INF 265 - Project 2 Report
Sander Marx, Simen Sørensen

The code discussed in this report can be found in the delivered jupyter notebook **INF265_Project2_Group_12.ipynb.** This technical report will discuss questions from section 4.

Division of labour:
Simen worked on Object detection and a little bit on reporting
Sander worked on Object localization, object detection data preparation and reporting

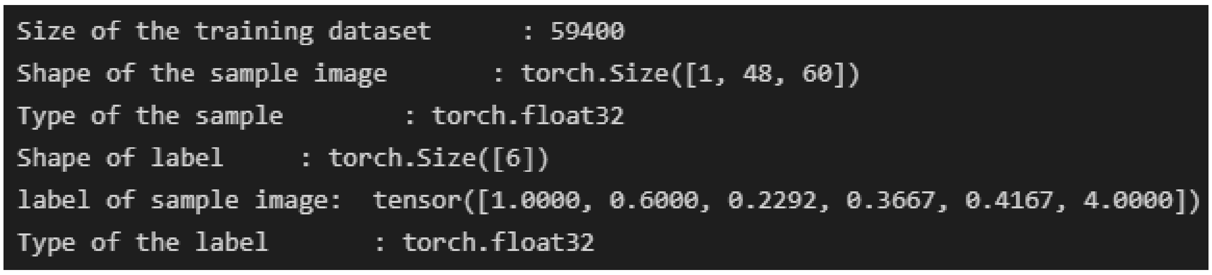
Data analysis and preprocessing

Section 2

For section 2, image localization we did some data analysis by looking at the shape of the data, the class distribution and then plotted some images and the bounding box of each class. We preprocessed the data by normalizing it.

Data analysis

Shape and size
We first started looking at the shape and size of the data to get some idea of what we are working with.



```
Size of the training dataset      : 59400
Shape of the sample image       : torch.Size([1, 48, 60])
Type of the sample        : torch.float32
Shape of label     : torch.Size([6])
label of sample image:  tensor([1.0000, 0.6000, 0.2292, 0.3667, 0.4167, 4.0000])
Type of the label       : torch.float32
```

Figure 1: Shape and sizes of localization data

Plotting og images

We then plotted 4 images of each class that contained an object, figure 2 shows this for class 0 and class 1. This is done to get a bit more insight into our data and find possible anomalies.

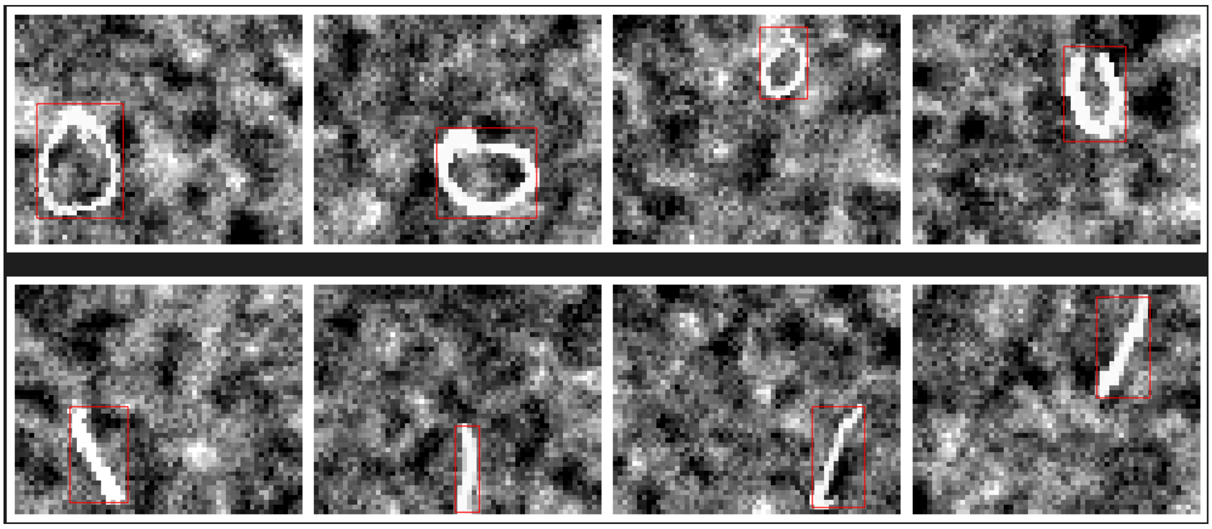


Figure 2: class 0 and 1 from training set plotted with true bounding boxes

Class distribution

And then we looked at the class distribution. This was done to ensure that we have a balanced class distribution. We do this to check if we need to up- or downsample our classes. It is also a good idea to look at the class distribution when choosing a performance measure as a performance measure like Accuracy does not do as well as something like F1 score when we are working with unbalanced class distributions. As we can see in figure 3, the classes are fairly balanced with the class with most occurrences being 1 and least occurrences being 5.

```
Number of classes:              10
Biggest class:   1   occurrence:  6075
Smallest class:  5   occurrence:  4889
Number of no images with no object:  5400

Text(0.5, 0, 'Labels')
```

