Assignment 2: Convolution - Yashwanth Kothuru

Importing the required packages

In [1]:
```
# Standard library imports
import os
import shutil
import random

# Data handling
import numpy as np
import pandas as pd

# Machine Learning and Neural Network libraries
import tensorflow as tf
from tensorflow.keras.layers import Conv2D, Add, MaxPooling2D, Dense, BatchNormaliz
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Visualization libraries
import matplotlib.pyplot as plt
import plotly.express as px
import seaborn as sns

# SciPy and Sklearn for additional functionalities
import scipy as sp
from scipy import ndimage
from sklearn.metrics import confusion_matrix, roc_curve, auc
```

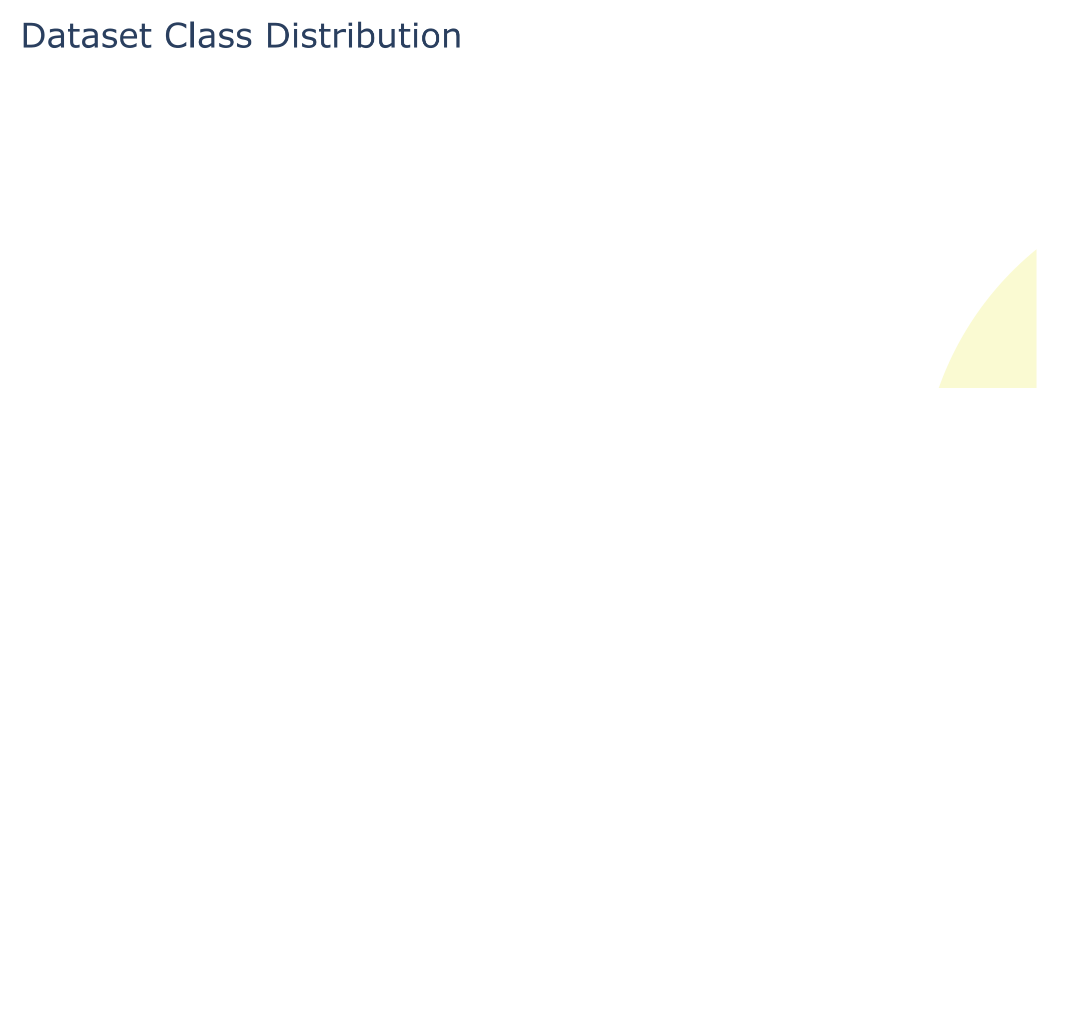
In [2]:
```
# File Paths for Dataset
cat_file_path = "C:/Users/yash/Downloads/archive/PetImages/Cat"
dog_file_path = "C:/Users/yash/Downloads/archive/PetImages/Dog"
```

In [3]:
```
import plotly.express as px

class_names = ['Cat', 'Dog']
n_dogs = len(os.listdir(cat_file_path))
n_cats = len(os.listdir(dog_file_path))
n_images = [n_cats, n_dogs]

fig = px.pie(names=class_names, values=n_images, title='Dataset Class Distribution'
             color=class_names,
             color_discrete_map={'Cat':'lightcyan', 'Dog':'lightgoldenrodyellow'})

fig.update_traces(textinfo='percent+label+value', pull=[0.1, 0])
fig.show()
```



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?

In [4]:
```
import os
import random

def split_data_fixed_sizes(main_dir, train_size=1000, val_size=500):
    files = []
    for file in os.listdir(main_dir):
        if os.path.getsize(os.path.join(main_dir, file)):  # check if the file's si
            files.append(file)  # appends file name to a list

    shuffled_files = random.sample(files, len(files))  # shuffles the data

    # Assign files to train, validation, and test sets based on specified sizes
    train_files = shuffled_files[:train_size]
    validation_files = shuffled_files[train_size:train_size+val_size]
    test_files = shuffled_files[train_size+val_size:train_size+val_size+500]  # Ass

    return train_files, validation_files, test_files
```

```
# Split the data for cats and dogs
train_cats, val_cats, test_cats = split_data_fixed_sizes(cat_file_path)
train_dogs, val_dogs, test_dogs = split_data_fixed_sizes(dog_file_path)

print("Cats - Training:", len(train_cats), "Validation:", len(val_cats), "Test:", l
print("Dogs - Training:", len(train_dogs), "Validation:", len(val_dogs), "Test:", l
```

```
Cats - Training: 1000 Validation: 500 Test: 500
Dogs - Training: 1000 Validation: 500 Test: 500
```

