INF265 Project 2

Anders Tøkje (antok8704)
Aurora Ingebrigtsen (ain015)

March 22, 2024

1 Introduction

In this project we first solved an object localization (part 2), and then an object detection task (part 3). The notebook delivered together with the report, named "project2.ipynb" contains what can be seen as the main code of our project. It utilizes functions, objects, constants and models from the files named project2_***.py. A README.md file is also delivered, explaining the setup more thoroughly. The reason for this is we want to keep the jupyter notebook as clean as possible. In order for it to appear cleaner, we decided to make the functions, models and objects in separate files from the notebook, and importing them at the start of the notebook.

Comments on the code and project structure

The codebase consists of lots of code, but if there is a specific function, model, class etc. from the notebook that you want to have a closer look at, it should be easy to navigate into the correct file and search up the code. The functions and classes is also well documented, so if you are using VSCode, PyCharm (or other IDEs with this functionality), hovering the function/ class will provide a basic overview of purpose, parameters, types, return types etc.

We also focused on making good naming conventions, so that it is easy to tell if a function or class is specific for one of the tasks. Ex. LocalizationLoss, plot_localization_data() and localization_performance() all are specific for task 2, meanwhile DetectionLoss, plot_detection_data() and detection_performance are specific for task 3.

Division of Labour

For each subpart of the project we approached, we started by discussing the details of how we would implement this part. Then we splitted the work, so each of us worked on some function/ implementation. We also used pair-programming on the more difficult tasks.

2 Object Localization

2.1 Loading and pre-processing the data

As we mentioned in the first project, handling the data correctly prior to training models is crucial for the model's performance. The steps included in handling the data comprehends the way we import-, analyze- and pre-process data. This is why we:

- Split into training-, validation- and test dataset as early as possible.
- Primarily do the data analysis on the training dataset (except checking for errors in validation- and test dataset, such as class imbalance). This is because we don't want any data leakage or biased perceptions from the data we validate or test with.
- Build the pre-processor based on the training set, with reduced features. Again to limit any data leakage from validation- and test data.
- Import the data again, using the pre-processor (normalizer).
- Train the model

We imported the data as tensordataset objects using the built-in torch.load() function. To get a quick overview of the train-, validation- and test data, we print their size to see how well they're distributed.

```
Train data size: 59400
Val data size: 6600
Test data size: 11000
```

The data we use comes pre-distributed, and is split into 77/8/15 datasets. Normally we would prefer a 70/15/15 split in the data, but because this was already done we didn't want to change the original distribution of data.

To get a quick impression of the data we extracted the first image and label from the training dataset, and inspected the size and type of this tensor.

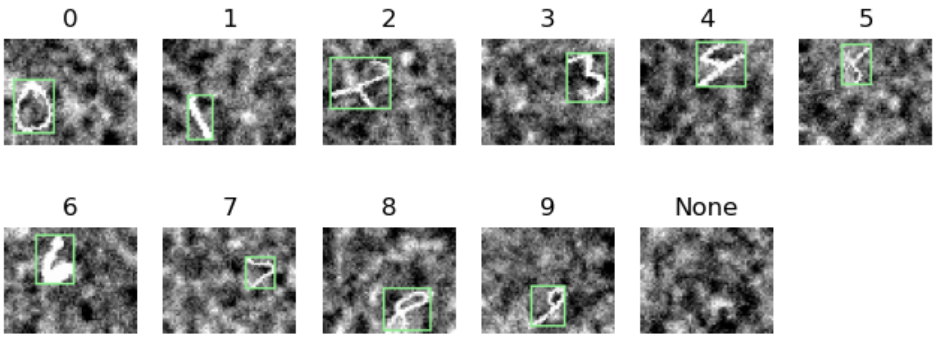
```
Shape of first image: torch.Size([1, 48, 60])
Type of first image: <class 'torch.Tensor'>

Shape of first label: torch.Size([6])
Type of first label: <class 'torch.Tensor'>)

tensor([1.0000, 0.6000, 0.2292, 0.3667, 0.4167, 4.0000], dtype=torch.float32)
```

We also wanted to see if all the label classes was weighted similarly, and we quickly realized that there are double the amount of images in class 1. We investigated and found out this was because the images containing no class label all had class 1, but object identified as 0. To get a real representation of the number of instances in each class, we set the images with 0 in first

