Machine Learning
LABORATORY: CNN Homework

NAME:	STUDENT ID#:

Objectives:

- In this assignment, you will train a Convolutional Neural Network (CNN) to classify images of cats and dogs using a dataset and starter template provided to you.
- The template code includes a simple CNN model that serves as a baseline. However, this baseline model does not achieve high accuracy.
- Your goal is to improve the model's performance by modifying and fine-tuning the CNN architecture.
- You are encouraged to: Adjust the layer, add data augmentation and apply regularization techniques such as Dropout, BatchNormalization, or custom pooling strategies.

Part 1. Instruction

- The template code includes a simple CNN model that serves as a baseline. However, this baseline model does not achieve high accuracy.
- Your task is to improve the model's performance by modifying and fine-tuning the CNN architecture.
- You are encouraged to:
    - Adjust layer configurations (number of layers, filters, kernel sizes, activations, etc.)
    - Add data augmentation to improve generalization
    - Apply regularization techniques such as Dropout, BatchNormalization, or custom pooling strategies
- **However**, you are not allowed to use pre-trained models such as VGG, ResNet, EfficientNet, etc. (No transfer learning or model imports from keras.applications).
- Your model should be designed, trained, and fine-tuned by yourself, from scratch using Keras/TensorFlow layers.
- If you're not familiar with TensorFlow or Keras, check these links:
    - Tensorflow Setup Guide https://github.com/JTKostman/keras-tensorflow-windows-installation
    - Keras Quick Start : https://keras.io/getting_started/
- If you're using your **local PC**, you may need to install TensorFlow and Keras manually. If you're using **Google Colab**, they're already installed — but be aware that Colab has **limited training time**.
- Dataset: https://drive.google.com/file/d/1eyrNGFlM83pf-TETo30Cvjm7kYuCFAqu/view?usp=sharing (Also included in the template code for easy loading)

Lecture: Prof. Hsien-I Lin
TA: Satrio Sanjaya and Muhammad Ahsan

Part 2. Code Template

Step	Procedure
1	

```
import pandas as pd
import IPython.display as display
import tensorflow as tf
from tensorflow.keras import layers
import numpy as np
import os,random
import matplotlib.pyplot as plt
print(tf.__version__)
```

2	

```
# ========== Download Dataset ==========
#!pip install -U gdown

import gdown, zipfile

file_id = "1eyrNGFlM83pf-TETo30Cvjm7kYuCFAqu"  # <-- from your real zip file link
gdown.download(f"https://drive.google.com/uc?id={file_id}", output="dataset.zip", quiet=False)

with zipfile.ZipFile("dataset.zip", 'r') as zip_ref:
   zip_ref.extractall("dataset")
```

3	

```
# ==========Dataset Preparation ==========
IMAGE_HEIGHT=128
IMAGE_WIDTH=128
BATCH_SIZE=64
def get_pathframe(path):
 '''
 Get all the images paths and its corresponding labels
 Store them in pandas dataframe
 '''
 filenames = os.listdir(path)
 categories = []
 paths=[]
 for filename in filenames:
   paths.append(path+filename)
   category = filename.split('.')[0]
   if category == 'dog':
     categories.append(1)
   else:
     categories.append(0)

 df= pd.DataFrame({
```

```
        'filename': filenames,
        'category': categories,
        'paths':paths
    })
    return df

df=get_pathframe("dataset/dataset/dataset/")
df.tail(5)
```

4

```
# ========== Convert to tensor ==========
def load_and_preprocess_image(path):
    '''
    Load each image and resize it to desired shape
    '''
    image = tf.io.read_file(path)
    image = tf.image.decode_jpeg(image, channels=3)
    image = tf.image.resize(image, [IMAGE_WIDTH, IMAGE_HEIGHT])
    image /= 255.0  # normalize to [0,1] range
    return image

def convert_to_tensor(df):
    '''
    Convert each data and labels to tensor
    '''
    path_ds = tf.data.Dataset.from_tensor_slices(df['paths'])
    image_ds = path_ds.map(load_and_preprocess_image)
    # onehot_label=tf.one_hot(tf.cast(df['category'], tf.int64),2) if
using softmax
    onehot_label=tf.cast(df['category'], tf.int64)
    label_ds = tf.data.Dataset.from_tensor_slices(onehot_label)

    return image_ds,label_ds


X,Y=convert_to_tensor(df)
print("Shape of X in data:", X)
print("Shape of Y in data:", Y)
```

5

```
#Plot Images
dataset=tf.data.Dataset.zip((X,Y)).shuffle(buffer_size=2000)
dataset_train=dataset.take(22500)
dataset_test=dataset.skip(22500)
```


```
dataset_train=dataset_train.batch(BATCH_SIZE, drop_remainder=True)
dataset_test=dataset_test.batch(BATCH_SIZE, drop_remainder=True)
dataset_train

def plotimages(imagesls):
 fig, axes = plt.subplots(1, 5, figsize=(20,20))
 axes = axes.flatten()
 for image,ax in zip(imagesls, axes):
   ax.imshow(image)
   ax.axis('off')

imagesls=[]
for n, image in enumerate(X.take(5)):
 imagesls.append(image)

plotimages(imagesls)
```

6

```
#Model Design
def My_CNNmodel():

 model = tf.keras.models.Sequential()
 model.add(layers.Conv2D(8, (3, 3), padding='same',activation='relu',
input_shape=(IMAGE_WIDTH, IMAGE_HEIGHT, 3)))
 model.add(layers.MaxPooling2D(pool_size=(2,2)))
 model.add(layers.Conv2D(16, (3, 3),
padding='same',activation='relu'))
 model.add(layers.MaxPooling2D(pool_size=(2,2)))
 model.add(layers.Conv2D(32, (3, 3),
padding='same',activation='relu'))
 model.add(layers.MaxPooling2D(pool_size=(2,2)))
 model.add(layers.Conv2D(64, (3, 3),
padding='same',activation='relu'))
 model.add(layers.MaxPooling2D(pool_size=(2,2)))
 model.add(layers.Flatten())
 model.add(layers.Dense(512, activation='relu'))
 model.add(layers.Dense(1, activation='sigmoid'))

 opt=tf.keras.optimizers.Adam(0.001)
 model.compile(optimizer=opt,
             loss='binary_crossentropy', #
loss='categorical_crossentropy' if softmax
             metrics=['accuracy'])
```


```
 return model

model=My_CNNmodel()
model.summary()
```

7

```
#Training the model
#You can adjust the epochs to get better training results, be aware
with overfitting

hist=model.fit(dataset_train,epochs=2,validation_data=dataset_test)
#Save trained model
model.save("student_ID.keras")
#Save the model with your student ID
```

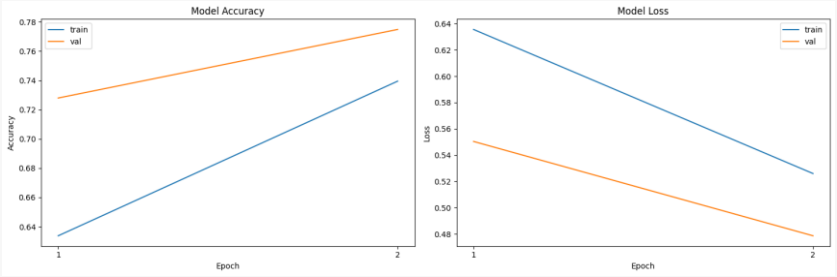
```
#Plot training results
def plot_model_history(model_history, acc='accuracy',
val_acc='val_accuracy'):
   fig, axs = plt.subplots(1, 2, figsize=(15, 5))

   # Accuracy plot
   axs[0].plot(range(1, len(model_history.history[acc]) + 1),
model_history.history[acc])
   axs[0].plot(range(1, len(model_history.history[val_acc]) + 1),
model_history.history[val_acc])
   axs[0].set_title('Model Accuracy')
   axs[0].set_ylabel('Accuracy')
   axs[0].set_xlabel('Epoch')
   axs[0].set_xticks(np.arange(1, len(model_history.history[acc]) +
1))
   axs[0].legend(['train', 'val'], loc='best')

   # Loss plot
   axs[1].plot(range(1, len(model_history.history['loss']) + 1),
model_history.history['loss'])
   axs[1].plot(range(1, len(model_history.history['val_loss']) + 1),
model_history.history['val_loss'])
   axs[1].set_title('Model Loss')
   axs[1].set_ylabel('Loss')
   axs[1].set_xlabel('Epoch')
   axs[1].set_xticks(np.arange(1, len(model_history.history['loss'])
+ 1))
   axs[1].legend(['train', 'val'], loc='best')
```


```
    plt.tight_layout()
    plt.show()
plot_model_history(hist)
```

```
#Evaluate the model
import seaborn as sns
from sklearn.metrics import confusion_matrix, classification_report
import matplotlib.pyplot as plt
import numpy as np

# Evaluate the model
loss, accuracy = model.evaluate(dataset_test)
print("Test: accuracy = %f  ;  loss = %f " % (accuracy, loss))

# Predict values
y_pred = model.predict(dataset_test)
y_p = np.where(y_pred > 0.5, 1, 0)  # for binary classification

# Extract ground truth labels
test_data = dataset_test.unbatch()
y_g = []
for image, label in test_data:
    y_g.append(label.numpy())

# Convert to flat array if needed
y_g = np.array(y_g).flatten()
y_p = y_p.flatten()

# Compute confusion matrix
confusion_mtx = confusion_matrix(y_g, y_p)

# Plot
f, ax = plt.subplots(figsize=(8, 8))
sns.heatmap(confusion_mtx, annot=True, linewidths=0.01, cmap="Blues",
linecolor="gray", fmt='.1f', ax=ax)
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix")
plt.show()
```