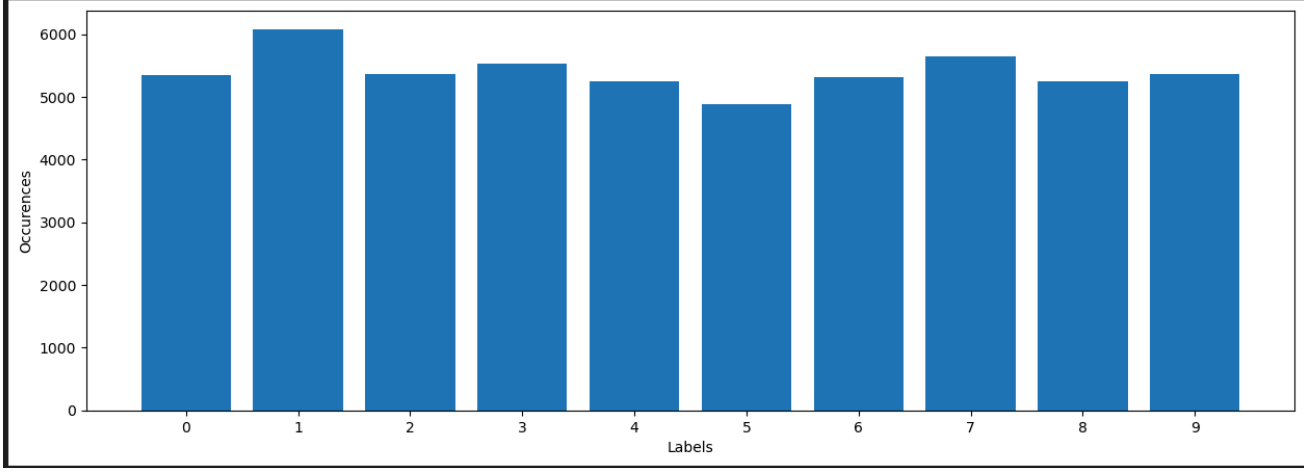


Figure 3: Class distribution of localization training dataset

Preprocessing

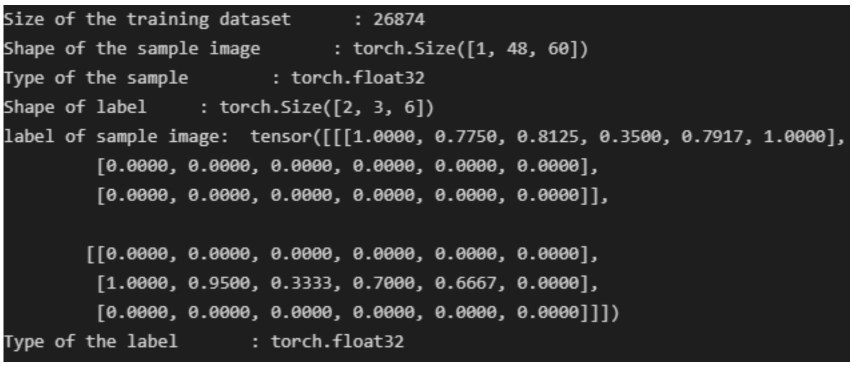
For our preprocessing in section 2 we preprocessed the data by normalizing it, this involves scaling the pixels to have a mean of 0 and a standard deviation of 1. This was achieved by using the Normalizer and Compose from techvisions transforms package.

Section 3

For section 2, object localization we did some data analysis by looking at the size and shape of the data, plotted some images and the corresponding bounding boxes of each class. We preprocessed the labels like the task 3.1.1 asks for and then we normalized the data to have a mean of 0 and a standard deviation of 1.

Data analysis

As in section 2 we looked at the size and shape of the data, figure 4 shows this. We did not plot the class distribution as this could become a bit complicated to implement since images can now have several objects.



```
Size of the training dataset      : 26874
Shape of the sample image       : torch.Size([1, 48, 60])
Type of the sample        : torch.float32
Shape of label     : torch.Size([2, 3, 6])
label of sample image:  tensor([[[1.0000, 0.7750, 0.8125, 0.3500, 0.7917, 1.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000]],

        [[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [1.0000, 0.9500, 0.3333, 0.7000, 0.6667, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000]]])
Type of the label       : torch.float32
```

Figure 4: Shape and size of detection dataset

As in section 2 we also plotted some images. This time we also plot the label of the object beside the bounding box since there can be more than one object in our pictures now.

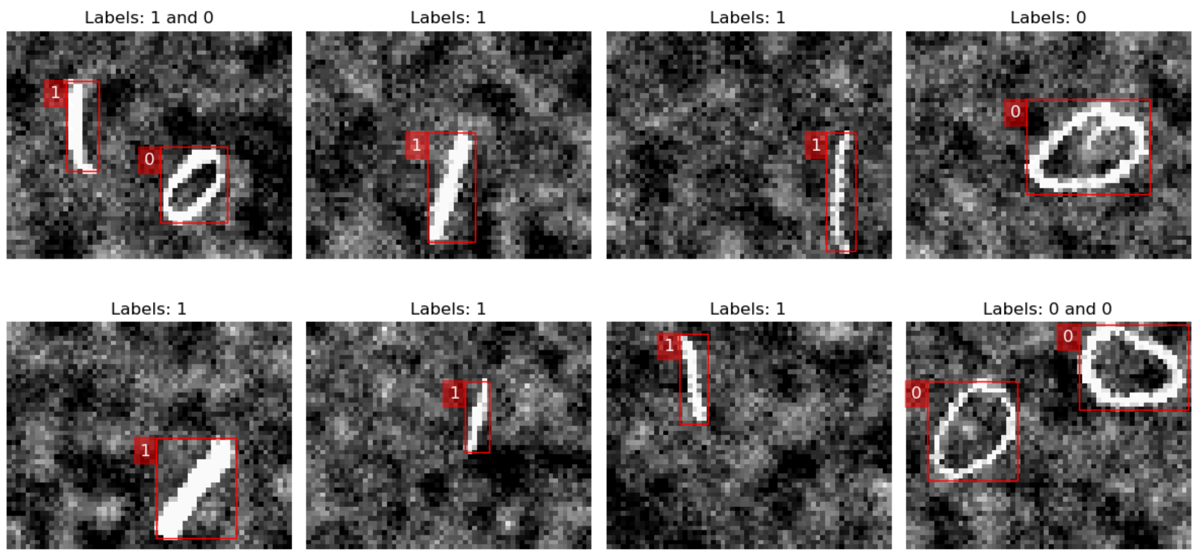
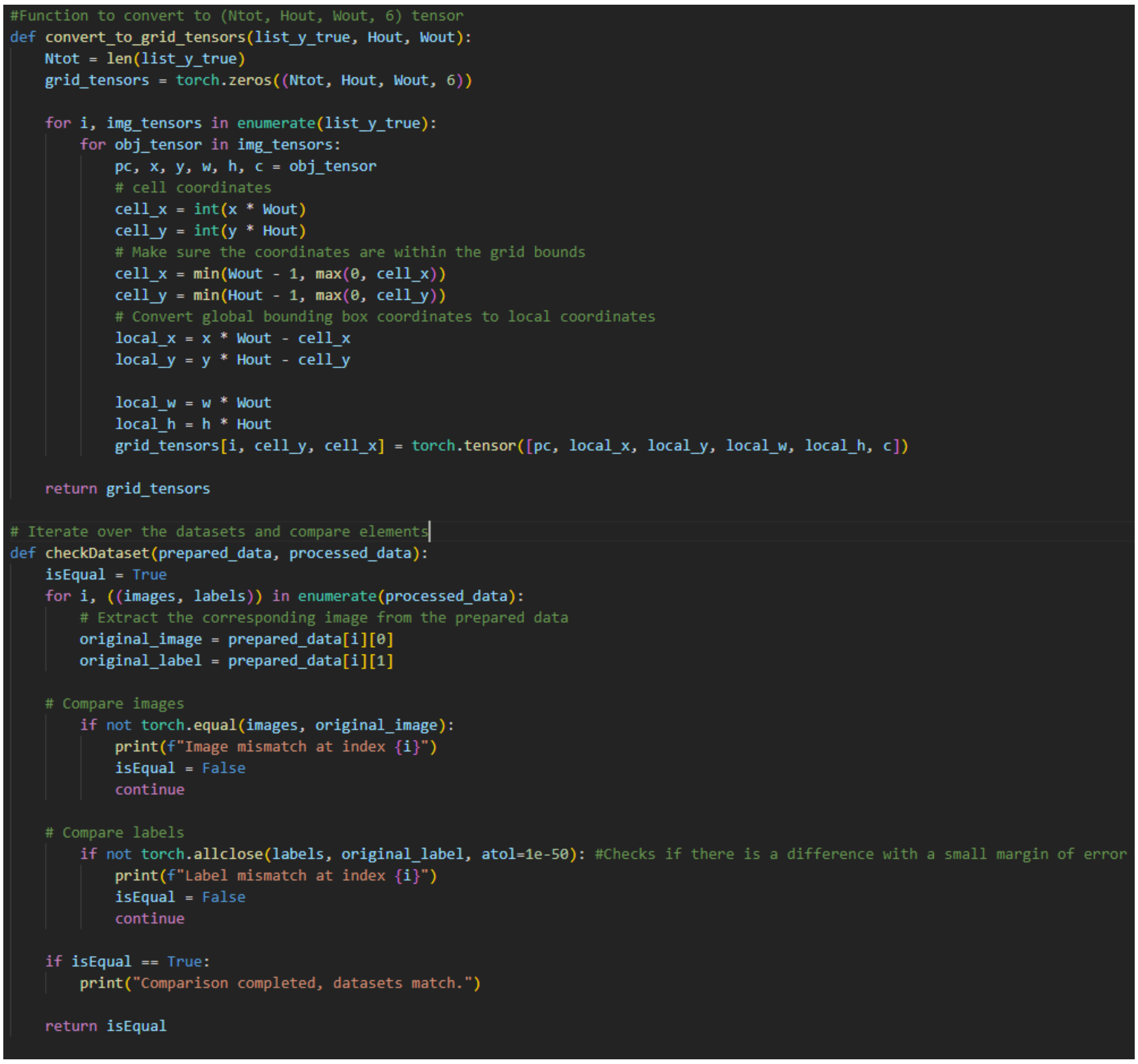


