Assignment-2: Convolution

Bharath Bhimreddy

Introduction:

The objective is the evaluation of the efficiency of the convnets designed for classifying pictures; the dogs vs cats' dataset has been used as a context. There were two main strategies that were taken into consideration: introducing pretrained convnet in case of training formerly and a network from scratch. With the focus on determining the influence of sample size and training strategy optimal choice on the model's performance, I had set out to do the analysis.

Methodology:

Data preprocessing:

I have taken the dataset of cats and dogs of 25000 images as my dataset here. I have divided them up into training and test beforehand and imported them into google colab.

```
[4] !unzip /content/drive/MyDrive/dogs-vs-cats.zip

Archive:  /content/drive/MyDrive/dogs-vs-cats.zip
  inflating: sampleSubmission.csv
  inflating: test1.zip
  inflating: train.zip

[5] !unzip /content/train.zip
    !unzip /content/test1.zip
```

```
[6] import glob
    import pandas as pd
    from tensorflow.keras.preprocessing.image import ImageDataGenerator
    image = []
    label = []
    base_dir = '/content/'
    train_path_content = os.listdir(f'{base_dir}/train')
    print(f'train_size: {len(train_path_content)}')
    train_dir = os.path.join(base_dir, 'train')

    train_imgs = glob.glob(os.path.join(train_dir, "*"))
    for img in train_imgs:
        image_name = os.path.splitext(os.path.basename(img))[0]
        if 'cat' in image_name:
        # move it to directory /content/train/cats
            image.append(img)
            label.append('cat')
        elif 'dog' in image_name:
        # move it to directory /content/train/dogs
            image.append(img)
            label.append('dog')
    df = pd.DataFrame({'path':image,'label':label})

    train_size: 25000
```

```
# The path to the directory where the original
# dataset was uncompressed
original_dataset_dir = '/content/train'

# The directory where we will
# store our smaller dataset
base_dir = '/content/dog_and_cat_small/'

os.mkdir(base_dir)

# Directories for our training,
# validation and test splits
train_dir = os.path.join(base_dir, 'train')
os.mkdir(train_dir)
validation_dir = os.path.join(base_dir, 'validation')
os.mkdir(validation_dir)
test_dir = os.path.join(base_dir, 'test')
os.mkdir(test_dir)

# Directory with our training cat pictures
train_cats_dir = os.path.join(train_dir, 'cats')
os.mkdir(train_cats_dir)

# Directory with our training dog pictures
train_dogs_dir = os.path.join(train_dir, 'dogs')
os.mkdir(train_dogs_dir)

# Directory with our validation cat pictures
validation_cats_dir = os.path.join(validation_dir, 'cats')
os.mkdir(validation_cats_dir)

# Directory with our validation dog pictures
validation_dogs_dir = os.path.join(validation_dir, 'dogs')
os.mkdir(validation_dogs_dir)

# Directory with our validation cat pictures
test_cats_dir = os.path.join(test_dir, 'cats')
os.mkdir(test_cats_dir)

# Directory with our validation dog pictures
test_dogs_dir = os.path.join(test_dir, 'dogs')
os.mkdir(test_dogs_dir)

# Copy first 1000 cat images to train_cats_dir
fnames = ['cat.{}.jpg'.format(i) for i in range(1000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(train_cats_dir, fname)
    shutil.copyfile(src, dst)

# Copy next 500 cat images to validation_cats_dir
fnames = ['cat.{}.jpg'.format(i) for i in range(1000, 1500)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(validation_cats_dir, fname)
    shutil.copyfile(src, dst)

# Copy next 500 cat images to test_cats_dir
fnames = ['cat.{}.jpg'.format(i) for i in range(1500, 2000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(test_cats_dir, fname)
    shutil.copyfile(src, dst)

# Copy first 1000 dog images to train_dogs_dir
fnames = ['dog.{}.jpg'.format(i) for i in range(1000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(train_dogs_dir, fname)
    shutil.copyfile(src, dst)

# Copy next 500 dog images to validation_dogs_dir
fnames = ['dog.{}.jpg'.format(i) for i in range(1000, 1500)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(validation_dogs_dir, fname)
    shutil.copyfile(src, dst)

# Copy next 500 dog images to test_dogs_dir
fnames = ['dog.{}.jpg'.format(i) for i in range(1500, 2000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(test_dogs_dir, fname)
    shutil.copyfile(src, dst)
```

I divided them into training and validation and test datasets.

