Lab Report: **Lab-9 Regularization and Model Compression**

Kaung Sett Thu
12503548
Embedded System
Instructor: Prof. Tobias Schaffer

Date

1 Introduction

The lab is divided into two parts, regularization and model compression.

- 9.1 Regularization
- 9.2 Model Compression

This lab report will be organized using the following sections

- Methodology
- Experiment Results
- Challenges, Limitations and Error Analysis
- Discussion of Finding
- Conclusion
- Reference
- Appendix

2 Methodology

The lab is performed using the data set that contains images of dogs and cats. The data set is first trained and validated using the historical CNN model. Afterward, different regularization methods will be applied and the results will be evaluated.

2.1 Software and Hardware Used

- Programming language: Python
- Libraries: TensorFlow, NumPy, scikit-learn, Matplotlib
- Hardware: Intel Core i7-118800H @ 2.30Hz; 16 Cores

2.2 Code Repository

The full source code for this project is available on GitHub at:

`https://github.com/yourusername/your-repository`

This repository includes:

- Source code files
- Example dataset
- Evidence images
- Saved models

2.3 Implementation of Regularization

The code for the regularization of the CNN is written in the file **lab9.1-regularization.ipynb**.

A zip file for the data set is being downloaded using gdown library. The zip file is then extracted and stored in the directory **"/data"**.

```
!pip install -q gdown
!gdown https://drive.google.com/uc?id=12WhCCpKTWpeBztLegcoYx2gMo2KbaxDG

import zipfile

with zipfile.ZipFile('dogs-vs-cats.zip', 'r') as zip_file:
    zip_file.extractall('data')

with zipfile.ZipFile('data/train.zip', 'r') as zip_file:
    zip_file.extractall('data/')
```

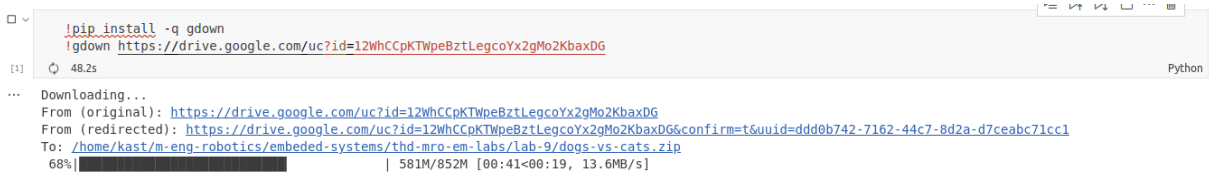

Figure 1: Downloading the data set

The data is then divided into the training data and the validation data and stored in the directories **"data/train_split"** and **"data/val_split"** respectively.

```
# Paths
base_dir = 'data/train'
train_dir = 'data/train_split'
val_dir = 'data/val_split'

# Create directories
os.makedirs(os.path.join(train_dir, 'dogs'), exist_ok=True)
os.makedirs(os.path.join(train_dir, 'cats'), exist_ok=True)
```

```
os.makedirs(os.path.join(val_dir, 'dogs'), exist_ok=True)
os.makedirs(os.path.join(val_dir, 'cats'), exist_ok=True)

# Split data
filenames = os.listdir(base_dir)
train_files, val_files = train_test_split(filenames, test_size=0.2,
    random_state=42)

for file in train_files:
    if 'dog' in file:
        shutil.move(os.path.join(base_dir, file), os.path.join(train_dir
            , 'dogs', file))
    elif 'cat' in file:
        shutil.move(os.path.join(base_dir, file), os.path.join(train_dir
            , 'cats', file))

for file in val_files:
    if 'dog' in file:
        shutil.move(os.path.join(base_dir, file), os.path.join(val_dir,
            'dogs', file))
    elif 'cat' in file:
        shutil.move(os.path.join(base_dir, file), os.path.join(val_dir,
            'cats', file))
```

The stored data is loaded and, the images are reshaped to a uniform size of 150 x 150 and are grouped into the batches size of 32.

```
train_datagen = ImageDataGenerator(rescale=1./255)
val_datagen = ImageDataGenerator(rescale=1./255)

# Load data from directories
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(150, 150),
    batch_size=32,
    class_mode='binary'
)

val_generator = val_datagen.flow_from_directory(
    val_dir,
    target_size=(150, 150),
    batch_size=32,
    class_mode='binary'
)
```

2.3.1 Baseline Accuracy

A classic CNN model is built using TensorFlow. The input of shape 150 x 150 x 3 is passed through the alternating **Convolution** layers of 3 x 3 kernel and M**ax Pooling** layers 3 times. The resulting feature maps are flattened and passed to the fully connected hidden layer of 128 neurons, using **ReLu** activation function. There is one output function with the **sigmoid** activation function to classify whether the image is a cat or a dog.

```
from tensorflow.keras import models, layers
```

```
# Build the model
model_basic = models.Sequential([
    layers.Input(shape=(150, 150, 3)),

    layers.Conv2D(32, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])
```

The created model is compiled using **Adam** optimizer and the **Binary cross entropy loss** as a loss function.

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

The model is then trained with the use of the training data set and validated with the use of validation data set over 10 epochs.

```
history_model_basic = model_basic.fit(
    train_generator,
    epochs=10,
    validation_data=val_generator,
)
```

2.3.2 Dropout Regularization

A dropout of 0.3 and 0.5 are added to the model to avoid the overfitting of the model. This means 30% and 50% of the neurons are randomly ignored during the training of each epoch. Dropout is configured in such a way that 30% of neurons is ignored in the first convolution layer, 30% in the last convolution layer, and 50% after the fully connected layer, before the output layer.

```
from tensorflow.keras import models, layers

model_dropout = models.Sequential([
    layers.Input(shape=(150, 150, 3)),

    layers.Conv2D(32, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Dropout(0.3),
```

```
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Dropout(0.3),

    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(1, activation='sigmoid')
])
```

The model is then complied using Adam optimizer and trained over 10 epochs for a fairness in evaluation.

2.3.3 L2 Regularization

A L2 regularization is performed by decaying the square or the weights with the regularization factor lambda, λ and adding it to the loss function.

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{data}} + \lambda \sum_{i=1}^{n} w_i^2$$

A lambda, λ value of 0.001 is used here for regularization.

```
from tensorflow.keras import regularizers

model_l2 = models.Sequential([
    layers.Input(shape=(150, 150, 3)),

    layers.Conv2D(32, (3, 3), activation='relu',
                  kernel_regularizer=regularizers.l2(0.001)),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu',
                  kernel_regularizer=regularizers.l2(0.001)),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(128, (3, 3), activation='relu',
                  kernel_regularizer=regularizers.l2(0.001)),
    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),
    layers.Dense(128, activation='relu',
                 kernel_regularizer=regularizers.l2(0.001)),
    layers.Dense(1, activation='sigmoid')
])
```

Once again, the model is complied using Adam optimizer and trained over 10 epochs for a fairness in evaluation.

