Identifying Monkey Species with a Custom Convolutional Neural Network

A Practice in Understanding Learning Machines, Pythons, Monkeys, and Ourselves
By Conner Borkowski and Todd Bean

Table of Contents

1. Introduction

We completed this project based on [this Kaggle dataset] which contains images of monkeys categorized into 10 different species. The authors of this data set created it for the development of a "fine-grain" classification test. We first aimed to implement a custom CNN (Convolutional Neural Network) to classify these images of monkeys into one of ten species groups. While developing the custom CNN we ran into roadblocks, like overfitting and long runtimes, but through overcoming them we were able to better understand how a CNN works.

To aid with analysis we visualized feature maps at different layer levels, so that we could observe how the network was constructing the images. After this we would look for new variations of a CNN that we could implement, which resulted in us developing a transfer learning system much like many of the participants in this Kaggle task.

2. Importing the Dataset and Libraries

While this first step seems simple, it is integral to the functioning of the rest of the code. We are importing the code necessary for manipulating and understanding images, as well as an optimizer, Adam. We also imported the functions that become part of the CNN, like the convolution and pooling functions.

In the format of a Google Colab Python Notebook we must use Google Drive to store our Dataset. This makes some things easier as the functions don't need to be downloaded onto a computer, and are instead stored on Google's servers. When using this Notebook, given that it has been shared with you, you can create a shortcut to project files and place it in your drive, where you access it through its pathname as if it were a normal file. Additionally you can copy the files from the files tab on the left of the Google Colab Menu.

In [ ]:

```
from google.colab import drive
drive.mount('/content/gdrive')
```

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

In [ ]:

```
import numpy as np                #numpy for linear algebra
import pandas as pd               #pandas for dataset creation
import matplotlib.pyplot as plt #matplotlib for plotting
import tensorflow as tf           #for CNN creation
import os                         #for file manipulation
import cv2                        #for understaindig and showing images
import imgaug.augmenters as iaa #for modifying images and hterby augmenting data
import random                     #random number generation

from pathlib import Path          #for getting data from datapaths
from keras.models import Sequential, Model, load_model   #for defining our CNN
from keras.layers import Dense, Conv2D, MaxPooling2D, BatchNormalization, GlobalAveragePoo
ling2D, Dropout, Flatten #for creating our CNN layers
from keras.optimizers import Adam  #for CNN optimization
from keras.callbacks import EarlyStopping, ModelCheckpoint #to stop the CNN if it gets smar
t enough
from keras.utils import to_categorical   #for dataset creation
from keras.preprocessing.image import load_img,img_to_array
from keras.applications.vgg16 import VGG16

from google.colab.patches import cv2_imshow
from skimage import io
from PIL import Image

from matplotlib.pyplot import figure

%matplotlib inline
```

3. Setting Seeds for Reproducibility

Machine learning sounds incredibly rigorous and methodical. However, it is often a fickle beast and therefore requires some code to maintain results through multiple trials and experiments. This section is dedicated to sowing the seeds of success for our young algorithm and making sure we can recreate trials in the future. In order to reproduce a particular run, all we need to do is input the same seed.

In [ ]:

```
seed=301
tf.random.set_seed(seed)
np.random.seed(seed)
os.environ['PYTHONHASHSEED'] = '0'
```

4. Collecting Data

This data set is formatted in a way that makes sense for humans, with each species of monkey going in one folder. Unfortunately, this makes less sense for CNN applications which would prefer all the images have their own label and all images were in the same directory. We will collect the data, and then modify it so that the data will be easier for the program to access and sort.

In [ ]:

```
trainData = Path('gdrive/MyDrive/MonkeyProject/Kaggle/training/')
validData = Path('gdrive/MyDrive/MonkeyProject/Kaggle/validation/')
labelPath = Path('gdrive/MyDrive/MonkeyProject/Kaggle/monkey_labels.txt')
```

In [ ]:

```
labelInfo = []

lines = labelPath.read_text().strip().splitlines()[1:]
#this is to get the lines from our labels .txt file and help us use it for later
for line in lines:
    line = line.split(',')
    line = [x.strip(' \n\t\r') for x in line]
    line[3], line[4] = int(line[3]), int(line[4])
    line = tuple(line)
    labelInfo.append(line)
```

```
labelInfo = pd.DataFrame(labelInfo, columns=['Label', 'Latin Name', 'Common Name', 'Train
Images', 'Validation Images'], index=None)
labelInfo.head(10)
```

Out[ ]:

	Label	Latin Name	Common Name	Train Images	Validation Images
0	n0	alouatta_palliata	mantled_howler	131	26
1	n1	erythrocebus_patas	patas_monkey	139	28
2	n2	cacajao_calvus	bald_uakari	137	27
3	n3	macaca_fuscata	japanese_macaque	152	30
4	n4	cebuella_pygmea	pygmy_marmoset	131	26
5	n5	cebus_capucinus	white_headed_capuchin	141	28
6	n6	mico_argentatus	silvery_marmoset	132	26
7	n7	saimiri_sciureus	common_squirrel_monkey	142	28
8	n8	aotus_nigriceps	black_headed_night_monkey	133	27
9	n9	trachypithecus_johnii	nilgiri_langur	132	26

Here we can the ten species of monkeys that are contained within our digital image park. We can also see how many images we have for training as well as training validation. The validation images are used to test the accuracy of the model in predicting the label of images it was not trained on which is also know as generalizing. Thisis the validation step and it is an important part of machine learning. Without it, our system could learn too well from the images it trains on. This would result in "overfitting" or the network not being able to identify new examples of monkeys from one of the trained species. This happens when the networks model fixates on increasing accuracy relative to the trained images. New images then appear to be too different from the network's ideal model to be classified correctly.

Overfitting is evident when the training accuracy of a model goes towards 100% but the validation accuracy of the same model plateaus as a constant less than 100%. This is a key feature that we noticed in our original models and we will later discuss possible ways to fix this issue.

In [ ]:

```
randomDir = random.choice([
    x for x in os.listdir(trainData)
    if os.path.isdir(os.path.join(trainData, x))
])
randomFile = random.choice([
    x for x in os.listdir(os.path.join(trainData, randomDir))
    if os.path.isfile(os.path.join(os.path.join(trainData, randomDir), x))
])
print("Image Path: " + os.path.join(os.path.join(trainData, randomDir), randomFile))
image = cv2.cvtColor(io.imread(os.path.join(os.path.join(trainData, randomDir), randomFile
)), cv2.COLOR_BGR2RGB)
cv2_imshow(image)
print("Image Shape: ")
print(image.shape)
```

```
Image Path: gdrive/MyDrive/MonkeyProject/Kaggle/training/n6/n6132.jpg
```

```
Image Shape:
(1329, 2000, 3)
```

This is an example image from our dataset, and is undoubtedly very cute. We have also shown the shape of the image to create a better understanding of our data. You can rerun this block of code to see a different monkey, which will also undoubtedly be very cute. With the shape of the images in mind, we can begin to prepare the data for our algorithm. First, we must assign labels that are easy to reference for our training, so each shortened label will be replaced with a corresponding number value.

In [ ]:

```
labelToNumTranslate = {'n0':0, 'n1':1, 'n2':2, 'n3':3, 'n4':4, 'n5':5, 'n6':6, 'n7':7, 'n8
':8, 'n9':9}
#we need to create simple associations for our CNN
namesDict = dict(zip(labelToNumTranslate.values(), labelInfo["Common Name"]))
```

Now that we have created our new labels and assigned them to the labels that are used in the dataset, we can sort our images and change their labels. Additonally, we have put them in containers that can be easily indexed by our model.