Training from Scratch:

- We will begin with 1000 samples for training, 500 samples for validation, and 500 samples for testing.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 148, 148, 32)      896

 max_pooling2d (MaxPooling2  (None, 74, 74, 32)        0
 D)

 conv2d_1 (Conv2D)           (None, 72, 72, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 36, 36, 64)        0
 g2D)

 conv2d_2 (Conv2D)           (None, 34, 34, 128)       73856

 max_pooling2d_2 (MaxPoolin  (None, 17, 17, 128)       0
 g2D)

 conv2d_3 (Conv2D)           (None, 15, 15, 128)       147584

 max_pooling2d_3 (MaxPoolin  (None, 7, 7, 128)         0
 g2D)

 flatten (Flatten)           (None, 6272)              0

 dense (Dense)               (None, 512)               3211776

 dense_1 (Dense)             (None, 1)                 513

=================================================================
Total params: 3453121 (13.17 MB)
Trainable params: 3453121 (13.17 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

- This is the model summary for the convolution model.
- Using Image generators I have rescaled all the images by 1./255 and the target size is (150,150). We will use binary_crossentropy as the loss function and RMSprop as optimizer.

```
    model.compile(loss='binary_crossentropy',
                  optimizer=optimizers.RMSprop(lr=1e-4),
                  metrics=['acc'])
```

```
    WARNING:absl:`lr` is deprecated in Keras optimizer, please use `learning_rate` or use the legacy optimizer, e.g.
```

```
[13] from keras.preprocessing.image import ImageDataGenerator

    # All images will be rescaled by 1./255
    train_datagen = ImageDataGenerator(rescale=1./255)
    test_datagen = ImageDataGenerator(rescale=1./255)

    train_generator = train_datagen.flow_from_directory(
            # This is the target directory
            train_dir,
            # All images will be resized to 150x150
            target_size=(150, 150),
            batch_size=20,
            # Since we use binary_crossentropy loss, we need binary labels
            class_mode='binary')

    validation_generator = test_datagen.flow_from_directory(
            validation_dir,
            target_size=(150, 150),
            batch_size=20,
            class_mode='binary')
```

```
    Found 2000 images belonging to 2 classes.
    Found 1000 images belonging to 2 classes.
```

```
[14] for data_batch, labels_batch in train_generator:
        print('data batch shape:', data_batch.shape)
        print('labels batch shape:', labels_batch.shape)
        break
```

```
    data batch shape: (20, 150, 150, 3)
    labels batch shape: (20,)
```

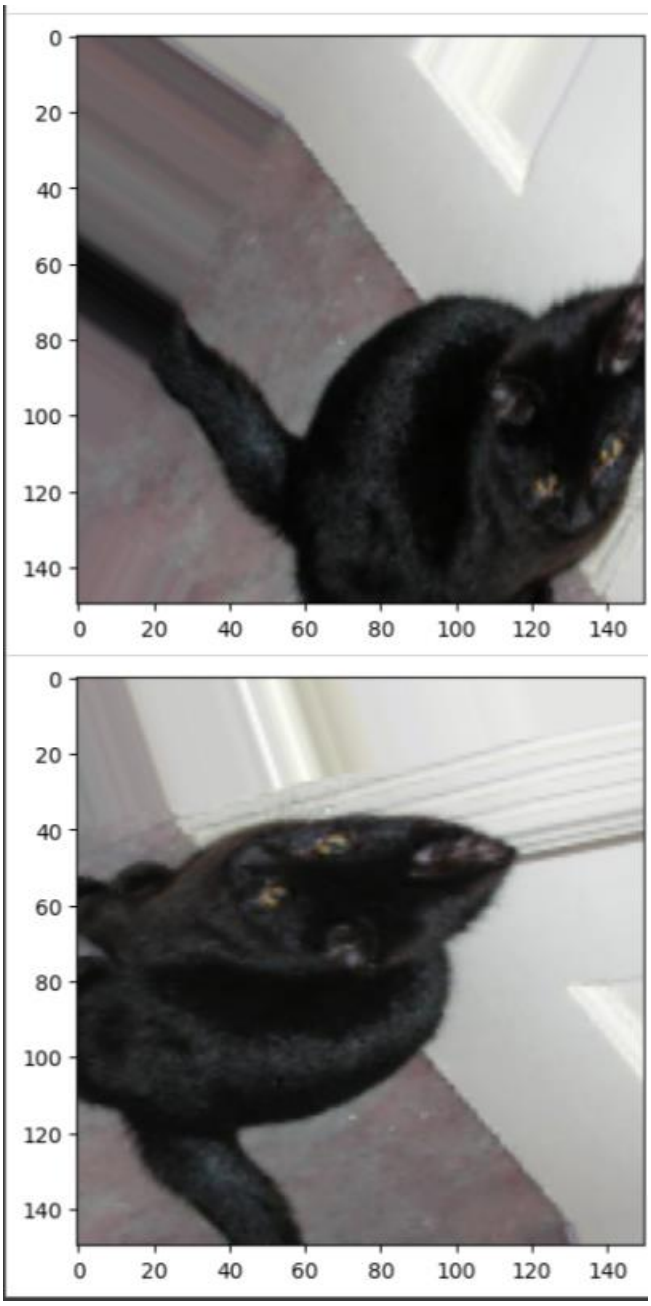
```
[15] history = model.fit_generator(
          train_generator,
          steps_per_epoch=100,
          epochs=30,
          validation_data=validation_generator,
          validation_steps=50)
```

```
Epoch 30/30
100/100 [==============================] - 6s 63ms/step - loss: 0.0229 - acc: 0.9945 - val_loss: 3.2653 - val_acc: 0.7260
```

- Got validation accuracy as 72% and there is overfitting as the accuracy changes.
- Used methods including regularization and data augmentation to lessen overfitting and enhance performance.
- Performance measures were kept track of, including accuracy and loss for training and validation sets.
- **Data Augmentation:** Data augmentation is a technique commonly used in deep learning to artificially increase the diversity of training data by applying various transformations to existing images. This helps prevent overfitting and improves the model's ability to generalize to unseen data.