2.3.4 Combination of Dropout and L2 Regularization - Configuration 1

A dropout and L2 regularizations, with prior configurations from 2.3.2 and 2.3.3, are used together to create a model that makes use of combined regularization. The model is

compiled using Adam optimizer and trained over 10 epochs.

```
from tensorflow.keras import models, layers, regularizers

model_combined_config1 = models.Sequential([
    layers.Input(shape=(150, 150, 3)),

    layers.Conv2D(32, (3, 3), activation='relu',
                  kernel_regularizer=regularizers.l2(0.001)),
    layers.MaxPooling2D((2, 2)),
    layers.Dropout(0.3),

    layers.Conv2D(64, (3, 3), activation='relu',
                  kernel_regularizer=regularizers.l2(0.001)),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(128, (3, 3), activation='relu',
                  kernel_regularizer=regularizers.l2(0.001)),
    layers.MaxPooling2D((2, 2)),
    layers.Dropout(0.3),

    layers.Flatten(),
    layers.Dense(128, activation='relu',
                 kernel_regularizer=regularizers.l2(0.001)),
    layers.Dropout(0.5),
    layers.Dense(1, activation='sigmoid')
])
```

2.3.5 Combination of Dropout and L2 Regularization - Configuration 2

A different configuration for the combined regularization of dropout and L2 is also implemented. In this model, a lower amount is dropout is implemented; a dropout of 30% is applied to first convolution layer and before the output layer. The same lambda, λ value of 0.001 is used for L2 regularization. The model is complied using Adam optimizer and trained over 10 epochs.

```
from tensorflow.keras import models, layers, regularizers

model_combined_config2 = models.Sequential([
    layers.Input(shape=(150, 150, 3)),

    layers.Conv2D(32, (3, 3), activation='relu',
                  kernel_regularizer=regularizers.l2(0.001)),
    layers.MaxPooling2D((2, 2)),
    layers.Dropout(0.3),

    layers.Conv2D(64, (3, 3), activation='relu',
                  kernel_regularizer=regularizers.l2(0.001)),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(128, (3, 3), activation='relu',
                  kernel_regularizer=regularizers.l2(0.001)),
    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),
```

```
    layers.Dense(128, activation='relu',
                 kernel_regularizer=regularizers.l2(0.001)),
    layers.Dropout(0.3),
    layers.Dense(1, activation='sigmoid')
])
```

2.3.6 Early Stopping

An early stopping with a patience of 3 on val_loss is implemented. The configuration of the layers is kept in the same way as the classic CNN model. The model is complied using Adam optimizer and trained over 10 epochs.

```
from tensorflow.keras.callbacks import EarlyStopping

early_stop = EarlyStopping(
    monitor='val_loss',
    patience=3,
    restore_best_weights=True
)

model_early_stop = models.Sequential([
    layers.Input(shape=(150, 150, 3)),

    layers.Conv2D(32, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])

model_early_stop.compile(optimizer='adam',
                         loss='binary_crossentropy',
                         metrics=['accuracy'])
```

2.4 Implementation of Model Compression

The base model of classic RNN without any regularization will be used for implementation of model compression. As stated in *2.3 Regularization*, the RNN model for model compression will also have 6 layers of alternating convolution and max pooling.

2.4.1 Full Keras Model

First the model is saved into the full Keras model without any quantization.

```
import os
os.makedirs("models", exist_ok=True)
```

```
# Save the model as keras
model_basic.save("models/model_basic.keras")
```

2.4.2 Default TFLite Model (FP32)

Then the model is compressed into TFLite model without any quantization; keeping the same precision of Float32.

```
model = tf.keras.models.load_model("models/model_basic.keras")

converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()

with open("models/model_basic_default.tflite", "wb") as f:
    f.write(tflite_model)
```

2.4.3 Weight quantization (weights to INT8)

The model is compressed by quantization of the weight to INT8 precision. The quantization to INT8 is a default optimization method for TFLite.

```
model = tf.keras.models.load_model("models/model_basic.keras")

converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_model = converter.convert()

with open("models/model_basic_weight_quant.tflite", "wb") as f:
    f.write(tflite_model)
```

2.4.4 Float16 quantization (weights to FP16)

The model is compressed by quantization of the weight to Float16 precision.

```
model = tf.keras.models.load_model("models/model_basic.keras")

converter.target_spec.supported_types = [tf.float16]
tflite_fp16 = converter.convert()

with open("models/model_basic_fp16.tflite", "wb") as f:
    f.write(tflite_fp16)
```

2.4.5 Integer quantization (weights and activations to INT8)

This is also known as full integer quantization; a quantization of both the weight and the activation is done to INT8 precision.

For this quantization to work, an example data set is required to correctly map the activation values from Float32 to INT8.

```
model = tf.keras.models.load_model("models/model_basic.keras")

converter = tf.lite.TFLiteConverter.from_keras_model(model)

def representative_data_gen():
    for _ in range(100):
        data, _ = next(train_generator)
        yield [data.astype("float32")]

converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_data_gen
converter.target_spec.supported_ops = [tf.lite.OpsSet.
   TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8
converter.inference_output_type = tf.int8

tflite_model_int8 = converter.convert()

with open("models/model_basic_int8.tflite", "wb") as f:
    f.write(tflite_model_int8)
```

3 Results

The results of Lab 9.1 Regularization and Lab 9.2 Model Compression will be shown in different sections for the sake of simplicity.

3.1 Regularization

3.1.1 Baseline Accuracy

In the classic CNN model, after epoch 2, training accuracy is keep on increasing but the validation accuracy remains almost constant. Also, after epoch 2, the validation loss stop decreasing and start increasing. The results are then plotted using the Matplotlib and can be found in *Appendix A.1*.

3.1.2 Dropout Regularization

When drop out regularization is applied, the overfitting is but the validation accuracy stops increasing and validation loss starts to increase after epoch 3. The results are then plotted using the Matplotlib and can be found in *Appendix A.2*.

3.1.3 L2 Regularization

Similar behavior can also be observed with the use of L2 Regularization. The validation accuracy stops increasing and the validation loss starts to increase after epoch 2. The results are then plotted using the Matplotlib and can be found in *Appendix A.3*.

3.1.4 Combined Regularization - Configuration 1

From the first configuration of combined regularization of dropout and L2 *(2.3.4)* the validation accuracy keeps on increasing and validation loss keeps on decreasing over 10

epochs. However, the training accuracy also keeps on increasing and the training loss also keeps on decreasing. Both the training accuracy and the validation accuracy are less than 66% when the training has completed. The results are then plotted using the Matplotlib and can be found in *Appendix A.4*.

3.1.5 Combined Regularization - Configuration 2

In the the second configuration of combined regularization of dropout and L2 *(2.3.5)*, both the training and validation accuracy increases together and the the training and validation loss also decreases together. The results are then plotted using the Matplotlib and can be found in *Appendix A.5*.