In [5]:
```
# Function to create a dataframe from file names and labels
def create_dataframe(file_names, label, main_dir):
    return pd.DataFrame({
        'filename': [os.path.join(main_dir, fname) for fname in file_names],
        'class': label
    })

# Create dataframes
train_df = pd.concat([create_dataframe(train_cats, 'cat', cat_file_path), create_da
val_df = pd.concat([create_dataframe(val_cats, 'cat', cat_file_path), create_datafr
test_df = pd.concat([create_dataframe(test_cats, 'cat', cat_file_path), create_data

# Shuffle the dataframes
train_df = train_df.sample(frac=1).reset_index(drop=True)
val_df = val_df.sample(frac=1).reset_index(drop=True)
test_df = test_df.sample(frac=1).reset_index(drop=True)
```

In [6]:
```
# Image Data Generators
train_gen = ImageDataGenerator(rescale=1./255)
validation_gen =  ImageDataGenerator(rescale=1./255.)
test_gen =  ImageDataGenerator(rescale=1./255.)
```

In [7]:
```
# Data Generators from Dataframes
train_generator = train_gen.flow_from_dataframe(dataframe=train_df, x_col='filename
validation_generator = validation_gen.flow_from_dataframe(dataframe=val_df, x_col='
test_generator = test_gen.flow_from_dataframe(dataframe=test_df, x_col='filename',
```

```
Found 2000 validated image filenames belonging to 2 classes.
Found 1000 validated image filenames belonging to 2 classes.
Found 1000 validated image filenames belonging to 2 classes.
```

In [8]:
```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import os

def plot_images_from_directory(file_paths, directory_path, title, num_images=4):
    fig, axes = plt.subplots(1, num_images, figsize=(15, 10))
    fig.suptitle(title, fontsize=20)
    for i, file_path in enumerate(file_paths[:num_images]):
        img_path = os.path.join(directory_path, file_path)
        img = mpimg.imread(img_path)
        axes[i].imshow(img)
        axes[i].axis('off')
    plt.show()

# Assuming `train_cats` and `train_dogs` contain the file names of the images
cat_images_to_plot = train_cats[:4]  # Select the first 4 cat images
dog_images_to_plot = train_dogs[:4]  # Select the first 4 dog images

# Replace 'cat_file_path' and 'dog_file_path' with the actual directories containin
plot_images_from_directory(cat_images_to_plot, cat_file_path, 'Plotting Few Cat Ima
plot_images_from_directory(dog_images_to_plot, dog_file_path, 'Plotting Few Dog Ima
```

Plotting Few Cat Images

Plotting Few Dog Images

In [9]:

```
# Building the CNN model
inputs = tf.keras.layers.Input(shape=(150,150,3))
# Convolutional layer with 32 filters of size 3x3, using ReLU activation function
x = tf.keras.layers.Conv2D(32, (3,3), activation='relu')(inputs)
# Another Convolutional layer with increased filters for capturing more complex fea
x = tf.keras.layers.Conv2D(64, (3,3), activation='relu')(x)
# MaxPooling to reduce dimensionality and prevent overfitting
x = tf.keras.layers.MaxPooling2D(2,2)(x)

# Additional Convolutional layers with increasing filter size for deeper feature ex
x = tf.keras.layers.Conv2D(64, (3,3), activation='relu')(x)
x = tf.keras.layers.Conv2D(128, (3,3), activation='relu')(x)
x = tf.keras.layers.MaxPooling2D(2,2)(x)

x = tf.keras.layers.Conv2D(128, (3,3), activation='relu')(x)
x = tf.keras.layers.Conv2D(256, (3,3), activation='relu')(x)

# Global Average Pooling layer for reducing overfitting and improving model efficie
x = tf.keras.layers.GlobalAveragePooling2D()(x)

# Dense layers for classification
```

```
x = Dense(1024, activation='relu')(x)
x = tf.keras.layers.Dense(2, activation='softmax')(x)

model = Model(inputs=inputs, outputs=x)
```

In [10]:
```
model.summary()
```

Model: "functional_1"

Layer (type)	Output Shape
input_layer (InputLayer)	(None, 150, 150, 3)
conv2d (Conv2D)	(None, 148, 148, 32)
conv2d_1 (Conv2D)	(None, 146, 146, 64)
max_pooling2d (MaxPooling2D)	(None, 73, 73, 64)
conv2d_2 (Conv2D)	(None, 71, 71, 64)
conv2d_3 (Conv2D)	(None, 69, 69, 128)
max_pooling2d_1 (MaxPooling2D)	(None, 34, 34, 128)
conv2d_4 (Conv2D)	(None, 32, 32, 128)
conv2d_5 (Conv2D)	(None, 30, 30, 256)
global_average_pooling2d (GlobalAveragePooling2D)	(None, 256)
dense (Dense)	(None, 1024)
dense_1 (Dense)	(None, 2)

Total params: 838,146 (3.20 MB)

Trainable params: 838,146 (3.20 MB)

Non-trainable params: 0 (0.00 B)

In [11]:
```
# Model Compilation
model.compile(optimizer=tf.keras.optimizers.RMSprop(learning_rate=0.001),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
# Model Training with ModelCheckpoint Callback
checkpoint_filepath = "epoch_at_{epoch:02d}.weights.h5"
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_filepath,
    save_weights_only=True,
    monitor='val_accuracy',
    mode='max',
    save_best_only=False,
    verbose=1
)
```