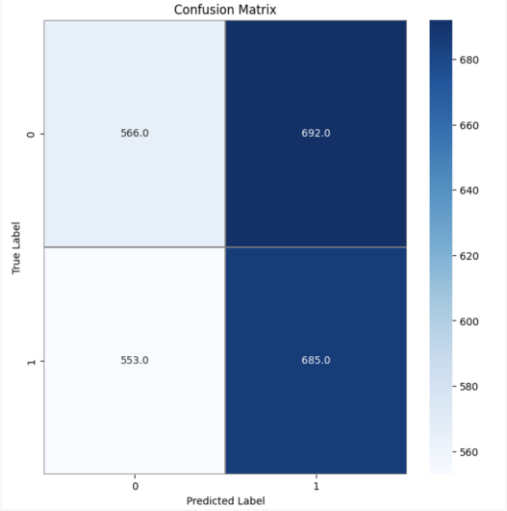
```
# Generate a classification report
report = classification_report(y_g, y_p, target_names=['0','1'])
print(report)
```


```
#Example Output (For references only)
#1. Plot Training Results
```




```
#2. Confusion Matrix
```

```
39/39 ━━━━━━━━━━━━━━━━━━━━ 19s 22ms/step - accuracy: 0.7777 - loss: 0.4819
Test: accuracy = 0.770433  ;  loss = 0.482054
39/39 ━━━━━━━━━━━━━━━━━━━━ 19s 17ms/step
```




```
#3. Classification Report
```