3.1.6 Early Stopping

Early stopping regularization shows the similar behavior with the classic CNN but the model is stopped if the validation loss is not decreasing after 3 epochs. The results are then plotted using the Matplotlib and can be found in *Appendix A.6*.

3.2 Model Compression

The file sizes of the different models in Kilobyte (KB) and Megabyte (MB) are as follow:

Model File	**Size (KB)**	**Size (MB)**
model_basic.keras	56628.02	55.30
model_basic_default.tflite	18865.16	18.42
model_basic_weight_quant.tflite	4726.91	4.62
model_basic_fp16.tflite	9435.71	9.21
model_basic_int8.tflite	4729.12	4.62

Table 1: Model File Sizes

4 Challenges, Limitations, and Error Analysis

During the implementation of this project, several challenges and errors were encountered. Below are some key points:

4.1 Challenges Faced

- Understanding of the motivation behind regularization and model compression.

  Before performing the experiments, I have to first refer to the lecture notes and the online resources to fully understand the reason for why regularization and model compression is done.

- Long training time of the model.

Due to the large number of image data set and the use of CPU instead of a more powerful CPU, the models take a long time to train. Since, at least 5 models need to be trained for comparison, a sufficient time must be allocated for training process.

- Pushing to a git hub has challenges due to the large data set.

  Due to having large data set, pushing an entire project folder including the data set is a very bandwidth consuming process. Therefore, only the dataset cannot be included in the repository and the download link to the compressed file of the data set is included in the code.

4.2 Error Analysis

Some common errors that occurred during the development:

- Missing python modules like sklearn or tensorflow results in error. Example error:

Figure 2: Example error

This was fixed by creating a virtual environment and a kernel with all required modules installed.

```
pip install scikit-learn
```

- Training or plotting the wrong model.

  Due to the models having similar names, incorrect models were mistakenly trained and plotted leading to incorrect results. This could be avoided by having distinct and meaningful model names.

4.3 Limitations of the Implementation

The model has limitations in the amount of data it can train and has a longer training time due to the use of the CPU instead of using a more power GPU.

The model is also only trained with the data set that includes only the cat and the dog photos and works with a linear regression. Due to the the model is more prone is overfitting

if the data set is not balanced and it could just be predicting the more common data out of the two.

The model also makes use of smaller CNN architecture with lesser number of lawyers compared to typical CNN models like VGG-16. Even though less complex model can avoid overfitting, the model also risks underfitting when regularization methods are applied.

Due to limitation in availability of edge devices, the accuracy of the compressed model on the prediction was also not tested.

5 Discussion

5.0.1 Regularization

Classic CNN *(Appendix A.1)* results show strong indications of the model overfitting. Instead of being trained, the model is trying to memorize the features during the training process.

From the results of dropout regularization results *(Appendix A.2)*, we can see that even though it reduces the gap between training and validation, it does not fix the overfitting when the model is trained over more epochs.

In L2 regularization *(Appendix A.3)*, the gap between training and validation is not as wide as the previous experiments, indicating that the model is overfitting less.

From the first configuration of combined regularization results *(Appendix A.4)*, it can be seen that the model is now underfitting and is likely to give incorrect predictions.

The results of second configuration of combined regularization *(Appendix A.5)* suggest that we have achieved a properly trained model from this configuration.

In early stopping results *(Appendix A.6)*, we can see that there is still an issue of slight overfitting here but the model has reduced overfit and reduced training time.

From the experiments on regularization, the effects of different regularization methods can be observed. It can be concluded that, for this model, even though applying individual regularization methods help, they have limited effectiveness. Combined configuration of different regularization methods yield better models. It is also important to note that too much of a regularization could result in model underfitting.

5.0.2 Model Compression

Applying the model compression evidently shows significant reduction in the model size. Compressing to TFLite model reduces the model size from 55.30 MB to 18.42 MB which is around 66% of reduction in size. Weight quantization to INT8 reduces the file size up to 4.62 MB and weight quantization to Float16 reduces the files size to 9.21 MB.

It can be observed that the quantization of both the weight and the activation to INT8 results in an almost similar file size with only weight quantization. This suggests that the weights are the dominant factor in the model size.

Compression of models reduces the file size significantly making the models more suitable to be deployed on edge devices. However, due to the reduction of precision there may also have impacts on the accuracy of the model.

6 Conclusion

The experiment successfully demonstrates the impacts of regularization and model compression on basic CNN model. From the regularization experiments, it can be concluded that a use of combined regularization can have better results compared to using just one regularization method. The model compression also shows how the quantization of the model can have significant impact on the size of the model.

For future experiments, more complex CNN models can be used to gain the better insights on effects on combined regularization and model compression. GPUs should be used for training of the models to save time. The compressed models should also be applied to the edge devices to analyze the accuracy of them when the precision is reduced and to evaluate the trade-off between size reduction and accuracy.

7 References

- Takano. Shigeyuki, "Thinking Machines", Academic Press, 2021.
- IBM: `https://www.ibm.com/think/topics/regularization#:~:text=Regularization%20is%20a%20set%20of,for%20an%20increase%20in%20generalizability.`
- Medium : `https://medium.com/game-of-bits/optimizing-tensorflow-models-using-quant`

8 Appendix A. Regularization Result Images

8.1 Baseline Accuracy

```
history_model_basic = model_basic.fit(
    train_generator,
    epochs=10,
    validation_data=val_generator,
)
```

[9] ✓ 74m 14.1s Python

```
Epoch 1/10
/home/kast/m-eng-robotics/embeded-systems/thd-mro-em-labs/.venv/lib/python3.10/site-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121: U
  self._warn_if_super_not_called()
625/625 ━━━━━━━━━━━━━━━━━━━━ 527s 833ms/step - accuracy: 0.5696 - loss: 0.6876 - val_accuracy: 0.7514 - val_loss: 0.5181
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 794s 1s/step - accuracy: 0.7697 - loss: 0.4836 - val_accuracy: 0.8014 - val_loss: 0.4421
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 511s 818ms/step - accuracy: 0.8238 - loss: 0.3872 - val_accuracy: 0.8144 - val_loss: 0.4040
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 364s 580ms/step - accuracy: 0.8616 - loss: 0.3145 - val_accuracy: 0.8268 - val_loss: 0.3970
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 346s 554ms/step - accuracy: 0.9018 - loss: 0.2329 - val_accuracy: 0.8084 - val_loss: 0.4655
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 482s 771ms/step - accuracy: 0.9450 - loss: 0.1463 - val_accuracy: 0.8352 - val_loss: 0.4960
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 367s 587ms/step - accuracy: 0.9707 - loss: 0.0801 - val_accuracy: 0.8256 - val_loss: 0.6622
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 361s 578ms/step - accuracy: 0.9886 - loss: 0.0367 - val_accuracy: 0.8282 - val_loss: 0.7130
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 325s 520ms/step - accuracy: 0.9879 - loss: 0.0361 - val_accuracy: 0.8158 - val_loss: 0.8807
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 377s 603ms/step - accuracy: 0.9886 - loss: 0.0347 - val_accuracy: 0.8136 - val_loss: 1.0118
```

