EE 628 – Final Project

Option 2

Use the below-assigned image classification task.

Task:

In this competition, you'll write an algorithm to classify whether images contain either a dog or a cat. This is easy for humans, dogs, and cats. Your computer will find it a bit more difficult.

You can exploit all the skill sets you have learned in this class to practice your model experience.

Data Description:

The training archive contains 25,000 images of dogs and cats. Train your algorithm on these files and predict the labels for test1.zip (1 = dog, 0 = cat).

Kaggle URL for training and testing data download:

https://www.kaggle.com/c/dogs-vs-cats

Solution:

Problem Statement:

Implementing an algorithm to classify whether images contain either a dog or a cat which is more difficult for computer**.**

Data Set Description:

Source: Kaggle - https://www.kaggle.com/c/dogs-vs-cats

The Asirra data set

Web services are often protected with a challenge that's supposed to be easy for people to solve, but difficult for computers. Such a challenge is often called a CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) or HIP (Human Interactive Proof). HIPs are used for many purposes, such as to reduce email and blog spam and prevent brute-force attacks on web site passwords.

Asirra (Animal Species Image Recognition for Restricting Access) is a HIP that works by asking users to identify photographs of cats and dogs. This task is difficult for computers, but studies have shown that people can accomplish it quickly and accurately. Many even think it's fun! Here is an example of the Asirra interface:

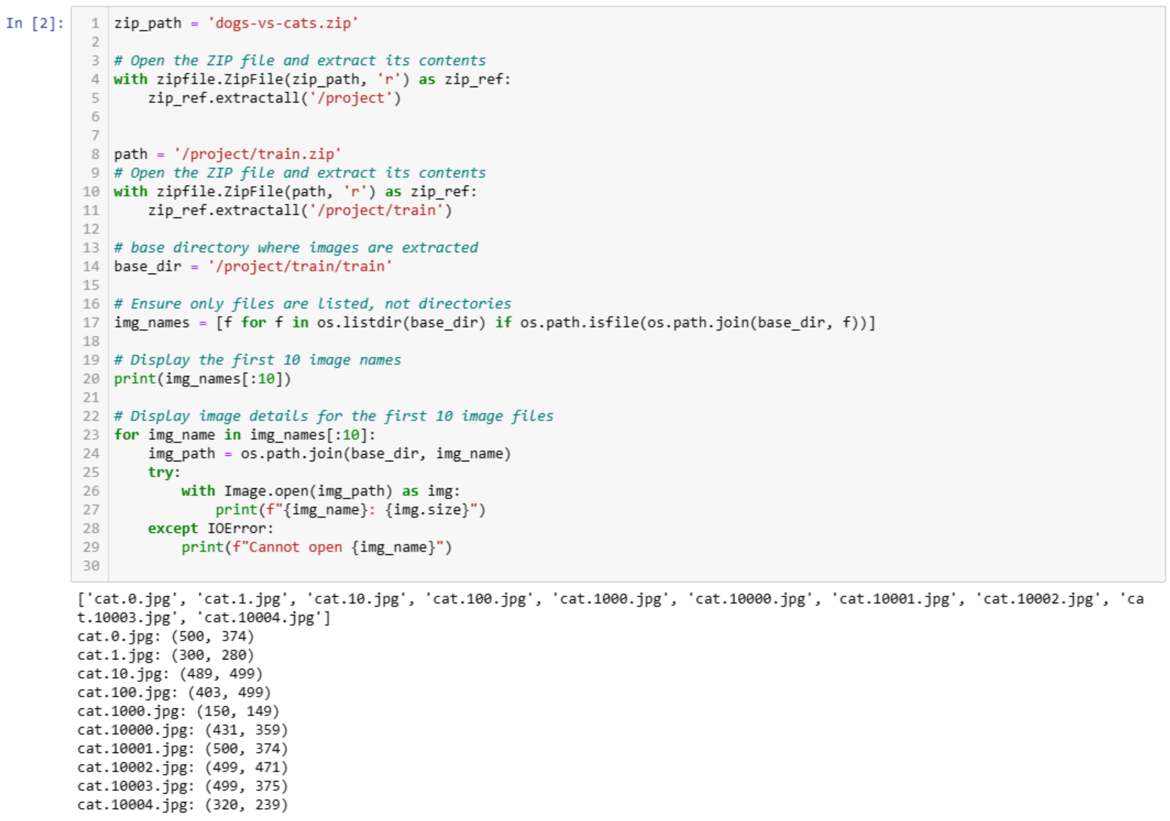
Asirra is unique because of its partnership with Petfinder.com, the world's largest site devoted to finding homes for homeless pets. They've provided Microsoft Research with over three million images of cats and dogs, manually classified by people at thousands of animal shelters across the United States. Kaggle is fortunate to offer a subset of this data for fun and research

Data size:

The training archive contains 25,000 images of dogs and cats.

Data Exploration:

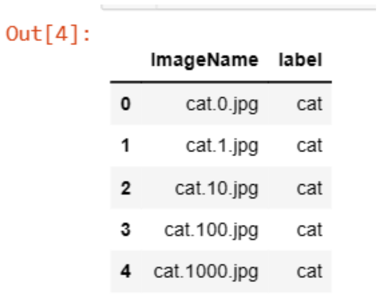
The code begins by specifying the paths to ZIP files (dogs-vs-cats.zip and train.zip). It then uses the zip file module to open these ZIP files and extract their contents to specific directories. After extraction, it defines a base directory where the image files are located. It then lists only the image filenames present in this directory and we saved that in a data frame For each successfully opened 10 image, it prints the image filename along with its dimensions (width and height).



In [2]:

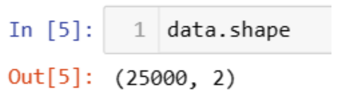
```python
zip_path = 'dogs-vs-cats.zip'

# Open the ZIP file and extract its contents
with zipfile.ZipFile(zip_path, 'r') as zip_ref:
    zip_ref.extractall('/project')


path = '/project/train.zip'
# Open the ZIP file and extract its contents
with zipfile.ZipFile(path, 'r') as zip_ref:
    zip_ref.extractall('/project/train')

# base directory where images are extracted
base_dir = '/project/train/train'

# Ensure only files are listed, not directories
img_names = [f for f in os.listdir(base_dir) if os.path.isfile(os.path.join(base_dir, f))]

# Display the first 10 image names
print(img_names[:10])

# Display image details for the first 10 image files
for img_name in img_names[:10]:
    img_path = os.path.join(base_dir, img_name)
    try:
        with Image.open(img_path) as img:
            print(f"{img_name}: {img.size}")
    except IOError:
        print(f"Cannot open {img_name}")

```

```
['cat.0.jpg', 'cat.1.jpg', 'cat.10.jpg', 'cat.100.jpg', 'cat.1000.jpg', 'cat.10000.jpg', 'cat.10001.jpg', 'cat.10002.jpg', 'ca
t.10003.jpg', 'cat.10004.jpg']
cat.0.jpg: (500, 374)
cat.1.jpg: (300, 280)
cat.10.jpg: (489, 499)
cat.100.jpg: (403, 499)
cat.1000.jpg: (150, 149)
cat.10000.jpg: (431, 359)
cat.10001.jpg: (500, 374)
cat.10002.jpg: (499, 471)
cat.10003.jpg: (499, 375)
cat.10004.jpg: (320, 239)
```



Out[4]:

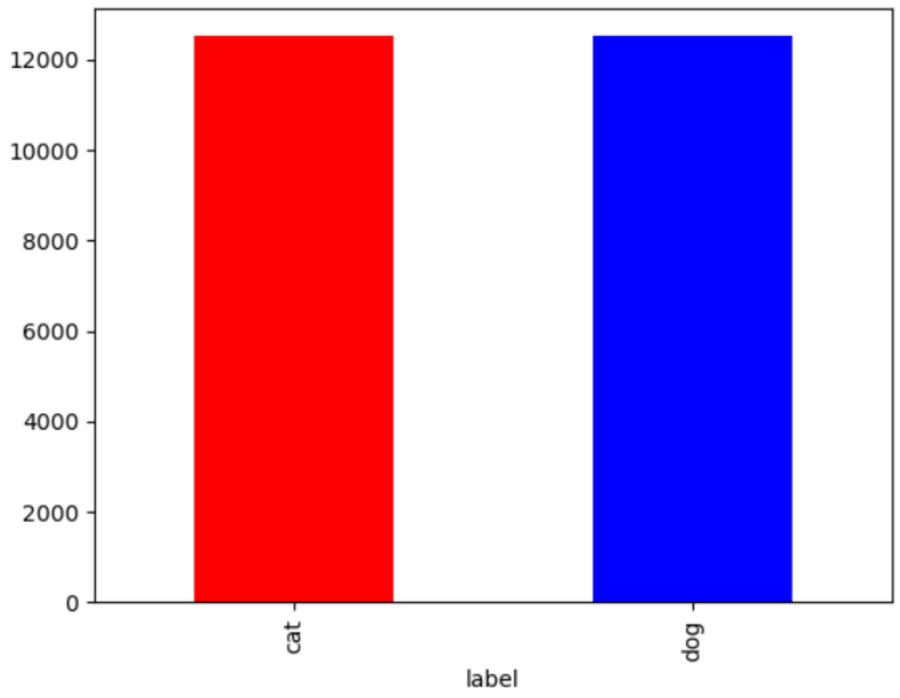
	ImageName	label
0	cat.0.jpg	cat
1	cat.1.jpg	cat
2	cat.10.jpg	cat
3	cat.100.jpg	cat
4	cat.1000.jpg	cat



```
In [5]:  data.shape
Out[5]: (25000, 2)
```

The above code snippet tells us the shape of the data dataframe which is 25000. That is we have successfully extracted all the training images. Based on the Image Name we have given labels to each row like dog and cat.