Figure 5: Plotted images and their corresponding bounding boxes from training data

Preprocessing

We preprocessed the labels by transforming the list_y_xxx.pt data into (N_tot, H_out, W_out, 6) tensor, then we created a new tensor data set where we added images from detection_xx.pt and the corresponding transformed labels from list_y_xx to get a combined dataset with images and labels. We then used a function, **checkDataset(prepared_data, processed_data)** to check that our newly created dataset was created correctly by comparing it to the detection_xx.pt dataset and checking that there are no differences. Figure 6 shows the functions for transforming the labels and comparing the datasets



```python
#Function to convert to (Ntot, Hout, Wout, 6) tensor
def convert_to_grid_tensors(list_y_true, Hout, Wout):
    Ntot = len(list_y_true)
    grid_tensors = torch.zeros((Ntot, Hout, Wout, 6))

    for i, img_tensors in enumerate(list_y_true):
        for obj_tensor in img_tensors:
            pc, x, y, w, h, c = obj_tensor
            # cell coordinates
            cell_x = int(x * Wout)
            cell_y = int(y * Hout)
            # Make sure the coordinates are within the grid bounds
            cell_x = min(Wout - 1, max(0, cell_x))
            cell_y = min(Hout - 1, max(0, cell_y))
            # Convert global bounding box coordinates to local coordinates
            local_x = x * Wout - cell_x
            local_y = y * Hout - cell_y

            local_w = w * Wout
            local_h = h * Hout
            grid_tensors[i, cell_y, cell_x] = torch.tensor([pc, local_x, local_y, local_w, local_h, c])

    return grid_tensors

# Iterate over the datasets and compare elements
def checkDataset(prepared_data, processed_data):
    isEqual = True
    for i, ((images, labels)) in enumerate(processed_data):
        # Extract the corresponding image from the prepared data
        original_image = prepared_data[i][0]
        original_label = prepared_data[i][1]

    # Compare images
        if not torch.equal(images, original_image):
            print(f"Image mismatch at index {i}")
            isEqual = False
            continue

    # Compare labels
        if not torch.allclose(labels, original_label, atol=1e-50): #Checks if there is a difference with a small margin of error
            print(f"Label mismatch at index {i}")
            isEqual = False
            continue

    if isEqual == True:
        print("Comparison completed, datasets match.")

    return isEqual
```

Figure 6: Functions to convert and check list_y_xxx.pt datasets

Implementation

Section 2

To implement and train several convolutional models for object localization we created.

- A convolutional model class for object localization
- Loss function class that implemented loss as described in the task.
- A training function to train our model
- Evaluation function to evaluate our models on validation data
- Trained some models with different hyperparameters and chose the best one based on validation performance that is the mean of accuracy and Iou.

After finding the best model we evaluate it on unseen test data, and plot both the predicted bounding box and the true bounding box.

Section 3

Implementation for object detection is done in a similar way as task 2.

- A convolutional model class for object detection

- A loss function that is based on the loss function specified in task 2, but we sum the location loss for each grid cell.

- Training function to train our model.

- Evaluation function to evaluate our models on validation data.

- Trained some models with different hyperparameters and chose the best one based on validation performance, that is a combination of IoU, validation accuracy and detection accuracy. Val_accuracy is the accuracy of correctly predicted labels, while detection accuracy is the accuracy of correctly identifying the amount of objects in the image.

After finding the best model we do the same as in task 2. We evaluate it on unseen test data, and plot both the predicted bounding box and the true bounding box as well as true and predicted labels for images with one or more objects.

Design choices

Section 2

Loss function

For our loss function we split our loss function into 3 components:

- Detection loss with nn.BCEWithLogitsLoss()
- Localization loss with nn.MSELoss()

- Classification loss with nn.CrossEntropyLoss()

We then add all the three losses into a total loss. We set our localization and classification loss to 0 if there is no object in the image.