[18]
```
datagen = ImageDataGenerator(
      rotation_range=90,
      width_shift_range=0.2,
      height_shift_range=0.2,
      shear_range=0.2,
      zoom_range=0.2,
      horizontal_flip=True,
      fill_mode='nearest')
```

```
# This is module with image preprocessing utilities
from keras.preprocessing import image

fnames = [os.path.join(train_cats_dir, fname) for fname in os.listdir(train_cats_dir)]

# We pick one image to "augment"
img_path = fnames[3]

# Read the image and resize it
img = image.load_img(img_path, target_size=(150, 150))

# Convert it to a Numpy array with shape (150, 150, 3)
x = image.img_to_array(img)

# Reshape it to (1, 150, 150, 3)
x = x.reshape((1,) + x.shape)

# The .flow() command below generates batches of randomly transformed images.
# It will loop indefinitely, so we need to `break` the loop at some point!
i = 0
for batch in datagen.flow(x, batch_size=1):
    plt.figure(i)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
    i += 1
    if i % 4 == 0:
        break

plt.show()
```



- After data augmentation I used the same dataset as before and got validation accuracy of 83%, which is a improvement from above validation accuracy.

```
Epoch 100/100
100/100 [==============================] - 20s 202ms/step - loss: 0.3368 - acc: 0.8575 - val_loss: 0.4467 - val_acc: 0.8380
```

Increasing Training Sample Size:

- Performance improved as training sample size increased, indicating the importance of data volume in training deep learning models.

```
# Copy first 1000 cat images to train_cats_dir
fnames = ['cat.{}.jpg'.format(i) for i in range(3000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(train_cats_dir, fname)
    shutil.copyfile(src, dst)

# Copy next 500 cat images to validation_cats_dir
fnames = ['cat.{}.jpg'.format(i) for i in range(1000, 1500)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(validation_cats_dir, fname)
    shutil.copyfile(src, dst)

# Copy next 500 cat images to test_cats_dir
fnames = ['cat.{}.jpg'.format(i) for i in range(1500, 2000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(test_cats_dir, fname)
    shutil.copyfile(src, dst)

# Copy first 1000 dog images to train_dogs_dir
fnames = ['dog.{}.jpg'.format(i) for i in range(3000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(train_dogs_dir, fname)
    shutil.copyfile(src, dst)

# Copy next 500 dog images to validation_dogs_dir
fnames = ['dog.{}.jpg'.format(i) for i in range(1000, 1500)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(validation_dogs_dir, fname)
    shutil.copyfile(src, dst)

# Copy next 500 dog images to test_dogs_dir
fnames = ['dog.{}.jpg'.format(i) for i in range(1500, 2000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(test_dogs_dir, fname)
    shutil.copyfile(src, dst)
```

```
[25] print('total training dog images:', len(os.listdir(train_dogs_dir)))
     print('total validation cat images:', len(os.listdir(validation_cats_dir)))
     print('total validation dog images:', len(os.listdir(validation_dogs_dir)))
     print('total test cat images:', len(os.listdir(test_cats_dir)))
     print('total test dog images:', len(os.listdir(test_dogs_dir)))

     total training dog images: 3000
     total validation cat images: 500
     total validation dog images: 500
     total test cat images: 500
     total test dog images: 500
```

```
Epoch 100/100
100/100 [==============================] - 20s 205ms/step - loss: 0.4056 - acc: 0.8200 - val_loss: 0.3183 - val_acc: 0.8590
```

- Validation accuracy increased to 85% which explains that larger sample sizes allowed the model to generalize better and capture more diverse features.

Optimizing Training Sample Size:

To optimize the training sample size and utilize data augmentation, we can follow these steps:

Increase Training Sample Size: Obtain a larger training dataset to provide more diverse examples for the model to learn from. This can involve collecting additional data or augmenting the existing dataset.