Number of Dog and Cat in the training data set:



Few Dog and Cat Images:

Splitting Training Data:

We have splitted the training data into 80% for training and 20% for validation/testing.

Model:

We have chose CNN model over fully deep neural networks as CNN is better in terms of their structure, functionality, and applications:

1. **Structure**:

- CNNs: CNNs consist of convolutional layers, pooling layers, and fully connected layers. Convolutional layers apply convolution operations to input images using learnable filters to extract features. Pooling layers down sample feature maps, reducing computational complexity and providing translation invariance. Fully connected layers are typically used at the end of the network for classification.
- DNNs: Fully connected DNNs consist of layers where each neuron is connected to every neuron in the subsequent layer. These networks do not have specific layers for spatial hierarchy understanding or parameter sharing like CNNs.

2. **Functionality**:

- CNNs: CNNs are specialized for processing grid-like data, such as images, due to their ability to capture spatial hierarchies and local patterns efficiently. They excel at tasks like image classification, object detection, and image segmentation.
- DNNs: Fully connected DNNs are more general-purpose and can be applied to various tasks beyond image processing, including text classification, speech recognition, and reinforcement learning.

3. Parameter Sharing and Spatial Hierarchical Understanding:

- CNNs: CNNs leverage parameter sharing and hierarchical structures to learn local patterns and their combinations at different scales. This allows them to effectively capture spatial relationships within images while reducing the number of parameters.
- DNNs: Fully connected DNNs lack the parameter sharing mechanism of CNNs and are not inherently designed to capture spatial hierarchies in data.

4. Efficiency:

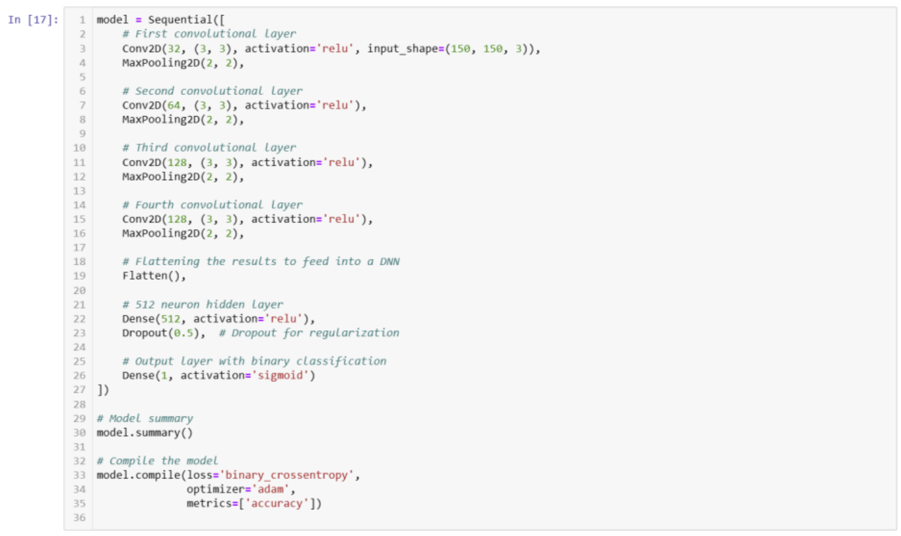
- CNNs: Due to parameter sharing and the use of convolutional and pooling layers, CNNs are more memory-efficient and computationally efficient than fully connected DNNs, especially for large-scale image datasets.
- DNNs: Fully connected DNNs can become computationally expensive and memory-intensive, particularly when dealing with high-dimensional data such as images, as each neuron in one layer is connected to every neuron in the next layer.

5. Performance:

- CNNs: CNNs typically outperform fully connected DNNs on image-related tasks, achieving state-of-the-art performance in tasks like image classification, object detection, and image segmentation.
- DNNs: Fully connected DNNs may perform well on certain tasks, but they are generally not as effective as CNNs for image-related tasks due to their inability to capture spatial hierarchies and patterns efficiently.

As CNNs are specifically tailored for processing grid-like data such as images so we chose CNN over deeply connected neural network.

CNN Model1:



```
In [17]:
model = Sequential([
    # First convolutional layer
    Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)),
    MaxPooling2D(2, 2),

    # Second convolutional layer
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),

    # Third convolutional layer
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),

    # Fourth convolutional layer
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),

    # Flattening the results to feed into a DNN
    Flatten(),

    # 512 neuron hidden layer
    Dense(512, activation='relu'),
    Dropout(0.5),  # Dropout for regularization

    # Output layer with binary classification
    Dense(1, activation='sigmoid')
])

# Model summary
model.summary()

# Compile the model
model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

```

Here is the summary of the CNN model1

Model 1 Architecture:

- Convolutional Layers: Includes four convolutional layers with increasing filter sizes (32, 64, 128, 128) and decreasing spatial dimensions due to max-pooling layers.
- Flattening: After convolutional layers, the output is flattened before passing through dense layers.
- Dense Layers: Contains one dense layer with 512 units and a ReLU activation function.
- Dropout: Applies dropout regularization with a rate of 0.5 after the dense layer.
- Output Layer: Single neuron output layer with a sigmoid activation function for binary classification.

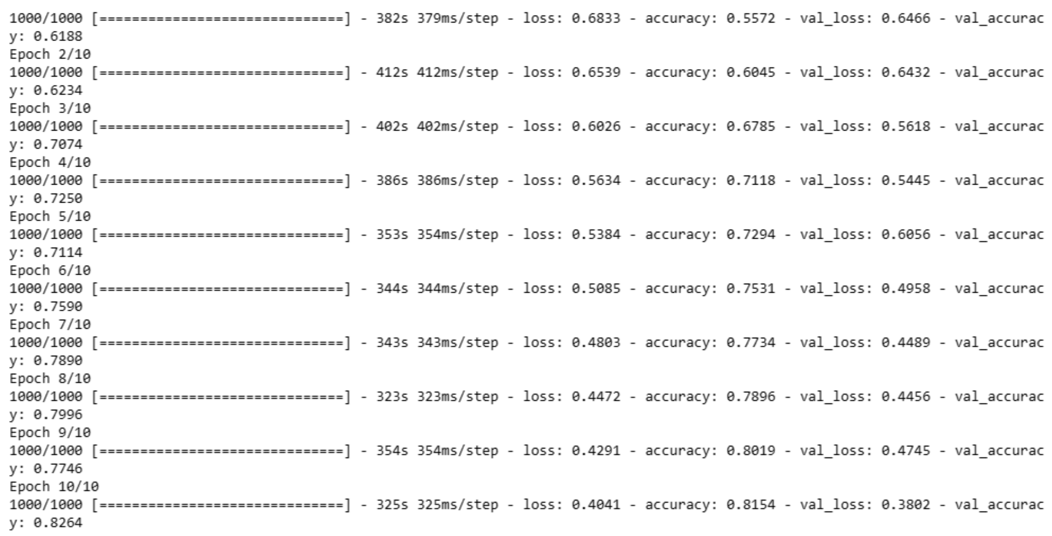
Model Compilation:

- Loss Function: Binary Cross entropy - suitable for binary classification tasks.
- Optimizer Adam optimizer - a popular choice for its adaptive learning rate capabilities.
- Metrics: Accuracy metric is used to monitor model performance during training and validation.

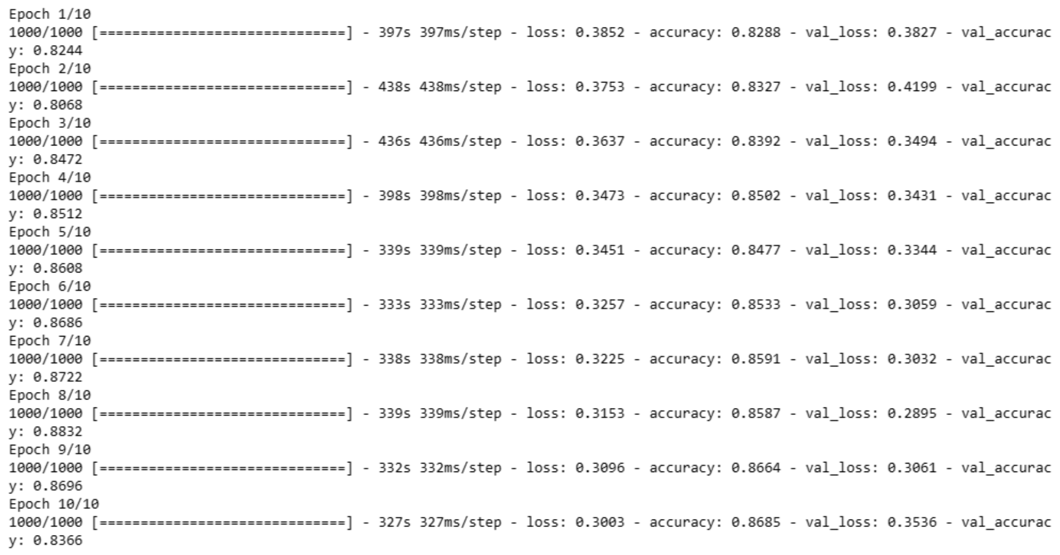
Regularization:

Dropout layer with a rate of 0.5 was added after the first Dense layer to reduce overfitting.

