Project 2 INF265

Division of labour

Christopher mainly focused on the coding part of object localization.

Jonas mainly focused on the coding part of object detection and writing the report.

Both cooperate well and are good at distributing the work fairly.

NB: Christopher has taken this class before, therefore the report and Jupyter notebook can have many similarities to my previous hand-in.

Content

Introduction

In this report we are defining and training convolutional networks to solve an object localization task and an object detection task. In the first task, it is assumed that there is at most one object per image and in the second task there can be multiple. The objectives in this task included: a) getting a better understanding of convolutional neural networks b) object localization and c) object detection tasks. Some things that were included in the project were:

- Image dimension H_{in} X W_{in} are 48 X 60.
- Digits are randomly located in the image.
- Digits are randomly slightly rotated.
- Digits are randomly slightly resized (smaller or larger.
- Random noise is added in the background.

To each of the tasks, there were .pt files that were supposed to be loaded in that included the pictures as tensors.

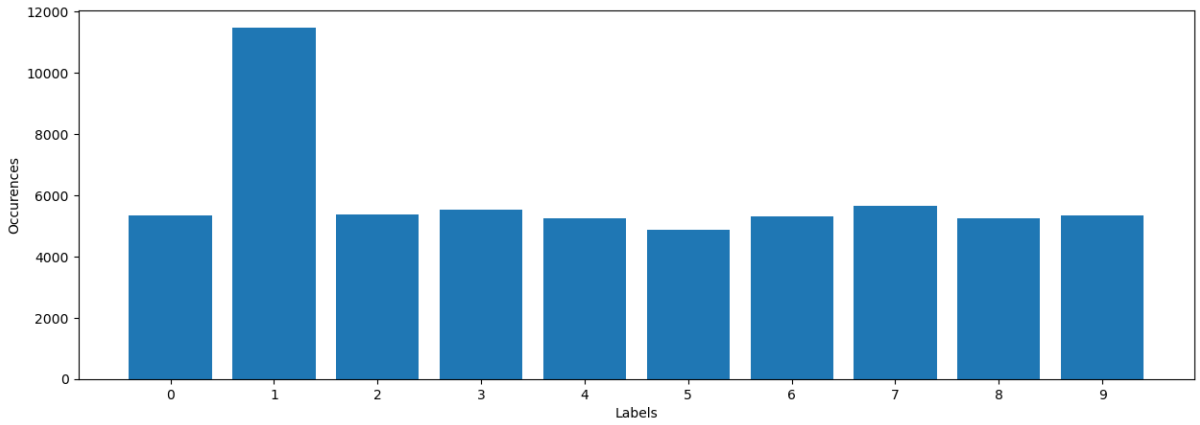
Approach and design choices

In the first task we loaded in the necessary libraries to complete the task, subsequently loading in the files that we were given using "torch.load()". After that we checked the size of the dataset, shape of the inputs, type of input, shape of the target and type of target. The reason for this was to get a better understanding of what type of data we were working on.
Afterwards we took the first image and used plt.show() to see what the picture looked like, subsequently making a code that showed how to visualize the bounding box annotations for the images provided. We extracted the x, y, w and h values from the label and calculated the top left and bottom right coordinates of the bounding box. We then used Matplotlib to plot the image and create a rectangle object using the bounding box coordinates. Finally, we added the rectangle object to visualize the annotation. This code was helpful in understanding the format of the annotations and checking if they were correct.

Visualization

Later we wanted to visualize the images. We did this in the function "plot_instances". In short it takes in the training data and a label for a specific class and plots a specified number of instances of that class using Matplotlib.