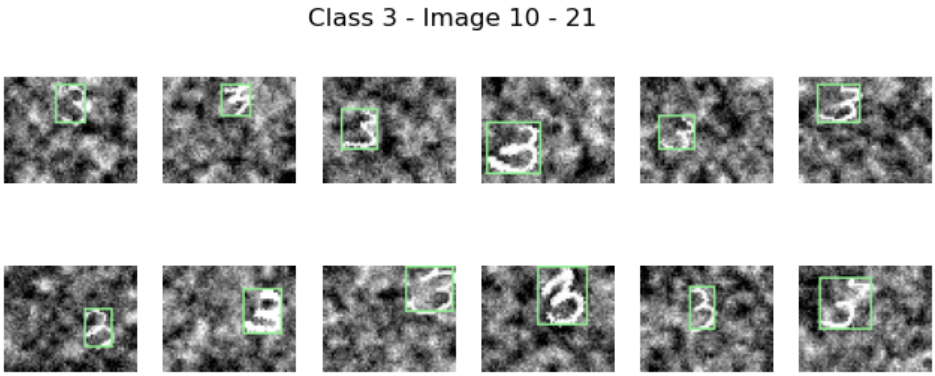
position to 99 in the count_instances() function.

Class distribution in Training	Class distribution in Validati	Class distribution in Test Data
0: 5345	0: 578	0: 980
1: 6075	1: 667	1: 1135
2: 5365	2: 593	2: 1032
3: 5522	3: 609	3: 1010
4: 5243	4: 599	4: 982
5: 4889	5: 532	5: 892
6: 5310	6: 608	6: 958
7: 5644	7: 621	7: 1028
8: 5249	8: 602	8: 974
9: 5358	9: 591	9: 1009
99: 5400	99: 600	99: 1000

Because this is a object localization task, we wanted to have a look at how true labels with bounding would look like. With a little effort we managed to draw one example of each label in the images with the bounding boxes encapsulating the object detected.



The function we created for printing examples of images is somewhat complex, but the complexity brings more functionality to the function and also prevents lots of code redundancy. Here we print examples of one specified label class, in the interval of the 10th - 21th of label class 3.



We had some experience with normalizing the data from the first project, so we made a function normalize() to handle normalizing all data from a source dataset. Briefly explained, we find the mean and the standard deviation from test data, and normalize all datasets with with these numbers. Normalization offers multiple benefits when training a model. Among these are faster convergence in neural networks during training and mitigating issues related to exploding and vanishing gradients.

Then the data is loaded into the torch dataloader.

2.2 Custom Localization Loss

We wrote the localization loss as a class named LocalizationLoss to match the loss function convention in PyTorch. We first initialize the three loss functions we are going to combine for our custom loss function: binary cross entropy loss for object detection, mean squared error for localization loss and cross entropy loss for image classification. The fourth loss function (L_c_binary) is used for classification loss if there are only 1 class ($[c_1]$). First the loss function calculates detection loss L_A for all predictions. Then, it simply makes a mask named object_detected, which sets true where y_true has object detected (1 at index 0), and false where it is not detected (0 at index 0). We then compute L_b and L_c for the masked predictions. The overall loss is the sum of the three losses $L_A + L_B + L_C$.

2.3 Performance Metric

As described in the project description, our overall performance metric is the mean of IOU and accuracy. It is calculated by the function named localization_performance(). The function loops trough each batch in the loader, which it takes as a parameter. For each batch we:

- make a mask for the predictions that has found an object. This is done by taking $sigmoid(p_c) > 0.5$ for each label.
- we then find the predicted class for each label by taking the class pred with highest value. Since each class correspond to their index when slicing out the class predictions (class 0 is at index 0, class 1 is at index 1 etc.), we simply return the index with the max value.
- the number of labels in the batch is added to total, and the number of correctly classified datapoints are added to correct. These are the predictions where
    1. it is not detected any object, and the image does not contains any object
    2. it is detected an object, the image does contain an object and the predicted class is the same as the true class.
- it adds the number of true labels that contains objects to total_iou. Next the IOU is found and added to iou_sum by using the object detected mask on outputs and the labels, and passing these to the function named calculate_iou(). This function is described below. [1]

We then find the overall Accuracy and IOU, and return those together with their mean.

