**GROUP-10 – ASSIGNMENT-2 CONVOLUTION – SUMMARY REPORT**

1. **Consider the Cats & Dogs example. 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?**

**Initializing the Training Dataset:**

A DataFrame named 'train' is initialized as blank to hold the training dataset. Utilizing a function titled 'load\_dataset' from the 'TRAIN\_DIR' directory, image files are processed, and labels are assigned (for instance, 0 for cats and 1 for dogs) to each respective image. These images and their labels are then incorporated into the 'train' DataFrame. To ensure the data is not biased by any inherent ordering, the dataset is randomized using the 'sample' function, and the DataFrame index is reset for uniformity. A preview of the initial rows of the 'train' DataFrame is presented for a preliminary review of the dataset composition.

**Loading Datasets for Testing and Validation:**

In a similar manner, datasets for testing and validation are prepared, involving the loading and labeling of image data into appropriate DataFrames.

**Data Visualization and Exploration:**

This code segment is dedicated to the examination and visualization of the training, testing, and validation datasets. A count plot illustrating the label distribution within the dataset is generated, offering insight into the class balance between cats and dogs. Additionally, the total count of images available in the training dataset is calculated and displayed. This exploratory step is crucial for gaining an understanding of the dataset's structure, which is instrumental in the model design and training phase.

**Visualizing Images in a Grid:**

**Library Imports:**

The process begins with the importation of essential libraries, including Keras for handling image operations and Matplotlib for the visualization of images.

**Setting Up the Visualization Canvas:**

A Matplotlib figure is set up, measuring 20x20 inches, serving as the canvas where a collection of images will be displayed in a structured grid format.

**Choosing a Subset from the Training Data:**

A specific portion of the training dataset is earmarked for visualization, particularly the initial 25 entries from the 'train' DataFrame. These images are arranged to form a 5x5 grid.

**Looping Through and Visualizing Images:**

Through iteration over the chosen image subset, each image's path and associated label are retrieved from the 'train' DataFrame.

For every image, a subplot is designated within the 5x5 grid, determined by the subplot function of Matplotlib, where the index+1 dictates each subplot's placement.

Images are loaded as NumPy arrays using the load\_img function from Keras and are stored in a variable named 'img'.

Each image is then displayed on its assigned subplot using plt.imshow, with plt.title(label) assigning the correct label as the subplot's title.

Axis labels and ticks are removed using plt.axis('off') to enhance the visual clarity of the displayed images.

**Rendering the Image Grid:**

Upon completing the iteration over the 25 selected images, the compiled grid is presented within the established Matplotlib figure.

**Convolutional neural network (CNN) model for image classification using Keras (contd**):

1. **Model Architecture Definition:**
   * model\_1 = Sequential([...]): This line defines a sequential model named 'model\_1.' The model consists of a sequence of layers defined within the square brackets.
2. **Convolutional Layers:**
   * Several convolutional layers are defined to extract features from input images:
     + The first convolutional layer has 16 filters, a 3x3 kernel size, and ReLU activation. It expects input images of size (224, 224, 3) in RGB format.
     + The second convolutional layer has 32 filters, a 3x3 kernel size, and ReLU activation.
     + The third convolutional layer has 32 filters, a 3x3 kernel size, and ReLU activation.
     + The fourth convolutional layer has 64 filters, a 3x3 kernel size, and ReLU activation.
3. **Max-Pooling Layers:**
   * Max-pooling layers are inserted to downsample the feature maps:
     + After the first convolutional layer, a max-pooling layer with a 2x2 pooling window is added.
     + After the second pair of convolutional layers, another max-pooling layer with a 2x2 pooling window is added.
4. **Flatten Layer:**
   * A flatten layer is included to transform the multi-dimensional feature maps into a one-dimensional vector.
5. **Fully Connected Layers:**
   * A dense (fully connected) layer with 2 output units and sigmoid activation is added. This layer is used for binary classification, and it outputs class probabilities.
6. **Model Compilation:**
   * model\_1.compile(...) is used to compile the model. The following settings are specified:
     + Loss function: Categorical cross-entropy ('categorical\_crossentropy') is used as the loss function, suitable for multi-class classification.
     + Optimizer: 'Adam' is chosen as the optimization algorithm, which is a popular choice for gradient-based optimization.
     + Metrics: 'accuracy' is used as the evaluation metric to monitor the model's performance during training.
7. **Model Training:**
   * The model is trained using the training dataset ('training\_set') with the following settings:
     + Number of epochs: Training is performed for 5 epochs (you can adjust this as needed).
     + Steps per epoch: The number of steps per epoch is set to the length of the training set. This ensures that the entire training dataset is used in each epoch.
     + Validation data: The model's performance is evaluated on a validation dataset ('validation\_set').
     + Validation steps: The number of steps for validation is set to the length of the validation set.