Metrics for each epoch:



```
1000/1000 [==============================] - 382s 379ms/step - loss: 0.6833 - accuracy: 0.5572 - val_loss: 0.6466 - val_accurac
y: 0.6188
Epoch 2/10
1000/1000 [==============================] - 412s 412ms/step - loss: 0.6539 - accuracy: 0.6045 - val_loss: 0.6432 - val_accurac
y: 0.6234
Epoch 3/10
1000/1000 [==============================] - 402s 402ms/step - loss: 0.6026 - accuracy: 0.6785 - val_loss: 0.5618 - val_accurac
y: 0.7074
Epoch 4/10
1000/1000 [==============================] - 386s 386ms/step - loss: 0.5634 - accuracy: 0.7118 - val_loss: 0.5445 - val_accurac
y: 0.7250
Epoch 5/10
1000/1000 [==============================] - 353s 354ms/step - loss: 0.5384 - accuracy: 0.7294 - val_loss: 0.6056 - val_accurac
y: 0.7114
Epoch 6/10
1000/1000 [==============================] - 344s 344ms/step - loss: 0.5085 - accuracy: 0.7531 - val_loss: 0.4958 - val_accurac
y: 0.7590
Epoch 7/10
1000/1000 [==============================] - 343s 343ms/step - loss: 0.4803 - accuracy: 0.7734 - val_loss: 0.4489 - val_accurac
y: 0.7890
Epoch 8/10
1000/1000 [==============================] - 323s 323ms/step - loss: 0.4472 - accuracy: 0.7896 - val_loss: 0.4456 - val_accurac
y: 0.7996
Epoch 9/10
1000/1000 [==============================] - 354s 354ms/step - loss: 0.4291 - accuracy: 0.8019 - val_loss: 0.4745 - val_accurac
y: 0.7746
Epoch 10/10
1000/1000 [==============================] - 325s 325ms/step - loss: 0.4041 - accuracy: 0.8154 - val_loss: 0.3802 - val_accurac
y: 0.8264
```



```
Epoch 1/10
1000/1000 [==============================] - 397s 397ms/step - loss: 0.3852 - accuracy: 0.8288 - val_loss: 0.3827 - val_accurac
y: 0.8244
Epoch 2/10
1000/1000 [==============================] - 438s 438ms/step - loss: 0.3753 - accuracy: 0.8327 - val_loss: 0.4199 - val_accurac
y: 0.8068
Epoch 3/10
1000/1000 [==============================] - 436s 436ms/step - loss: 0.3637 - accuracy: 0.8392 - val_loss: 0.3494 - val_accurac
y: 0.8472
Epoch 4/10
1000/1000 [==============================] - 398s 398ms/step - loss: 0.3473 - accuracy: 0.8502 - val_loss: 0.3431 - val_accurac
y: 0.8512
Epoch 5/10
1000/1000 [==============================] - 339s 339ms/step - loss: 0.3451 - accuracy: 0.8477 - val_loss: 0.3344 - val_accurac
y: 0.8608
Epoch 6/10
1000/1000 [==============================] - 333s 333ms/step - loss: 0.3257 - accuracy: 0.8533 - val_loss: 0.3059 - val_accurac
y: 0.8686
Epoch 7/10
1000/1000 [==============================] - 338s 338ms/step - loss: 0.3225 - accuracy: 0.8591 - val_loss: 0.3032 - val_accurac
y: 0.8722
Epoch 8/10
1000/1000 [==============================] - 339s 339ms/step - loss: 0.3153 - accuracy: 0.8587 - val_loss: 0.2895 - val_accurac
y: 0.8832
Epoch 9/10
1000/1000 [==============================] - 332s 332ms/step - loss: 0.3096 - accuracy: 0.8664 - val_loss: 0.3061 - val_accurac
y: 0.8696
Epoch 10/10
1000/1000 [==============================] - 327s 327ms/step - loss: 0.3003 - accuracy: 0.8685 - val_loss: 0.3536 - val_accurac
y: 0.8366
```

Training :

First we have trained for 10 epochs and observed the increasing in accuracy and decrease in loss. So later we have trained for again 10 epochs for better performance which increased the accuracy to 86%. During training, both training and validation accuracy and loss metrics were recorded. Over epochs, training accuracy improved from around 55% to 86%, and validation accuracy improved from around 62% to 88%. Training time per epoch ranges from approximately 320 to 440 seconds.

Overfitting Analysis:

- There is no clear sign of overfitting observed in the training history.
- Both training and validation accuracies improve consistently without major fluctuations or a widening gap between them

Model Performance:

- Final training accuracy: ~86%
- Final validation accuracy: ~84%

Model 2 CNN:

```python
from tensorflow.keras.optimizers import RMSprop

def create_model():

    model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(32, (3, 3), input_shape=(150, 150, 3)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Activation('LeakyReLU'),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Conv2D(64, (3, 3)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Activation('LeakyReLU'),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Conv2D(128, (3, 3)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Activation('LeakyReLU'),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Conv2D(128, (3, 3)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Activation('LeakyReLU'),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Conv2D(128, (3, 3)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Activation('LeakyReLU'),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation='LeakyReLU'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(1, activation='sigmoid')
  ])

    model.compile(optimizer = RMSprop(learning_rate=0.0001),
                loss = 'binary_crossentropy',
                metrics = ['accuracy'])


    return model

model_2 = create_model()
model_2.summary()

```

Model 2 Architecture:

- **Convolutional Layers**: Five convolutional layers with increasing filter sizes (32, 64, 128) and LeakyReLU activation, followed by batch normalization and max-pooling (2x2) after each convolutional layer.
- **Flatten Layer**: Converts the output of convolutional layers into a 1D array.
- **Dense Layers**: One dense hidden layer with 512 neurons and LeakyReLU activation, followed by a dropout layer (50% rate).
- **Output Layer**: One neuron with sigmoid activation for binary classification.

Model Compilation:

Optimized using RMSprop optimizer with a learning rate of 0.0001, binary crossentropy loss function, and accuracy metric for evaluation

Metrics:

```
Epoch 1/20
1000/1000 [==============================] - 386s 384ms/step - loss: 0.6841
- accuracy: 0.6269 - val_loss: 0.6467 - val_accuracy: 0.6534
Epoch 2/20
1000/1000 [==============================] - 392s 392ms/step - loss: 0.6031
- accuracy: 0.6892 - val_loss: 0.6108 - val_accuracy: 0.6800
Epoch 3/20
1000/1000 [==============================] - 429s 429ms/step - loss: 0.5584
- accuracy: 0.7162 - val_loss: 0.6068 - val_accuracy: 0.6816
Epoch 4/20
1000/1000 [==============================] - 420s 420ms/step - loss: 0.5279
- accuracy: 0.7440 - val_loss: 1.0506 - val_accuracy: 0.5748
Epoch 5/20
1000/1000 [==============================] - 377s 377ms/step - loss: 0.5049
- accuracy: 0.7573 - val_loss: 0.4658 - val_accuracy: 0.7828
Epoch 6/20
1000/1000 [==============================] - 383s 383ms/step - loss: 0.4834
- accuracy: 0.7661 - val_loss: 0.4622 - val_accuracy: 0.7816
Epoch 7/20
1000/1000 [==============================] - 473s 473ms/step - loss: 0.4668
- accuracy: 0.7762 - val_loss: 0.4716 - val_accuracy: 0.7808
Epoch 8/20
1000/1000 [==============================] - 389s 389ms/step - loss: 0.4497
- accuracy: 0.7899 - val_loss: 0.4721 - val_accuracy: 0.7804
Epoch 9/20
1000/1000 [==============================] - 365s 365ms/step - loss: 0.4345
- accuracy: 0.7988 - val_loss: 0.4216 - val_accuracy: 0.8082
Epoch 10/20
1000/1000 [==============================] - 357s 357ms/step - loss: 0.4139
- accuracy: 0.8094 - val_loss: 0.4022 - val_accuracy: 0.8152
Epoch 11/20
1000/1000 [==============================] - 356s 356ms/step - loss: 0.3991
- accuracy: 0.8173 - val_loss: 0.3846 - val_accuracy: 0.8286
Epoch 12/20
1000/1000 [==============================] - 384s 384ms/step - loss: 0.3882
- accuracy: 0.8219 - val_loss: 0.5107 - val_accuracy: 0.7470
Epoch 13/20
1000/1000 [==============================] - 369s 369ms/step - loss: 0.3829
- accuracy: 0.8271 - val_loss: 0.3797 - val_accuracy: 0.8262
Epoch 14/20
1000/1000 [==============================] - 365s 365ms/step - loss: 0.3734
- accuracy: 0.8320 - val_loss: 0.3576 - val_accuracy: 0.8484
Epoch 15/20
1000/1000 [==============================] - 367s 367ms/step - loss: 0.3538
- accuracy: 0.8413 - val_loss: 0.3943 - val_accuracy: 0.8260
Epoch 16/20
```

```
1000/1000 [==============================] - 359s 359ms/step - loss: 0.3520
- accuracy: 0.8420 - val_loss: 0.4023 - val_accuracy: 0.8162
Epoch 17/20
1000/1000 [==============================] - 364s 364ms/step - loss: 0.3443
- accuracy: 0.8464 - val_loss: 0.3941 - val_accuracy: 0.8262
Epoch 18/20
1000/1000 [==============================] - 358s 358ms/step - loss: 0.3360
- accuracy: 0.8486 - val_loss: 0.3004 - val_accuracy: 0.8666
Epoch 19/20
1000/1000 [==============================] - 360s 360ms/step - loss: 0.3249
- accuracy: 0.8559 - val_loss: 0.3694 - val_accuracy: 0.8372
Epoch 20/20
1000/1000 [==============================] - 358s 358ms/step - loss: 0.3207
- accuracy: 0.8574 - val_loss: 0.2979 - val_accuracy: 0.8668
```

Training History:

- Training accuracy improves from about 62% to 85% over epochs.
- Validation accuracy improves from about 65% to 87% over epochs.
- Both training and validation accuracies follow a similar increasing trend without significant divergence, indicating good generalization.