Convolutional model

The convolutional model is designed to be relatively simple yet functional for the task at hand. It consists of:

- 3 convolutional layers with kernel size 3x3, stride 1 and padding 1.
- MaxPool to reduce the spatial dimensions after each of the convolutional layers
- Then we split the model into 3 paths for the three different predictions we want

- 2 linear layers for image classification
- 1 linear for object presence
- 2 linear for bounding box

At the end we do torch.cat for the three different predictions we made to get our final prediction. The number of neurons chosen aims to balance generalization without under or overfitting, and the choice came down to trial and error based on the performance on the validation data set.
We use ReLU as activation functions to introduce non linearity. And sigmoid activation function for object presence and bounding box coordinates to ensure that they stay within a valid range of 0 to 1.

Performance measure

For object localization we need to measure how good the model predicts classes and how good the model predicts bounding boxes. For this class prediction we used accuracy and for bounding boxes we used Intersection over union (IoU). We calculate each of the measures for all the predictions the model makes and then divide it by the total number of predictions to get model accuracy and model IoU. After we have those two measures we calculate the mean of both model accuracy and model IoU to get a combined performance measure that we use when choosing the best model on validation data.

Hyper Parameters

For training models we used **20 epochs** and **Adam** as our optimizer.
We trained a total of 9 models by looping through a list of different batch_sizes and learning rates:
batch_size = [32, 64, 128] and lr_list = [0.1, 0.01, 0.001]

We chose **Adam** due to its efficiency and it also provided some better results than when we first tried SGD. A good idea would probably be to test all 9 models on both Adam and SGD, but this would double the amount of models to train, due to time constraints this was not done.

We chose **epochs** as 20 since we saw little improvement in loss after 20 epochs.
We chose our **batch size**s to start at 32 and double each time up to 128 to balance computational time and performance.

We chose our **learning rates** to test from a bigger learning rate down to a smaller one, doing this we could check how sensitive our model is to the learning rate. Although it could be a good idea to test with learning rates that do not include such a broad spectrum, and maybe even going smaller than 0.001. This was ultimately not tested, again due to time constraints.

Section 3

Convolutional model

The convolutional model used in section 3 is aimed to be simple, while still being effective and handling the correct dimensions. It has the following structure:

- 3 convolutional layers, with kernel size 3x3, stride 1 and padding 1.
- MaxPool to reduce the spatial dimensions after each of the convolutional layers
- AdaptiveMaxPool will perform a max pool but will also have a fixed output size. The model uses AdaptiveMaxPool(2,3) since we use Hout=2 and Wout=3 (for the grid dimensions).
- A final convolutional layer with output C+5 channels. These 5 additional channels are for the bounding box parameters and pc.

In the forward pass every convolutional layer is followed by a ReLU function to introduce non-linearity. The adaptive pool is used to ensure the fixed output size of (2,3), while the final convolutional layer combines the result into C+5 output channels. permute is used in the end to rearrange the channels in the correct order.

Loss function

For the loss function we still split into 3 components:

- Detection loss with nn.BCEWithLogitsLoss()
- Localization loss with nn.MSELoss()
- Classification loss with nn.CrossEntropyLoss()

Since we now may have more than one object per image, we loop through each grid cell and calculate the loss for each of them. This is calculated using the 3 components described above. Localization loss and classification loss is 0 if there is not an object in the grid cell. The three components are added to the cell loss and then after added to the total loss.

Performance measure

For object detection the choice of performance measure is not as clear as in object localization. This is since the model, in addition to localizing and predicting the class label, needs to predict how many objects there are in the image. Mean average precision is a performance measure used for object detection, however since that can be a little bit technical to implement while not being required in this assignment, we decided to go for a more straightforward performance measure. We measure how good the model predicts boundary boxes (IoU), the accuracy of correctly classified label for every box and if it identifies the correct amount of objects. We then combine the three measures to a combined performance measure that is used to choose the model that performs best on validation data.

Hyper Parameters

For training several image detection models we used **15 epochs** and **Adam** as our optimizer.
We trained a total of 9 models by looping through a list of different batch_sizes and learning rates:
batch_size = [32, 64, 128] and lr_list = [0.01, 0.001, 0.0005]

We chose **Adam** in this section as well due to its efficiency and it also provided some better results than when we first tried SGD. As in section 2, we could have tried both SGD and Adam on all 9 models, but chose not to due to time constraints.

We chose **epochs** as 15 since we saw little improvement in loss after 15 epochs.
We chose our **batch size**s to start at 32 and double each time up to 128 to balance computational time and performance.

We chose our **learning rates** to test from a bigger learning rate down to a smaller one, doing this we could check how sensitive our model is to the learning rate. This time we used a learningrates 0.01, 0.001 and 0.0005 Instead of 0.1, 0.01 and 0.001 like we did in section 2. This is because we saw very poor results in our training/validation losses and performance using a learning rate of 0.1 and 0.0005.

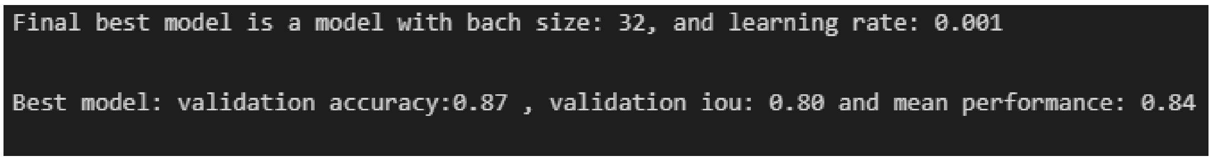
Again the same goes here as in section 2. To tune our hyperparameters further we could test with more hyper parameters and try to train more models than just 9. However due to the training time this was not feasible in this project

Results

Section 2

After training all of our 9 models and finding the best model on performance based on validation data we print the best model validation performance, hyperparameters as seen in figure 7 and then plot the loss of the best model, this is seen in figure 8.