Afterwards we created three different functions: “compute_statistics”, “plot_histogram” and “analyse_dataset”. The “compute_statistics” function computes the basic statistics in the “analyse_dataset”, such as the standard deviation, the mean, minimum and maximum values of pixel intensities. The “plot_histogram” is a helper function to plot histograms of the number of instances in each class in the “analyse_dataset” function. The “analyse_dataset” function is a function that performs some basic analysis of an unknown dataset. It calls the “compute_statistics” to compute the basic statistics of the images, then counts the occurrences of each label in the dataset using the Counter class. It prints the number of classes in the dataset, the biggest and smallest class labels and their occurrences, and then class “plot_histogram” to plot a histogram of the number of instances in each class.



Plot from the notebook. It shows occurrences of each label in the training dataset. The division was approximately the same in validation and test (see notebook).

The different architectures

The first architecture we used is called “MyNet”. This architecture defines a neural network which inherits from nn.Module. This network is Inspired from YOLO. The class has five convolutional layers with 16, 32, 64, 128 and 256 output channels. The kernel size, stride, bias, and padding are set as arguments with default values of (3,3), (1,1) (1,1) and True. The convolutional layers are followed by five max pooling layers of kernel size=2 and stride=2, a flatten layer, and two fully connected layers with output of dimensions of 128 and 15, respectively. We found the different in_features and out_features as a result of different sources recommending them. We tested the values and it worked. The forward() method is defined to perform a forward pass through the network. The input tensors are passed through the first convolutional layer later followed by a ReLU layer and then a max pooling layer.

Then the output of the previous layer is passed through the next layer for all convolutional layers. The output of the last convolutional block is then flattened and passed through fully connected layers with ReLU activation function. The code then creates an instance of the MyNet class and passes an image through it.

The design choices we made for this model seemed reasonable according to the different sources explaining YOLO. In short, the convolutional layers learn spatial features of the input image, while the fully connected layers are supposed to make predictions from these features.

The next architecture is called “MyNet2”. In short, it is a similar implementation of “LeNet-5” architecture. The architecture consists of two convolutional layers, two average pooling layers and Tanh activation functions. For the convolutional layers, we have 6 and 16 as “out_channels”. Kernel size, stride and padding for the convolutional layers are (3,3), (1,1) and (0,0). While for the average pooling layers, the kernel and stride are (2,2) for both. For the fully connected layers, we added three layers instead of two. Where the output features are 512, 120 and 15.

For the last architecture called “MyNet3”, we implemented VGG16. However, we dropped the last convolutional to make it more computationally efficient. Also, we concluded that the extra computation probably was not necessary since the architecture is designed for 224x224 images, while ours are 48x60. This network consists of Three layers with two convolutional layers and ReLU activation functions each. Each layer is finished with max pooling layer with kernel size and stride of (2,2). The kernel size and stride for the convolutional layers are (3,3) and (1,1). For the three different layers, the out channels from convolutional layers are 16, 32 and 64. There are three fully connected layers with output features of 512, 120, 15.

Overall, we tried implementing YOLO, LeNet-5 and VGG16 with some modifications. We used these networks as they have been reported to have good results in such tasks. We implemented LeNet-5 to compare a less complex architecture with YOLO and VGG16, which are networks that are designed for bigger images. Comparing these networks gave us an insight into the trade-off between complexity and computational efficiency. We also tried adding and removing layers, and tuning the hyperparameters for the different layers to find the best match.

Hyper Parameters

MyNet:

- 5 convolutional layers, each with 16, 32, 64, 128, 256 output channels
- 2 fully connected layers, each with 128 and 15 output nodes
- Hyperparameters: kernel_size=3, stride=1, padding=1, bias=True

MyNet2:

- 2 convolutional layers, each with 6, 16 output channels
- 2 fully connected layers with 120 and 15 output nodes
- Hyperparameters: kernel_size=3, stride=1, padding=0, bias=True

MyNet2:

- 3 layers of:
    - 2 convolutional layers, each with 16, 32, 64
- 3 fully connected layers with 512, 120 and 15 output nodes
- Hyperparameters: kernel_size=3, stride=1, padding=0, bias=True