IOU Calculation

To prevent code redundancy, we made a function named calculate_iou(), that is used both in the calculation of localization performance and detection performance. The function slices out the true bounding boxes as well as the predicted bounding boxes. It then uses PyTorch's box_convert() function to change the coordinate format from $[cx, cy, w, h]$ to $[x, y, x, y]$. The IOU is then calculated using PyTorch's box_iou() function. This function returns a tensor containg all possible combinations of bounding boxes from the true labels and the predictions. We only want the index pairwise IOU's (IOU of true label 1 and prediction 1, true label 2 and prediction 2 etc., not IOU of true label 1 and prediction 2), which simply could be solved by taking the diagonal of the returned tensor.

2.4 Defining Models

We spent quite a lot of time designing our models, mostly because we found it interesting and fun. The models used in this task, uses a function called

[1] We chose to calculate IOU of the predictions that has detected objects. This means that the IOU is calculated for true positives as well. We would argue that this is resaonable, since the iou will be 0 for true posistives, meanwhile the total denominator is increased by 1. This provides a stronger "connection" between the localization and detection task.

fc_size() to automatically calculate the input size to the first fully connected layer. A quick description of each model is described below:

- Model 1 (LocCNN1): The first model we made is a simple CNN model with 3 conv layers, 2 max pool layers and a fully connected layer in the end.
- Model 2 (LocCNN2): Since the model were supposed to solve a relatively complex task, we wanted to make the model more complex. Therefore model 2 contains double the amount of conv, pooling and fully connected layers.
- Model 3 (LocCNN3): We also wanted to try reducing the variance, since more layers can increase variance. Since dropout does not seem to increase bias, we wanted to try adding this to the network.
- Model 4 (LocCNN4): This model is quite experimental, and we made it because we wanted to try the hourglass architecture using deconv layers and max unpooling.
- Model 5 (LocCNN5): The last model has larger kernel size than the other models, and 3 fully connected layers at the end.

2.5 Model selection

A way to tell how well a model performs, is to train several models and compare their performance to each other. This requires some planning and setting up a test scheme. The first project gave us a notion of how important several features of the machine learning algorithm is, e.g learning rate, momentum and decay. This project we focus more on the model's architecture, which is also very defining for how well the model performs. But in order for our model to yield the best results, we decided to train each model architecture with several different hyper parameters. Some of our fellow students also proclaimed that Adam optimizer would provide both faster and better results than SGD. This was something we wanted to try out for our selves.

2.5.1 Making an evaluation plan

Our rough plan was to first train our models with different hyper parameters using SGD, then do the same using Adam as optimizer. Prior to this training and evaluation test, we set the learning rate, momentum and decay when model designing. The main reason for this was that when we designed the first model architecture, our model wasn't performing great. We had to figure out if there was something wrong with the architecture, loss function or performance measure. We had some prior knowledge to "standard" settings

of hyper parameters, and we implemented our first model with these hyper parameters:

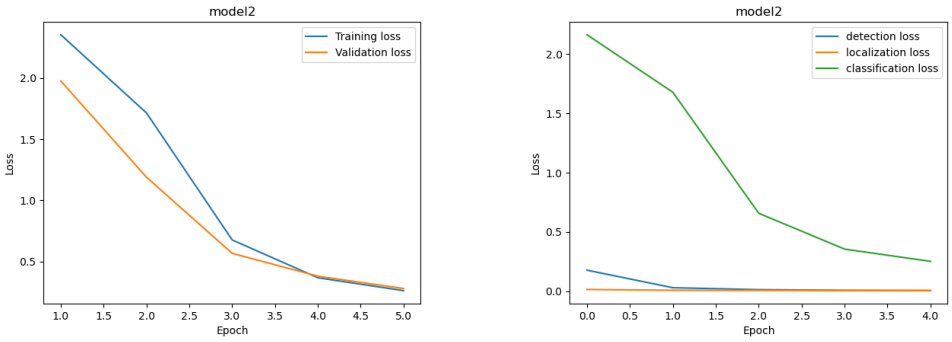
- learning rate : 0.01
- momentum : 0.9
- decay : 0.001
- batch size : 64
- epochs : 5

The reason for using 5 epochs was that we only wanted a quick overview of the model, and training it several times tends to be a tedious task. Also, in some cases an epoch may take 10 minutes from start to finish, depending on model architecture. 5 epochs would result in 50 minutes training time .As for batch size, we set this to 64 to pass more images through the model in each training for the sake of better training time.

