Derek Lee — Deep Learning Fall 2020

Professor Curro — Assignment #4

```python
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import os
import pickle

C10 = True     # If true, training on CIFAR10; Otherwise, training on CIFAR100

# From:
# https://www.tensorflow.org/guide/gpu#limiting_gpu_memory_growth
gpus = tf.config.experimental.list_physical_devices('GPU')
if gpus:
    # Restrict TensorFlow to only allocate 1GB of memory on the first GPU
    try:
        tf.config.experimental.set_virtual_device_configuration(
            gpus[0],
            [tf.config.experimental.VirtualDeviceConfiguration(memory_limit=3072)
])
        logical_gpus = tf.config.experimental.list_logical_devices('GPU')
        print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical GPUs")
    except RuntimeError as e:
        # Virtual devices must be set before GPUs have been initialized
        print(e)


# Constants
NUM_EPOCHS = 160
BATCH_SIZE = 64
VALID_SIZE = 0.2    # Size of validation set
FIG_HEIGHT = 5      # Height of figure for final plot

# Model optimizer
OPTIMIZER = tf.keras.optimizers.Adam()

# CIFAR10
if C10:


    NUM_CLASSES = 10
    REGULARIZER = tf.keras.regularizers.l2( 0.001 )
    METRIC = tf.keras.metrics.SparseCategoricalAccuracy()
    PLOT_METRIC = 'sparse_categorical_accuracy'
    PLOT_LABEL = 'Accuracy'
    FIG_WIDTH = 10

# CIFAR100
else:

    NUM_CLASSES = 100
    REGULARIZER = tf.keras.regularizers.l2( 0.001 )
    METRIC = tf.keras.metrics.SparseTopKCategoricalAccuracy( k = 5 )    # Top 5 A
ccuracy
    PLOT_METRIC = 'sparse_top_k_categorical_accuracy'
    PLOT_LABEL = 'Top 5 Accuracy'
    FIG_WIDTH = 15


# Data processing from:
# https://www.cs.toronto.edu/~kriz/cifar.html
def unpickle( file ):

    with open( file, 'rb' ) as fo:
        dict = pickle.load(fo, encoding='bytes')

    return dict


# loadTrain - True for training, False for testing
# C10 - True for CIFAR-10, False for CIFAR-100
def loadCIFAR( loadTrain, C10 ):

    # CIFAR 10
    file10Train = [

        'cifar-10-batches-py/data_batch_1',
        'cifar-10-batches-py/data_batch_2',
        'cifar-10-batches-py/data_batch_3',
        'cifar-10-batches-py/data_batch_4',
        'cifar-10-batches-py/data_batch_5',

    ]
    file10Test = [ 'cifar-10-batches-py/test_batch' ]
```

```python
    # CIFAR 100
    file100Train = [ 'cifar-100-python/train' ]
    file100Test = [ 'cifar-100-python/test' ]

    # Dictionaries
    tempDict = {}

    if C10:
        finalDict = { b'data': [], b'labels': [] }

        if loadTrain:
            fileList = file10Train
        else:
            fileList = file10Test

    else:
        finalDict = { b'data': [], b'fine_labels': [] }

        if loadTrain:
            fileList = file100Train
        else:
            fileList = file100Test

    # Get keys representing data and labels
    dictData = list( finalDict.keys() )[0]
    dictLabels = list( finalDict.keys() )[1]

    # Load to dictionary
    for fileName in fileList:

        tempDict.update( unpickle( fileName ) )
        finalDict[ dictData ].extend( tempDict[ dictData ] )