In [12]:
```
# Model Training
r = model.fit(
    train_generator,
    epochs=10,
```

```
    validation_data=validation_generator,
    callbacks=[model_checkpoint_callback]  # Add the callback here
)
```

```
Epoch 1/10
C:\Users\yeswa\anaconda3\Lib\site-packages\keras\src\trainers\data_adapters\py_dat
aset_adapter.py:120: UserWarning:

Your `PyDataset` class should call `super().__init__(**kwargs)` in its constructo
r. `**kwargs` can include `workers`, `use_multiprocessing`, `max_queue_size`. Do n
ot pass these arguments to `fit()`, as they will be ignored.

32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.4968 - loss: 0.6982
Epoch 1: saving model to epoch_at_01.weights.h5
32/32 ━━━━━━━━━━━━━━━━━━━━ 239s 7s/step - accuracy: 0.4969 - loss: 0.6982 - val_ac
curacy: 0.5000 - val_loss: 0.6932
Epoch 2/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.5083 - loss: 0.6931
Epoch 2: saving model to epoch_at_02.weights.h5
32/32 ━━━━━━━━━━━━━━━━━━━━ 222s 7s/step - accuracy: 0.5080 - loss: 0.6931 - val_ac
curacy: 0.5170 - val_loss: 0.6917
Epoch 3/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.4832 - loss: 0.7055
Epoch 3: saving model to epoch_at_03.weights.h5
32/32 ━━━━━━━━━━━━━━━━━━━━ 222s 7s/step - accuracy: 0.4842 - loss: 0.7054 - val_ac
curacy: 0.5860 - val_loss: 0.6882
Epoch 4/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.5173 - loss: 0.6923
Epoch 4: saving model to epoch_at_04.weights.h5
32/32 ━━━━━━━━━━━━━━━━━━━━ 217s 7s/step - accuracy: 0.5178 - loss: 0.6923 - val_ac
curacy: 0.6050 - val_loss: 0.6836
Epoch 5/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.5461 - loss: 0.6927
Epoch 5: saving model to epoch_at_05.weights.h5
32/32 ━━━━━━━━━━━━━━━━━━━━ 222s 7s/step - accuracy: 0.5463 - loss: 0.6926 - val_ac
curacy: 0.6080 - val_loss: 0.6771
Epoch 6/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.5490 - loss: 0.6862
Epoch 6: saving model to epoch_at_06.weights.h5
32/32 ━━━━━━━━━━━━━━━━━━━━ 226s 7s/step - accuracy: 0.5490 - loss: 0.6862 - val_ac
curacy: 0.5160 - val_loss: 0.6845
Epoch 7/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.5494 - loss: 0.6832
Epoch 7: saving model to epoch_at_07.weights.h5
32/32 ━━━━━━━━━━━━━━━━━━━━ 226s 7s/step - accuracy: 0.5498 - loss: 0.6832 - val_ac
curacy: 0.6080 - val_loss: 0.6685
Epoch 8/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.6069 - loss: 0.6589
Epoch 8: saving model to epoch_at_08.weights.h5
32/32 ━━━━━━━━━━━━━━━━━━━━ 220s 7s/step - accuracy: 0.6066 - loss: 0.6590 - val_ac
curacy: 0.5610 - val_loss: 0.6750
Epoch 9/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.5713 - loss: 0.6684
Epoch 9: saving model to epoch_at_09.weights.h5
32/32 ━━━━━━━━━━━━━━━━━━━━ 219s 7s/step - accuracy: 0.5717 - loss: 0.6682 - val_ac
curacy: 0.6070 - val_loss: 0.6566
Epoch 10/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.6091 - loss: 0.6546
Epoch 10: saving model to epoch_at_10.weights.h5
32/32 ━━━━━━━━━━━━━━━━━━━━ 216s 7s/step - accuracy: 0.6088 - loss: 0.6548 - val_ac
curacy: 0.6100 - val_loss: 0.6576
```

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?

In [13]:
```
import os
import random

def split_data_fixed_sizes(main_dir, train_size=5000, val_size=500):
    files = []
    for file in os.listdir(main_dir):
        if os.path.getsize(os.path.join(main_dir, file)):  # check if the file's si
            files.append(file)  # appends file name to a list

    shuffled_files = random.sample(files, len(files))  # shuffles the data

    # Assign files to train, validation, and test sets based on specified sizes
    train_files = shuffled_files[:train_size]
    validation_files = shuffled_files[train_size:train_size+val_size]
    test_files = shuffled_files[train_size+val_size:train_size+val_size+500]  # Ass

    return train_files, validation_files, test_files

# Split the data for cats and dogs with updated sizes
train_cats_2, val_cats_2, test_cats_2 = split_data_fixed_sizes(cat_file_path, trair
train_dogs_2, val_dogs_2, test_dogs_2 = split_data_fixed_sizes(dog_file_path, trair

print("Cats - Training:", len(train_cats_2), "Validation:", len(val_cats_2), "Test:
print("Dogs - Training:", len(train_dogs_2), "Validation:", len(val_dogs_2), "Test:
```

```
Cats - Training: 5000 Validation: 500 Test: 500
Dogs - Training: 5000 Validation: 500 Test: 500
```