As we can see from the plot both the validation loss and training loss starts to decrease from epoch 1 to around epoch 7, where the validation loss starts to increase while the training loss continues to decrease. The fact that the training loss decreases until slightly flattening out after around 18 epochs shows that training for more epochs would not result in significant loss decrease.
While the fact that the validation loss increases around epoch 7 could be a sign of overfitting, a possible solution to this would be to use the regularization technique, early stopping. In early stopping we stop the learning when the validation loss begins to increase.



```
Final best model is a model with bach size: 32, and learning rate: 0.001

Best model: validation accuracy:0.87 , validation iou: 0.80 and mean performance: 0.84
```

Figure 7: Result after training several models and choosing best based on validation data

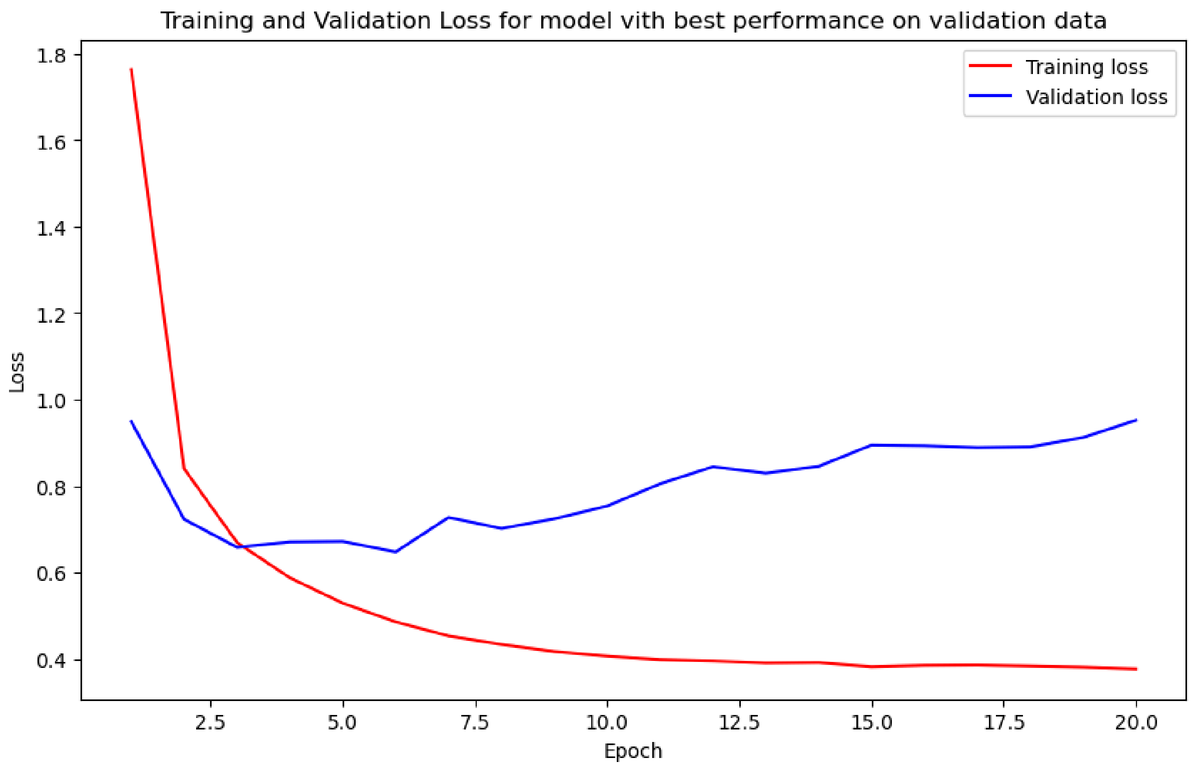
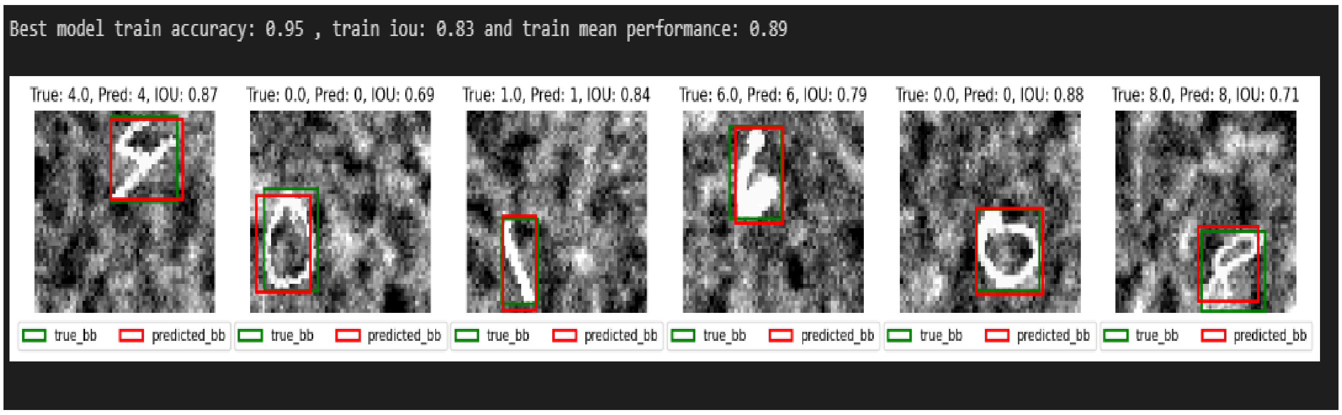


Figure 8: Training and validation loss of best chosen image localization model

After finding the best model, we plot some results with the predicted and true class as well as the predicted and true bounding box. We do this for training, validation and test data. Figures 9 to 10 show our results on training and validation data.

<u>Training data with true and predicted bounding box</u>

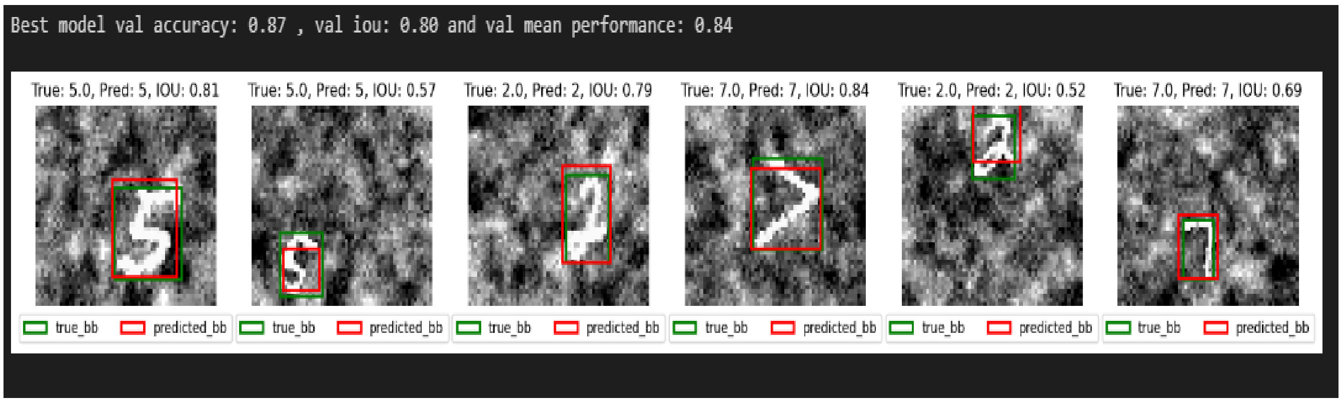


```
Best model train accuracy: 0.95 , train iou: 0.83 and train mean performance: 0.89
```

