D, AveragePooling2D,
MaxPooling2D, Flatten, Input, concatenate, ZeroPadding2D, LeakyReLU,
GlobalAveragePooling2D
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.regularizers import l2
from tensorflow.keras.optimizers import Adam, SGD, RMSprop
from tensorflow.keras.callbacks import ReduceLROnPlateau,
ModelCheckpoint, LearningRateScheduler
from sklearn.model_selection import train_test_split
from keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.metrics import TopKCategoricalAccuracy

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
"""
The CIFAR10 and CIFAR100 datasets were downloaded through the official
website. Each training batch for CIFAR10 was unpickled and then
appended to each other to create one large training set. The images
were preprocessed to convert the initial row vector to shape (32, 32,
3) through reshaping and transposing. The class output data was one
hot encoded. My preliminary architecture recycled my model for the
MNIST dataset, with an alteration for the input size. Thisresulted in
a test accuracy of 67%. The second architecture I tried was a VGG with
fractional max pooling, based on a paper by Benjamin Graham. While
this definitely outperformed the previous model, the computational
time and rate at which the model learned seemed to slow, and
ultimately converged around 86% accuracy. At this point I tried
playing around with a GoogleNet implementation, which was effective,
but still very slow to train. I continued to roughly follow the VGG
type CNN architecture, opting to make it shallower and introducing
data augmentation, learning rate decay, variable learning rate, and
early stopping. All of these factors combinedresulted in a network
that had very fast steps and converged well after only a hundred or so
epochs. I setteld on this architecture since it had reached an
accuracy of 90% for CIFAR10, and over 95% for CIFAR100. As instructed
in office hours, I also implemented a residual network, and found that
this model got me similar results in the 88-90% range for CIFAR10. To
keep my submission short, I have only included the model I settled on
(which I termed ravnet) and the ResNet, though I coded up many
different models in a similar way.
I would like to give shoutouts to Bob for the tip to try and use
```

```
GoogleNet, and Joey Berkowitz for introducing me to early stopping.
Danny Hong also helped me find good dropout percentages for the
ravnet.

Misc attempts at improvement:
- I tried reducing training time by using only one of the 5 training
batches and doubling the batch size to 512, but the tradeoff with
accuracy was way too high.
- I played around with different optimization functions, including SGD
with momentum, and RMS prop.
- Using ravnet with no dropout in the initial layers worked great for
CIFAR10, but was not useful for CIFAR100.
- I played around with model checkpoints to save progress, but ended
up never actually reloading weights.
- Played around with different kernel initializers.

Thus concludes my voyage for hw4.
"""
# Fractional max pooling
# - https://arxiv.org/abs/1412.6071
# -
https://github.com/laplacetw/vgg-like-cifar10/blob/master/fmp_cifar10.
py
# https://www.binarystudy.com/2021/09/how-to-load-preprocess-
visualize-CIFAR-10-and-CIFAR-100.html#routine

BTEST = '../data/test_batch'
meta_file = '../CIFAR10-data/batches.meta'

NUM_TRAINING_BATCHES = 5
BATCH_SIZE = 128 #128
LAMBDA = 1e-5
EPOCHS = 150
IMG_SIDE_LEN = 32
LR = 5e-3
DATASET = "CIFAR10"
MODEL = "resnet34"
COARSE = False

def unpickle(file):
    with open(file, 'rb') as fo:
        u = pickle._Unpickler( fo )
        u.encoding = 'latin1'
        dict = u.load()
    return dict

def load_training_data():
    # The whole data_batch_1 has 10,000 images. And each image is a 1-
D array having 3,072 entries.
    # First 1024 entries for Red, the next 1024 entries for Green and
```

```
last 1024 entries for Blue channels.
    print("Loading Data:")
    features, classes = np.empty((0,32,32,3)), np.empty((0,10))
    for i in range(NUM_TRAINING_BATCHES):
        print(f"Batch {i+1}")
        batch_path = f'../data/data_batch_{i+1}'
        x, y = reshape_features(batch_path)
        features = np.append(features, x, axis=0)
        classes = np.append(classes, y, axis=0)
    return features, classes

def reshape_features(feat_path, CIFAR100=False):
    labels = ('coarse_labels' if COARSE else 'fine_labels') if
CIFAR100 else 'labels'
    unpickled_data = unpickle(feat_path)
    return
(unpickled_data['data'].reshape(len(unpickled_data['data']),3,32,32).t
ranspose(0,2,3,1) / 255,
            tf.keras.utils.to_categorical(unpickled_data[labels]))