Data Augmentation: Apply various transformations to the training images to create additional synthetic data. These transformations can include rotations, shifts, flips, shearing, zooming, and more.

```
Epoch 100/100
100/100 [==============================] - 20s 204ms/step - loss: 0.3933 - acc: 0.8300 - val_loss: 0.2604 - val_acc: 0.8910
```

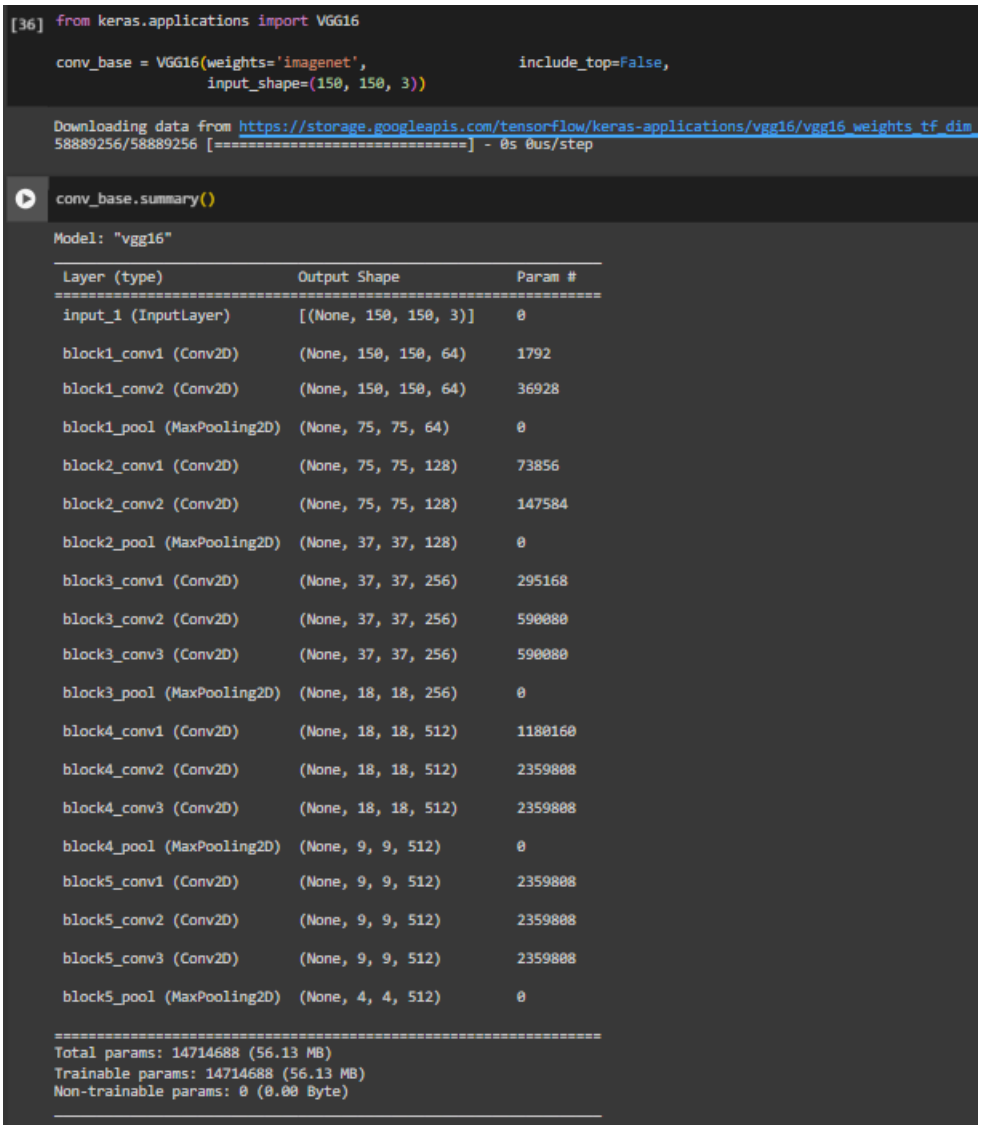
Validation accuracy increased to 89%. So we can optimize it by increasing size and data augmentation, changing batch sizes etc.

Pre-Trained Model:

VGG16:

A convolutional neural network (CNN) architecture called VGG16 was put forth by academics at the University of Oxford's Visual Geometry Group (VGG). In 2014, Karen Simonyan and Andrew Zisserman presented it in their paper "Very Deep Convolutional Networks for Large-Scale Image Recognition". VGG16 is a popular choice for picture classification applications because of its efficiency and ease of use.

VGG16's architecture is made up of sixteen completely connected, convolutional layers. Its simple structure is made up of three completely connected layers at the end, which are followed by max-pooling layers and convolutional layers.



Let's now analyze the model summary above:

Input Layer (Input_1): This layer establishes the pictures' input shape. Images having three channels (RGB) with a size of 150 x 150 pixels are anticipated in this scenario.

(block1_conv1, block1_conv2, block2_conv1, etc.) Convolutional Blocks The convolutional layers in question oversee taking features out of the input images. One or more convolutional layers are followed by rectified linear unit (ReLU) activations in each convolutional block.

Layers designated as MaxPooling2D (block1_pool, block2_pool, etc.): These layers carry out max-pooling procedures, lowering the feature maps' spatial dimensions while keeping the most crucial data intact.

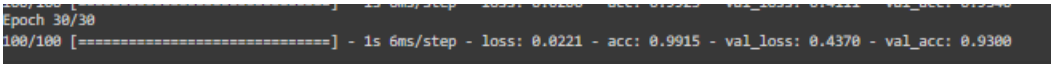
Total Params: This indicates the total number of parameters (weights and biases) in the model.