In [14]:
```
import pandas as pd
import os
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Assuming you've already executed the split_data_fixed_sizes function with the upc
# And you've updated your file paths 'cat_file_path' and 'dog_file_path'

# Use the updated lists for creating dataframes
train_df_2 = pd.concat([
    create_dataframe(train_cats_2, 'cat', cat_file_path),
    create_dataframe(train_dogs_2, 'dog', dog_file_path)
])

val_df_2 = pd.concat([
    create_dataframe(val_cats_2, 'cat', cat_file_path),
    create_dataframe(val_dogs_2, 'dog', dog_file_path)
])

test_df_2 = pd.concat([
    create_dataframe(test_cats_2, 'cat', cat_file_path),
    create_dataframe(test_dogs_2, 'dog', dog_file_path)
])
```

```python
# Shuffle the dataframes
train_df_2 = train_df_2.sample(frac=1).reset_index(drop=True)
val_df_2 = val_df_2.sample(frac=1).reset_index(drop=True)
test_df_2 = test_df_2.sample(frac=1).reset_index(drop=True)

# Assuming the ImageDataGenerator instantiation stays the same
train_gen = ImageDataGenerator(rescale=1./255)
validation_gen = ImageDataGenerator(rescale=1./255)
test_gen = ImageDataGenerator(rescale=1./255)

# Creating the data generators from the new dataframes
train_generator_2 = train_gen.flow_from_dataframe(
    dataframe=train_df_2,
    x_col='filename',
    y_col='class',
    target_size=(150, 150),
    batch_size=64,
    class_mode='binary'
)

validation_generator_2 = validation_gen.flow_from_dataframe(
    dataframe=val_df_2,
    x_col='filename',
    y_col='class',
    target_size=(150, 150),
    batch_size=64,
    class_mode='binary'
)

test_generator_2 = test_gen.flow_from_dataframe(
    dataframe=test_df_2,
    x_col='filename',
    y_col='class',
    target_size=(150, 150),
    batch_size=64,
    class_mode='binary'
)
```

```
Found 9999 validated image filenames belonging to 2 classes.
C:\Users\yeswa\anaconda3\Lib\site-packages\keras\src\legacy\preprocessing\image.p
y:920: UserWarning:

Found 1 invalid image filename(s) in x_col="filename". These filename(s) will be i
gnored.

Found 1000 validated image filenames belonging to 2 classes.
Found 1000 validated image filenames belonging to 2 classes.
```

Optimizing the model

In [20]:

```python
# Building the CNN model
inputs = tf.keras.layers.Input(shape=(150,150,3))
# Convolutional layer with 32 filters of size 3x3, using ReLU activation function
x = tf.keras.layers.Conv2D(32, (3,3), activation='relu')(inputs)
# Another Convolutional layer with increased filters for capturing more complex fea
x = tf.keras.layers.Conv2D(64, (3,3), activation='relu')(x)
# MaxPooling to reduce dimensionality and prevent overfitting
x = tf.keras.layers.MaxPooling2D(2,2)(x)

# Additional Convolutional layers with increasing filter size for deeper feature ex
x = tf.keras.layers.Conv2D(64, (3,3), activation='relu')(x)
```

```
x = tf.keras.layers.Conv2D(128, (3,3), activation='relu')(x)
x = tf.keras.layers.MaxPooling2D(2,2)(x)

x = tf.keras.layers.Conv2D(128, (3,3), activation='relu')(x)
x = tf.keras.layers.Conv2D(256, (3,3), activation='relu')(x)

# Global Average Pooling layer for reducing overfitting and improving model efficie
x = tf.keras.layers.GlobalAveragePooling2D()(x)

# Dense layers for classification
x = Dense(1024, activation='relu')(x)
x = tf.keras.layers.Dense(2, activation='softmax')(x)

model_2 = Model(inputs=inputs, outputs=x)
```

In [21]:
```
model_2.summary()
```

Model: "functional_5"

Layer (type)	Output Shape	
input_layer_2 (InputLayer)	(None, 150, 150, 3)	
conv2d_12 (Conv2D)	(None, 148, 148, 32)	
conv2d_13 (Conv2D)	(None, 146, 146, 64)	
max_pooling2d_4 (MaxPooling2D)	(None, 73, 73, 64)	
conv2d_14 (Conv2D)	(None, 71, 71, 64)	
conv2d_15 (Conv2D)	(None, 69, 69, 128)	
max_pooling2d_5 (MaxPooling2D)	(None, 34, 34, 128)	
conv2d_16 (Conv2D)	(None, 32, 32, 128)	
conv2d_17 (Conv2D)	(None, 30, 30, 256)	
global_average_pooling2d_2 (GlobalAveragePooling2D)	(None, 256)	
dense_4 (Dense)	(None, 1024)	
dense_5 (Dense)	(None, 2)	

Total params: 838,146 (3.20 MB)

Trainable params: 838,146 (3.20 MB)

Non-trainable params: 0 (0.00 B)

In [22]:
```
# Model Compilation
model_2.compile(optimizer=tf.keras.optimizers.RMSprop(learning_rate=0.001),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
# Model Training with ModelCheckpoint Callback
checkpoint_filepath = "epoch_at_{epoch:02d}.weights.h5"
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_filepath,
    save_weights_only=True,
```

```
    monitor='val_accuracy',
    mode='max',
    save_best_only=False,
    verbose=1
)
# Model Training
history_2 = model.fit(
    train_generator_2,
    epochs=3,
    validation_data=validation_generator_2,
    callbacks=[model_checkpoint_callback]  # Add the callback here
)
```

```
Epoch 1/3
157/157 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.6260 - loss: 0.6434
Epoch 1: saving model to epoch_at_01.weights.h5
157/157 ━━━━━━━━━━━━━━━━━━━━ 955s 6s/step - accuracy: 0.6260 - loss: 0.6433 - val_
accuracy: 0.6500 - val_loss: 0.6195
Epoch 2/3
157/157 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.6504 - loss: 0.6260
Epoch 2: saving model to epoch_at_02.weights.h5
157/157 ━━━━━━━━━━━━━━━━━━━━ 954s 6s/step - accuracy: 0.6504 - loss: 0.6260 - val_
accuracy: 0.6280 - val_loss: 0.6543
Epoch 3/3
157/157 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.6640 - loss: 0.6099
Epoch 3: saving model to epoch_at_03.weights.h5
157/157 ━━━━━━━━━━━━━━━━━━━━ 949s 6s/step - accuracy: 0.6640 - loss: 0.6099 - val_
accuracy: 0.6650 - val_loss: 0.6084
```

In [23]:
```
# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(test_generator_2)
print(f"Test Loss: {test_loss}, Test Accuracy: {test_accuracy}")
```

```
16/16 ━━━━━━━━━━━━━━━━━━━━ 37s 2s/step - accuracy: 0.6926 - loss: 0.5963
Test Loss: 0.6108686327934265, Test Accuracy: 0.6869999766349792
```

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.

