ASSIGNMENT-2

TRAINING A CONVNET FROM SCRATCH ON A SMALL DATASET

```
import zipfile
```

```
zip_file_path = "/fs/ess/PGS0333/BA_64061_KSU/data/dogs-vs-cats.zip"
```

Basic Convnet from Scratch with small data:

Q1. Start initially with a training sample of 1000, a validation sample of 500, and a test sample of 500 (like in the text). Use any technique to reduce overfitting and improve performance in developing a network that you train from scratch. What performance did you achieve?

```
import os, shutil, pathlib

original_dir = pathlib.Path("./archive/train/train/")
new_base_dir = pathlib.Path("./C_vs_dogs_small")

def make_subset(subset_name, start_index, end_index):
    for category in ("cat", "dog"):
        dir = new_base_dir / subset_name / category
        os.makedirs(dir)
        fnames = [f"{category}.{i}.jpg" for i in range(start_index, end_index)]
        for fname in fnames:
            shutil.copyfile(src=original_dir / fname,
                            dst=dir / fname)

make_subset("train", start_index=0, end_index=2000)
make_subset("validation", start_index=2000, end_index=2500)
make_subset("test", start_index=2500, end_index=3000)
```

We took a subset of the dataset and divided the images into three folders, namely train, validate, and test. Using the function make a subset.

```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(180, 180, 3))
x = layers.Rescaling(1./255)(inputs)
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=256, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=256, kernel_size=3, activation="relu")(x)
x = layers.Flatten()(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

In order to illustrate the CNN architecture, we stack a number of layers in this code: an input layer, followed by a feature rescaling layer, a 2-Dimensional convolution layer with Maxpooling, and a single dense layer with a sigmoid function.

```
Model: "model_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_7 (InputLayer)        [(None, 180, 180, 3)]     0

 rescaling_2 (Rescaling)     (None, 180, 180, 3)       0

 conv2d_10 (Conv2D)          (None, 178, 178, 32)      896

 max_pooling2d_8 (MaxPooling  (None, 89, 89, 32)       0
 2D)

 conv2d_11 (Conv2D)          (None, 87, 87, 64)        18496

 max_pooling2d_9 (MaxPooling  (None, 43, 43, 64)       0
 2D)

 conv2d_12 (Conv2D)          (None, 41, 41, 128)       73856

 max_pooling2d_10 (MaxPoolin  (None, 20, 20, 128)      0
 g2D)

 conv2d_13 (Conv2D)          (None, 18, 18, 256)       295168

 max_pooling2d_11 (MaxPoolin  (None, 9, 9, 256)        0
 g2D)

 conv2d_14 (Conv2D)          (None, 7, 7, 256)         590080

 flatten_4 (Flatten)         (None, 12544)             0

 dense_6 (Dense)             (None, 1)                 12545

=================================================================
Total params: 991,041
Trainable params: 991,041
Non-trainable params: 0
_________________________________________________________________
```

Model.summary() explains information about the structure of CNN

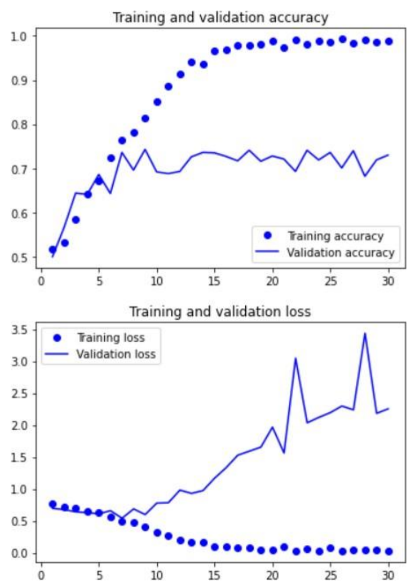
```
In [ ]: model.compile(loss="binary_crossentropy",
                      optimizer="rmsprop",
                      metrics=["accuracy"])
```

rmsprop is the optimizer and binary_crossentrophy is the loss factor.