	precision	recall	f1-score	support
0	0.51	0.45	0.48	1258
1	0.50	0.55	0.52	1238
accuracy			0.50	2496
macro avg	0.50	0.50	0.50	2496
weighted avg	0.50	0.50	0.50	2496

```
#4. Model Summary
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 128, 128, 8)	224
max_pooling2d (MaxPooling2D)	(None, 64, 64, 8)	0
conv2d_1 (Conv2D)	(None, 64, 64, 16)	1,168
max_pooling2d_1 (MaxPooling2D)	(None, 32, 32, 16)	0
conv2d_2 (Conv2D)	(None, 32, 32, 32)	4,640
max_pooling2d_2 (MaxPooling2D)	(None, 16, 16, 32)	0
conv2d_3 (Conv2D)	(None, 16, 16, 64)	18,496
max_pooling2d_3 (MaxPooling2D)	(None, 8, 8, 64)	0
flatten (Flatten)	(None, 4096)	0
dense (Dense)	(None, 512)	2,097,664
dense_1 (Dense)	(None, 1)	513

Total params: 2,122,705 (8.10 MB)
Trainable params: 2,122,705 (8.10 MB)
Non-trainable params: 0 (0.00 B)

Grading Assignment & Submission (70%)

Implementation(45%):

1. (5%) **Fine-Tuning:** Adjust model parameters to achieve better accuracy
2. (20%) **Model redesign with techniques:** Redesign the architecture using advanced techniques such as data augmentation, Dropout, BatchNormalization, or self-designed structures (should achieve better accuracy), brief **explain your model summary in the report.**
3. (15%) **Accuracy Improvement:** Demonstrate significantly improved validation accuracy compared to the baseline model. **Explain your performance improvement.**
4. (5%) **Include all evaluation results:** model summary, training plots, confusion matrix, and classification report.

Question(25%):

5. (7%) What changes did you make to the CNN architecture? Why did you choose those modifications, and how did they affect model performance?
6. (8%) What methods did you use to reduce overfitting (if any)? How effective were they? Explain with reference to training/validation curves.
7. (10%) Based on your confusion matrix, what kinds of misclassifications occurred most frequently? What might be causing these errors? How would you attempt to reduce these errors?

Submission :

1. Report: Provide your screenshots of your results in the last pages of this PDF File.
2. Code: Submit your complete Python script in either .py or .ipynb format.
3. Upload both your report and code to the E3 system (**Labs6 Homework Assignment**). Name your files correctly:
    a. Report: StudentID_Lab6_Homework.pdf
    b. Code: StudentID_Lab6_ Homework.py or StudentID_Lab6_Homework.ipynb
4. Deadline: 16:20 PM
5. Plagiarism is **strictly prohibited**. Submitting copied work from other students will result in penalties.

Results and Discussion:

2.18.0

Downloading...

From (original): https://drive.google.com/uc?id=1eyrNGFlM83pf-TETo30Cvjm7kYuCFAqu

From (redirected): https://drive.google.com/uc?id=1eyrNGFlM83pf-TETo30Cvjm7kYuCFAqu&confirm=t&uuid=d3026d6e-0e19-4e8e-9c2c-f9fe39bd8374

To: /content/dataset.zip

100%|██████████| 1.16G/1.16G [00:16<00:00, 71.1MB/s]

	filename	category	paths
24995	cat.3632.jpg	0	dataset/dataset/dataset/cat.3632.jpg
24996	cat.2256.jpg	0	dataset/dataset/dataset/cat.2256.jpg
24997	cat.10598.jpg	0	dataset/dataset/dataset/cat.10598.jpg
24998	cat.8822.jpg	0	dataset/dataset/dataset/cat.8822.jpg
24999	dog.7297.jpg	1	dataset/dataset/dataset/dog.7297.jpg

Shape of X in data: <_MapDataset element_spec=TensorSpec(shape=(150, 150, 3), dtype=tf.float32, name=None)>

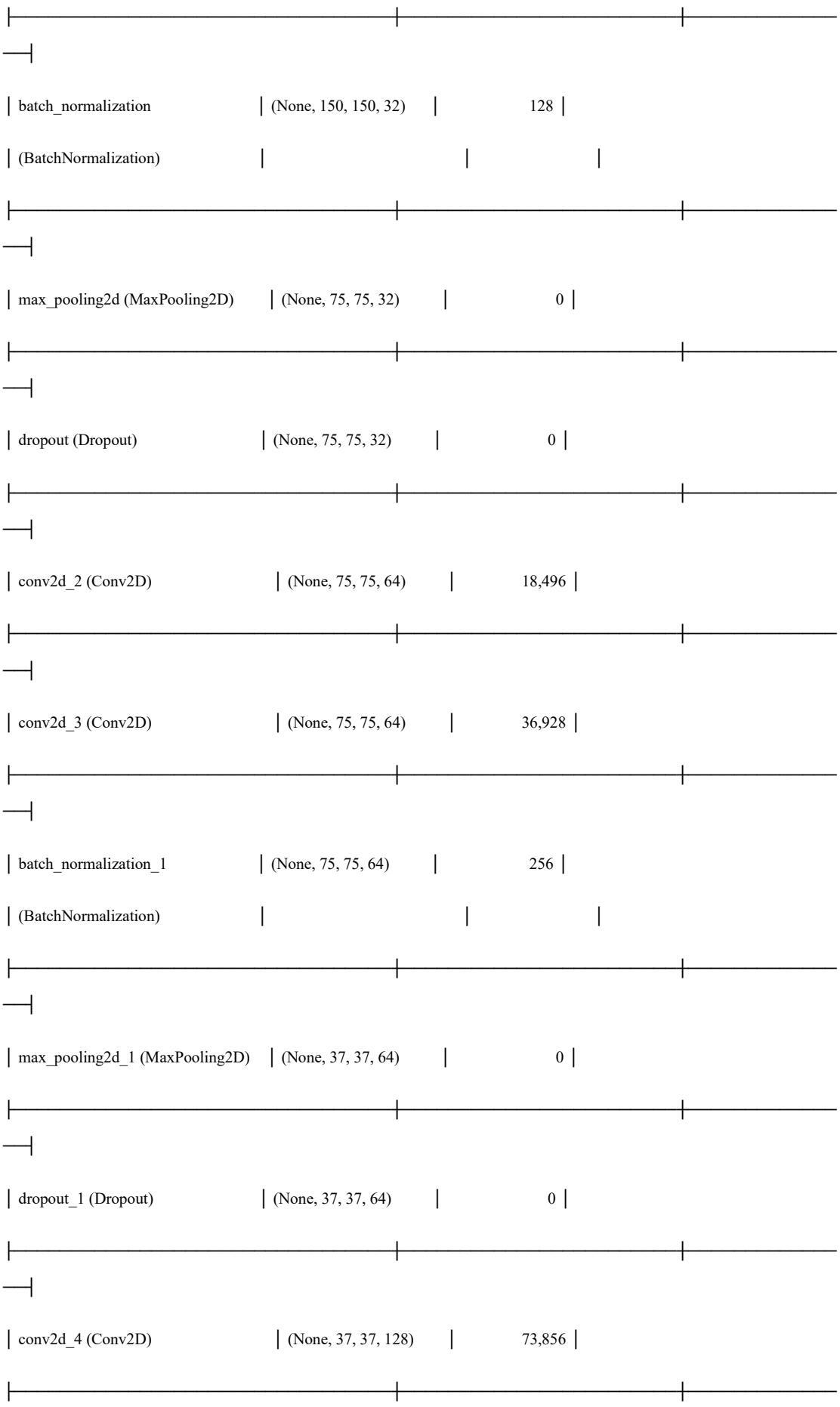
Shape of Y in data: <_TensorSliceDataset element_spec=TensorSpec(shape=(), dtype=tf.int64, name=None)>

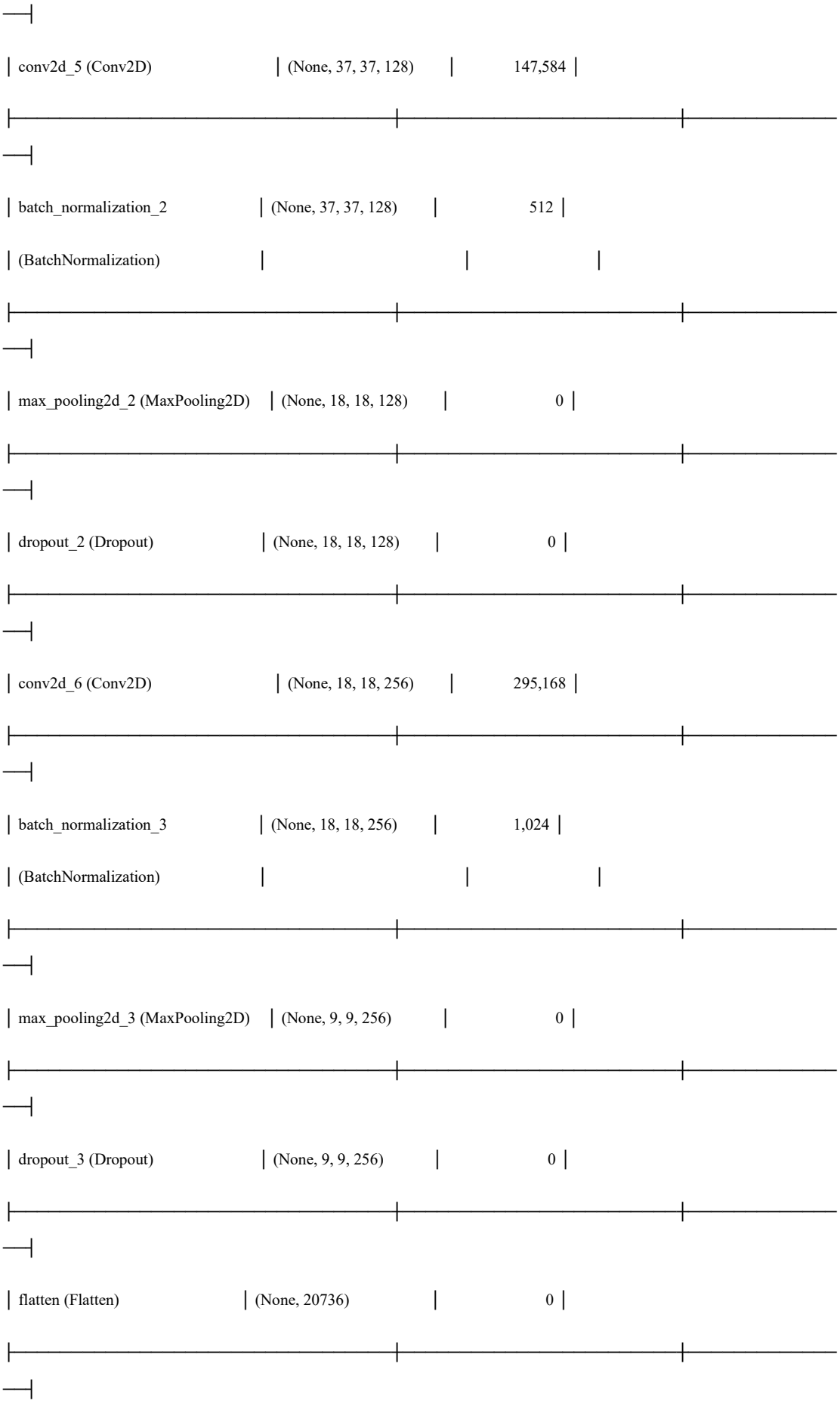
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

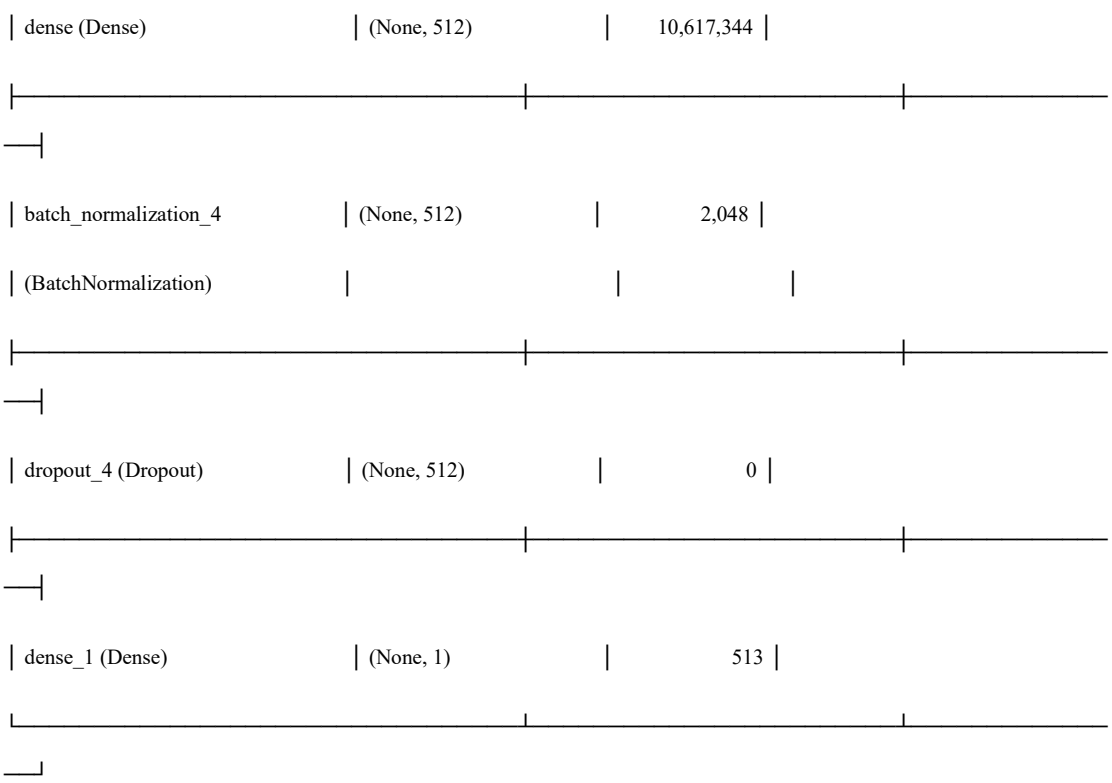
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 150, 150, 32)	896
conv2d_1 (Conv2D)	(None, 150, 150, 32)	9,248


batch_normalization (BatchNormalization) (None, 150, 150, 32) 128
max_pooling2d (MaxPooling2D) (None, 75, 75, 32) 0
dropout (Dropout) (None, 75, 75, 32) 0
conv2d_2 (Conv2D) (None, 75, 75, 64) 18,496
conv2d_3 (Conv2D) (None, 75, 75, 64) 36,928
batch_normalization_1 (BatchNormalization) (None, 75, 75, 64) 256
max_pooling2d_1 (MaxPooling2D) (None, 37, 37, 64) 0
dropout_1 (Dropout) (None, 37, 37, 64) 0
conv2d_4 (Conv2D) (None, 37, 37, 128) 73,856


conv2d_5 (Conv2D) (None, 37, 37, 128) 147,584
batch_normalization_2 (BatchNormalization) (None, 37, 37, 128) 512
max_pooling2d_2 (MaxPooling2D) (None, 18, 18, 128) 0
dropout_2 (Dropout) (None, 18, 18, 128) 0
conv2d_6 (Conv2D) (None, 18, 18, 256) 295,168
batch_normalization_3 (BatchNormalization) (None, 18, 18, 256) 1,024
max_pooling2d_3 (MaxPooling2D) (None, 9, 9, 256) 0
dropout_3 (Dropout) (None, 9, 9, 256) 0
flatten (Flatten) (None, 20736) 0



Total params: 11,204,001 (42.74 MB)

Trainable params: 11,202,017 (42.73 MB)

Non-trainable params: 1,984 (7.75 KB)

Epoch 1/50

703/703 ━━━━━━━━━━━━━━━━━━━━ **94s** 103ms/step - accuracy: 0.5956 - loss: 0.8644 - val_accuracy: 0.5809 - val_loss: 4.9273 - learning_rate: 1.0000e-04

Epoch 2/50

703/703 ━━━━━━━━━━━━━━━━━━━━ **127s** 101ms/step - accuracy: 0.6765 - loss: 0.6589 - val_accuracy: 0.6558 - val_loss: 1.7053 - learning_rate: 1.0000e-04

Epoch 3/50

703/703 ━━━━━━━━━━━━━━━━━━━━ **82s** 101ms/step - accuracy: 0.7180 - loss: 0.5813 - val_accuracy: 0.7536 - val_loss: 0.6462 - learning_rate: 1.0000e-04

Epoch 4/50

703/703 ━━━━━━━━━━━━━━━━━━━━ **83s** 101ms/step - accuracy: 0.7547 - loss: 0.5131 - val_accuracy: 0.7540 - val_loss: 0.6056 - learning_rate: 1.0000e-04

Epoch 5/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **82s** 103ms/step - accuracy: 0.7841 - loss: 0.4636 - val_accuracy: 0.8257 - val_loss: 0.3817 - learning_rate: 1.0000e-04

Epoch 6/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **76s** 102ms/step - accuracy: 0.8075 - loss: 0.4214 - val_accuracy: 0.8073 - val_loss: 0.4680 - learning_rate: 1.0000e-04

Epoch 7/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **77s** 102ms/step - accuracy: 0.8283 - loss: 0.3887 - val_accuracy: 0.8670 - val_loss: 0.3048 - learning_rate: 1.0000e-04

Epoch 8/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **81s** 103ms/step - accuracy: 0.8406 - loss: 0.3704 - val_accuracy: 0.8722 - val_loss: 0.3169 - learning_rate: 1.0000e-04

Epoch 9/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **76s** 102ms/step - accuracy: 0.8564 - loss: 0.3353 - val_accuracy: 0.8726 - val_loss: 0.3259 - learning_rate: 1.0000e-04

Epoch 10/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **81s** 102ms/step - accuracy: 0.8685 - loss: 0.3133 - val_accuracy: 0.8333 - val_loss: 0.3980 - learning_rate: 1.0000e-04

Epoch 11/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **103s** 130ms/step - accuracy: 0.8732 - loss: 0.2865 - val_accuracy: 0.8854 - val_loss: 0.2901 - learning_rate: 1.0000e-04

