Report INF265 project02

Task 2 - Object localization

1) Approach and design choices

We struggled quite a lot with the issue that the models wouldn't learn (loss function issues we believe). Therefore, our first model was designed to overfit to the training data and then see if we could modify it. The model had four convolutional layers with pooling after each layer and ReLU activation functions. It also had three fully connected layers in the end, with ReLU activations for the two first. We used nn.BCEWithLogitsLoss and nn.CrossEntropyLoss, so that the output layer didn't need any activations. However, right before training it, we changed our minds about the overfitting and decided to add some momentum to the optimizer in the training. This seemed to be an okay model, but we wanted to try a different architecture for model 2 anyways. Model 2 had a very low training loss. Thus, for model 3 we used the same architecture, but added some regularization (weight decay). This, however, did not work as planned, and the training loss kept fluctuating around 2.5. As this model turned out catastrophic when computing validation accuracy and IoU, we tried a different architecture for the next model. For model 4 we added one more convolutional layer and added pooling to the first and third convolutional layers. Here, we did not use regularization in the training. The training loss of model 4 seemed to develop healthy, steadily decreasing and ending at around 0.8. It proved to perform quite poorly on the validation set, so we tried one more model. This model (5), also performed poorly on the validation set. And even though the training loss was quite low, it also performed poorly on the training data. We thus made one last model (6), with more convolutional layers, to try and see if the depth of the network was the issue.

2) Models and hyperparameters

Model 1: The first model had four 2D convolutional layers with 16, 32, 64 and 128 out channels respectively. All of them with kernel size 3x3, padding = 1, and no stride. The max-pooling layer between each convolutional layer had kernel size 2 and stride 0. The first fully connected layer had 120 neurons, the second had 84 neurons. The output layer had 15 neurons, obviously. For the training, we used stochastic gradient descent with a

learning rate of 0.1 and momentum 0.6. All layers, but the last one, had ReLU activations.

Model 2: The second model had 2 convolutional layers with 16 and 32 out channels respectively, both with kernel size 3x3, padding = stride = 0. After the second convolutional layer, there was a maxpool layer with kernel size = stride = 2. After this, two fully connected layers with 120 and 15 neurons. For the training, we used stochastic gradient descent with a learning rate of 0.1. All layers, but the last one, had ReLU activations.

Model 3: The third model had the same architecture as model 2. For the training, we used stochastic gradient descent with a learning rate of 0.1 and added weight decay = 0.3.

Model 4: The fourth model had 3 convolutional layers with 16, 32 and 64 out channels respectively, all of which had kernel size 3x3, padding=stride=0. After the first and third layer, we added max pooling with kernel size=stride=2. Then, we used two fully connected layers, with 120 and 15 neurons, respectively. For the training, we used stochastic gradient descent, with a learning rate of 0.01. All layers, but the last one, had ReLU activations.

Model 5: The fifth model had three convolutional layers with 16, 32 and 64 out channels respectively. The first layer had kernel size 5x5, while the second and third had kernel size 3x3, padding=stride=0. After the first and third layer, we added max pooling with kernel size=stride=2. Then, we used two fully connected layers, with 120 and 15 neurons, respectively. For the training, we used stochastic gradient descent, with a learning rate of 0.01. All layers, but the last one, had ReLU activations.

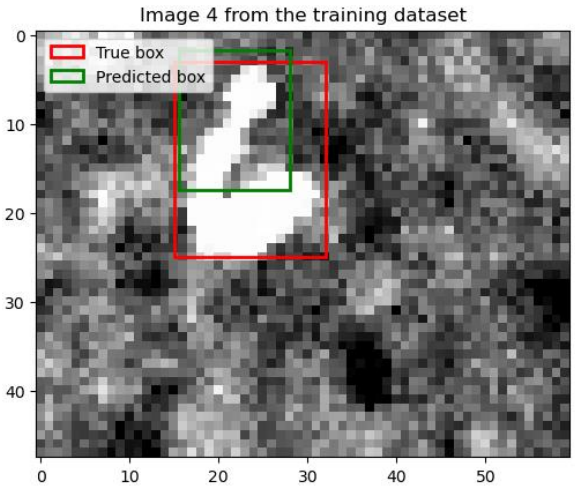
Model 6: For the sixth model we used an architecture with four convolutional layers. The first layer had 16 out channels, kernel size 5x5 and padding=2. The second layer had 32 out channels, kernel size 3x3 and padding=1. The third layer had 64 out channels and kernel size 3x3. The fourth layer had 128 out channels, and kernel size 3x3. We used pooling after the third and fourth layer, with kernel size=stride=2. The first fully connected layer had 512 neurons, and the second layer had 15 neurons. For the training

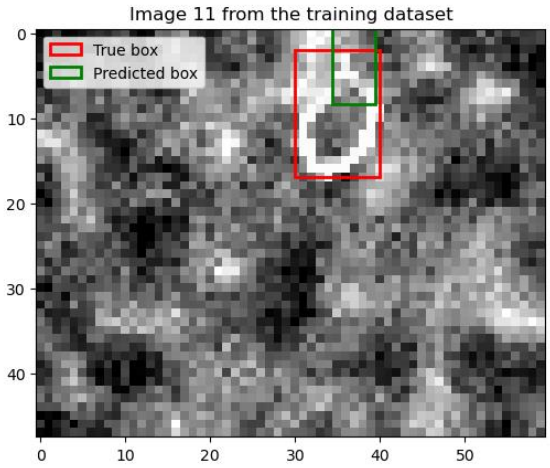
we used stochastic gradient descent, with a learning rate of 0.01. All layers, but the last one, had ReLU activations.