```
In [ ]: from tensorflow.keras.utils import image_dataset_from_directory

        train_dataset = image_dataset_from_directory(
            new_base_dir / "train",
            image_size=(180, 180),
            batch_size=32)
        validation_dataset = image_dataset_from_directory(
            new_base_dir / "validation",
            image_size=(180, 180),
            batch_size=32)
        test_dataset = image_dataset_from_directory(
            new_base_dir / "test",
            image_size=(180, 180),
            batch_size=32)
```

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

Here, we are training with 30 epochs and validate with validation set.



It has been observed that an overfitting model is evident when it exhibits good performance on training data but poor performance on validation or test data.

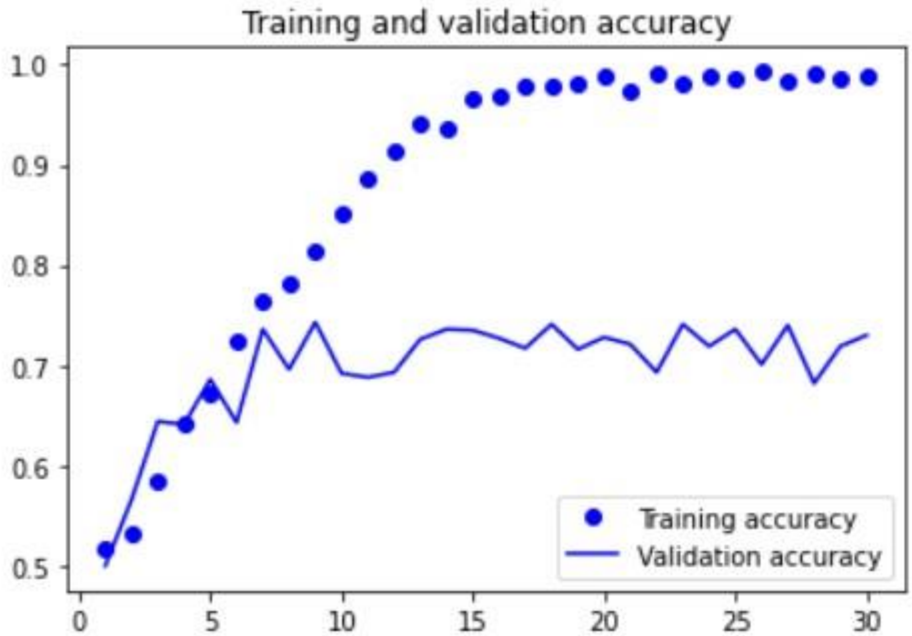
Methods to solve overfitting problems:

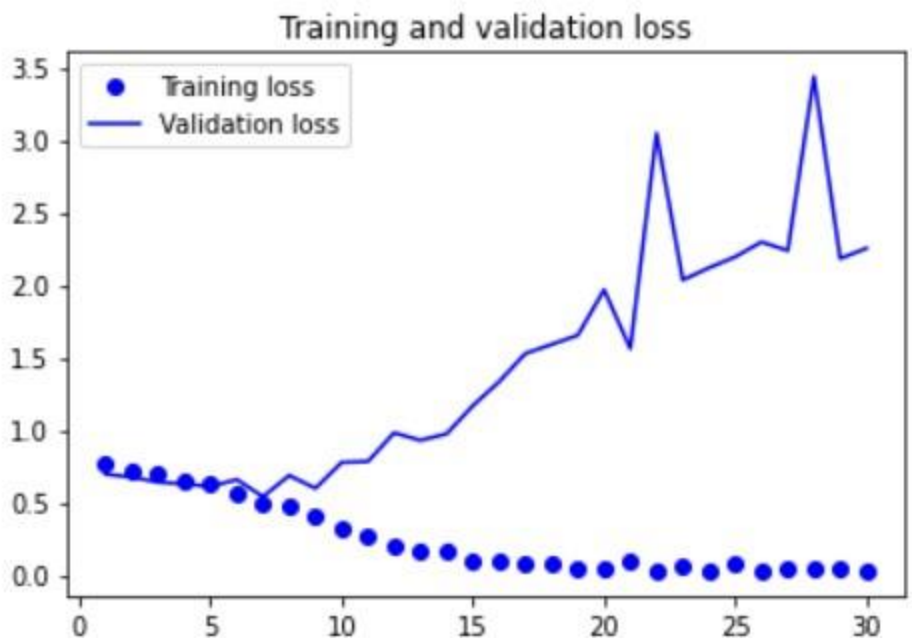
- Train with more data
- Data augmentation
- Addition of noise to the input data

- Feature selection
- Cross-validation
- Simplify data
- Regularization
- Ensembling
- Add Dropouts

Basic Convnet from Scratch with 2000 training samples and 500 validation and testing data:

```
Epoch 25/30
63/63 [==============================] - 5s 71ms/step - loss: 0.0767 - accuracy: 0.9870 - val_loss: 2.1934 - val_accuracy: 0.7360
Epoch 26/30
63/63 [==============================] - 5s 71ms/step - loss: 0.0274 - accuracy: 0.9930 - val_loss: 2.2964 - val_accuracy: 0.7010
Epoch 27/30
63/63 [==============================] - 5s 72ms/step - loss: 0.0441 - accuracy: 0.9840 - val_loss: 2.2365 - val_accuracy: 0.7400
Epoch 28/30
63/63 [==============================] - 5s 70ms/step - loss: 0.0377 - accuracy: 0.9905 - val_loss: 3.4371 - val_accuracy: 0.6820
Epoch 29/30
63/63 [==============================] - 5s 71ms/step - loss: 0.0441 - accuracy: 0.9865 - val_loss: 2.1819 - val_accuracy: 0.7190
Epoch 30/30
63/63 [==============================] - 5s 71ms/step - loss: 0.0300 - accuracy: 0.9895 - val_loss: 2.2526 - val_accuracy: 0.7300
```





Evaluating the model on the test set:

In [ ]:
```
test_model = keras.models.load_model("convnet_from_scratch.keras")
test_loss, test_acc = test_model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```

```
32/32 [==============================] - 2s 36ms/step - loss: 0.5702 - accuracy: 0.6980
Test accuracy: 0.698
```

Here is the summary for the train, test, validation accuracy for: Training Accuracy:98.95% Test accuracy:69.80% Validation Accuracy:70%