All results are displayed in a table at the end of this section.

2.5.2 Run 1 using SGD

This test was set with the hyper parameters explained above, which had more practical reasons for being implemented. However, our models actually performed great, where our winner was Model 2 providing Accuracy 92%, IOU 51% and Mean 71%. This after only 5 epochs and about 2 minutes training.



The models loss, displayed in the images above, also suggested that we could benefit from training the model longer. This because the model was not converging after the 5 epochs.

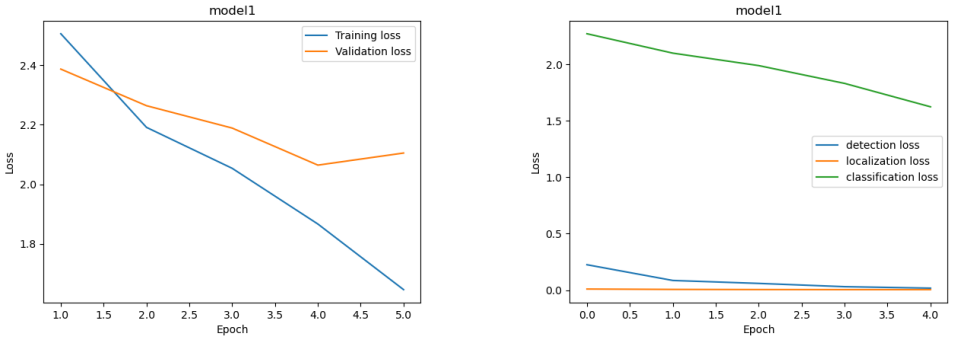
Model 4 also performed great, Accuracy 88%, IOU 45% and Mean 66%, but the complexity of it's architecture brought a huge challenge with it. The

model was very slow to train and it was also annoyingly slow at predicting. We phased this model out after this run, because we didn't want to spend too much time waiting for this model. A little bit sad for this, because it was a challenging and fun model to design.

2.5.3 Run 2 using SGD

Model 4 was disqualified for being too time hungry. Now the next thing we wanted to test was how the models would perform without any momentum and decay. Here we expected that models wouldn't perform too good after 5 epochs.

As expected, the models didn't perform too good, with model 1 being the one with best performance, Accuracy 39%, IOU 40% and Mean 39%. But it only spent 1 minute training before it was stopped.

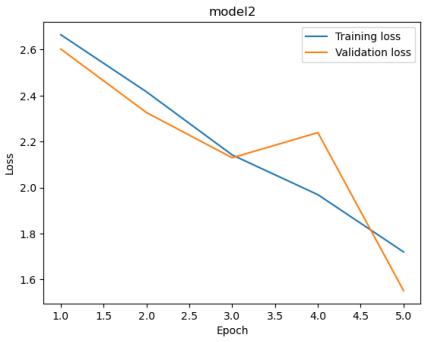


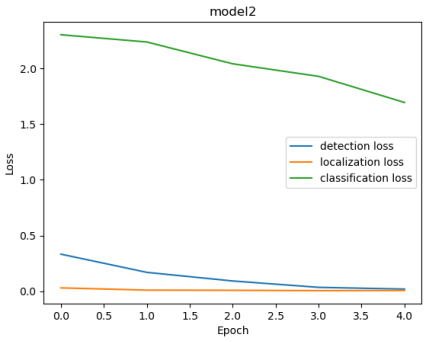
The losses displayed in the image above suggests that the model was not finished training, so it would probably benefit from higher epochs.

2.5.4 Run 3 using SGD

We wanted to change hyper parameters back to as in run 1, but see what happened if we enlarged the batch size. Here we expected a lower training time and lower performance.

Comparing the results with Model 2, which won run 1 with the same hyper parameters, the training time halved and the performance dropped significantly. Accuracy 50%, IOU 35% and Mean 42%. Still the best model was model 1, and it's results are displayed in a table below.