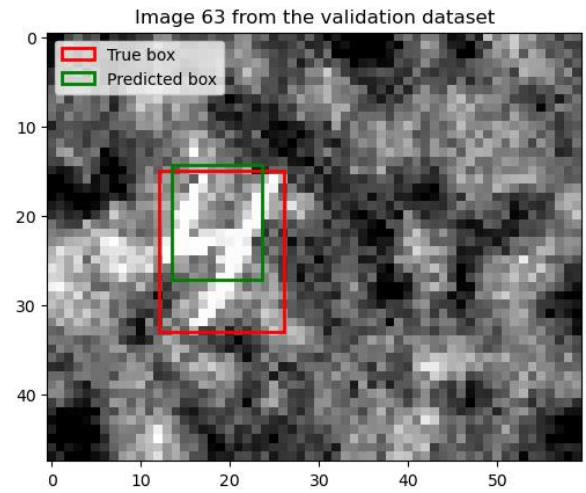
3) Performance of selected model

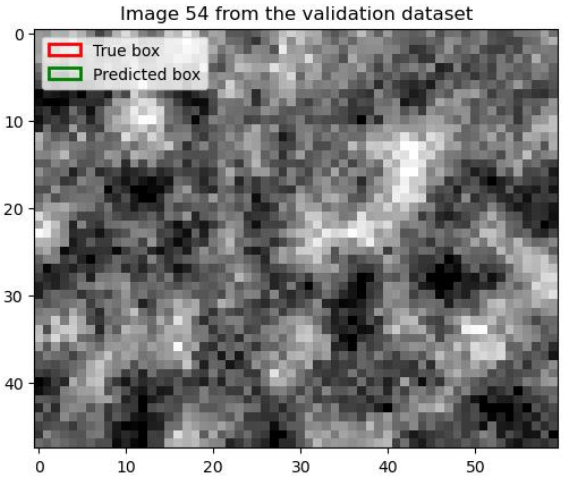
The model performed quite poorly. It had a decent percentage of correct presence and class predictions (0.734), but a quite low IoU score (0.339). When plotting some of the images, it seemed to be onto something, but didn't quite match the true box. The average between IoU and accuracy was 0.536.

4) Images and predictions from the validation and training set









The model managed to predict the correct label for all of the images above.

5) Comments on the results

The models did okay on the classification part of the task, predicting object presence and class. However, their ability of predict bounding boxes was not very good. The strange thing is that it performed almost as well on the training set as the test set (when regarding the bounding boxes). When printing all of the predicted values in one batch, we saw that the values were almost equal for all predictions. The network seemed to not learn the features of the box.

Task 3 - Object detection

1) Approach and design choices

We did not want to get stuck on task 3.1.1, so we started with 3.1.2 using the preprocessed labels. However, we had time to do 3.1.1 afterwards, and so we just did some checks to see that we managed to preprocess the y_true_list correctly. Also, the models in this task outputs 7 numbers for each grid cell. We initially thought that it should output 6 numbers, using BCELossWithLogits for the classification part, but interpreted that the model should output a tensor of size C+5 as 7. Thus, we also used CrossEntropyLoss. This might have affected the classification (as we saw that it almost exclusively predicted class 1).

After struggling so much with the loss in task 2, we decided to do it simpler, but more computationally expensive (using 4 nested for loops). We started out using a quite simple model with 4 convolutional layers, with ReLU activations and pooling layers after each one. Also, we struggled quite a bit getting the right output size, but just permuting the output seemed to work fine.

For the second model we added a layer and experimented with different kernel sizes, as the first model seemed to underfit the training data. This model seemed to perform quite well, and thus we chose not to make another model.