Overfitting Analysis:

- There is no clear sign of overfitting observed in the training history.
- Both training and validation accuracies improve consistently without major fluctuations or a widening gap between them.
- The use of dropout layers (rate=0.5) after dense layers contributes to regularization, helping prevent overfitting.

Model Performance:

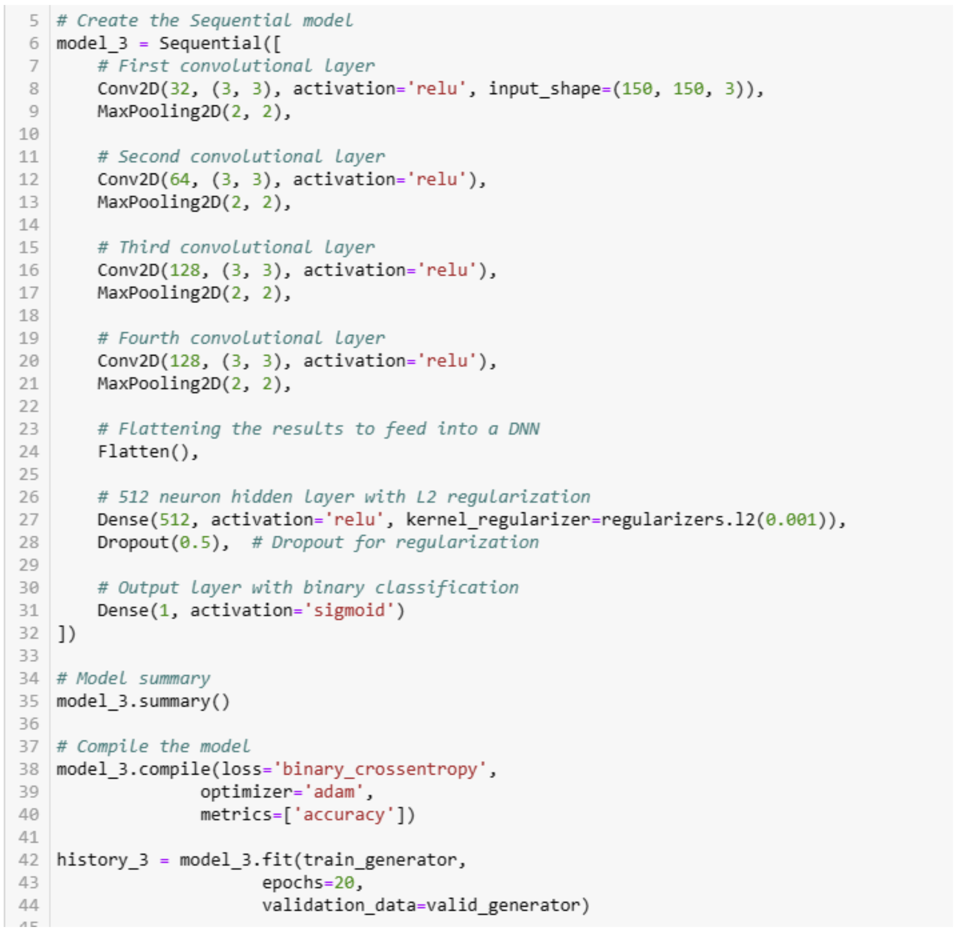
- Final training accuracy: ~86%
- Final validation accuracy: ~87%

The model achieves decent accuracy on both training and validation sets without signs of overfitting, indicating that it generalizes well to unseen data.

Summary:

The model architecture, along with the regularization techniques like dropout, seems effective in preventing overfitting. The model demonstrates good convergence and generalization capabilities without exhibiting significant overfitting issues across the provided training epochs.

Model 3 CNN:



```python
# Create the Sequential model
model_3 = Sequential([
    # First convolutional layer
    Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)),
    MaxPooling2D(2, 2),

    # Second convolutional layer
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),

    # Third convolutional layer
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),

    # Fourth convolutional layer
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),

    # Flattening the results to feed into a DNN
    Flatten(),

    # 512 neuron hidden layer with L2 regularization
    Dense(512, activation='relu', kernel_regularizer=regularizers.l2(0.001)),
    Dropout(0.5),  # Dropout for regularization

    # Output layer with binary classification
    Dense(1, activation='sigmoid')
])

# Model summary
model_3.summary()

# Compile the model
model_3.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

history_3 = model_3.fit(train_generator,
                    epochs=20,
                    validation_data=valid_generator)
```

Model Architecture:

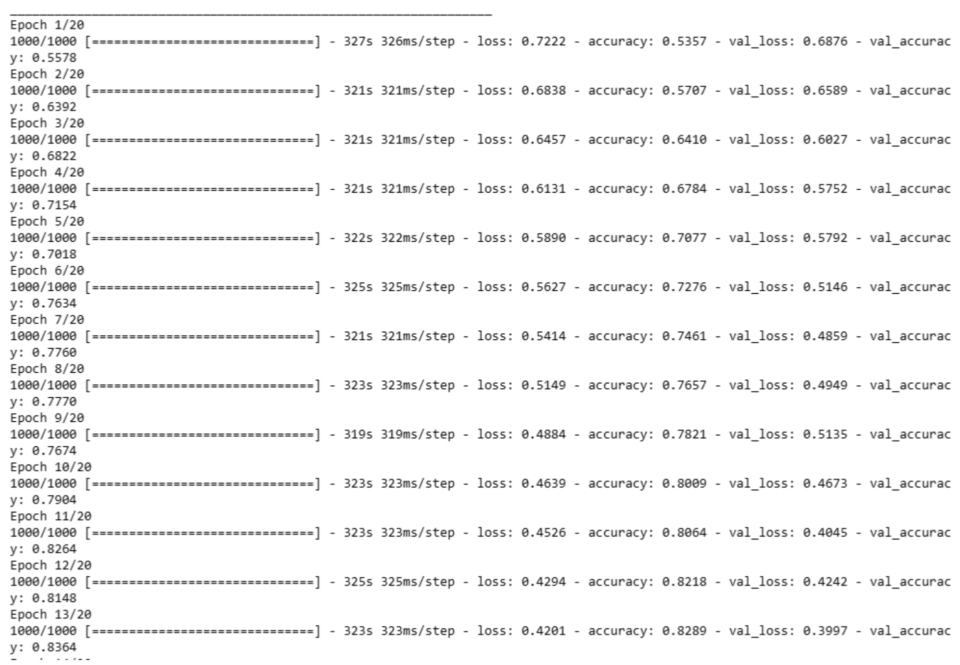
- **Convolutional Layers**: Four convolutional layers with increasing filters (32, 64, 128) and ReLU activation, followed by max-pooling layers.
- **Flatten Layer**: Converts the output of convolutional layers into a 1D array.
- **Dense Layers**: One dense hidden layer with 512 neurons, ReLU activation, and L2 regularization (kernel_regularizer=regularizers.l2(0.001)). Includes dropout (50%) for regularization.
- **Output Layer**: One neuron with sigmoid activation for binary classification.
- **Model Compilation**: Optimized using the Adam optimizer, binary crossentropy loss function, and accuracy metric for evaluation.

The model follows a typical CNN architecture for image classification, with added regularization techniques like L2 regularization and dropout to improve generalization and prevent overfitting.

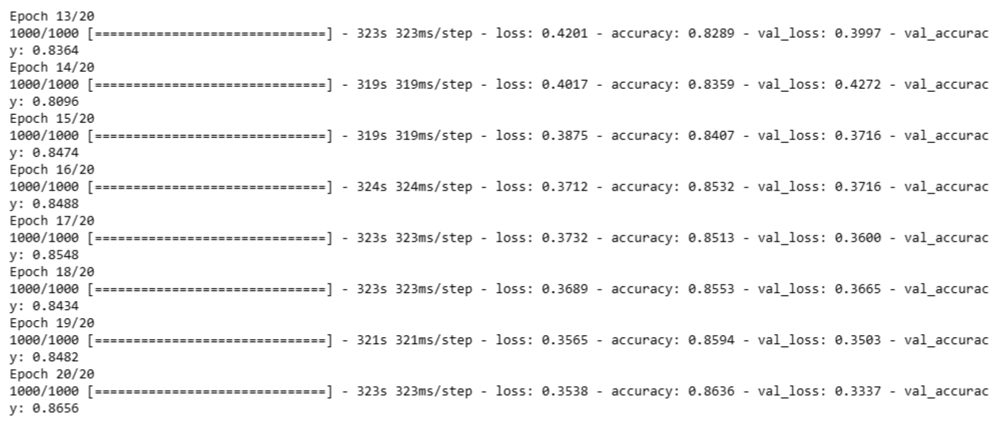
Model Compilation:

- Loss Function: Binary Cross entropy - suitable for binary classification tasks.
- Optimizer Adam optimizer - a popular choice for its adaptive learning rate capabilities.
- Metrics: Accuracy metric is used to monitor model performance during training and validation.