In [ ]:

```
trainDataContain = []
validDataContain = []

#for loops to label images and put in containers for the Machine to understand
for folder in os.listdir(trainData):
  imgPath = trainData / folder
  imgs = sorted(imgPath.glob('*.jpg'))
  for imgName in imgs:
    trainDataContain.append((str(imgName), labelToNumTranslate[folder]))
for folder in os.listdir(validData):
  imgPath = validData / folder
  imgs = sorted(imgPath.glob('*.jpg'))
  for img_name in imgs:
    validDataContain.append((str(img_name), labelToNumTranslate[folder]))

#useing pandas to help standardize and normalize what we have
trainDataContain = pd.DataFrame(trainDataContain, columns=['image', 'label'], index=None)
validDataContain = pd.DataFrame(validDataContain, columns=['image', 'label'], index=None)
trainDataContain = trainDataContain.sample(frac=1.).reset_index(drop=True)
validDataContain = validDataContain.sample(frac=1.).reset_index(drop=True)

#checking how many samples we have
print("Samples For Training: ", len(trainDataContain))
print("Samples For Validation: ", len(validDataContain))
```

```
Samples For Training:  1096
Samples For Validation:  272
```

We should take note of how many data points we have, only 1096 training images and 272 validation images. There

We should take note of how many data points we have, only 1096 training images and 272 validation images. There is no written rule for how many images you must have to create a quality CNN, however, training datasets often have tens of thousands of datapoints or more for image classification, according to some. Having such a small number of datapoints or images makes the challenge of avoiding overfitting even more difficult, as outliers in the images could alter the model significantly more than in a larger batch.

In [ ]:

```
#sample image dimensions
imgRows = 224
imgCols = 224
imgChannels = 3
#learning dimensions
batchSize = 32  #num of samples per pass
numOfClasses = 10  #number of monkey species
```

Our last data collection step is to set some constants that we will use for our model. The image rows and columns reference the height and width of our images when we put them into our model.

The image channels reference one of the three color channels of our images, red, green, and blue. The batch size is the number of images per pass of our model or the number of images we use at a time. Finally, we set our number of classes/labels which is the number of types of monkeys we have.

With our data prepared, we are ready to create and feed our model. However, we still have some concerns about overfitting so in the next section we will take some steps to try to avoid it.

5. Augmenting Data

Humans, Snakes, Monkeys, and CNN alike benefit from an appropriate amount of food. Luckily, our CNN likes to eat monkeys. In the case of a CNN the more food, or data, we have, the better. Unfortunately, this particular dataset is rather small. To account for this, and to help with overfitting, we modify images from our dataset to create a larger number of data points. Think of this as an image that is warped or has had a filter put on it. We have been experimenting to see which filters are the best to create more data without hurting accuracy.

We found that greater rotations of the images tended to hurt accuracy as most of the monkeys are oriented upwards, despite spending much time upside down and otherwise in nature. Smaller rotations, on the other hand, mimic the slight changes in angle that would come from taking another picture of the monkey at a slightly different angle. The warping then accounts for monkeys that may be of different body shapes, either fatter or otherwise.

Larger multipliers and filters that change color would have provided more images, but the coloration and shape of each species of monkey is very important to identify it, and we felt that these augmentations would hurt accuracy. In short, we need to feed it nutritious data, not junk food data.

In [ ]:

```
#modifying for increase number of samples when learning
seq = iaa.OneOf([iaa.Fliplr(), iaa.Affine(rotate=20), iaa.Multiply((1.2, 1.5))])
```

6. Generating Data

Now that we have both the proper libraries to learn from and some good brain food for our CNN, all that is left before we begin learning is to break our data into bite-size pieces that can be used to learn features of the monkey species. This code collects our data and organizes it into the containers we created earlier. It also modifies training data with the augmentation we created earlier. With this step our data is primed and ready to be trained on by our network.

In [ ]:

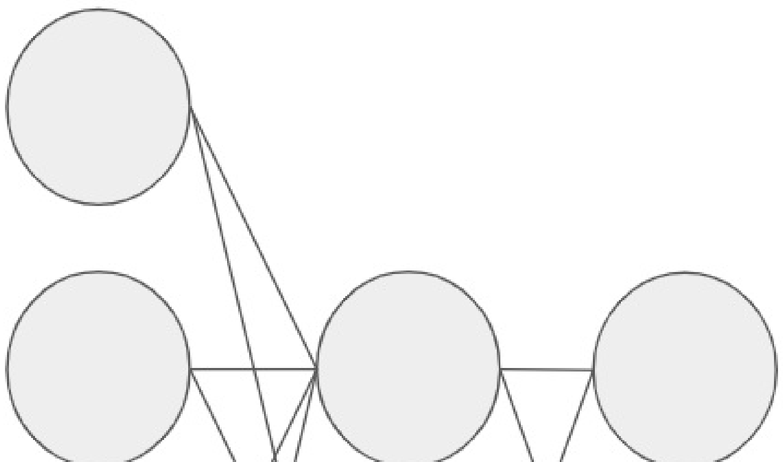
```
def dataSetter(data, batchSize, valData=False):
    """
   This function takes data folders and creates bathces of data to be used by the
   CNN
   Parameters
   ----------
   data : pandas dataframe
```

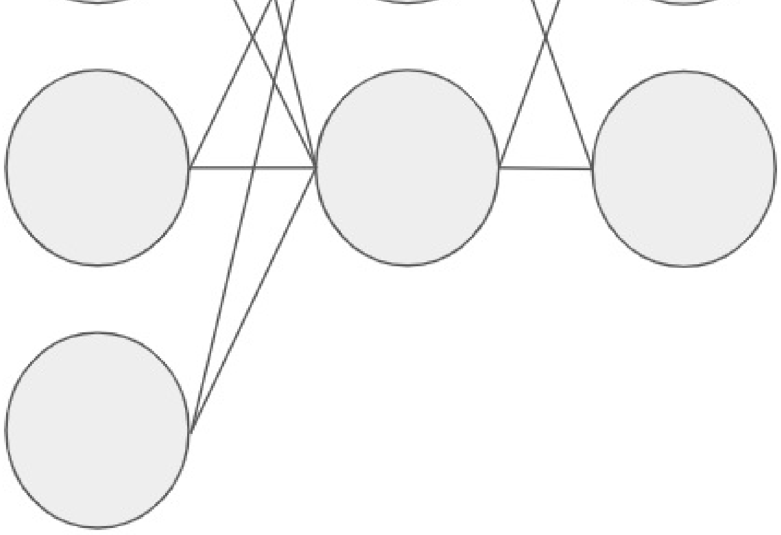
```
    dataframe containing images with labels
  batchSize : int
    indicating the number of samples in each group or batch
  valData : bool
    true if it is validation data false if training data
  """
  #calculating the necesary values for dataset creation
  batchIndeces = np.arange(len(data))
  numOfBatches = int(np.ceil(len(data)/batchSize))
  batchData = np.zeros((batchSize, imgRows, imgCols, imgChannels), dtype=np.float32)
  batchLabels = np.zeros((batchSize, numOfClasses), dtype=np.float32)

  while True:
    #shuffle training data
    if not valData:
      np.random.shuffle(batchIndeces)

    #find next batch indeces
    for i in range(numOfBatches):
      batchIndicesNext = batchIndeces[i*batchSize:(i+1)*batchSize]

      #correct cv2 color order and lable data
      for j, idx in enumerate(batchIndicesNext):
        img = cv2.imread(data.iloc[idx]["image"])
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        label = data.iloc[idx]["label"]

        #augment image if its training data
        if not valData:
          img = seq.augment_image(img)

        #make the image the right size and add it to the proper batch
        img = cv2.resize(img, (imgRows, imgCols)).astype(np.float32)
        batchData[j] = img
        batchLabels[j] = to_categorical(label,num_classes=numOfClasses)

      #return batches
      yield batchData, batchLabels
```

7. The Math

With our data prepared, we are ready to create and run our model. However, we first need to understand the math behind it. The two most important steps of a CNN that occur are the convolution which is both the namesake of this algorithm type and the way that a CNN distills an image into trainable features. The second is gradient descent which is a multidimensional slope analysis that allows the model to improve itself after training and testing on the validation data.