Training and performance

Custom loss

The “custom_loss” function defines a loss function that is used in object localization. It takes in two inputs: “outputs” and “labels” that correspond to the predicted outputs and the truth labels. The function computes three different losses: detection loss, classification loss and localization loss. The detection loss is computed using the binary cross entropy loss function. It is computed between the predicted detection and the ground truth detection score. The detection score is a binary value that indicates whether there is an object in the image or not. The classification loss is computed using the cross entropy function. It is computed between the predicted class probabilities and the truth labels in the grid cells where an object is detected. The localization loss is computed using the MSE loss function. It is computed between the bounding box coordinates and the truth bounding box coordinates where an object is present. The total loss is acquired by adding the classification loss, localization loss and detection loss together.

Calculate IoU

The “calculate_iou” function takes as input two bounding boxes represented by tensors: “predicted_box” and “target_box”. By using “max” and “min” it calculates the coordinates of the intersection rectangle between the two boxes. Then it calculates the area of the intersection rectangle using the “interArea” variable. Afterwards the function computes the areas of both predicted and target bounding boxes using arithmetic operations on the tensor elements. At last, the function calculates the intersection over union by the sum of prediction and truth areas minus the intersection area. It then returns the IOU value as a tensor. Overall, the design choice is a quite standardized way of calculating the IOU. The function takes in two tensors as input and returns a scalar value, making it easy to use in conjunction with loss functions or other PyTorch functions.

Train function

The “train” function is a standard train function for training neural networks. It takes as input the number of epochs to train for, the model, the loss function, an optimizer and the training data loader. The first line decides which device the computations will be carried out. Here it is set at the CPU. The function initializes the loss value to zero and the optimizer gradient to zero for each epoch. The loop over the epochs then begins. The training loops goes as follows:

1. The input and the labels are transferred to the device where the different computations will be carried out.
2. The output is computed by passing the input to the model.
3. The loss is computed between the output and the corresponding labels.
4. Whilst using backpropagation and whilst the optimizer is being updated, the gradients are computed.
5. The optimizer gradients are set to zero.

At the end of each epoch, the average training loss is computed and stored in “losses_train”.

The overall design choice is quite standard. The loss function is user-defined and the optimizer is passed as an argument, which allows for a better flexibility in the choice of

components. The “train” function only trains the function and does not perform validation or testing.