Metrics:



```
Epoch 1/20
1000/1000 [==============================] - 327s 326ms/step - loss: 0.7222 - accuracy: 0.5357 - val_loss: 0.6876 - val_accurac
y: 0.5578
Epoch 2/20
1000/1000 [==============================] - 321s 321ms/step - loss: 0.6838 - accuracy: 0.5707 - val_loss: 0.6589 - val_accurac
y: 0.6392
Epoch 3/20
1000/1000 [==============================] - 321s 321ms/step - loss: 0.6457 - accuracy: 0.6410 - val_loss: 0.6027 - val_accurac
y: 0.6822
Epoch 4/20
1000/1000 [==============================] - 321s 321ms/step - loss: 0.6131 - accuracy: 0.6784 - val_loss: 0.5752 - val_accurac
y: 0.7154
Epoch 5/20
1000/1000 [==============================] - 322s 322ms/step - loss: 0.5890 - accuracy: 0.7077 - val_loss: 0.5792 - val_accurac
y: 0.7018
Epoch 6/20
1000/1000 [==============================] - 325s 325ms/step - loss: 0.5627 - accuracy: 0.7276 - val_loss: 0.5146 - val_accurac
y: 0.7634
Epoch 7/20
1000/1000 [==============================] - 321s 321ms/step - loss: 0.5414 - accuracy: 0.7461 - val_loss: 0.4859 - val_accurac
y: 0.7760
Epoch 8/20
1000/1000 [==============================] - 323s 323ms/step - loss: 0.5149 - accuracy: 0.7657 - val_loss: 0.4949 - val_accurac
y: 0.7770
Epoch 9/20
1000/1000 [==============================] - 319s 319ms/step - loss: 0.4884 - accuracy: 0.7821 - val_loss: 0.5135 - val_accurac
y: 0.7674
Epoch 10/20
1000/1000 [==============================] - 323s 323ms/step - loss: 0.4639 - accuracy: 0.8009 - val_loss: 0.4673 - val_accurac
y: 0.7904
Epoch 11/20
1000/1000 [==============================] - 323s 323ms/step - loss: 0.4526 - accuracy: 0.8064 - val_loss: 0.4045 - val_accurac
y: 0.8264
Epoch 12/20
1000/1000 [==============================] - 325s 325ms/step - loss: 0.4294 - accuracy: 0.8218 - val_loss: 0.4242 - val_accurac
y: 0.8148
Epoch 13/20
1000/1000 [==============================] - 323s 323ms/step - loss: 0.4201 - accuracy: 0.8289 - val_loss: 0.3997 - val_accurac
y: 0.8364
```



```
Epoch 13/20
1000/1000 [==============================] - 323s 323ms/step - loss: 0.4201 - accuracy: 0.8289 - val_loss: 0.3997 - val_accurac
y: 0.8364
Epoch 14/20
1000/1000 [==============================] - 319s 319ms/step - loss: 0.4017 - accuracy: 0.8359 - val_loss: 0.4272 - val_accurac
y: 0.8096
Epoch 15/20
1000/1000 [==============================] - 319s 319ms/step - loss: 0.3875 - accuracy: 0.8407 - val_loss: 0.3716 - val_accurac
y: 0.8474
Epoch 16/20
1000/1000 [==============================] - 324s 324ms/step - loss: 0.3712 - accuracy: 0.8532 - val_loss: 0.3716 - val_accurac
y: 0.8488
Epoch 17/20
1000/1000 [==============================] - 323s 323ms/step - loss: 0.3732 - accuracy: 0.8513 - val_loss: 0.3600 - val_accurac
y: 0.8548
Epoch 18/20
1000/1000 [==============================] - 323s 323ms/step - loss: 0.3689 - accuracy: 0.8553 - val_loss: 0.3665 - val_accurac
y: 0.8434
Epoch 19/20
1000/1000 [==============================] - 321s 321ms/step - loss: 0.3565 - accuracy: 0.8594 - val_loss: 0.3503 - val_accurac
y: 0.8482
Epoch 20/20
1000/1000 [==============================] - 323s 323ms/step - loss: 0.3538 - accuracy: 0.8636 - val_loss: 0.3337 - val_accurac
y: 0.8656
```

Loss:

- Training Loss: Decreases from 0.7222 to 0.3538 over 20 epochs.
- Validation Loss: Decreases from 0.6876 to 0.3337 over 20 epochs.

Accuracy:

- Training Accuracy: Increases from 0.5357 to 0.8636 over 20 epochs.
- Validation Accuracy: Increases from 0.5578 to 0.8656 over 20 epochs.

The decreasing trend in both training and validation loss indicates that the model is learning the patterns in the training data and generalizing well to unseen validation data. The increasing trend in training and validation accuracy suggests that the model's predictions are improving over epochs. The final validation accuracy of approximately 86.56% indicates that the model performs well on unseen validation data, showcasing its ability to generalize.

The use of dropout and regularization in "model_3" can have a significant impact on the model's performance and generalization ability

Dropout:

- Dropout randomly drops a fraction of neurons during training to prevent overfitting.
- Dropout helps in reducing overfitting by preventing the model from relying too heavily on specific neurons or features in the training data. This leads to better generalization and improved accuracy on unseen data.
- Dropout can lead to slower convergence during training but often results in better validation performance. The model becomes less sensitive to noise in the training data, leading to better performance on validation data.

Regularization (L2 Regularization):

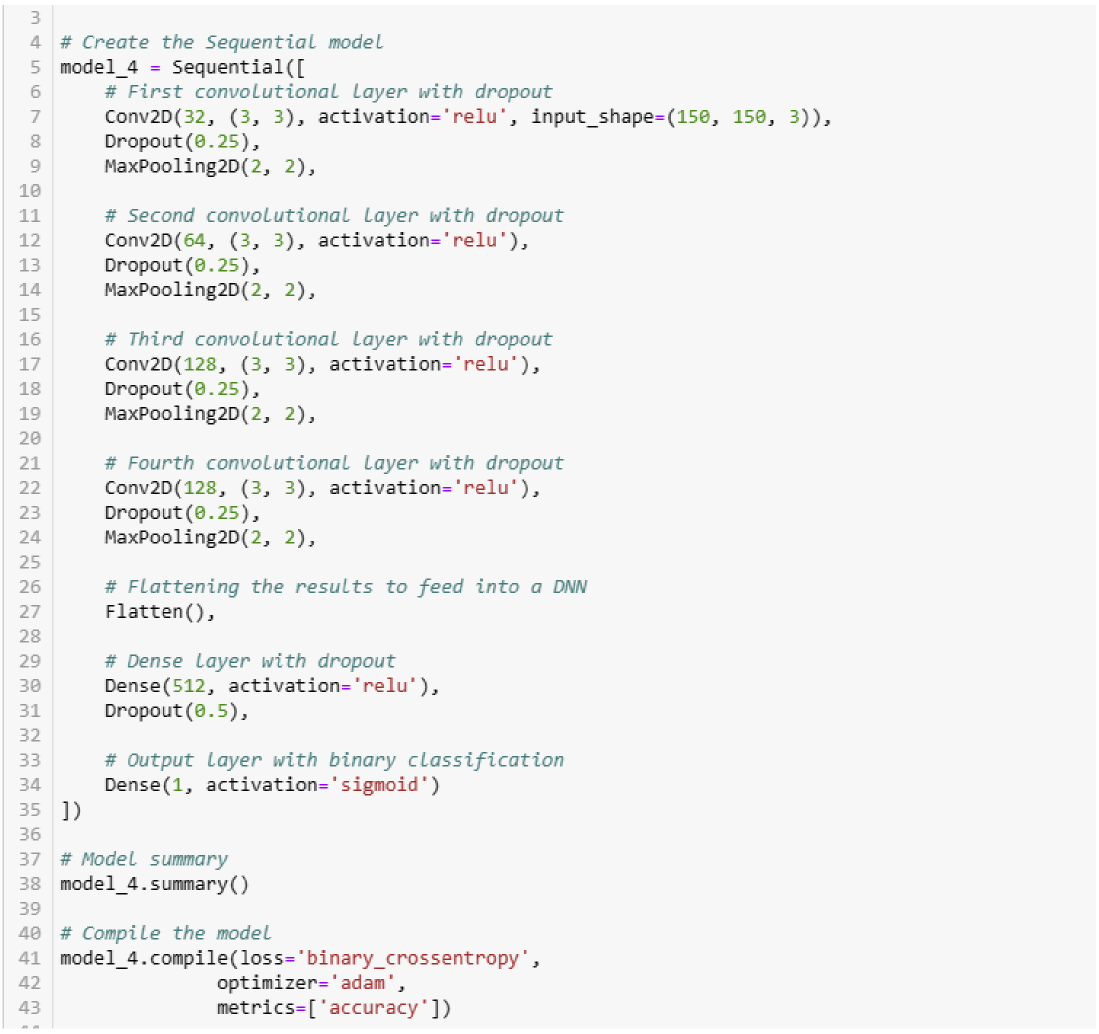
- L2 regularization penalizes large weights in the neural network to prevent overfitting.
- L2 regularization encourages the model to learn simpler patterns by penalizing large weights. This helps in reducing overfitting and improving generalization, leading to lower training loss and higher accuracy.
- Similar to dropout, L2 regularization contributes to better validation performance by reducing overfitting tendencies. The model becomes more robust and less prone to memorizing noise in the training data, resulting in better generalization to unseen validation data.

Model Performance:

- Final training accuracy: 86.36%
- Final validation accuracy: 86.56%

The model achieves decent accuracy on both training and validation sets without signs of overfitting, due to use of drop out and regularization, indicating that it generalizes well to unseen data.