Given those parts, we must cover the basic concept and form of a CNN. A CNN works by making each pixel a node or neuron in the network. Each layer of a CNN then creates a new set of neurons that are linked to the previous layer with a weight. This is modeled after how neurons are connected in living things and is very complex. However, we can show a simple example of how layers are connected. Each circle represents a pixel and each line a connection with a specific weight.





If each input is represented in the first layer of four, each classification or label is represented in the last layer of two. Therefore, by determining the weights of each connection, we can create a path that leads to similar inputs or features being mapped to a consistent output.

7a. Kernel Convolution

We know that convolution can be expressed with the following formula.

$$y[n] = (x * h)[n] = \sum_k x[k]h[n-k]$$

This leaves us with the question of how we apply this to an image? It helps to consider an image to be a two dimensional matrix with the "brightness" value populating it. Through this lense, we can begin to see convolution in a new light. Much like the flip and slide method of discrete convolution, we can pass a small matrix over our image and get a new sampled signal. This can be described by the formula below.

$$y[n,m] = (x * h)[n,m] = \sum_j \sum_k x[j,k]h[n-j,m-k]$$

We can also demonstrate this graphically. In the context of CNNs the result of convoluting an image with a kernel is a feature map. It is called this because it is what holds the features of the image for a CNN to make predictions from.

```
        Image            Kernel       Feature Map
    [ 0 0 0 0 0 0 ]
    [ 0 0 0 0 0 0 ]    [ 1 3 1 ]
    [ 1 1 1 1 1 1 ] *  [ 0 0 0 ] = ?
    [ 1 1 1 1 1 1 ]    [-1-3-1 ]
    [ 0 0 0 0 0 0 ]
    [ 0 0 0 0 0 0 ]

    [ 0 0 0 0 0 0 ]
    [ 0 0 0 0 0 0 ]    [ 1 3 1 ]    1*0 + 3*0 + 1*0 +     [ 0         ]
    [ 1 1 1 1 1 1 ] *  [ 0 0 0 ] = 0*0 + 0*0 + 0*0 +   = [           ]
    [ 1 1 1 1 1 1 ]    [-1-3-1 ]   -1*1 +-3*1 +-1*1       [           ]
    [ 0 0 0 0 0 0 ]                                       [           ]
    [ 0 0 0 0 0 0 ]

    [ 0 0 0 0 0 0 ]
    [ 0 0 0 0 0 0 ]    [ 1 3 1 ]    [-5-5-5-5 ]
    [ 1 1 1 1 1 1 ] *  [ 0 0 0 ] = [-5-5-5-5 ]
    [ 1 1 1 1 1 1 ]    [-1-3-1 ]    [ 5 5 5 5 ]
    [ 0 0 0 0 0 0 ]                 [ 5 5 5 5 ]
    [ 0 0 0 0 0 0 ]
```

Kernel Convolution is done for each Convolutional layer and for each image passed through that layer. The summation of the sample into one value condenses the image. As we progress through the layers, the images are reduced in size and the kernel convolution creates a feature map that represents more abstract features. For example, the first layer will identify edges and lines, the next layer may create a feature map that identifies individual body parts of the monkey such as the eyes, and a later feature map will distinguish the whole shape of the monkey's body. This slide method is similar to how pooling layers work as well, but with a different operation than convolution/multiplications. You can learn more about the math behind layers here.

7b. Gradient Descent

Once we have feature maps from the convolution, it may seem like we are home-free and we can tell you what kind of monkey is in an image from 1 hair. However, even with the feature maps, our model does not necessarily know how to connect the feature maps in a way that makes sense for labeling the monkeys' species.

This is where gradient descent comes in. When we build a CNN model we assign a loss function. This loss function is used to determine the effectiveness of the model based on the weights of the connection between each layer.

For a fucntion *f(a,b,c, ...)*, the gradient vector is defined as:

$$f(a,b,c,...) = \begin{bmatrix} \frac{\partial f}{\partial a} \\ \frac{\partial f}{\partial b} \\ \frac{\partial f}{\partial c} \\ \cdots \end{bmatrix}$$

This vector points us in the direction of the most rapid increase of the function *f* and it will eventually lead us to a peak local maximum of *f*. If we negate it and use it to direct us to a minimum. Because we are trying to minimize the loss, this is exactly what we want. In our case, our inputs are the weights of connections between layers and the function is our loss function which compares teh label the model determines and what the image is actually labeled.

The optimizer we use is slightly modified to avoid finding a local minimum. Gradient descent is used in a variety of ways and has many different applications. This resource is a great place to learn more about its CNN applications or watch this quick video.

8. The Model

With our data prepared and a good understanding of the math behind a CNN, we can now create our model. We began by adding convolutional layers to make sure we met the base requirements, and then we began to look at what makes a CNN achieve its goal. We found that pooling allows the image to be reduced to regular sizes and lets each layer determine features of unique resolution. Finally, we add Dropout, Flatten, and Dense layers. These are used to correct the dimensions of our layers, and add generalizability.

In [ ]:

```
#Sequential model type applies layers in order
model = Sequential()

#first block
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(imgRows,imgCo
ls,imgChannels)))
model.add(MaxPooling2D((2, 2), padding='same'))

#second block
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2), padding='same'))

#third block
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2), padding='same'))
```

```
#fourth block
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2), padding='same'))

#fifth block
model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2), padding='same'))

#sixth block
model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2), padding='same'))
model.add(Dropout(0.5))

#final layer group  flatten the data into fewer dimensions and then apply
# dense and dropout layers. Denses layers add parameters that the CNN can train
# on and droupout layers reduce the chance of overfitting and increase
# generalizability
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(numOfClasses, activation='softmax'))

#the compilation uses three arguments. Catagorical Cross Entropy is a comparison
# of loss between multiple labels and multiple groups. Adam is an optimization
# method that implements stochastic gradient descent Adam optimization is a
# stochastic gradient descent method
model.compile(loss="categorical_crossentropy",optimizer='adam', metrics="acc")
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 224, 224, 32)      896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 112, 112, 32)      0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 112, 112, 32)      9248
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 56, 56, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 56, 56, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 28, 28, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 28, 28, 64)        36928
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 14, 14, 128)       73856
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 7, 7, 128)         0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 7, 7, 128)         147584
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 4, 4, 128)         0
_________________________________________________________________
dropout (Dropout)            (None, 4, 4, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 512)               1049088
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                5130
=================================================================
Total params: 1,341,226
Trainable params: 1,341,226
Non-trainable params: 0
_________________________________________________________________
```

Here we can see the layers of our CNN as well as the dimensions of our data at each layer. We can see how the

Here we can see the layers of our CNN as well as the dimensions of our data at each layer. We can see how the layers reduce the height and width of the images but increase the number of channels. These channels do not represent color as in the original images, but rather they represent feature maps. The param column list how many parameters or weights are added by each layer.

This information was used to tune this to the best of our abilities after training and iterating through many attempts. Below, we run our model and can see the loss and accuracy for both the training data and the validation data for each epoch.