Figure 3: Training Process of Classic CNN

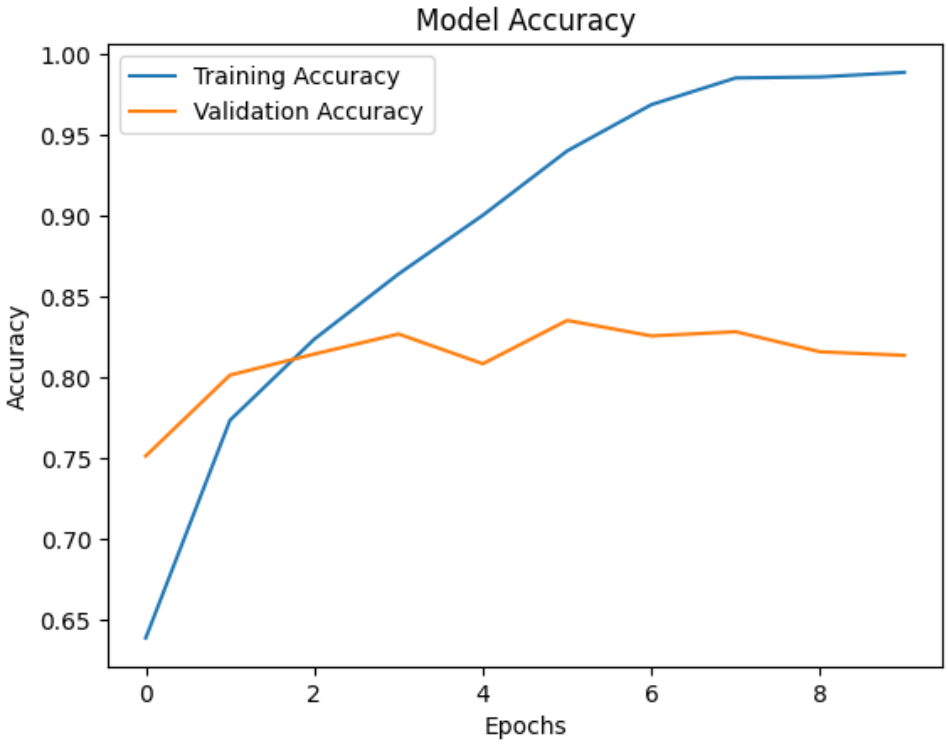

Figure 4: Training and Validation Accuracy over Epochs

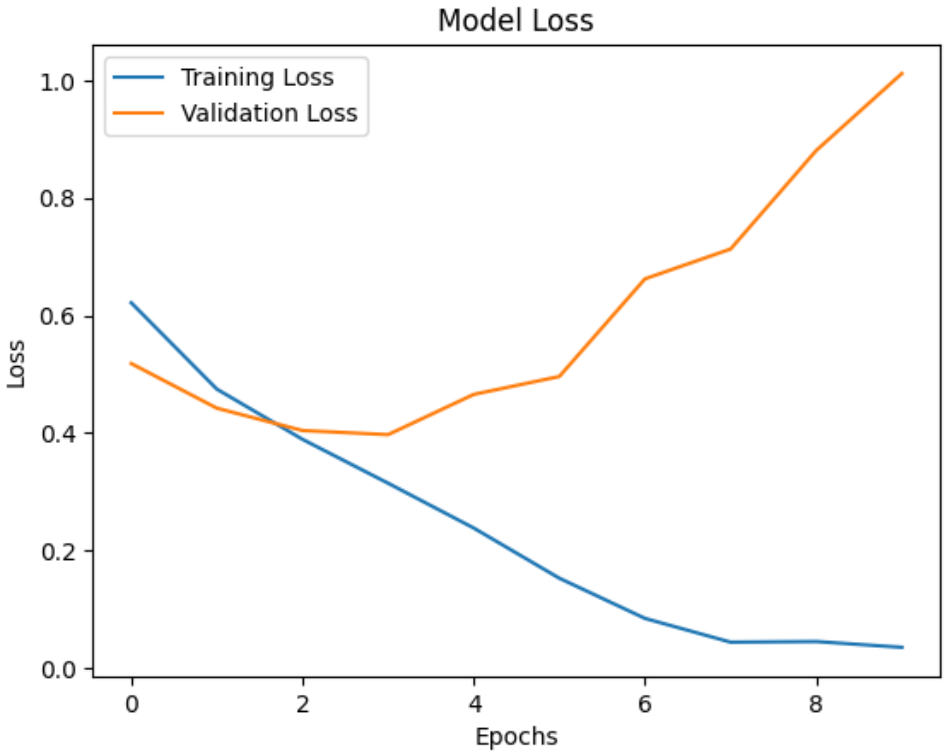

Figure 5: Training and Validation Loss over Epochs

8.2 Dropout Regularization

```
history_model_dropout = model_dropout.fit(
    train_generator,
    epochs=10,
    validation_data=val_generator
)
```

✓ 92m 17.8s Python

```
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 531s 845ms/step - accuracy: 0.5562 - loss: 0.7100 - val_accuracy: 0.6618 - val_loss: 0.6337
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 407s 651ms/step - accuracy: 0.6747 - loss: 0.6073 - val_accuracy: 0.6988 - val_loss: 0.5516
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 426s 682ms/step - accuracy: 0.7471 - loss: 0.5182 - val_accuracy: 0.7954 - val_loss: 0.4533
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 433s 692ms/step - accuracy: 0.7980 - loss: 0.4366 - val_accuracy: 0.8052 - val_loss: 0.4295
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 747s 1s/step - accuracy: 0.8308 - loss: 0.3852 - val_accuracy: 0.8138 - val_loss: 0.4160
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 591s 946ms/step - accuracy: 0.8515 - loss: 0.3375 - val_accuracy: 0.8208 - val_loss: 0.3997
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 656s 1s/step - accuracy: 0.8740 - loss: 0.2892 - val_accuracy: 0.8320 - val_loss: 0.3816
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 617s 988ms/step - accuracy: 0.8954 - loss: 0.2509 - val_accuracy: 0.8284 - val_loss: 0.4199
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 570s 911ms/step - accuracy: 0.9151 - loss: 0.2060 - val_accuracy: 0.8438 - val_loss: 0.4282
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 559s 895ms/step - accuracy: 0.9326 - loss: 0.1732 - val_accuracy: 0.8360 - val_loss: 0.4148
```