Model 4 CNN:



```python

# Create the Sequential model
model_4 = Sequential([
    # First convolutional layer with dropout
    Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)),
    Dropout(0.25),
    MaxPooling2D(2, 2),

    # Second convolutional layer with dropout
    Conv2D(64, (3, 3), activation='relu'),
    Dropout(0.25),
    MaxPooling2D(2, 2),

    # Third convolutional layer with dropout
    Conv2D(128, (3, 3), activation='relu'),
    Dropout(0.25),
    MaxPooling2D(2, 2),

    # Fourth convolutional layer with dropout
    Conv2D(128, (3, 3), activation='relu'),
    Dropout(0.25),
    MaxPooling2D(2, 2),

    # Flattening the results to feed into a DNN
    Flatten(),

    # Dense layer with dropout
    Dense(512, activation='relu'),
    Dropout(0.5),

    # Output layer with binary classification
    Dense(1, activation='sigmoid')
])

# Model summary
model_4.summary()

# Compile the model
model_4.compile(loss='binary_crossentropy',
                optimizer='adam',
                metrics=['accuracy'])
```

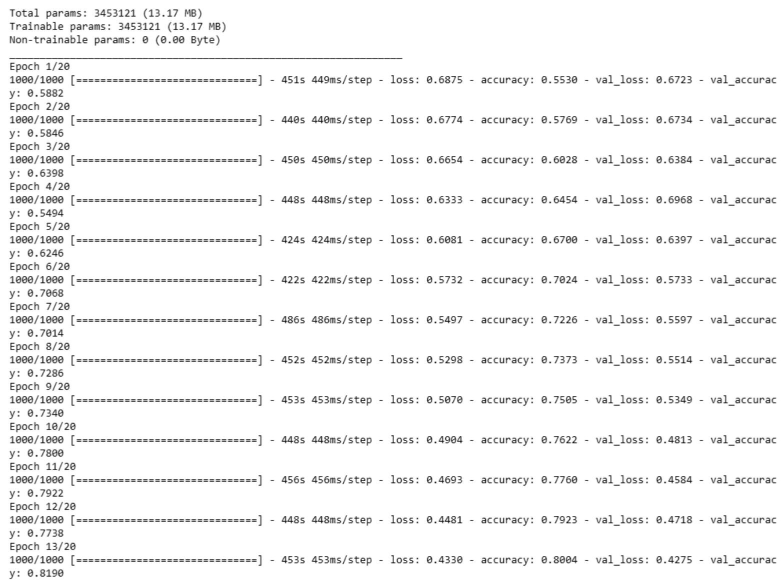
Model Architecture:

- Convolutional Layers with Dropout: Four convolutional layers with dropout rates of 25% after each layer to prevent overfitting.
- Flatten Layer: Converts the output of convolutional layers into a 1D array for input to the Dense layers.
- Dense Layers with Dropout: One dense hidden layer with 512 neurons and a dropout rate of 50% to further prevent overfitting.
- Output Layer: One neuron with sigmoid activation for binary classification.

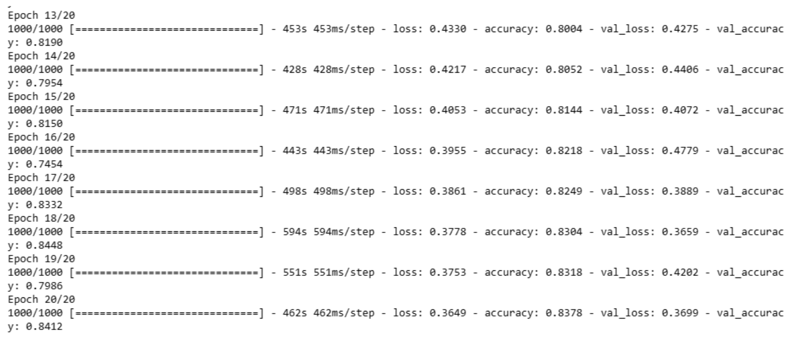
Model Compilation: Optimized using the Adam optimizer, binary cross entropy loss function, and accuracy metric for evaluation.

The model follows a typical CNN architecture for image classification, with added regularization techniques like dropout to each layer to improve generalization and prevent overfitting.

Metrics:



```
Total params: 3453121 (13.17 MB)
Trainable params: 3453121 (13.17 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Epoch 1/20
1000/1000 [==============================] - 451s 449ms/step - loss: 0.6875 - accuracy: 0.5530 - val_loss: 0.6723 - val_accurac
y: 0.5882
Epoch 2/20
1000/1000 [==============================] - 440s 440ms/step - loss: 0.6774 - accuracy: 0.5769 - val_loss: 0.6734 - val_accurac
y: 0.5846
Epoch 3/20
1000/1000 [==============================] - 450s 450ms/step - loss: 0.6654 - accuracy: 0.6028 - val_loss: 0.6384 - val_accurac
y: 0.6398
Epoch 4/20
1000/1000 [==============================] - 448s 448ms/step - loss: 0.6333 - accuracy: 0.6454 - val_loss: 0.6968 - val_accurac
y: 0.5494
Epoch 5/20
1000/1000 [==============================] - 424s 424ms/step - loss: 0.6081 - accuracy: 0.6700 - val_loss: 0.6397 - val_accurac
y: 0.6246
Epoch 6/20
1000/1000 [==============================] - 422s 422ms/step - loss: 0.5732 - accuracy: 0.7024 - val_loss: 0.5733 - val_accurac
y: 0.7068
Epoch 7/20
1000/1000 [==============================] - 486s 486ms/step - loss: 0.5497 - accuracy: 0.7226 - val_loss: 0.5597 - val_accurac
y: 0.7014
Epoch 8/20
1000/1000 [==============================] - 452s 452ms/step - loss: 0.5298 - accuracy: 0.7373 - val_loss: 0.5514 - val_accurac
y: 0.7286
Epoch 9/20
1000/1000 [==============================] - 453s 453ms/step - loss: 0.5070 - accuracy: 0.7505 - val_loss: 0.5349 - val_accurac
y: 0.7340
Epoch 10/20
1000/1000 [==============================] - 448s 448ms/step - loss: 0.4904 - accuracy: 0.7622 - val_loss: 0.4813 - val_accurac
y: 0.7800
Epoch 11/20
1000/1000 [==============================] - 456s 456ms/step - loss: 0.4693 - accuracy: 0.7760 - val_loss: 0.4584 - val_accurac
y: 0.7922
Epoch 12/20
1000/1000 [==============================] - 448s 448ms/step - loss: 0.4481 - accuracy: 0.7923 - val_loss: 0.4718 - val_accurac
y: 0.7738
Epoch 13/20
1000/1000 [==============================] - 453s 453ms/step - loss: 0.4330 - accuracy: 0.8004 - val_loss: 0.4275 - val_accurac
y: 0.8190
```



```
Epoch 13/20
1000/1000 [==============================] - 453s 453ms/step - loss: 0.4330 - accuracy: 0.8004 - val_loss: 0.4275 - val_accurac
y: 0.8190
Epoch 14/20
1000/1000 [==============================] - 428s 428ms/step - loss: 0.4217 - accuracy: 0.8052 - val_loss: 0.4406 - val_accurac
y: 0.7954
Epoch 15/20
1000/1000 [==============================] - 471s 471ms/step - loss: 0.4053 - accuracy: 0.8144 - val_loss: 0.4072 - val_accurac
y: 0.8150
Epoch 16/20
1000/1000 [==============================] - 443s 443ms/step - loss: 0.3955 - accuracy: 0.8218 - val_loss: 0.4779 - val_accurac
y: 0.7454
Epoch 17/20
1000/1000 [==============================] - 498s 498ms/step - loss: 0.3861 - accuracy: 0.8249 - val_loss: 0.3889 - val_accurac
y: 0.8332
Epoch 18/20
1000/1000 [==============================] - 594s 594ms/step - loss: 0.3778 - accuracy: 0.8304 - val_loss: 0.3659 - val_accurac
y: 0.8448
Epoch 19/20
1000/1000 [==============================] - 551s 551ms/step - loss: 0.3753 - accuracy: 0.8318 - val_loss: 0.4202 - val_accurac
y: 0.7986
Epoch 20/20
1000/1000 [==============================] - 462s 462ms/step - loss: 0.3649 - accuracy: 0.8378 - val_loss: 0.3699 - val_accurac
y: 0.8412
```

Loss:

- Training Loss: Decreases from around 0.6875 to 0.3649 over 20 epochs.
- Validation Loss: Decreases from around 0.6723 to 0.3699 over 20 epochs.

Accuracy:

- Training Accuracy: Increases from approximately 0.5530 to 0.8378 over 20 epochs.
- Validation Accuracy: Increases from around 0.5882 to 0.8412 over 20 epochs.