In [ ]:

```
trainDataSet = dataSetter(trainDataContain, batchSize)
validDataSet = dataSetter(validDataContain, batchSize, True)

numOfEpochs=100

trainingSteps = int(np.ceil(len(trainDataContain)/batchSize))
validationSteps = int(np.ceil(len(validDataContain)/batchSize))

es = EarlyStopping(patience=10, restore_best_weights=True)

callbackList=[es]

history = model.fit(trainDataSet,
                    validation_data = validDataSet,
                    epochs = numOfEpochs,
                    steps_per_epoch = trainingSteps,
                    validation_steps = validationSteps,
                    callbacks = callbackList)
```

```
Epoch 1/100
35/35 [==============================] - 373s 11s/step - loss: 10.6610 - acc: 0.0895 - val_
loss: 2.2749 - val_acc: 0.1458
Epoch 2/100
35/35 [==============================] - 124s 4s/step - loss: 2.2714 - acc: 0.1399 - val_lo
ss: 2.3027 - val_acc: 0.1493
Epoch 3/100
35/35 [==============================] - 128s 4s/step - loss: 2.2361 - acc: 0.1728 - val_lo
ss: 2.2450 - val_acc: 0.1875
Epoch 4/100
35/35 [==============================] - 127s 4s/step - loss: 2.2133 - acc: 0.1665 - val_lo
ss: 2.0874 - val_acc: 0.2465
Epoch 5/100
35/35 [==============================] - 127s 4s/step - loss: 2.0738 - acc: 0.2237 - val_lo
ss: 2.0653 - val_acc: 0.3333
Epoch 6/100
35/35 [==============================] - 126s 4s/step - loss: 2.0232 - acc: 0.2593 - val_lo
ss: 1.9176 - val_acc: 0.3229
Epoch 7/100
35/35 [==============================] - 123s 4s/step - loss: 1.8552 - acc: 0.3345 - val_lo
ss: 1.7398 - val_acc: 0.4271
Epoch 8/100
35/35 [==============================] - 130s 4s/step - loss: 1.7696 - acc: 0.3787 - val_lo
ss: 1.7675 - val_acc: 0.3542
Epoch 9/100
35/35 [==============================] - 124s 4s/step - loss: 1.7163 - acc: 0.4017 - val_lo
ss: 1.5877 - val_acc: 0.4306
Epoch 10/100
35/35 [==============================] - 125s 4s/step - loss: 1.6907 - acc: 0.3970 - val_lo
ss: 1.5852 - val_acc: 0.4375
Epoch 11/100
35/35 [==============================] - 125s 4s/step - loss: 1.5597 - acc: 0.4373 - val_lo
ss: 1.5425 - val_acc: 0.4479
Epoch 12/100
35/35 [==============================] - 123s 4s/step - loss: 1.4883 - acc: 0.4338 - val_lo
ss: 1.4700 - val_acc: 0.4965
Epoch 13/100
35/35 [==============================] - 123s 4s/step - loss: 1.4921 - acc: 0.4676 - val_lo
ss: 1.4714 - val_acc: 0.4792
Epoch 14/100
35/35 [==============================] - 123s 4s/step - loss: 1.3413 - acc: 0.5370 - val_lo
ss: 1.3679 - val_acc: 0.5417
Epoch 15/100
```

```
35/35 [==============================] - 124s 4s/step - loss: 1.2666 - acc: 0.5370 - val_lo
ss: 1.5549 - val_acc: 0.4965
Epoch 16/100
35/35 [==============================] - 125s 4s/step - loss: 1.2950 - acc: 0.5607 - val_lo
ss: 1.3162 - val_acc: 0.5764
Epoch 17/100
35/35 [==============================] - 126s 4s/step - loss: 1.2306 - acc: 0.5540 - val_lo
ss: 1.4522 - val_acc: 0.4861
Epoch 18/100
35/35 [==============================] - 124s 4s/step - loss: 1.1812 - acc: 0.6336 - val_lo
ss: 1.3401 - val_acc: 0.5104
Epoch 19/100
35/35 [==============================] - 124s 4s/step - loss: 1.0856 - acc: 0.6079 - val_lo
ss: 1.2122 - val_acc: 0.5868
Epoch 20/100
35/35 [==============================] - 125s 4s/step - loss: 0.9767 - acc: 0.6689 - val_lo
ss: 1.2327 - val_acc: 0.5972
Epoch 21/100
35/35 [==============================] - 123s 4s/step - loss: 0.9780 - acc: 0.6743 - val_lo
ss: 1.2071 - val_acc: 0.6042
Epoch 22/100
35/35 [==============================] - 124s 4s/step - loss: 0.9459 - acc: 0.6789 - val_lo
ss: 1.1622 - val_acc: 0.5694
Epoch 23/100
35/35 [==============================] - 123s 4s/step - loss: 0.9537 - acc: 0.6557 - val_lo
ss: 1.0460 - val_acc: 0.6493
Epoch 24/100
35/35 [==============================] - 124s 4s/step - loss: 0.7352 - acc: 0.7583 - val_lo
ss: 1.0791 - val_acc: 0.6701
Epoch 25/100
35/35 [==============================] - 124s 4s/step - loss: 0.7188 - acc: 0.7465 - val_lo
ss: 1.1325 - val_acc: 0.6215
Epoch 26/100
35/35 [==============================] - 127s 4s/step - loss: 0.7672 - acc: 0.7380 - val_lo
ss: 1.1711 - val_acc: 0.5938
Epoch 27/100
35/35 [==============================] - 125s 4s/step - loss: 0.7890 - acc: 0.7639 - val_lo
ss: 1.0020 - val_acc: 0.6632
Epoch 28/100
35/35 [==============================] - 131s 4s/step - loss: 0.6862 - acc: 0.7702 - val_lo
ss: 0.9871 - val_acc: 0.6840
Epoch 29/100
35/35 [==============================] - 125s 4s/step - loss: 0.6523 - acc: 0.7630 - val_lo
ss: 1.1607 - val_acc: 0.6285
Epoch 30/100
35/35 [==============================] - 125s 4s/step - loss: 0.5902 - acc: 0.8067 - val_lo
ss: 1.0751 - val_acc: 0.6424
Epoch 31/100
35/35 [==============================] - 124s 4s/step - loss: 0.6988 - acc: 0.7478 - val_lo
ss: 1.0272 - val_acc: 0.6597
Epoch 32/100
35/35 [==============================] - 125s 4s/step - loss: 0.5798 - acc: 0.8080 - val_lo
ss: 0.9441 - val_acc: 0.6701
Epoch 33/100
35/35 [==============================] - 122s 4s/step - loss: 0.5971 - acc: 0.8036 - val_lo
ss: 0.9888 - val_acc: 0.6910
Epoch 34/100
35/35 [==============================] - 123s 4s/step - loss: 0.4731 - acc: 0.8365 - val_lo
ss: 1.0016 - val_acc: 0.6562
Epoch 35/100
35/35 [==============================] - 123s 4s/step - loss: 0.4908 - acc: 0.8358 - val_lo
ss: 0.8942 - val_acc: 0.7326
Epoch 36/100
35/35 [==============================] - 121s 3s/step - loss: 0.4148 - acc: 0.8548 - val_lo
ss: 0.9578 - val_acc: 0.7049
Epoch 37/100
35/35 [==============================] - 122s 3s/step - loss: 0.3854 - acc: 0.8602 - val_lo
ss: 1.1641 - val_acc: 0.6493
Epoch 38/100
35/35 [==============================] - 122s 4s/step - loss: 0.4424 - acc: 0.8494 - val_lo
ss: 1.0095 - val_acc: 0.7083
Epoch 39/100
35/35 [==============================] - 122s 3s/step - loss: 0.3917 - acc: 0.8609 - val_lo
```