Figure 6: Training Process of Classic CNN

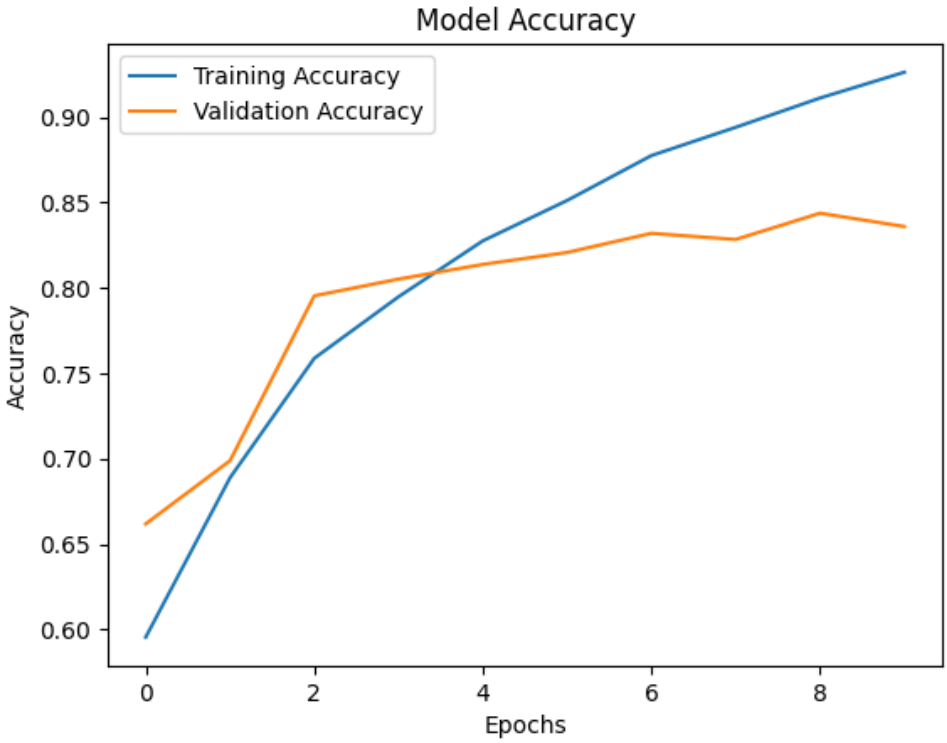

Figure 7: Training and Validation Accuracy over Epochs

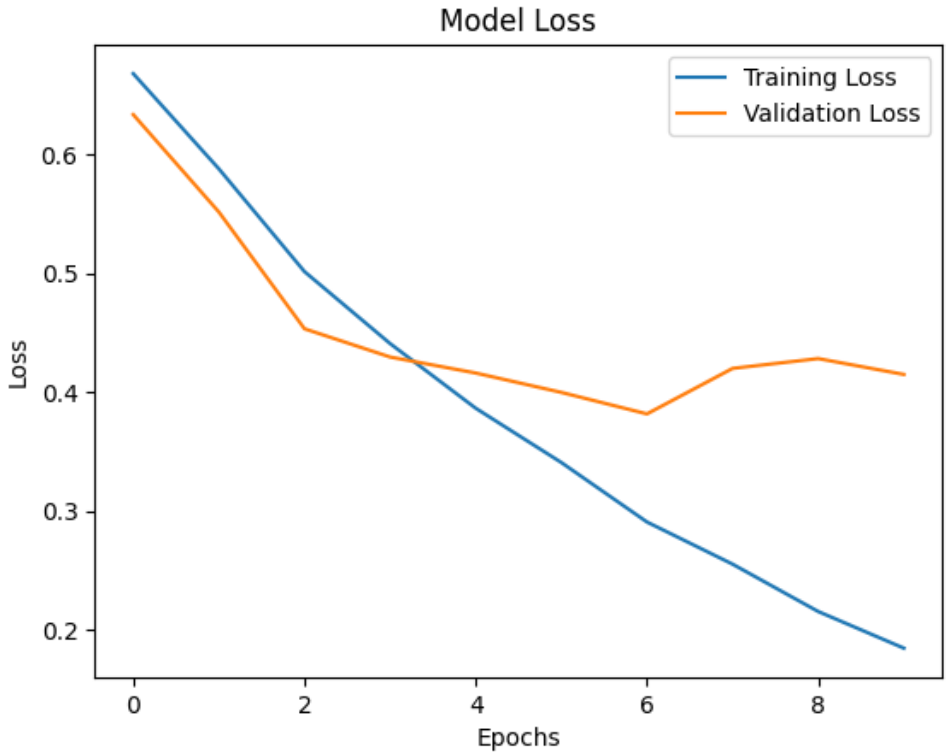

Figure 8: Training and Validation Loss over Epochs

8.3 L2 Regularization

```
model_l2.compile(optimizer='adam',
                 loss='binary_crossentropy',
                 metrics=['accuracy'])

history_l2 = model_l2.fit(
    train_generator,
    epochs=10,
    validation_data=val_generator
)
```

✓ 52m 56.3s Python

```
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 418s 666ms/step - accuracy: 0.5755 - loss: 0.7748 - val_accuracy: 0.7352 - val_loss: 0.5846
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 397s 636ms/step - accuracy: 0.7205 - loss: 0.5874 - val_accuracy: 0.7518 - val_loss: 0.5581
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 350s 560ms/step - accuracy: 0.7681 - loss: 0.5284 - val_accuracy: 0.7936 - val_loss: 0.4983
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 183s 293ms/step - accuracy: 0.7904 - loss: 0.4909 - val_accuracy: 0.7996 - val_loss: 0.4892
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 183s 293ms/step - accuracy: 0.8051 - loss: 0.4796 - val_accuracy: 0.7940 - val_loss: 0.5028
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 343s 549ms/step - accuracy: 0.8164 - loss: 0.4656 - val_accuracy: 0.7896 - val_loss: 0.5110
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 367s 588ms/step - accuracy: 0.8310 - loss: 0.4411 - val_accuracy: 0.8142 - val_loss: 0.4689
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 373s 596ms/step - accuracy: 0.8388 - loss: 0.4274 - val_accuracy: 0.8046 - val_loss: 0.4871
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 379s 606ms/step - accuracy: 0.8452 - loss: 0.4210 - val_accuracy: 0.8130 - val_loss: 0.4909
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 183s 292ms/step - accuracy: 0.8521 - loss: 0.4049 - val_accuracy: 0.8296 - val_loss: 0.4540
```

Figure 9: Training Process of Classic CNN

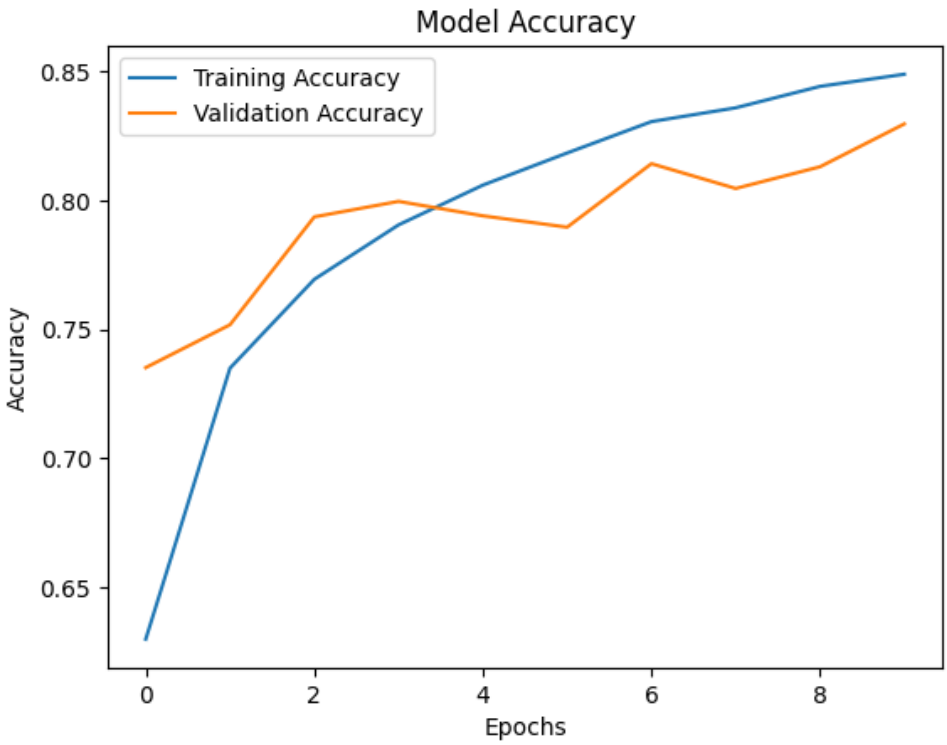

Figure 10: Training and Validation Accuracy over Epochs

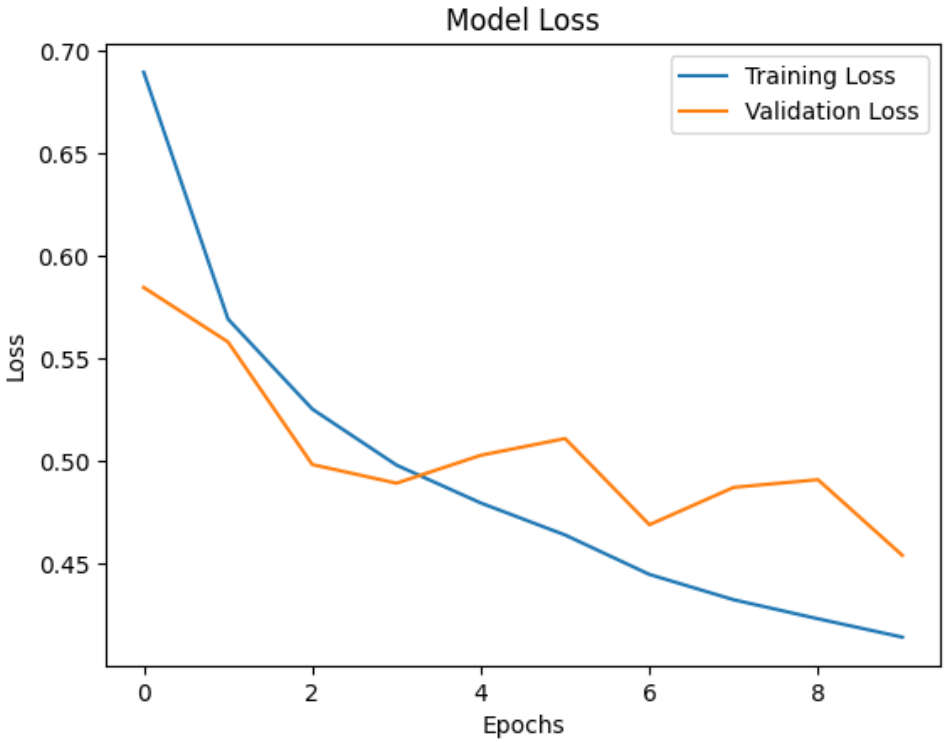

Figure 11: Training and Validation Loss over Epochs

8.4 Combined Regularization - Configuration 1

```
history_combined_config1 = model_combined_config1.fit(
    train_generator,
    epochs=10,
    validation_data=val_generator
)
```

✓ 55m 16.5s Python

```
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 413s 658ms/step - accuracy: 0.4983 - loss: 0.8732 - val_accuracy: 0.5762 - val_loss: 0.7146
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 404s 647ms/step - accuracy: 0.5600 - loss: 0.7085 - val_accuracy: 0.5806 - val_loss: 0.6890
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 379s 606ms/step - accuracy: 0.5684 - loss: 0.6920 - val_accuracy: 0.5888 - val_loss: 0.6809
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 370s 592ms/step - accuracy: 0.5753 - loss: 0.6855 - val_accuracy: 0.6026 - val_loss: 0.6741
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 203s 325ms/step - accuracy: 0.5818 - loss: 0.6817 - val_accuracy: 0.6214 - val_loss: 0.6632
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 200s 321ms/step - accuracy: 0.5679 - loss: 0.6801 - val_accuracy: 0.6342 - val_loss: 0.6531
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 237s 380ms/step - accuracy: 0.5995 - loss: 0.6662 - val_accuracy: 0.6310 - val_loss: 0.6473
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 367s 588ms/step - accuracy: 0.6063 - loss: 0.6620 - val_accuracy: 0.6444 - val_loss: 0.6354
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 384s 615ms/step - accuracy: 0.6124 - loss: 0.6556 - val_accuracy: 0.6316 - val_loss: 0.6362
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 358s 572ms/step - accuracy: 0.6193 - loss: 0.6475 - val_accuracy: 0.6642 - val_loss: 0.6251
```

Figure 12: Training Process of Classic CNN

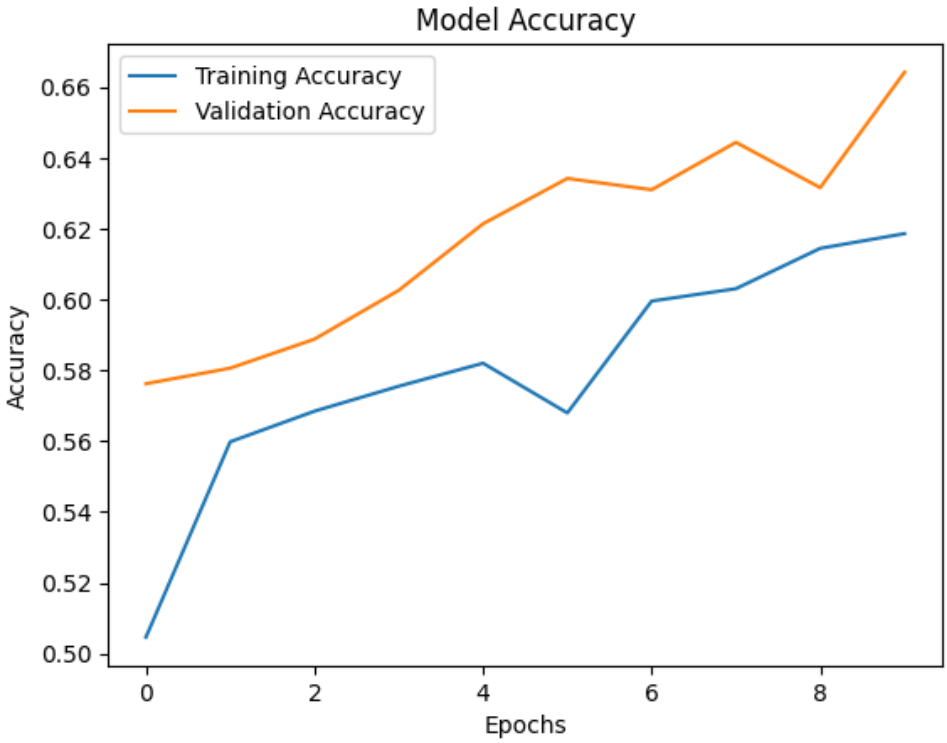

Figure 13: Training and Validation Accuracy over Epochs

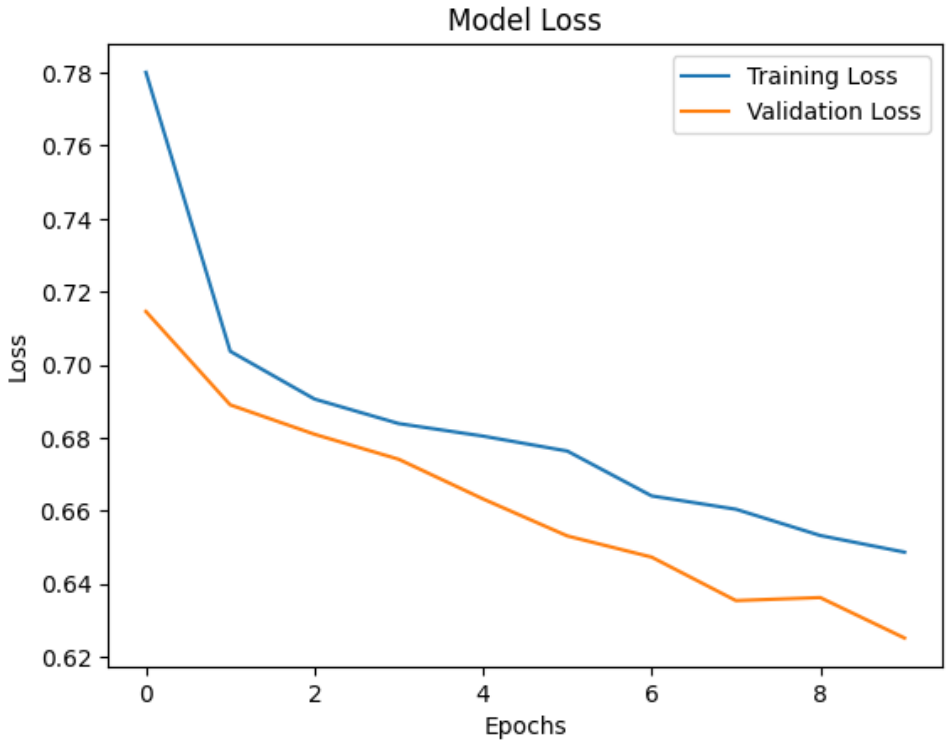

Figure 14: Training and Validation Loss over Epochs

8.5 Combined Regularization - Configuration 2

```
history_combined_config2 = model_combined_config2.fit(
    train_generator,
    epochs=10,
    validation_data=val_generator
)
```

✓ 48m 43.6s Python

```
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 292s 466ms/step - accuracy: 0.5565 - loss: 0.8149 - val_accuracy: 0.6586 - val_loss: 0.6370
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 380s 608ms/step - accuracy: 0.6747 - loss: 0.6325 - val_accuracy: 0.7066 - val_loss: 0.5844
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 292s 468ms/step - accuracy: 0.7175 - loss: 0.5875 - val_accuracy: 0.7172 - val_loss: 0.5846
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 269s 430ms/step - accuracy: 0.7370 - loss: 0.5614 - val_accuracy: 0.7672 - val_loss: 0.5207
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 276s 440ms/step - accuracy: 0.7667 - loss: 0.5256 - val_accuracy: 0.7520 - val_loss: 0.5402
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 280s 448ms/step - accuracy: 0.7774 - loss: 0.5160 - val_accuracy: 0.7794 - val_loss: 0.5083
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 286s 458ms/step - accuracy: 0.7850 - loss: 0.5018 - val_accuracy: 0.7924 - val_loss: 0.4925
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 287s 458ms/step - accuracy: 0.7976 - loss: 0.4884 - val_accuracy: 0.7826 - val_loss: 0.5104
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 278s 445ms/step - accuracy: 0.8067 - loss: 0.4753 - val_accuracy: 0.7938 - val_loss: 0.4946
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 284s 453ms/step - accuracy: 0.8077 - loss: 0.4830 - val_accuracy: 0.8094 - val_loss: 0.4809
```

Figure 15: Training Process of Classic CNN

Figure 16: Training and Validation Accuracy over Epochs

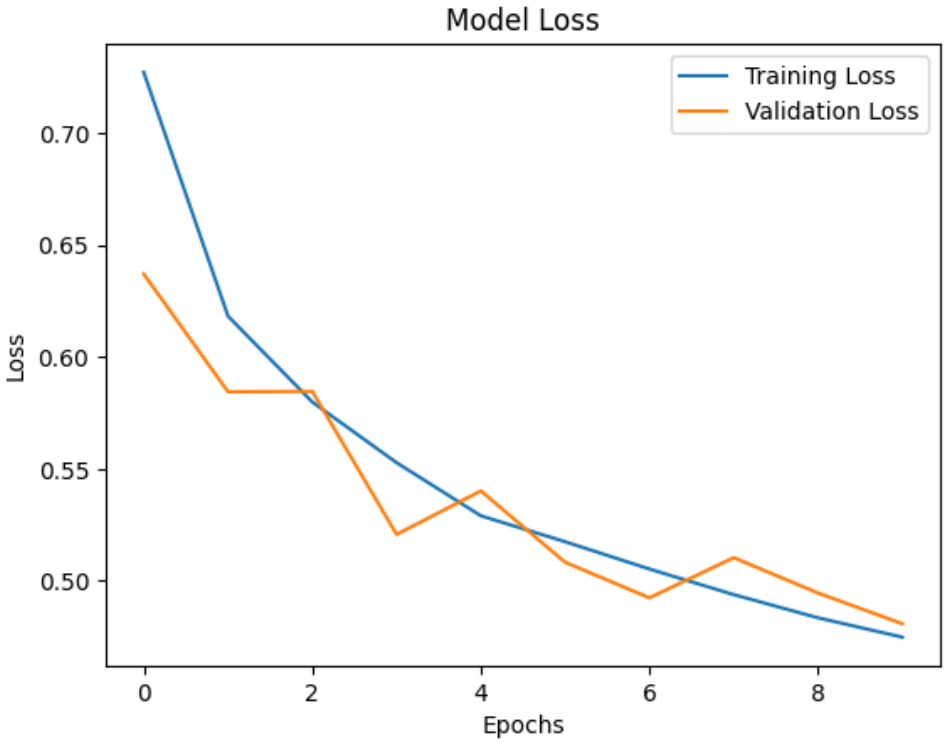

Figure 17: Training and Validation Loss over Epochs

8.6 Early Stopping

```
history_model_early_stop = model_early_stop.fit(
    train_generator,
    epochs=10,
    validation_data=val_generator,
    callbacks=[early_stop]
)
```

✓ 56m 2.3s Python

```
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 425s 678ms/step - accuracy: 0.6066 - loss: 0.6589 - val_accuracy: 0.7450 - val_loss: 0.5213
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 461s 737ms/step - accuracy: 0.7733 - loss: 0.4719 - val_accuracy: 0.7664 - val_loss: 0.4786
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 478s 766ms/step - accuracy: 0.8303 - loss: 0.3770 - val_accuracy: 0.8144 - val_loss: 0.4186
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 460s 737ms/step - accuracy: 0.8707 - loss: 0.2972 - val_accuracy: 0.8278 - val_loss: 0.4035
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 487s 780ms/step - accuracy: 0.9207 - loss: 0.2019 - val_accuracy: 0.8286 - val_loss: 0.4436
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 508s 814ms/step - accuracy: 0.9616 - loss: 0.1036 - val_accuracy: 0.8196 - val_loss: 0.5918
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 542s 868ms/step - accuracy: 0.9776 - loss: 0.0628 - val_accuracy: 0.8252 - val_loss: 0.7125
```

Figure 18: Training Process of Classic CNN

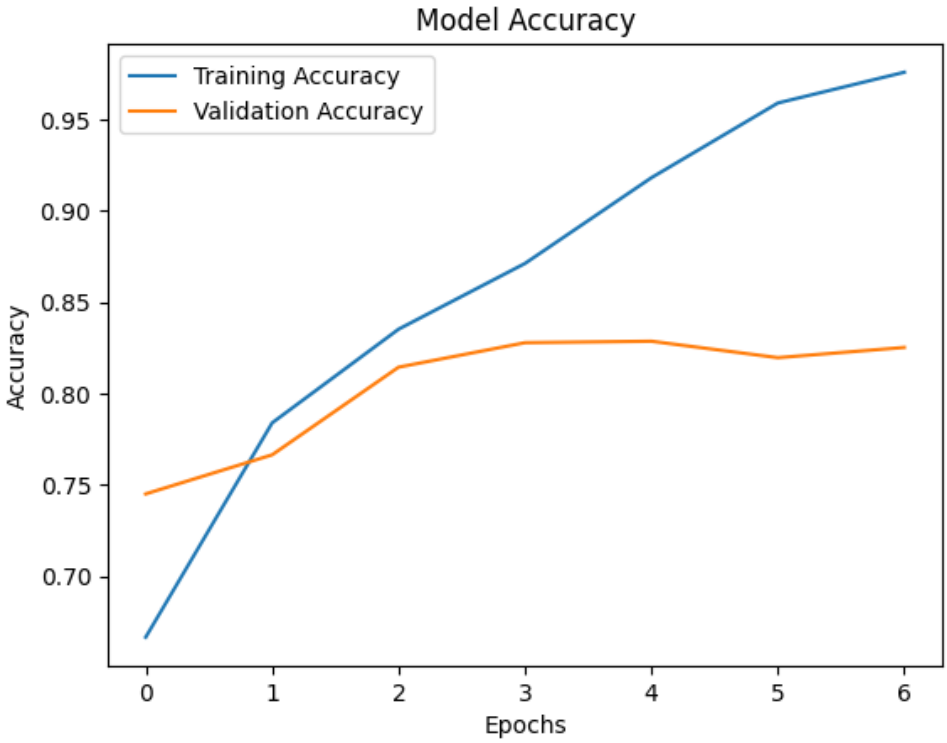

Figure 19: Training and Validation Accuracy over Epochs

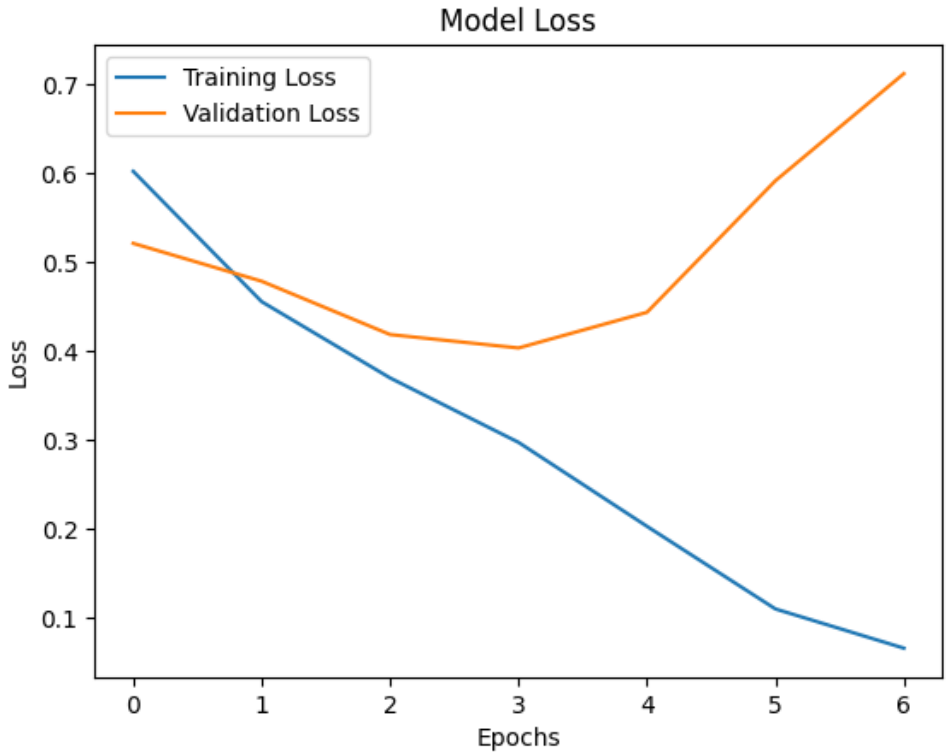

Figure 20: Training and Validation Loss over Epochs