**Data Augmentation and Preprocessing for the Training Set:**

**Library Importation:**

The procedure initiates by importing Keras' ImageDataGenerator, a versatile tool for image preprocessing and augmentation.

**Setting Up ImageDataGenerator:**

* A new instance of ImageDataGenerator, named 'train\_datagen', is instantiated for augmenting the training images.
* Applied data augmentation techniques include:
* rescale=1./255 to normalize image pixel values.
* shear\_range=0.2 introduces shearing transformations.
* zoom\_range=0.2 applies random zoom for scale variation.
* horizontal\_flip=True for generating horizontally flipped versions of images.
* Initializing the Training Data Generator:
* A generator for the training data, referred to as 'training\_set', is established using 'train\_datagen'.
* The flow\_from\_dataframe method is employed to load images and labels from the 'train' DataFrame, specifying:
* dataframe=train for the source DataFrame.
* x\_col='image' to identify where image paths are stored.
* y\_col='label' for the corresponding labels.
* target\_size=(224, 224) to resize images.
* batch\_size=Batch\_size to set the processing batch size.
* class\_mode='categorical' indicating that labels are one-hot encoded.
* This process is mirrored for both training and validation datasets.

**Building a CNN Model for Image Classification with Keras:**

**Model Foundation:**

A Sequential model is instantiated, providing a base for stacking layers linearly.

**Incorporating Convolutional Layers:**

* The model integrates a Conv2D layer with 32 filters, a 3x3 kernel size, ReLU activation, and an input shape of (224, 224, 3).
* 32 filters are utilized for detecting diverse image features.
* A 3x3 kernel size is chosen for the convolution operation.
* ReLU activation introduces non-linearity, enhancing learning capability.
* The specified input shape accommodates images of 224x224 pixels in RGB.
* Followed by the initial convolutional layer, a MaxPooling2D layer with a 2x2 window reduces feature map dimensions.
* Additional convolutional and max-pooling layers are added to deepen the model.

**Flattening Output:**

A Flatten layer transitions the model from convolutional layers to fully connected layers by converting 2D feature maps into a 1D vector.

**Adding Dense Layers:**

* A dense layer with 64 neurons and ReLU activation is included for complex pattern learning within the extracted features.
* The final layer, a dense layer with 2 neurons and sigmoid activation, is employed for binary classification, outputting probabilities for each class.