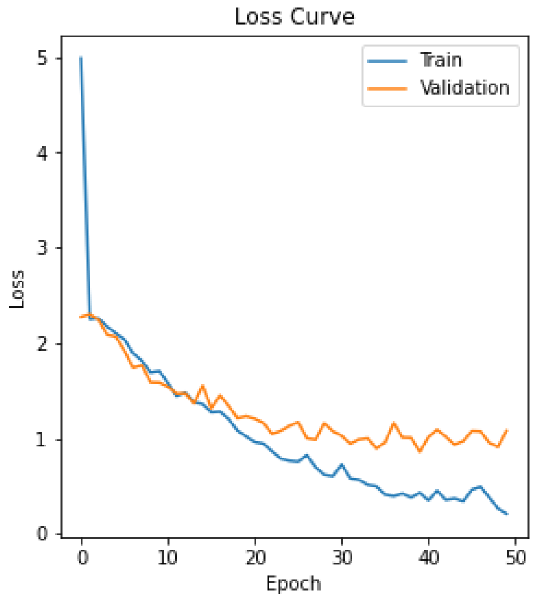
```
ss: 1.0063 - val_acc: 0.6840
Epoch 40/100
35/35 [==============================] - 122s 3s/step - loss: 0.4330 - acc: 0.8657 - val_lo
ss: 0.8585 - val_acc: 0.7396
Epoch 41/100
35/35 [==============================] - 123s 4s/step - loss: 0.3032 - acc: 0.8991 - val_lo
ss: 1.0139 - val_acc: 0.6771
Epoch 42/100
35/35 [==============================] - 124s 4s/step - loss: 0.4639 - acc: 0.8468 - val_lo
ss: 1.0914 - val_acc: 0.6875
Epoch 43/100
35/35 [==============================] - 123s 4s/step - loss: 0.3673 - acc: 0.8753 - val_lo
ss: 1.0167 - val_acc: 0.7049
Epoch 44/100
35/35 [==============================] - 123s 4s/step - loss: 0.3754 - acc: 0.8702 - val_lo
ss: 0.9338 - val_acc: 0.7118
Epoch 45/100
35/35 [==============================] - 121s 3s/step - loss: 0.3175 - acc: 0.9012 - val_lo
ss: 0.9706 - val_acc: 0.7188
Epoch 46/100
35/35 [==============================] - 124s 4s/step - loss: 0.3803 - acc: 0.8601 - val_lo
ss: 1.0782 - val_acc: 0.6944
Epoch 47/100
35/35 [==============================] - 123s 4s/step - loss: 0.4805 - acc: 0.8382 - val_lo
ss: 1.0741 - val_acc: 0.7083
Epoch 48/100
35/35 [==============================] - 124s 4s/step - loss: 0.4301 - acc: 0.8665 - val_lo
ss: 0.9534 - val_acc: 0.7118
Epoch 49/100
35/35 [==============================] - 125s 4s/step - loss: 0.2855 - acc: 0.8972 - val_lo
ss: 0.9103 - val_acc: 0.7500
Epoch 50/100
35/35 [==============================] - 124s 4s/step - loss: 0.2168 - acc: 0.9281 - val_lo
ss: 1.0816 - val_acc: 0.7083
```

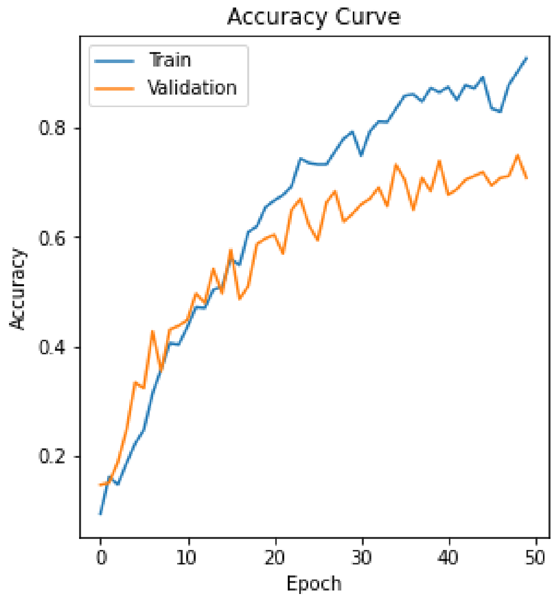
We originally thought that the number of epochs would play a large role in the success of our CNN. However, we found two things. First, that having too few epochs leads to having our model stop learning before reaching its maximum potential. Secondly, we found the EarlyStopping function which determines if the model is reducing its validation loss over epochs and if it is not, then it stops the model. This saves time and resources that would otherwise be wasted on overfitting or maxing out your fitting instead of making progress.

We can see the trends of loss and accuracy better if we graph them. We do this below and also print the final results of the model including the accuracy.

In [ ]:

```python
trainLoss = history.history['loss']
validLoss = history.history['val_loss']

trainAccuracy = history.history['acc']
validAccuracy = history.history['val_acc']

x = np.arange(len(trainAccuracy))

f,ax = plt.subplots(1,2, figsize=(10,5))
ax[0].plot(x, trainLoss)
ax[0].plot(x, validLoss)
ax[0].set_title("Loss Curve")
ax[0].set_xlabel("Epoch")
ax[0].set_ylabel("Loss")
ax[0].legend(['Train', 'Validation'])

ax[1].plot(x, trainAccuracy)
ax[1].plot(x, validAccuracy)
ax[1].set_title("Accuracy Curve")
ax[1].set_xlabel("Epoch")
ax[1].set_ylabel("Accuracy")
ax[1].legend(['Train', 'Validation'])

plt.show()
```





In [ ]:

```
validLoss, validAccuracy = model.evaluate(validDataSet, steps=validationSteps)
print(f"Final accuracy: {validAccuracy*100:.2f}%")
```

```
9/9 [==============================] - 13s 1s/step - loss: 0.8585 - acc: 0.7396
Final accuracy: 73.96%
```

9. Custom Model Observations

Even after many iterations, our model seems to be overfitting. There is a point when the training loss and accuracy continue improving while the validation loss and accuracy do not improve. We have tried common methods for fixing this such as adding dropout layers, and reducing the parameters, however, this at best did nothing and at worst made the CNN worse.

We also tried to change the size of the pooling, which did not seem to affect the accuracy either positively or negatively for strides and dimensions as large as 16 pixels. It did, however, significantly reduce the number of parameters. In this model that reduction of parameters did not significantly change the runtime, so the pooling was reset, but this knowledge could help with other projects.

We predict this overfitting is due to the lack of data that we have to train with, but we want to study the feature maps to see if they offer any hints as to what we can improve.

10. Visualizing Feature Maps

Feature maps are the result of convoluting a kernel with an image. We know our CNN is overtraining, so we decided to look at the first and last feature maps to see what features are being focused on by our model. First, we remind ourselves of the names of the layers.