True: 4.0, Pred: 4, IOU: 0.87
True: 0.0, Pred: 0, IOU: 0.69
True: 1.0, Pred: 1, IOU: 0.84
True: 6.0, Pred: 6, IOU: 0.79
True: 0.0, Pred: 0, IOU: 0.88
True: 8.0, Pred: 8, IOU: 0.71
true_bb
predicted_bb

Figure 9: Showing the chosen best models accuracy, iou and mean of those two as well as predicted and true bounding boxes on training data for image localization model

<u>Validation data with true and predicted bounding box</u>

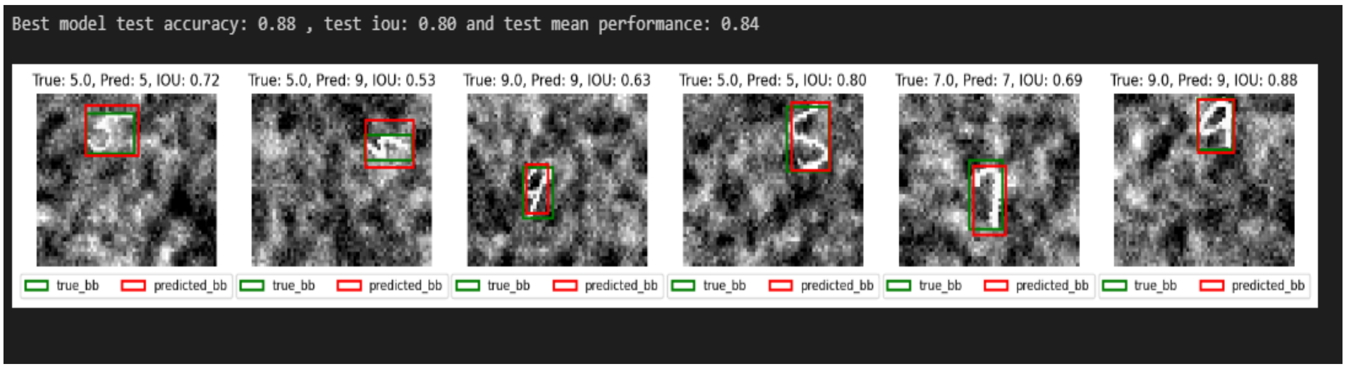


```
Best model val accuracy: 0.87 , val iou: 0.80 and val mean performance: 0.84
```

True: 5.0, Pred: 5, IOU: 0.81
True: 5.0, Pred: 5, IOU: 0.57
True: 2.0, Pred: 2, IOU: 0.79
True: 7.0, Pred: 7, IOU: 0.84
True: 2.0, Pred: 2, IOU: 0.52
True: 7.0, Pred: 7, IOU: 0.69
true_bb
predicted_bb

Figure 10: Showing the chosen best models accuracy, iou and mean of those two as well as predicted and true bounding boxes on validation data for image localization model

<u>Test data with true and predicted bounding box</u>



```
Best model test accuracy: 0.88 , test iou: 0.80 and test mean performance: 0.84
```

True: 5.0, Pred: 5, IOU: 0.72
True: 5.0, Pred: 9, IOU: 0.53
True: 9.0, Pred: 9, IOU: 0.63
True: 5.0, Pred: 5, IOU: 0.80
True: 7.0, Pred: 7, IOU: 0.69
True: 9.0, Pred: 9, IOU: 0.88
true_bb
predicted_bb

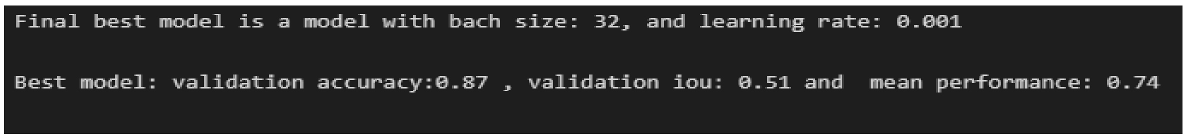
Figure 11: Showing the chosen best models accuracy, iou and mean of those two as well as predicted and true bounding boxes on test data for image localization model

Evaluation on unseen test data

The evaluation on test data that we can see from figure 11 shows a mean performance of 0.83 with accuracy being higher than IoU. We see that the class predictions seem to be going well, while the results of the bounding boxes fluctuate a bit. From the few examples we plotted, we observe IoU as high as 0.88 and as low as 0.53. In picture number two from the left we have a sideways object of class 5, the model seems to be predicting the class well, however it seems to have issues fitting a good bouncing box with only a IoU of 0.53.The results on unseen test data show that the model does a fairly good job in image classification, but could use some fine tuning of the bounding box predictions to make it a good model for image localization.