Epoch 12/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **76s** 102ms/step - accuracy: 0.8922 - loss: 0.2616 - val_accuracy: 0.9383 - val_loss: 0.1559 - learning_rate: 1.0000e-04

Epoch 13/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **102s** 130ms/step - accuracy: 0.8993 - loss: 0.2462 - val_accuracy: 0.9239 - val_loss: 0.1849 - learning_rate: 1.0000e-04

Epoch 14/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **121s** 100ms/step - accuracy: 0.9024 - loss: 0.2310 - val_accuracy: 0.9219 - val_loss: 0.1967 - learning_rate: 1.0000e-04

Epoch 15/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **75s** 100ms/step - accuracy: 0.9089 - loss: 0.2229 - val_accuracy: 0.9411 - val_loss: 0.1601 - learning_rate: 1.0000e-04

Epoch 16/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **76s** 103ms/step - accuracy: 0.9202 - loss: 0.1971 - val_accuracy: 0.9319 - val_loss: 0.1678 - learning_rate: 1.0000e-04

Epoch 17/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **81s** 100ms/step - accuracy: 0.9213 - loss: 0.1910 - val_accuracy: 0.8818 - val_loss: 0.3621 - learning_rate: 1.0000e-04

Epoch 18/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **81s** 101ms/step - accuracy: 0.9261 - loss: 0.1793 - val_accuracy: 0.9555 - val_loss: 0.1123 - learning_rate: 2.0000e-05

Epoch 19/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **75s** 102ms/step - accuracy: 0.9387 - loss: 0.1505 - val_accuracy: 0.9551 - val_loss: 0.1227 - learning_rate: 2.0000e-05

Epoch 20/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **83s** 103ms/step - accuracy: 0.9435 - loss: 0.1454 - val_accuracy: 0.9679 - val_loss: 0.0924 - learning_rate: 2.0000e-05

Epoch 21/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **76s** 102ms/step - accuracy: 0.9433 - loss: 0.1471 - val_accuracy: 0.9647 - val_loss: 0.0918 - learning_rate: 2.0000e-05

Epoch 22/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **82s** 103ms/step - accuracy: 0.9457 - loss: 0.1390 - val_accuracy: 0.9732 - val_loss: 0.0921 - learning_rate: 2.0000e-05

Epoch 23/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **80s** 109ms/step - accuracy: 0.9476 - loss: 0.1371 - val_accuracy: 0.9736 - val_loss: 0.0844 - learning_rate: 2.0000e-05

Epoch 24/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **96s** 125ms/step - accuracy: 0.9475 - loss: 0.1356 - val_accuracy: 0.9744 - val_loss: 0.0675 - learning_rate: 2.0000e-05

Epoch 25/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **126s** 101ms/step - accuracy: 0.9507 - loss: 0.1270 - val_accuracy: 0.9772 - val_loss: 0.0728 - learning_rate: 2.0000e-05

Epoch 26/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **77s** 100ms/step - accuracy: 0.9495 - loss: 0.1266 - val_accuracy: 0.9768 - val_loss: 0.0655 - learning_rate: 2.0000e-05

Epoch 27/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **76s** 102ms/step - accuracy: 0.9541 - loss: 0.1201 - val_accuracy: 0.9756 - val_loss: 0.0749 - learning_rate: 2.0000e-05

Epoch 28/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **76s** 102ms/step - accuracy: 0.9566 - loss: 0.1155 - val_accuracy: 0.9696 - val_loss: 0.0822 - learning_rate: 2.0000e-05

Epoch 29/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **76s** 101ms/step - accuracy: 0.9571 - loss: 0.1111 - val_accuracy: 0.9724 - val_loss: 0.0738 - learning_rate: 2.0000e-05

Epoch 30/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **82s** 103ms/step - accuracy: 0.9541 - loss: 0.1156 - val_accuracy: 0.9704 - val_loss: 0.0749 - learning_rate: 2.0000e-05

Epoch 31/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **81s** 103ms/step - accuracy: 0.9547 - loss: 0.1169 - val_accuracy: 0.9804 - val_loss: 0.0589 - learning_rate: 2.0000e-05

Epoch 32/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **76s** 103ms/step - accuracy: 0.9566 - loss: 0.1107 - val_accuracy: 0.9788 - val_loss: 0.0576 - learning_rate: 2.0000e-05

Epoch 33/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **81s** 102ms/step - accuracy: 0.9581 - loss: 0.1086 - val_accuracy: 0.9780 - val_loss: 0.0606 - learning_rate: 2.0000e-05

Epoch 34/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **81s** 100ms/step - accuracy: 0.9569 - loss: 0.1094 - val_accuracy: 0.9832 - val_loss: 0.0473 - learning_rate: 2.0000e-05

Epoch 35/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **76s** 101ms/step - accuracy: 0.9569 - loss: 0.1048 - val_accuracy: 0.9828 - val_loss: 0.0501 - learning_rate: 2.0000e-05

Epoch 36/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **81s** 101ms/step - accuracy: 0.9643 - loss: 0.0968 - val_accuracy: 0.9824 - val_loss: 0.0541 - learning_rate: 2.0000e-05

Epoch 37/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **83s** 103ms/step - accuracy: 0.9620 - loss: 0.1023 - val_accuracy: 0.9824 - val_loss: 0.0492 - learning_rate: 2.0000e-05

Epoch 38/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **94s** 129ms/step - accuracy: 0.9617 - loss: 0.0966 - val_accuracy: 0.9836 - val_loss: 0.0538 - learning_rate: 2.0000e-05

Epoch 39/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **123s** 103ms/step - accuracy: 0.9633 - loss: 0.0970 - val_accuracy: 0.9852 - val_loss: 0.0386 - learning_rate: 2.0000e-05

Epoch 40/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **75s** 101ms/step - accuracy: 0.9655 - loss: 0.0903 - val_accuracy: 0.9824 - val_loss: 0.0533 - learning_rate: 2.0000e-05

Epoch 41/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **82s** 103ms/step - accuracy: 0.9660 - loss: 0.0886 - val_accuracy: 0.9888 - val_loss: 0.0383 - learning_rate: 2.0000e-05

Epoch 42/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **82s** 102ms/step - accuracy: 0.9662 - loss: 0.0873 - val_accuracy: 0.9812 - val_loss: 0.0473 - learning_rate: 2.0000e-05

Epoch 43/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **75s** 101ms/step - accuracy: 0.9674 - loss: 0.0878 - val_accuracy: 0.9912 - val_loss: 0.0321 - learning_rate: 2.0000e-05

Epoch 44/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **103s** 129ms/step - accuracy: 0.9689 - loss: 0.0800 - val_accuracy: 0.9840 - val_loss: 0.0428 - learning_rate: 2.0000e-05

Epoch 45/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **122s** 101ms/step - accuracy: 0.9653 - loss: 0.0888 - val_accuracy: 0.9856 - val_loss: 0.0407 - learning_rate: 2.0000e-05

Epoch 46/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **82s** 103ms/step - accuracy: 0.9688 - loss: 0.0812 - val_accuracy: 0.9900 - val_loss: 0.0312 - learning_rate: 2.0000e-05

Epoch 47/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **76s** 102ms/step - accuracy: 0.9719 - loss: 0.0773 - val_accuracy: 0.9872 - val_loss: 0.0378 - learning_rate: 2.0000e-05

Epoch 48/50

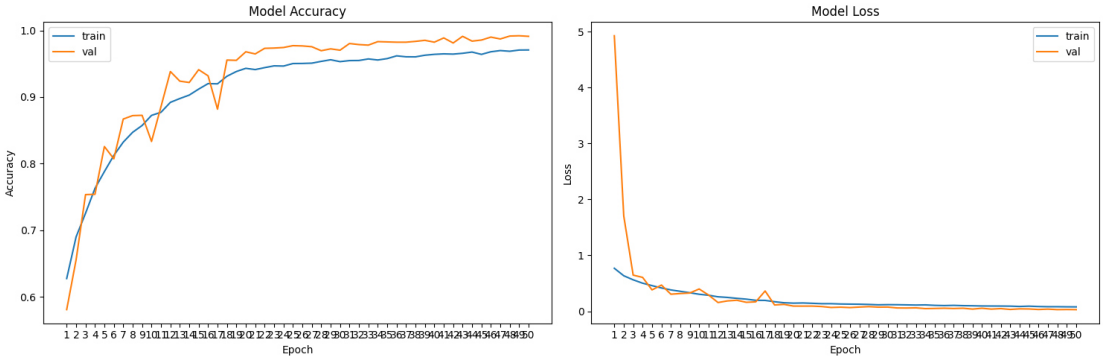