Train all function

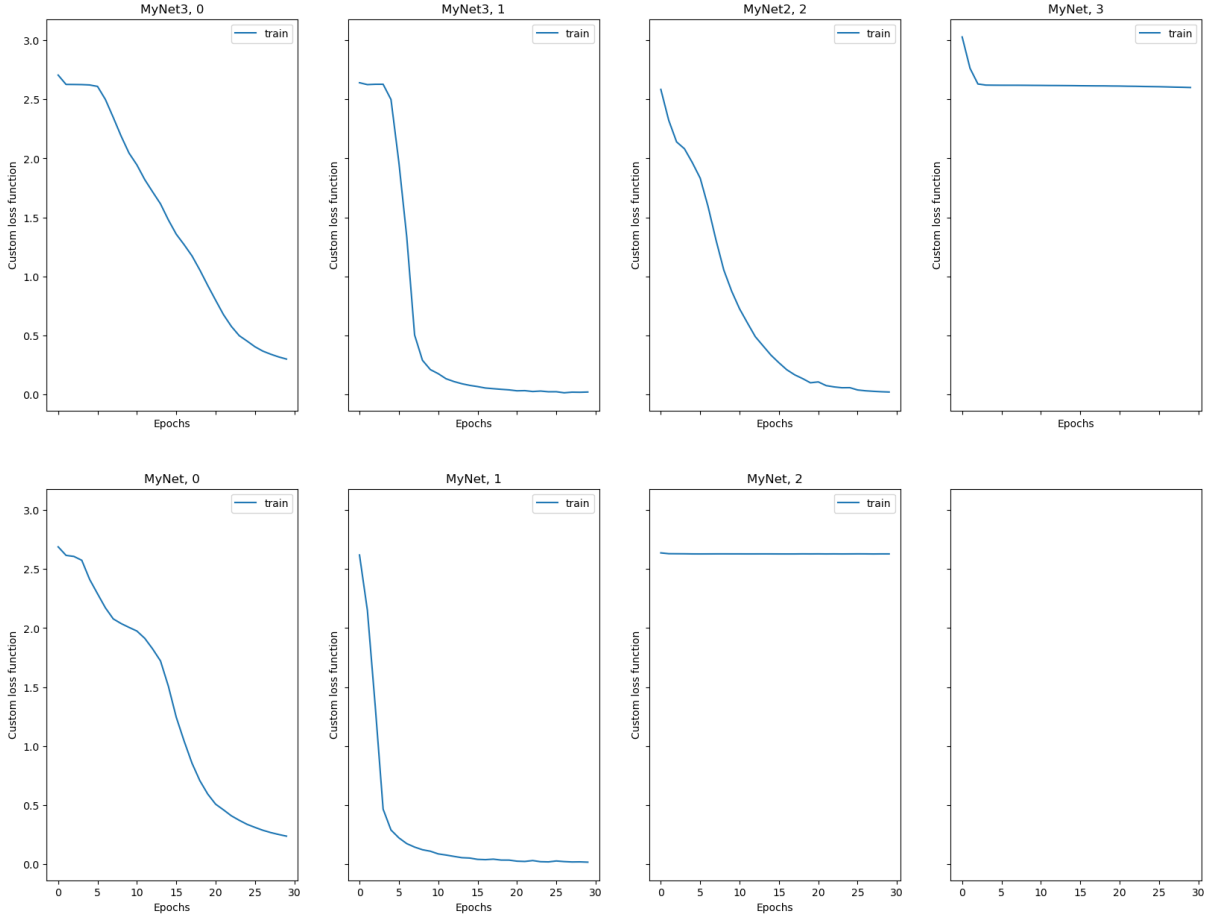
The “train_all” function trains different models and returns the training loss for each model. The reason for this is because we need to compare the different models to evaluate which one performs best.

With these models:

```
# ----------  MyNet() with diffenrent optimizers ----------

model = MyNet()
models.append(model)
optimizers.append(optim.SGD(model.parameters(), lr=1e-3))

model = MyNet()
models.append(model)
optimizers.append(optim.SGD(model.parameters(), lr=1e-3, momentum=0.9))

model = MyNet()
models.append(model)
optimizers.append(optim.SGD(model.parameters(), lr=1e-2, momentum=0.9))

model = MyNet()
models.append(model)
optimizers.append(optim.Adam(model.parameters(), lr=1e-2))
```

lr stands for “learning rate”. We have the exact same hyperparameters for both architectures. We tried four different models for each of the architectures with slightly different hyperparameters.

These were the loss results from the different models:



Notice that there are only seven models plotted, and not 12 (four for each architecture with different hyperparameters). When all 12 models where running, it cut short in the middle of running the first version of the “MyNet”. So we ran “MyNet3” and “MyNet2” with only the best hyperparameters, and all four combinations of “MyNet” again. We did this because it took nearly 24 hours to train all 12 models.

Computing the performance

The function “compute_performance” evaluates the performance of a trained model on a dataset by computing the accuracy of the model on the classification and localization. During the evaluation, we have used .eval() to make the process quicker, as well as using less memory. This happens because the method disables the gradient computations. The performance of the model is then evaluated on each batch of data in the input dataloader.

