Devarsh Patel, Sze Man Tang, Minhhanh Tran, Yazmin Yanez

CECS 456 - Machine Learning
Spring 2022

Animal Classification

Abstract

Convolutional Neural Network is a very effective deep learning model for image classification. In this project, we take a dataset of animal pictures with the goal to correctly classify them. Four algorithms were used to find the one that would produce the greatest accuracy and have the lowest error rate. After training and testing our individual models, we came to the conclusion that the dataset trained using InceptionV3 and Alexnet had the greatest accuracy and the lowest loss: 87.50% and 0.3922 and 86.17% and 0.4386 respectively.

Introduction

In our final project, we are using an animal dataset taken from Kaggle (https://www.kaggle.com/datasets/alessiocorrado99/animals10). The dataset contained a total of 26,179 images with ten separate animal classes. To train our deep convolutional neural network (CNN), we attempted four different methods to produce a model that would give us the greatest accuracy and lowest error rate. We are using convoluted neural networks (CNNs) because the number of parameters grows very quickly when the number of layers increases. CNNs are a very powerful tool to reduce the number of parameters, since each pixel in an image is a feature, without diminishing the quality of the model.
The github repository for our project is:
https://github.com/Devarsh-Patel/CECS-456-Machine-Learning-Project
Contributions of each member:

Minhhanh Tran	-Custom model based on the CNN demo in class - wrote the Abstract & Introduction Section
Devarsh Patel	-InceptionV3
Sze Man Tang	-AlexNet -wrote Conclusion
Yazmin Yanez	-VGG16 -wrote Dataset & Analysis and Comparison sections

Dataset

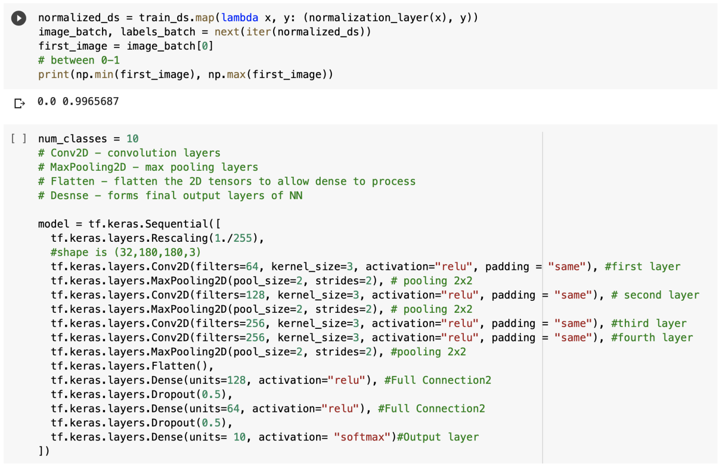
The dataset we worked with was called Animals 10, referring to the ten animal categories provided. This includes 4,862 dog images, 2,623 horse images, 1,447 elephant images, 2,112 butterfly images, 3,098 chicken images, 1,668 cat images, 1,866 cow images, 1,820 sheep images, 4,821 spider images, and 1,862 squirrel images. Not only was there an unbalanced dataset, but the dimensions of each image varied as well. The image sizes ranged from many

sizes including 300 pixels by 300 pixels, 92 pixels by 300 pixels, etc. The images were not preprocessed, so each member of our group decided how to correct the dataset to use in our models. As for the training/validation split, SzeMan Tang, Devarsh Patel, and Minhhanh Tran decided to allocate 80% of the data for training and 20% for validation, and Yazmin Yanez decided to allocate 90% of the data for training and 10% for validation.
https://www.kaggle.com/datasets/alessiocorrado99/animals10

Method

Minhhanh Tran's Model:

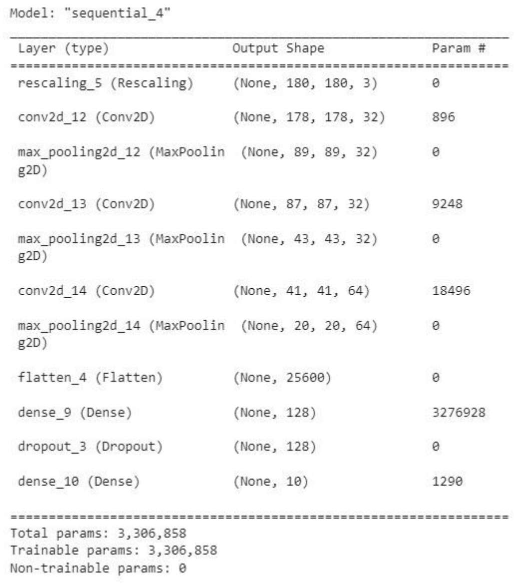
To create my custom model, I used the CNN Demo from our class as a template. I first ran and created the model using the same layers as the one we used in class, but got an accuracy of about 32% and a loss of 1.82. From there, I experimented with different values and was able to get a greater accuracy and a smaller loss. First, I reshaped all my images to be of size 180x180, set my batch size to sixty-four, and normalized my dataset to be between 0-1. In my model, I used four convolution layers, three pooling layers, and two dropout layers. Then I used the rectified linear activation function (ReLU) to speed up training of my model and a small filter size of three to increase the efficiency of my training. A small filter size ensures that my model will process quicker and there is less ambiguity with my parameters.



```
normalized_ds = train_ds.map(lambda x, y: (normalization_layer(x), y))
image_batch, labels_batch = next(iter(normalized_ds))
first_image = image_batch[0]
# between 0-1
print(np.min(first_image), np.max(first_image))
```

```
0.0 0.9965687
```