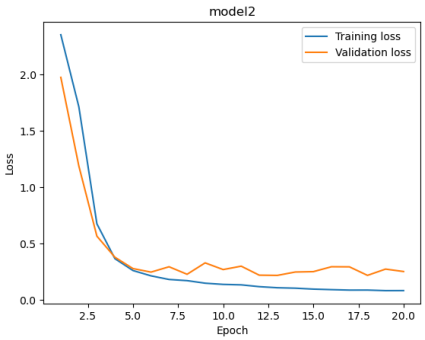


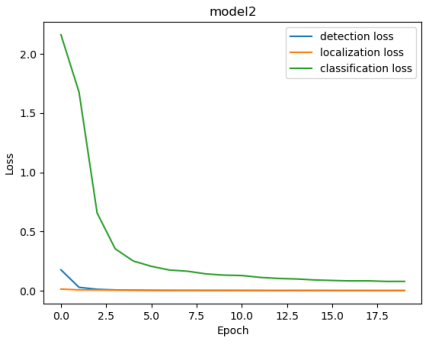
The losses suggests that the model would benefit from more training than 5 epochs.

2.5.5 Run 4 using SGD

Now we just wanted to see what happened if we run the model for 20 epochs. We also set the batch size to 64, because we wanted to yield the best possible results, while model training still was time efficient. We expected a better performance in all models.

Model 2 won with Accuracy 94%, IOU 56% and Mean 75%.





As the image of the losses above suggests, the model converges after 7 epochs. There's little to no learning after epoch 7. Our model does not seem to overfit, because validation loss remains relative stable after reaching it's maximum.

Table 1: SGD as optimizer

Run number	Models	Batch Size	Epochs	Learning Rate	Momentum	Decay
1	All Models	64	5	0.01	0.9	0.001
2	1,2,3,5	64	5	0.01	0	0
3	1,2,3,5	256	5	0.01	0.9	0.001
4	1,2,3,5	64	20	0.01	0.9	0.001

Table 2: SGD results

Run number	Best Model	Accuracy	IOU	Mean
1	2	92%	51%	71%
2	1	38%	36%	37%
3	1	76%	44%	60%
4	2	94%	56%	75%

2.5.6 Using Adam

We had three runs using Adam as optimizer in our models, but our models didn't perform better nor did it train more time efficient. The reason for this might be that there is a bug in PyTorch, where the models cannot use dtype double when training with cuda cores. In order for testing we had to change dtype to float32, which might have had a significant impact on our training time. This makes it unfair to compare SGD and Adam for the time being, as they are trained with different data types. The hyper parameters and results are displayed below.

Table 3: Adam as optimizer

Run number	Models	Batch Size	Epochs	Learning Rate
1	1,2,3,5	64	5	0.01
2	1,2,3,5	256	5	0.01
3	1,2,3,5	64	20	0.01

Table 4: Adam results

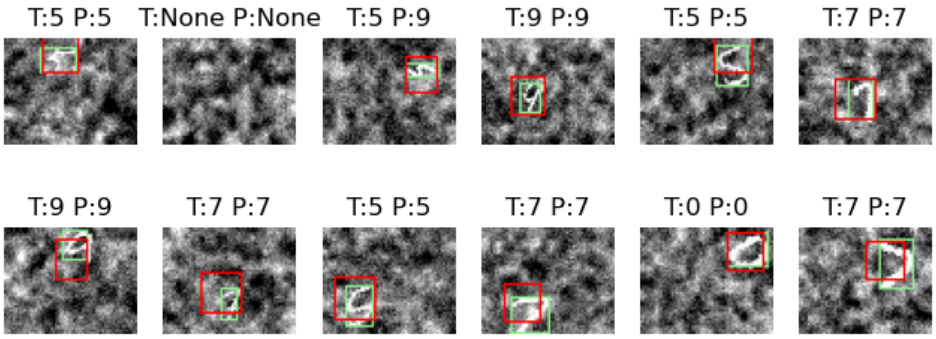
Run number	Best Model	Accuracy	IOU	Mean
1	2	72%	23%	74%
2	3	38%	36%	28%
3	2	80%	15%	47%