In [24]:
```
import os
import random

def split_data(main_dir, split_size=0.9):
    files = []
    for file in os.listdir(main_dir):
        if os.path.getsize(os.path.join(main_dir, file)):  # Check if the file's si
            files.append(file)  # Appends file name to a list

    shuffled_files = random.sample(files, len(files))  # Shuffles the data
    split = int(split_size * len(shuffled_files))  # The training split cast into i
    train_files = shuffled_files[:split]  # Training split
    split_valid_test = int(split + (len(shuffled_files) - split) / 2)

    validation_files = shuffled_files[split:split_valid_test]  # Validation split
    test_files = shuffled_files[split_valid_test:]  # Test split

    return train_files, validation_files, test_files
```

```
# Split the data for cats and dogs for version 3
train_cats_v3, val_cats_v3, test_cats_v3 = split_data(cat_file_path)
train_dogs_v3, val_dogs_v3, test_dogs_v3 = split_data(dog_file_path)

print("Version 3 - Cats - Training:", len(train_cats_v3), "Validation:", len(val_ca
print("Version 3 - Dogs - Training:", len(train_dogs_v3), "Validation:", len(val_do
```

```
Version 3 - Cats - Training: 11250 Validation: 625 Test: 625
Version 3 - Dogs - Training: 11250 Validation: 625 Test: 625
```

In [25]:

```
import pandas as pd
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Function to create a dataframe from file names and labels
def create_dataframe(file_names, label, main_dir):
    return pd.DataFrame({
        'filename': [os.path.join(main_dir, fname) for fname in file_names],
        'class': label
    })

# Create dataframes for version 3
train_df_v3 = pd.concat([
    create_dataframe(train_cats_v3, 'cat', cat_file_path),
    create_dataframe(train_dogs_v3, 'dog', dog_file_path)
])

val_df_v3 = pd.concat([
    create_dataframe(val_cats_v3, 'cat', cat_file_path),
    create_dataframe(val_dogs_v3, 'dog', dog_file_path)
])

test_df_v3 = pd.concat([
    create_dataframe(test_cats_v3, 'cat', cat_file_path),
    create_dataframe(test_dogs_v3, 'dog', dog_file_path)
])

# Shuffle the dataframes
train_df_v3 = train_df_v3.sample(frac=1).reset_index(drop=True)
val_df_v3 = val_df_v3.sample(frac=1).reset_index(drop=True)
test_df_v3 = test_df_v3.sample(frac=1).reset_index(drop=True)

# Initialize Image Data Generators with rescaling
train_gen = ImageDataGenerator(rescale=1./255)
validation_gen = ImageDataGenerator(rescale=1./255)
test_gen = ImageDataGenerator(rescale=1./255)

# Create the data generators from dataframes for version 3
train_generator_v3 = train_gen.flow_from_dataframe(
    dataframe=train_df_v3,
    x_col='filename',
    y_col='class',
    target_size=(150, 150),
    batch_size=64,
    class_mode='binary'  # or 'categorical' if you have more than two classes
)

validation_generator_v3 = validation_gen.flow_from_dataframe(
    dataframe=val_df_v3,
    x_col='filename',
    y_col='class',
    target_size=(150, 150),
    batch_size=64,
```

```
    class_mode='binary'  # or 'categorical'
)

test_generator_v3 = test_gen.flow_from_dataframe(
    dataframe=test_df_v3,
    x_col='filename',
    y_col='class',
    target_size=(150, 150),
    batch_size=64,
    class_mode='binary'  # or 'categorical'
)
```

```
Found 22498 validated image filenames belonging to 2 classes.
C:\Users\yeswa\anaconda3\Lib\site-packages\keras\src\legacy\preprocessing\image.p
y:920: UserWarning:

Found 2 invalid image filename(s) in x_col="filename". These filename(s) will be i
gnored.

Found 1250 validated image filenames belonging to 2 classes.
Found 1250 validated image filenames belonging to 2 classes.
```

In [26]:

```
import tensorflow as tf
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, GlobalAveragePooli
from tensorflow.keras.models import Model
from tensorflow.keras.callbacks import EarlyStopping

# Define the model as before, but let's make sure to use the updated optimizer
inputs = Input(shape=(150,150,3))
x = Conv2D(32, (3,3), activation='relu')(inputs)
x = Conv2D(64, (3,3), activation='relu')(x)
x = MaxPooling2D(2,2)(x)
x = Conv2D(64, (3,3), activation='relu')(x)
x = Conv2D(128, (3,3), activation='relu')(x)
x = MaxPooling2D(2,2)(x)
x = Conv2D(128, (3,3), activation='relu')(x)
x = Conv2D(256, (3,3), activation='relu')(x)
x = GlobalAveragePooling2D()(x)
x = Dense(1024, activation='relu')(x)
x = Dense(2, activation='softmax')(x)
model_3 = Model(inputs=inputs, outputs=x)
```

In [28]:

```
# Model Compilation
model_3.compile(optimizer=tf.keras.optimizers.RMSprop(learning_rate=0.001),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
# Model Training with ModelCheckpoint Callback
checkpoint_filepath = "epoch_at_{epoch:02d}.weights.h5"
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_filepath,
    save_weights_only=True,
    monitor='val_accuracy',
    mode='max',
    save_best_only=False,
    verbose=1
)
# Model Training
history_3 = model.fit(
    train_generator_v3,
    epochs=3,
    validation_data=validation_generator_v3,
    callbacks=[model_checkpoint_callback]  # Add the callback here
)
```

```
Epoch 1/3
175/352 ━━━━━━━━━━━━━━━━━━━━ 17:20 6s/step - accuracy: 0.6858 - loss: 0.5940
C:\Users\yeswa\anaconda3\Lib\site-packages\PIL\TiffImagePlugin.py:858: UserWarnin
g:

Truncated File Read

352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.6855 - loss: 0.5919
Epoch 1: saving model to epoch_at_01.weights.h5
352/352 ━━━━━━━━━━━━━━━━━━━━ 2128s 6s/step - accuracy: 0.6855 - loss: 0.5919 - val
_accuracy: 0.6944 - val_loss: 0.5786
Epoch 2/3
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.7045 - loss: 0.5770
Epoch 2: saving model to epoch_at_02.weights.h5
352/352 ━━━━━━━━━━━━━━━━━━━━ 2117s 6s/step - accuracy: 0.7045 - loss: 0.5769 - val
_accuracy: 0.7048 - val_loss: 0.5719
Epoch 3/3
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.7276 - loss: 0.5482
Epoch 3: saving model to epoch_at_03.weights.h5
352/352 ━━━━━━━━━━━━━━━━━━━━ 2099s 6s/step - accuracy: 0.7276 - loss: 0.5482 - val
_accuracy: 0.7272 - val_loss: 0.5425
```