def frac_max_pool(x):
    return tf.nn.fractional_max_pool(x, [1.0, 1.41, 1.41, 1.0],
pseudo_random=True, overlapping=True)[0]

def poly_decay(epoch):
  maxEpochs = EPOCHS
  baseLR = LR
  power = 1.0
  alpha = baseLR * (1 - (epoch   / float(maxEpochs))) ** power
  return alpha

datagen = ImageDataGenerator(
    rotation_range=15,
    horizontal_flip=True,
    width_shift_range=0.1,
    height_shift_range=0.1
)

stopping = tf.keras.callbacks.EarlyStopping(
          monitor="val_accuracy",
          min_delta=0,
          patience=25,
          verbose=1,
          mode="max",
          baseline=None,
          restore_best_weights=True)

def normalize_x_data(x_train, x_test):
    eps = 1e-7
    mean = np.mean(x_train,axis = (0, 1, 2, 3))
```

```
    std = np.std(x_train,axis = (0, 1, 2, 3))
    x_train = (x_train - mean)/(std + eps)
    x_test = (x_test - mean)/(std + eps)
    return x_train, x_test

class Data10(object):
    def __init__(self):
        self.x_train, self.y_train = load_training_data()
        self.x_test, self.y_test = reshape_features(BTEST)
        self.x_train, self.x_test = normalize_x_data(self.x_train,
self.x_test)
        self.x_train, self.x_val, self.y_train, self.y_val =
train_test_split(self.x_train, self.y_train, test_size=0.2,
random_state=31415)

class Data100(object):
    def __init__(self):
      self.x_train, self.y_train = reshape_features('../data/train',
CIFAR100=True)
      self.x_test, self.y_test = reshape_features('../data/test',
CIFAR100=True)
      self.x_train, self.x_val, self.y_train, self.y_val =
train_test_split(self.x_train, self.y_train, test_size=0.2,
random_state=31415)

def double_conv_module(input, num_filters, activation, kern_reg,
dropout, padding="same"):
    input = Conv2D(filters = num_filters, kernel_size = (3, 3),
activation = activation, padding = padding, kernel_regularizer =
kern_reg)(input)
    input = BatchNormalization(axis=-1)(input)
    input = Conv2D(filters = num_filters, kernel_size = (3, 3),
activation = activation, padding = padding, kernel_regularizer =
kern_reg)(input)
    input = BatchNormalization(axis=-1)(input)
    input = MaxPooling2D(pool_size = (2, 2))(input)
    input = Dropout(dropout)(input)

    return input

# Primary model I was going to submit until Monday office hours resnet
suggestion
def rav_model(width, height, depth, classes):
    inputShape=(height, width, depth)
    weight_decay = 0.001

    inputs = Input(shape=inputShape)
    KR = None #l2(weight_decay)
    x = double_conv_module(inputs, 32, activation='relu', kern_reg=KR,
dropout = 0.1, padding='same')
```

```
    x = double_conv_module(x, 64, activation='relu', kern_reg=KR,
dropout = 0.2, padding='same')
    x = double_conv_module(x, 128, activation='relu', kern_reg=KR,
dropout = 0.3, padding='same')
    x = double_conv_module(x, 128, activation='relu', kern_reg=KR,
dropout = 0.4, padding='same')

    x = Flatten()(x)
    x = Dense(512, activation='relu',kernel_regularizer=None)(x)
    x = BatchNormalization(axis=-1)(x)
    x = Dropout(0.5)(x)
    x = Dense(classes)(x)
    x = Activation("softmax")(x)

    model = Model(inputs, x, name="rav_net")
    return model

# Materials Referenced for Residual Implementation:
# https://www.analyticsvidhya.com/blog/2021/08/how-to-code-your-
resnet-from-scratch-in-tensorflow/
# https://towardsdatascience.com/building-a-resnet-in-keras-
e8f1322a49ba
# https://machinelearningknowledge.ai/googlenet-architecture-
implementation-in-keras-with-cifar-10-dataset/
# https://sebastianwallkoetter.wordpress.com/2018/04/08/layered-
layers-residual-blocks-in-the-sequential-keras-api/
#
https://github.com/Adeel-Intizar/CIFAR-10-State-of-the-art-Model/blob/
master/CIFAR-10%20Best.ipynb

def identity_block(x, filter):
    # Maintain the input for the skip connection
    x_skip = x

    x = Conv2D(filter, (3,3), padding = 'same')(x)
    x = BatchNormalization(axis=3)(x)
    x = Activation('relu')(x)

    x = Conv2D(filter, (3,3), padding = 'same')(x)
    x = BatchNormalization(axis=3)(x)

    # Residual part --> add the input to the output
    x = Add()([x, x_skip])
    x = Activation('relu')(x)
    return x

def convolutional_block(x, filter):
    x_skip = x
```

```
    x = Conv2D(filter, (3,3), padding = 'same', strides = (2,2))(x)
    x = BatchNormalization(axis=3)(x)
    x = Activation('relu')(x)

    x = Conv2D(filter, (3,3), padding = 'same')(x)
    x = BatchNormalization(axis=3)(x)
    # Process the residual with a (1,1) convolution
    x_skip = Conv2D(filter, (1,1), strides = (2,2))(x_skip)

    x = Add()([x, x_skip])
    x = Activation('relu')(x)
    return x

def ResNet(shape = (32, 32, 3), classes = 10):
    inputs = Input(shape)
    x = ZeroPadding2D((3, 3))(inputs)

    # Initial Conv layer and Maxpooling
    x = Conv2D(64, kernel_size=7, strides=2, padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    x = MaxPool2D(pool_size=3, strides=2, padding='same')(x)

    # Size of sub-blocks and initial filter size
    block_layers = [3, 4, 6, 3]
    filter_size = 64

    # Resnet Blocks
    for i in range(4):
        if i == 0:
            for j in range(block_layers[i]):
                x = identity_block(x, filter_size)
        else:
            # One Residual/Convolutional Block followed by Identity
blocks
            filter_size = filter_size*2     # Filter size increases by
factors of 2
            x = convolutional_block(x, filter_size)
            for j in range(block_layers[i] - 1):
                x = identity_block(x, filter_size)

    # Recycle the dense layers from my previous model
    x = AveragePooling2D((2,2), padding = 'same')(x)
    x = Flatten()(x)
    x = Dense(512, activation = 'relu')(x)
    x = Dense(classes)(x)
    x = Activation("softmax")(x)
    model = Model(inputs = inputs, outputs = x, name = "ResNet")
    return model
```

```
if __name__ == "__main__":
  if DATASET == 'CIFAR10':
    data = Data10()
    NUM_CLASSES = 10
  else:
    data = Data100()
    NUM_CLASSES = 20 if COARSE else 100

  x_train, y_train = data.x_train, data.y_train
  x_test, y_test = data.x_test, data.y_test
  x_val, y_val = data.x_val, data.y_val

  lr_scheduler = LearningRateScheduler(poly_decay)
  variable_learning_rate = ReduceLROnPlateau(monitor='val_loss',
factor = 0.2, patience = 2)

  if MODEL == "ravnet":
    model = rav_model(width=32, height=32, depth=3,
classes=NUM_CLASSES)

  elif MODEL == "resnet":
    model = ResNet(classes=NUM_CLASSES)

  opt=Adam(learning_rate=0.001,decay=0, beta_1=0.9, beta_2=0.999,
epsilon=1e-08)

  if DATASET == "CIFAR10":
    model.compile(loss=tf.keras.losses.categorical_crossentropy,
metrics=['accuracy'],optimizer=opt)
  else:
    model.compile(loss=tf.keras.losses.categorical_crossentropy,
metrics=[TopKCategoricalAccuracy(k = 5)],optimizer=opt)

  model.summary()
  history=model.fit(datagen.flow(x_train, y_train,
batch_size=BATCH_SIZE),
                    batch_size=BATCH_SIZE,
                    epochs=EPOCHS,
                    callbacks=[variable_learning_rate, lr_scheduler,
stopping],
                    validation_data=(x_val, y_val),
                    verbose=1,
                    steps_per_epoch = len(x_train) // BATCH_SIZE)
  score = model.evaluate(x_test, y_test, verbose=0)


  print('Test loss:', score[0])
  print('Test accuracy:', score[1])
```

CIFAR 10 Dataset:

```
================================================================
Total params: 854,826
Trainable params: 852,394
Non-trainable params: 2,432
________________________________________________________________
Epoch 1/200
312/312 [==============================] - 35s 67ms/step - loss: 1.8938 - accuracy: 0.3562 - val_loss: 1.8340 - val_accuracy: 0.4146 - lr: 0.0050
Epoch 2/200
312/312 [==============================] - 20s 64ms/step - loss: 1.4649 - accuracy: 0.4864 - val_loss: 1.7454 - val_accuracy: 0.4555 - lr: 0.0050
Epoch 3/200
312/312 [==============================] - 20s 64ms/step - loss: 1.2222 - accuracy: 0.5714 - val_loss: 1.3573 - val_accuracy: 0.5847 - lr: 0.0049
Epoch 4/200
312/312 [==============================] - 20s 64ms/step - loss: 1.0167 - accuracy: 0.6432 - val_loss: 1.0941 - val_accuracy: 0.6520 - lr: 0.0049
Epoch 5/200
312/312 [==============================] - 20s 64ms/step - loss: 0.9078 - accuracy: 0.6858 - val_loss: 0.8463 - val_accuracy: 0.7113 - lr: 0.0049
```

```
Epoch 190/200
312/312 [==============================] - 20s 64ms/step - loss: 0.1220 - accuracy: 0.9576 - val_loss: 0.3223 - val_accuracy: 0.9089 - lr: 2.7500e-04
Epoch 191/200
312/312 [==============================] - 20s 65ms/step - loss: 0.1241 - accuracy: 0.9567 - val_loss: 0.3203 - val_accuracy: 0.9096 - lr: 5.0000e-05
Epoch 192/200
312/312 [==============================] - 20s 65ms/step - loss: 0.1201 - accuracy: 0.9581 - val_loss: 0.3213 - val_accuracy: 0.9105 - lr: 2.2500e-04
Epoch 193/200
312/312 [==============================] - 20s 64ms/step - loss: 0.1203 - accuracy: 0.9572 - val_loss: 0.3184 - val_accuracy: 0.9124 - lr: 4.0000e-05
Epoch 194/200
312/312 [==============================] - 20s 64ms/step - loss: 0.1176 - accuracy: 0.9584 - val_loss: 0.3199 - val_accuracy: 0.9126 - lr: 1.7500e-04
Epoch 195/200
312/312 [==============================] - 20s 65ms/step - loss: 0.1194 - accuracy: 0.9573 - val_loss: 0.3206 - val_accuracy: 0.9120 - lr: 3.0000e-05
Epoch 196/200
312/312 [==============================] - 20s 65ms/step - loss: 0.1189 - accuracy: 0.9571 - val_loss: 0.3199 - val_accuracy: 0.9119 - lr: 1.2500e-04
Epoch 197/200
312/312 [==============================] - 20s 64ms/step - loss: 0.1182 - accuracy: 0.9585 - val_loss: 0.3207 - val_accuracy: 0.9120 - lr: 2.0000e-05
Epoch 198/200
312/312 [==============================] - 20s 65ms/step - loss: 0.1190 - accuracy: 0.9577 - val_loss: 0.3200 - val_accuracy: 0.9111 - lr: 7.5000e-05
Epoch 199/200
312/312 [==============================] - 20s 65ms/step - loss: 0.1166 - accuracy: 0.9588 - val_loss: 0.3221 - val_accuracy: 0.9109 - lr: 1.0000e-05
Epoch 200/200
312/312 [==============================] - 20s 65ms/step - loss: 0.1147 - accuracy: 0.9600 - val_loss: 0.3202 - val_accuracy: 0.9114 - lr: 2.5000e-05
Test loss: 0.3363804817199707
Test accuracy: 0.907800018787384
```

CIFAR 100: (ignore the warnings, I accidentally left the same early stopping metric I used for my CIFAR10 run, and by the time I realized I was too deep into the run).

```
=================================================================
Total params: 859,956
Trainable params: 857,524
Non-trainable params: 2,432
_________________________________________________________________
Epoch 1/200
312/312 [==============================] - ETA: 0s - loss: 2.8912 - top_k_categorical_accuracy: 0.5330
WARNING:tensorflow:Early stopping conditioned on metric `val_accuracy` which is not available. Available metrics are: loss,top_k_categorical_accuracy,v
al_loss,val_top_k_categorical_accuracy,lr
312/312 [==============================] - 34s 68ms/step - loss: 2.8912 - top_k_categorical_accuracy: 0.5330 - val_loss: 2.7977 - val_top_k_categorical
_accuracy: 0.4984 - lr: 0.0050
Epoch 2/200
312/312 [==============================] - ETA: 0s - loss: 2.5115 - top_k_categorical_accuracy: 0.6461
WARNING:tensorflow:Early stopping conditioned on metric `val_accuracy` which is not available. Available metrics are: loss,top_k_categorical_accuracy,v
al_loss,val_top_k_categorical_accuracy,lr
312/312 [==============================] - 19s 62ms/step - loss: 2.5115 - top_k_categorical_accuracy: 0.6461 - val_loss: 2.6826 - val_top_k_categorical
_accuracy: 0.6061 - lr: 0.0050
Epoch 3/200
312/312 [==============================] - ETA: 0s - loss: 2.3005 - top_k_categorical_accuracy: 0.7029
WARNING:tensorflow:Early stopping conditioned on metric `val_accuracy` which is not available. Available metrics are: loss,top_k_categorical_accuracy,v
al_loss,val_top_k_categorical_accuracy,lr
312/312 [==============================] - 19s 62ms/step - loss: 2.3005 - top_k_categorical_accuracy: 0.7029 - val_loss: 2.2102 - val_top_k_categorical
_accuracy: 0.7198 - lr: 0.0049
Epoch 4/200
312/312 [==============================] - ETA: 0s - loss: 2.2736 - top_k_categorical_accuracy: 0.7196
WARNING:tensorflow:Early stopping conditioned on metric `val_accuracy` which is not available. Available metrics are: loss,top_k_categorical_accuracy,v
al_loss,val_top_k_categorical_accuracy,lr
312/312 [==============================] - 20s 65ms/step - loss: 2.2736 - top_k_categorical_accuracy: 0.7196 - val_loss: 2.6097 - val_top_k_categorical
_accuracy: 0.6179 - lr: 0.0049

WARNING:tensorflow:Early stopping conditioned on metric `val_accuracy` which is not available. Available metrics are: loss,top_k_categorical_accuracy,v
al_loss,val_top_k_categorical_accuracy,lr
312/312 [==============================] - 19s 61ms/step - loss: 0.4372 - top_k_categorical_accuracy: 0.9890 - val_loss: 0.9320 - val_top_k_categorical
_accuracy: 0.9544 - lr: 3.0000e-05
Epoch 196/200
312/312 [==============================] - ETA: 0s - loss: 0.4342 - top_k_categorical_accuracy: 0.9891
WARNING:tensorflow:Early stopping conditioned on metric `val_accuracy` which is not available. Available metrics are: loss,top_k_categorical_accuracy,v
al_loss,val_top_k_categorical_accuracy,lr
312/312 [==============================] - 19s 61ms/step - loss: 0.4342 - top_k_categorical_accuracy: 0.9891 - val_loss: 0.9339 - val_top_k_categorical
_accuracy: 0.9538 - lr: 1.2500e-04
Epoch 197/200
312/312 [==============================] - ETA: 0s - loss: 0.4347 - top_k_categorical_accuracy: 0.9887
WARNING:tensorflow:Early stopping conditioned on metric `val_accuracy` which is not available. Available metrics are: loss,top_k_categorical_accuracy,v
al_loss,val_top_k_categorical_accuracy,lr
312/312 [==============================] - 19s 61ms/step - loss: 0.4347 - top_k_categorical_accuracy: 0.9887 - val_loss: 0.9366 - val_top_k_categorical
_accuracy: 0.9540 - lr: 2.0000e-05
Epoch 198/200
312/312 [==============================] - ETA: 0s - loss: 0.4344 - top_k_categorical_accuracy: 0.9895
WARNING:tensorflow:Early stopping conditioned on metric `val_accuracy` which is not available. Available metrics are: loss,top_k_categorical_accuracy,v
al_loss,val_top_k_categorical_accuracy,lr
312/312 [==============================] - 19s 62ms/step - loss: 0.4344 - top_k_categorical_accuracy: 0.9895 - val_loss: 0.9328 - val_top_k_categorical
_accuracy: 0.9549 - lr: 7.5000e-05
Epoch 199/200
312/312 [==============================] - ETA: 0s - loss: 0.4313 - top_k_categorical_accuracy: 0.9891
WARNING:tensorflow:Early stopping conditioned on metric `val_accuracy` which is not available. Available metrics are: loss,top_k_categorical_accuracy,v
al_loss,val_top_k_categorical_accuracy,lr
312/312 [==============================] - 20s 64ms/step - loss: 0.4313 - top_k_categorical_accuracy: 0.9891 - val_loss: 0.9339 - val_top_k_categorical
_accuracy: 0.9547 - lr: 1.0000e-05
Epoch 200/200
312/312 [==============================] - ETA: 0s - loss: 0.4249 - top_k_categorical_accuracy: 0.9891
WARNING:tensorflow:Early stopping conditioned on metric `val_accuracy` which is not available. Available metrics are: loss,top_k_categorical_accuracy,v
al_loss,val_top_k_categorical_accuracy,lr
312/312 [==============================] - 19s 61ms/step - loss: 0.4249 - top_k_categorical_accuracy: 0.9891 - val_loss: 0.9339 - val_top_k_categorical
_accuracy: 0.9547 - lr: 2.5000e-05
Test loss: 0.9218876957893372
Test accuracy: 0.9524000287055969
```