2.6 Model Evaluation

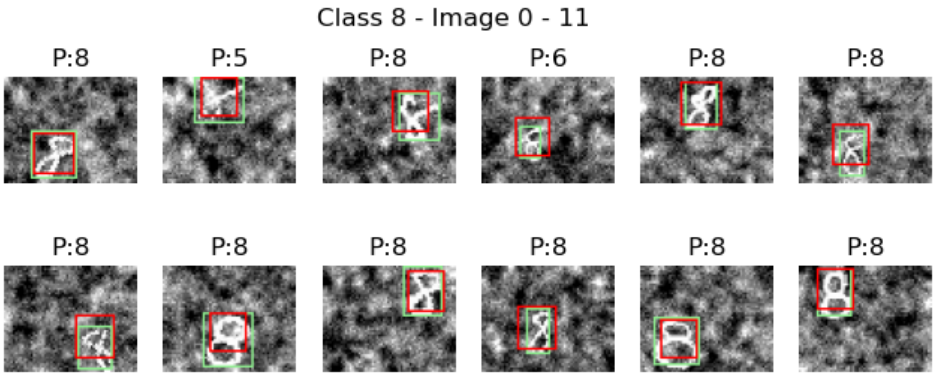
Model 2 yielded the greatest results on validation data, and therefore we need an unbiased evaluation using test data.

On test data it has Accuracy 94.2%, IOU 55.7% and Mean 74.9%. This is pretty close to the metrics on validation data, which also gives us a strong indication that our thoughts of it not overfitting was correct.

The images below are how well the model draws bounding boxes and predicts the label classes for each separate label class. The one below that makes the same predictions for objects of the same label class (8).



It seems to predict pretty well, and it draws the bounding boxes very close to whats in the true label. The third to the right in the top row, our model makes a mistake classifying a 5 as 9. This however was a tough one, which we easily could misclassify ourselves.



Here our model misclassifies two labels.

Over all we are very satisfied with the results of our model.

3 Object Detection

3.1 Loading and pre-processing the data

The data preparation in this section is similar to the one above. However, this part came with some challenges because the models should be able to predict multiple objects in one image. This expands the complexity of labeled data, where it should be represented as a 4D tensor when training.

The data came as two separate datasets. One containing a list of lists, with tensors representing the labels. The second dataset contained the image data and labels in the correct format (we don't want to use this).

First off we started importing the data and looking at its sizes.

```
Train label size: 26874
Val label size: 2967
Test label size: 4981
```

True labels in the image datasets gives us a clue of how the labels should be represented. This shows that the images can be divided into a grid of 32, where each grid cell is "checked" for objects.

```
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]]],
       dtype=torch.float32)
```

However the true labels are represented like this, and we need to get it to the tensor format above.

```
tensor([1.0000, 0.2583, 0.4062, 0.1167, 0.3958, 1.0000], dtype=torch.float32)
tensor([1.0000, 0.6500, 0.6667, 0.2333, 0.3333, 0.0000], dtype=torch.float32)
```

We needed to transform our listed labels to a 4D tensor as displayed above, where the inner dimensions of the tensor should represent one grid cell. However the task suggest a different order of dimensions than this, so the function reorders the dimensions according to the task. How to quality assure this is mentioned below.

In order for this to happen we had to make a function prepare_labels() which returns a new tensor with the label data separated into the specified amount of grid cells.

global_to_local() function is called by prepare_labels(), and does the transformation on the data. prepare_labels() stacks these transformations, reorders the dimensions and returns the specified tensor. This outputs:

```
tensor([[[1.0000, 0.0000, 0.0000],
         [0.0000, 1.0000, 0.0000]],

        [[0.7750, 0.0000, 0.0000],
         [0.0000, 0.9500, 0.0000]],

        [[0.8125, 0.0000, 0.0000],
         [0.0000, 0.3333, 0.0000]],

        [[0.3500, 0.0000, 0.0000],
         [0.0000, 0.7000, 0.0000]],

        [[0.7917, 0.0000, 0.0000],
         [0.0000, 0.6667, 0.0000]],

        [[1.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000]]])
```

Comment out line 414 from function prepare_labels() in functions.py new_tensor = new_tensor.permute(0, 3, 1, 2) in order to check if this has the same format as the original tensor label. train_labels_local = prepare_labels(train_labels, (2,3,6)) val_labels_local = prepare_labels(val_labels, (2,3,6)) test_labels_local = prepare_labels(test_labels, (2,3,6)) will now produce the same.

Similar to global_to_local(), we made a function called local_to_global(), which takes the tensor back to its original list of lists format.

The function merge_datasets() is a function to combine our new tensor containing labels with the image data.

3.2 Custom Detection Loss

As with the localization loss, the detection loss is implemented as a class named DetectionLoss to match PyTorch's convention. The loss function reshapes the tensors to match the expectation of the localization loss. This means that the tensor is permuted from shape (B, C, H, W) to (B, H, W, C). We then reshape the tensor to (B, C) meaning that each grid cell in each prediction becomes a seperate label (Which increases batch size, so that Batch size passed to LocalizationLoss = $B*C$). We then take the localization loss for each label.

3.3 Performance Metric

For object detection, a suitable performance metric is mAP (mean average precision).

$$\text{MAP} = \frac{1}{N} \sum_{i=1}^{N} AP(i)$$

Where N is the number of classes.

This is implemented in the function named detection_performance(). It uses the calculate_ap() function to calculate the AP for each class. We used the README.md from Padilla et al. (2021), to understand the details of AP (average precision) calculation. AP is defined as:

$$\text{AP} = \frac{1}{11} \sum_{r \in \{0, 0.1, \ldots, 1\}} \rho_{\text{interp}}(r)$$

In short, the function:

- finds the confidence of an object being present in each class ($sigmoid(p_c)$)
- makes a new tensor that contains the IOU calculation for each grid cell
- defines the conditions for a true positive (TP) and a false positive (FP). Such that a TP means a detection with IOU threshold and a FP means a detection with IOU $<$ threshold.
- finds the indices of the confidence scores sorted descending

- iterates trough the sorted confidence levels, and for each confidence level sets an accumulated TP and FP score, and calculates recall and precision.
- then we use 11-point interpolation to summarize the shape of the precision x recall curve. We interpolate 11 equally spaced points, called recall_levels from 0-1. We then find the interpolated values for each recall level by taking the maximum precision whose recall value $>$ than its current recall value with help of a mask.
- the function then return the sum of all 11 interpolated values and divide by 11.

3.4 Defining models

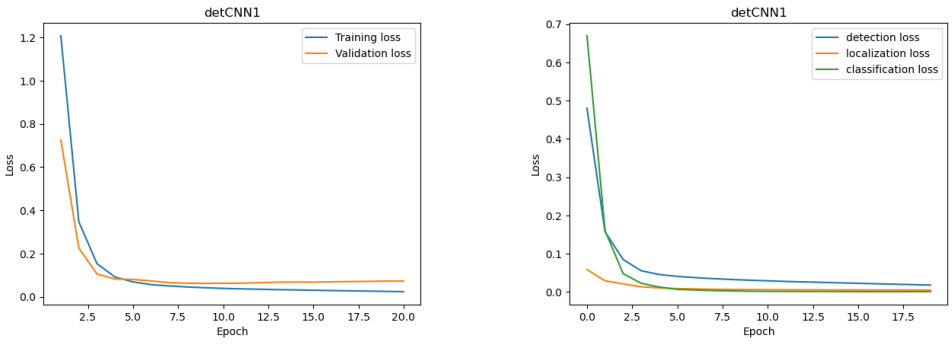
For this part, we decided only two make architectures, since it was both time consuming and more restrictive in this part (since we only used conv layers). The two models are described below:

- Model 1 (DetCNN1): The first model is pretty standard, and most conv layers applies what is called same convolutions. The image is mostly downsized by the max pooling layers,
- Model 2 (DetCNN2): The second model is a little more shallow, and uses larger kernel sizes in the beginning of the network.

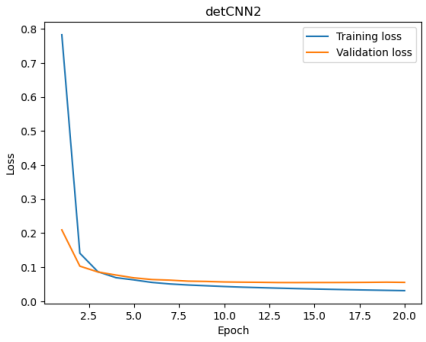
3.5 Model selection

With knowledge from the prior tests we made, we decided to train two simple model architectures with the hyper parameters from the last tests which yielded the best results. Test results are displayed below.

Loss for model 1 suggests that it converges and that it doesn't overfit.



Loss for model 2 suggests that it converges.



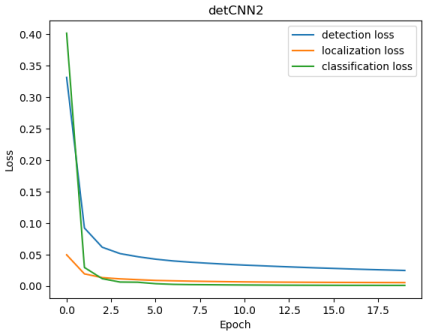


Table 5: SGD as optimizer

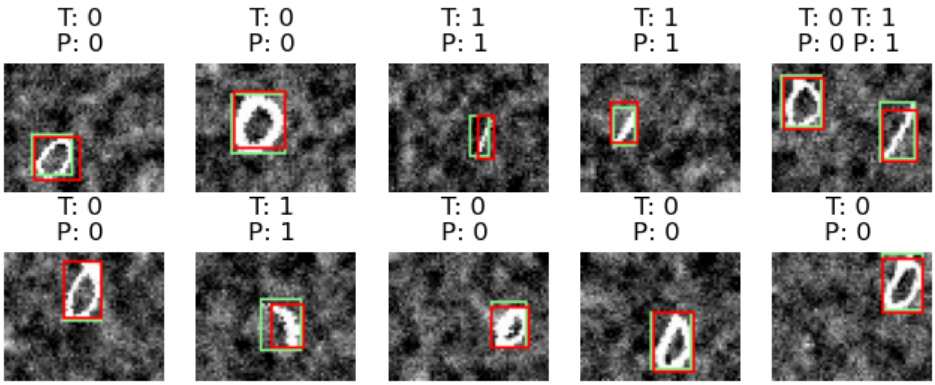
Run number	Models	Batch Size	Epochs	Learning Rate	Momentum	Decay
1	1,2	64	5	0	0	0
2	1,2	64	5	0.01	0.9	0.001
3	1,2	64	20	0.01	0.9	0.001

Table 6: SGD results

Run number	Best Model	MAP
1	1	82%
2	2	88%
3	1	91%

3.6 Model Evaluation

We then evaluated the best model. It performs with 90.7 % mAP. And the predictions from start idx = 10, can be seen below.



4 Comments on the Results

Overall, we are happy with our final results. We got high performance scores on both the tasks, which could be increased even more with more training and hyperparameter tuning.

Regarding object detection, our model seemed to perform suprisingly well. We expected this task to be more challenging since there would be than one object in since objects may have different rotation and size. Yet, our best model from task to got a mAP of 90.7 %, which is pretty good. We also saw that it performed really well on the bounding boxes. Potential reasons for this can be that the model was solving a binary class task instead of a multiclass task as in the first part. The cases where it seemed to struggle, was when the bounding box was placed in the middle of two grid cells. This sometimes led to double predictions, one from each grid cell.

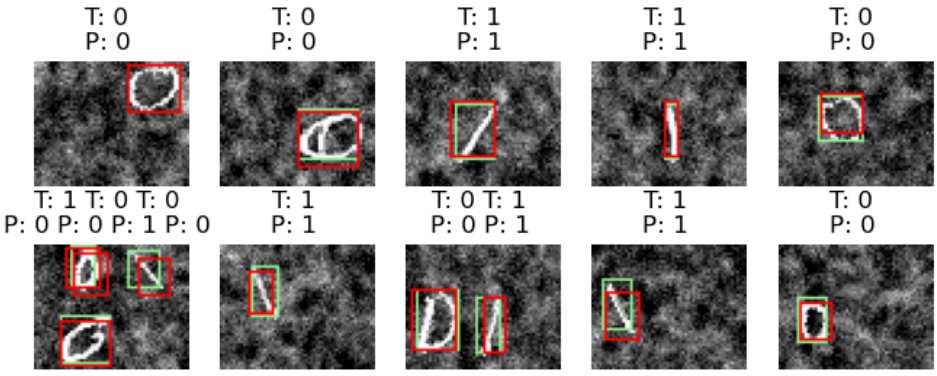


Figure 8: Double predictions in first image at row 2

References

R. Padilla, W. L. Passos, T. L. B. Dias, S. L. Netto, and E. A. B. da Silva. A comparative analysis of object detection metrics with a companion open-source toolkit. *Electronics*, 10(3), 2021. ISSN 2079-9292. doi: 10.3390/electronics10030279. URL https://www.mdpi.com/2079-9292/10/3/279.