        finalDict[ dictLabels ].extend( tempDict[ dictLabels ] )

    return finalDict


# Testing if images were properly loaded from:
# https://www.tensorflow.org/tutorials/images/cnn
def testLoad( img, label ):

    class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
                   'dog', 'frog', 'horse', 'ship', 'truck']

    plt.figure( figsize=(10,10) )
    for i in range(25):
        plt.subplot( 5, 5, i+1 )
        plt.xticks([])
        plt.yticks([])
        plt.grid( False )
        plt.imshow( img[i] )
        plt.xlabel( class_names[ label[i] ] )

    plt.show()


# Decreases learning rate at specific epochs from:
# https://towardsdatascience.com/understand-and-implement-resnet-50-with-
tensorflow-2-0-1190b9b52691
def lrdecay(epoch):

    lr = 1e-3
    if epoch > 180:
        lr *= 0.5e-3
    elif epoch > 160:
        lr *= 1e-3
    elif epoch > 120:
        lr *= 1e-2
    elif epoch > 80:
        lr *= 1e-1

    return lr


# Architecture based on Full Pre-Activation from:
# https://arxiv.org/pdf/1603.05027.pdf
# Implementation inspired by:
# https://www.tensorflow.org/tutorials/customization/custom_layers
class identityBlock( tf.keras.Model ):

    def __init__( self, filters ):

        super( identityBlock, self ).__init__( name = '' )
        f1, f2 = filters
        k = 3   # Kernel size

        self.conv2a = tf.keras.layers.Conv2D( f1, kernel_size = (1, 1), strides =
 (1, 1), padding = 'valid', kernel_regularizer = REGULARIZER )
        self.bn2a = tf.keras.layers.BatchNormalization()
        self.conv2b = tf.keras.layers.Conv2D( f1, kernel_size = (k, k), strides =
 (1, 1), padding = 'same', kernel_regularizer = REGULARIZER )
        self.bn2b = tf.keras.layers.BatchNormalization()

        self.conv2c = tf.keras.layers.Conv2D( f2, kernel_size = (1, 1), strides =
 (1, 1), padding = 'valid', kernel_regularizer = REGULARIZER )
        self.bn2c = tf.keras.layers.BatchNormalization()


    def call( self, inputTensor, training = False ):

        x = inputTensor

        # Block 1
        x = self.bn2a( x, training = training )
        x = tf.nn.leaky_relu( x )
        x = self.conv2a( x )

        # Block 2
        x = self.bn2b( x, training = training )
        x = tf.nn.leaky_relu( x )
        x = self.conv2b( x )

        # Block 3
        x = self.bn2c( x, training = training )
        x = tf.nn.relu( x )
        x = self.conv2c( x )

        # Output
        x += inputTensor

        return x


class convBlock( tf.keras.Model ):

    def __init__( self, filters, s ):

        super( convBlock, self ).__init__( name = '' )
        f1, f2 = filters
        k = 3   # Kernel size

        self.conv2a = tf.keras.layers.Conv2D( f1, kernel_size = (1, 1), strides =
 (s, s), padding = 'valid', kernel_regularizer = REGULARIZER )
        self.bn2a = tf.keras.layers.BatchNormalization()
        self.conv2b = tf.keras.layers.Conv2D( f1, kernel_size = (k, k), strides =
 (1, 1), padding = 'same', kernel_regularizer = REGULARIZER )
        self.bn2b = tf.keras.layers.BatchNormalization()