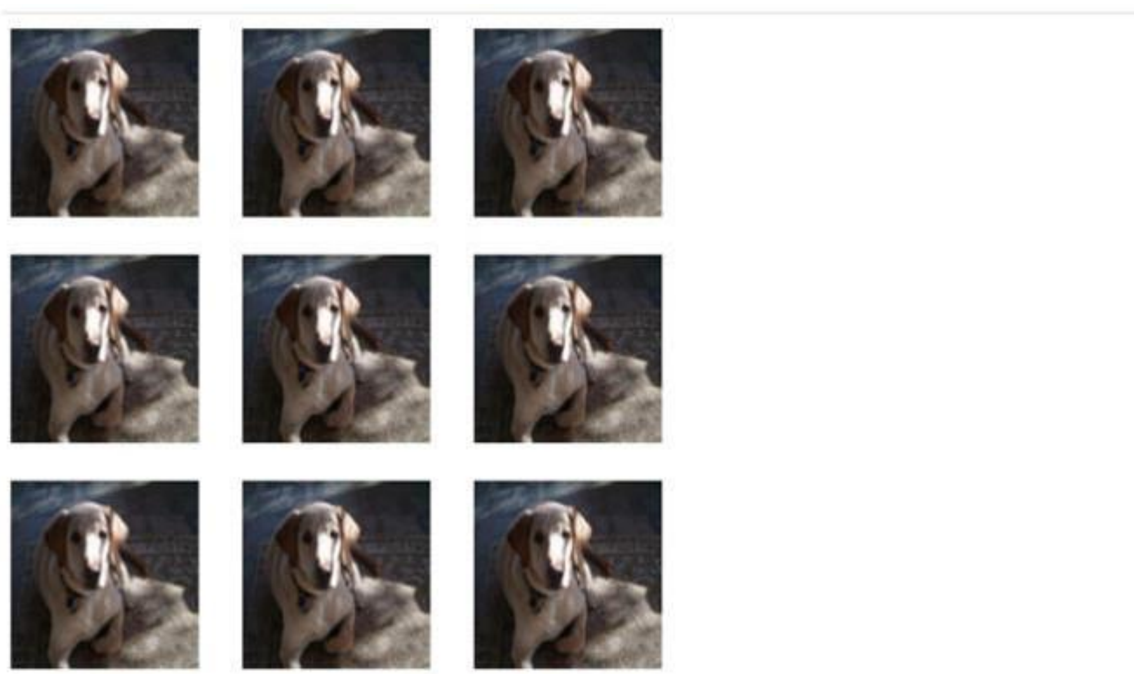
Basic Convnet with Data Augmentation and Dropouts

In [ ]:

```
data_augmentation = keras.Sequential(
    [
        layers.RandomFlip("horizontal"),
        layers.RandomRotation(0.1),
        layers.RandomZoom(0.2),
    ]
)
```

“ADAM” is considered as the best optimizer and used dropouts to avoid overfitting.

- Here we see how data augmentation effects and solves the overfitting with dropout layers.



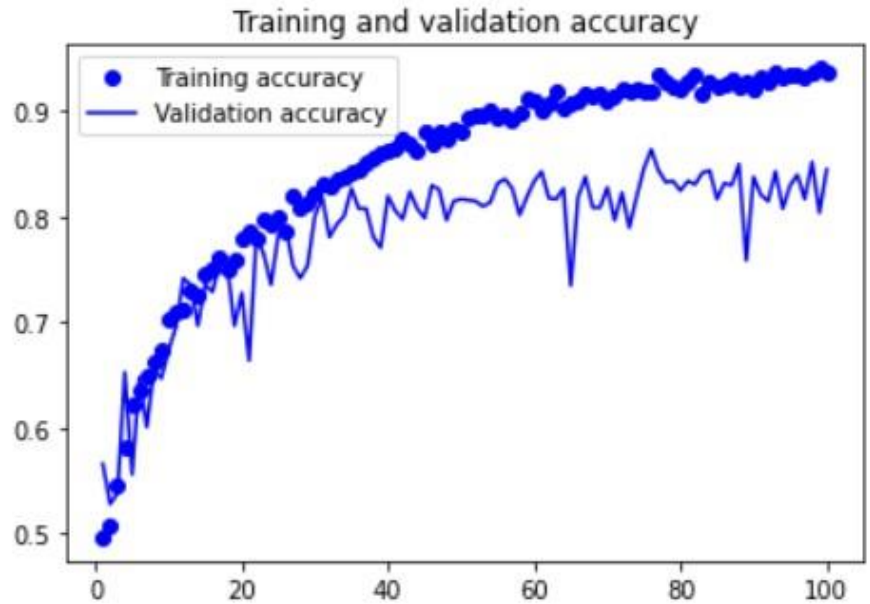
These are random images with data augmentation flips and rotations.