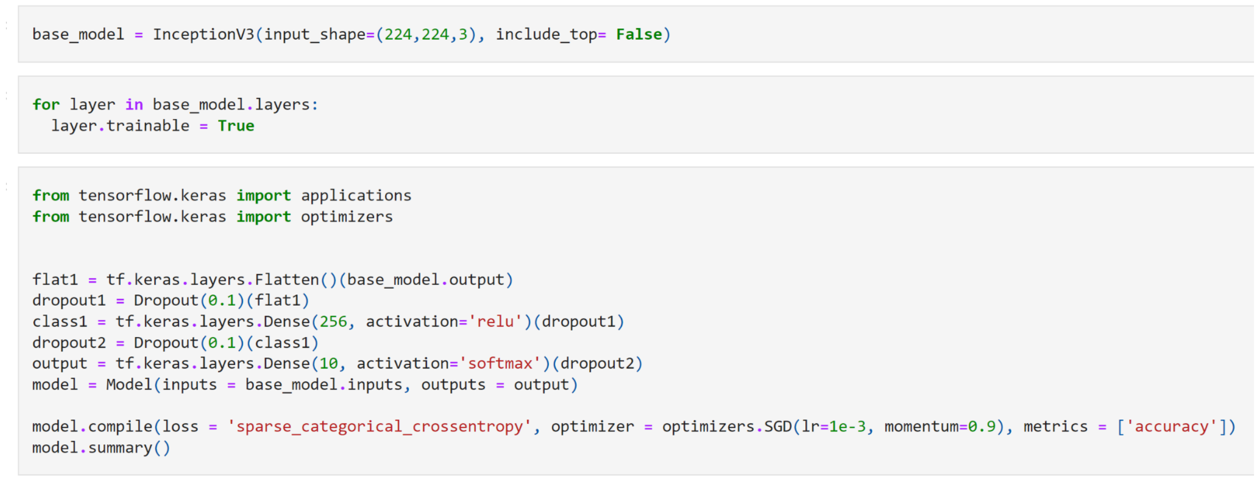
```
num_classes = 10
# Conv2D - convolution layers
# MaxPooling2D - max pooling layers
# Flatten - flatten the 2D tensors to allow dense to process
# Desnse - forms final output layers of NN

model = tf.keras.Sequential([
  tf.keras.layers.Rescaling(1./255),
  #shape is (32,180,180,3)
  tf.keras.layers.Conv2D(filters=64, kernel_size=3, activation="relu", padding = "same"), #first layer
  tf.keras.layers.MaxPooling2D(pool_size=2, strides=2), # pooling 2x2
  tf.keras.layers.Conv2D(filters=128, kernel_size=3, activation="relu", padding = "same"), # second layer
  tf.keras.layers.MaxPooling2D(pool_size=2, strides=2), # pooling 2x2
  tf.keras.layers.Conv2D(filters=256, kernel_size=3, activation="relu", padding = "same"), #third layer
  tf.keras.layers.Conv2D(filters=256, kernel_size=3, activation="relu", padding = "same"), #fourth layer
  tf.keras.layers.MaxPooling2D(pool_size=2, strides=2), #pooling 2x2
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(units=128, activation="relu"), #Full Connection2
  tf.keras.layers.Dropout(0.5),
  tf.keras.layers.Dense(units=64, activation="relu"), #Full Connection2
  tf.keras.layers.Dropout(0.5),
  tf.keras.layers.Dense(units= 10, activation= "softmax")#Output layer
])
```



```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 rescaling_5 (Rescaling)     (None, 180, 180, 3)       0

 conv2d_12 (Conv2D)          (None, 178, 178, 32)      896

 max_pooling2d_12 (MaxPoolin  (None, 89, 89, 32)       0
 g2D)

 conv2d_13 (Conv2D)          (None, 87, 87, 32)        9248

 max_pooling2d_13 (MaxPoolin  (None, 43, 43, 32)       0
 g2D)

 conv2d_14 (Conv2D)          (None, 41, 41, 64)        18496

 max_pooling2d_14 (MaxPoolin  (None, 20, 20, 64)       0
 g2D)

 flatten_4 (Flatten)         (None, 25600)             0

 dense_9 (Dense)             (None, 128)               3276928

 dropout_3 (Dropout)         (None, 128)               0

 dense_10 (Dense)            (None, 10)                1290

=================================================================
Total params: 3,306,858
Trainable params: 3,306,858
Non-trainable params: 0
```

Devarsh Patel's Model:

To create the CNN architecture model, I have implemented the InceptionV3 where I am setting up the input_shapr and include_top = false. The InceptionV3 model has a total of 48 layers. To get accurate results, I have implemented dropout between the last two layers to see if my accuracy will change. It did change, but it did not change at the level that I was expecting it to. I decided to keep it because the accuracy kind of looks more accurate using dropouts. I am only dropping 10%.



