CV-Project-3 Instance Segmentation

Jan Gruszczyński [145464]
Kacper Trębacz [145453]

1. **Description of the data set**

Due to limitation of our hardware (lack of ram and computational GPU power), we decided to use ThrashCan 1.0[1]. It consists of images of underwater thrash. Each image has its own sets of masks describing the location of certain items, as well as bounding boxes and their labels.

Size of the dataset: 7212 images and annotations.
Collection period: 2019-06-24 to 2020-06-30
Dataset format: COCO

[1] https://conservancy.umn.edu/handle/11299/214865

2. **Description of the problem**

In the final project, we tried to tackle the instance segmentation problem using neural networks.

Definition of the problem: instance segmentation is the task of detecting and delineating each distinct object of interest appearing in an image.[2] So in layman's terms, we wanted to detect exactly where and what objects are on the image.

3. **Description of used architectures**
After doing extensive research, we thought we would have the best shot if we implemented the two currently most popular and promising architectures: YOLOACT and Mask-RCNN.

YOLACT

Repo[3]
Paper[4]
We use 2 metrics one of them is Mean average precision (mAP) for boxes and the other is (mAP) for masks.
As a loss function we use a sum of 4 losses: mask,box,classification and segmentation loss.
classification - Binary Cross Entropy
box - smooth L1 loss
masks - Binary Cross Entropy with logits per pixel for a given object
segmentation - Binary Cross Entropy with logits per pixel for whole image
Yolact architecture looks as follows:

[2] https://paperswithcode.com/task/instance-segmentation
[3] https://github.com/dbolya/yolact
[4] https://arxiv.org/abs/1904.02689

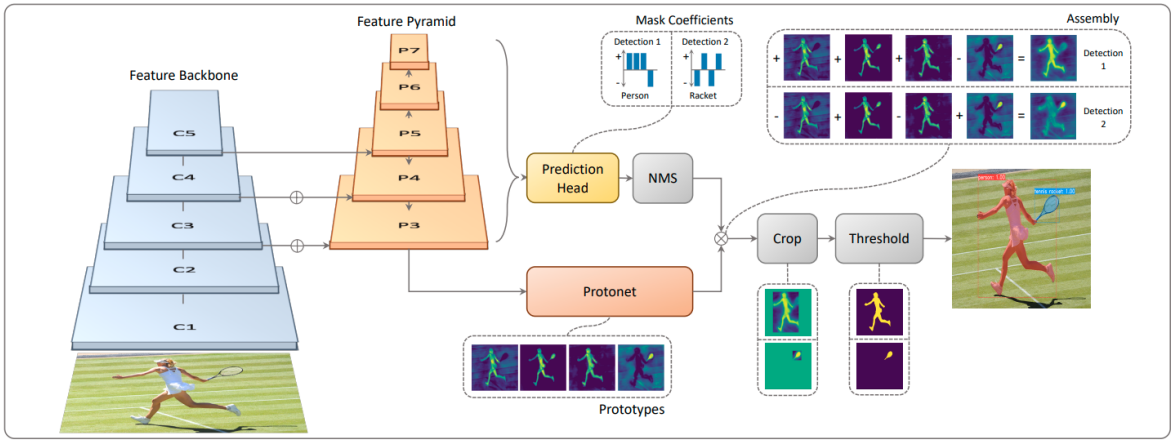
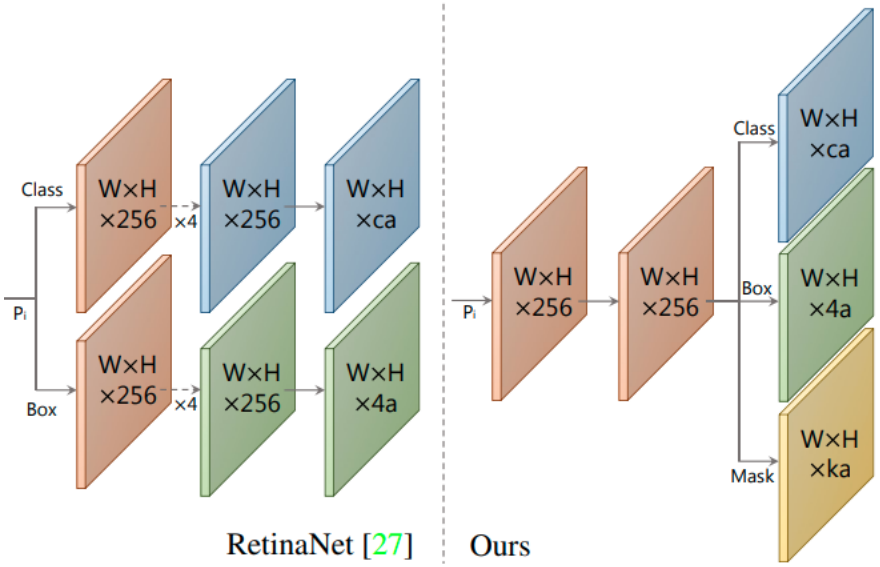