In [29]:

```
# Model Evaluation
model.evaluate(test_generator_v3)
```

```
20/20 ━━━━━━━━━━━━━━━━━━━━ 40s 2s/step - accuracy: 0.6998 - loss: 0.5676
```

Out[29]:

```
[0.5526072978973389, 0.7200000286102295]
```

In [31]:

```
# Last Training Results
results = pd.DataFrame(history_3.history)
results.tail()
```

Out[31]:

	accuracy	loss	val_accuracy	val_loss
0	0.687483	0.587665	0.6944	0.578557
1	0.709130	0.567228	0.7048	0.571875
2	0.726954	0.547017	0.7272	0.542539

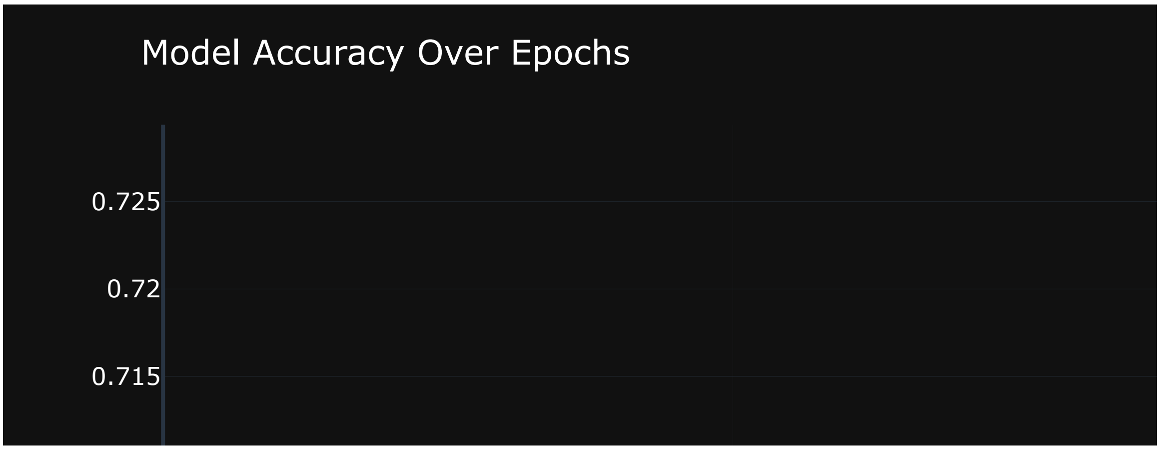
In [32]:

```
fig = px.line(results, y=['accuracy', 'val_accuracy'],
              labels={'value': 'Accuracy', 'variable': 'Dataset'},
              template="plotly_dark")

fig.update_layout(
    title='Model Accuracy Over Epochs',
    title_font_color='white',
    xaxis=dict(color='white', title='Epochs'),
    yaxis=dict(color='white', title='Accuracy'),
    legend_title_text='Dataset'
)

fig.update_traces(marker=dict(size=8),
                  selector=dict(type='scatter', mode='lines+markers'))

fig.show()
```

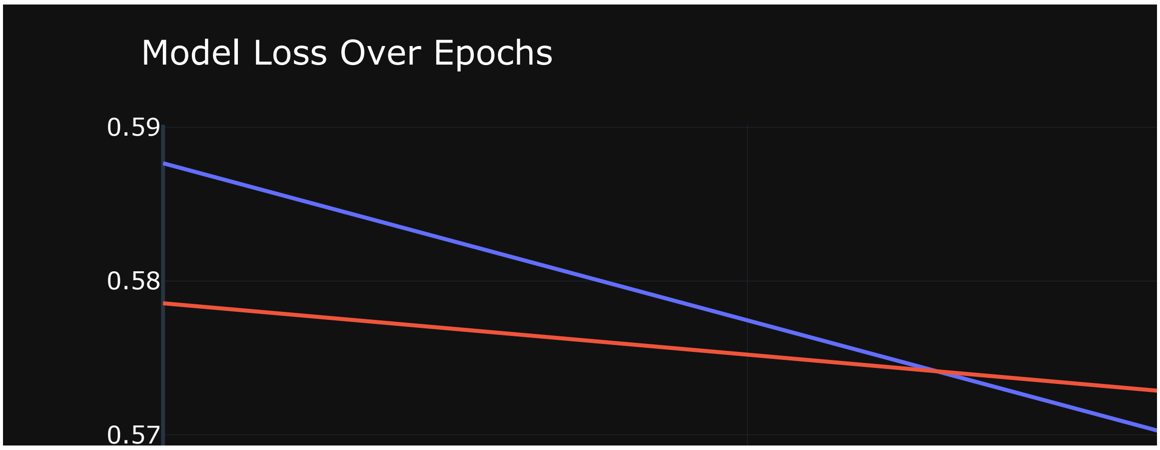


In [33]:
```
fig = px.line(results, y=['loss', 'val_loss'],
              labels={'value': 'Loss', 'variable': 'Dataset'},
              template="plotly_dark")

fig.update_layout(
    title='Model Loss Over Epochs',
    title_font_color='white',
    xaxis=dict(color='white', title='Epochs'),
    yaxis=dict(color='white', title='Loss'),
    legend_title_text='Dataset'
)

fig.update_traces(marker=dict(size=8),
                  selector=dict(type='scatter', mode='lines+markers'))

fig.show()
```



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.