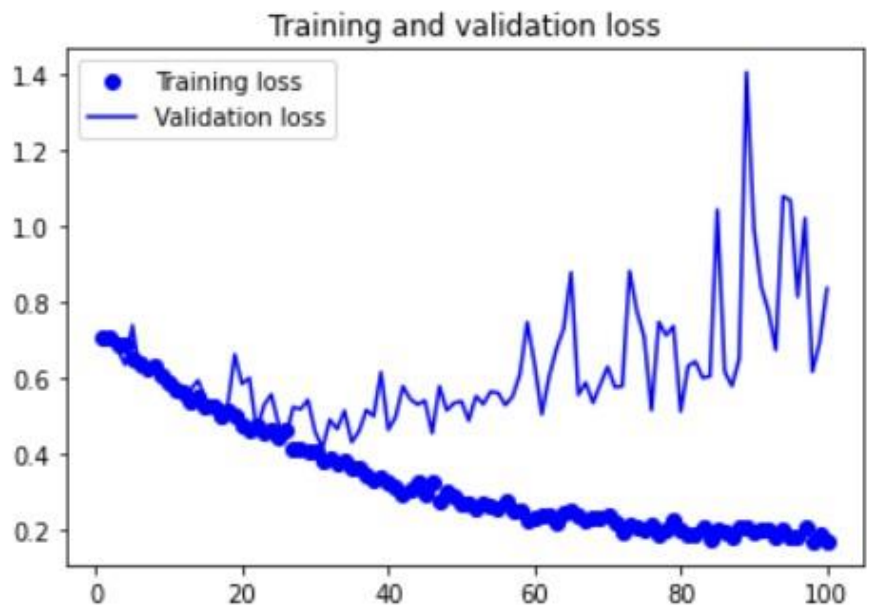
```
Epoch 95/100
63/63 [==============================] - 6s 97ms/step - loss: 0.1792 - accuracy: 0.9355 - val_loss: 1.0679 - val_accuracy: 0.8300
Epoch 96/100
63/63 [==============================] - 6s 95ms/step - loss: 0.1795 - accuracy: 0.9340 - val_loss: 0.8136 - val_accuracy: 0.8400
Epoch 97/100
63/63 [==============================] - 6s 96ms/step - loss: 0.2081 - accuracy: 0.9335 - val_loss: 1.0210 - val_accuracy: 0.8170
Epoch 98/100
63/63 [==============================] - 6s 98ms/step - loss: 0.1711 - accuracy: 0.9360 - val_loss: 0.6162 - val_accuracy: 0.8520
Epoch 99/100
63/63 [==============================] - 6s 96ms/step - loss: 0.1857 - accuracy: 0.9410 - val_loss: 0.6931 - val_accuracy: 0.8040
Epoch 100/100
63/63 [==============================] - 6s 98ms/step - loss: 0.1694 - accuracy: 0.9360 - val_loss: 0.8357 - val_accuracy: 0.8450
```

Here, we can see the train accuracy of 93.60 and validation accuracy of 84.50. We here notice a substantial difference and determine that our overfitting issue has been resolved by this method.

Basic Convnet from scratch with Dropout and Data Augmentation with more training, validation, and test samples:


```
Epoch 95/100
63/63 [==============================] - 6s 97ms/step - loss: 0.1792 - accuracy: 0.9355 - val_loss: 1.0679 - val_accuracy: 0.8300
Epoch 96/100
63/63 [==============================] - 6s 95ms/step - loss: 0.1795 - accuracy: 0.9340 - val_loss: 0.8136 - val_accuracy: 0.8400
Epoch 97/100
63/63 [==============================] - 6s 96ms/step - loss: 0.2081 - accuracy: 0.9335 - val_loss: 1.0210 - val_accuracy: 0.8170
Epoch 98/100
63/63 [==============================] - 6s 98ms/step - loss: 0.1711 - accuracy: 0.9360 - val_loss: 0.6162 - val_accuracy: 0.8520
Epoch 99/100
63/63 [==============================] - 6s 96ms/step - loss: 0.1857 - accuracy: 0.9410 - val_loss: 0.6931 - val_accuracy: 0.8040
Epoch 100/100
63/63 [==============================] - 6s 98ms/step - loss: 0.1694 - accuracy: 0.9360 - val_loss: 0.8357 - val_accuracy: 0.8450
```





Let's now compare the cases for the network that is created from scratch.

Instance	Training Accuracy	Validation Accuracy	Training Loss	Validation Loss	Test accuracy	Observations
Basic Convnet from scratch (no dropout, data augmentation)	98.80	70.50	0.05	3.05	68.80	Here we have an overfitting problem since data is working good

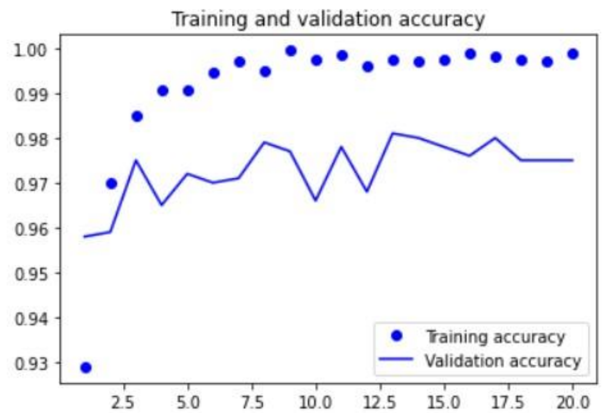
						for training but not for test.
Basic Convnet with data augmentation and dropouts	98.95	84.50	0.16	0.49	73.00	Here the model is showing good results with training and validation loss and accuracy. Though our training accuracy is reduced to 93.
Convnet with more training samples, validation, and test samples	93.60	77.20	0.03	1.85	84.50	When adding more data, overfitting is being reduced.
With Data Augmentation and dropout with more training, validation and test samples	93.63	86.30	0.4	0.46	84.50	In this case, it is weird that our results accuracy has dropped and its is more consistent.

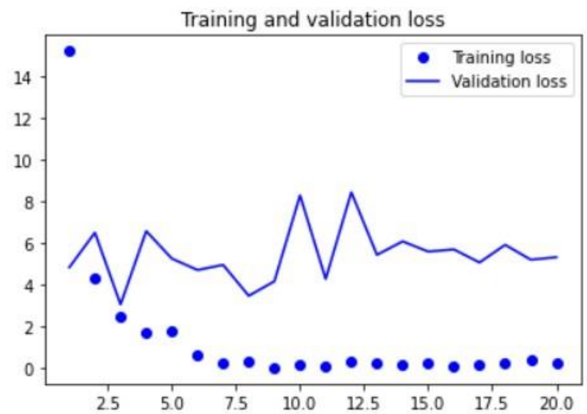
With Pretrained Network:

Using pretrained VGG16 network.

[ ]

```
Model: "vgg16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_5 (InputLayer)        [(None, None, None, 3)]   0

 block1_conv1 (Conv2D)       (None, None, None, 64)    1792

 block1_conv2 (Conv2D)       (None, None, None, 64)    36928

 block1_pool (MaxPooling2D)  (None, None, None, 64)    0

 block2_conv1 (Conv2D)       (None, None, None, 128)   73856

 block2_conv2 (Conv2D)       (None, None, None, 128)   147584

 block2_pool (MaxPooling2D)  (None, None, None, 128)   0

 block3_conv1 (Conv2D)       (None, None, None, 256)   295168

 block3_conv2 (Conv2D)       (None, None, None, 256)   590080

 block3_conv3 (Conv2D)       (None, None, None, 256)   590080

 block3_pool (MaxPooling2D)  (None, None, None, 256)   0

 block4_conv1 (Conv2D)       (None, None, None, 512)   1180160

 block4_conv2 (Conv2D)       (None, None, None, 512)   2359808

 block4_conv3 (Conv2D)       (None, None, None, 512)   2359808

 block4_pool (MaxPooling2D)  (None, None, None, 512)   0

 block5_conv1 (Conv2D)       (None, None, None, 512)   2359808

 block5_conv2 (Conv2D)       (None, None, None, 512)   2359808

 block5_conv3 (Conv2D)       (None, None, None, 512)   2359808

 block5_pool (MaxPooling2D)  (None, None, None, 512)   0

=================================================================
Total params: 14,714,688
```



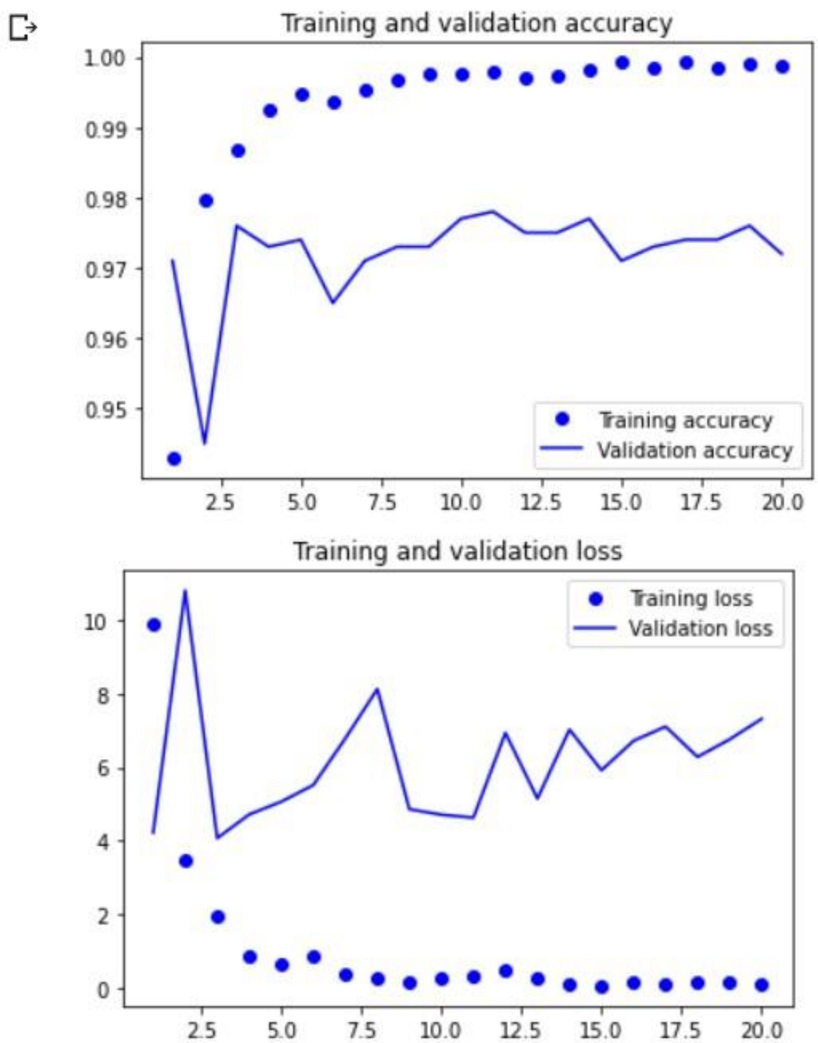


Case: Using VGG16 as base with data augmentation and dropout layer

```
Epoch 46/50
63/63 [==============================] - 14s 213ms/step - loss: 0.2972 - accuracy: 0.9950 - val_loss: 2.0507 - val_accuracy: 0.9800
Epoch 47/50
63/63 [==============================] - 14s 213ms/step - loss: 0.6939 - accuracy: 0.9850 - val_loss: 2.3633 - val_accuracy: 0.9750
Epoch 48/50
63/63 [==============================] - 14s 213ms/step - loss: 0.5183 - accuracy: 0.9890 - val_loss: 2.4613 - val_accuracy: 0.9780
Epoch 49/50
63/63 [==============================] - 13s 206ms/step - loss: 0.7485 - accuracy: 0.9865 - val_loss: 2.6667 - val_accuracy: 0.9760
Epoch 50/50
63/63 [==============================] - 13s 207ms/step - loss: 0.5266 - accuracy: 0.9890 - val_loss: 2.4629 - val_accuracy: 0.9790
```

Case: Using VGG16 as base with more training, validation and test samples

```
63/63 [==============================] - 14s 211ms/step - loss: 0.6261 - accuracy: 0.9910 - val_loss: 2.7284 - val_accuracy: 0.9770
Epoch 46/50
63/63 [==============================] - 14s 213ms/step - loss: 0.2972 - accuracy: 0.9950 - val_loss: 2.0507 - val_accuracy: 0.9800
Epoch 47/50
63/63 [==============================] - 14s 213ms/step - loss: 0.6939 - accuracy: 0.9850 - val_loss: 2.3633 - val_accuracy: 0.9750
Epoch 48/50
63/63 [==============================] - 14s 213ms/step - loss: 0.5183 - accuracy: 0.9890 - val_loss: 2.4613 - val_accuracy: 0.9780
Epoch 49/50
63/63 [==============================] - 13s 206ms/step - loss: 0.7485 - accuracy: 0.9865 - val_loss: 2.6667 - val_accuracy: 0.9760
Epoch 50/50
63/63 [==============================] - 13s 207ms/step - loss: 0.5266 - accuracy: 0.9890 - val_loss: 2.4629 - val_accuracy: 0.9790
```

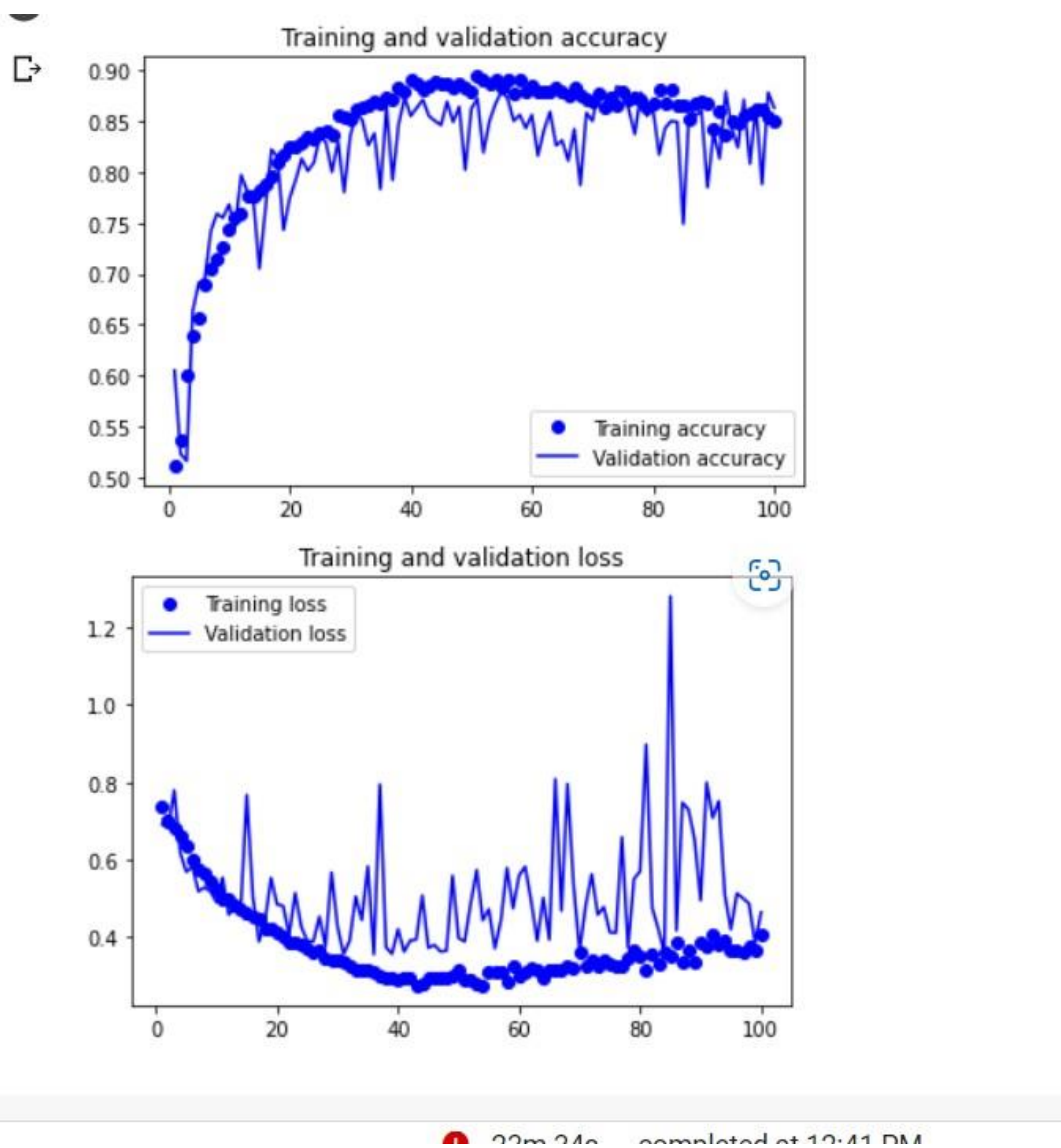


Basic convnet with ADAM:

```
125/125 [==============================] - 8s 63ms/step - loss: 0.0165 - accuracy: 0.9948 - val_loss: 1.3820 - val_accuracy: 0.7780
Epoch 23/30
125/125 [==============================] - 8s 63ms/step - loss: 0.0320 - accuracy: 0.9908 - val_loss: 1.1682 - val_accuracy: 0.7920
Epoch 24/30
125/125 [==============================] - 8s 64ms/step - loss: 0.0635 - accuracy: 0.9770 - val_loss: 1.1841 - val_accuracy: 0.7930
Epoch 25/30
125/125 [==============================] - 8s 63ms/step - loss: 0.0188 - accuracy: 0.9935 - val_loss: 1.1469 - val_accuracy: 0.7940
Epoch 26/30
125/125 [==============================] - 8s 63ms/step - loss: 0.0053 - accuracy: 0.9987 - val_loss: 1.1634 - val_accuracy: 0.7770
Epoch 27/30
125/125 [==============================] - 8s 62ms/step - loss: 0.0028 - accuracy: 0.9995 - val_loss: 1.2016 - val_accuracy: 0.7930
Epoch 28/30
125/125 [==============================] - 8s 62ms/step - loss: 5.9875e-04 - accuracy: 1.0000 - val_loss: 1.1786 - val_accuracy: 0.8080
Epoch 29/30
125/125 [==============================] - 8s 63ms/step - loss: 2.5744e-04 - accuracy: 1.0000 - val_loss: 1.2075 - val_accuracy: 0.8060
Epoch 30/30
125/125 [==============================] - 8s 63ms/step - loss: 1.8093e-04 - accuracy: 1.0000 - val_loss: 1.2300 - val_accuracy: 0.8080
```

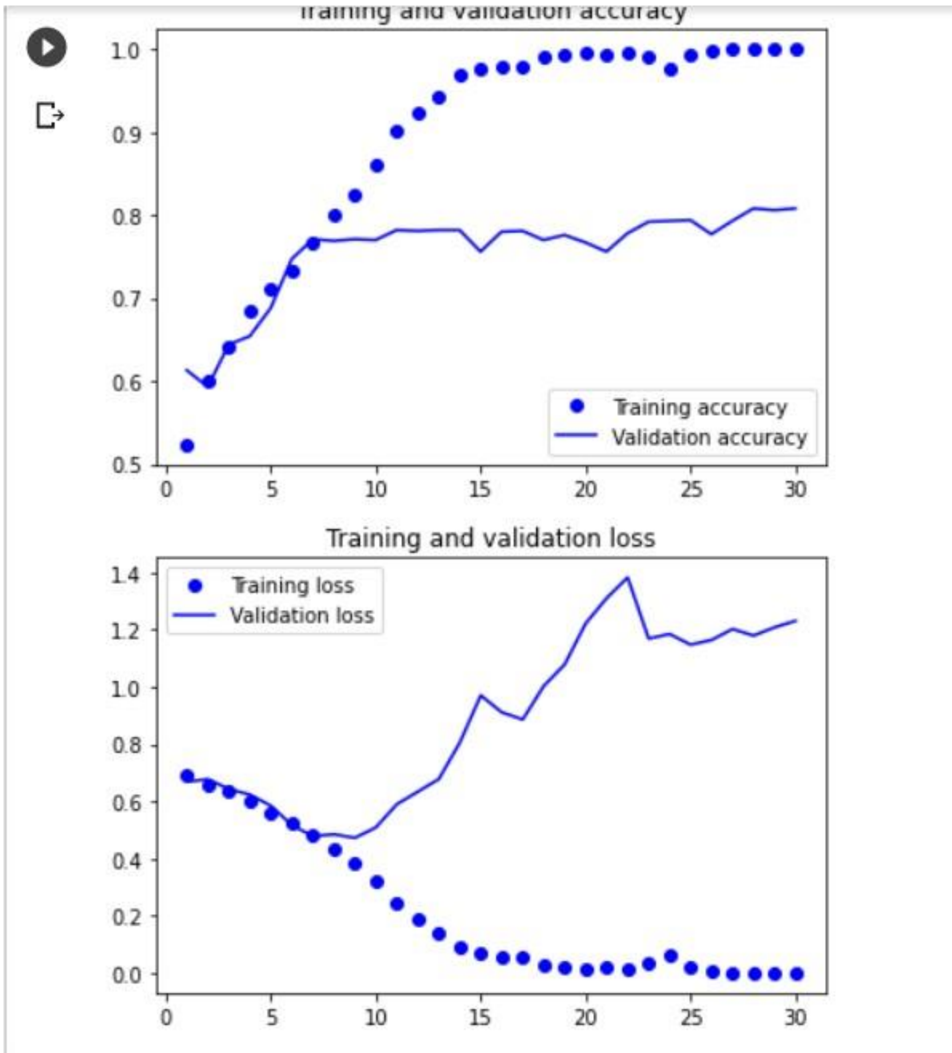
Basic convnet with data augmentation:

```
Epoch 96/100
125/125 [==============================] - 11s 90ms/step - loss: 0.1149 - accuracy: 0.9542 - val_loss: 0.3750 - val_accuracy: 0.9000
Epoch 97/100
125/125 [==============================] - 11s 89ms/step - loss: 0.1179 - accuracy: 0.9553 - val_loss: 0.3860 - val_accuracy: 0.8950
Epoch 98/100
125/125 [==============================] - 12s 90ms/step - loss: 0.1169 - accuracy: 0.9515 - val_loss: 0.3710 - val_accuracy: 0.8880
Epoch 99/100
125/125 [==============================] - 11s 90ms/step - loss: 0.0935 - accuracy: 0.9657 - val_loss: 0.4336 - val_accuracy: 0.8950
Epoch 100/100
125/125 [==============================] - 11s 89ms/step - loss: 0.1235 - accuracy: 0.9538 - val_loss: 0.4042 - val_accuracy: 0.8950
```