        self.conv2c = tf.keras.layers.Conv2D( f2, kernel_size = (1, 1), strides =
 (1, 1), padding = 'valid', kernel_regularizer = REGULARIZER )
        self.bn2c = tf.keras.layers.BatchNormalization()

        self.conv2Shortcut = tf.keras.layers.Conv2D( f2, kernel_size = (1, 1), st
rides = (s, s), padding = 'valid', kernel_regularizer = REGULARIZER )
        self.bn2Shortcut = tf.keras.layers.BatchNormalization()


    def call( self, inputTensor, training = False ):

        x = inputTensor
        xShort = inputTensor

        # Block 1
        x = self.bn2a( x, training = training )
        x = tf.nn.leaky_relu( x )
        x = self.conv2a( x )

        # Block 2
        x = self.bn2b( x, training = training )
        x = tf.nn.leaky_relu( x )
        x = self.conv2b( x )

        # Block 3
        x = self.bn2c( x, training = training )
        x = tf.nn.relu( x )
        x = self.conv2c( x )

        # Shortcut
        xShort = self.bn2Shortcut( xShort, training = training )
        xShort = tf.nn.relu( xShort )
        xShort = self.conv2Shortcut( xShort )

        # Output
        x += xShort

        return x
```

```
# Module containing Image Classification model
# Architecture based on:
# https://towardsdatascience.com/understand-and-implement-resnet-50-with-
tensorflow-2-0-1190b9b52691
class imgClassMod( tf.Module ):

    def __init__( self ):

        self.model = tf.keras.models.Sequential()

        # Stem Layer
        self.model.add( tf.keras.layers.ZeroPadding2D( (3, 3) ) )
        self.model.add( tf.keras.layers.Conv2D( 64, (7, 7), strides = (2, 2) ) )
        self.model.add( tf.keras.layers.BatchNormalization() )
        self.model.add( tf.keras.layers.ReLU() )
        self.model.add( tf.keras.layers.MaxPooling2D( (3, 3), strides = (2, 2) )
)

        # Hidden Layers
        # Stage 1
        self.model.add( convBlock( filters = [ 64, 256 ], s = 1 ) )
        for _ in range(2):
            self.model.add( identityBlock( filters = [ 64, 256 ] ) )

        # Stage 2
        self.model.add( convBlock( filters = [ 128, 512 ], s = 2 ) )
        for _ in range(3):
            self.model.add( identityBlock( filters = [ 128, 512 ] ) )

        if not C10:
            self.model.add( tf.keras.layers.Dropout( 0.15 ) )

        # Stage 3
        self.model.add( convBlock( filters = [ 256, 1024 ], s = 2 ) )
        for _ in range(5):
            self.model.add( identityBlock( filters = [ 256, 1024 ] ) )

        if not C10:
            self.model.add( tf.keras.layers.Dropout( 0.2 ) )

        # Stage 4
        self.model.add( convBlock( filters = [ 512, 2048 ], s = 2 ) )
        for _ in range(2):
            self.model.add( identityBlock( filters = [ 512, 2048 ] ) )
        # Pooling
        self.model.add( tf.keras.layers.AveragePooling2D( (2, 2), padding = 'same
' ) )

        # Output
        self.model.add( tf.keras.layers.Flatten() )

        if not C10:
            self.model.add( tf.keras.layers.Dropout( 0.2 ) )

        self.model.add( tf.keras.layers.Dense( NUM_CLASSES, activation = 'softmax