In [ ]:

```
for i in range(len(model.layers)):
 layer = model.layers[i]
 # check for convolutional layer
 if 'conv' not in layer.name:
  continue
 # summarize output shape
 print(i, layer.name, layer.output.shape)
```

```
0 conv2d (None, 224, 224, 32)
2 conv2d_1 (None, 112, 112, 32)
4 conv2d_2 (None, 56, 56, 64)
6 conv2d_3 (None, 28, 28, 64)
8 conv2d_4 (None, 14, 14, 128)
10 conv2d_5 (None, 7, 7, 128)
```

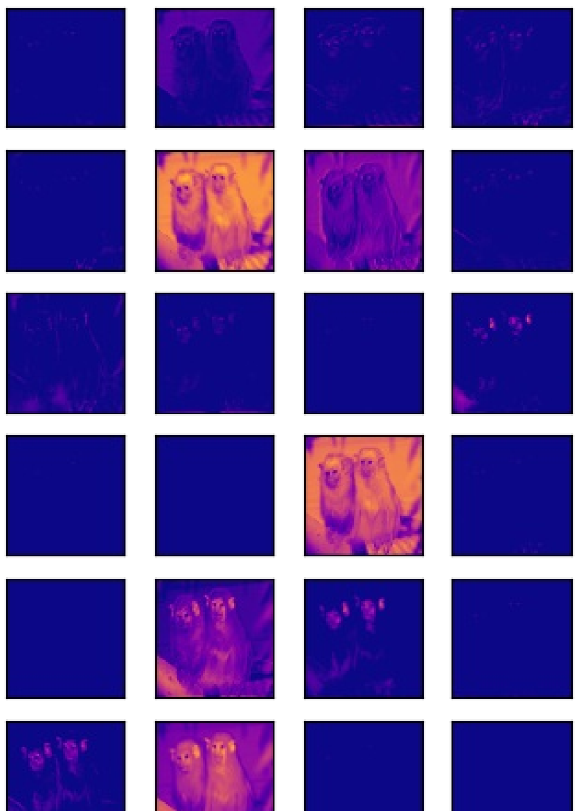
We specifically are interested in the first and last layers, "conv2d" and "conv2d_5," respectively. Below we pick

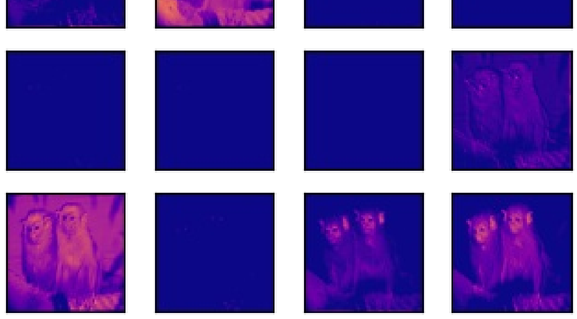
that layer from our list of layers, retrieve the output from it and plot them on a set of subplots for the image we showed at the beginning.

```
In [ ]:
```

```
#for layer in model.layers:
#  if 'conv' not in layer.name:
#    continue
# above code did not graph each graph because of google colab.
#below code will graph one layers feature map.
layer_dict = dict([(layer.name, layer) for layer in model.layers])

layer_name = 'conv2d'

model = Model(inputs=model.inputs, outputs=layer_dict[layer_name].output)
#indent code below if using for layer loop

image = load_img(os.path.join(os.path.join(trainData, randomDir), randomFile), target_size
=(224, 224))
image = img_to_array(image)
image = np.expand_dims(image, axis=0)

print(layer_name)

feature_maps = model.predict(image)
figure(figsize=(4, 8), dpi=100)
#maxInd = layer.output_shape[0][3]
numRows = 8
numCols = 4
index = 1
for _ in range(numRows):
  for _ in range(numCols):
    ax = plt.subplot(numRows, numCols, index)
    ax.set_xticks([])
    ax.set_yticks([])
    plt.imshow(feature_maps[0, :, :, index-1], cmap='plasma')        #'viridis', 'plasma',
'inferno', 'magma', 'cividis'
    index += 1
    #if index > maxInd:
      #break
  #if index > maxInd:
    #break
plt.show()
```

```
conv2d
```



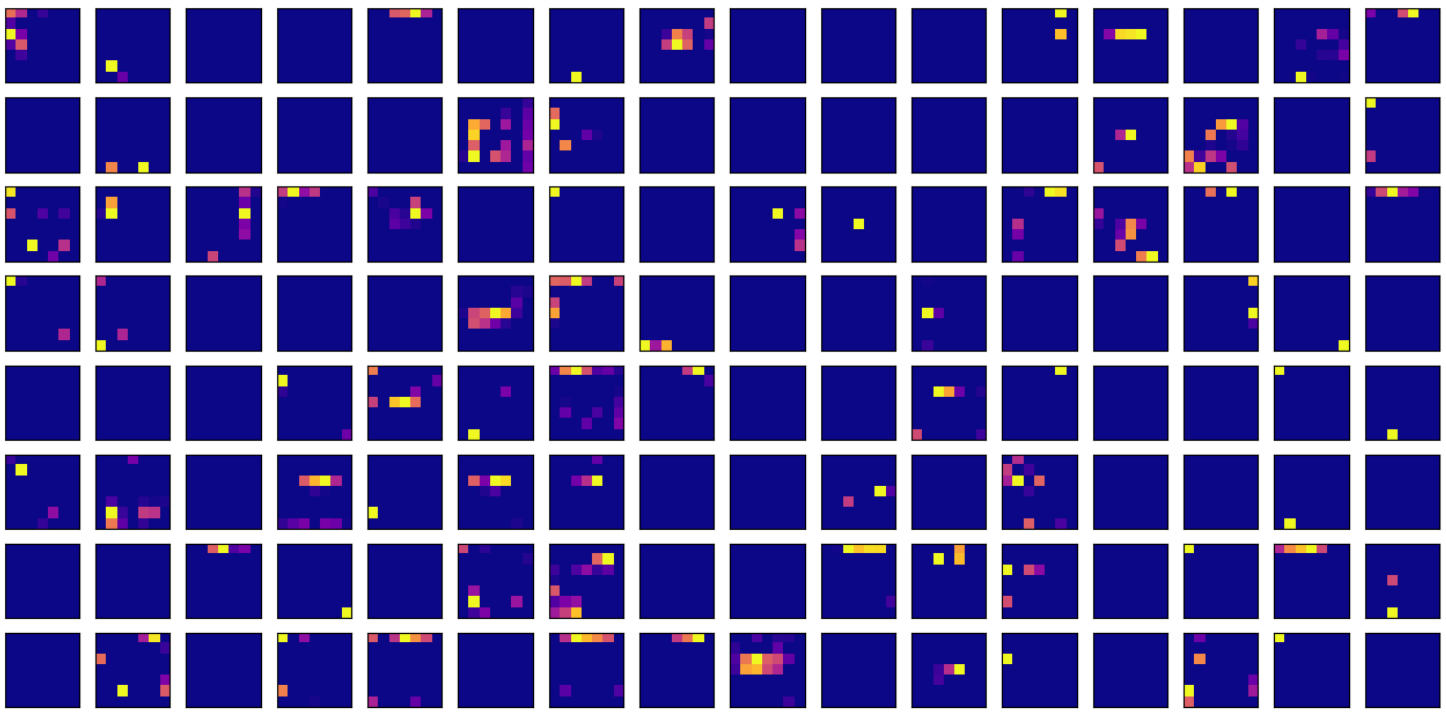


In this image, blue is a low value and represents a feature not considered important on that feature map. Yellow is a high value and represents a feature that is important to that feature map. We can see how fine the details are that are highlighted for this first convolutional layer.

```
In [ ]:
```

```
layer_name2 = 'conv2d_5'

model = Model(inputs=model.inputs, outputs=layer_dict[layer_name2].output)

image = load_img(os.path.join(os.path.join(trainData, randomDir), randomFile), target_size
=(224, 224))
image = img_to_array(image)
image = np.expand_dims(image, axis=0)

print(layer_name2)

feature_maps = model.predict(image)
figure(figsize=(16, 8), dpi=100)
numRows = 8
numCols = 16
index = 1
for _ in range(numRows):
  for _ in range(numCols):
    ax = plt.subplot(numRows, numCols, index)
    ax.set_xticks([])
    ax.set_yticks([])
    plt.imshow(feature_maps[0, :, :, index-1], cmap='plasma')
    index += 1
plt.show()
```

```
conv2d_5
```



These are the feature maps for the last convolutional layer. We can see how low resolution they are. The yellow on these maps represents whole sections of our original image. This is how larger features such as the body of a

monkey are represented. We can also see how we have more feature maps at this layer. This is a byproduct of maintaining the data while manipulating its dimensions.