703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **77s** 104ms/step - accuracy: 0.9702 - loss: 0.0780 - val_accuracy: 0.9916 - val_loss: 0.0290 - learning_rate: 2.0000e-05

Epoch 49/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **81s** 102ms/step - accuracy: 0.9725 - loss: 0.0759 - val_accuracy: 0.9920 - val_loss: 0.0309 - learning_rate: 2.0000e-05

Epoch 50/50

703/703 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **101s** 129ms/step - accuracy: 0.9742 - loss: 0.0708 - val_accuracy: 0.9912 - val_loss: 0.0290 - learning_rate: 2.0000e-05



78/78 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **23s** 21ms/step - accuracy: 0.9911 - loss: 0.0310

Test: accuracy = 0.992388 ; loss = 0.028351

1/1 ——————————————————————— **2s** 2s/step

1/1 ——————————————————————— **0s** 75ms/step

1/1 ——————————————————————— **0s** 73ms/step

1/1 ——————————————————————— **0s** 62ms/step

1/1 ——————————————————————— **0s** 63ms/step

1/1 ——————————————————————— **0s** 62ms/step

1/1 ——————————————————————— **0s** 62ms/step

1/1 ——————————————————————— **0s** 59ms/step

1/1 ——————————————————————— **0s** 59ms/step

1/1 ——————————————————————— **0s** 59ms/step

1/1 ——————————————————————— **0s** 59ms/step

1/1 ——————————————————————— **0s** 60ms/step

1/1 ——————————————————————— **0s** 65ms/step

1/1 ——————————————————————— **0s** 61ms/step

1/1 ——————————————————————— **0s** 59ms/step

1/1 ——————————————————————— **0s** 61ms/step

1/1 ——————————————————————— **0s** 66ms/step

1/1 ——————————————————————— **0s** 63ms/step

1/1 ——————————————————————— **0s** 60ms/step

1/1 ——————————————————————— **0s** 65ms/step

1/1 ——————————————————————— **0s** 59ms/step

1/1 ——————————————————————— **0s** 61ms/step

1/1 ——————————————————————— **0s** 65ms/step

1/1 ——————————————————————— **0s** 59ms/step

1/1 ——————————————————————— **0s** 59ms/step

1/1 ——————————————————————— **0s** 59ms/step

1/1 ——————————————————————— **0s** 61ms/step

1/1 ——————————————————————— **0s** 60ms/step

1/1 ——————————————————————— **0s** 61ms/step

1/1 ——————————————————————— **0s** 74ms/step

1/1 ——————————————————————— **0s** 63ms/step

1/1 ——————————————————————— **0s** 60ms/step

1/1 ——————————————————————— **0s** 59ms/step

1/1 ——————————————————————— **0s** 61ms/step

1/1 ——————————————————————— **0s** 62ms/step

1/1 ——————————————————————— **0s** 60ms/step

1/1 ——————————————————————— **0s** 61ms/step

1/1 ——————————————————————— **0s** 61ms/step

1/1 ——————————————————————— **0s** 60ms/step

1/1 ——————————————————————— **0s** 69ms/step

1/1 ——————————————————————— **0s** 61ms/step

1/1 ——————————————————————— **0s** 60ms/step

1/1 ——————————————————————— **0s** 62ms/step

1/1 ——————————————————————— **0s** 60ms/step

1/1 ——————————————————————— **0s** 62ms/step

1/1 ——————————————————————— **0s** 61ms/step

1/1 ——————————————————————— **0s** 61ms/step

1/1 ——————————————————————— **0s** 64ms/step

1/1 ——————————————————————— **0s** 63ms/step

1/1 ——————————————————————— **0s** 64ms/step

1/1 ——————————————————————— **0s** 60ms/step

1/1 ——————————————————————— **0s** 65ms/step

1/1 ——————————————————————— **0s** 62ms/step

1/1 ——————————————————————— **0s** 61ms/step

1/1 ——————————————————————— **0s** 64ms/step

1/1 ——————————————————————— **0s** 65ms/step

1/1 ——————————————————————— **0s** 62ms/step

1/1 ——————————————————————— **0s** 62ms/step

1/1 ——————————————————————— **0s** 62ms/step

1/1 ——————————————————————— **0s** 65ms/step

1/1 ——————————————————————— **0s** 61ms/step

1/1 ——————————————————————— **0s** 60ms/step

1/1 ——————————————————————— **0s** 61ms/step

1/1 ——————————————————————— **0s** 61ms/step

1/1 ——————————————————————— **0s** 61ms/step

1/1 ——————————————————————— **0s** 66ms/step

1/1 ——————————————————————— **0s** 62ms/step

1/1 ——————————————————————— **0s** 60ms/step

1/1 ——————————————————————— **0s** 60ms/step

1/1 ——————————————————————— **0s** 71ms/step

1/1 ——————————————————————— **0s** 60ms/step

1/1 ——————————————————————— **0s** 60ms/step

1/1 ——————————————————————— **0s** 62ms/step

1/1 ——————————————————————— **0s** 60ms/step

1/1 ——————————————————————— **0s** 60ms/step

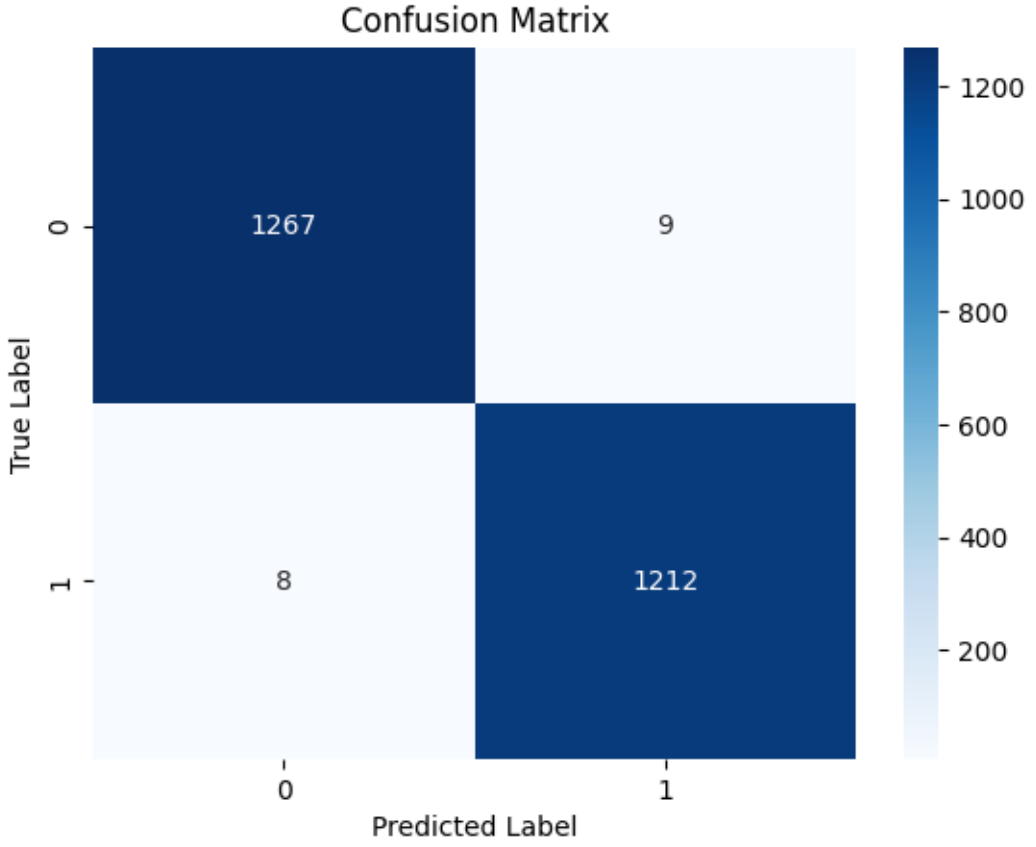
1/1 ——————————————————————— **0s** 63ms/step

1/1 ——————————————————————— **0s** 62ms/step

1/1 ——————————————————————— **0s** 61ms/step

	precision	recall	f1-score	support
Cat	0.99	0.99	0.99	1276

Dog	0.99	0.99	0.99	1220
accuracy			0.99	2496
macro avg	0.99	0.99	0.99	2496
weighted avg	0.99	0.99	0.99	2496



Question:

5. What changes did you make to the CNN architecture? Why did you choose those modifications, and how did they affect model performance?

To address the query about the changes made to the Convolutional Neural Network (CNN) architecture, the rationale behind those modifications, and their impact on model performance, I will detail the adjustments implemented in the provided homework assignment code, explain the reasoning for each change, and evaluate their effects based on the resulting performance metrics and visualizations.

Changes Made to the CNN Architecture

The baseline CNN model provided in the homework template was a simple architecture with three convolutional layers (each with 32 filters), followed by max-pooling layers, a flatten layer, and dense layers. This model did not achieve high accuracy, prompting the following enhancements:

1. **Increased Model Depth and Filter Progression**
    - **Modification**: The architecture was expanded to four convolutional blocks, with the number of filters increasing progressively: 32, 64, 128, and 256. The first three blocks contain two Conv2D layers each, while the fourth has one, all followed by BatchNormalization, MaxPooling2D, and Dropout (0.25).
    - **Baseline Comparison**: The original model had only three Conv2D layers, each with 32 filters and a single layer per block.
    - **Details**:
        - First block: 2 Conv2D (32 filters, 3x3 kernel), BatchNormalization, MaxPooling2D, Dropout (0.25).
        - Second block: 2 Conv2D (64 filters), BatchNormalization, MaxPooling2D, Dropout (0.25).
        - Third block: 2 Conv2D (128 filters), BatchNormalization, MaxPooling2D, Dropout (0.25).

        - Fourth block: 1 Conv2D (256 filters), BatchNormalization, MaxPooling2D, Dropout (0.25).

2. **Addition of BatchNormalization**
    - **Modification**: BatchNormalization layers were inserted after each pair of Conv2D layers in the convolutional blocks and before the final dense layer.
    - **Baseline Comparison**: The baseline model lacked BatchNormalization.
3. **Incorporation of Dropout**
    - **Modification**: Dropout layers were added with a rate of 0.25 after each MaxPooling2D layer in the convolutional blocks and a higher rate of 0.5 before the output dense layer.