Training Metrics:

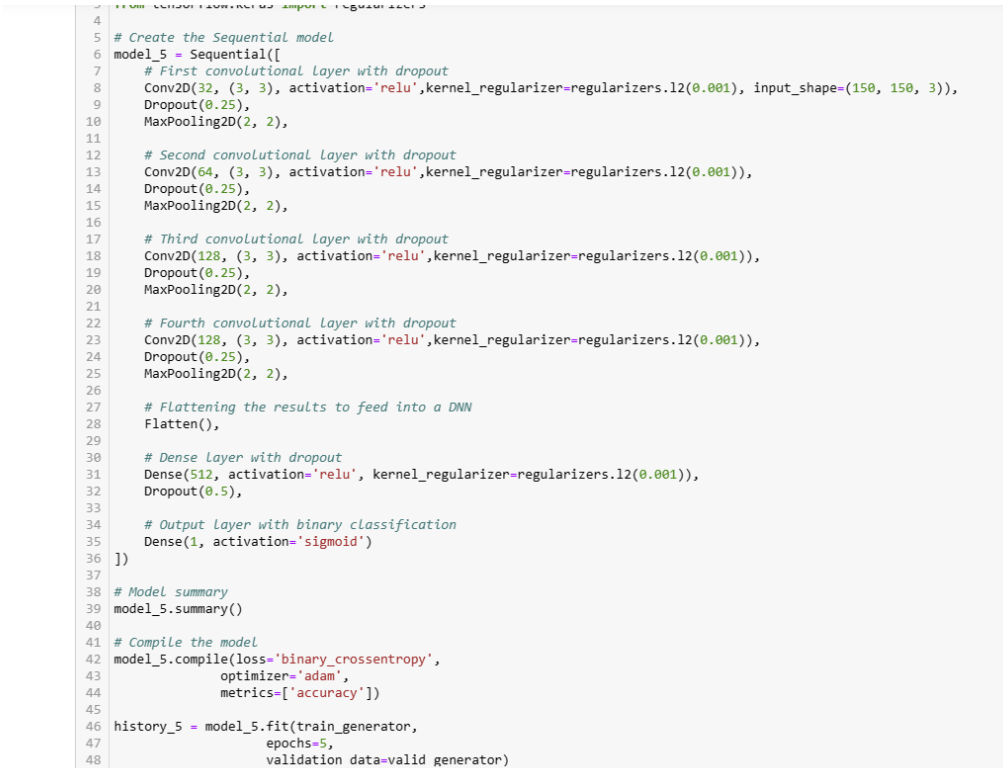
- The training loss decreases consistently, indicating that the model is learning and improving its predictions on the training data.
- The training accuracy steadily increases, showing that the model is becoming more accurate in classifying the training samples.

Validation Metrics:

- The validation loss and accuracy follow a similar trend to the training metrics, which is a positive sign. It suggests that the model is generalizing well to unseen data, as indicated by the decreasing loss and increasing accuracy on the validation set.

The model demonstrates good performance as both training and validation metrics improve over epochs, indicating effective learning and generalization.

Model 5 CNN:



```python
from tensorflow.keras import regularizers

# Create the Sequential model
model_5 = Sequential([
    # First convolutional layer with dropout
    Conv2D(32, (3, 3), activation='relu',kernel_regularizer=regularizers.l2(0.001), input_shape=(150, 150, 3)),
    Dropout(0.25),
    MaxPooling2D(2, 2),

    # Second convolutional layer with dropout
    Conv2D(64, (3, 3), activation='relu',kernel_regularizer=regularizers.l2(0.001)),
    Dropout(0.25),
    MaxPooling2D(2, 2),

    # Third convolutional layer with dropout
    Conv2D(128, (3, 3), activation='relu',kernel_regularizer=regularizers.l2(0.001)),
    Dropout(0.25),
    MaxPooling2D(2, 2),

    # Fourth convolutional layer with dropout
    Conv2D(128, (3, 3), activation='relu',kernel_regularizer=regularizers.l2(0.001)),
    Dropout(0.25),
    MaxPooling2D(2, 2),

    # Flattening the results to feed into a DNN
    Flatten(),

    # Dense layer with dropout
    Dense(512, activation='relu', kernel_regularizer=regularizers.l2(0.001)),
    Dropout(0.5),

    # Output layer with binary classification
    Dense(1, activation='sigmoid')
])

# Model summary
model_5.summary()

# Compile the model
model_5.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

history_5 = model_5.fit(train_generator,
                    epochs=5,
                    validation_data=valid_generator)
```

Model Architecture:

Convolutional Layers with Dropout and Regularization:

- Four convolutional layers with increasing filters (32, 64, 128, 128).
- Kernel size of (3, 3) for all convolutional layers.
- ReLU activation for all convolutional layers.
- L2 regularization (0.001) applied to all convolutional layers.
- Dropout of 25% after each convolutional layer.
- MaxPooling2D layers with pool size (2, 2) after each convolutional layer.

Flatten Layer: Flattens the convolutional output for Dense layers.

Dense Layers with Dropout and Regularization:

- One dense hidden layer with 512 neurons and ReLU activation.
- L2 regularization (0.001) applied to the dense layer.
- Dropout of 50% after the dense layer.

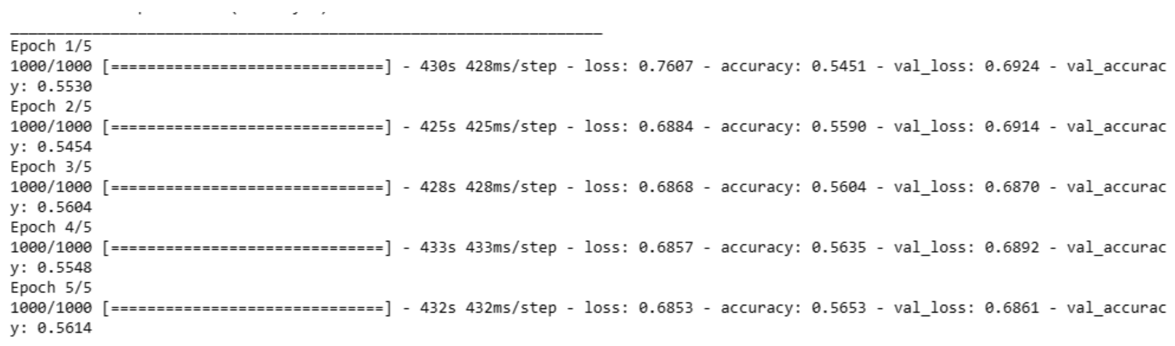
Output Layer: One neuron with sigmoid activation for binary classification

Compilation: Adam optimizer, binary cross entropy loss, and accuracy metric used for training

Effect:

- Regularization (L2) helps control overfitting by penalizing large weights.
- Dropout layers prevent overfitting by randomly dropping neurons during training.
- The model architecture is designed for robustness and generalization in image classification tasks.

Metrics:



```
Epoch 1/5
1000/1000 [==============================] - 430s 428ms/step - loss: 0.7607 - accuracy: 0.5451 - val_loss: 0.6924 - val_accurac
y: 0.5530
Epoch 2/5
1000/1000 [==============================] - 425s 425ms/step - loss: 0.6884 - accuracy: 0.5590 - val_loss: 0.6914 - val_accurac
y: 0.5454
Epoch 3/5
1000/1000 [==============================] - 428s 428ms/step - loss: 0.6868 - accuracy: 0.5604 - val_loss: 0.6870 - val_accurac
y: 0.5604
Epoch 4/5
1000/1000 [==============================] - 433s 433ms/step - loss: 0.6857 - accuracy: 0.5635 - val_loss: 0.6892 - val_accurac
y: 0.5548
Epoch 5/5
1000/1000 [==============================] - 432s 432ms/step - loss: 0.6853 - accuracy: 0.5653 - val_loss: 0.6861 - val_accurac
y: 0.5614
```

Loss:

- Training Loss: Started at around 0.7607 and decreased to approximately 0.6853 by the end of 5 epochs.
- Validation Loss: Started around 0.6924 and decreased to about 0.6861 after 5 epochs.

Accuracy:

- Training Accuracy: Started at about 0.5451 and increased to roughly 0.5653 after 5 epochs.
- Validation Accuracy: Started around 0.5530 and increased to about 0.5614 after 5 epochs.

Training Metrics:

- The training loss decreases, indicating that the model is learning to minimize errors on the training data.
- The training accuracy increases slightly, showing the model's ability to classify training samples better.

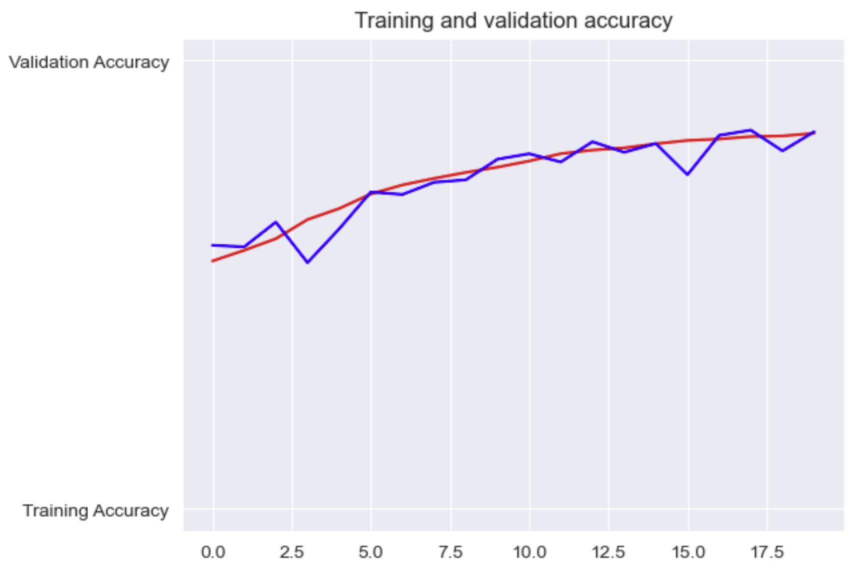
Validation Metrics:

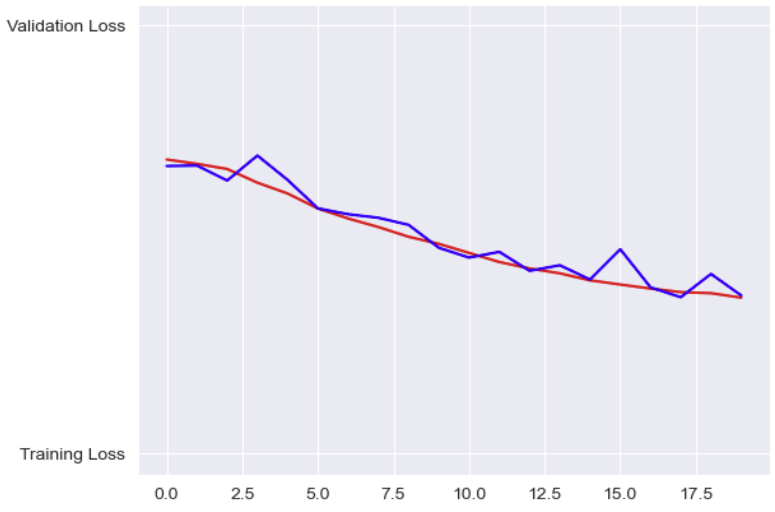
- The validation loss fluctuates slightly but does not show a significant decreasing trend.
- The validation accuracy also shows slight fluctuations without a clear increasing trend.