First, we extract the first value from the output tensor. The predicted label is acquired by using a sigmoid function to the output tensor at index 0, which corresponds to the objectness

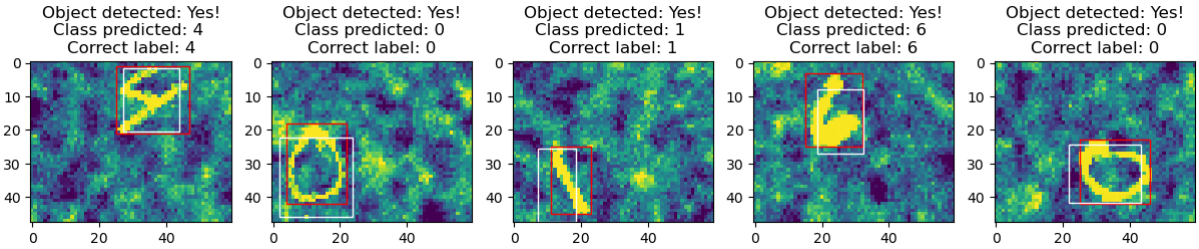
score. A predicted label of 1 is assigned if the objectness score is greater than 0.5, and 0 otherwise.

For the localization task, the predicted box coordinates are obtained from the output tensors at the indices 1 and 4. IOU is used to evaluate the accuracy of the localization task. Furthermore, the average accuracy is computed over all the batches of data using np.mean(). The function returns the accuracy of the classification and localization tasks, as well as printing it out. Lastly, we extract the final labels (index 5 to 15). The index with the highest value corresponds to which class has been predicted. This value is then compared to the true class.

Results

Predicting the subplots

"pred_subplots" will generate a set of subplots for predicted and true bounding boxes. The function takes in a data loader, a pre-trained model and a dataset as input. The function first sets the model in evaluation mode, which turns off the gradient calculation and dropout layers. It then iterates over the data loader to extract images and corresponding labels. Within each iteration it creates a Matplotlib figure and five subplots for each image in the batch. It will then extract the predicted and the true bounding boxes coordinates from the label tensor and the output. Using this, the function will calculate top-left and bottom-right coordinates of the bounding boxes for both true boxes and predicted ones. It then adds a rectangle object for the true label box and predicted box. Finally it displays the result using .show().



Subplots from the bottom of task 2 notebook. There are several more in the notebook

We did not try any other models other than the ones that are in the code. We got some help from the seminars creating the "MyNet2" model. Since "MyNet" worked, which was already a pre-made network from the different tutorials, we figured that we should just use the same model and not try to use a different model we did not understand fully.

Object detection task

For the objection detection task, we discovered a new way to define how our convolutional neural network were built up. For this task we create one class CNNBlock to create each layer. And another class that we called yolo that reads a list of tuples, strings and lists and calls CNNBlock accordingly. We found this to be a more readable and cleaner implementation of multiple architectures. More information about this implementation can be found in the task3.ipynb file.

We also implemented functions to save a trained model as a file. We did this as we found training a model for object detection was very computationally heavy for our computers.

The **Yolo_loss** function is like the one we used in task one, only with a few changes to work with the new tensor shapes.

The **train_fn** simply trains the model for one epoch and calculates the mean loss. Validation calculates the validation loss used for early stopping.

Then there is the **main** function which calls the train_fn and validation epoch number of times or until validation loss does not decrease for 3 epochs. If this happens the model with the lowest average validation loss is saved and used later in model selection. We call main with every model architecture we created and saves all models.