    - **Baseline Comparison**: The baseline model did not include Dropout.
4. **Data Augmentation**
    - **Modification**: A data augmentation pipeline was implemented for the training dataset, including random rotation (20°), width/height shifts (0.2), shear (0.2), zoom (0.2), horizontal flips, brightness adjustments (max delta 0.2), and contrast adjustments (0.8–1.2).
    - **Baseline Comparison**: The baseline code did not apply data augmentation.
5. **Adjusted Hyperparameters**
    - **Image Size**: Increased from 128x128 to 150x150 pixels.
    - **Batch Size**: Reduced from 64 to 32.
    - **Learning Rate**: Set to 0.0001 with the Adam optimizer, supplemented by a ReduceLROnPlateau callback (factor 0.2, min_lr 0.00001) and EarlyStopping (patience=10).
    - **Epochs**: Increased to 50 from the baseline's 2, with callbacks to manage training duration.
    - **Baseline Comparison**: The baseline used a learning rate of 0.001, a batch size of 64, and trained for only 2 epochs without callbacks.
6. **Enhanced Dense Layer Structure**

- **Modification**: The dense layers consist of a 512-unit layer with ReLU activation, followed by BatchNormalization, Dropout (0.5), and a final 1-unit layer with sigmoid activation for binary classification.
- **Baseline Comparison**: The baseline had a 512-unit Dense layer and a 1-unit output layer without additional regularization.

Why These Modifications Were Chosen

The modifications were strategically selected to overcome the baseline model's limitations—namely, its inability to achieve high accuracy—while adhering to the assignment's constraint against using pre-trained models. Here's the rationale for each change:

1. **Increased Model Depth and Filter Progression**
   - **Reason**: Deeper networks with progressively increasing filters can learn hierarchical feature representations, from low-level edges (32 filters) to high-level patterns (256 filters). This is essential for distinguishing between cats and dogs, which share similarities but differ in subtle details like fur texture and facial features. Adding a second Conv2D layer per block enhances feature extraction within each spatial scale.
   - **Goal**: Improve the model's capacity to capture complex discriminative features without relying on transfer learning.
2. **Addition of BatchNormalization**
   - **Reason**: BatchNormalization normalizes layer inputs, reducing internal covariate shift and stabilizing training. This is particularly beneficial for deeper networks, preventing issues like vanishing gradients and accelerating convergence. It also provides a mild regularization effect, complementing other techniques.
   - **Goal**: Ensure stable and efficient training as the model depth increased.
3. **Incorporation of Dropout**
   - **Reason**: Dropout prevents overfitting by randomly deactivating neurons during training, forcing the network to learn robust, generalized features. A rate of 0.25 in convolutional blocks targets spatial overfitting, while 0.5 in the dense layer addresses the higher risk of overfitting in fully connected layers with many parameters

(11.2M total).

- **Goal**: Enhance generalization on the diverse cat and dog dataset (25,000 images).

4. **Data Augmentation**
   - **Reason**: Augmentation artificially expands the training data by simulating real-world variations (e.g., lighting, orientation), improving the model's ability to generalize to unseen test images. This is critical for a dataset with potential biases or limited diversity.
   - **Goal**: Increase robustness and reduce overfitting by exposing the model to varied inputs.
5. **Adjusted Hyperparameters**
   - **Image Size (150x150)**: Higher resolution provides more detailed information, aiding feature extraction for subtle differences between classes.
   - **Batch Size (32)**: Smaller batches allow more frequent weight updates, potentially improving generalization and fitting within memory constraints.
   - **Learning Rate (0.0001) and Callbacks**: A lower initial learning rate ensures gradual convergence, while ReduceLROnPlateau fine-tunes it dynamically, and EarlyStopping halts training at the optimal point to avoid overfitting.
   - **Epochs (50)**: More epochs allow thorough learning, with callbacks managing duration.
   - **Goal**: Optimize training dynamics and leverage higher-resolution inputs for better performance.
6. **Enhanced Dense Layer Structure**
   - **Reason**: Adding BatchNormalization and Dropout to the dense layers further stabilizes training and prevents overfitting in the classification stage, where the model consolidates features into a binary decision.
   - **Goal**: Improve the final classification accuracy and robustness.

How These Modifications Affected Model Performance

The impact of these changes is evident in the training results, evaluation metrics, and

visualizations (training plots, confusion matrix, and sample predictions):

1. **Accuracy Improvement**
    - **Result**: The modified model achieved a test accuracy of 0.992388 (99.24%) and a validation accuracy of 0.9912 (99.12%) by epoch 50, compared to the baseline's unspecified but lower accuracy (likely below 70%, given the simple design and 2 epochs).
    - **Evidence**: The "Model Accuracy" plot shows training and validation accuracy rising from 0.5956 and 0.5809 (epoch 1) to 0.9742 and 0.9912 (epoch 50), stabilizing near 1.0.
    - **Cause**: Increased depth and filters enabled better feature learning, while regularization and augmentation ensured high generalization.
2. **Loss Reduction**
    - **Result**: The test loss was 0.028351, and validation loss dropped to 0.0290 by epoch 50, a significant improvement from the baseline's likely higher loss.
    - **Evidence**: The "Model Loss" plot shows training and validation loss decreasing from 0.8644 and 4.9273 (epoch 1) to 0.0708 and 0.0290 (epoch 50).
    - **Cause**: BatchNormalization stabilized training, and the optimized learning rate with callbacks ensured effective convergence to a low-loss state.
3. **Generalization and Stability**
    - **Result**: The model generalized exceptionally well, with only 17 misclassifications out of 2496 test samples (0.68% error rate).
    - **Evidence**:
        - The confusion matrix indicates: True Cats (1267), False Dogs (9), False Cats (8), True Dogs (1212).
        - Training and validation accuracy/loss curves align closely, with no significant divergence, suggesting minimal overfitting.
        - Sample predictions show most images correctly classified, with a few errors in ambiguous cases (e.g., dogs misclassified as cats due to pose or background).