VGG16 has many parameters due to its deep architecture, totaling approximately 14.7 million parameters.

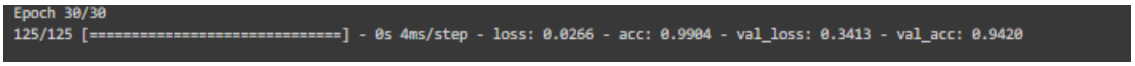
The quantity of parameters that can be trained during the training process is indicated by the term "trainable parameters." Since the value equals the total number of parameters in this instance, all of the model's parameters are trainable.

The number of parameters that are not trainable is indicated by the term "non-trainable parameters." Since all the parameters are updated during training, there are no non-trainable parameters in this instance.

4.1: I used VGG16 as a pre-trained model and got an accuracy as 93% for 2000 training data, 1000 for validation and test data and by increasing the data size we can achieve more accuracy than this. Accuracy is increased to 93% using pre-trained model.



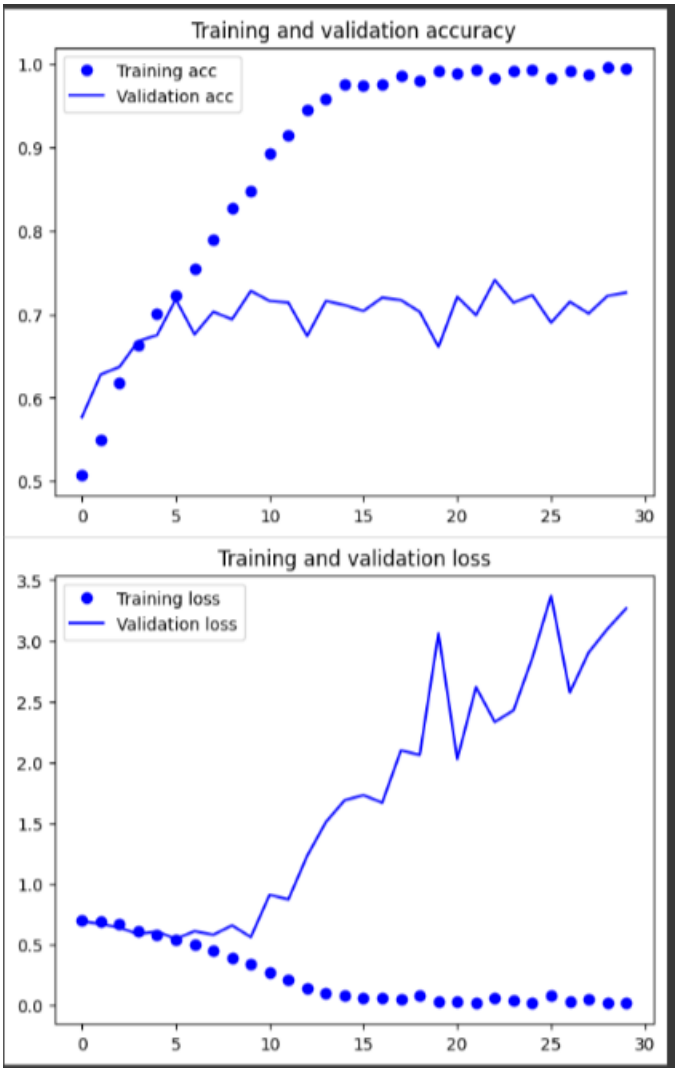
We will increase the training data to 2500, test and validation remains same as 1000 data values which increased the validation accuracy to 94.2%.



We increased the training data to 3300 data values. And the validation accuracy is increased to 95.8%.

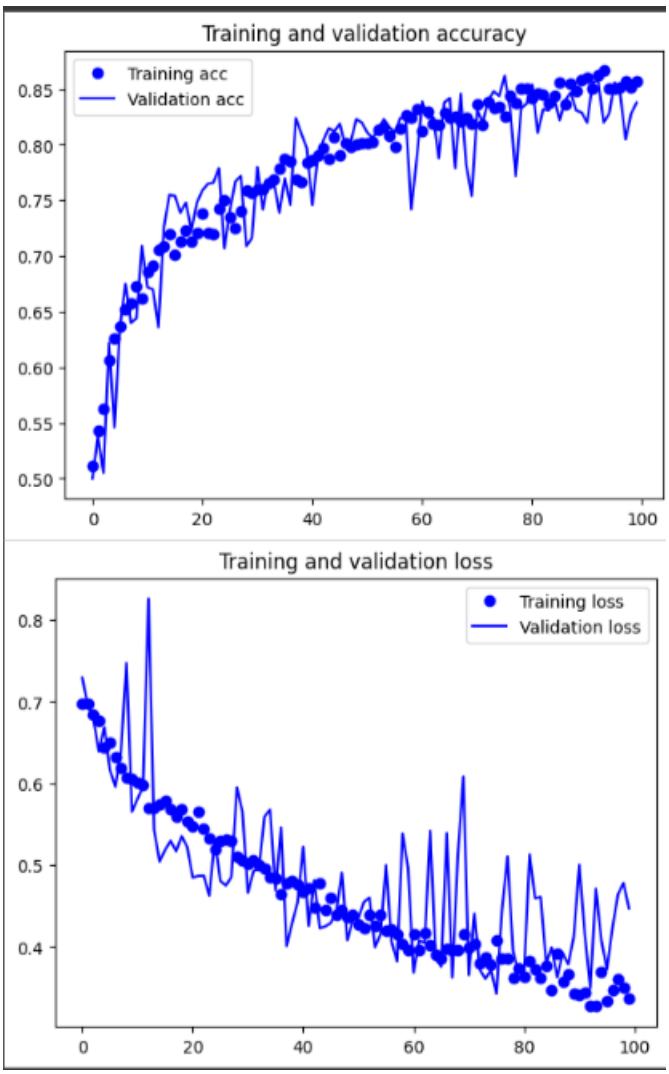
Results:

Training from Scratch:



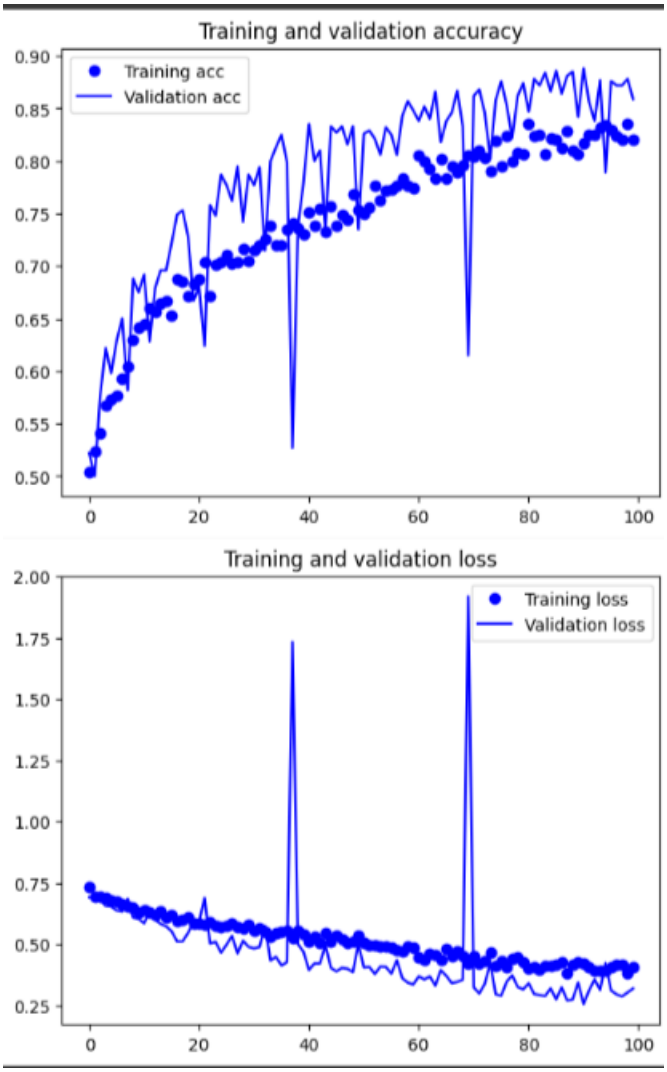
From the above graphs we can find the model is overfitting and there are more discrepancies in the accuracy and loss and validation accuracy as 72%.

Data Augmentation:



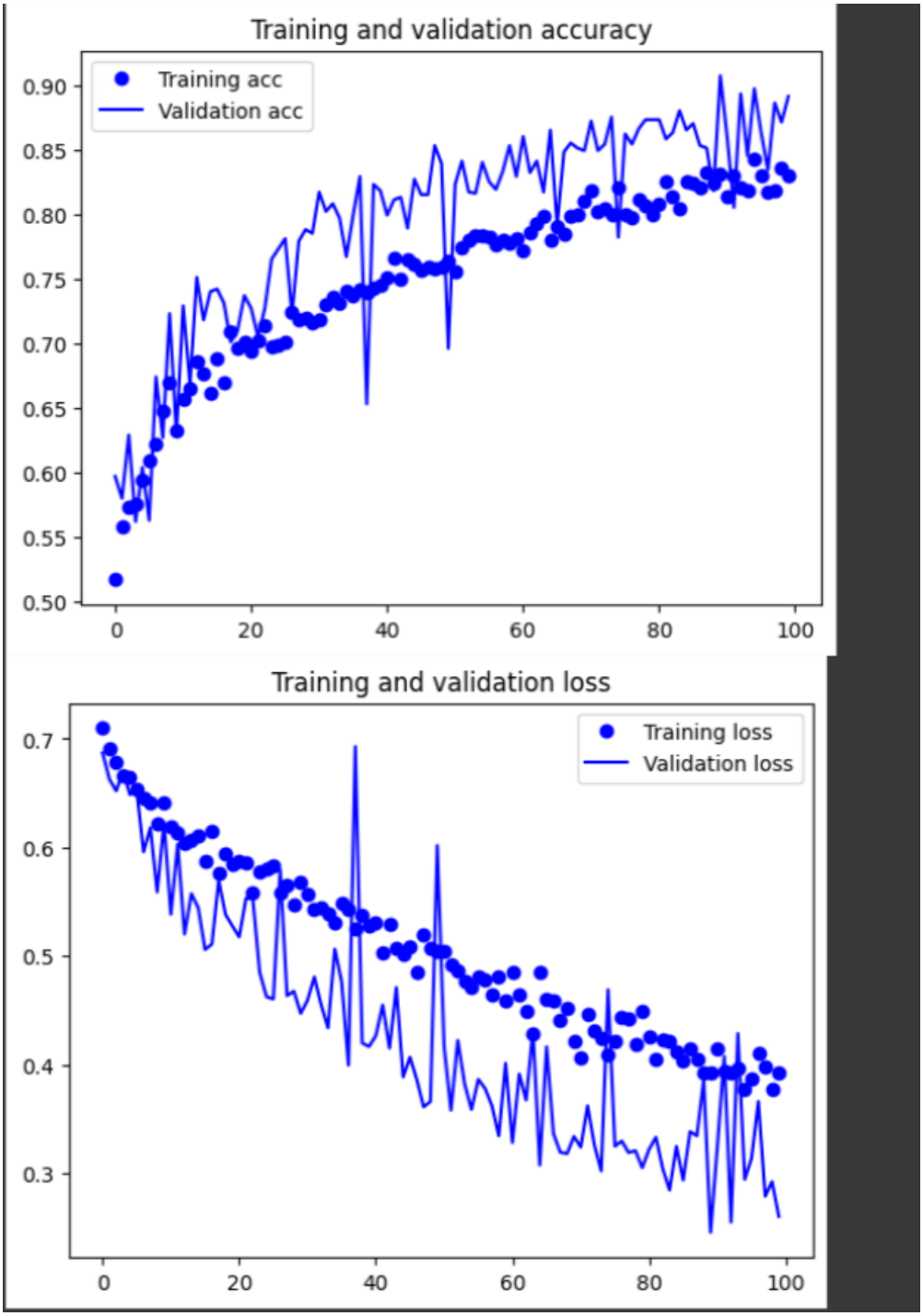
Validation accuracy increased from 72% to 83% and there are less discrepancies to the epoch ratio.

Increasing Training Sample Size:



Validation accuracy is increased and there are less discrepancies than above results.

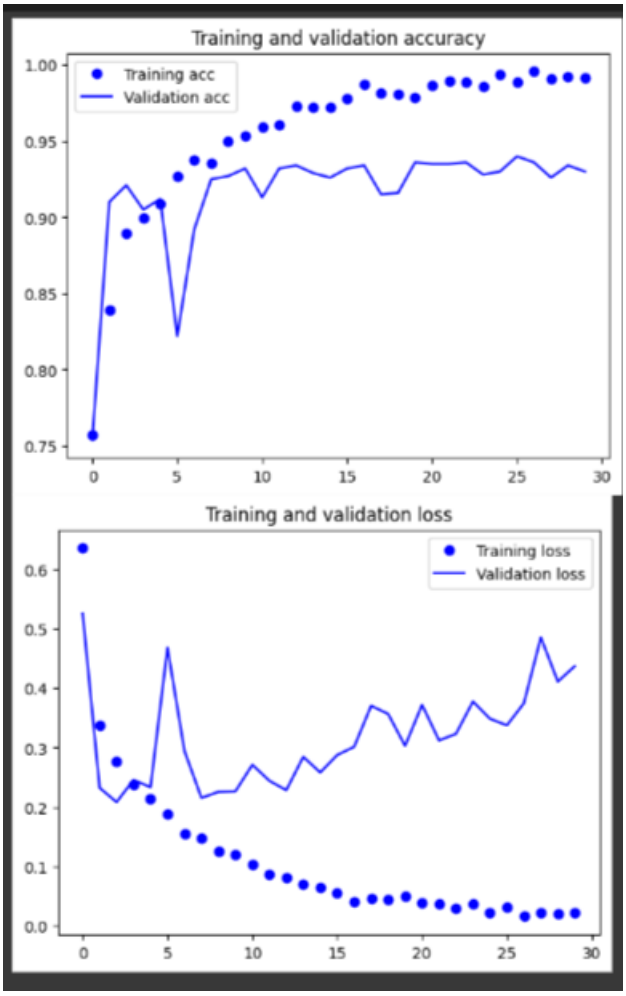
Optimizing Training Sample Size:



Validation accuracy has increased from 83% to 89%.