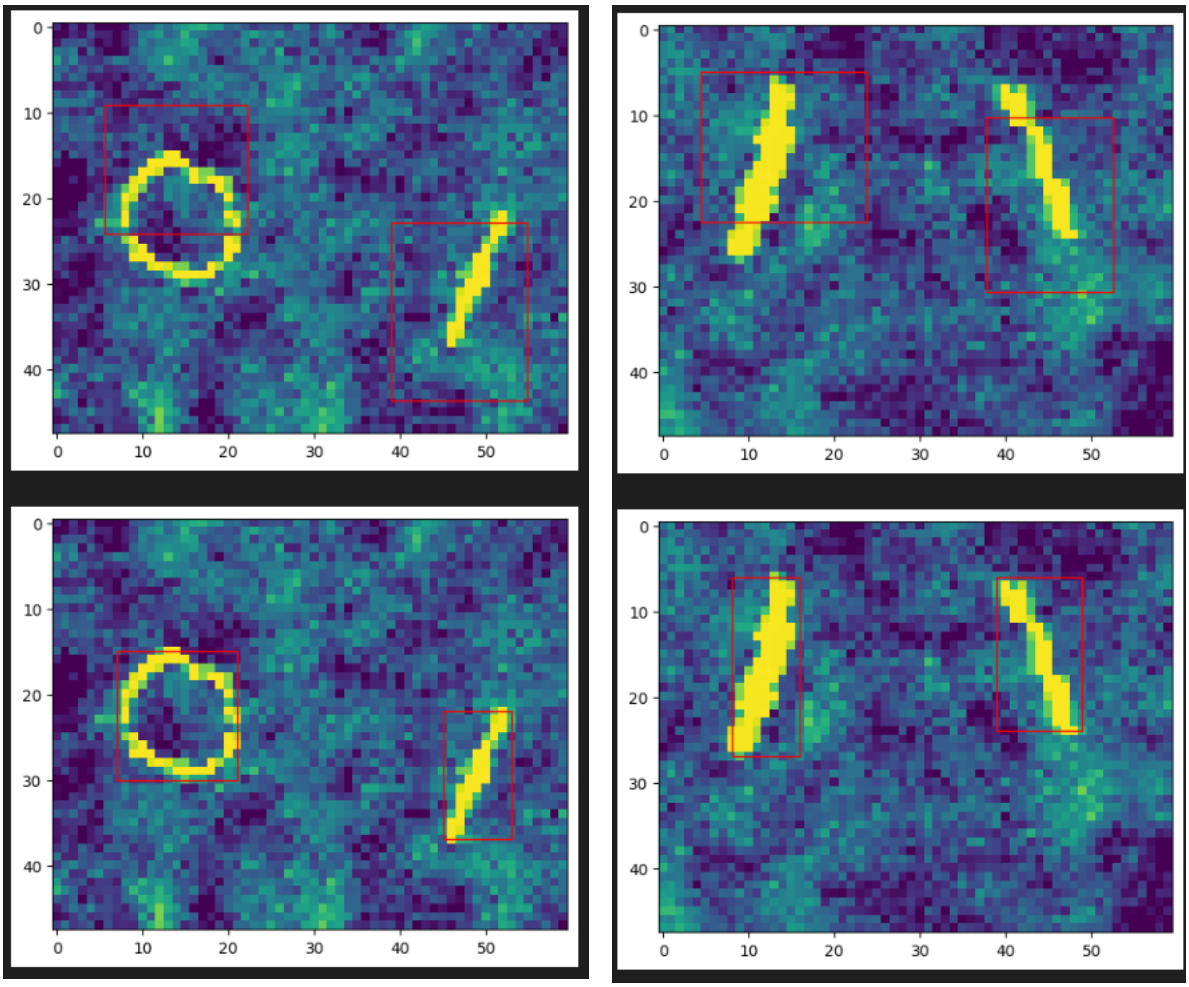
Cell_to_img goes through the bounding boxes and scales it to the entire image.

Lastly, we have our model_selection function. Here we load our models one by one and print 4 images with predicted bounding boxes for each of our models. We unfortunately did not have enough time to train each model architecture with several different hyperparameters due to the amount of time it took to train a single model. Unfortunately, we were not able to implement a working performance measure, but we can easily say that our first model was the best. If we had been able to implement a working performance measure then we would have chosen the best one automatically and tested it on unseen test data.