Section 3

We trained 9 different models with different hyperparameter combinations. Then, model selection is done by choosing the model that performs best on validation data using the performance measure described. Results and optimal hyperparameters are shown in figure 12. Training and validation loss for the chosen model is shown in figure 13.



```
Final best model is a model with bach size: 32, and learning rate: 0.001

Best model: validation accuracy:0.87 , validation iou: 0.51 and  mean performance: 0.74
```

Figure 12: Results and hyperparameters for the best model based on performance on validation data

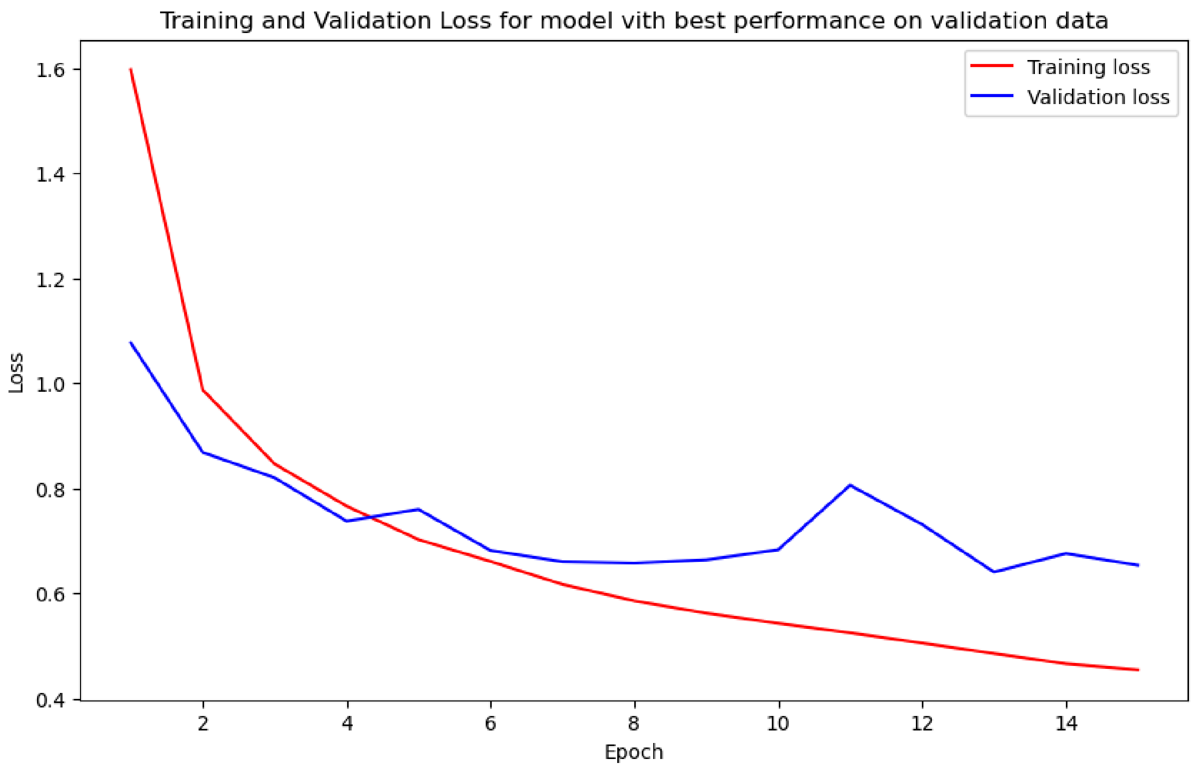
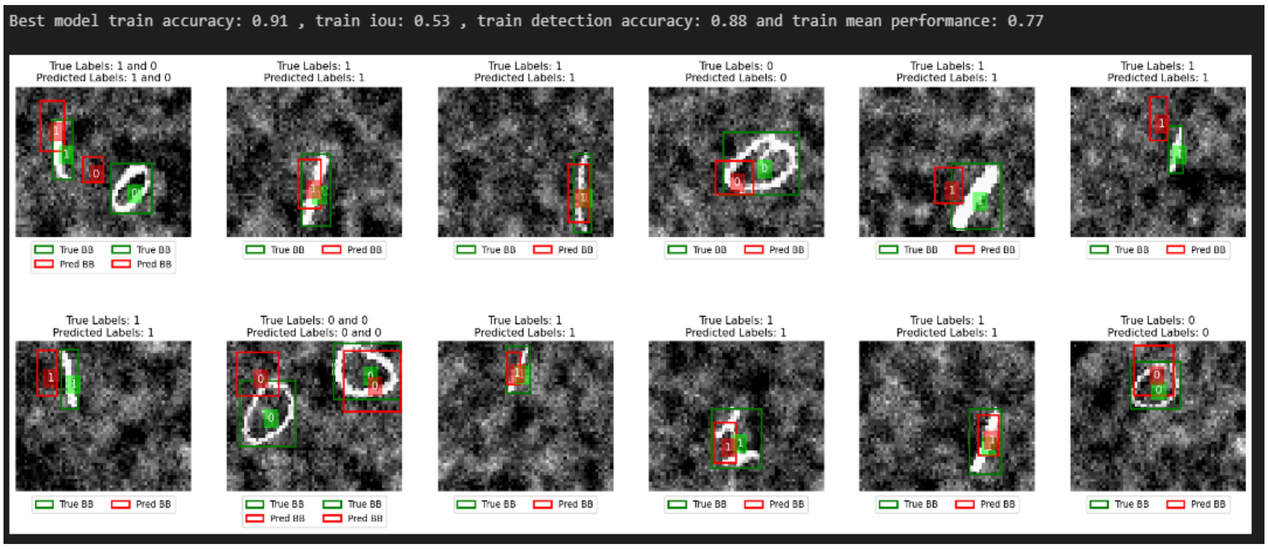


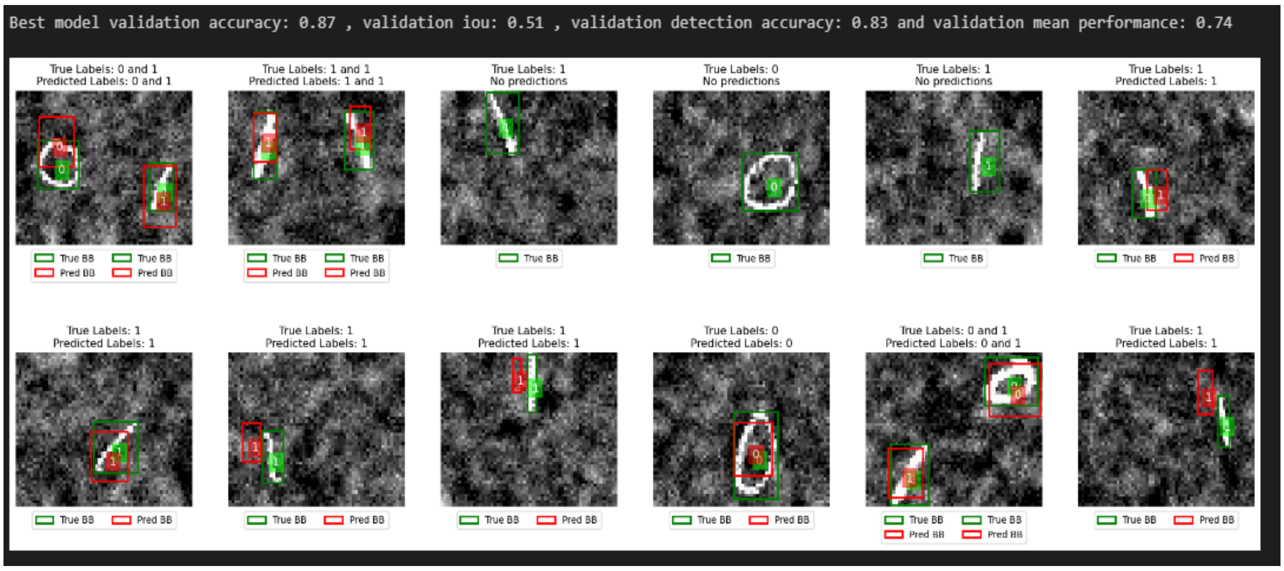
Figure 13: Training and validation loss of best chosen image detection model