**CNN model with convolutional layers, max-pooling layers, and fully connected layers for image classification**

1. **Model Architecture Definition:**
   * A sequential model named 'model\_1' is defined with a sequence of layers:
     + The first layer is a convolutional layer with 16 filters, a 3x3 kernel size, and ReLU activation. It expects input images of size (224, 224, 3) in RGB format.
     + The second layer is another convolutional layer with 32 filters, a 3x3 kernel size, and ReLU activation.
     + A max-pooling layer follows with a 2x2 pooling window and 'valid' padding.
     + Two additional convolutional layers follow with 32 and 64 filters, both using 3x3 kernels and ReLU activation.
     + Another max-pooling layer follows with a 2x2 pooling window.
     + A flatten layer is included to transform the multi-dimensional feature maps into a one-dimensional vector.
     + The final layer is a dense (fully connected) layer with 2 output units and sigmoid activation, suitable for binary classification.
2. **Model Compilation:**
   * model\_1.compile(...) is used to compile the model. The following settings are specified:
     + Loss function: Categorical cross-entropy ('categorical\_crossentropy') is chosen as the loss function, typically used for multi-class classification tasks.
     + Optimizer: 'Adam' is selected as the optimization algorithm, which is a popular choice for gradient-based optimization.
     + Metrics: 'accuracy' is used as the evaluation metric to monitor the model's performance during training.
3. **Model Training:**
   * The model is trained using the training dataset ('training\_set') with the following settings:
     + Number of epochs: Training is performed for 10 epochs (you can adjust this as needed).
     + Steps per epoch: The number of steps per epoch is set to the length of the training set. This ensures that the entire training dataset is used in each epoch.
     + Validation data: The model's performance is evaluated on a validation dataset ('validation\_set').
     + Validation steps: The number of steps for validation is set to the length of the validation set.

**Model Evaluation:**

* model\_1: This is the CNN model that was defined, compiled, and trained previously.
* .evaluate(testing\_set): This method is called on the model to assess its performance. It takes the testing dataset (testing\_set) as input for evaluation



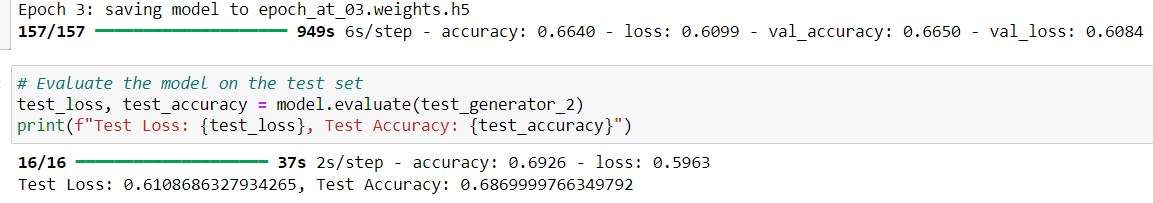
**2. Increase your training sample size. You may pick any amount. Keep the validation and test samples the same as above. Optimize your network (again training from scratch). What performance did you achieve?**

Here the training dataset has been increased to 5000 inorder to train the data to get the better results with accuracy of the model.

1. **Importing Libraries:**
   * The code starts by importing the necessary libraries, including Pandas for data manipulation, Seaborn for data visualization, and Matplotlib for creating plots.
2. **Creating a Training DataFrame (train\_v2):**
   * A Pandas DataFrame named 'train\_v2' is created to store the training dataset.
   * The code attempts to load image data and their corresponding labels from a directory specified in 'TRAIN\_V2\_DIR' using the 'load\_dataset' function.
   * The 'load\_dataset' function is expected to read image files, process them, and assign labels to each image.
3. **Shuffling the Dataset:**
   * The 'train\_v2' DataFrame is shuffled to randomize the order of the data. This helps avoid any biases that might be introduced by a specific order of data. Shuffling is performed using the 'sample' function with 'frac=1' to retain all data and reset the index.
4. **Creating a Count Plot of Labels:**
   * A count plot of the labels in the 'train\_v2' DataFrame is generated using Seaborn. The 'countplot' function is used to visualize the distribution of labels in the training dataset.
   * The plot is displayed within a figure of size 10x6, with a title ('Label Count Plot') for clarity.
   * The x-axis is labeled as 'Label,' and the y-axis is labeled as 'Count.'
5. **Printing Total Image Count:**
   * The code calculates and prints the total number of images in the 'train\_v2' dataset. This count is determined by finding the length of the 'train\_v2' DataFrame.
6. **Displaying the Plot:**
   * Finally, the plot is shown using 'plt.show()'.