2) Models and hyperparameters

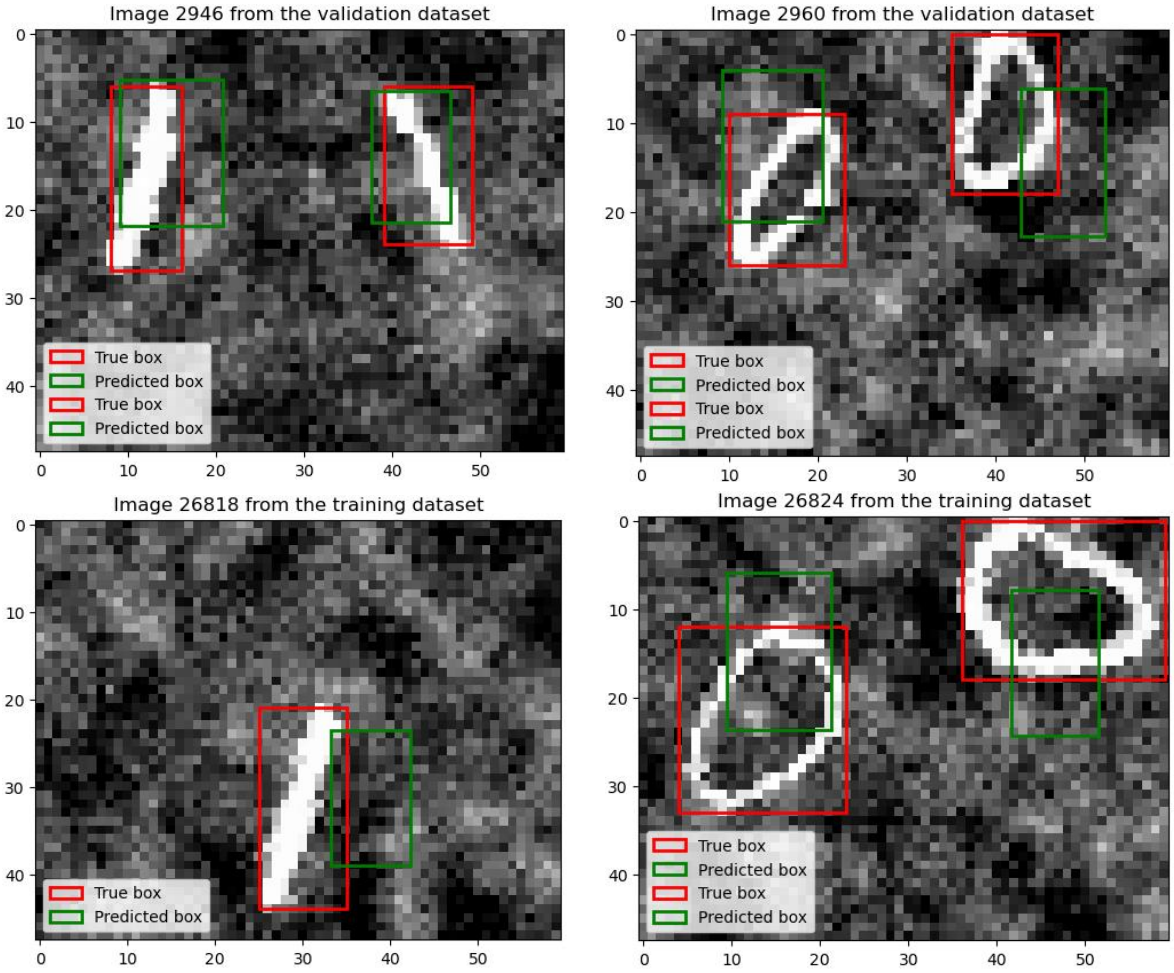
Model 1: The first model had 4 convolutional layers with 16, 32, 64 and 7 (C+5) out channels respectively. The first and second layer had kernel size 3x3, and stride=padding=1. The third layer had kernel size 3x5, stride (6, 5) and no padding. The fourth layer simply had kernel size 1x1. After each of the first three layers, there were ReLU activations, and a maxpool layer with kernel size = stride = 2. For the training, we used stochastic gradient descent with a learning rate of 0.001.

Model 2: The second model had 5 convolutional layers with 32, 64, 128, 256 and 7 out channels respectively. The first two layers had kernel size 5x5, stride = 1, and padding = 2. The third and fourth layer had kernel size 3x3 and padding = stride = 1. The fifth layer simply had kernel size 1x1 and stride = padding = 0. After the first, second and third convolutional layer, there was a ReLU activation, and a maxpool layer with kernel size = stride = 2. After the fourth convolutional layer, there was a ReLU activation, and a maxpool layer with kernel size 3x2, and stride 3x2. For the training, we used stochastic gradient descent with a learning rate of 0.1.

3) Performance of selected model

We chose model 2, and it performed surprisingly well. It had an accuracy on the test dataset = 0.968, and an average IoU of 0.655. The average of this was thus 0.811. The reason for the good accuracy was probably the large amount of grid cells not containing an object. When printing some outputs and labels, we saw that it performed pretty well on object presence in each grid cell. That the model performed so much better in terms of IoU than the object localization model was quite surprising. However, most of the IoU scores for the individual boxes were quite high (>0.7), so the score made sense.

4) Images and predictions from the validation and training set



The model predicted the correct label for both objects in image 2946 from the validation set. It predicted the correct label for one of the objects in image 2960 from the validation set. It predicted the correct label for the object in image 26818 from the training set. Unfortunately, it predicted wrong label for both objects in image 26824 from the training set.

5) Comments on the results

The reason for the good accuracy was probably the large amount of grid cells not containing an object. When printing some outputs and labels, we saw that it performed pretty well on object presence in each grid cell. However, we also saw that the model almost exclusively predicted label 1, so the class prediction part of the model was probably not very good. That the model performed so much better in terms of IoU than

the object localization model was quite surprising. However, most of the IoU scores for the individual boxes were quite high (>0.7), so the score made sense.