Image Classification

&

Object Detection using Faster R-CNN & YOLOv5

42028 Deep Learning and Convolutional Neural Network

Assignment 2

Done by:
Justin Qi (14337911)

Contents

Introduction

In this report, we will be exploring image classification and object detection by deploying Convolutional Neural Networks (CNNs), tailoring these networks to enhance their performance for our tasks. The report offers a comprehensive summary of our approaches, findings, and subsequent analyses associated with image classification and object detection problems.

Central to our strategy are CNNs, a category of deep learning models that have exhibited exceptional prowess in tasks involving images. These models stand out due to their ability to automatically and adaptively learn spatial hierarchies of features, making them especially suitable for dealing with image data. In this study, we examine the AlexNet architecture as our baseline. AlexNet serves as the foundational block upon which we construct our custom CNN for image classification. Characterized by its depth and the use of rectified linear units (ReLU), AlexNet has been a significant contributor to the advancements in deep learning for image classification tasks.

Transitioning our focus to the task of object detection, we delve into the intricacies of two advanced models, namely Faster R-CNN and YOLOv5. Faster R-CNN is a model that merges the power of deep, fully convolutional networks with region proposals, establishing a seamless system for object detection. Its architecture is constructed upon a backbone of ResNet50, a CNN renowned for its residual learning framework that simplifies the training of deeper networks.

In contrast, YOLOv5, the latest version of the 'You Only Look Once' series, is marked by its end-to-end nature, processing images in a single pass and thereby providing faster inference while maintaining competitive results. Our YOLOv5 configuration focuses on adjusting its depth and width multiples, as well as custom anchor boxes, to better accommodate the specifics of our task.

Through the course of this report, I strive to provide insightful interpretations of our empirical findings, contrasting the performance of different methodologies, and suggesting potential paths for future exploration. It is our hope that our investigations contribute meaningfully to the continuous endeavors in refining and applying CNNs for image classification and object detection tasks.

Dataset description

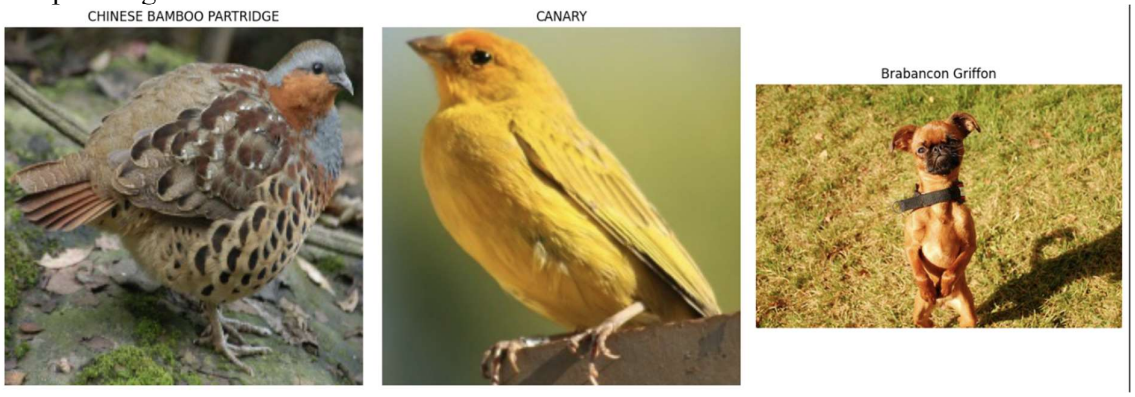
Image classification

The image classification dataset is composed of a diverse range of birds and dog breeds. This dataset is organized into 20 folders, each containing a certain number of images representing specific classes. The classes and their respective image counts are as follows:

	Bird breeds		**Dog breeds**	
#	**Name**	**Count**	**Name**	**Count**
1	Chinese Bamboo Partridge	159	Japanese Spaniel	185
2	Canary	156	Kerry Blue Terrier	179
3	Eared Pita	153	Irish Wolfhound	218
4	Red-Tailed Hawk	177	Papillon	196
5	Iwi	154	Lakeland Terrier	197
6	Green Winged Dove	153	Brabancon Griffon	153
7	Okinawa Rail	156	Shetland Sheepdog	157
8	Cinnamon Teal	158	English Foxhound	157
9	Rock Dove	137	Old English Sheepdog	169
10	Mallard Duck	140	German Shepherd	152

The total number of classes is 20, ten for each category, birds and dogs. The number of images per class varies, with the least represented class being the Rock Dove with 137 images and the most represented being the Irish Wolfhound with 218 images. This imbalance in data distribution may lead to biased results favoring classes with more examples.

Sample images:



Object detection

The object detection dataset is organized into 3 folders, Coco, Pascal, and Yolo. Which are to be used for separately for the required implementation methods. In this report, Pascal was used for Faster R-CNN and Yolo was used for YOLOv5.

Both files contain the same number of images in training, validation, and test directories: 1435, 307, 307. Alongside their annotations, which are .xml files for Pascal and .txt files for Yolo.

```
Number of .jpg files in training directory: 1435
Number of .jpg files in validation directory: 307
Number of .jpg files in test directory: 307
```

The classes for object detection are as such:
[dock, boat, car, jetski, lift]

Dataset split

The image classification dataset was being split into train, test, and validation set in the proportion of 60%, 20%, 20%. The split was done on each of the classes to ensure all classes have similar weightage in each set.

Train: 1975 images
Test: 652 images
Valid: 679 images

```
Number of files in train directory: 1975
Number of files in validation directory: 652
Number of files in test directory: 679
```

The object detection dataset has already been pre-split. Hence, no further alteration was made from my end.

Image classification

Baseline architecture

The baseline model I have chosen is the AlexNet model. The model consists of several convolutional layers, interspersed with max pooling layers and batch normalization layers, followed by a series of fully connected layers. Here's a summary of the model:

- The first layer is a convolutional layer (Conv_1) with 96 filters, each of size 11x11, with a stride of 4. This layer uses the 'relu' activation function.
- This is followed by a max pooling layer (Pooling_1) with a 2x2 pool size and stride of 2.
- A batch normalization layer (Batch Normalisation_1) is then applied.
- The second convolutional layer (Conv_2) has 256 filters, each of size 5x5, with a stride of 1. This layer also uses the 'relu' activation function.
- Another max pooling layer (Pooling_2) with the same parameters as before is then used.
- A second batch normalization layer (Batch Normalisation_2) is applied.
- The model then has three consecutive convolutional layers (Conv_3, Conv_4, Conv_5) with 384, 384, and 256 filters respectively, each of size 3x3 with a stride of 1. All these layers use the 'relu' activation function.
- After each of these convolutional layers, a batch normalization layer is applied.
- Another max pooling layer (Pooling_3) and a fourth batch normalization layer (Batch Normalisation_4) are applied after Conv_5.
- The model is then flattened to prepare the data for the fully connected layers.
- Three fully connected layers (Dense layer_1, Dense layer_2, Dense layer_3) with 4096, 4096, and 1000 nodes respectively are applied. Each of these layers uses the 'relu' activation function, and each is followed by a dropout layer with a dropout rate of 0.5, as well as a batch normalization layer.
- The final layer is a fully connected layer with 20 nodes, one for each class. This layer uses the 'softmax' activation function to provide a distribution of probabilities across the classes.

The model is trained using a categorical cross-entropy loss function and an RMSprop optimizer with a learning rate of 1e-4. I also applied image augmentation to artificially expand the size of the training set and reduce overfitting. This is done by applying a series of random transformations (like rotation, width shift, height shift, shear transformation, zoom, and horizontal flip) to the training images.

The model is trained for 100 epochs, using batches of 20 images. The model's accuracy on the validation set being used to monitor its performance and avoid overfitting.

```
model = tf.keras.models.Sequential([
    #Conv_1
    tf.keras.layers.Conv2D(96, (11,11),strides=4, padding='valid', activation='relu', input_shape=(224, 224, 3)),
    # Pooling_1
    tf.keras.layers.MaxPooling2D((2, 2), strides=(2,2),padding='valid'),
    # Batch Normalisation_1
    tf.keras.layers.BatchNormalization(),
    # Conv_2
    tf.keras.layers.Conv2D(256, (5,5),strides=1, padding='valid', activation='relu'),
    # Pooling_2
    tf.keras.layers.MaxPooling2D((2, 2), strides=(2,2),padding='valid'),
    #Batch Normalisation_2
    tf.keras.layers.BatchNormalization(),
    # Conv_3
    tf.keras.layers.Conv2D(384, (3,3),strides=1, padding='valid', activation='relu'),
    # Batch Normalisation_3
    tf.keras.layers.BatchNormalization(),
    # Conv_4
    tf.keras.layers.Conv2D(384, (3,3),strides=1, padding='valid', activation='relu'),
    # Batch Normalisation_3
    tf.keras.layers.BatchNormalization(),
    #conv_5
    tf.keras.layers.Conv2D(256, (3,3),strides=1, padding='valid', activation='relu'),
    #pooling_3
    tf.keras.layers.MaxPooling2D((2, 2), strides=(2,2),padding='valid'),
    #Batch Normalization_4
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Flatten(),
    #Dense layer_1
    tf.keras.layers.Dense(4096, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.BatchNormalization(),
    #Dense layer_2
    tf.keras.layers.Dense(4096, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.BatchNormalization(),
    #Dense layer_3
    tf.keras.layers.Dense(1000, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(20, activation='softmax')  # 20 classes
    ])
```

```
# Image augmentation
train_datagen = ImageDataGenerator(
      rescale=1./255,
      rotation_range=40,
      width_shift_range=0.2,
      height_shift_range=0.2,
      shear_range=0.2,
      zoom_range=0.2,
      horizontal_flip=True,
      fill_mode='nearest')

test_datagen = ImageDataGenerator(rescale=1./255)

# Flow training images in batches of 20 using train_datagen generator
train_generator = train_datagen.flow_from_directory(
        train_dir,  # This is the source directory for training images
        target_size=(224, 224),  # All images will be resized to 150x150
        batch_size=20,
        class_mode='categorical')

# Flow validation images in batches of 20 using test_datagen generator
validation_generator = test_datagen.flow_from_directory(
        valid_dir,
        target_size=(224, 224),
        batch_size=20,
        class_mode='categorical')
```

```
Found 1975 images belonging to 20 classes.
Found 652 images belonging to 20 classes.
```

```
model.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(learning_rate=1e-4),
              metrics=['acc'])
```

```
#checkpoint = ModelCheckpoint('weights.{epoch:02d}-{val_loss:.2f}.hdf5', monitor='val_loss', save_best_only=True, verbose=1, save_freq='epoch')
filepath='/tmp/weights.{epoch:02d}-{val_loss:.2f}.hdf5'
checkpoint=tf.keras.callbacks.ModelCheckpoint(filepath, monitor='val_loss', verbose=0, save_best_only=False, save_weights_only=False, mode='auto', save_freq='epoch')
```

```
# Train the model
history = model.fit(
      train_generator,
      steps_per_epoch=1975//20,
      epochs=100,
      validation_data=validation_generator,
      validation_steps=652//20,
      callbacks = [checkpoint],
      verbose=2)
```

Customized architecture

In the customized AlexNet model, I've made two main modifications to the baseline model:

- **Addition of a Convolutional Layer:** After the last convolutional layer of the baseline model, I added an additional convolutional layer (Conv2D) with 256 filters of size 2x2 and stride 1, using a 'relu' activation function. The intuition behind adding an extra convolutional layer can be to extract more complex and abstract features from the image, which could potentially improve the performance of the model.

- **Removal of a Dense Layer:** I removed the second last dense layer (with 4096 units) from the baseline model. This dense layer was one of the three fully connected layers in the original architecture. Removing this layer reduces the complexity of the model and can help in avoiding overfitting. This could be particularly beneficial for smaller dataset or if the baseline model was observed to overfit the data.

The weights of the first 13 layers from the baseline model, which remain the same in the new model, have been transferred to the new model. This is an usage of transfer learning, which can speed up the training process and improve performance.

The model is compiled and trained in the same way as the baseline model, using categorical cross-entropy as the loss function and RMSprop with a learning rate of 1e-4 as the optimizer. The same image augmentation techniques, batch size, and number of epochs are used as well.

```
# Remove the second last dense layer (4096 units)
# Add a convolutional layer after the last convolutional layer

new_model = tf.keras.models.Sequential([
    # The first 13 layers are the same as the baseline model
    tf.keras.layers.Conv2D(96, (11,11),strides=4, padding='valid', activation='relu', input_shape=(224, 224, 3)),
    tf.keras.layers.MaxPooling2D((2, 2), strides=(2,2),padding='valid'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(256, (5,5),strides=1, padding='valid', activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2), strides=(2,2),padding='valid'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(384, (3,3),strides=1, padding='valid', activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(384, (3,3),strides=1, padding='valid', activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(256, (3,3),strides=1, padding='valid', activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2), strides=(2,2),padding='valid'),
    tf.keras.layers.BatchNormalization(),

    # Added an extra convolutional layer
    tf.keras.layers.Conv2D(256, (2,2),strides=1, padding='valid', activation='relu'),

    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(4096, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.BatchNormalization(),

    # Removed second last layer
    tf.keras.layers.Dense(1000, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(20, activation='softmax')  # 20 classes
])
```

```
# transferring weights for the first 13 layers which are the same in both models
for new_layer, layer in zip(new_model.layers[:13], baseline_model.layers[:13]):
    new_layer.set_weights(layer.get_weights())
```

Assumptions/Intuitions

Convolutional Layers: The models employ multiple convolutional layers, which are designed to automatically and adaptively learn spatial hierarchies of features from the input images. The intuition is that lower layers learn simple and generic features like edges and colors, while deeper layers learn more complex and abstract features.

Pooling Layers: The models use pooling layers following some of the convolutional layers to reduce the spatial size of the representation, to reduce the amount of parameters and computation in the network, and also to control overfitting. The assumption is that some spatial information can be sacrificed for a more compact and generalized representation of the image content.

Dropout Layers: Dropout layers are added with the assumption that they help prevent overfitting during training. By randomly dropping out (i.e., setting to zero) a number of output features of the layer during training, the model is forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons.

Batch Normalization Layers: Batch normalization is used to normalize the activations of the neurons in the network to speed up learning and reduce the sensitivity to network initialization. The underlying assumption is that this can stabilize the learning process and dramatically reduce the number of training steps required to reach high accuracy.

Transfer Learning: In the customized model, the first 13 layers' weights are transferred from the baseline model. The intuition behind this is that the initial layers have already learned useful representations for image features, which can be utilized as a starting point for the new model, reducing training time and potentially improving performance.

Removing Dense Layers: The second last dense layer is removed in the customized model with the assumption that this will reduce overfitting. Dense layers have a lot of parameters, making them prone to overfitting, especially with smaller datasets.

Adding Convolutional Layers: An extra convolutional layer is added in the customized model under the assumption that this will enable the model to learn more complex features, potentially improving the model's performance.

Model summary

The baseline model is a standard implementation of the AlexNet architecture. It contains eight layers (five convolutional and three fully connected layers), followed by a final output layer. The model uses ReLU activation functions, MaxPooling after some of the convolutional layers, and a Softmax output layer for multiclass classification. Each of the convolutional and fully connected layers is followed by a batch normalization layer and the fully connected layers also include a Dropout layer with a rate of 0.5 to prevent overfitting.

The customized model is a variant of the AlexNet architecture. It includes an additional convolutional layer and removes one of the fully connected layers from the baseline model.

Experimental setting

The model (baseline and customized) was trained using the AlexNet architecture for a total of 100 epochs. The training data was divided into 98 batches per epoch. A validation set was also used after each epoch to gauge the model's performance on unseen data.

The model used a categorical cross-entropy loss function, which is commonly used for multi-class classification problems. The loss value reported is the value of this function, which the model attempts to minimize during training. The model's accuracy, both on the training data and the validation data, is also reported after each epoch. The accuracy is the proportion of correct predictions made by the model.

The model's learning rate, optimizer, and other hyperparameters were specified earlier. The learning rate was set to a value that balances the speed of learning with the risk of overshooting the minimum. Standard practice like dropout and batch normalization was used to prevent overfitting.

Experimental results

Baseline model

The performance of the baseline model over 100 epochs showed significant improvement over time. This can be seen in the decrease in the model's loss and the steady increase in accuracy.

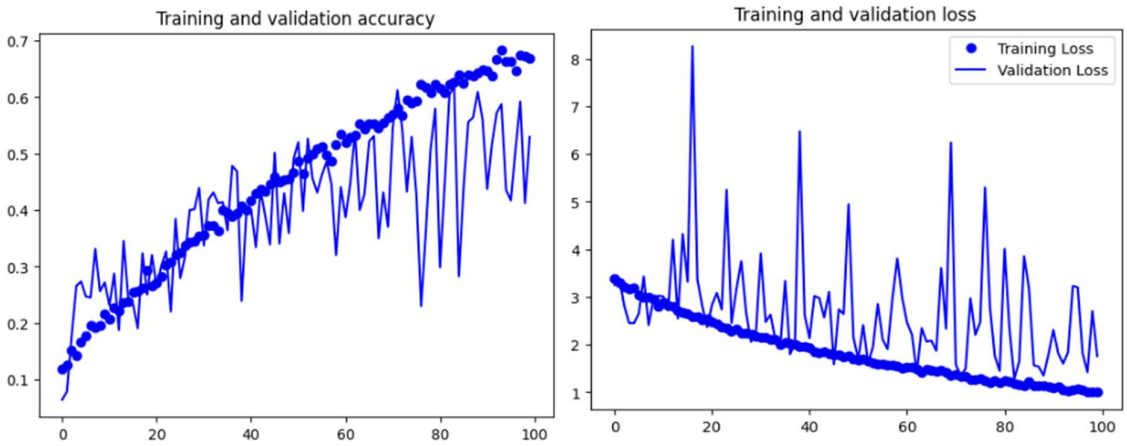
Initially, at the first epoch, the model had a training accuracy of 0.1187 and a validation accuracy of 0.0641, with a loss of 3.3847 on the training data and 3.2543 on the validation data. As the model continued training, the accuracy steadily increased while the loss decreased.

By the 50th epoch, the model achieved a training accuracy of 0.4670 and a validation accuracy of 0.4906, with a loss of 1.6988 on the training data and 2.1527 on the validation data.

The best validation accuracy peaked at around 0.6.

Though the model showed some fluctuation in performance, possibly due to overfitting, the overall trend was positive. It may be due to that the model's performance varied greatly depending on the complexity of the data at each epoch.

Based on these results, the baseline model has shown significant potential for improvement. With further tuning, the model might benefit from further improvements in its architecture, such as more sophisticated layers, optimization strategies, or improved handling of the input data.



The model was then tested on the test set, which yielded an accuracy result of 0.5439

```
# Create a test data generator
test_datagen = ImageDataGenerator(rescale=1./255)

# Use the generator to load the test data
test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(224, 224),
    batch_size=20,
    class_mode='categorical',  # Use 'categorical' if your model uses categorical_crossentropy loss
    shuffle=False)  # Important: do not shuffle test data

# Evaluate the model on the test data
loss, accuracy = model.evaluate(test_generator, steps=test_generator.samples // test_generator.batch_size)
print("Test accuracy:", accuracy)
```

```
Found 679 images belonging to 20 classes.
33/33 [==============================] - 3s 85ms/step - loss: 1.6844 - acc: 0.5439
Test accuracy: 0.5439394116401672
```

Finally, I randomly sampled 5 images from the test set and showed the actual vs predicted class.

Customized model

With the exact same experimental setup as the baseline model, the customized model was trained over 100 epochs, and the experimental results for each epoch were recorded. The initial epoch started with a training loss of 2.8047 and an accuracy of 0.2184. This quickly improved by the second epoch to a loss of 1.9688 and accuracy of 0.4286.

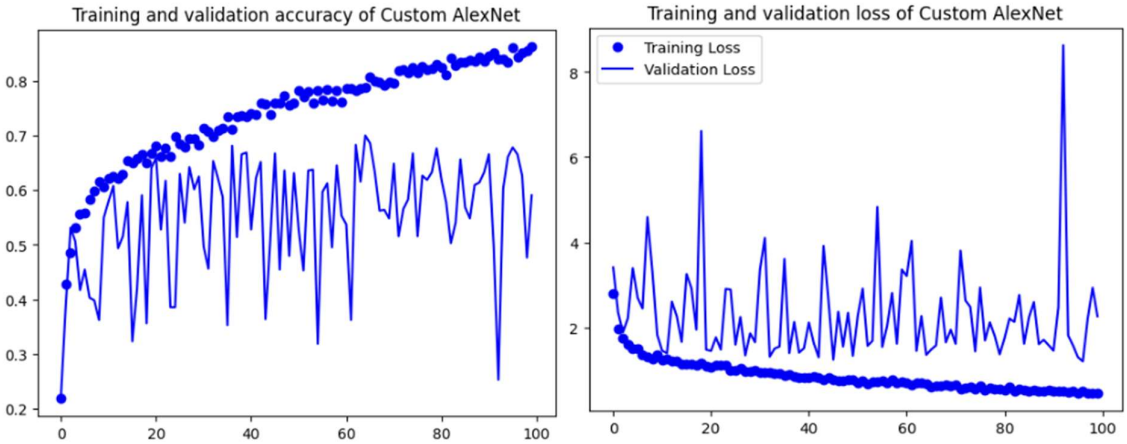
The training accuracy continued to increase with each epoch, reaching over 0.5 by the third epoch, and then hovering around 0.55 by the fifth epoch. Meanwhile, the validation accuracy also increased, albeit at a slower rate, reaching 0.5312 by the third epoch, but falling slightly by the fifth epoch to 0.5063 due to overfitting. However, by the tenth epoch, both training

and validation accuracies significantly improved, with values of 0.6066 and 0.55, respectively.

The model's training and validation losses showed a similar pattern of improvement, falling from 2.8047 to 1.3500 for training loss, and from 3.4100 to 1.8264 for validation loss over the first ten epochs. However, it's worth noting that the validation loss started to increase again after the fifth epoch, indicating possible overfitting.

Despite the steady improvement in training accuracy, reaching 0.7816 by the 51st epoch, the validation accuracy remained relatively constant, indicating that the model might have reached its limit in learning from the data. Also, the validation loss showed a similar pattern to the validation accuracy, declining at first but then remaining relatively stable.

In conclusion, the customized model showed steady improvement in training accuracy and loss over the 100 epochs, although validation accuracy and loss plateaued after approximately 50 epochs. This suggests that while the model was able to learn effectively from the training data, it may have struggled to generalize its learning to new data.

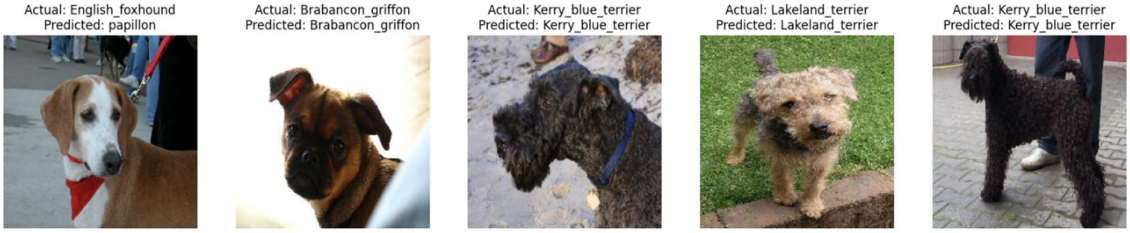


The test set accuracy achieved on the customised model was 0.6333, which saw an improvement of approximately 10% from the baseline model.

```
# Evaluate the model on the test data
loss, accuracy = new_model.evaluate(test_generator, steps=test_generator.samples // test_generator.batch_size)
print("Test accuracy:", accuracy)
```

```
33/33 [==============================] - 3s 84ms/step - loss: 1.9383 - acc: 0.6333
Test accuracy: 0.6333333253860474
```

Similarly, I randomly sampled 5 images from the test set to show its actual vs predicted labels.



Object Detection

Faster R-CNN

The Faster R-CNN is a popular model for object detection tasks. It improves upon previous versions (R-CNN and Fast R-CNN) by integrating the region proposal network (RPN) within the architecture, hence making the model faster and more efficient.

Faster R-CNN architecture consists of two modules. The first module is a deep fully convolutional neural network which proposes regions, and the second module is the Fast R-CNN detector that uses the proposed regions.

- **Image Input and Feature Extraction**: An image is input into the system, which is processed by a series of convolutional and pooling layers. These early layers perform feature extraction. In this case, we're using ResNet50 as our backbone model. ResNet50 is a variant of ResNet model which has 50 layers deep. It can identify the important features in the input image.
- **Region Proposal Network (RPN)**: The feature map produced by the ResNet50 is passed on to the Region Proposal Network. This network proposes candidate object bounding boxes and gives an objectness score for each proposal. The RPN does this by sliding a small network over the convolutional feature map output by the previous layer.
- **ROI Pooling**: Once the proposed regions are generated, they're reshaped into a fixed size, so that they can be fed into a fully connected network. This is done through a process known as Region of Interest (RoI) Pooling.
- **Bounding Box Regression and Classification**: The RoI pooled features are then fed into fully connected layers which output a discrete probability distribution (over the object classes plus a 'background' class) and four real-valued numbers for each RoI. These four numbers are the coordinates of a bounding box.

The class with the maximum probability is considered the prediction of the network, and the corresponding bounding box is the location of the object within the image.

This process allows Faster R-CNN to process images and accurately identify and locate objects within the image.

YOLOv5

YOLOv5 (You Only Look Once, version 5) is a state-of-the-art object detection model. Like Faster R-CNN, it is designed to identify and classify objects within images and provide bounding boxes for these objects. However, the two models follow different strategies and have different strengths.

YOLO models are built on the fundamental concept of viewing object detection as a single regression problem, directly predicting class probabilities and bounding box coordinates from a single scan of the input image. This makes YOLO models, including YOLOv5, generally faster and more efficient than two-stage detectors like Faster R-CNN.

YOLOv5 introduces several architectural improvements on its predecessors, including the use of a feature pyramid for detecting objects at different scales and aspect ratios, and a more efficient backbone network for feature extraction.

Configuration Parameters and Anchors: The YAML file starts by setting the number of classes (nc), and the depth and width multiples, which influence the depth and width of the model. It also specifies the size of the anchors that will be used at different stages of the model. These anchors represent different scales and aspect ratios that the model will consider when predicting bounding boxes.

Backbone: This is the first part of the model, where the input image is processed through a series of layers to extract feature maps. This starts with the Focus layer, which concatenates adjacent pixels in the input image to create a richer representation. This is followed by a series of Conv and BottleneckCSP layers, which perform convolutions and create skip connections within the model. The SPP layer near the end of the backbone introduces multi-scale features by applying max pooling with different kernel sizes.

Head: This is the second part of the model, which uses the feature maps produced by the backbone to predict the class and bounding box for objects in the image. This part of the model also includes Conv, Upsample, and BottleneckCSP layers, similar to the backbone. The Concat layers combine features from different stages of the model, providing a form of skip connection between different levels of the feature pyramid. These concatenated features are processed through additional layers until they reach the Detect layer, which uses the anchor boxes defined earlier to predict the class and bounding box for objects at different scales and aspect ratios. Each Detect layer is associated with a different level of the feature pyramid, allowing the model to detect objects at different scales.

Anchors: Anchors are pre-defined bounding boxes that are used as reference points for predicting the bounding boxes of detected objects. They are defined at different scales corresponding to the three different levels of the feature pyramid (P3/8, P4/16, P5/32), allowing the model to detect objects of different sizes.

Experimental results

Faster R-CNN

The Faster R-CNN model was trained and evaluated over various metrics. The evaluation metrics include mean average precision (mAP) and mean average recall (MAR) at different intersection over union (IoU) thresholds and object sizes.

```
{'map': tensor(0.6469),
 'map_50': tensor(0.9220),
 'map_75': tensor(0.6912),
 'map_large': tensor(0.7563),
 'map_medium': tensor(0.5563),
 'map_per_class': tensor([0.6343, 0.7006, 0.5874, 0.6987, 0.6136]),
 'map_small': tensor(0.3280),
 'mar_1': tensor(0.4957),
 'mar_10': tensor(0.7097),
 'mar_100': tensor(0.7097),
 'mar_100_per_class': tensor([0.6838, 0.7355, 0.6869, 0.7656, 0.6768]),
 'mar_large': tensor(0.7998),
 'mar_medium': tensor(0.6591),
 'mar_small': tensor(0.3569)}


"Classes: ['__background__', 'boat', 'car', 'jetski', 'lift', 'dock']"


AP / AR per class
-------------------------------------------------------------------------
|    | Class                | AP                | AR                |
-------------------------------------------------------------------------
|1   | boat                 | 0.634             | 0.684             |
|2   | car                  | 0.701             | 0.735             |
|3   | jetski               | 0.587             | 0.687             |
|4   | lift                 | 0.699             | 0.766             |
|5   | dock                 | 0.614             | 0.677             |
-------------------------------------------------------------------------
|Avg                        | 0.647             | 0.710             |
```

The mAP for this model was 0.647, and the mAP at IoU thresholds of 50% and 75% were 0.922 and 0.691, respectively. This shows that the model maintains a high average precision at standard IoU thresholds. When we break this down by object sizes, the mAP was highest for large objects (0.756), followed by medium objects (0.556), and was lowest for small objects (0.328). This suggests that the model performs best when detecting larger objects.

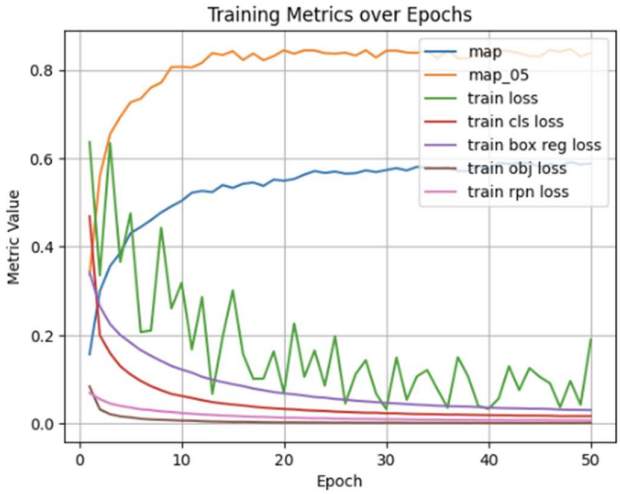
The MAR, on the other hand, was evaluated at 1, 10, and 100 detections per image. The MAR at these levels were 0.496, 0.710, and 0.710, respectively, with no difference between MAR10 and MAR100. Again, looking at the object sizes, the MAR was highest for large objects (0.800), followed by medium objects (0.659), and was lowest for small objects (0.357).

The performance of the model for each object class was as follows:

Boat: The model demonstrated an average precision (AP) of 0.634 and average recall (AR) of 0.684.
Car: The model had an AP of 0.701 and an AR of 0.735, indicating its strong performance in detecting cars.
Jetski: The model's performance on jetskis was slightly weaker, with an AP of 0.587 and an

AR of 0.687.
Lift: The model showed a solid performance on lifts, with an AP of 0.699 and an AR of 0.766.
Dock: The model's performance on docks was comparable to boats and lifts, with an AP of 0.614 and an AR of 0.677.

In summary, the Faster R-CNN model demonstrated a good performance across all object classes, with the strongest performance in detecting cars and lifts. However, the model appears to struggle somewhat with detecting smaller objects as well as objects of the similar color, which could be an area of focus for further improvement.



4 images were sampled to illustrate how the object detection model applied bounding boxes on the objects shown in the images.

YOLOv5

The YOLOv5 model was trained using a custom YAML configuration, new_train_yaml, with a network of 182 layers and approximately 7.26 million parameters. The model was trained for 200 epochs.

The model performance was measured in terms of precision (P), recall (R), mean average precision at an intersection over union (IoU) of 50% (mAP50), and mean average precision between an IoU of 50% and 95% (mAP50-95).

```
200 epochs completed in 0.667 hours.
Optimizer stripped from runs/train/exp/weights/last.pt, 14.8MB
Optimizer stripped from runs/train/exp/weights/best.pt, 14.8MB

Validating runs/train/exp/weights/best.pt...
Fusing layers...
new_train_yaml summary: 182 layers, 7257306 parameters, 0 gradients
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 10/10 [00:04<00:00,  2.04it/s]
                   all        307        986      0.948      0.869      0.903      0.668
                  boat        307        421      0.952      0.846      0.901      0.646
                   car        307         26      0.938      0.923      0.911      0.653
                  dock        307         36      0.969      0.882      0.944      0.733
                jetski        307         89       0.91      0.796      0.825      0.579
                  lift        307        414      0.969      0.899      0.931      0.726
```

The overall results show that the model achieved a precision of 0.948 and a recall of 0.869, indicating a strong ability to accurately detect objects while also maintaining a good balance by correctly identifying a significant proportion of actual instances. The mAP50 value of 0.903 and mAP50-95 value of 0.668 further underscore the model's robust performance, showing that it maintains high average precision even when the IoU threshold is varied.

Results per object class are as follows:

Boat: The model identified boats with a precision of 0.952 and a recall of 0.846. The mAP50 and mAP50-95 values were 0.901 and 0.646, respectively.

Car: The model performed slightly less well on cars, with precision of 0.938 and recall of 0.923. However, it maintained good mAP50 and mAP50-95 values of 0.911 and 0.653, respectively.

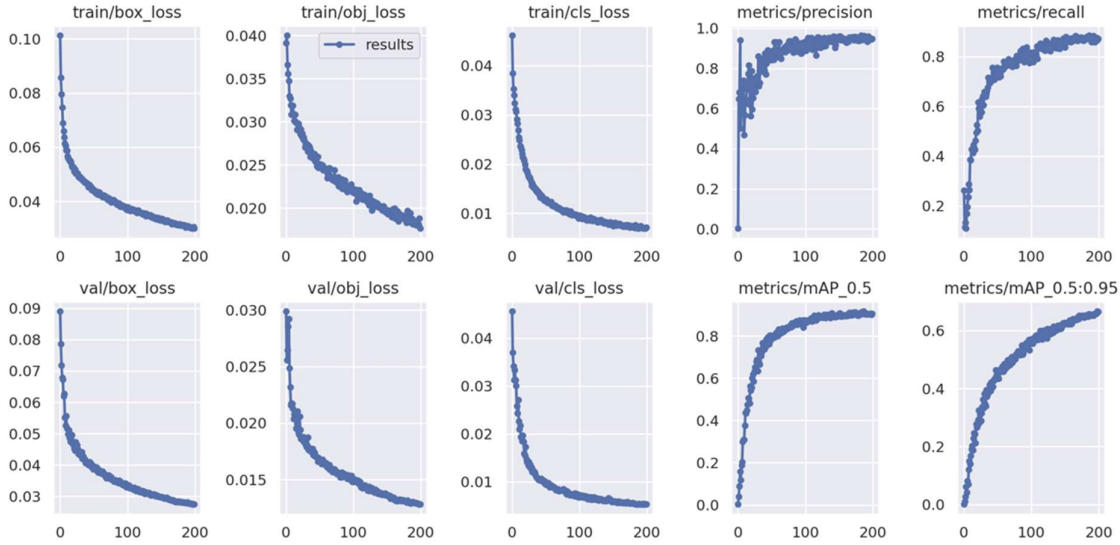
Dock: The model's performance on docks was excellent, achieving a precision of 0.969 and recall of 0.882. The mAP50 was very high at 0.944, as was the mAP50-95 at 0.733.

Jetski: The model's performance on jetskis was slightly weaker, with precision of 0.91 and recall of 0.796. The mAP50 and mAP50-95 values were 0.825 and 0.579, respectively, indicating a lower average precision compared to other classes.

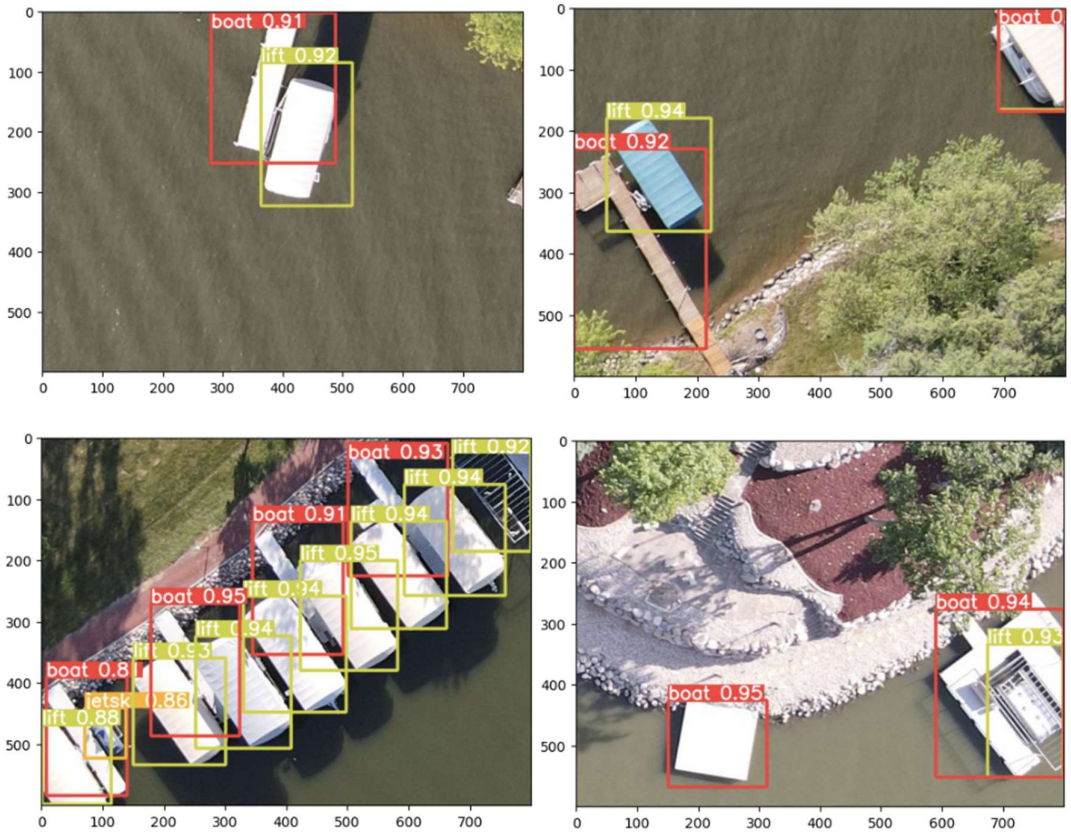
Lift: The model identified lifts with high precision (0.969) and recall (0.899), and the mAP50 and mAP50-95 values were strong at 0.931 and 0.726, respectively.

Overall, the YOLOv5 model demonstrated strong performance across multiple object classes, with the highest performance seen in the dock and lift classes. These results suggest that the model is suitable for detecting a range of different object types in images, even in the absence

of gradient information during training. The graph below displays the model performance throughout the span of 200 epochs.



I've randomly sampled 5 images and ran the YOLOv5 object detection model on them.



Conclusion

In this report, we explored experimentations with image classification and object detection, harnessing the power of Convolutional Neural Networks (CNNs) to tackle these challenging tasks. We witnessed how CNNs, particularly AlexNet for image classification and Faster R-CNN and YOLOv5 for object detection, could be skilfully adapted to enhance their performance according to our specific needs.

The study's findings underscored the effectiveness of customizing our CNN architecture based on AlexNet for image classification, with modifications showing a marked improvement in performance. The exploratory nature of this exercise has demonstrated the potential for further tuning, emphasizing the dynamic nature of deep learning models and their adaptability to various tasks.

In our object detection pursuits, we found the Faster R-CNN and YOLOv5 models to be both potent and versatile, capable of discerning multiple object classes within images with notable precision. Comparing the two, Faster R-CNN delivered superior results on medium and large objects, leveraging its region proposal mechanism to handle complex scenes. However, YOLOv5 demonstrated its strengths by providing faster inference times and competitive performance, particularly on smaller objects. These results hint at the importance of considering the specific needs of a given task when selecting an appropriate model - trade-offs between speed and accuracy, along with the size of the objects of interest, can significantly sway the optimal choice.

While we have made considerable progress in image classification and object detection tasks, the experimental results were not all fully accurate, there were several images/objects that were wrongly classified (shown in earlier sections). Hence, there are still improvements to be made to create more effective and efficient architectures.

Overall, our study contributes to the ongoing conversation about the application of CNNs to image analysis tasks. The promising results gleaned from our custom models reiterate the power and potential of deep learning in computer vision and pave the way for future exploration in this exciting realm.