In [34]:
```
import tensorflow as tf
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.layers import GlobalAveragePooling2D, Dense
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import RMSprop
```

In [35]:
```
# Load MobileNetV2 pre-trained on ImageNet data
base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape=(150, 1

# Freeze the base_model
base_model.trainable = False

# Create a new model on top
inputs = tf.keras.Input(shape=(150, 150, 3))
x = base_model(inputs, training=False)
x = GlobalAveragePooling2D()(x)
x = Dense(1024, activation='relu')(x)  # You can adjust the number of units
outputs = Dense(1, activation='sigmoid')(x)  # Binary output for cat vs. dog
model_pretrained = Model(inputs, outputs)
```

```
# Model Compilation
model_pretrained.compile(optimizer=RMSprop(learning_rate=0.001),
                         loss='binary_crossentropy',
                         metrics=['accuracy'])
```

```
C:\Users\yeswa\AppData\Local\Temp\ipykernel_11792\1507043546.py:2: UserWarning:

`input_shape` is undefined or non-square, or `rows` is not in [96, 128, 160, 192,
224]. Weights for input shape (224, 224) will be loaded as the default.
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-application
s/mobilenet_v2/mobilenet_v2_weights_tf_dim_ordering_tf_kernels_1.0_224_no_top.h5
9406464/9406464 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
```

In [37]:

```
# Model Training with ModelCheckpoint Callback
checkpoint_filepath = "epoch_at_{epoch:02d}.weights.h5"
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_filepath,
    save_weights_only=True,
    monitor='val_accuracy',
    mode='max',
    save_best_only=False,
    verbose=1
)
# Model Training
history_3 = model.fit(
    train_generator,
    epochs=10,
    validation_data=validation_generator,
    callbacks=[model_checkpoint_callback]  # Add the callback here
)
```

```
Epoch 1/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.7240 - loss: 0.5292
Epoch 1: saving model to epoch_at_01.weights.h5
32/32 ━━━━━━━━━━━━━━━━━━━━ 218s 7s/step - accuracy: 0.7241 - loss: 0.5295 - val_ac
curacy: 0.7650 - val_loss: 0.5197
Epoch 2/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.7337 - loss: 0.5322
Epoch 2: saving model to epoch_at_02.weights.h5
32/32 ━━━━━━━━━━━━━━━━━━━━ 219s 7s/step - accuracy: 0.7337 - loss: 0.5324 - val_ac
curacy: 0.7560 - val_loss: 0.5053
Epoch 3/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.7528 - loss: 0.5099
Epoch 3: saving model to epoch_at_03.weights.h5
32/32 ━━━━━━━━━━━━━━━━━━━━ 218s 7s/step - accuracy: 0.7527 - loss: 0.5101 - val_ac
curacy: 0.7590 - val_loss: 0.5194
Epoch 4/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.7282 - loss: 0.5340
Epoch 4: saving model to epoch_at_04.weights.h5
32/32 ━━━━━━━━━━━━━━━━━━━━ 219s 7s/step - accuracy: 0.7283 - loss: 0.5344 - val_ac
curacy: 0.7580 - val_loss: 0.5150
Epoch 5/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.7724 - loss: 0.4994
Epoch 5: saving model to epoch_at_05.weights.h5
32/32 ━━━━━━━━━━━━━━━━━━━━ 220s 7s/step - accuracy: 0.7721 - loss: 0.4997 - val_ac
curacy: 0.7060 - val_loss: 0.5560
Epoch 6/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.7467 - loss: 0.5115
Epoch 6: saving model to epoch_at_06.weights.h5
32/32 ━━━━━━━━━━━━━━━━━━━━ 218s 7s/step - accuracy: 0.7465 - loss: 0.5117 - val_ac
curacy: 0.7540 - val_loss: 0.4949
Epoch 7/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.7684 - loss: 0.4927
Epoch 7: saving model to epoch_at_07.weights.h5
32/32 ━━━━━━━━━━━━━━━━━━━━ 211s 7s/step - accuracy: 0.7681 - loss: 0.4931 - val_ac
curacy: 0.7680 - val_loss: 0.4908
Epoch 8/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.7643 - loss: 0.4938
Epoch 8: saving model to epoch_at_08.weights.h5
32/32 ━━━━━━━━━━━━━━━━━━━━ 211s 7s/step - accuracy: 0.7640 - loss: 0.4944 - val_ac
curacy: 0.7710 - val_loss: 0.4931
Epoch 9/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.7448 - loss: 0.5273
Epoch 9: saving model to epoch_at_09.weights.h5
32/32 ━━━━━━━━━━━━━━━━━━━━ 210s 6s/step - accuracy: 0.7450 - loss: 0.5268 - val_ac
curacy: 0.7500 - val_loss: 0.5155
Epoch 10/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6s/step - accuracy: 0.7699 - loss: 0.4985
Epoch 10: saving model to epoch_at_10.weights.h5
32/32 ━━━━━━━━━━━━━━━━━━━━ 209s 6s/step - accuracy: 0.7697 - loss: 0.4989 - val_ac
curacy: 0.7500 - val_loss: 0.4939
```

In [39]:

```python
# Optional: Evaluate the model on the test set
test_loss, test_accuracy = model_pretrained.evaluate(test_generator)
print(f'Test Loss: {test_loss}, Test Accuracy: {test_accuracy}')
```

```
16/16 ━━━━━━━━━━━━━━━━━━━━ 14s 830ms/step - accuracy: 0.5006 - loss: 1.1733
Test Loss: 1.154368281364441, Test Accuracy: 0.5009999871253967
```

In [40]:

```python
# Last Training Results
results = pd.DataFrame(history_3.history)
results.tail()
```

Out[40]:

	accuracy	loss	val_accuracy	val_loss
5	0.7405	0.516746	0.754	0.494869
6	0.7585	0.505731	0.768	0.490792
7	0.7530	0.513503	0.771	0.493070
8	0.7525	0.512518	0.750	0.515486
9	0.7625	0.510479	0.750	0.493852