Overall, this does not offer much insight into why we are overtraining. However, we can see some interesting things. We can see how the CNN focuses on the whole image, not just the monkey. And we can also better understand how the CNN "sees" the world. The one last trick we learned that we can attempt to solve our overtraining issue is transfer learning.

11. Transfer Learning Approach

Transfer Learning is the use of a pre-trained network to help make classifications on new data. This is much like a mentorship. Our CNN does not have enough experience classifying data to learn from such a small pool of images. A CNN that has been trained already, has some knowledge of features that it can use to identify an image. If we modify it slightly we can apply it to our data and it will be able to use its experience and our data to become a better fit for classifying our monkeys. This also comes with some drawbacks. Some data scientists have pointed out the resource cost of transfer learning . While this is somewhat concerning, we have ignored this for now since we are outsourcing our computing power to Google.

Our biggest decision when attempting the transfer learning approach was what pre-trained net to use. We picked VGG16.

VGG16 is a well-established network that also implements CNN. We picked it becuase we believe that we will have the best understanding of it. Our ealier neural net was also a CNN and we believe its important to compare our custom model to another CNN. We picked VGG16 becuase it matched these parameters.

In [ ]:

```
baseModel = VGG16(input_shape=(imgRows, imgCols, imgChannels), include_top=True)

baseModelOutput = baseModel.layers[-2].output

temp = Dropout(0.5)(baseModelOutput)
output = Dense(10, activation='softmax')(temp)

modelTransferLearn = Model(baseModel.input, output)

for layer in baseModel.layers[:-1]:
    layer.trainable=False

modelTransferLearn.compile(loss="categorical_crossentropy",optimizer='adam', metrics="acc"
)
historyTransferLearn = modelTransferLearn.fit(trainDataSet,
                    validation_data = validDataSet,
                    epochs = numOfEpochs,
                    steps_per_epoch = trainingSteps,
                    validation_steps = validationSteps,
                    callbacks = callbackList)
```

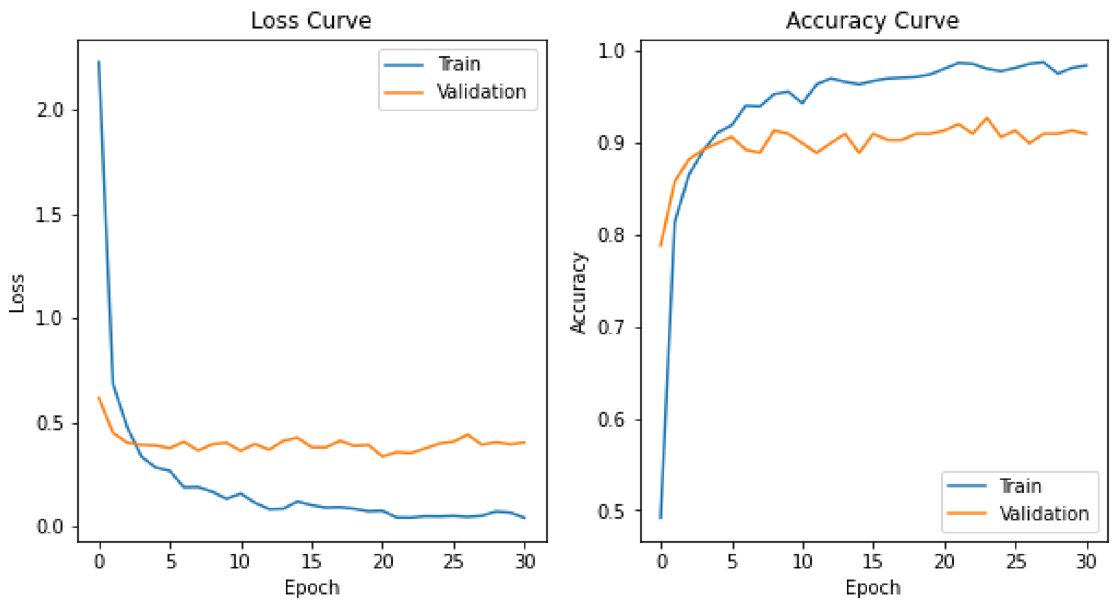
```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vg
g16_weights_tf_dim_ordering_tf_kernels.h5
553467904/553467096 [==============================] - 5s 0us/step
Epoch 1/100
35/35 [==============================] - 760s 22s/step - loss: 3.3133 - acc: 0.3294 - val_l
oss: 0.6164 - val_acc: 0.7882
Epoch 2/100
35/35 [==============================] - 762s 22s/step - loss: 0.6801 - acc: 0.8196 - val_l
oss: 0.4479 - val_acc: 0.8576
Epoch 3/100
35/35 [==============================] - 757s 22s/step - loss: 0.4751 - acc: 0.8643 - val_l
oss: 0.4009 - val_acc: 0.8819
Epoch 4/100
35/35 [==============================] - 742s 21s/step - loss: 0.3417 - acc: 0.8939 - val_l
oss: 0.3904 - val_acc: 0.8924
Epoch 5/100
35/35 [==============================] - 750s 22s/step - loss: 0.3059 - acc: 0.9034 - val_l
oss: 0.3874 - val_acc: 0.8993
Epoch 6/100
35/35 [==============================] - 761s 22s/step - loss: 0.2566 - acc: 0.9134 - val_l
```

```
oss: 0.3746 - val_acc: 0.9062
Epoch 7/100
35/35 [==============================] - 749s 22s/step - loss: 0.1947 - acc: 0.9341 - val_l
oss: 0.4052 - val_acc: 0.8924
Epoch 8/100
35/35 [==============================] - 746s 21s/step - loss: 0.2126 - acc: 0.9398 - val_l
oss: 0.3624 - val_acc: 0.8889
Epoch 9/100
35/35 [==============================] - 758s 22s/step - loss: 0.1871 - acc: 0.9439 - val_l
oss: 0.3934 - val_acc: 0.9132
Epoch 10/100
35/35 [==============================] - 751s 22s/step - loss: 0.1236 - acc: 0.9548 - val_l
oss: 0.4000 - val_acc: 0.9097
Epoch 11/100
35/35 [==============================] - 745s 21s/step - loss: 0.1767 - acc: 0.9446 - val_l
oss: 0.3612 - val_acc: 0.8993
Epoch 12/100
35/35 [==============================] - 752s 22s/step - loss: 0.1103 - acc: 0.9691 - val_l
oss: 0.3951 - val_acc: 0.8889
Epoch 13/100
35/35 [==============================] - 749s 22s/step - loss: 0.0839 - acc: 0.9717 - val_l
oss: 0.3659 - val_acc: 0.8993
Epoch 14/100
35/35 [==============================] - 748s 21s/step - loss: 0.0729 - acc: 0.9689 - val_l
oss: 0.4094 - val_acc: 0.9097
Epoch 15/100
35/35 [==============================] - 747s 21s/step - loss: 0.1367 - acc: 0.9591 - val_l
oss: 0.4249 - val_acc: 0.8889
Epoch 16/100
35/35 [==============================] - 760s 22s/step - loss: 0.1244 - acc: 0.9668 - val_l
oss: 0.3789 - val_acc: 0.9097
Epoch 17/100
35/35 [==============================] - 747s 21s/step - loss: 0.1079 - acc: 0.9667 - val_l
oss: 0.3782 - val_acc: 0.9028
Epoch 18/100
35/35 [==============================] - 747s 21s/step - loss: 0.0873 - acc: 0.9743 - val_l
oss: 0.4097 - val_acc: 0.9028
Epoch 19/100
35/35 [==============================] - 753s 22s/step - loss: 0.0898 - acc: 0.9717 - val_l
oss: 0.3860 - val_acc: 0.9097
Epoch 20/100
35/35 [==============================] - 754s 22s/step - loss: 0.0654 - acc: 0.9717 - val_l
oss: 0.3899 - val_acc: 0.9097
Epoch 21/100
35/35 [==============================] - 754s 22s/step - loss: 0.0731 - acc: 0.9849 - val_l
oss: 0.3340 - val_acc: 0.9132
Epoch 22/100
35/35 [==============================] - 744s 21s/step - loss: 0.0488 - acc: 0.9834 - val_l
oss: 0.3547 - val_acc: 0.9201
Epoch 23/100
35/35 [==============================] - 749s 22s/step - loss: 0.0459 - acc: 0.9835 - val_l
oss: 0.3503 - val_acc: 0.9097
Epoch 24/100
35/35 [==============================] - 747s 21s/step - loss: 0.0520 - acc: 0.9756 - val_l
oss: 0.3725 - val_acc: 0.9271
Epoch 25/100
35/35 [==============================] - 758s 22s/step - loss: 0.0550 - acc: 0.9704 - val_l
oss: 0.3966 - val_acc: 0.9062
Epoch 26/100
35/35 [==============================] - 767s 22s/step - loss: 0.0485 - acc: 0.9818 - val_l
oss: 0.4056 - val_acc: 0.9132
Epoch 27/100
35/35 [==============================] - 747s 21s/step - loss: 0.0547 - acc: 0.9805 - val_l
oss: 0.4395 - val_acc: 0.8993
Epoch 28/100
35/35 [==============================] - 745s 21s/step - loss: 0.0324 - acc: 0.9928 - val_l
oss: 0.3911 - val_acc: 0.9097
Epoch 29/100
35/35 [==============================] - 756s 22s/step - loss: 0.0641 - acc: 0.9782 - val_l
oss: 0.4028 - val_acc: 0.9097
Epoch 30/100
35/35 [==============================] - 755s 22s/step - loss: 0.0662 - acc: 0.9777 - val_l
oss: 0.3930 - val_acc: 0.9132
```

```
Epoch 31/100
35/35 [==============================] - 746s 21s/step - loss: 0.0329 - acc: 0.9856 - val_l
oss: 0.4015 - val_acc: 0.9097
```