Pre-Trained Model:

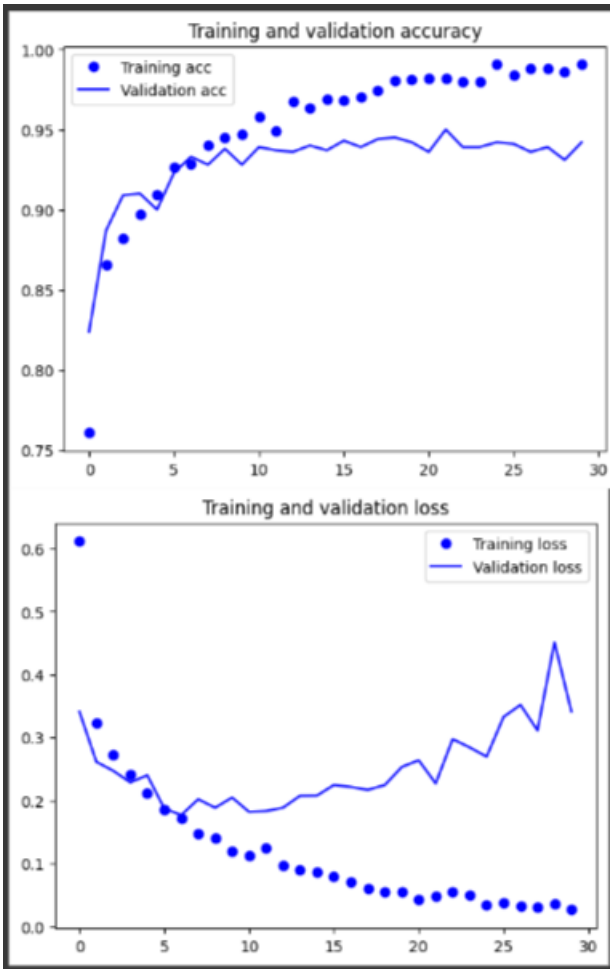
I used VGG16 as a pre-trained model and got an accuracy as 93% for 2000 training data, 1000 for validation and test data and by increasing the data size we can achieve more accuracy than this. Accuracy is increased to 93% using pre-trained model.



Overfitting is happening as the graph has many discrepancies.

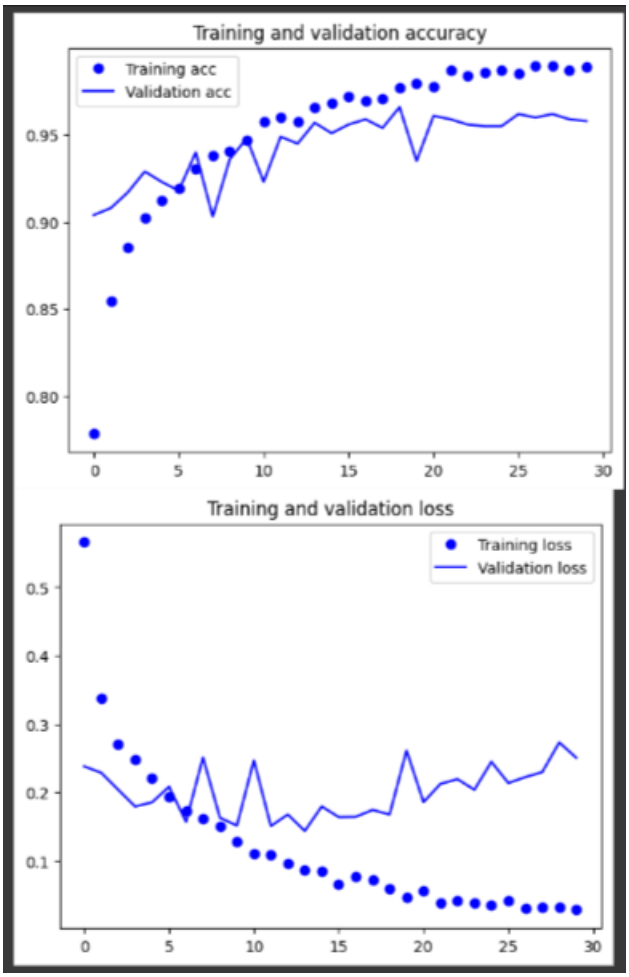
Increasing data size:

We will increase the training data to 2500, test and validation remains same as 1000 data values which increased the validation accuracy to 94.2%.



Lesser overfitting occurred as the training size is increased to 2500 data values.

Increased to 3300 data values:



We increased the training data to 3300 data values. And the validation accuracy is increased to 95.8%.

Conclusion:

- The resources at hand and the required performance level determine whether to train a network from the start or use one that has already been pretrained.
- Performance is usually enhanced by larger training sample sizes, but there is a point of diminishing returns beyond which more data may not produce appreciable improvements.
- Because trained networks may leverage the information from large-scale datasets, they present a tempting solution, especially when working with sparse data.
- Pretrained models can perform competitively when fine-tuned with smaller sample numbers than when bigger datasets are used for initial training.