Observations:

- The model seems to struggle to significantly improve accuracy and reduce loss over the epochs.
- There might be issues with high model complexity or other factors affecting model performance.

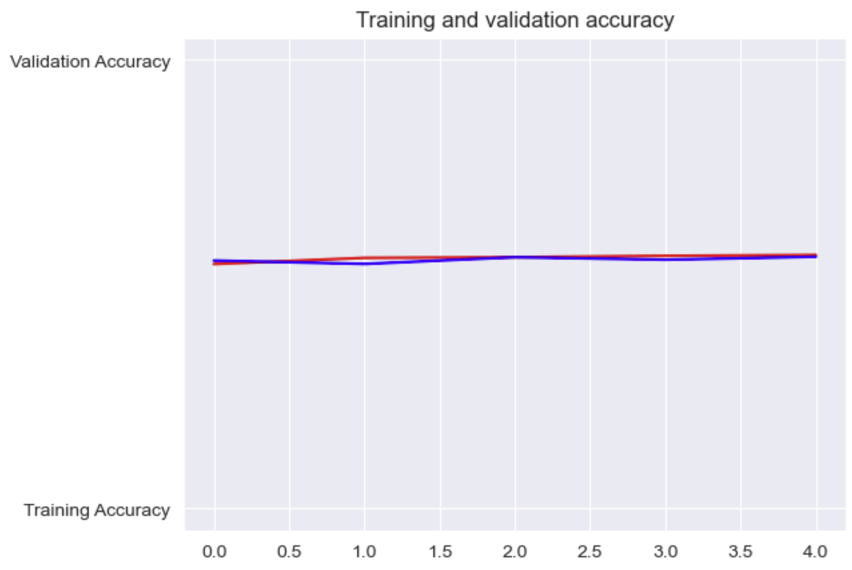
Models Evaluation:

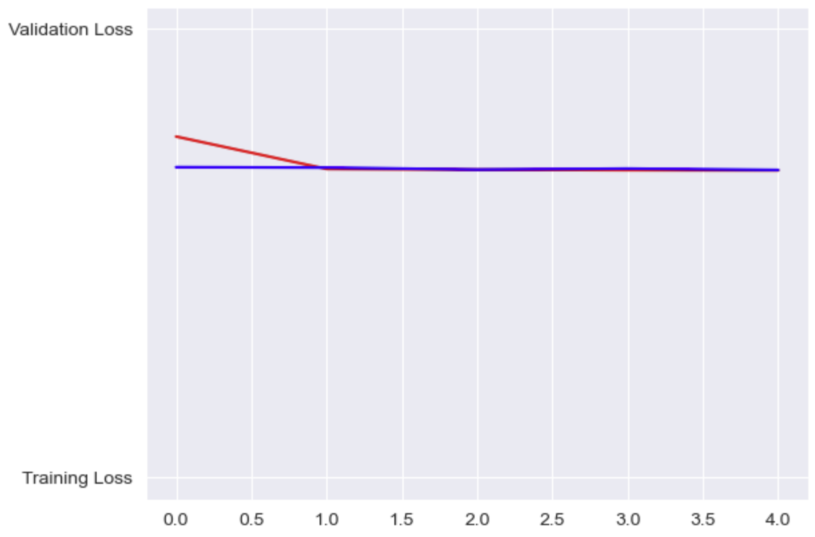




Model 1, 2, 3 and 4 are following same trend that is accuracy is improving by epochs and decrease in Validation loss which means the model is learning for each epoch.

The accuracy for almost 4 models are similar. The model that is best according to architecture, accuracy and overfitting is model 4. When we compare the activation functions LeakyReLU performed better than ReLU and by adding the layer to model 1 leaded to better model which is model 2. By adding drop out and regularization we were able to control over fitting and more validation accuracy which means model is performing better on unknown data.

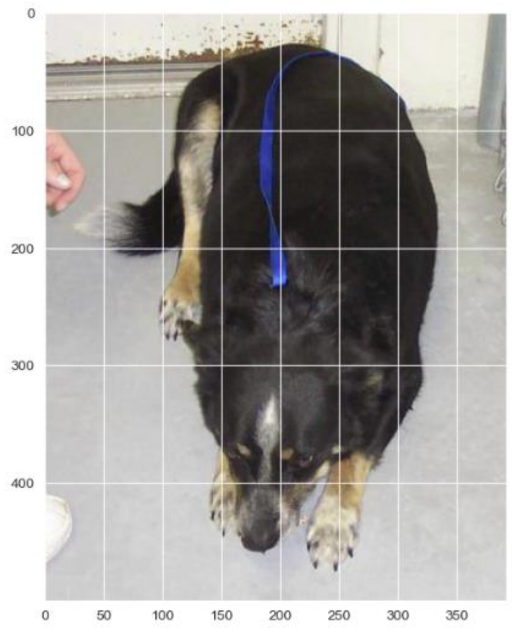




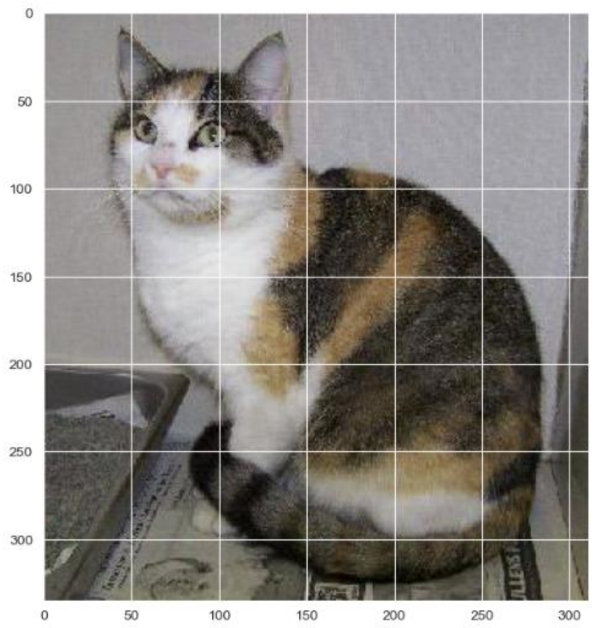
The above 2 graphs are for model 5 which is saying that the model accuracy is not increasing much, that is model is facing difficulty to learn from epochs. The Validation Loss is also not much decreasing. From this we can say that the model5 is complex for the data.

Training data:

Extracted images from test folder and predicted. Plotted the predicted results.

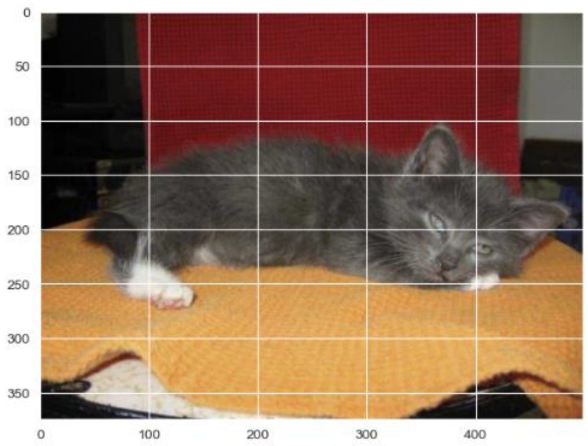


24.jpg Category: (Dog)



624.jpg Category: (Cat)

11773.jpg Category: (Dog)



1618.jpg Category: (Cat)

Extracted the results into submission.csv file.

Conclusion:

- The model that is best according to architecture, accuracy and overfitting is model 4.
- When we compare the activation functions LeakyReLU performed better than ReLU and by adding the layer to model 1 leaded to better model which is model 2.
- By adding drop out and regularization we were able to control over fitting and more validation accuracy which means model is performing better on unknown data.
- Model 5 didn’t perform better due to model complexity.
- Tried running Resnet which is taking more time to run due to lack of high computational machines. So tried different models in CNN.

Future work:

- If we see the metrics for model 1 -4 which says that if we increase the number of epochs to between 50 to 100 the model accuracy might increase and perform better.
- By hypertuning the parameters we can make the model better.
- We can use imagenet, ResNet for better results. Resnet needs more computational power to execute.