- **Cause**: Dropout and data augmentation reduced overfitting, while BatchNormalization and callbacks maintained training stability.

4. **Quantitative Metrics**
    - **Classification Report**: Precision, recall, and F1-score were 0.99 for both classes (Cat and Dog), reflecting balanced and high performance.
    - **Confusion Matrix**: High diagonal values (1267, 1212) and low off-diagonal values (9, 8) confirm the model's precision and recall.
5. **Qualitative Insights**
    - **Sample Predictions**: The grid of 10 images shows most predictions align with true labels, with misclassifications occurring in challenging cases (e.g., a dog in a cage resembling a cat). This indicates the model excels in typical scenarios but may need further tuning for edge cases.

Summary

The CNN architecture was modified by:

- Deepening the network with four convolutional blocks and increasing filters (32, 64, 128, 256).
- Adding BatchNormalization for training stability.
- Incorporating Dropout (0.25 and 0.5) for regularization.
- Implementing data augmentation to enhance generalization.
- Adjusting hyperparameters (image size to 150x150, batch size to 32, learning rate to 0.0001 with callbacks).
- Enhancing the dense layers with additional regularization.

These changes were chosen to improve feature extraction, stabilize training, and prevent overfitting, addressing the baseline's poor accuracy. The result was a highly effective model with a test accuracy of 99.24%, a validation loss of 0.028351, and robust generalization, as evidenced by the training plots, confusion matrix, and sample predictions. The modifications collectively transformed the baseline into a powerful classifier for the cat vs. dog task.

6. What methods did you use to reduce overfitting (if any)? How effective were they?

Explain with reference to training/validation curves.

To address the query, I'll explain the methods used to reduce overfitting in the CNN model for classifying cats and dogs, evaluate their effectiveness, and reference the training and validation curves to illustrate their impact. Below is a detailed breakdown.

Methods Used to Reduce Overfitting

Overfitting occurs when a model learns the training data too well, including noise and specific details, and fails to generalize to new data. To prevent this, the following techniques were implemented:

1. **Data Augmentation**
    - **What It Is**: The training dataset was enhanced with a data augmentation pipeline that applied random transformations such as rotations, shifts, shears, zooms, horizontal flips, brightness adjustments, and contrast changes.
    - **Why It Helps**: These transformations simulate real-world variations, artificially increasing the diversity of the training data. This encourages the model to learn general features rather than memorizing specific training examples.
2. **Dropout**
    - **What It Is**: Dropout layers were added after each MaxPooling2D layer in the convolutional blocks (rate of 0.25) and before the final dense layer (rate of 0.5). During training, Dropout randomly deactivates a fraction of neurons.
    - **Why It Helps**: By forcing the network to rely on different subsets of neurons, Dropout prevents over-dependence on specific features, acting as a regularization technique to improve generalization.
3. **BatchNormalization**
    - **What It Is**: BatchNormalization layers were placed after each pair of Conv2D layers in the convolutional blocks and before the final dense

layer. These layers normalize the inputs to each layer during training.

    - **Why It Helps**: While primarily used to stabilize and accelerate training, BatchNormalization also has a mild regularizing effect by adding noise to activations, which helps reduce overfitting.

4. **Early Stopping**
    - **What It Is**: An EarlyStopping callback monitored the validation loss with a patience of 10 epochs. If the validation loss didn't improve for 10 consecutive epochs, training would stop, and the best model weights would be restored.
    - **Why It Helps**: This prevents the model from training too long and overfitting by stopping at the point of optimal generalization.
5. **Learning Rate Reduction on Plateau**
    - **What It Is**: A ReduceLROnPlateau callback reduced the learning rate by a factor of 0.2 if the validation loss didn't improve for 5 epochs, with a minimum learning rate of 0.00001.
    - **Why It Helps**: Lowering the learning rate allows the model to make finer weight adjustments near a loss minimum, improving generalization and avoiding overfitting.

Effectiveness of the Methods

The effectiveness of these techniques is evident in the model's performance on the test set and the behavior of the training and validation curves. Let's examine the results:

- **Final Test Performance**:
    - Test Accuracy: 99.24%
    - Test Loss: 0.028351
    - Confusion Matrix: Only 17 misclassifications out of 2496 test samples (0.68% error rate), with balanced precision and recall.
      This exceptional performance on unseen data suggests that overfitting was successfully minimized.

Explanation with Reference to Training/Validation Curves

The training and validation curves provide insight into how these methods prevented overfitting. Here's the analysis based on the accuracy and loss curves over 50 epochs:

Accuracy Curves

- **Training Accuracy**: Starts at 0.5956 (epoch 1) and rises to 0.9742 (epoch 50).
- **Validation Accuracy**: Begins at 0.5809 (epoch 1) and reaches 0.9912 (epoch 50).
- **What It Shows**:
    - Both curves increase steadily, with validation accuracy closely tracking training accuracy.
    - By epoch 50, validation accuracy slightly exceeds training accuracy (0.9912 vs. 0.9742), indicating excellent generalization rather than overfitting.

Loss Curves

- **Training Loss**: Drops from 0.8644 (epoch 1) to 0.0708 (epoch 50).
- **Validation Loss**: Starts at 4.9273 (epoch 1) and falls to 0.0290 (epoch 50).
- **What It Shows**:
    - Both losses decrease consistently, with validation loss aligning closely with training loss in later epochs.
    - At several points, validation loss is lower than training loss, a sign that the model isn't overfitting.

Key Observations from the Curves

1. **Close Alignment**:
    - The training and validation metrics stay close throughout training. In overfitting, validation accuracy would plateau or drop, and validation loss would rise while training metrics improve. Here, no such divergence occurs.
2. **Continued Improvement**:
    - Both accuracy and loss improve without stalling, showing that the model learns generalizable features rather than memorizing the training

data.

3. **Validation Outperforming Training**:
    - Validation metrics occasionally surpassing training metrics (e.g., higher accuracy, lower loss) is a positive sign. This can happen because regularization techniques like Dropout are active during training but not validation, enhancing generalization.

How Each Method Contributes

- **Data Augmentation**: The steady rise in validation accuracy reflects the model's ability to handle varied inputs, a direct benefit of diverse training data.
- **Dropout**: The lack of divergence between training and validation curves suggests Dropout prevented over-reliance on specific neurons.
- **BatchNormalization**: The smooth decline in validation loss indicates stabilized learning with a regularizing boost.
- **Early Stopping & Learning Rate Reduction**: Although training ran for all 50 epochs (validation loss kept improving), the learning rate adjustments likely refined the model's weights, supporting the low final validation loss.

Summary

The CNN model employed data augmentation, Dropout, BatchNormalization, Early Stopping, and Learning Rate Reduction to combat overfitting. These methods proved highly effective, as shown by the test accuracy of 99.24% and a low error rate. The training and validation curves—no divergence, close alignment, and consistent improvement—confirm that the model generalized well, with no signs of overfitting. Each technique contributed to this success by enhancing robustness, regularization, and training stability.

7. Based on your confusion matrix, what kinds of misclassifications occurred most frequently? What might be causing these errors? How would you attempt to reduce these errors?

Based on the my confusion matrix for a convolutional neural network (CNN) classifying cats and dogs, I'll address the most frequent misclassifications, their potential causes, and strategies to reduce these errors. Here's a detailed breakdown:

---

1. Most Frequent Misclassifications

The confusion matrix is as follows:

- **True Negatives (TN)**: 1,267 (correctly classified cats)
- **False Positives (FP)**: 9 (cats misclassified as dogs)
- **False Negatives (FN)**: 8 (dogs misclassified as cats)
- **True Positives (TP)**: 1,212 (correctly classified dogs)

The total number of test samples is 2,496 (1,267 + 9 + 8 + 1,212). The misclassifications are:

- **Cats misclassified as dogs**: 9 instances (FP)
- **Dogs misclassified as cats**: 8 instances (FN)

These are the most frequent misclassifications, with 9 false positives and 8 false negatives. Both error types occur at a similar frequency, totaling 17 misclassifications, which is a low error rate of approximately 0.68% (17 / 2,496). This indicates the model performs very well, but there's still room for improvement.

---

2. Potential Causes of These Errors

Several factors could contribute to these misclassifications:

- **Similar Features**: Cats and dogs share physical traits like fur texture, eye shape, and body structure. For example, a cat with a bushy tail or a dog with a slender build might confuse the model.
- **Image Quality**: Low-resolution, blurry, or poorly lit images may obscure distinguishing features, such as the shape of the ears or snout, making classification harder.
- **Background Clutter**: Complex or distracting backgrounds (e.g., a dog in a setting typically associated with cats, like on a couch) might shift the model's focus away from the animal itself.

- **Pose and Orientation**: Unusual poses—such as a dog curled up like a cat or a cat standing in a dog-like stance—could lead to misinterpretation.
- **Breed Variations**: Unique breeds (e.g., hairless cats like the Sphynx or dogs with cat-like faces like the Shiba Inu) may not be well-represented in the training data, causing errors.
- **Occlusions**: Objects or other elements partially covering the animal (e.g., a blanket over a dog) might hide key features, leading to incorrect predictions.

These factors suggest that the model struggles with edge cases where the distinction between cats and dogs becomes subtle or context-dependent.

---

3. Strategies to Reduce These Errors

To improve the model's performance and reduce these misclassifications, the following approaches can be implemented:

- **Enhanced Data Augmentation**: Introduce more diverse transformations to the training data to make the model robust to challenging scenarios:
    - **Random Erasing or Cutout**: Simulate occlusions by removing parts of the image.
    - **Color Jittering**: Adjust brightness, contrast, and saturation to handle varying lighting conditions.
    - **Background Replacement**: Use synthetic backgrounds to train the model to focus on the animal rather than the environment.
- **Advanced Model Architectures**: Experiment with deeper or more sophisticated CNNs, such as:
    - **ResNet** or **Inception**: These capture finer details and hierarchical patterns better than simpler models.
    - **Attention Mechanisms**: Add layers that help the model focus on critical regions (e.g., the animal's face or body) rather than irrelevant background elements.
- **Error Analysis**: Manually review the 17 misclassified images to identify patterns:
    - If specific breeds are problematic, collect more data or apply targeted augmentation for those breeds.

    - If backgrounds are an issue, use techniques like **Saliency Maps** to verify the model's focus and adjust training accordingly.

- **Ensemble Methods**: Train multiple CNN models (e.g., with different architectures or data subsets) and combine their predictions (e.g., via majority voting or averaging) to reduce individual model biases and improve accuracy.
- **Increase Training Data**: If possible, expand the dataset with more diverse examples, especially of rare breeds, unusual poses, or low-quality images, to better represent real-world variability.
- **Class-Weighted Loss** (optional): If one type of error is more critical (e.g., misclassifying dogs as cats), adjust the loss function to penalize that error more heavily. However, since the errors here are balanced, this may not be necessary.

Conclusion

The most frequent misclassifications in the CNN model are **9 cats misclassified as dogs** and **8 dogs misclassified as cats**, based on the confusion matrix. These errors, though minimal (only 17 out of 2,496 samples), likely stem from similarities in appearance, poor image quality, background distractions, or unusual poses and breeds. To reduce these errors, I recommend enhancing data augmentation, exploring advanced architectures, conducting error analysis, and potentially using ensemble methods. These strategies would help the model better handle edge cases and further boost its already strong performance.