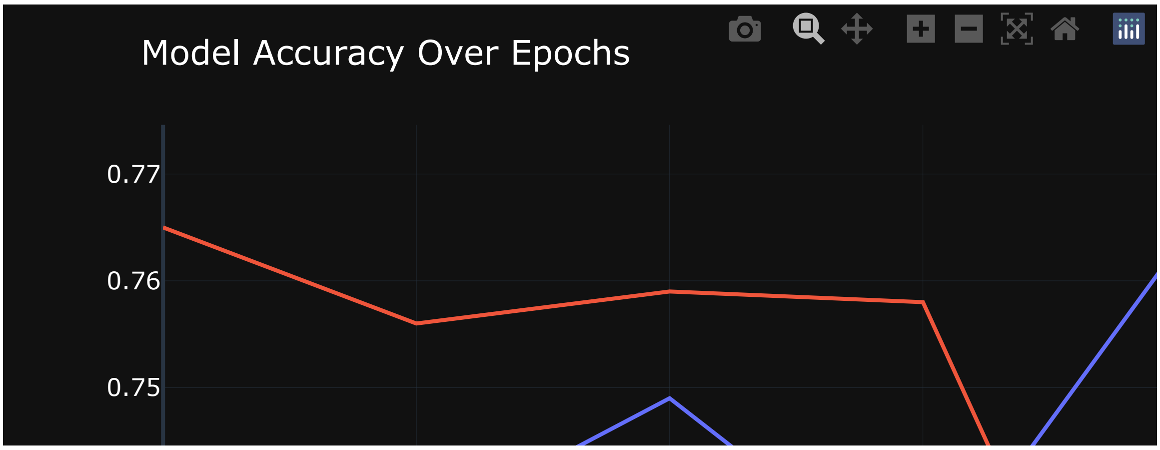
In [41]:

```
fig = px.line(results, y=['accuracy', 'val_accuracy'],
              labels={'value': 'Accuracy', 'variable': 'Dataset'},
              template="plotly_dark")

fig.update_layout(
    title='Model Accuracy Over Epochs',
    title_font_color='white',
    xaxis=dict(color='white', title='Epochs'),
    yaxis=dict(color='white', title='Accuracy'),
    legend_title_text='Dataset'
)

fig.update_traces(marker=dict(size=8),
                  selector=dict(type='scatter', mode='lines+markers'))

fig.show()
```



In [42]:
```
fig = px.line(results, y=['loss', 'val_loss'],
              labels={'value': 'Loss', 'variable': 'Dataset'},
              template="plotly_dark")

fig.update_layout(
    title='Model Loss Over Epochs',
    title_font_color='white',
    xaxis=dict(color='white', title='Epochs'),
    yaxis=dict(color='white', title='Loss'),
    legend_title_text='Dataset'
)

fig.update_traces(marker=dict(size=8),
                  selector=dict(type='scatter', mode='lines+markers'))

fig.show()
```

SUMMARY

Training Convolutional Neural Networks (CNNs) with larger datasets is a fundamental strategy to improve the model's ability to learn and generalize from data. This approach is based on the principle that the more examples a model is exposed to, the better it can understand the underlying patterns and make accurate predictions. However, simply increasing the dataset size is not a silver bullet; it introduces several challenges and considerations that must be addressed to harness the full potential of larger datasets effectively. Here’s an elaboration on why larger datasets matter and the importance of optimization techniques in this context:

Importance of Larger Datasets

Enhanced Learning and Generalization: Larger datasets provide a more comprehensive representation of the real world, encompassing a wider variety of examples. This diversity helps CNN models to learn more nuanced and complex patterns, which is crucial for tasks such as image recognition, where subtle differences can determine the correct classification.

Reduction in Overfitting: Overfitting occurs when a model learns the noise or random fluctuations in the training data instead of the actual underlying patterns. By using larger datasets, the risk of overfitting is reduced because the model is less likely to focus on the noise, given the abundance of legitimate data points illustrating the true patterns.

Improved Robustness: Models trained on larger and more diverse datasets tend to be more robust to variations and changes in input data. This is particularly important in real-world applications where the data may differ significantly from what the model was trained on.

Challenges with Larger Datasets

While larger datasets offer significant benefits, they also present challenges that require careful consideration:

Computational Resources: Training on larger datasets demands more computational power and memory. This can increase the time and cost of model training, making it less feasible for individuals or organizations with limited resources.

Diminishing Returns: As datasets grow, the incremental improvement in model performance can diminish. At some point, simply adding more data does not yield significant improvements, especially if the additional data is not diverse or is of low quality.

Data Quality and Diversity: The benefits of larger datasets are only realized if the data is both high quality and diverse. Poor quality data can lead to misleading patterns being learned, while a lack of diversity can result in a model that performs well on the training data but poorly on unseen data.

Importance of Optimization Techniques

To address these challenges and effectively utilize larger datasets, several optimization techniques are essential:

Efficient Data Loading and Preprocessing: Techniques such as data prefetching and parallel processing can significantly reduce the time required to load and preprocess data, making training on large datasets more feasible.

Advanced Training Algorithms: Optimization algorithms like Stochastic Gradient Descent (SGD) with momentum, Adam, and RMSprop can help in faster convergence of the training process, even with vast amounts of data. These algorithms adjust the learning rate dynamically, making the training process more efficient.

Regularization Techniques: Methods like dropout, weight decay, and data augmentation are crucial for preventing overfitting, especially when training with large datasets. These techniques help the model to generalize better to unseen data by making it less sensitive to the specific details of the training data.

Distributed and Parallel Computing: Utilizing multiple GPUs or cloud-based computing resources can significantly speed up the training process. Distributed training splits the dataset across multiple machines, allowing for parallel processing and more efficient handling of large datasets.

In summary, while larger datasets can significantly improve CNN model performance by providing more examples and increasing data diversity, leveraging these benefits requires strategic optimization techniques. These include efficient data management, advanced training algorithms, regularization to prevent overfitting, and the use of distributed computing to handle the increased computational load. By combining these elements, it's possible to efficiently train robust and high-performing CNN models capable of handling the complexities of real-world data.

In [ ]: