AIN433 - Report on Computer Vision Lab Assignment 4

Student Name:	Emre Çoban
Student ID:	2200765028
Date:	January 4, 2024
Email:	b2200765028@cs.hacettepe.edu.tr

1 Introduction

In the realm of computer vision, a significant impact has been made by something known as Convolutional Neural Networks (CNNs) in altering the way images are classified. In this project, two approaches to image classification are explored, resembling an adventure.

Three main steps will be undergone. Initially, the recognition of objects in pictures using the MIT Indoor Scenes dataset will be facilitated by creating a CNN. Subsequently, an assessment will be made on its performance under various settings, such as different learning rates and batch sizes.

Following that, consideration will be given to the utilization of a pre-trained VGG-16 network, which will be adjusted to enhance its understanding of our dataset. This section aims to determine whether using a pre-trained network surpasses the efficacy of creating one from scratch.

Lastly, a shift will be made towards the recognition and localization of objects in pictures, employing a dataset featuring raccoons. This involves enabling the computer to comprehend the location of objects in a picture. The model will be fine-tuned, and adjustments will be made to its learning process, with subsequent evaluation of its performance.

This project is not solely about the utilization of sophisticated tools like TensorFlow. It encompasses a thoughtful consideration of the performance of our models and the lessons we can derive from each stage. The subsequent sections will narrate the story of the actions taken, how they were executed, and the discoveries made.

2 Data Preparation For Part 1 and Part 2

The dataset utilized in both Part 1 and Part 2 comprises 67 classes and 15,670 images. For this assignment, a subset of 15 classes, encompassing the highest image counts, was selected. Subsequently, 70% of these images were allocated for training, 15% for validation, and the remaining 15% for testing. The training set consists of 5,387 images, the validation set includes 1,148 images, and the test set comprises 1,171 images.

To facilitate the creation of datasets based on the specified parameters, a function named `create_datasets` was developed. This function takes the `batch_size` as a parameter and generates datasets accordingly. Data augmentation techniques were implemented during the creation of the training dataset to prevent overfitting and enhance the model's generalization to unseen data. These augmentation methods encompass random horizontal flips, random zooming, random rotations, among others.

Furthermore, it is noteworthy that all images in the datasets were resized to a standard size of 224 by 224 pixels, which is a commonly adopted size in various models to ensure compatibility and uniformity during the training process.

Example usage of `create_datasets` function

```
train_dataset, val_dataset, test_dataset = create_datasets(batch_size=16)
```

3 Part 1 : Modeling and Training a CNN classifier from Scratch

3.1 Custom Model Architecture

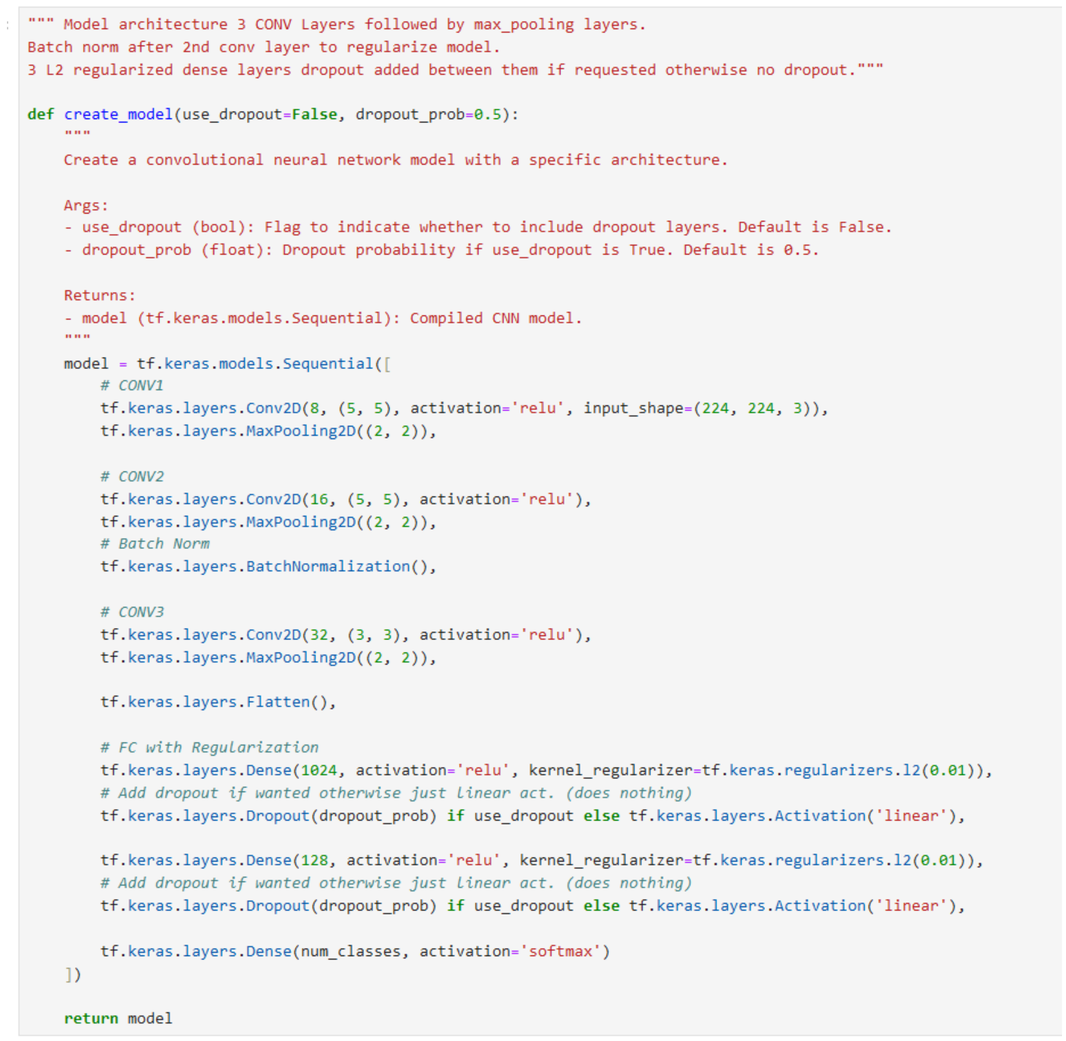
In Part 1, a Convolutional Neural Network (CNN) model was meticulously crafted from scratch using the TensorFlow framework. The model architecture, illustrated in Figure 2, featured three Convolution blocks. The first two Convolution blocks utilized a 5x5 kernel size, while the last Convolution block employed a 3x3 kernel size. A stride of 1 and the default valid padding in TensorFlow were applied, ensuring a comprehensive extraction of spatial features from the input data.

Each Convolution block comprised a convolution layer followed by a max-pooling layer. The utilization of a larger kernel size in the initial blocks facilitated the capturing of global patterns and features, while the adoption of a smaller kernel size in the final block allowed for more detailed, localized feature extraction.

Following the second convolution block, batch normalization was applied to enhance model generalization. Batch normalization contributes to stable and accelerated training by normalizing the inputs to a layer, thereby assisting in mitigating internal covariate shift.

The model also incorporated four dense layers with 20,000, 1,024, 128, and 15 units, aligned with the output shape of the data. The modular approach, ReLU activation, batch normalization, and L2 Kernel regularizers were introduced to the architecture, making dropout regularization an optional feature. This flexibility enables experimentation with dropout layers between dense layers, as demonstrated in the second experiment.

The code snippet for the `create_model` function is presented here, showcasing the optional creation of dropout layers between dense layers, as can be observed in Figure 1.



```python
""" Model architecture 3 CONV Layers followed by max_pooling layers.
Batch norm after 2nd conv layer to regularize model.
3 L2 regularized dense layers dropout added between them if requested otherwise no dropout."""

def create_model(use_dropout=False, dropout_prob=0.5):
    """
    Create a convolutional neural network model with a specific architecture.

    Args:
    - use_dropout (bool): Flag to indicate whether to include dropout layers. Default is False.
    - dropout_prob (float): Dropout probability if use_dropout is True. Default is 0.5.

    Returns:
    - model (tf.keras.models.Sequential): Compiled CNN model.
    """
    model = tf.keras.models.Sequential([
        # CONV1
        tf.keras.layers.Conv2D(8, (5, 5), activation='relu', input_shape=(224, 224, 3)),
        tf.keras.layers.MaxPooling2D((2, 2)),

        # CONV2
        tf.keras.layers.Conv2D(16, (5, 5), activation='relu'),
        tf.keras.layers.MaxPooling2D((2, 2)),
        # Batch Norm
        tf.keras.layers.BatchNormalization(),

        # CONV3
        tf.keras.layers.Conv2D(32, (3, 3), activation='relu'),
        tf.keras.layers.MaxPooling2D((2, 2)),

        tf.keras.layers.Flatten(),

        # FC with Regularization
        tf.keras.layers.Dense(1024, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.01)),
        # Add dropout if wanted otherwise just linear act. (does nothing)
        tf.keras.layers.Dropout(dropout_prob) if use_dropout else tf.keras.layers.Activation('linear'),

        tf.keras.layers.Dense(128, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.01)),
        # Add dropout if wanted otherwise just linear act. (does nothing)
        tf.keras.layers.Dropout(dropout_prob) if use_dropout else tf.keras.layers.Activation('linear'),

        tf.keras.layers.Dense(num_classes, activation='softmax')
    ])

    return model
```

Figure 1: Code Snippet For Creating Custom CNN Model

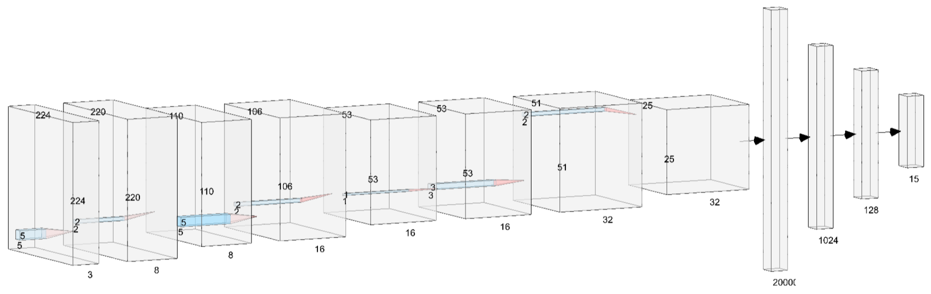


Figure 2: Custom CNN Model Architecture.

3.2 Training Details

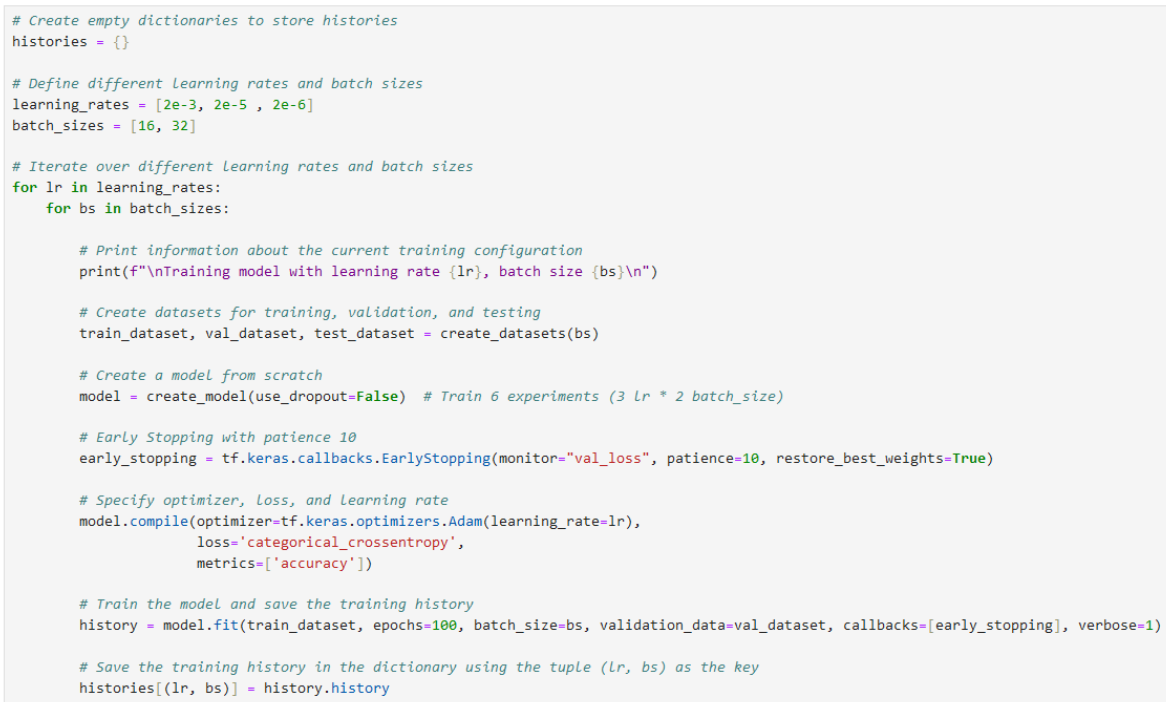
The models underwent training utilizing the Adam optimizer for a duration of 100 epochs. Categorical cross-entropy loss was employed as the chosen loss function, suitable for multi-class classification tasks. To enhance the training process and mitigate the risk of overfitting, an early stopping mechanism was incorporated. This mechanism actively monitored the validation loss, halting the training if there was no improvement in the validation loss for a consecutive period of 10 epochs. This approach aimed to strike a balance between achieving optimal model performance and preventing overfitting by dynamically adapting to the validation loss trends during the training phase.

3.3 Learning Rate and Batch Size Exploration

The first experiment encompassed training the model with three distinct learning rates (2e-3, 2e-5, 2e-6) and two varied batch sizes (16, 32). The results, as depicted in Figure 4, offered insightful glimpses into the training and validation performance. It was discerned that the model trained with a learning rate of 2e-5 and a batch size of 32 outperformed others, particularly in the validation dataset.

A detailed examination of the learning rate variations shed light on their impact on the training process. The learning rate of 2e-3 was observed to be too high, leading to oscillations and fluctuations in the loss curve, indicative of erratic model updates during training. On the contrary, a learning rate of 2e-6 was deemed excessively slow, resulting in a sluggish convergence of the model, characterized by a prolonged and gradual decrease in the loss. In contrast, the learning rate of 2e-5 exhibited a near-optimal balance, allowing for a steady convergence of the model with minimal oscillations. This observation motivated the selection of 2e-5 as the preferred learning rate, as it struck a desirable equilibrium between rapid convergence and stability, making it the prime choice for subsequent experiments.

The code snippet for the training loop for learning rate and batch size experiments is presented in Figure 3



```python
# Create empty dictionaries to store histories
histories = {}

# Define different learning rates and batch sizes
learning_rates = [2e-3, 2e-5 , 2e-6]
batch_sizes = [16, 32]

# Iterate over different learning rates and batch sizes
for lr in learning_rates:
    for bs in batch_sizes:

        # Print information about the current training configuration
        print(f"\nTraining model with learning rate {lr}, batch size {bs}\n")

        # Create datasets for training, validation, and testing
        train_dataset, val_dataset, test_dataset = create_datasets(bs)

        # Create a model from scratch
        model = create_model(use_dropout=False)  # Train 6 experiments (3 lr * 2 batch_size)

        # Early Stopping with patience 10
        early_stopping = tf.keras.callbacks.EarlyStopping(monitor="val_loss", patience=10, restore_best_weights=True)

        # Specify optimizer, loss, and learning rate
        model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=lr),
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])

        # Train the model and save the training history
        history = model.fit(train_dataset, epochs=100, batch_size=bs, validation_data=val_dataset, callbacks=[early_stopping], verbose=1)

        # Save the training history in the dictionary using the tuple (lr, bs) as the key
        histories[(lr, bs)] = history.history
```

Figure 3: Code Snippet For LR and Batch Size Experiments

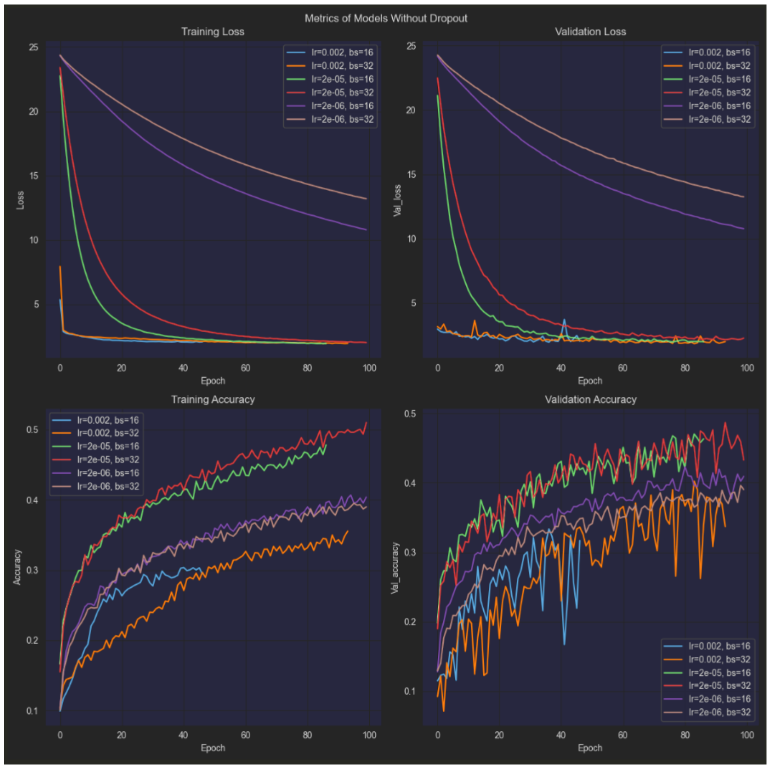


Figure 4: Experiment 1: Learning Rate and Batch Size Exploration.

3.4 Effect of Dropout Layers

In the second experiment, a thorough exploration of dropout, a regularization technique, was undertaken to discern its effect on the model's performance. Dropout is a regularization method that involves randomly deactivating a fraction of neurons during training, forcing the model to rely on a diverse set of features and preventing overfitting.

The `create_model` function played a pivotal role in introducing dropout layers strategically between dense layers. This design choice was inspired by the seminal paper authored by Hinton et al. (2012), where dropout, with a dropout probability of $p = 0.5$, was applied to each fully connected (dense) layer before the output layer. The strategic placement of dropout layers encourages the model to be more robust and less dependent on specific features, enhancing its generalization to unseen data.

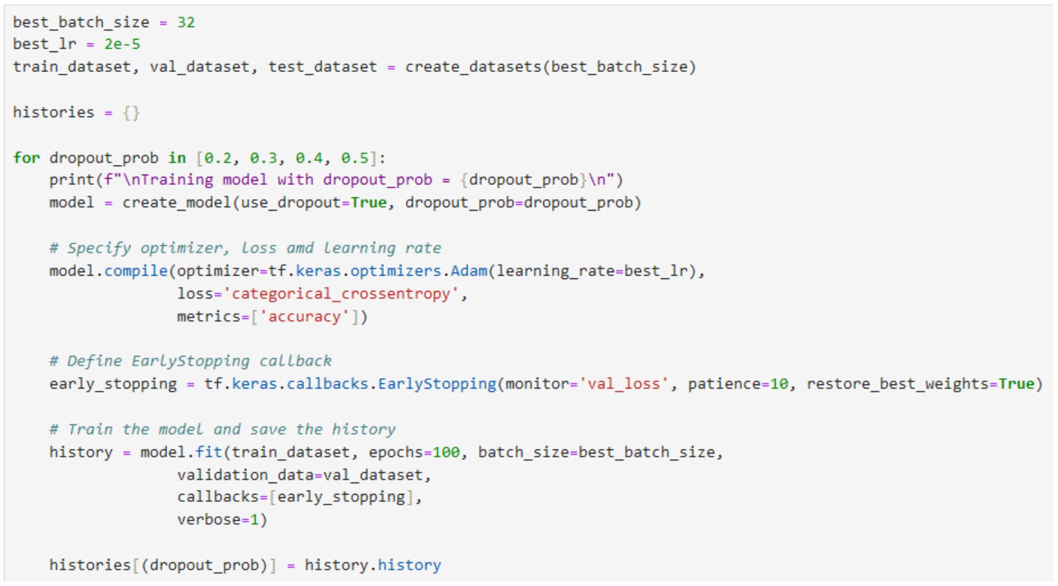
Figure 6 visually represents the outcomes of the experiment, shedding light on the regularization effect of dropout. Four dropout probabilities, namely 0.2, 0.3, 0.4, and 0.5, were tested to observe their impact on the model's performance.

Among these probabilities, dropout values of 0.2 and 0.3 exhibited promising results. The preference for a dropout probability of 0.3 emerged due to its striking balance between validation accuracy and loss.

The rationale behind selecting a specific dropout probability lies in achieving an equilibrium. A dropout value that is too high may lead to excessive information loss during training, hindering the model's capacity to learn essential features. Conversely, a dropout value that is too low may not provide sufficient regularization, potentially resulting in overfitting to the training data.

Upon careful examination of the experiment results, a dropout probability of 0.3 was chosen for further model refinement. This choice was informed by its superior validation accuracy and lower validation loss, indicating an effective regularization impact. The selected dropout probability of 0.3 contributes to a well-balanced model that not only performs well on the training data but also demonstrates a robust capability to generalize to new and unseen data.

The code snippet for the training loop for dropout experiments is presented in Figure 5



```python
best_batch_size = 32
best_lr = 2e-5
train_dataset, val_dataset, test_dataset = create_datasets(best_batch_size)

histories = {}

for dropout_prob in [0.2, 0.3, 0.4, 0.5]:
    print(f"\nTraining model with dropout_prob = {dropout_prob}\n")
    model = create_model(use_dropout=True, dropout_prob=dropout_prob)

    # Specify optimizer, loss amd learning rate
    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=best_lr),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

    # Define EarlyStopping callback
    early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)

    # Train the model and save the history
    history = model.fit(train_dataset, epochs=100, batch_size=best_batch_size,
                  validation_data=val_dataset,
                  callbacks=[early_stopping],
                  verbose=1)

    histories[(dropout_prob)] = history.history
```

Figure 5: Code Snippet For Dropout Experiments

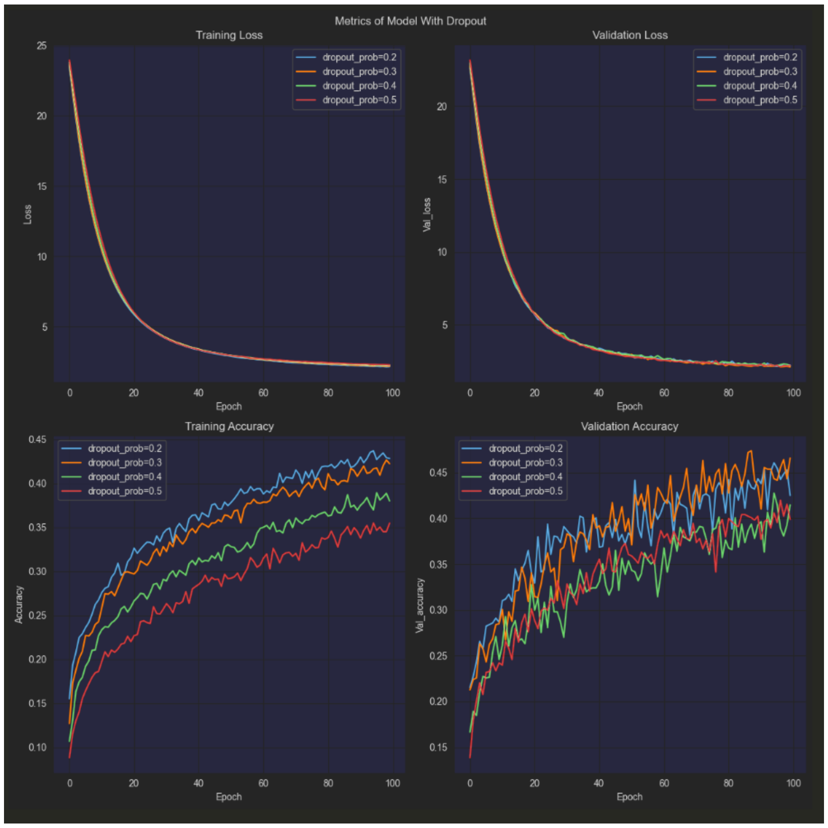


Figure 6: Experiment 2: Dropout Probabilities Investigation.

3.5 Training Final Best Model of Part 1

With the optimal hyperparameters selected, including a batch size of 32, learning rate of 2e-5, and dropout probability of 0.3, the final model underwent training and evaluation. The metrics for this best-performing model are illustrated in Figure 7, providing insights into its performance.

Additionally, Figure 8 showcases the confusion matrix for the test dataset, offering a detailed understanding of the model's classification performance across different classes. The model, after the comprehensive exploration and refinement in Part 1, achieved a commendable test accuracy of 41.8%. Moreover, the test loss, standing at 2.19, closely aligns with the validation accuracy and loss metrics, underscoring the robustness and generalization capability of the finalized CNN model.

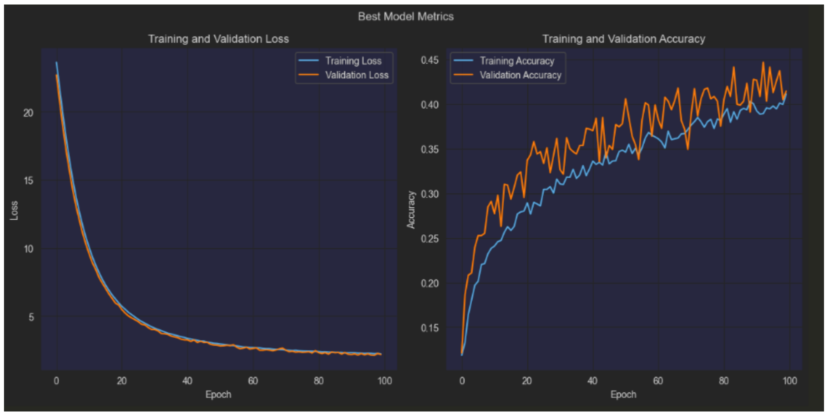


Figure 7: Final Model Metrics.

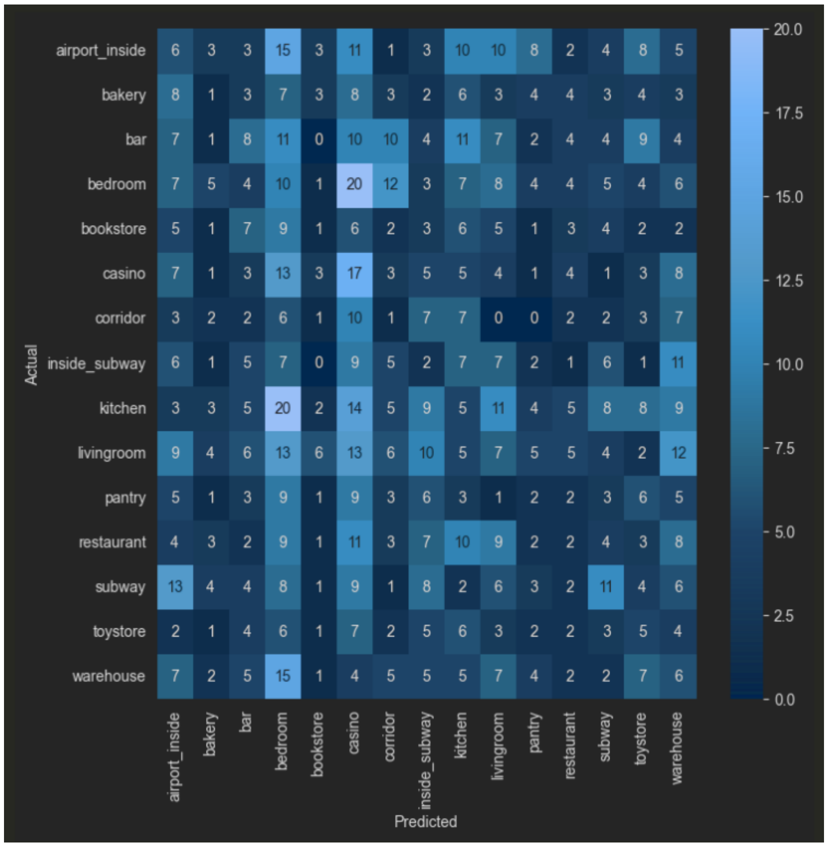


Figure 8: Confusion Matrix for the Test Dataset.

4 Part 2: Transfer Learning with CNNs

In this section, the application of transfer learning is explored using the pre-trained VGG-16 network available in TensorFlow. Transfer learning involves the utilization of knowledge gained from training on one task and its application to a different, but related, task. A strong foundation for feature extraction in the specific classification task at hand is provided by employing the VGG-16 model, initially trained on the ImageNet dataset.

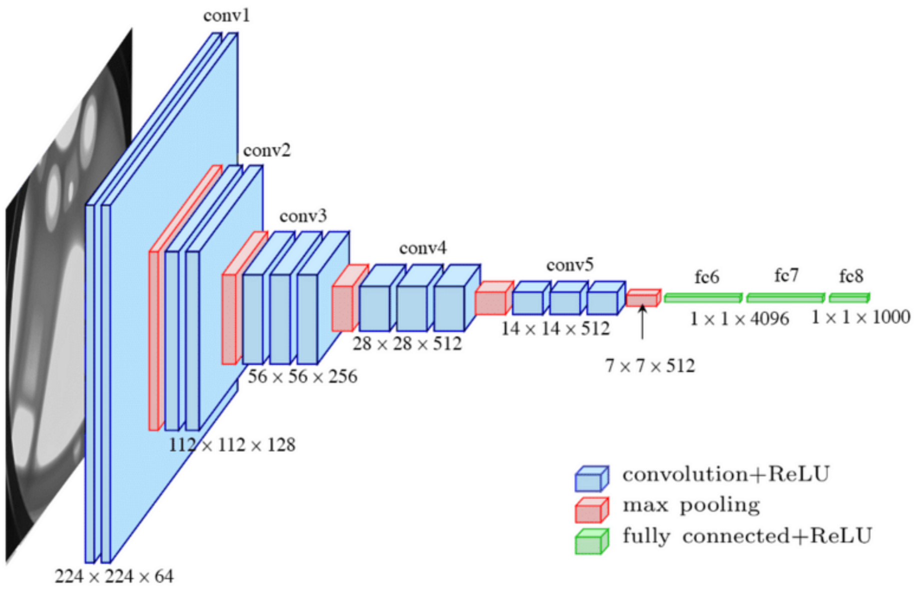


Figure 9: VGG-16 Model Architecture.

4.1 Fine-Tuning the VGG-16 Network

The process of fine-tuning involves adapting a pre-trained model to a specific task. In this case, the VGG-16 network is chosen as the base model, leveraging the learned features from ImageNet. The original VGG-16 architecture includes fully connected (FC) layers with 4096, 4096, and 1000 units. However, considering the dataset with 15 output classes, a decision was made to modify these layers to 2048, 256, and 15 units, respectively. This adjustment aligns the model more closely with the scale of the classification task, promoting better learning and generalization. L2 Regularization was applied to dense layers to prevent overfitting.

4.2 Rationale for Fine-Tuning

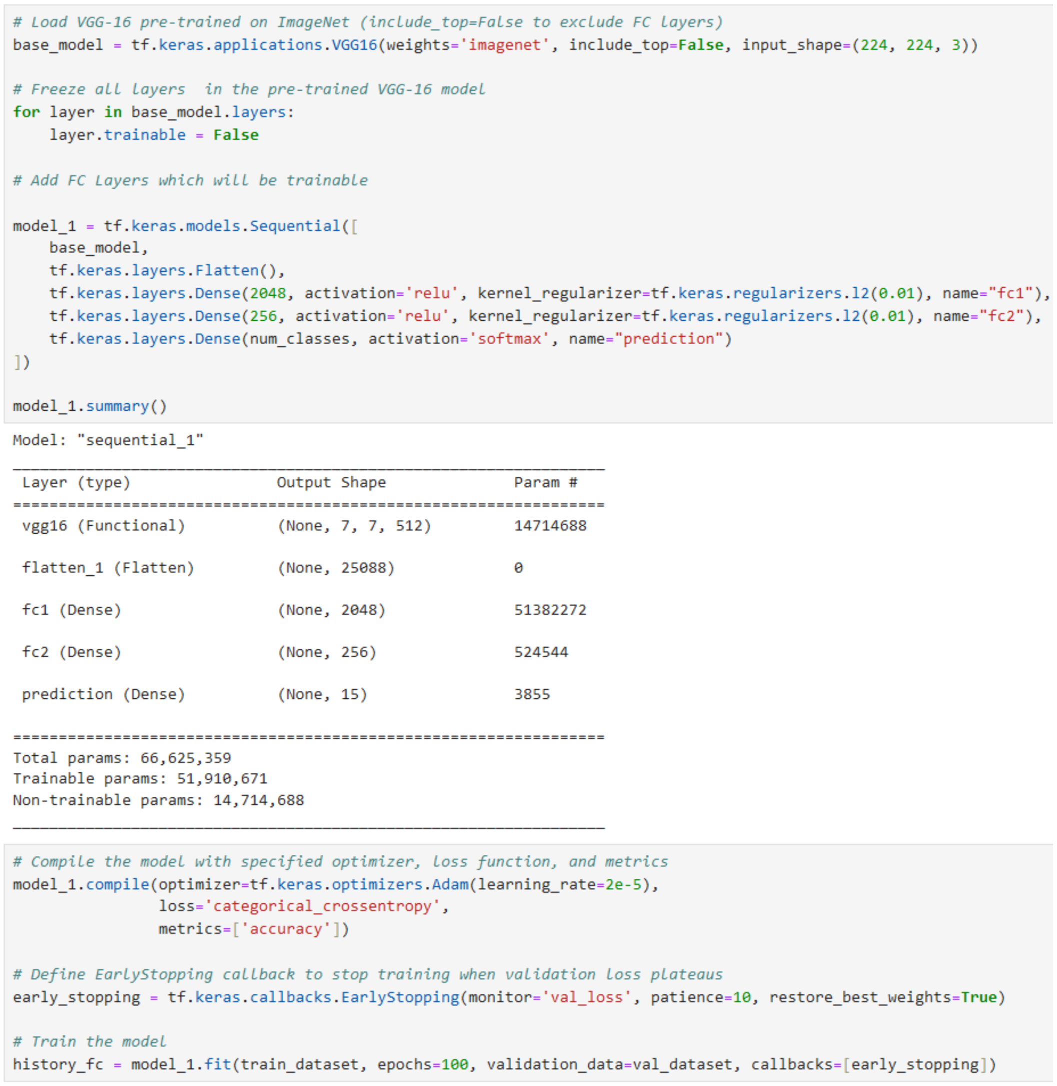
The application of fine-tuning becomes beneficial when a pre-trained model has already learned valuable hierarchical features from a vast dataset, such as ImageNet. This knowledge transfer accelerates the training process, enhancing the model's ability to generalize to our target task. By freezing the pre-trained layers, these valuable features are retained, with only the weights of the task-specific FC layers being updated, thereby optimizing the model for our classification task.

4.3 Exploring Different Training Cases

Two training scenarios are explored to evaluate the impact on model performance:

- **Case 1: Training Only FC Layers**

- In this scenario, only the FC layers are trained, while the rest of the layers remain frozen.As it can be seen in figure 10



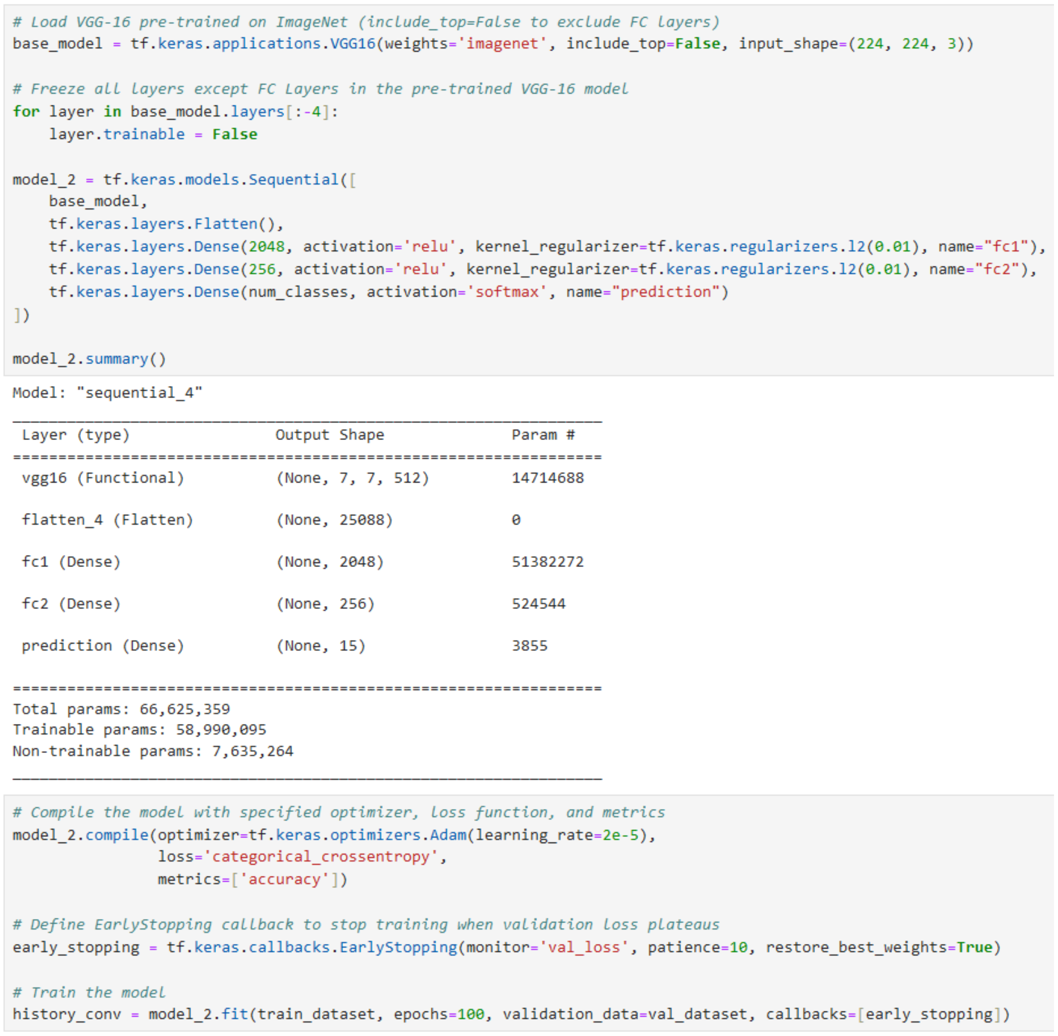
```
# Load VGG-16 pre-trained on ImageNet (include_top=False to exclude FC layers)
base_model = tf.keras.applications.VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

# Freeze all layers  in the pre-trained VGG-16 model
for layer in base_model.layers:
    layer.trainable = False

# Add FC Layers which will be trainable

model_1 = tf.keras.models.Sequential([
    base_model,
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(2048, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.01), name="fc1"),
    tf.keras.layers.Dense(256, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.01), name="fc2"),
    tf.keras.layers.Dense(num_classes, activation='softmax', name="prediction")
])

model_1.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 7, 7, 512)         14714688

 flatten_1 (Flatten)         (None, 25088)             0

 fc1 (Dense)                 (None, 2048)              51382272

 fc2 (Dense)                 (None, 256)               524544

 prediction (Dense)          (None, 15)                3855

=================================================================
Total params: 66,625,359
Trainable params: 51,910,671
Non-trainable params: 14,714,688
_________________________________________________________________
```

```
# Compile the model with specified optimizer, loss function, and metrics
model_1.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=2e-5),
                loss='categorical_crossentropy',
                metrics=['accuracy'])

# Define EarlyStopping callback to stop training when validation loss plateaus
early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)

# Train the model
history_fc = model_1.fit(train_dataset, epochs=100, validation_data=val_dataset, callbacks=[early_stopping])
```

Figure 10: Code Snippet For Model For FC Layers Fine Tuned VGG16

- **Case 2: Training Last Two Convolutional Layers and FC Layers**
  - Here, the last two convolutional layers and the FC layers are trained, while the remaining layers are frozen. As it can be seen in figure 11



```
# Load VGG-16 pre-trained on ImageNet (include_top=False to exclude FC layers)
base_model = tf.keras.applications.VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

# Freeze all layers except FC Layers in the pre-trained VGG-16 model
for layer in base_model.layers[:-4]:
    layer.trainable = False

model_2 = tf.keras.models.Sequential([
    base_model,
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(2048, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.01), name="fc1"),
    tf.keras.layers.Dense(256, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.01), name="fc2"),
    tf.keras.layers.Dense(num_classes, activation='softmax', name="prediction")
])

model_2.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 7, 7, 512)         14714688

 flatten_4 (Flatten)         (None, 25088)             0

 fc1 (Dense)                 (None, 2048)              51382272

 fc2 (Dense)                 (None, 256)               524544

 prediction (Dense)          (None, 15)                3855

=================================================================
Total params: 66,625,359
Trainable params: 58,990,095
Non-trainable params: 7,635,264
_________________________________________________________________
```

```
# Compile the model with specified optimizer, loss function, and metrics
model_2.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=2e-5),
                loss='categorical_crossentropy',
                metrics=['accuracy'])

# Define EarlyStopping callback to stop training when validation loss plateaus
early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)

# Train the model
history_conv = model_2.fit(train_dataset, epochs=100, validation_data=val_dataset, callbacks=[early_stopping])
```

Figure 11: Code Snippet For Model For 3 CONV + FC Layers Fine Tuned VGG16

The best-performing model will be selected at a later stage, after which it will undergo experimentation with two different batch sizes (16, 32) and three distinct learning rates (2e-4, 2e-5, 2e-6), mirroring the experimental setup employed in the initial part of the study. The model yielding the best performance will then be trained, and its performance will be assessed through a comparative analysis of the confusion matrix and test evaluation metrics against the best model identified in the initial part of the study.

4.4 Training Details

The models are optimized using the Adam optimizer with categorical cross-entropy loss. Training spans 100 epochs with early stopping (patience 10) on validation loss to prevent overfitting.

4.5 Comparison of Case 1 and Case 2

In Figure 12, it can be observed that the model fine-tuned with 3 convolutional layers + FC layers outperforms the model solely fine-tuned with FC layers, as anticipated. This is attributed to the additional fine-tuning of convolutional layers in Case 2, aligning the model more effectively with the characteristics of our dataset.

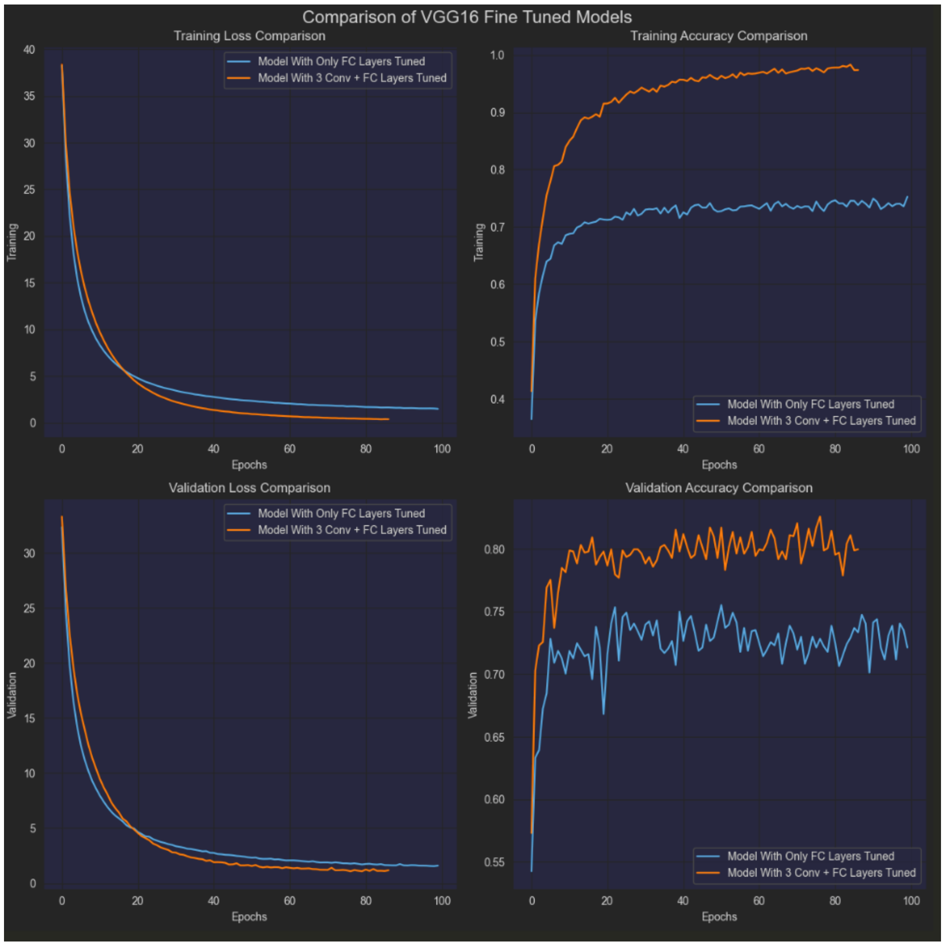


Figure 12: VGG-16 Case 1 and Case 2 Comparison

The superior performance of the model fine-tuned with 3 convolutional layers + FC layers, as evidenced in the validation data graphs (Figure 12), leads us to choose this model for further experimentation. The decision to favor Case 2 is driven by its better alignment with our dataset characteristics, particularly attributed to the additional fine-tuning of convolutional layers. This adaptation enhances the model's ability to capture intricate patterns and features within our specific classification task, resulting in improved validation performance.

4.6 Exploration of Learning Rates and Batch Sizes

After selecting the VGG16 fine-tuned model with 3 convolutional layers + FC layers fine-tuned, attention is directed towards conducting experiments to investigate the impact of different batch sizes and learning rates on the model's performance. The outcomes of this experiment are depicted in Figure 13.

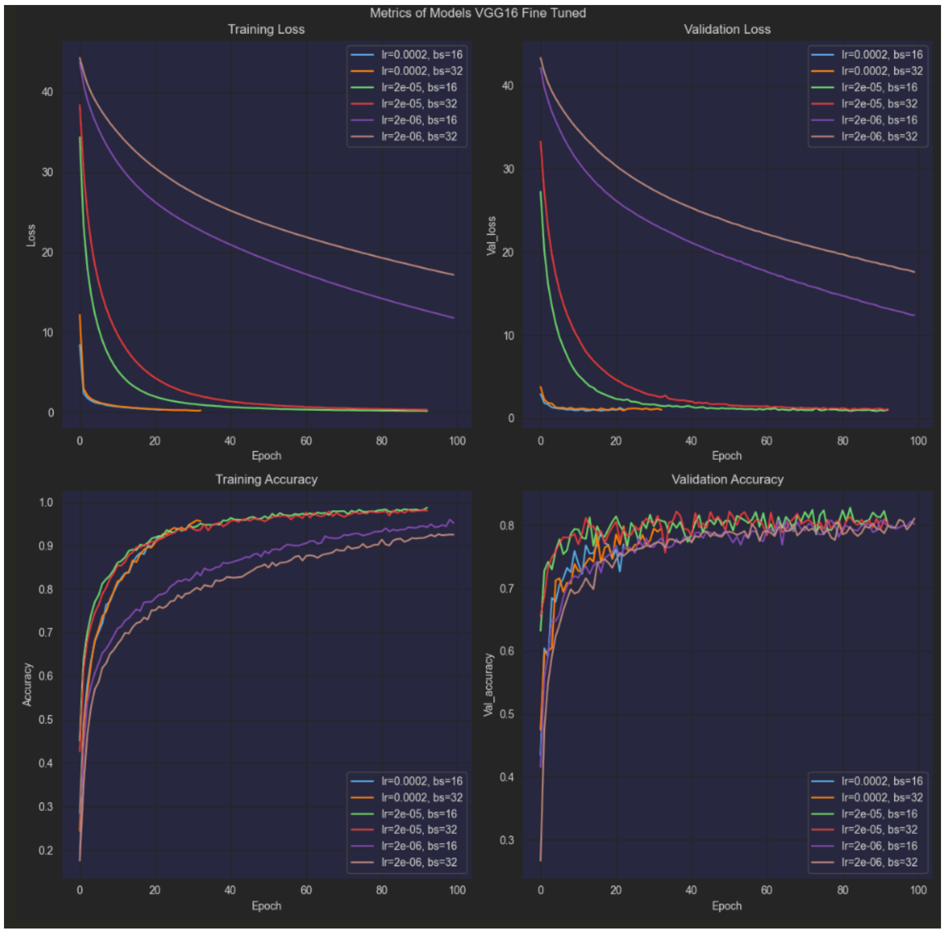


Figure 13: VGG-16 Batch Size and Learning Rate Experiments

From these experiments, it is observed that the model with a batch size of 16 and a learning rate of 2e-5 performs the best in terms of validation loss and accuracy. The learning rate of 2e-6 is deemed too slow, while the learning rate of 2e-4 is considered too fast, resulting in excessive oscillations.

4.7 Training and Evaluation of Best Model in Part 2

The VGG16 model fine-tuned with 3 convolutional layers + FC layers has been selected. Subsequently, optimization steps were undertaken by choosing the best-performing batch size (16) and learning rate (2e-5). In this phase, the model undergoes the training process and is evaluated using the test dataset.

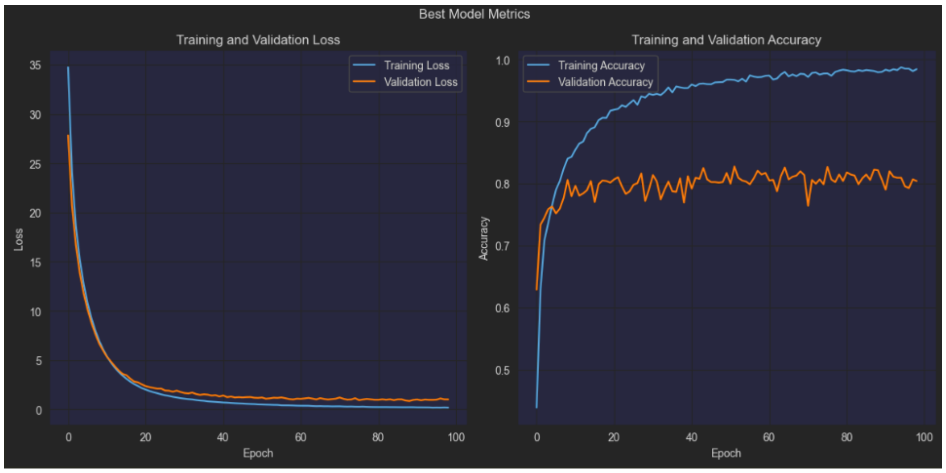


Figure 14: Training and Validation Metrics of the Best Model

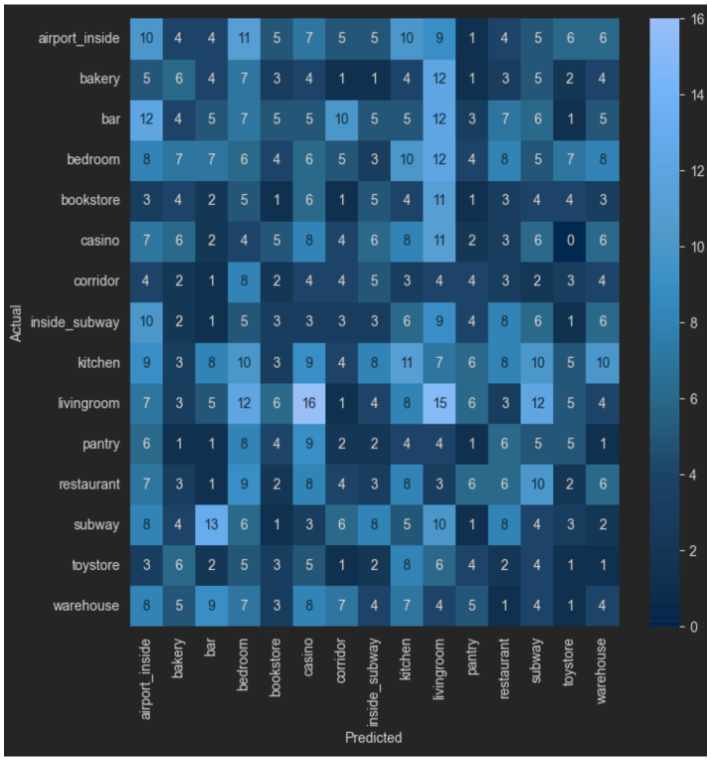


Figure 15: Confusion Matrix of the Best Model on Test Dataset

The metrics of the best model, as depicted in Figure 14, showcase the training and validation losses along with accuracies. Additionally, the confusion matrix in Figure 15 illustrates the performance on the test dataset. The test accuracy and loss of the model are also noted for comprehensive evaluation. The test accuracy is reported to be 0.798, while the test loss stands at 0.90, demonstrating consistency with the validation performance.

4.8 Comparison of Models From Part 1 and Part 2

In this section, we delve into a qualitative analysis of the performance comparison between the models developed in Part 1 (custom CNN) and Part 2 (VGG-16 fine-tuned). This examination aims to provide insights into the factors contributing to the observed differences in accuracies and losses.

4.8.1 Advantages of VGG-16 Fine-Tuning

The VGG-16 fine-tuned model from Part 2 exhibits notable advantages over the custom CNN models created in Part 1. Several factors contribute to this superior performance:

- **Transfer Learning Benefits:** VGG-16, pre-trained on the ImageNet dataset, brings valuable knowledge of hierarchical features, which proves beneficial for feature extraction in our specific classification task. This knowledge accelerates the training process and enhances the model's ability to generalize to our target task.
- **Complexity and Capacity:** VGG-16 is a deep and complex architecture with a large number of parameters. This complexity allows the model to capture intricate patterns and representations in the data, potentially leading to better generalization.
- **Adaptability:** Fine-tuning the VGG-16 model by modifying the fully connected layers to align with our dataset's scale ensures better adaptation to the specific characteristics of our classification task. The flexibility of adjusting these layers contributes to improved learning and generalization.
- **Hierarchical Feature Extraction:** The hierarchical feature extraction capabilities of VGG-16, learned during its training on ImageNet, enable the model to identify and leverage complex patterns in the dataset, enhancing its performance in comparison to the custom CNN models.

4.8.2 Test Dataset Evaluation

The real-world performance of the models is assessed by evaluating them on the test dataset. Notably, the Part 2 model, which utilized VGG-16 fine-tuning, demonstrates a significantly higher accuracy of 0.79 and a lower loss of 0.90 on the test dataset. In contrast, the Part 1 model, built from scratch, exhibits a lower accuracy of 0.41 and a higher loss of 2.19. This performance disparity can be attributed to various factors, including the depth and complexity of the architectures, with the VGG-16 fine-tuned model benefiting from transfer learning and hierarchical feature extraction. Additionally, the custom Part 1 model, being relatively shallower, may struggle to capture intricate patterns in the data compared to the VGG-16 fine-tuned model.

5 Part 3: Object Classification and Localization

Object localization is a critical aspect of computer vision, involving the identification and precise localization of objects within an image through bounding boxes. This chapter focuses on implementing an object classification and localization system using the Raccoon dataset.

5.1 Preparing Data

The dataset has been partitioned into three subsets: 150 training images, 29 validation images, and 17 test images. Notably, certain images containing multiple raccoons were intentionally excluded, aligning with the inherent constraint of predicting only a single bounding box per image.

The `load_data` function facilitates the retrieval of image paths and bounding boxes from the annotations file. Subsequently, the `create_dataset` function, which accepts the batch size as a parameter, is employed to generate the training, validation, and test datasets.

Internally, the `create_dataset` function leverages the `load_and_preprocess_image` function. This function, taking image paths and corresponding bounding boxes as input, reads the image, resizes it to a 224x224 shape, and adjusts the bounding box accordingly to accommodate the resizing. This systematic process ensures that the dataset is appropriately read and preprocessed for utilization in the model.

Here is an example usage of these functions.

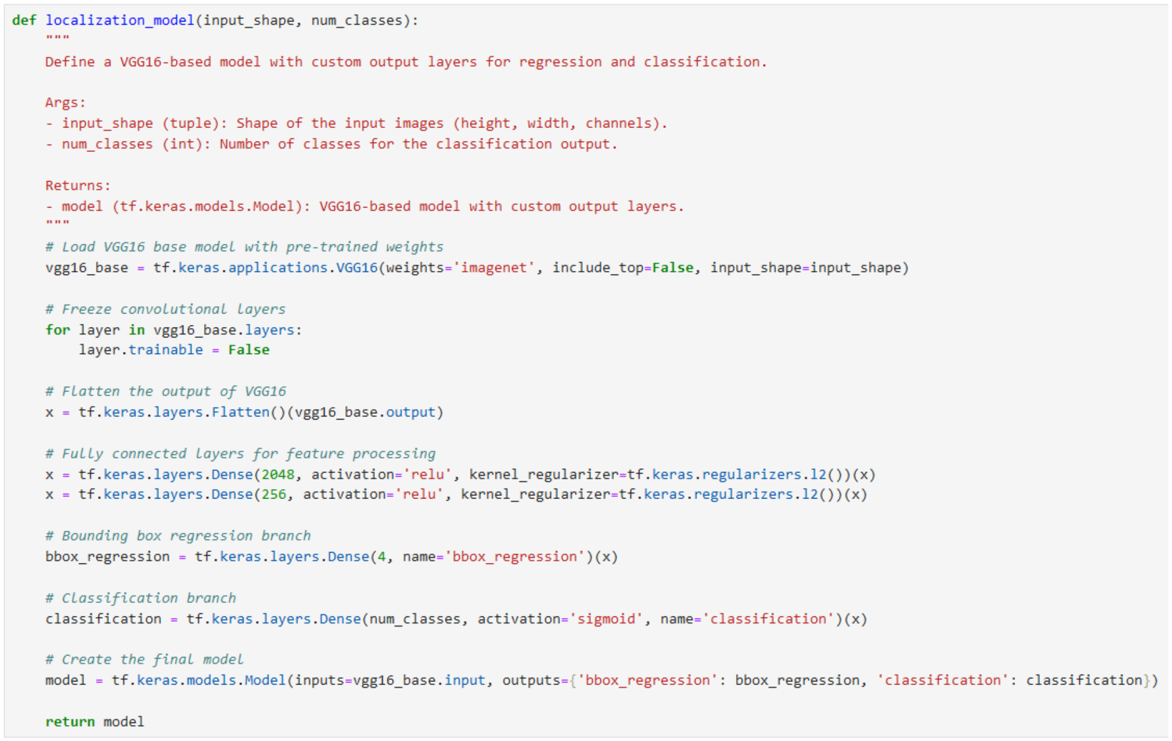
```
# Reads image paths and bounding boxes from the annotations file
train_image_paths, train_bounding_boxes = load_data(DATA_DIR, "train")

# Reads images from the given paths, resizes images, adjusts bounding boxes
X_train, train_bboxes, train_labels = load_and_preprocess_image(train_image_paths,
train_bounding_boxes)

# Converts this into a TensorFlow dataset
train_dataset, val_dataset, test_dataset = create_dataset(batch_size=bs)
```

5.2 Creating Model

In the standard workflow, the initial plan involved utilizing the ResNet-18 model for this task. However, due to its unavailability in TensorFlow and the prior use of VGG16 in Part 2 of the project, a more practical decision was made to employ VGG16. In implementing the object classification and localization system, a pretrained VGG16 model was utilized, building upon its successful application in the previous phase of the project.

The model architecture was extended by introducing a regression head after the final convolutional layer to predict the bounding box, while the classification head played a crucial role in object identification. Throughout the training process, a combined loss was applied. This encompassed binary cross-entropy loss for classification, considering the two classes: raccoon and no raccoon, and L2 loss for regression.

The code snippet for the model is presented below:



```python
def localization_model(input_shape, num_classes):
    """
    Define a VGG16-based model with custom output layers for regression and classification.

    Args:
    - input_shape (tuple): Shape of the input images (height, width, channels).
    - num_classes (int): Number of classes for the classification output.

    Returns:
    - model (tf.keras.models.Model): VGG16-based model with custom output layers.
    """
    # Load VGG16 base model with pre-trained weights
    vgg16_base = tf.keras.applications.VGG16(weights='imagenet', include_top=False, input_shape=input_shape)

    # Freeze convolutional layers
    for layer in vgg16_base.layers:
        layer.trainable = False

    # Flatten the output of VGG16
    x = tf.keras.layers.Flatten()(vgg16_base.output)

    # Fully connected layers for feature processing
    x = tf.keras.layers.Dense(2048, activation='relu', kernel_regularizer=tf.keras.regularizers.l2())(x)
    x = tf.keras.layers.Dense(256, activation='relu', kernel_regularizer=tf.keras.regularizers.l2())(x)

    # Bounding box regression branch
    bbox_regression = tf.keras.layers.Dense(4, name='bbox_regression')(x)

    # Classification branch
    classification = tf.keras.layers.Dense(num_classes, activation='sigmoid', name='classification')(x)

    # Create the final model
    model = tf.keras.models.Model(inputs=vgg16_base.input, outputs={'bbox_regression': bbox_regression, 'classification': classification})

    return model
```

Figure 16: Localization Model Code Snippet

The key components of the code snippet for the model:

- **Loading Pretrained Model:** The code incorporates the VGG16 base model with pre-trained weights from ImageNet by setting the `include_top` parameter to `False`. When `include_top` is set to `False`, the fully connected layers of the VGG16 model, which is responsible for classification, is excluded. This allows us to utilize the convolutional layers and the subsequent features of the pretrained model while excluding the original classification layer. By excluding the top layers, we can customize the output layers to suit the specific requirements of our object classification and localization task. This pretrained VGG16 base serves as the foundational component for our model, providing a robust feature extraction capability.

- **Freezing Convolutional Layers:** To preserve the pre-trained features, the code freezes the convolutional layers of the VGG16 base, preventing them from being updated during training.

- **Feature Processing Layers:** Following the VGG16 base, the code introduces two fully connected layers for feature processing. The first fully connected layer consists of 2048 units with a Rectified Linear Unit (ReLU) activation function. This layer serves to enhance the representation of extracted features from the VGG16 base. Subsequently, another fully connected layer with 256 units and a ReLU activation function further refines the feature representation. These additional dense layers contribute to capturing intricate patterns and higher-level features in the data. Also L2 kernel regularizer were used in these layers to prevent overfitting.

- **Bounding Box Regression Branch:** After the feature processing layers, a dedicated branch for bounding box regression is introduced. This branch includes a dense layer with 4 units, responsible for predicting the coordinates of the bounding box. The linear activation function is applied to this layer, enabling it to output continuous values for regression.

- **Classification Branch:** Parallel to the bounding box regression branch, the code introduces a classification branch. This branch incorporates a dense layer with the number of units equal to the specified `num_classes` parameter. In our case, considering the binary classification task (raccoon, no raccoon), a sigmoid activation function is employed to produce probabilities for each class.

- **Final Model:** The final model is constructed by combining these branches. The model takes the input images and produces two sets of outputs: one for bounding box regression (`bbox_regression`) and the other for classification (`classification`). This architecture is tailored to simultaneously address object localization and classification tasks.

This comprehensive model architecture aims to effectively combine features learned by the VGG16 base for both accurate bounding box regression and precise object classification.

5.3 Training Details

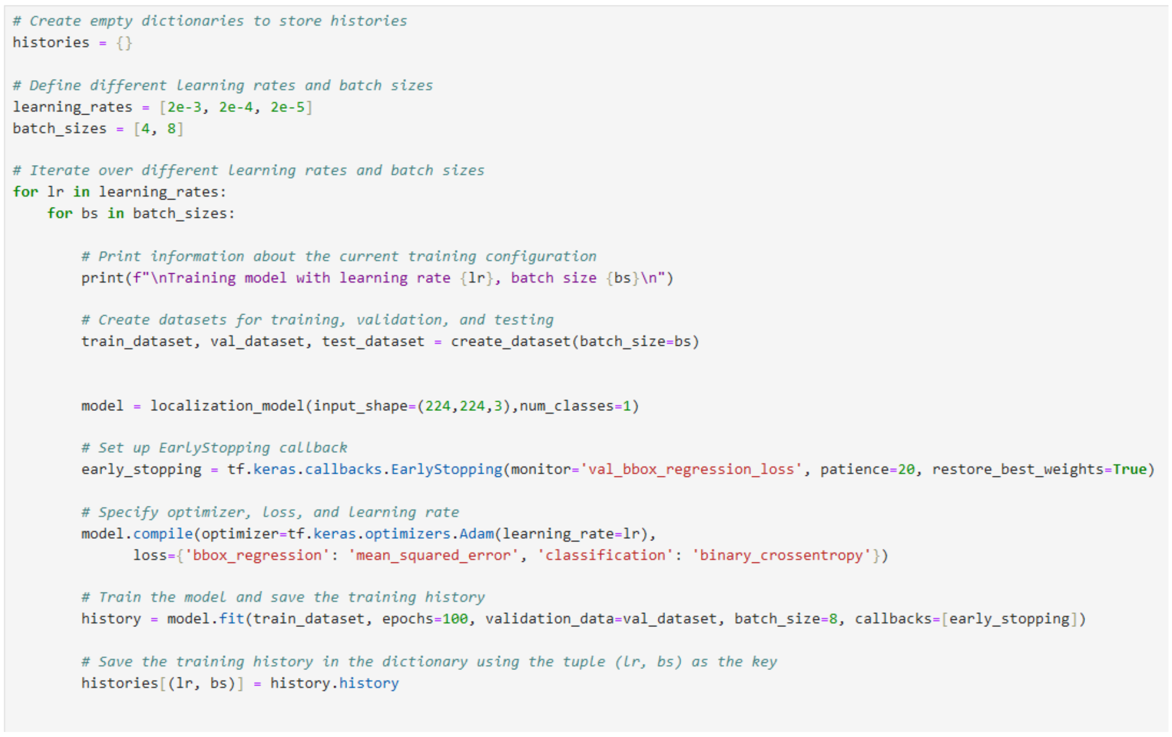
Similar to parts 1 and 2, the Adam optimizer was employed for training. The training process spanned 100 epochs, with an extended patience of 20 epochs, monitoring the validation bounding box loss. The decision to monitor validation bounding box loss is made considering that the classification loss stabilizes at almost zero after the first epoch, attributable to the dataset exclusively comprising raccoon images.

In the bounding box regression branch, the loss function applied was L2 loss, while the classification branch utilized binary cross-entropy loss.

5.4 Experiments on Learning Rates and Batch Sizes

Experiments are conducted to optimize the model's performance. The impact of different learning rates (2e-3, 2e-4, 2e-5) and batch sizes (4, 8) is explored. These experiments provide insights into how hyperparameter variations influence the model's accuracy and loss.

The code snippet for the experiment is shown below:



```python
# Create empty dictionaries to store histories
histories = {}

# Define different learning rates and batch sizes
learning_rates = [2e-3, 2e-4, 2e-5]
batch_sizes = [4, 8]

# Iterate over different learning rates and batch sizes
for lr in learning_rates:
    for bs in batch_sizes:

        # Print information about the current training configuration
        print(f"\nTraining model with learning rate {lr}, batch size {bs}\n")

        # Create datasets for training, validation, and testing
        train_dataset, val_dataset, test_dataset = create_dataset(batch_size=bs)


        model = localization_model(input_shape=(224,224,3),num_classes=1)

        # Set up EarlyStopping callback
        early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_bbox_regression_loss', patience=20, restore_best_weights=True)

        # Specify optimizer, loss, and learning rate
        model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=lr),
              loss={'bbox_regression': 'mean_squared_error', 'classification': 'binary_crossentropy'})

        # Train the model and save the training history
        history = model.fit(train_dataset, epochs=100, validation_data=val_dataset, batch_size=8, callbacks=[early_stopping])

        # Save the training history in the dictionary using the tuple (lr, bs) as the key
        histories[(lr, bs)] = history.history
```

Figure 17: Experiment Code Snippet (Learning Rates and Batch Sizes)

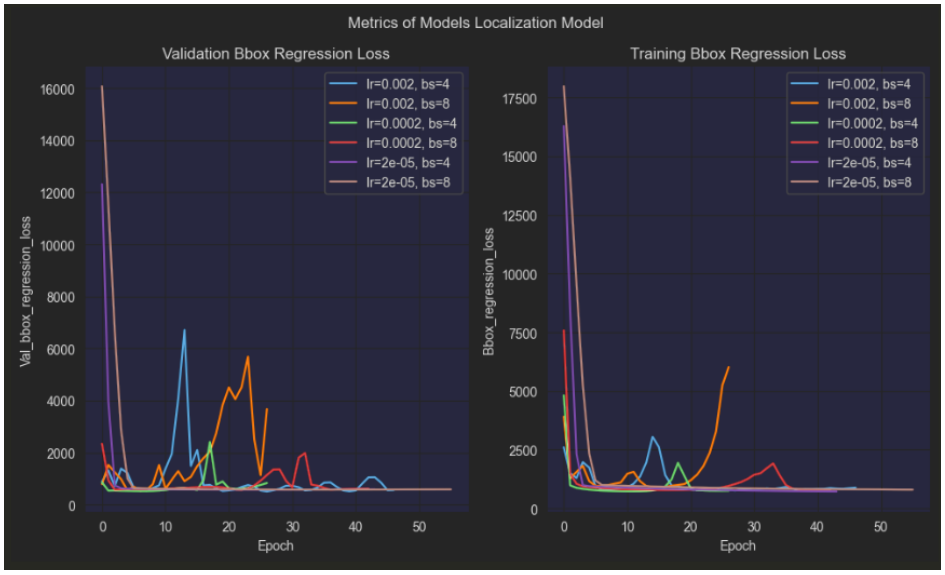


Figure 18: Experiment Metrics

The experiment involves training the model with different combinations of learning rates and batch sizes, enabling the identification of the optimal hyperparameter configuration.

Figure 18 illustrates the metrics obtained during the experiments. Notably, the learning rates of

2e-3 and 2e-4 exhibit oscillations in their performance after a certain number of epochs, indicating instability. In contrast, the learning rate of 2e-5 consistently demonstrates superior performance, achieving the lowest validation loss.

Furthermore, concerning batch sizes, the model trained with a batch size of 8 outperforms its counterpart with a batch size of 4 in terms of validation loss. This observation highlights the efficiency of utilizing a larger batch size, indicating improved convergence during training.

These findings emphasize the importance of carefully selecting hyperparameters, with a learning rate of 2e-5 and a batch size of 8 emerging as the optimal configuration for this particular task.

5.5 Selecting Best Model and Training

After conducting experiments, the best-performing model with batch size 8 and learning rate 2e-5 is selected based on validation performance. This model is further trained with specified training details.

5.6 Evaluating The Model

The performance of the object classification and localization system is rigorously evaluated using the mean intersection over union (mIoU) metric. This metric provides a quantitative measure of the model's accuracy in predicting bounding box locations compared to the ground truth.

The mIoU is computed over the test dataset, which comprises 17 images specifically reserved for evaluation. This dataset serves as an independent benchmark to assess the generalization capability of the trained model.

The calculation of mIoU involves determining the intersection over union for each predicted bounding box and its corresponding ground truth bounding box. The formula for IoU is given by:

$$IoU = \frac{Area\_of\_Overlap}{Area\_of\_Union}$$

The mean IoU is then calculated as the average IoU across all predictions in the test dataset, providing an overall assessment of the model's localization accuracy.

In our evaluation, the obtained mIoU was 0.6291, indicating a substantial overlap between predicted and ground truth bounding boxes.

Additionally, it's noteworthy to mention that the test loss, measured with L2 loss, was 591. This loss value suggests a relatively small pixel mismatch, around 24-25 pixels on average, between the predicted bounding boxes and the ground truth. While there may be slight variations, this outcome underscores the effectiveness of the model in localizing raccoons in test images.

5.7 Example Predictions

To demonstrate the practical application of the object classification and localization system, a set of example predictions is presented. These predictions showcase the accurate identification and localization of raccoons within given images by the model, taking into consideration the constraint of predicting a single bounding box. As observed from the predictions, the main issue lies in the incorrect sizing of the bounding box compared to the ground truth bounding box. However, it should be noted that this discrepancy can be attributed to the limited amount of training data, consisting of only 150 images, with validation comprising only 29 images. Despite the challenges posed by the relatively small dataset, these results are deemed to be reasonably decent.

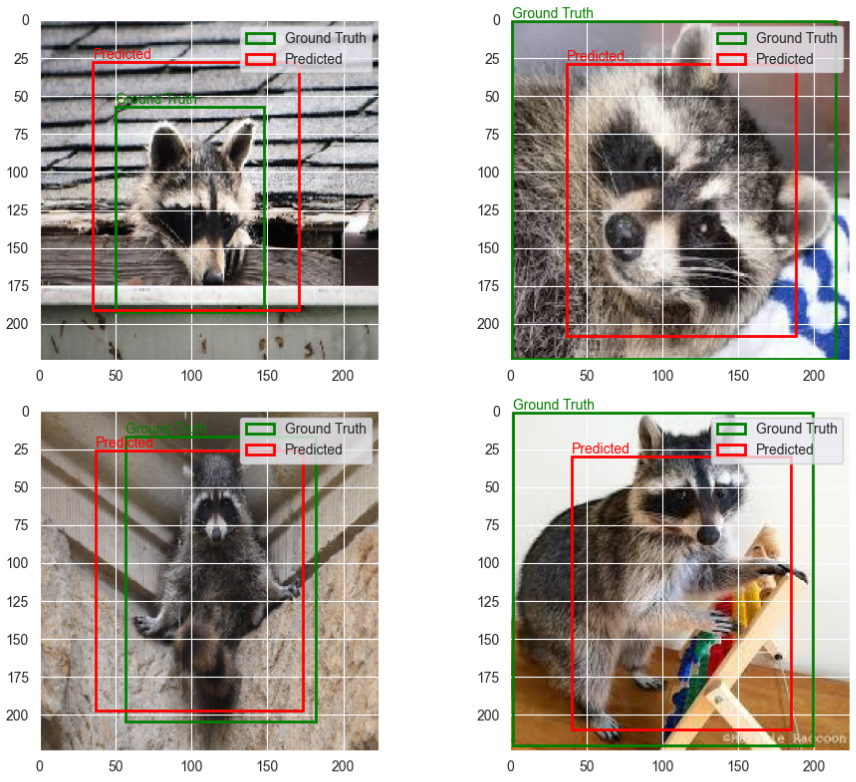


Figure 19: Example Predictions