Figure 2: **YOLACT Architecture** Blue/yellow indicates low/high values in the prototypes, gray nodes indicate functions that are not trained, and $k = 4$ in this example. We base this architecture off of RetinaNet [27] using ResNet-101 + FPN.

Its main element is a feature backbone. We can use any convolutional neural network. We used resnet-50 and darknet-53, which is the YOLO-3 backbone. From backbone we select top 3-4 layers in order to build a feature pyramid. Feature pyramid is then fed into a prediction head:



Prediction head works similarly like in yolo, so the screen is divided into WxH regions and for each we predict classification scores, box coordinates and k mask coefficients. Why coefficients? Because YOLACT predicts k different masks for a whole image and those coefficients are then used to compose a final mask for the object. After Prediction Head there is a modified version of NMS, which is optimized in a way that it does not take into account already selected boxes. After NMS masks generated by Protonet are combined for each object in order to produce a final mask, which is then cropped to the bounding box and thresholded.

In our training we used default hyperparameters provided by the authors. We trained our model using 2 backbones

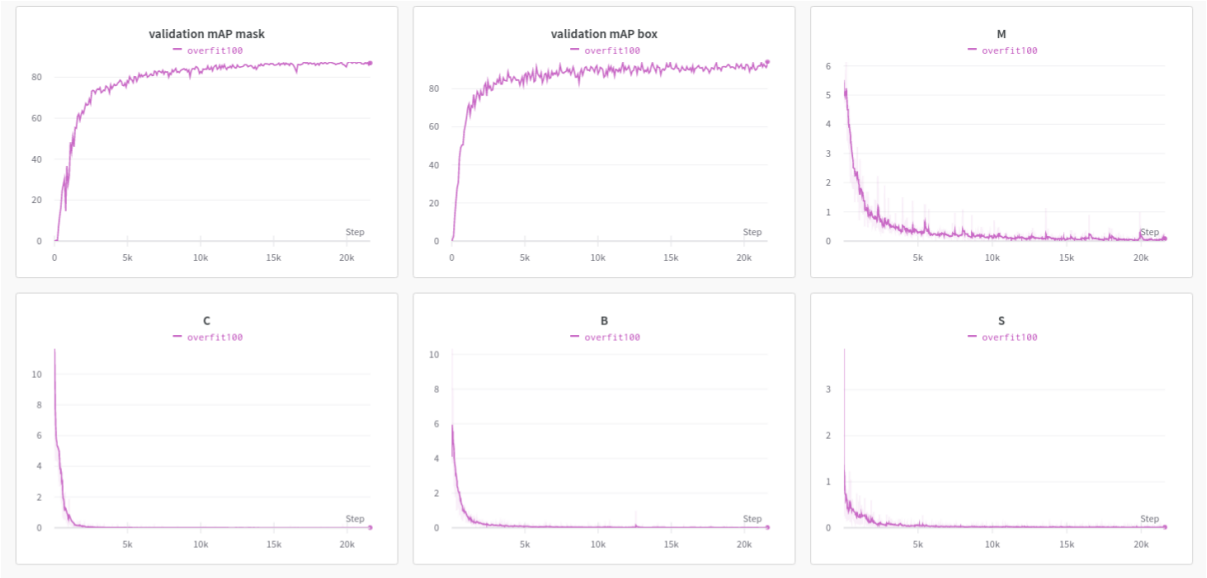
When it comes to training, we trained as long as our computer could stay in one place, so from 10-12 hours, we selected the model with the highest (mAP mask + mAP box)/2 on a valid dataset. We trained on GTX 1650 with 4GB RAM. List of used libraries is in requirements.txt. We logged everything on wandb.

We trained for 160 000/batch_size iterations.

Resnet-50

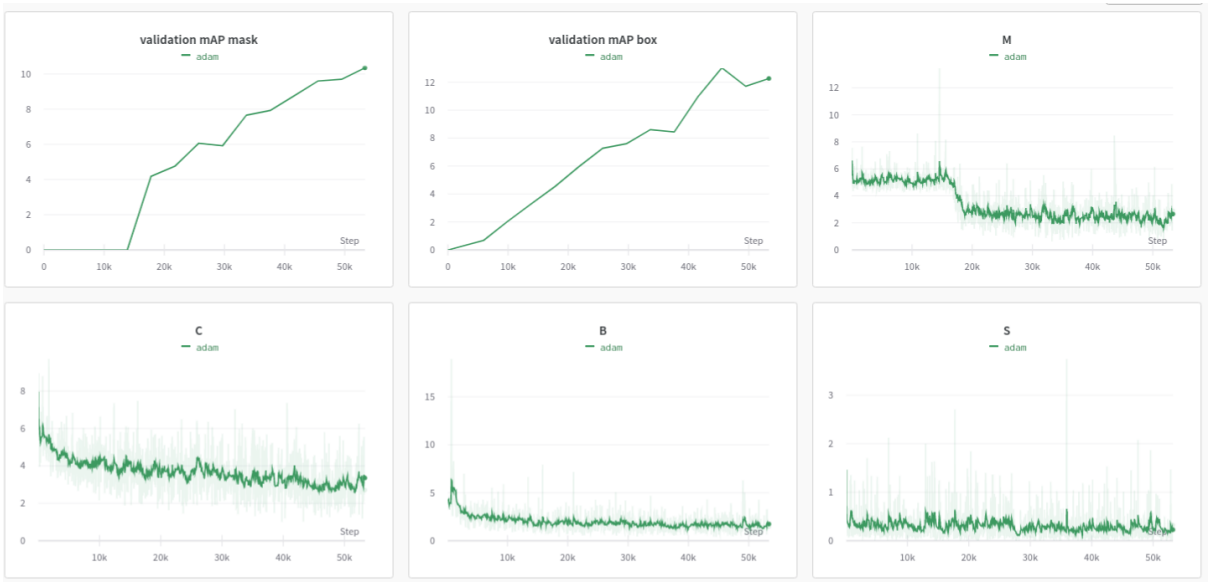
Batch size: 3
number of params: 30748967
Evaluation time: during prediction our model reaches up to 9 fps to inference time is 1/9 s

First, we overfitted on 100 photos:



Than we trained with default optimizer

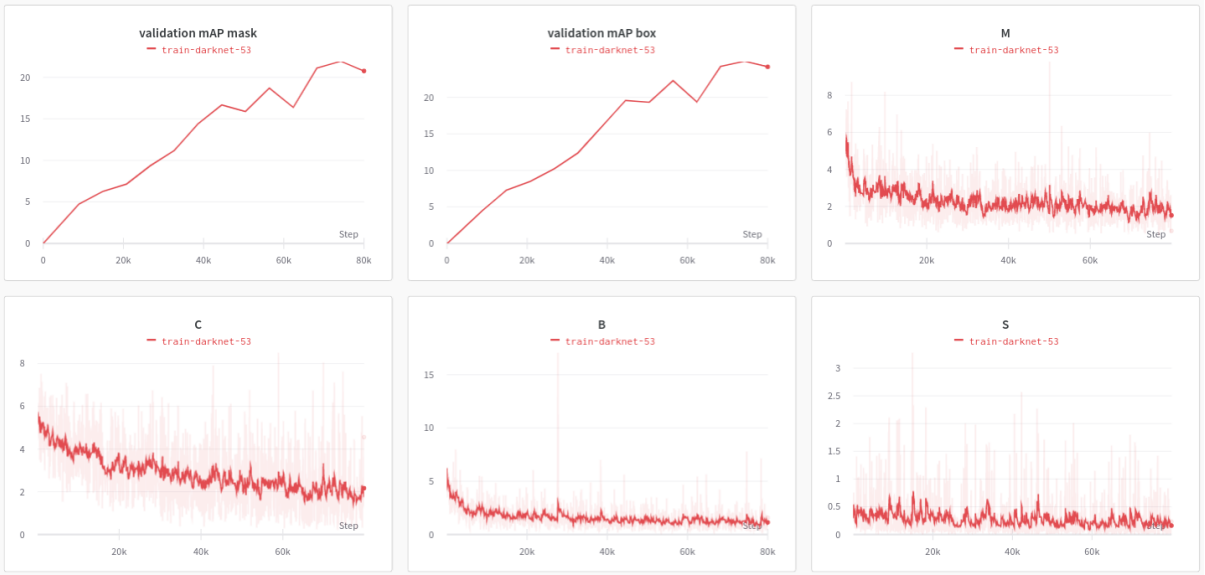
On validation set we reached: mAP for masks: 23.8 and mAP for boxes: 29.13
On test set we reached: mAP for masks: 22.75 and mAP for boxes: 27.31
We also trained using Adam with default hyperparameters:



On validation set we reached: mAP for masks: 13.3 and mAP for boxes: 10.35
On test set we reached: mAP for masks: 9.10 and mAP for boxes: 11.34

Darknet-53

Batch size: 1
number of parameters: 47367111
Evaluation time: during prediction our model reaches up to 9 fps to inference time is 1/9 s
Training:

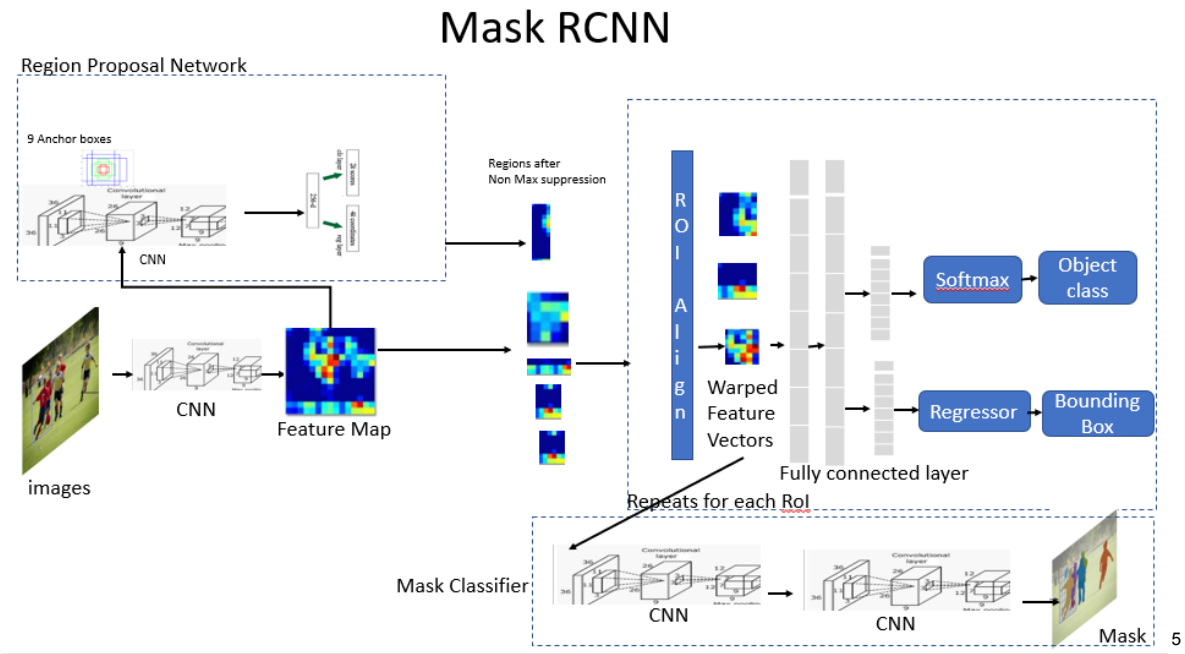


On validation set we reached: mAP for masks: 21.9 and mAP for boxes: 24.9
On test set we reached: mAP for masks: 22.85 and mAP for boxes: 25.85

It turned out that the default SGD Optimizer was better than Adam. Also mostly due to time constraints, the resnet-50 model slightly outperformed the darknet-53 model

You can run our yolact model through API on Docker. To run api type:
docker build . -t yoloapi:1
If you want to send requet to an API you can run:
python src/api/test_script.py

Mask-RCNN:
General graph of the architecture:



[5]

We used two different backbones mobilenet_v2 and maskrcnn_resnet50_fpn.

Number of parameters for resnet50[6]:

```
In [13]: model = get_model_instance_segmentation(23)
         # model.eval()
         pytorch_total_params = sum(p.numel() for p in model.parameters())
         pytorch_trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
         print(pytorch_total_params)
         print(pytorch_trainable_params)

          44035417
          43813017
```

Memory requirement for all parameters and buffers

```
In [14]: mem_params = sum([param.nelement()*param.element_size() for param in model.parameters()])
         mem_bufs = sum([buf.nelement()*buf.element_size() for buf in model.buffers()])
         mem = mem_params + mem_bufs
         print(mem) # in bytes

          176566628
```

[5]https://towardsdatascience.com/computer-vision-instance-segmentation-with-mask-r-cnn-7983502fcad1
[6] https://discuss.pytorch.org/t/gpu-memory-that-model-uses/56822

Number of parameters for mobilenet_v2

```
In [11]: model = get_model_instance_segmentation(23)
         # model.eval()
         pytorch_total_params = sum(p.numel() for p in model.parameters())
         pytorch_trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
         print(pytorch_total_params)
         print(pytorch_trainable_params)

         82460606
         82460606
```

Memory requirement for all parameters and buffers:

```
In [14]: mem_params = sum([param.nelement()*param.element_size() for param in model.parameters()])
         mem_bufs = sum([buf.nelement()*buf.element_size() for buf in model.buffers()])
         mem = mem_params + mem_bufs
         print(mem) # in bytes

         176566628
```

Description of the training:
We have used the following optimizer and learning rate_scheduler for both networks.

```
params = [p for p in model.parameters() if p.requires_grad]
optimizer = torch.optim.SGD(params, lr=0.005,
                            momentum=0.9, weight_decay=0.0005)
lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
                                               step_size=3,
                                               gamma=0.1)
```

Due to ram limitation we were able to train the networks with batch = 1. On average one epoch took 17 minutes for resnet, and 11 minutes for mobile net. Both networks were trained for 10 epochs.

Description of used metrics, loss, and evaluation:

Metrics:

```
outputs {'loss_classifier': tensor(1.4128e+09, grad_fn=<NllLossBackward0>), 'loss_box_reg': tensor(2.3486e+08, grad_fn=<DivBa
ckward0>), 'loss_mask': tensor(1.1091e+08, grad_fn=<BinaryCrossEntropyWithLogitsBackward0>), 'loss_objectness': tensor(365163
40., grad_fn=<BinaryCrossEntropyWithLogitsBackward0>), 'loss_rpn_box_reg': tensor(21352778., grad_fn=<DivBackward0>)}
```

Loss functions:
The loss being used is per-pixel sigmoid + binary loss.[7]

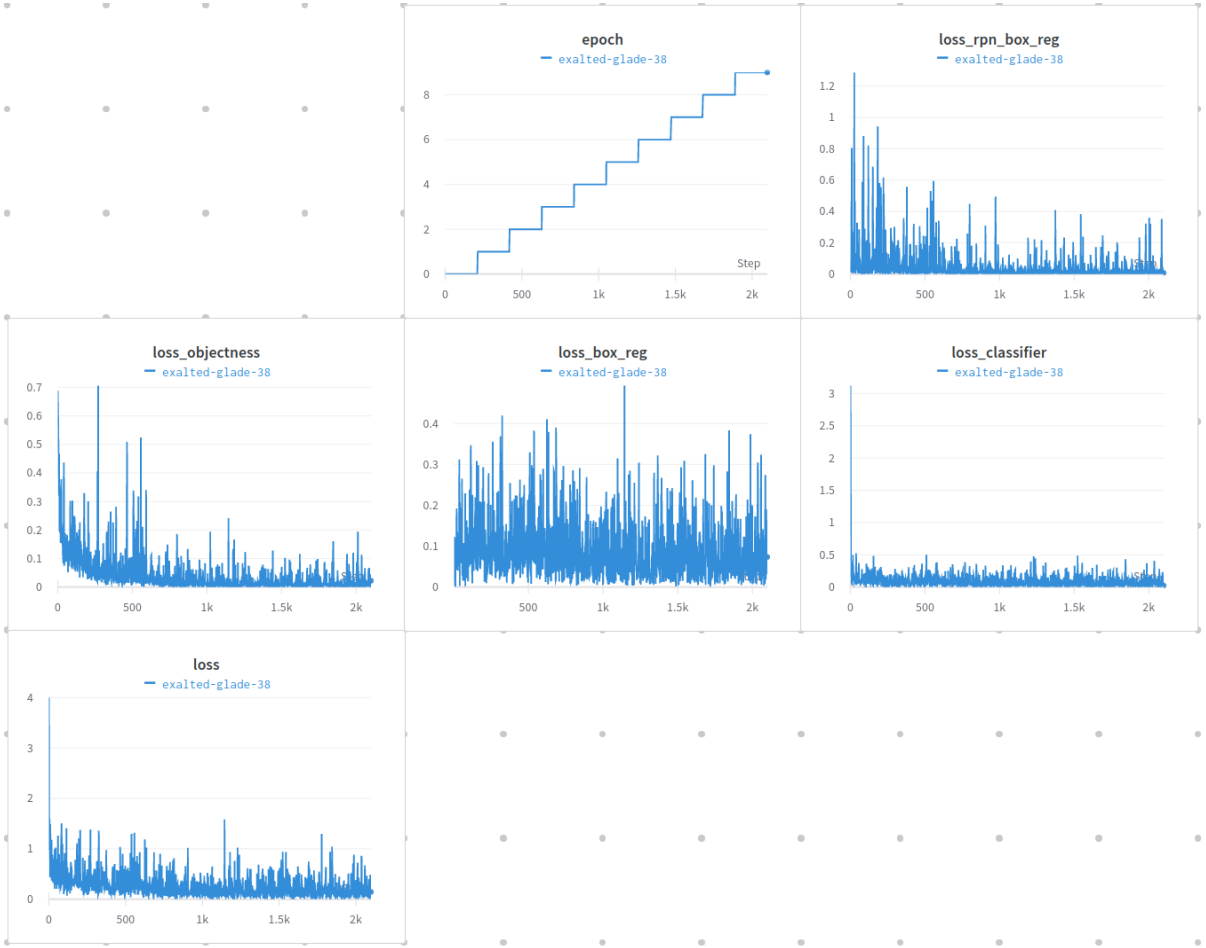
For evaluation we have used CocoEvaluator as recommended by the Mask-RCNN repository.