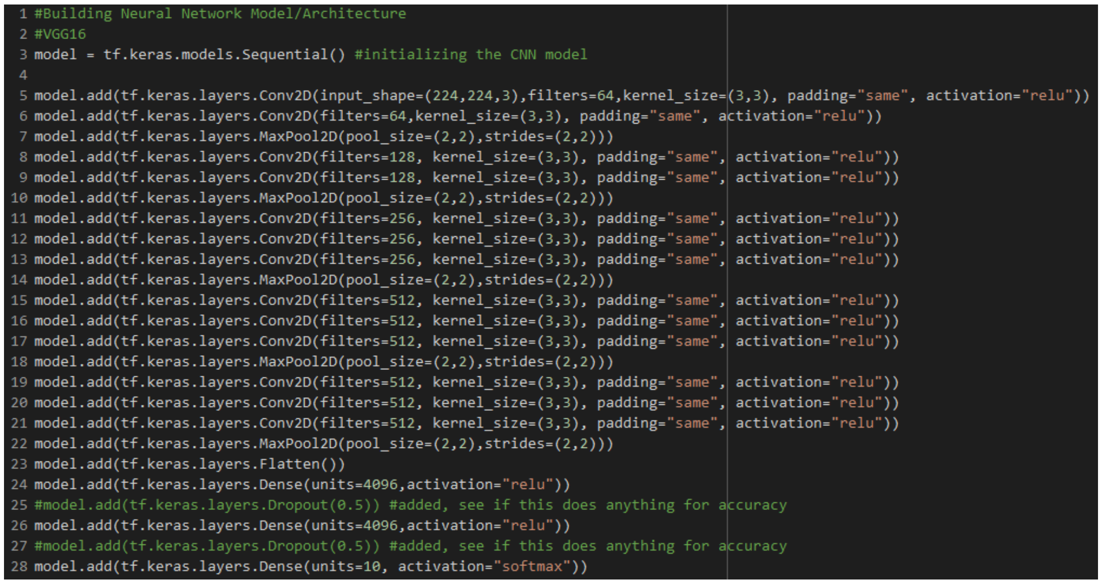
```
base_model = InceptionV3(input_shape=(224,224,3), include_top= False)
```

```
for layer in base_model.layers:
  layer.trainable = True
```

```
from tensorflow.keras import applications
from tensorflow.keras import optimizers


flat1 = tf.keras.layers.Flatten()(base_model.output)
dropout1 = Dropout(0.1)(flat1)
class1 = tf.keras.layers.Dense(256, activation='relu')(dropout1)
dropout2 = Dropout(0.1)(class1)
output = tf.keras.layers.Dense(10, activation='softmax')(dropout2)
model = Model(inputs = base_model.inputs, outputs = output)

model.compile(loss = 'sparse_categorical_crossentropy', optimizer = optimizers.SGD(lr=1e-3, momentum=0.9), metrics = ['accuracy'])
model.summary()
```

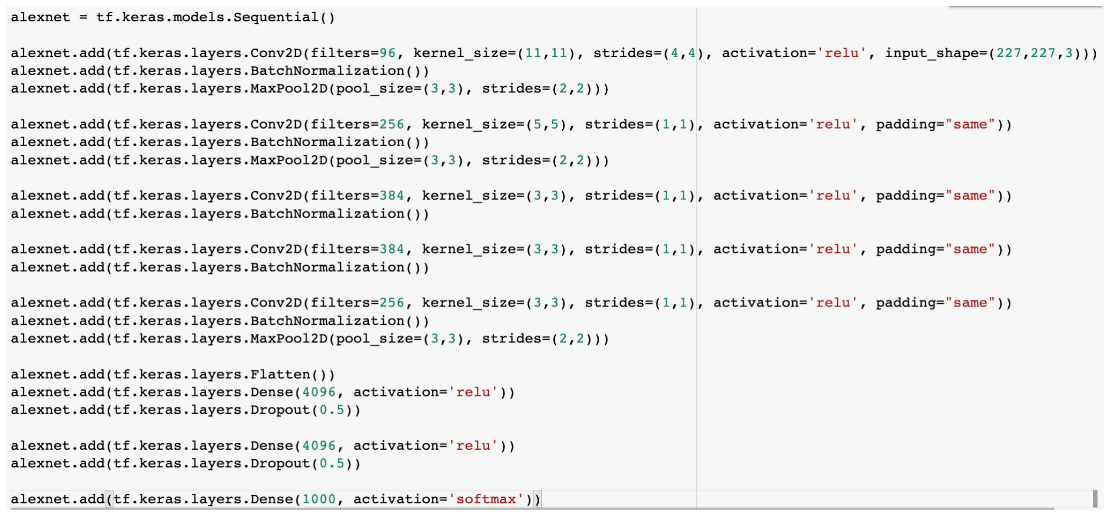
Yazmin Yanez's Model:

To create my architecture, I followed the template of a VGG16 model. The most common input size I found for the model was 224 by 224 by 3, so I altered my data accordingly. The architecture consisted of the layers in this order: two convolution layers, maxpool, two convolution layers, maxpool, three convolution layers, maxpool, three convolution layers, maxpool, three convolution layers, maxpool, a flattening layers, two dense layers with the activation as ReLU, and one final dense layer with the activation softmax and 10 units for the output. Through my research of the VGG architecture, I had seen that some models used two dropout layers sandwiched between the final three dense layers. I tested my model with the dropout layers and did not see an immediate significant change in my accuracy, so I decided to leave them out. I was also limited by my resources. I used Google Colab to run my software, but because I spent so much time trying to fix and test my model, a limit was placed on my usage of my GPU for testing, so I used Google's provided runtime, which was much slower than my GPU runtime.



```
#Building Neural Network Model/Architecture
#VGG16
model = tf.keras.models.Sequential() #initializing the CNN model

model.add(tf.keras.layers.Conv2D(input_shape=(224,224,3),filters=64,kernel_size=(3,3), padding="same", activation="relu"))
model.add(tf.keras.layers.Conv2D(filters=64,kernel_size=(3,3), padding="same", activation="relu"))
model.add(tf.keras.layers.MaxPool2D(pool_size=(2,2),strides=(2,2)))
model.add(tf.keras.layers.Conv2D(filters=128, kernel_size=(3,3), padding="same", activation="relu"))
model.add(tf.keras.layers.Conv2D(filters=128, kernel_size=(3,3), padding="same", activation="relu"))
model.add(tf.keras.layers.MaxPool2D(pool_size=(2,2),strides=(2,2)))
model.add(tf.keras.layers.Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
model.add(tf.keras.layers.Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
model.add(tf.keras.layers.Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
model.add(tf.keras.layers.MaxPool2D(pool_size=(2,2),strides=(2,2)))
model.add(tf.keras.layers.Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(tf.keras.layers.Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(tf.keras.layers.Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(tf.keras.layers.MaxPool2D(pool_size=(2,2),strides=(2,2)))
model.add(tf.keras.layers.Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(tf.keras.layers.Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(tf.keras.layers.Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(tf.keras.layers.MaxPool2D(pool_size=(2,2),strides=(2,2)))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(units=4096,activation="relu"))
#model.add(tf.keras.layers.Dropout(0.5)) #added, see if this does anything for accuracy
model.add(tf.keras.layers.Dense(units=4096,activation="relu"))
#model.add(tf.keras.layers.Dropout(0.5)) #added, see if this does anything for accuracy
model.add(tf.keras.layers.Dense(units=10, activation="softmax"))
```

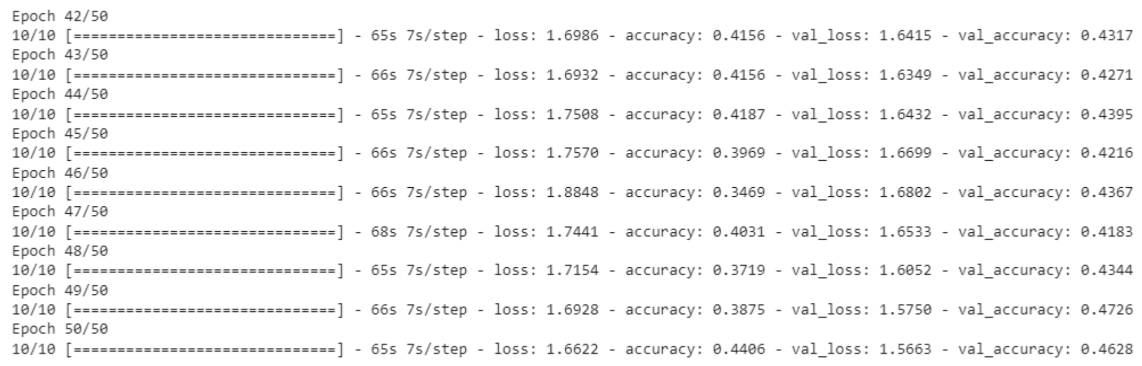
SzeMan Tang's Model:
To create the AlexNet model, I use 5 convolution layers and 3 fully connected layers with 227x227x3 input size. The batch size is 64 instead of the default 32 to provide more data points per iteration to help with loss rate. The model consisted of convolutions, max pooling, dropout, batch normalization and ReLU activations. The total parameters is about 58 millions, which is very high compared with other models. There are two dropouts 0.5, which means 50% are set to zero. Dropout and Batch normalization methods are being used to reduce overfitting and make the network more stable. ReLU activation function is used to add non-linearity, which can help to reduce training time. 1st, 2nd and 5th layers have Max-Pooling for feature extraction and reduce the size of the network.



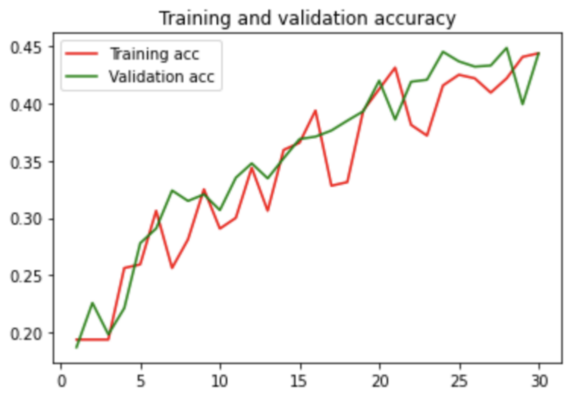
```
alexnet = tf.keras.models.Sequential()

alexnet.add(tf.keras.layers.Conv2D(filters=96, kernel_size=(11,11), strides=(4,4), activation='relu', input_shape=(227,227,3)))
alexnet.add(tf.keras.layers.BatchNormalization())
alexnet.add(tf.keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)))

alexnet.add(tf.keras.layers.Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), activation='relu', padding="same"))
alexnet.add(tf.keras.layers.BatchNormalization())
alexnet.add(tf.keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)))

alexnet.add(tf.keras.layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu', padding="same"))
alexnet.add(tf.keras.layers.BatchNormalization())

alexnet.add(tf.keras.layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu', padding="same"))
alexnet.add(tf.keras.layers.BatchNormalization())

alexnet.add(tf.keras.layers.Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu', padding="same"))
alexnet.add(tf.keras.layers.BatchNormalization())
alexnet.add(tf.keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)))

alexnet.add(tf.keras.layers.Flatten())
alexnet.add(tf.keras.layers.Dense(4096, activation='relu'))
alexnet.add(tf.keras.layers.Dropout(0.5))

alexnet.add(tf.keras.layers.Dense(4096, activation='relu'))
alexnet.add(tf.keras.layers.Dropout(0.5))

alexnet.add(tf.keras.layers.Dense(1000, activation='softmax'))
```

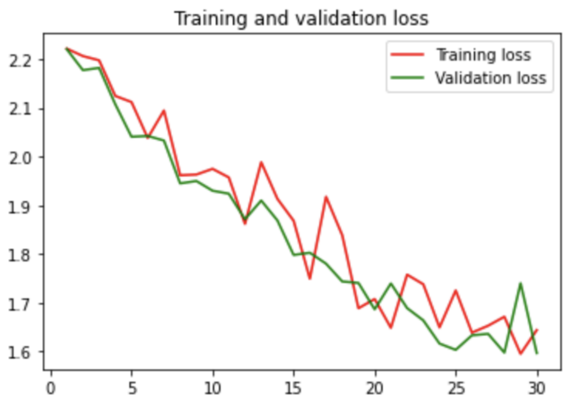
Experiment and Results

Minhhanh Tran's Model:
My custom model ran fifty epochs with ten steps per epoch. As you can see from the figure below, during the last few epochs, my training loss varied a little between epochs but stayed in a range from 1.6622 to 1.8848 while my validation loss ranged from 1.5663 to 1.6802. My training accuracy at its lowest was around 34.69% and at its highest was 44.06%. My validation accuracy ranged from 41.83% to 47.26% at its highest. Although the results I obtained were not the best, we can still see a downward trend in both training and validation loss and an upward trend in both training and validation accuracy.



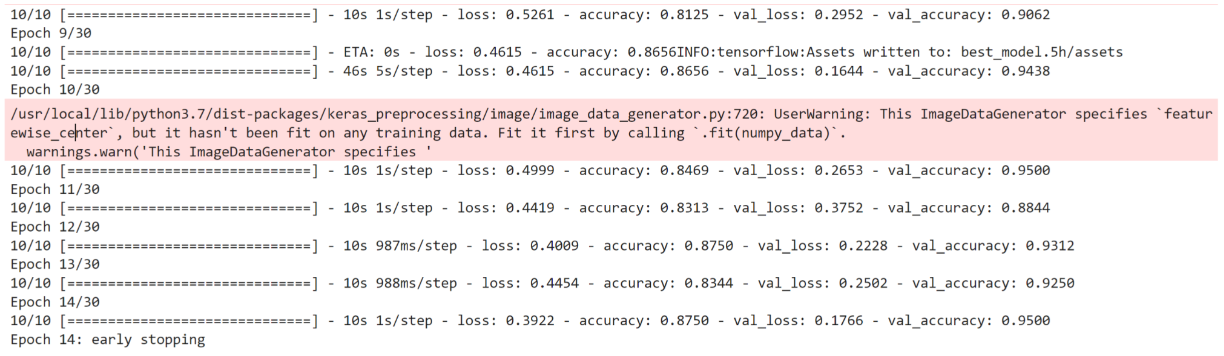
```
Epoch 42/50
10/10 [==============================] - 65s 7s/step - loss: 1.6986 - accuracy: 0.4156 - val_loss: 1.6415 - val_accuracy: 0.4317
Epoch 43/50
10/10 [==============================] - 66s 7s/step - loss: 1.6932 - accuracy: 0.4156 - val_loss: 1.6349 - val_accuracy: 0.4271
Epoch 44/50
10/10 [==============================] - 65s 7s/step - loss: 1.7508 - accuracy: 0.4187 - val_loss: 1.6432 - val_accuracy: 0.4395
Epoch 45/50
10/10 [==============================] - 66s 7s/step - loss: 1.7570 - accuracy: 0.3969 - val_loss: 1.6699 - val_accuracy: 0.4216
Epoch 46/50
10/10 [==============================] - 66s 7s/step - loss: 1.8848 - accuracy: 0.3469 - val_loss: 1.6802 - val_accuracy: 0.4367
Epoch 47/50
10/10 [==============================] - 68s 7s/step - loss: 1.7441 - accuracy: 0.4031 - val_loss: 1.6533 - val_accuracy: 0.4183
Epoch 48/50
10/10 [==============================] - 65s 7s/step - loss: 1.7154 - accuracy: 0.3719 - val_loss: 1.6052 - val_accuracy: 0.4344
Epoch 49/50
10/10 [==============================] - 66s 7s/step - loss: 1.6928 - accuracy: 0.3875 - val_loss: 1.5750 - val_accuracy: 0.4726
Epoch 50/50
10/10 [==============================] - 65s 7s/step - loss: 1.6622 - accuracy: 0.4406 - val_loss: 1.5663 - val_accuracy: 0.4628
```





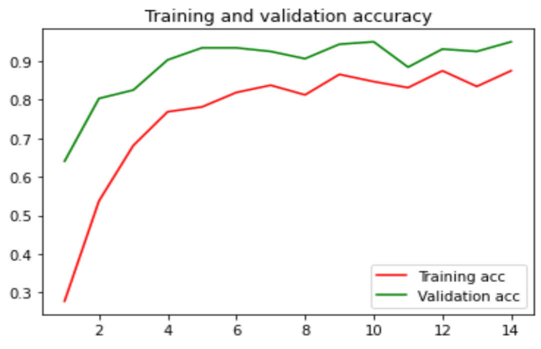
Devarsh Patel's Model:

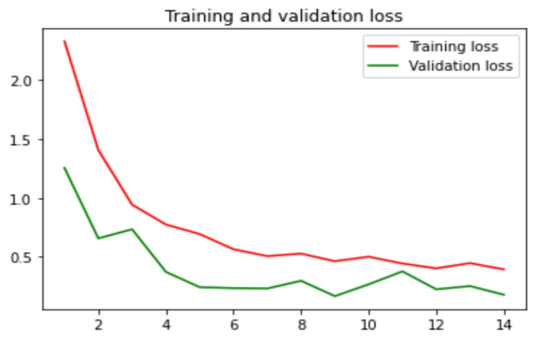
I have used early stopping to stop overfitting when the data is training. The training test has done a few auto augmentations, which is the reason I believe that my test train results have less accuracy compared to validation, since validations have chosen to regulate pictures. The steps-per-epochs 10 and epochs 30. If you see the result provided below, it only has 14 epochs run after which it stops to prevent overfitting.



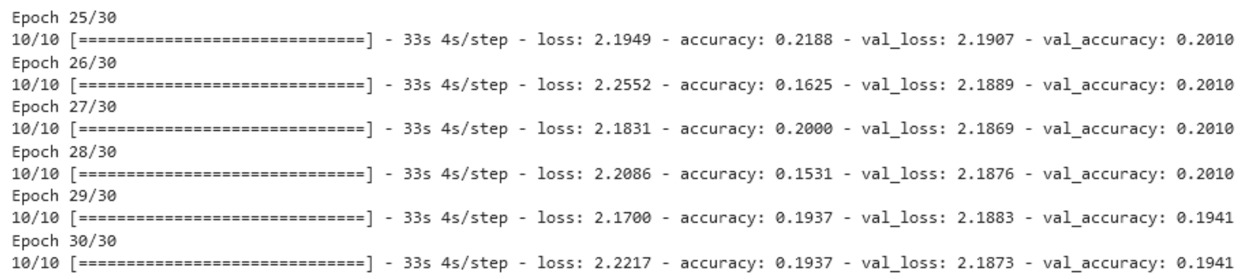
```
10/10 [==============================] - 10s 1s/step - loss: 0.5261 - accuracy: 0.8125 - val_loss: 0.2952 - val_accuracy: 0.9062
Epoch 9/30
10/10 [==============================] - ETA: 0s - loss: 0.4615 - accuracy: 0.8656INFO:tensorflow:Assets written to: best_model.5h/assets
10/10 [==============================] - 46s 5s/step - loss: 0.4615 - accuracy: 0.8656 - val_loss: 0.1644 - val_accuracy: 0.9438
Epoch 10/30
/usr/local/lib/python3.7/dist-packages/keras_preprocessing/image/image_data_generator.py:720: UserWarning: This ImageDataGenerator specifies `featur
ewise_center`, but it hasn't been fit on any training data. Fit it first by calling `.fit(numpy_data)`.
  warnings.warn('This ImageDataGenerator specifies '
10/10 [==============================] - 10s 1s/step - loss: 0.4999 - accuracy: 0.8469 - val_loss: 0.2653 - val_accuracy: 0.9500
Epoch 11/30
10/10 [==============================] - 10s 1s/step - loss: 0.4419 - accuracy: 0.8313 - val_loss: 0.3752 - val_accuracy: 0.8844
Epoch 12/30
10/10 [==============================] - 10s 987ms/step - loss: 0.4009 - accuracy: 0.8750 - val_loss: 0.2228 - val_accuracy: 0.9312
Epoch 13/30
10/10 [==============================] - 10s 988ms/step - loss: 0.4454 - accuracy: 0.8344 - val_loss: 0.2502 - val_accuracy: 0.9250
Epoch 14/30
10/10 [==============================] - 10s 1s/step - loss: 0.3922 - accuracy: 0.8750 - val_loss: 0.1766 - val_accuracy: 0.9500
Epoch 14: early stopping
```

The graphical results for the training and validation accuracy and loss are shown below. The result does not contain any overfitting or underfitting. The highest accuracy of these results is 87.50% and loss came out at 0.3922.

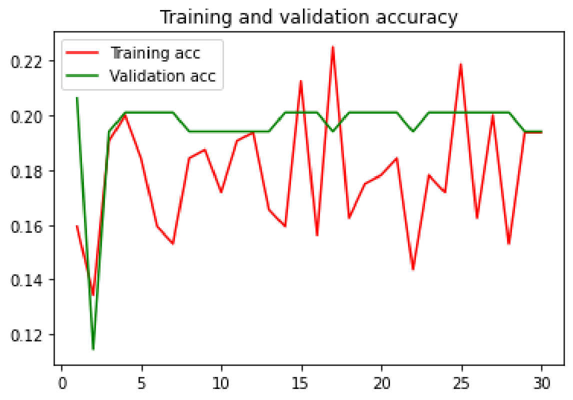


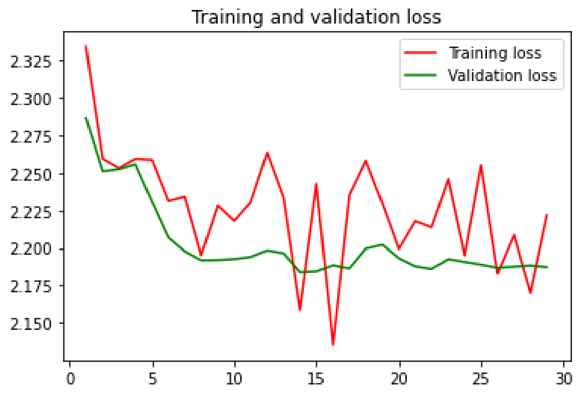


Yazmin Yanez's Model:
My VGG16 model ran through 30 epochs with 10 steps-per-epoch. Because I did not import a pre-trained VGG model, my data varied greatly from epoch to epoch. The highest training accuracy the model achieved was 22.5% and the lowest training loss achieved was 2.1353. However, the final training accuracy and loss were 19.37% and 2.2217, respectively. The highest validation accuracy and lowest loss were 20.63% and 2.1839. However, the final validation accuracy and loss were 19.41% and 2.1873.



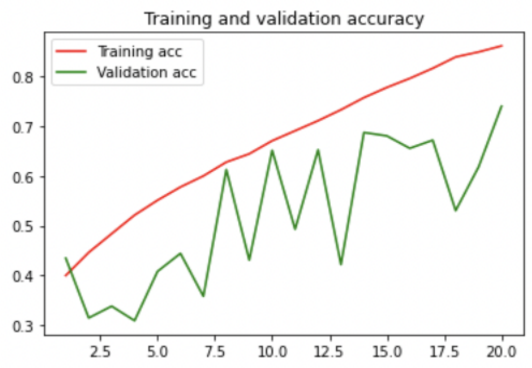
```
Epoch 25/30
10/10 [==============================] - 33s 4s/step - loss: 2.1949 - accuracy: 0.2188 - val_loss: 2.1907 - val_accuracy: 0.2010
Epoch 26/30
10/10 [==============================] - 33s 4s/step - loss: 2.2552 - accuracy: 0.1625 - val_loss: 2.1889 - val_accuracy: 0.2010
Epoch 27/30
10/10 [==============================] - 33s 4s/step - loss: 2.1831 - accuracy: 0.2000 - val_loss: 2.1869 - val_accuracy: 0.2010
Epoch 28/30
10/10 [==============================] - 33s 4s/step - loss: 2.2086 - accuracy: 0.1531 - val_loss: 2.1876 - val_accuracy: 0.2010
Epoch 29/30
10/10 [==============================] - 33s 4s/step - loss: 2.1700 - accuracy: 0.1937 - val_loss: 2.1883 - val_accuracy: 0.1941
Epoch 30/30
10/10 [==============================] - 33s 4s/step - loss: 2.2217 - accuracy: 0.1937 - val_loss: 2.1873 - val_accuracy: 0.1941
```

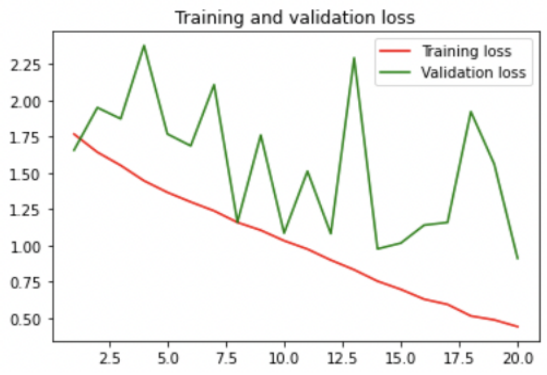




SzeMan Tang's Model:
There are 20 epochs with steps-per-epochs 328 for the AlexNet model. The best result among 20 epochs is 86.17% for the training accuracy and 0.4386 for training loss. The validation accuracy is 74.02% and validation loss is 0.9103. There's a downward trend for accuracy and upward trend for loss graph. From the graphical result, it shows that there's an overfitting issue. From the loss graph, validation error is higher than training error, which means the model is not generalized. Dropout is added to help with generalizing the model. With dropout, a percentage of the features are set to zero.

```
Epoch 10/20
328/328 [==============================] - 29s 89ms/step - loss: 1.0318 - accuracy: 0.6708 - val_loss: 1.0841 - val_accuracy: 0.6514
Epoch 11/20
328/328 [==============================] - 29s 89ms/step - loss: 0.9738 - accuracy: 0.6909 - val_loss: 1.5117 - val_accuracy: 0.4928
Epoch 12/20
328/328 [==============================] - 29s 89ms/step - loss: 0.8980 - accuracy: 0.7111 - val_loss: 1.0804 - val_accuracy: 0.6525
Epoch 13/20
328/328 [==============================] - 29s 89ms/step - loss: 0.8313 - accuracy: 0.7332 - val_loss: 2.2940 - val_accuracy: 0.4218
Epoch 14/20
328/328 [==============================] - 29s 89ms/step - loss: 0.7525 - accuracy: 0.7575 - val_loss: 0.9750 - val_accuracy: 0.6875
Epoch 15/20
328/328 [==============================] - 29s 89ms/step - loss: 0.6959 - accuracy: 0.7781 - val_loss: 1.0153 - val_accuracy: 0.6806
Epoch 16/20
328/328 [==============================] - 29s 89ms/step - loss: 0.6278 - accuracy: 0.7963 - val_loss: 1.1397 - val_accuracy: 0.6556
Epoch 17/20
328/328 [==============================] - 29s 89ms/step - loss: 0.5921 - accuracy: 0.8166 - val_loss: 1.1578 - val_accuracy: 0.6720
Epoch 18/20
328/328 [==============================] - 29s 89ms/step - loss: 0.5123 - accuracy: 0.8394 - val_loss: 1.9222 - val_accuracy: 0.5303
Epoch 19/20
328/328 [==============================] - 29s 89ms/step - loss: 0.4854 - accuracy: 0.8495 - val_loss: 1.5609 - val_accuracy: 0.6181
Epoch 20/20
328/328 [==============================] - 29s 89ms/step - loss: 0.4386 - accuracy: 0.8617 - val_loss: 0.9103 - val_accuracy: 0.7402
```





Analysis and Comparison

Of the models created in this team, the one with the highest accuracy was Devarsh Patel's model using Inception V3 with 87.50% and a loss of 0.3922. Compared to the rest of the group, the Inception V3 model was the only model that was imported with transfer learning from a pre-trained model. Because of this, Devarsh was able to obtain a higher accuracy without extensive training. The other three models were not transferred from pre-trained models and only trained on the Animals 10 dataset.

All of the group's models used or attempted dropout. For Minhhanh, SzeMan, and Devarsh, using dropout helped increase their accuracy, but for Yazmin dropout did not help increase the accuracy. Parameter-wise, the model with the most parameters was VGG16 with 134,301,514 parameters. The AlexNet model had 58,327,818 parameters, the Inception V3 model had 34,912,810 parameters, and the lowest number of parameters was 7,320,010 from the baseline CNN model. Also, all the models ended with a few dense layers, meaning that the last few layers were fully connected layers. However, depending on whether each model used dropout between those layers, some of the connections would have been dropped making them not fully connected but still connected. This would affect each model's efficiency and accuracy.

Conclusion

Overall, InceptionV3 has the best performance for the animals dataset. AlexNet's performance is second based on the number, but it has overfitting and underfitting issues. Compared to other two models, VGG 16 and Custom model's performance were not the most ideal based on the lower accuracy scores, and they still have room to improve. One example would be that VGG16 could be improved with importing a pre-trained model since Inception V3 was very successful as an imported pre-trained model. Overall, more testing and tweaking of the architectures could lead to a potential higher accuracy for all models.

References

https://towardsdatascience.com/step-by-step-vgg16-implementation-in-keras-for-beginners-a833c686ae6c
https://neurohive.io/en/popular-networks/vgg16/
https://neurohive.io/en/popular-networks/alexnet-imagenet-classification-with-deep-convolutional-neural-networks/