', kernel_initializer = 'he_normal' ) )


    def train( self, train, valid, trainSteps, validSteps ):

        self.lrdecay = tf.keras.callbacks.LearningRateScheduler(lrdecay) # Learni
ng rate decay
        self.model.compile( loss = tf.keras.losses.SparseCategoricalCrossentropy(
),
                            optimizer = OPTIMIZER, metrics = METRIC )
        self.history = self.model.fit( train, epochs = NUM_EPOCHS,
                    steps_per_epoch = trainSteps, validation_steps = validSteps,
                    validation_data = valid, batch_size = BATCH_SIZE, callbacks =
 [ self.lrdecay ] )


    def test( self, testImg, testLabel ):

        self.model.evaluate( testImg, testLabel )


    # Plots accuracy over time
    def plotAccuracy( self ):

        plt.figure( figsize = ( FIG_WIDTH, FIG_HEIGHT ) )
        plt.plot( self.history.history[ PLOT_METRIC ] )
        plt.plot( self.history.history[ 'val_' + PLOT_METRIC] )
        plt.title( 'Model ' + PLOT_LABEL )
        plt.xlabel( 'Epochs' )
        plt.ylabel( PLOT_LABEL, rotation = 'horizontal', ha = 'right' )
        plt.legend( [ 'Train', 'Valid' ], loc = 'upper left' )
        plt.show()
```

```
def main():

    # Load data
    dictCIFARTrain = loadCIFAR( True, C10 )
    dictCIFARTest = loadCIFAR( False, C10 )

    # Get keys
    dictData = list( dictCIFARTrain.keys() )[0]
    dictLabels = list( dictCIFARTrain.keys() )[1]

    img = dictCIFARTrain[ dictData ]
    label = dictCIFARTrain[ dictLabels ]
    testImg = dictCIFARTest[ dictData ]
    testLabel = dictCIFARTest[ dictLabels ]

    # Reshapes each image into 32x32 and 3 channels ( RGB )
    img = np.reshape( img, [ -1, 32, 32, 3 ], order = 'F' )
    testImg = np.reshape( testImg, [ -1, 32, 32, 3 ], order = 'F' )

    # Train / Valid Split
    trainImg, validImg, trainLabel, validLabel = train_test_split( img, label, te
st_size = VALID_SIZE )

    # Rotate, normalize, and convert to tensor
    trainImg = tf.convert_to_tensor( tf.image.rot90( trainImg, k=3 ) / 255, dtype
=tf.float32 )
    trainLabel = tf.convert_to_tensor( trainLabel )
    validImg = tf.convert_to_tensor( tf.image.rot90( validImg, k=3 ) / 255, dtype
=tf.float32 )
    validLabel = tf.convert_to_tensor( validLabel )
    testImg = tf.convert_to_tensor( tf.image.rot90( testImg, k=3 ) / 255, dtype=t
f.float32 )
    testLabel = tf.convert_to_tensor( testLabel )

    # Check if image was loaded properly
    #testLoad( trainImg, trainLabel )

    # Data Augmentation
    dataGenTrain = tf.keras.preprocessing.image.ImageDataGenerator(zoom_range=0.2
, width_shift_range=0.15, height_shift_range = 0.15, horizontal_flip=True)
    dataGenValid = tf.keras.preprocessing.image.ImageDataGenerator()
    trainSet = dataGenTrain.flow(trainImg, trainLabel, batch_size = BATCH_SIZE)
    validSet = dataGenValid.flow(validImg, validLabel, batch_size = BATCH_SIZE)
    model = imgClassMod()
    model.train( trainSet, validSet, trainImg.shape[0] / BATCH_SIZE, validImg.sha
pe[0] / BATCH_SIZE )
    model.test( testImg, testLabel )
    model.plotAccuracy()


main()
```

<u>CIFAR10</u>

I initially looked at TResNet, as it achieved a 99% accuracy on CIFAR10, but I decided not to use it because I didn't completely understand the Squeeze and Excitation Blocks (meaning I would essentially copy/paste code) and because the time required to train TResNet-M is almost 24 hours (according to the paper), which I considered impractical.

I achieved an 83.4% accuracy. This definitely is not state-of-the-art, but I believe it is reasonable considering my knowledge and the time limit set by the assignment. I could have used a more complex model, but I deliberately did not implement a model that took more than a few hours to train. My model took a little over an hour to train, which was short enough to allow me to experiment and improve it.

I used a combination of ResNet50 and Full Pre-Activation (from the paper assigned for reading this week). I tried using the regular ResNet50, but I found adding Full Pre-Activation gave better results. It converged faster, and slightly higher than the regular ResNet50.

I used L2 Regularization. On my initial attempts, I used the default value of 0.01 for my penalization parameter. This proved to be too large and caused my network to converge to around 65% accuracy. On subsequent attempts, I reduced it to 0.001.

I used Data Augmentation on the images, which I found helped with combatting overfitting. I also added a decay in the learning rate, which gave more stable learning near the end and reduced the chance that I would get a random drop in accuracy on the last iteration.

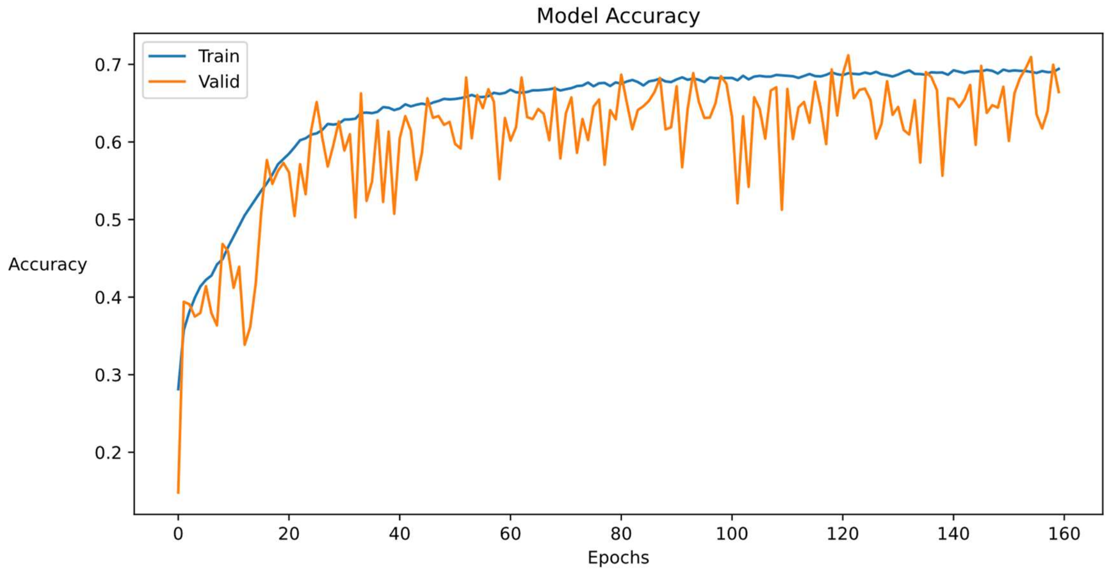


Figure 1: Example plot for the training and validation accuracy on CIFAR10 using ResNet50 with Full Pre-Activation during training with L2 Parameter = 0.01