Plots:

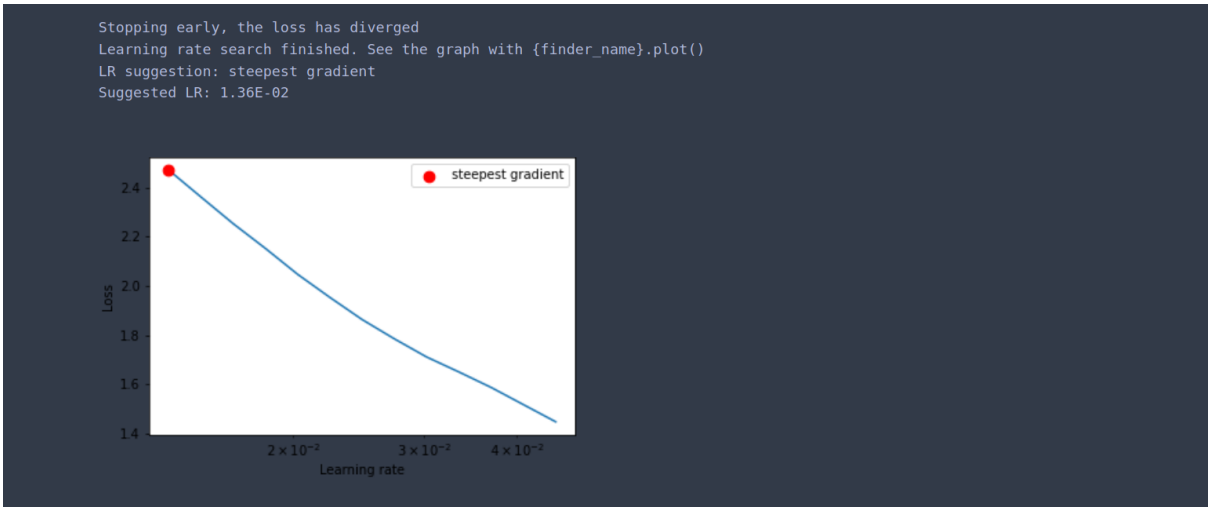
Resnet:

[7] https://stackoverflow.com/questions/46272841/what-is-the-loss-function-of-the-mask-rcnn

mobilenet_v2:



Hyper Parameters:



For each backbone we determined the best learning rate by the means of finding steepest gradient using the wonderful library LR finder.

Comparison:

green - mobilenetv2
orange - resnet

Resnet:

```
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.286
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.453
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.323
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.279
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.274
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.315
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.369
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.424
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.424
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.379
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.420
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.434
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.235
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.431
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.225
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.221
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.228
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.301
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.315
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.347
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.347
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.321
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.365
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.377
```

Mobilenetv2

```
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.083
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.176
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.068
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.049
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.085
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.142
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.142
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.164
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.164
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.091
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.157
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.293
```

Description of runtime environment:

jupyter notebook, GPU GTX 1060, Ubuntu

List of the most important libraries and tools used:

- DVC
- jupyter
- pytorch
- docker
- fastapi
- wandb
- mask-rcnn
- yoloact

Link to repository:
https://github.com/trebacz626/cv-project-3
Documentation of the project can be found in the readme.md of the repository linked above.

Bibliography:
Sources were added to the footnotes regarding certain topics.