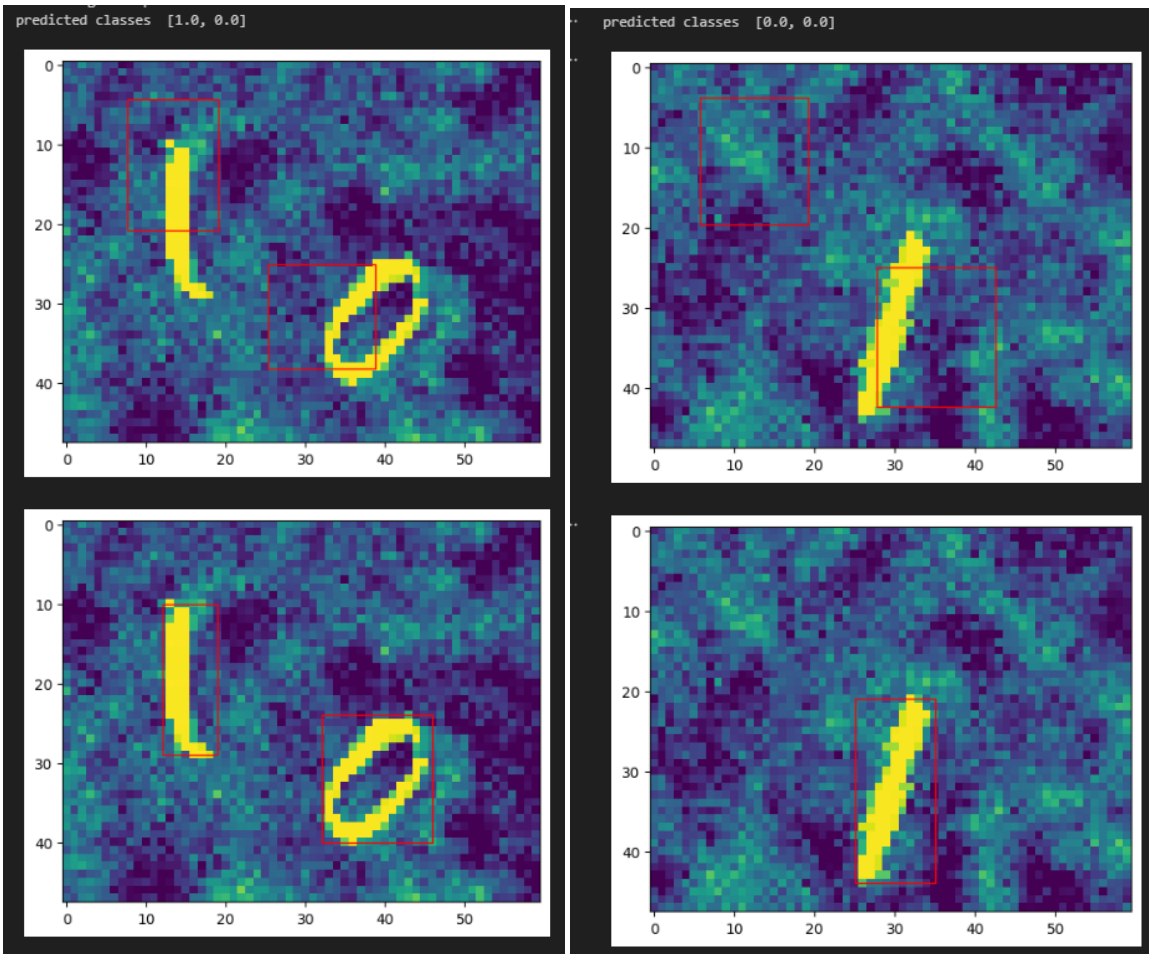
Results

Our first model is heavily inspired by yoloV1s architecture. Where we only changes input and output shapes. The other 3 architecture we used were a lot simpler than this as we realised that training multiple very deep convolutional neural networks would be to demanding for our hardware.