After finding the best model, we plot some results with the predicted and true classes as well as the predicted and true bounding boxes. We do this for training, validation and test data. Figures 14, 15 and 16 show our results on training, validation and test data respectively.



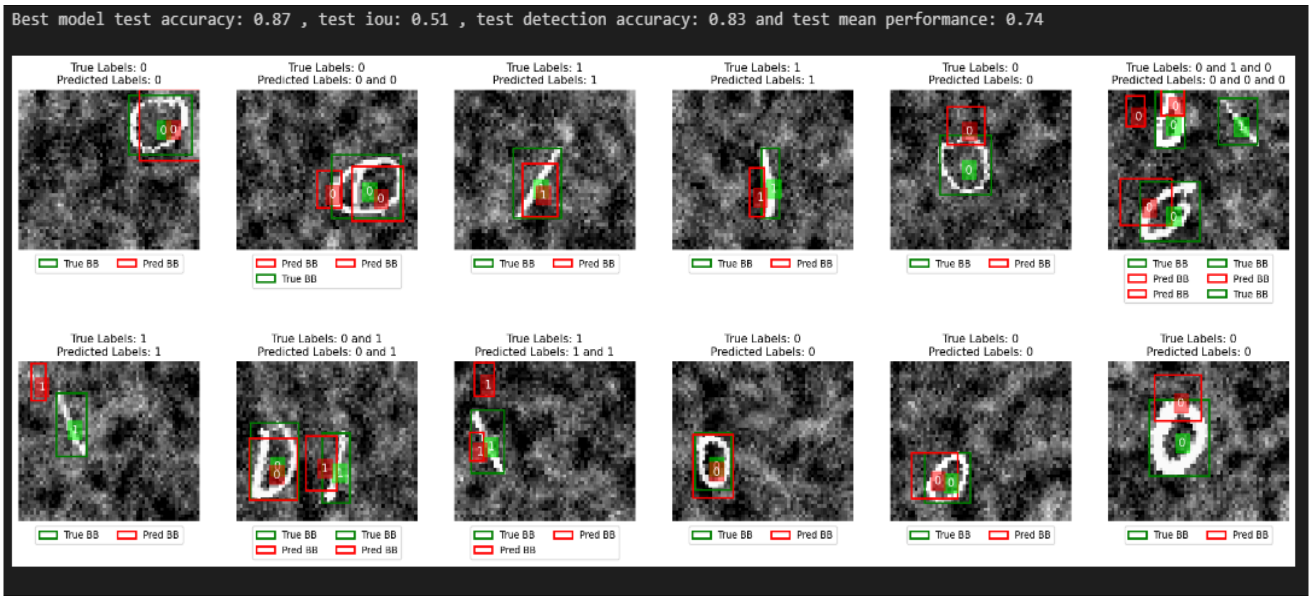
```
Best model train accuracy: 0.91 , train iou: 0.53 , train detection accuracy: 0.88 and train mean performance: 0.77
```

Figure 14:Showing the chosen best models classification accuracy, iou, detection accuracy and mean of those three as well as predicted and true bounding boxes on training data for image detection model



```
Best model validation accuracy: 0.87 , validation iou: 0.51 , validation detection accuracy: 0.83 and validation mean performance: 0.74
```

Figure 15:Showing the chosen best models classification accuracy, iou, detection accuracy and mean of those three as well as predicted and true bounding boxes on validation data for image detection model



```
Best model test accuracy: 0.87 , test iou: 0.51 , test detection accuracy: 0.83 and test mean performance: 0.74
```

Figure 16: Showing the chosen best models classification accuracy, iou, detection accuracy and mean of those three as well as predicted and true bounding boxes on test data for image detection model

Evaluation on unseen test data

The results from figure 16 gives us a test accuracy of 0.87, IoU mean of 0.51, detection accuracy 0.83 and total mean performance of 0.74. The model is good at classifying correct labels with an accuracy of 0.87, and the model does decently with predicting the correct amount of objects 83% of the time. However, it seems like our models struggle a bit with imprecise bounding boxes. By looking at the results in the plots and the IoU of 0.51, we notice that the model fits some boundary boxes well while others are predicted outside of the actual boundary box. The results show that the model does a good job classifying and detecting the correct amount of objects, but has clear improvement when it comes to fitting good boundary boxes.

Possible improvements

Our implementations are not perfect and there definitely are some improvements we could consider given enough time.

Section 2

Some improvements we could consider for section 2 includes:

- Tried different designs of networks with different numbers of hidden layers and adjusted the kernel, stride and padding size to see if we got better results.
- Used different hyperparameters than only the few we used from batch size and learning rate.
- Used different epochs or optimizers
- Do some data augmentation, like for example augmenting the existing dataset with some slight differences, like rotation of the images and the corresponding bounding boxes, this would allow us to have more data to train on.
- Introduce some regularization techniques like for example early stop to reduce the risk of overfitting

Section 3

Given enough time and resources, there are some potential ideas we could try to improve our results:

- Try different architectures for our convolutional model. Our architecture is fairly simple, and it is possible that with a different design with a different amount of layers and different values in kernel, stride and padding could give better results.
- Try more hyperparameter combinations (including optimizers).
- Change the amount of epochs
- Implementing early stopping can help reduce the risk of overfitting making the model potentially do better on validation and test data. This is especially important if a high number of epochs are chosen.
- Change the loss function to prioritize differently, add weights to the 3 different losses. An idea can be to for example weigh IoU higher, such that the model gets penalized more for bad IoU results and then learns from it.