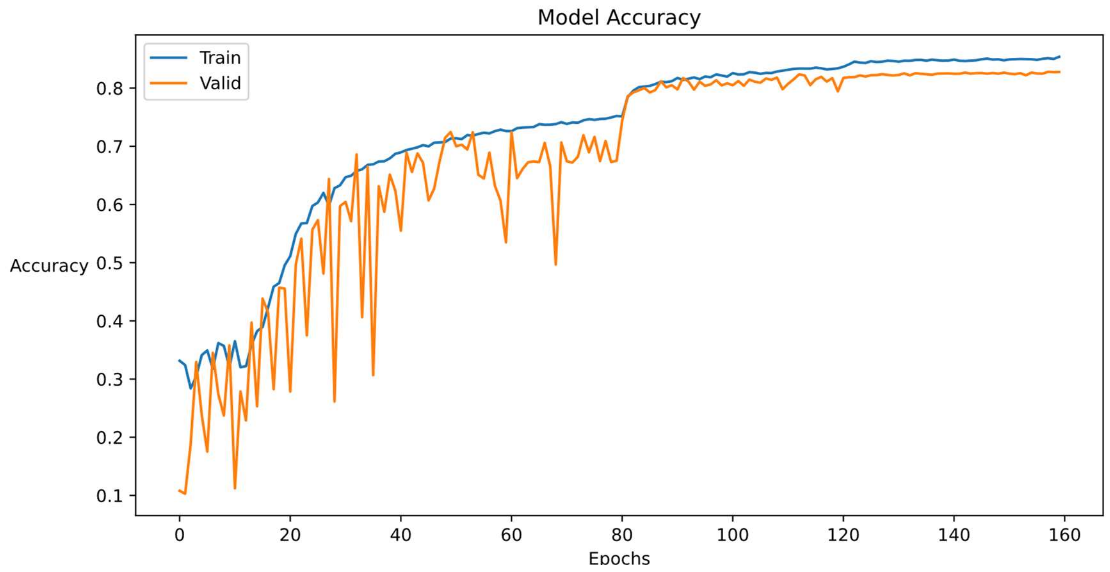
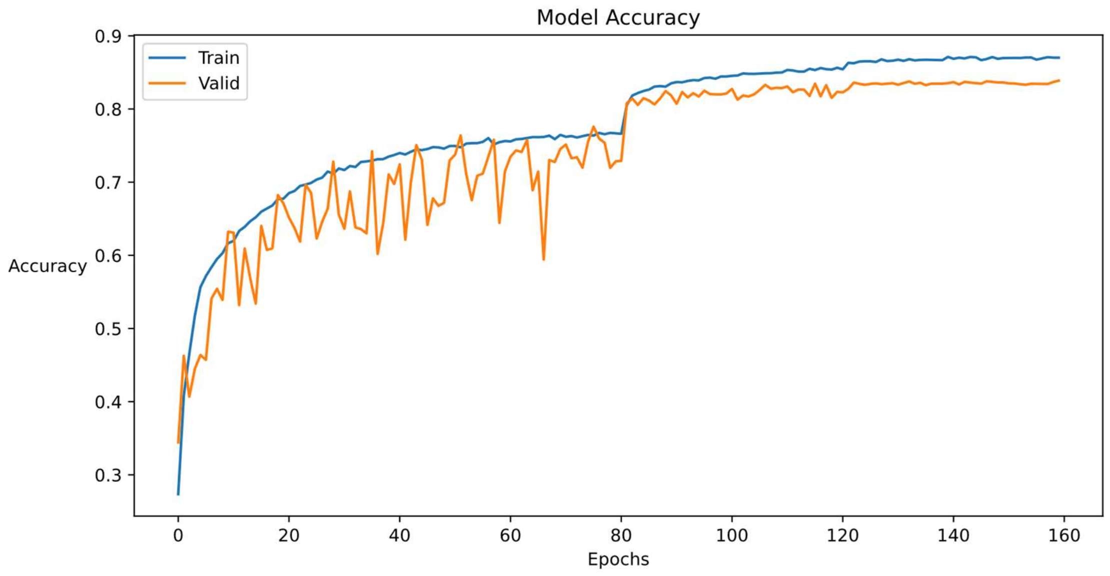


Figure 2: Example plot for the training and validation accuracy on CIFAR10 using ResNet50 during training with L2 Parameter = 0.001

```
313/313 [==============================] - 4s 11ms/step - loss: 0.6549 - accuracy: 0.8170
```

Figure 3: Final accuracy on the CIFAR10 test set using ResNet50 with L2 Parameter = 0.001



```
313/313 [==============================] - 4s 12ms/step - loss: 0.6280 - accuracy: 
0.8340
```

Figure 5: Final accuracy on the CIFAR10 test set using ResNet50 with Full Pre-Activation with L2 Parameter = 0.001

<u>CIFAR100</u>

I used the same program for CIFAR100. On my first attempt, I had an almost 10% discrepancy between my training and validation accuracies. My test accuracy was 79%, barely under the target. On my next attempt, I added a 20% dropout layer right before the output layer. This reduced the discrepancy to about 7%. I barely got over 80%, so I decided to try another modification. I added a 15% dropout layer after Stage 2 and a 20% dropout layer after Stage 3 to further counter overfitting. The discrepancy dropped to around 5% and gave me a slightly better validation accuracy.