We trained the yoloV1 architecture model with 100 epochs, learning rate of 1e-2 and weight decay of 0. This was the model that performed the best on both training and validation data. The bounding boxes were ok for this model, but the class predictions were unfortunately underwhelming. We do however believe that this is due to too little training as we had to cut the training set down from 27000 to 5000 just to train one model with 100 epochs. This is the results with validation data. Top two images are predicted and the bottom two are label.



And this is for the training data.



Our other architectures unfortunately did not work as well, or really at all. We think the reason for this is that they were too simple. Resulting in the models not being able to learn. This is evident when we saw that the models training loss did not have a downward trend. These models were not able to print bounding boxes as they never reached the threshold of confidence that there was an object in any of the pictures. The results from these models can be found in the task3.ipynb file.

Final thoughts

Thoughts around the task

Throughout the tasks we had a hard time understanding how to implement the code in practice and extract the correct values to the correct functions, but ultimately understanding the task was not that hard.

Comments on results

Based on looking at the training and validation accuracy, our best model seems to be in a sweet spot between underfitting and overfitting. We cannot say for sure if it underfits or overfits purely on this, but according to the data it seems promising. We also checked it against the test set. We got approximately the same result. Which means the model seems to have learned the underlying patterns, and are robust to being exposed to new data.

The results were surprisingly good and satisfying. On the localization part it seems like we have gotten fine results. And when we plot the images with bounding boxes, the predictions seem to correspond to the accuracy.

It might seem like the accuracy in the IoU performance might be slightly below the real accuracy. The loss function handles the fact that there are no objects there. The IoU function, however, does not take into consideration that it should not do the IoU calculations when there are no objects there. Ultimately, this means that the IoU will return 0 when there is no object. Which will affect the overall accuracy score by adding 0 when it should not do any calculations at all.

If we were given unlimited time, we would try even more complex models with different values of hyperparameters and the entire dataset especially for the object detection task. As our results from trying one combination of hyperparameters with the yolov1 architecture were surprisingly good considering the amount of training data we used.

The code runs as expected. There is a lot of data, and our computational power is not strong enough for it not to run for multiple hours.

Conclusion

Our object localization task seems to work as expected. We get ok results, additionally we are able to produce images with localization boxes and predictive boxes. Our object detection showed some promise with the yolov1 architecture, and it was able to predict some reasonable bounding boxes.

Random facts

Libraries

“torch” and “torchvision”: “torch” is made for manipulating and creating tensors whilst “torchvision” provides utilities for preprocessing and loading image datasets

“torch.utils.data” : provides a package of loading and transforming datasets in pytorch

“Dataloader”: used to load and prepocess data in batches

“torch.optim” : provides various optimization algorithms for training neural networks

“pandas” and “numpy”: data analysis and manipulation libraries. Useful for data analysis and preprocessing steps before training our neural network

“matplotlib”: data visulation tool. Used to visualize the plots and images in the code

“collections”: provides various alternatives to built-in types such as tuples, lists, and dictionaries. We used “Counter” from this module

In our code we have used several different libraries that are quite relevant to the code. They are relevant in training and evaluating a deep neural network for image localization and detection. They provide the necessary tools to preprocess, structure, load, simplify and analyse the data. All the libraries were used.

Programming style

We programmed in an object-oriented way. For example in our “MyNet2” model the class inherits from the PyTorch “nn.Module” class and defines two main concepts - the extractor layer and the classifier layer. Overall, the use of classes and objects in this code are similar to OOP.

Code efficiency

Generally, the implementations are efficient and do not use a lot of unnecessary memory. It may take some time to run through all the models and images and analyse the data, but that is expected when the data is so huge. We only computed the necessary things to the task at hand, and all the different variables we created were used. In short, the code is quite efficient and calculates the right results.

The code is not unnecessary long as well. We included every part that was necessary to give the most optimal results.