In [ ]:

```
trainTLLoss = historyTransferLearn.history['loss']
validTLLoss = historyTransferLearn.history['val_loss']

trainTLAccuracy = historyTransferLearn.history['acc']
validTLAccuracy = historyTransferLearn.history['val_acc']

x = np.arange(len(trainTLAccuracy))

f,ax = plt.subplots(1,2, figsize=(10,5))
ax[0].plot(x, trainTLLoss)
ax[0].plot(x, validTLLoss)
ax[0].set_title("Loss Curve")
ax[0].set_xlabel("Epoch")
ax[0].set_ylabel("Loss")
ax[0].legend(['Train', 'Validation'])

ax[1].plot(x, trainTLAccuracy)
ax[1].plot(x, validTLAccuracy)
ax[1].set_title("Accuracy Curve")
ax[1].set_xlabel("Epoch")
ax[1].set_ylabel("Accuracy")
ax[1].legend(['Train', 'Validation'])

plt.show()
```



In [ ]:

```
validTLLoss, validTLAccuracy = modelTransferLearn.evaluate(validDataSet, steps=validationS
teps)
print(f"Final accuracy: {validTLAccuracy*100:.2f}%")
```

```
9/9 [==============================] - 152s 17s/step - loss: 0.3340 - acc: 0.9132
Final accuracy: 91.32%
```

12. Conclusions

12a. Model Conclusions

From this set of experiments, we can better understand how a Convolutional Neural Net works as well as its strengths and weaknesses. The most obvious weakness is the need for a large dataset. However, we also can attest to the complexity of data handling that may be needed. The strengths come in the capability to generalize from complex data as well as the ability to transfer previously trained nets to new data and have them perform.

These strengths and weaknesses are demonstrated in the final accuracies of the two networks we implemented. With our custom CNN that only had a little more than 1000 data points having around 70%-75% accuracy over many trials, and the transfer learning model achieving ~90%. We believe that while we were not able to successfully create an accurate Custom CNN, the transfer learning model is a successful model. However, we do think that it could be improved with more time.

While these results and conclusions are important, in the next section we will also cover lessons learned which can show some less measurable conclusions drawn.

12b. Lessons Learned

David Conner Borkowski

Starting this project, I had a conceptual understanding of machine learning, but not a mathematical understanding or a technical understanding. This shows in our initial approach. While I think our approach was not bad it did not work as well as we had hoped. We then shared the idea to "see what the CNN sees at each layer." This lead to me first understanding what a feature map is and whats its importance--which also lead me to develop a better mathematical understanding--before learning how to and implementing the code that shows the feature maps. While this did not give me the insight into the CNN I thought it would, it was very important for me in developing an understanding of CNNs. Then I learned about transfer learning and how it can fix the problems we had. Implementing this was easy, but the concepts that it implements such as the benefit of larger datasets further developing my understanding of CNNs.

On top of these lessons, I also learned python, Keras, and the technical side of implementing a CNN. This was the least impactful of the lessons in my opinion but did take the most time.

Todd Bean

When we started working I did not fully understand image convolution and had no experience working with machine learning. I also had not used python or Google Colaboratory. In the course of this project I feel I have gained an understanding of image convolution and I am now comfortable working with Google Colaboratory, though not as comfortable with python yet. I was amazed at how accurate the CNN program was immediately but dissappointed with how we hit the wall on accuracy. Once we reached about seventy percent accuracy, nothing seemed to help, even though we ran many different models as trial and error with different transformations, layers and numbers of layers.

I also feel like I learned a lot about how CNNs intake data. Specifically I think if we had more time we'd need to augment the data in a way that removes most of the background, because I feel that that, along with watermarks and photographer signatures, was hurting our accuracy.

13. Summary Of Contributions

While we both contributed to most sections, we focused on different things and our final result is due to the following contributions.

13a. David Conner Borkowski

I focused mostly on the development of the custom convolutional model, displaying the feature maps, and finalizing the transfer learning model. I also contributed to all of the written sections, and individually wrote the sections related to the mathematical reasoning of a CNN. I think that these sections show how we came in with a naive idea of what to do and learned how to change that into something that was actually achievable.

13b. Todd Bean

I worked mostly on testing out different values for the transforms, testing different types and numbers of layers, and ways to feed in the image data to avoid overfitting. I wrote some descriptions for each section, except the math section, and edited the report including that section to help unify the style. I also started in the transfer learning code but Conner had to finish it because I don't know enough about python.

14. References

All images are from our dataset, generated by us, or generated by us from our dataset.

Gradient descent, how neural networks learn | Deep learning, chapter 2. YouTube, 2017.

Mario, 10 Monkey Species, 28-Jun-2018. [Online]. Available: https://www.kaggle.com/slothkong/10-monkey-species. [Accessed: 30-Apr-2021].

N. Donges, "Gradient Descent: An Introduction to 1 of Machine Learning's Most Popular Algorithms," Built In, 16-Jun-2019. [Online]. Available: https://builtin.com/data-science/gradient-descent. [Accessed: 30-Apr-2021].

P. Skalski, "Gentle Dive into Math Behind Convolutional Neural Networks," Medium, 14-Apr-2019. [Online]. Available: https://towardsdatascience.com/gentle-dive-into-math-behind-convolutional-neural-networks-79a07dd44cf9. [Accessed: 30-Apr-2021].

R. Sevey, "How Much Data is Needed to Train a (Good) Model?," DataRobot, 04-Aug-2017. [Online]. Available: https://www.datarobot.com/blog/how-much-data-is-needed-to-train-a-good-model/. [Accessed: 30-Apr-2021].