We repeat the same steps using the above question for Model classification Using Convolutional Neural Network(CNN)

Performance achieved from the new Dataset:



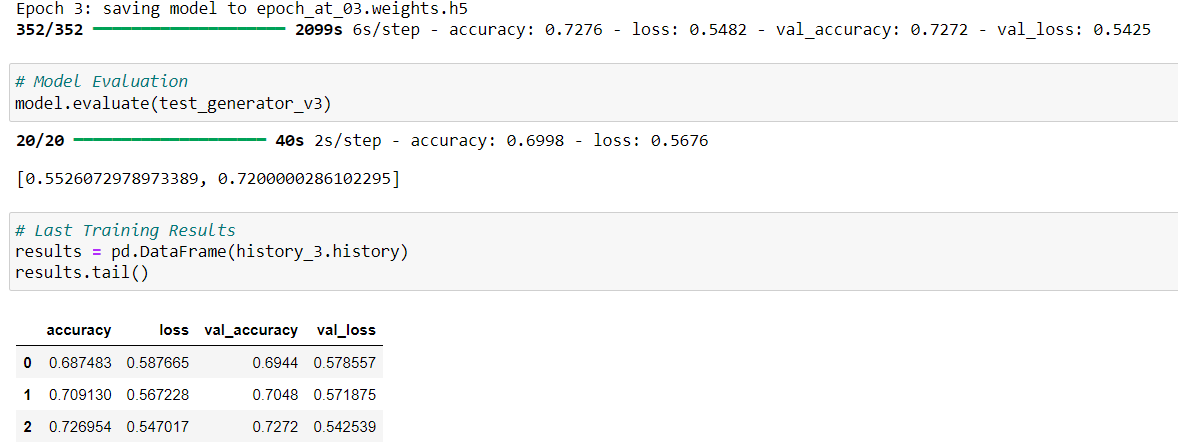
**3. Now change your training sample so that you achieve better performance than those from Steps 1 and 2. This sample size may be larger, or smaller than those in the previous steps. The objective is to find the ideal training sample size to get best prediction results.**

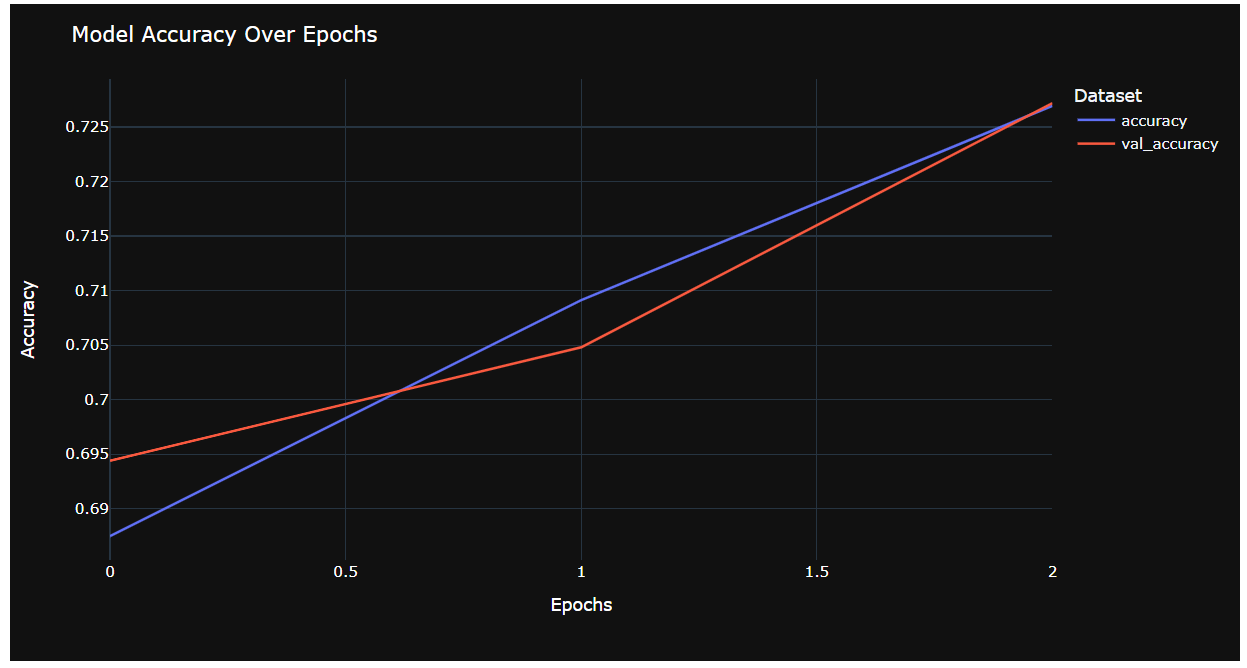
Based on the 1st and 2nd question it could be inferred that with increase in dataset the model accuracy could be improved.

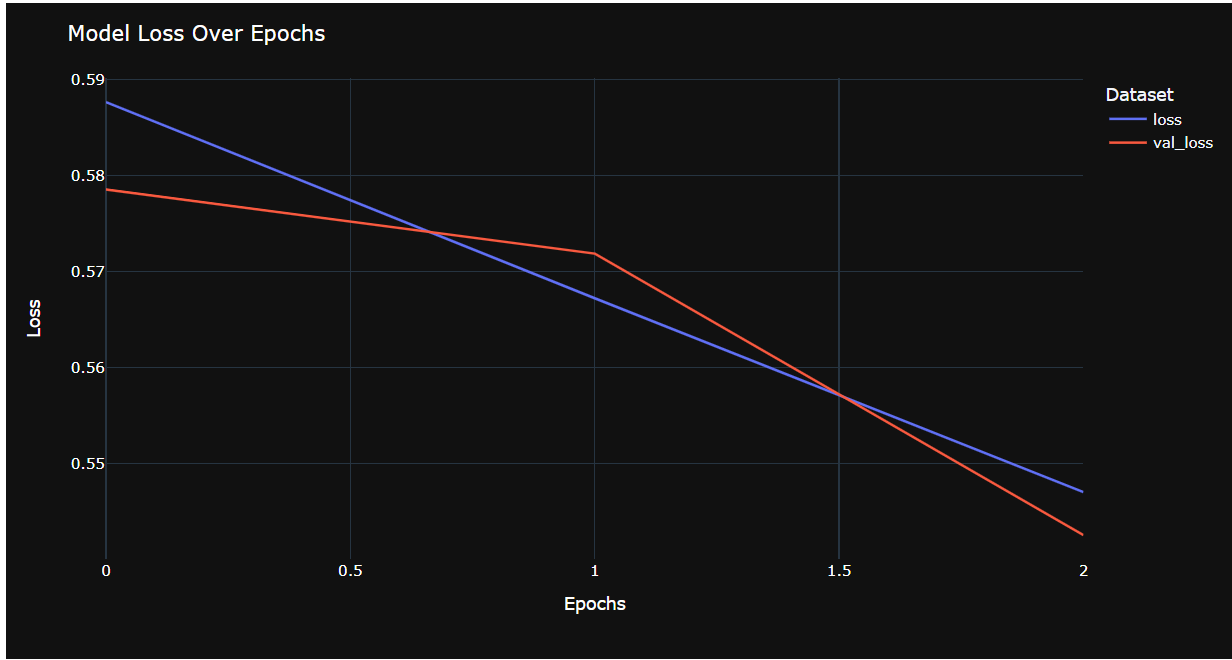
The dataset is now increased to 22000 images in training and it is trained by using the above steps, after training the data with 22000 images it is identified that the accuracy has improved a bit.

The results for the same can be seen below:

**Model Evaluation:**







**4. Repeat Steps 1-3, but now using a pretrained network. The sample sizes you use in Steps 2 and 3 for the pretrained network may be the same or different from those using the network where you trained from scratch. Again, use any and all optimization techniques to get best performance**

**MobileVnet** is the pretrained model used here to train the dataset.

Model Architecture

Base Model: The core of the model utilizes MobileNetV2, pre-trained on the extensive ImageNet dataset. This choice ensures the model benefits from high-quality feature representations for a wide variety of images. The pre-trained model is integrated with the input shape of 150x150 pixels across three color channels (RGB), tailored to the task's specific requirements.

Model Customization: To adapt MobileNetV2 for binary classification, the top layer of the network (originally designed for 1000 classes) is removed. This allows for the addition of custom layers tailored to the task.

Global Average Pooling: Following the base model, a Global Average Pooling 2D layer is applied. This layer helps reduce the dimensions of the feature maps to a single vector per map, significantly decreasing the model's complexity and computational cost while retaining essential spatial information.

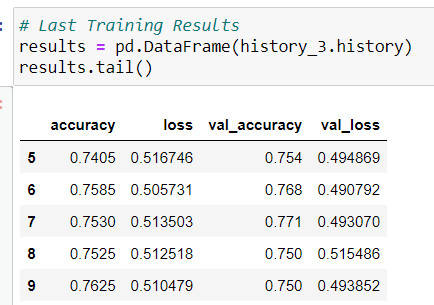
Dense Layers: The flattened output is then passed through a Dense layer with 1024 neurons, employing the ReLU activation function for non-linear processing. This layer serves to learn high-level features derived from the base model's output. The final Dense layer, consisting of a single neuron with a sigmoid activation function, produces the binary output indicating the class probabilities (cat or dog).

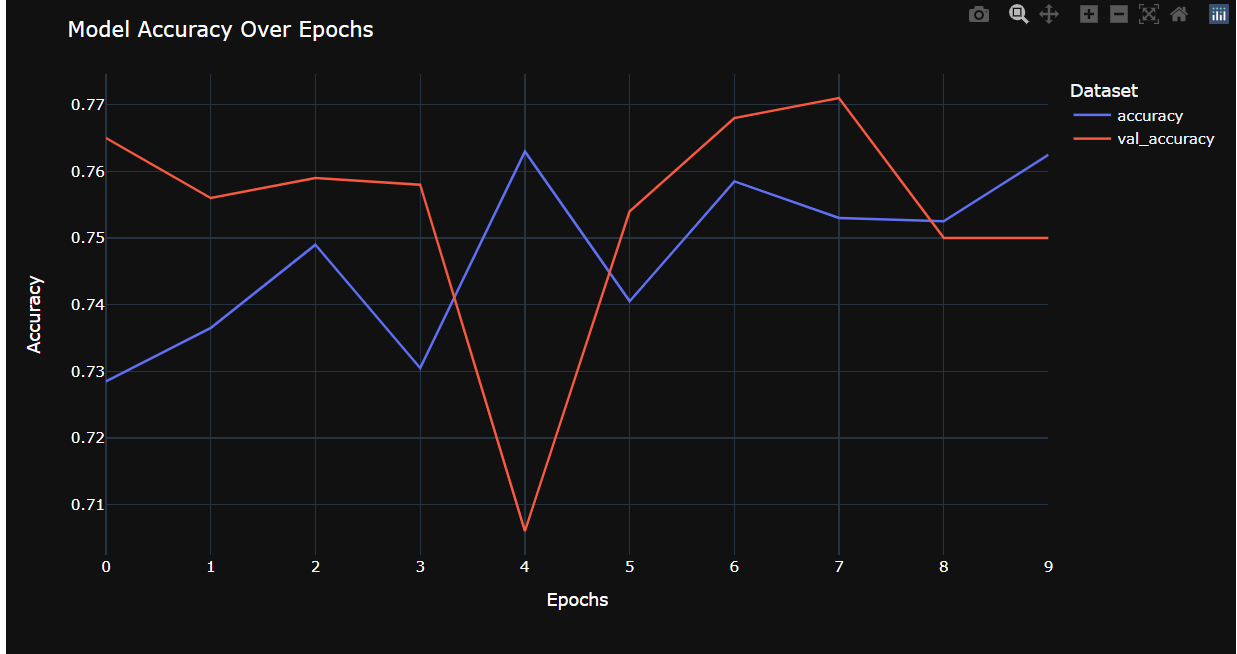
Model Compilation

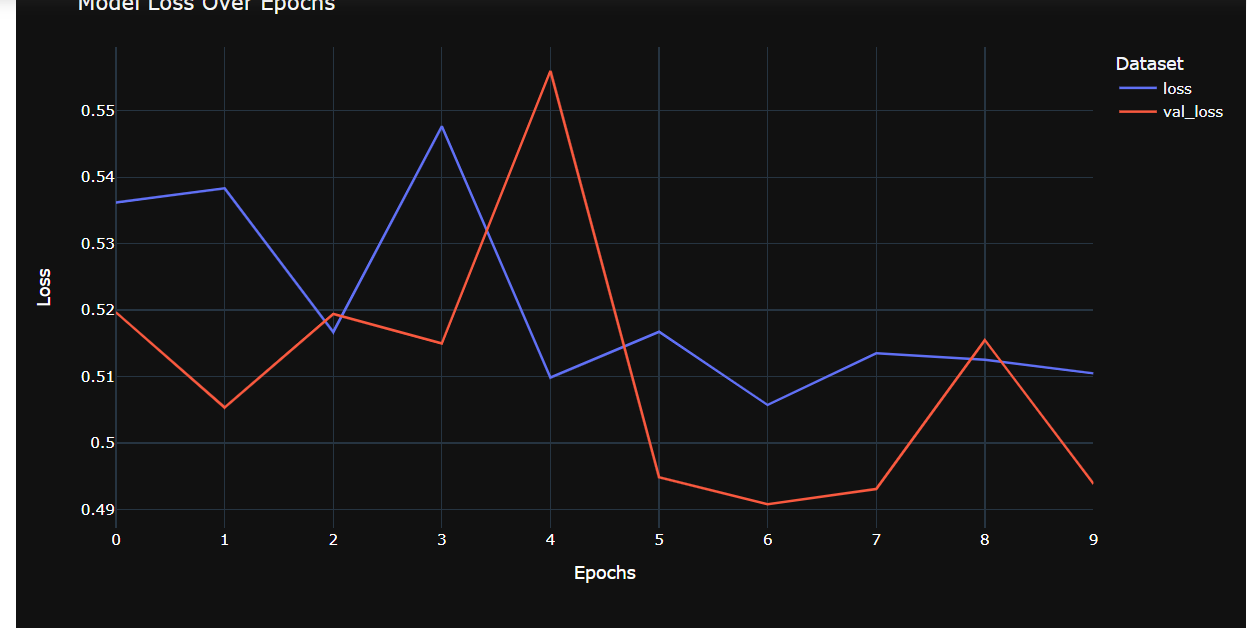
Optimizer: The model utilizes the RMSprop optimizer, with a learning rate set to 0.001. RMSprop is chosen for its efficiency in handling the gradients in mini-batch learning, especially beneficial for tasks with highly non-linear objectives like image classification.

Loss Function: Binary crossentropy is selected as the loss function, fitting for binary classification problems where the goal is to minimize the distance between the actual and predicted probability distributions.

Metrics: Accuracy is used as the metric to evaluate the model's performance, providing a straightforward interpretation of its effectiveness in classifying images as either cats or dogs.







**CNN model on the training dataset for 10 epochs while validating its performance on a separate validation dataset**

model\_pre.fit(...): This method is used to train the CNN model (model\_pre) with the specified training and validation datasets. It performs the following tasks:

* training\_v3\_set: This is the training dataset provided for model training.
* steps\_per\_epoch=len(training\_v3\_set): The number of steps (batches) per training epoch is set to the length of the training dataset. This ensures that the entire training dataset is used in each epoch.
* epochs=10: The model is trained for 10 epochs. An epoch is a complete pass through the entire training dataset.
* validation\_data=validation\_set: The model's performance is evaluated on the validation dataset (validation\_set) after each epoch. This allows monitoring how well the model generalizes to new, unseen data.
* validation\_steps=len(validation\_set): The number of steps (batches) for validation is set to the length of the validation dataset

**Evaluation of the model based on the Training dataset:**

* model\_pre.evaluate(testing\_set): This line of code invokes the evaluate method on the model\_pre model to assess its performance on the testing dataset. The evaluation process involves the following steps:
  + testing\_set: This is the testing dataset provided for evaluation. It contains a set of data examples and their corresponding labels.
  + The model processes the data in the testing dataset and computes its performance metrics, such as accuracy and loss, on this dataset.
* The evaluation results are typically returned as a tuple of values, which may include accuracy, loss, and other metrics specified during model compilation. These results can be used to assess how well the model generalizes to unseen data and to make informed decisions about its performance.

**Visualization of Model Training Outcomes:**

**Retrieving Training Metrics:**

Initially, the process involves extracting data from the history\_pre object, which houses the metrics recorded throughout the model's training phase. This includes both the accuracy and loss metrics for training and validation phases.

**Plotting the Accuracy:**

An accuracy chart is constructed, showcasing the progression of training accuracy (depicted in blue) against validation accuracy (illustrated in red) across epochs.

The horizontal axis marks the epoch count, whereas the vertical axis indicates the accuracy percentage.

The chart is titled 'Accuracy Graph', and includes a legend to distinguish between training and validation accuracy.

**Illustrating the Loss:**

Similarly, a loss graph is plotted, comparing training loss (in blue) to validation loss (in red) over the epochs.

The epochs are again plotted on the horizontal axis, with loss values on the vertical axis.

The graph is aptly named 'Loss Graph' and features a legend to separate training loss from validation loss figures.

**Graph Display:**

Utilizing the plt.show() function, both the accuracy and loss charts are rendered for review.

**Overview:**

**Enhancements in CNN Performance Through Expanded Datasets:**

**Augmentation of Data Diversity:**

Expansive datasets introduce a broader spectrum of examples, enhancing the model's ability to generalize effectively to new data.

An increased image count exposes the model to varied conditions, fostering a robust learning environment.

**Minimization of Overfitting Risks:**

Larger datasets dilute the risk of overfitting by broadening the learning scope with an extensive array of examples.

This breadth of data steers the model towards learning significant patterns rather than memorizing specific instances.

**Enhancement of Feature Detection:**

With more data at its disposal, a CNN can refine its ability to detect nuanced features, leading to a deeper understanding of the data.

This enables the recognition of more intricate and abstract features, thereby capturing complex data relationships.

**Optimization Benefits:**

Larger datasets contribute to a more stable and accurate optimization process, aiding in the model's convergence to optimal solutions.

Stability in gradient updates is enhanced, promoting smoother optimization trajectories.

**Strategies for Efficient Training on Large Datasets:**

**Batch Normalization:**

Facilitates training stability and model generalization by normalizing layer inputs.

**Adaptive Learning Rate Adjustments:**

Implementing schedules for learning rate adjustments can significantly aid in model convergence, especially crucial when handling extensive datasets.

**Utilization of Data Augmentation:**

By artificially expanding the dataset through transformations, models can achieve better generalization, a technique particularly valuable for extensive datasets.

**Incorporation of Regularization Methods:**

Regularization strategies, including dropout and L2 regularization, ensure the model focuses on learning generalizable patterns.

**Employment of Ensemble Techniques:**

Ensemble methods, which combine predictions from multiple models, can leverage model diversity to boost overall performance.

**Application of Transfer Learning:**

Leveraging models pretrained on vast datasets as foundational structures for specific tasks can yield superior outcomes, courtesy of their pre-learned feature representations.