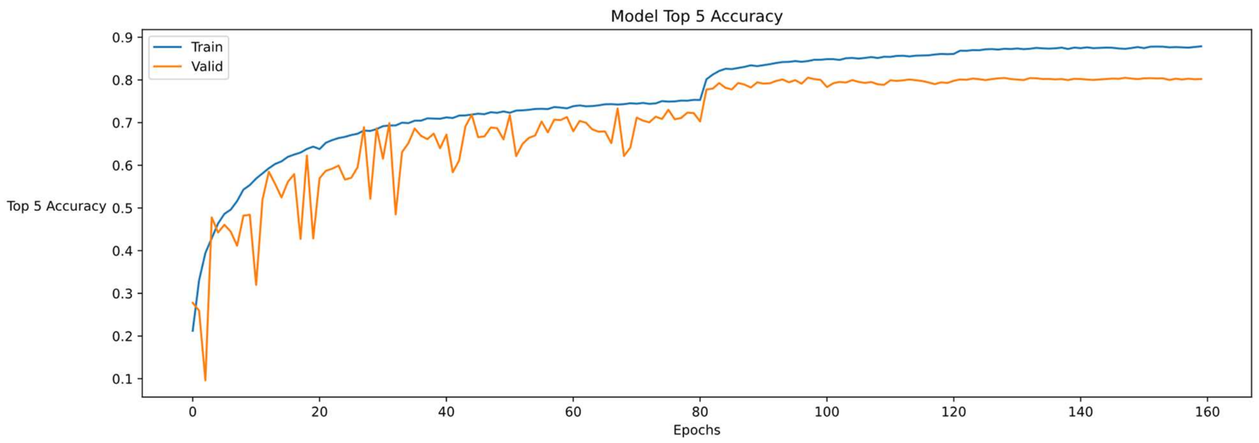


Figure 6: Example plot for the training and validation accuracy on CIFAR100 using ResNet50 with Full Pre-Activation with L2 Parameter = 0.001 and Dropout = 0.2

```
313/313 [==============================] - 4s 11ms/step - loss: 2.1422 -
sparse_top_k_categorical_accuracy: 0.8004
```

Figure 7: Final accuracy on the CIFAR100 test set using ResNet50 with Full Pre-Activation with L2 Parameter = 0.001 and Dropout = 0.2

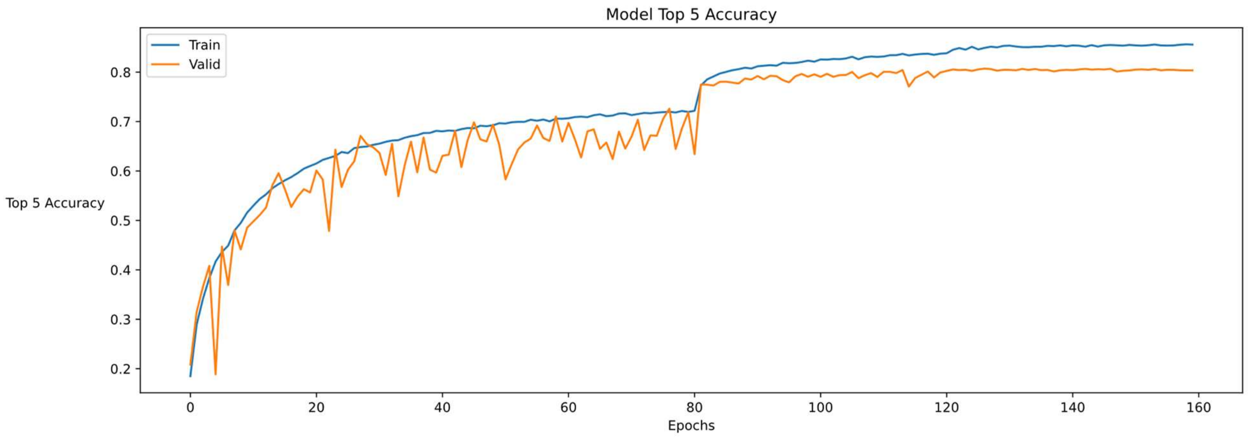


Figure 8: Example plot for the training and validation accuracy on CIFAR100 using ResNet50 with Full Pre-Activation with L2 Parameter = 0.001 and Dropout = 0.15, 0.2, 0.2

```
313/313 [==============================] - 4s 13ms/step - loss: 2.0996 -
sparse_top_k_categorical_accuracy: 0.8026
```

Figure 9: Final accuracy on the CIFAR100 test set using ResNet50 with Full Pre-Activation with L2 Parameter = 0.001 and Dropout = 0.15, 0.2, 0.2