Case: VGG16 with ADAM:

```
Epoch 16/20
125/125 [==============================] - 1s 5ms/step - loss: 0.0530 - accuracy: 0.9992 - val_loss: 6.0441 - val_accuracy: 0.9720
Epoch 17/20
125/125 [==============================] - 1s 5ms/step - loss: 0.0355 - accuracy: 0.9992 - val_loss: 10.0121 - val_accuracy: 0.9700
Epoch 18/20
125/125 [==============================] - 1s 5ms/step - loss: 0.2625 - accuracy: 0.9983 - val_loss: 5.9547 - val_accuracy: 0.9720
Epoch 19/20
125/125 [==============================] - 1s 5ms/step - loss: 0.1143 - accuracy: 0.9987 - val_loss: 8.8187 - val_accuracy: 0.9620
Epoch 20/20
125/125 [==============================] - 1s 5ms/step - loss: 0.1603 - accuracy: 0.9983 - val_loss: 7.1673 - val_accuracy: 0.9720
```




```
test_model = keras.models.load_model("convnet_from_scratch.keras")
test_loss, test_acc = test_model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```

```
32/32 [==============================] - 2s 37ms/step - loss: 0.5332 - accuracy: 0.7720
Test accuracy: 0.772
```

Overall Summary:

Let's now compare for the network that is created from scratch.

Instance	Training Accuracy	Validation Accuracy	Training Loss	Validation Loss	Test accuracy	Observations
Basic Convnet from scratch (no dropout, data augmentation)	98.80	70.50	0.05	3.05	68.80	Here we have an overfitting problem since data is working good for training but not for test.
Basic Convnet with data augmentation and dropouts	98.95	84.50	0.16	0.49	73.00	Here the model is showing good results with training and validation loss and accuracy. Though our training accuracy is reduced to 93.

Convnet with more training samples, validation, and test samples	93.60	77.20	0.03	1.85	84.50	When adding more data, overfitting is being reduced.
With Data Augmentation and dropout with more training, validation and test samples	93.63	86.30	0.4	0.46	97.20	In this case, it is weird that our results accuracy has dropped and it is more consistent.

Pretrained Network - VGG16 Cases:

Cases	Training Accuracy	Validation Accuracy	Training Loss	Validation Loss	Test accuracy	Observations
Using VGG16 as base	98.90	97.90	0.5266	2.46	97.60	The result was good using the VGG16

						and validation loss is more that can be reduced with some optimizations.
Using VGG16 with data augmentation and dropouts	99.80	98.20	0.05	1.6	97.80	Here Accuracy is increased, and Validation loss is decreased.
Using VGG16 as base with more training, validation, and test	95.38	97.90	0.4	2.4	89.50	It also shows the good result, but it has validation loss
With Data Augmentation & Dropouts	99.83	98.20	0.04	0.8	98.20	Best result so far with optimizations & Data Augmentation techniques.

Conclusion: In this concluding section, I have undertaken a comparative analysis between a basic Convolutional Neural Network (Convnet) and the more advanced VGG16 network. Initially, the basic Convnet displayed overfitting issues, which were successfully mitigated through the implementation of data augmentation and dropout techniques, leading to notable accuracy improvements. Subsequently, incorporating additional data, including test samples, resulted in enhanced training and testing accuracy.

Moreover, I conducted experiments with various advanced optimizers to assess their performance. These optimizations collectively contributed to improved accuracy and reduced loss in the model. Later on, I transitioned to utilizing the VGG16 architecture as the foundational model and conducted training. The preliminary results proved promising, with an observed accuracy surpassing 95%. Through the implementation of additional techniques, this accuracy was further significantly enhanced.