Report

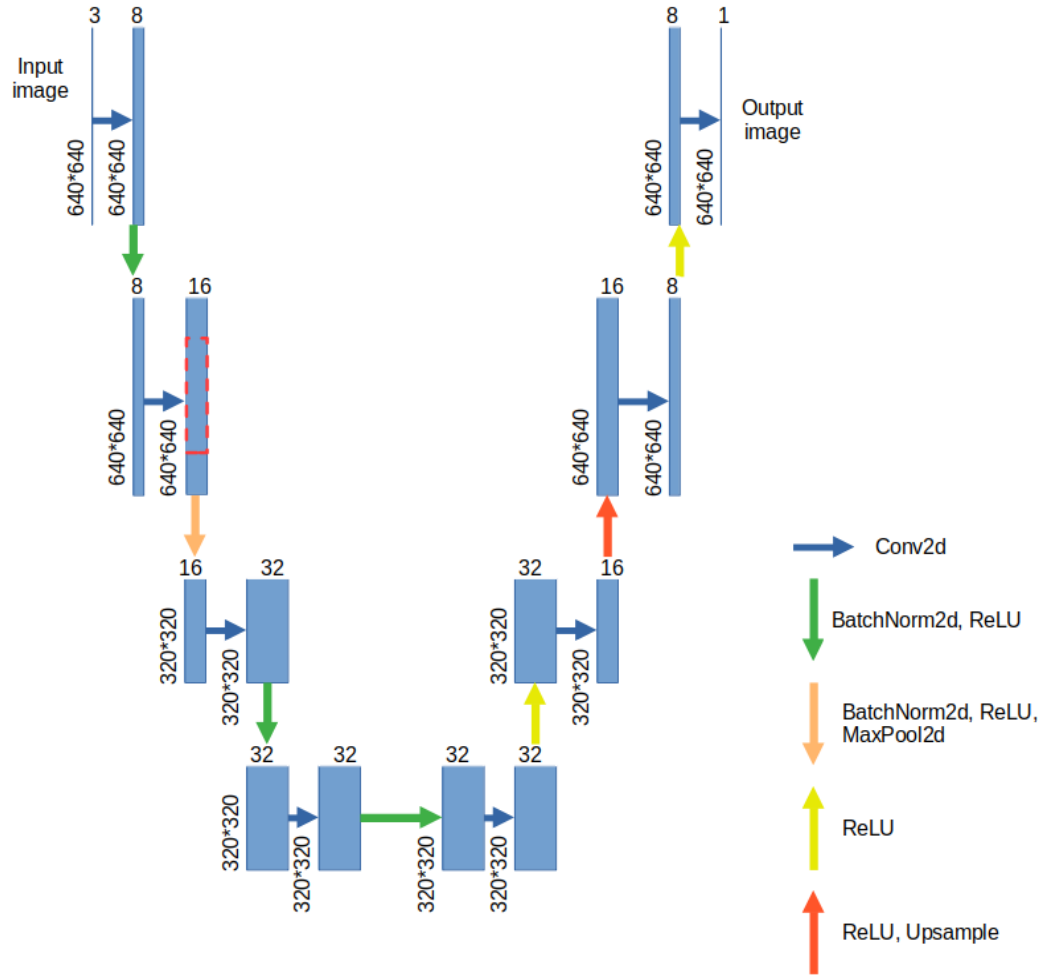
Goal

In this task my goal is to provide a solution that contains a data processing pipeline which processes input image with the graph and handwritten text and returns denoised gray scaled image. The goal on level above is to get familiar with the Pytorch framework and deep learning pipeline.
Business logic for this task was not defined.

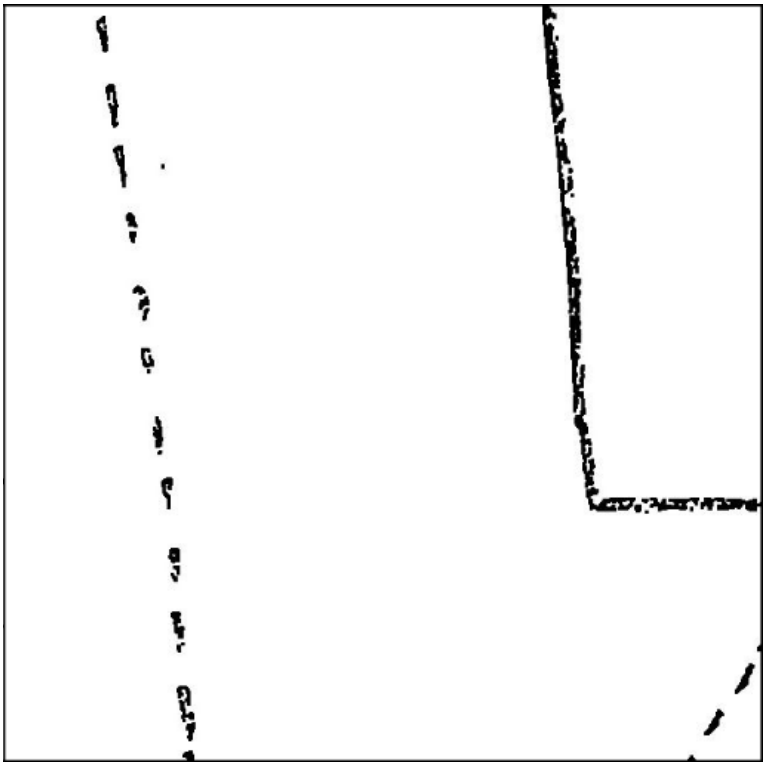
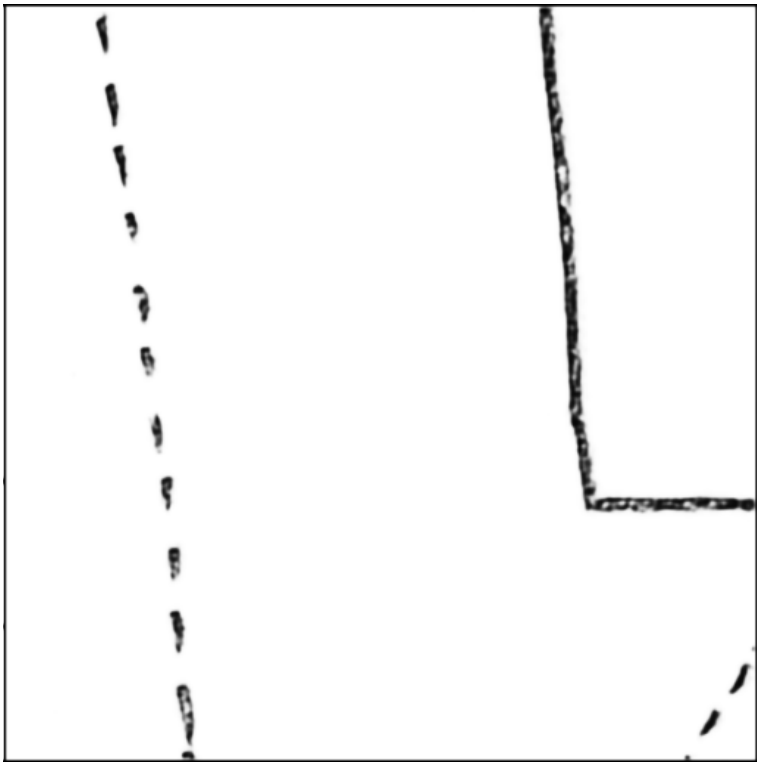
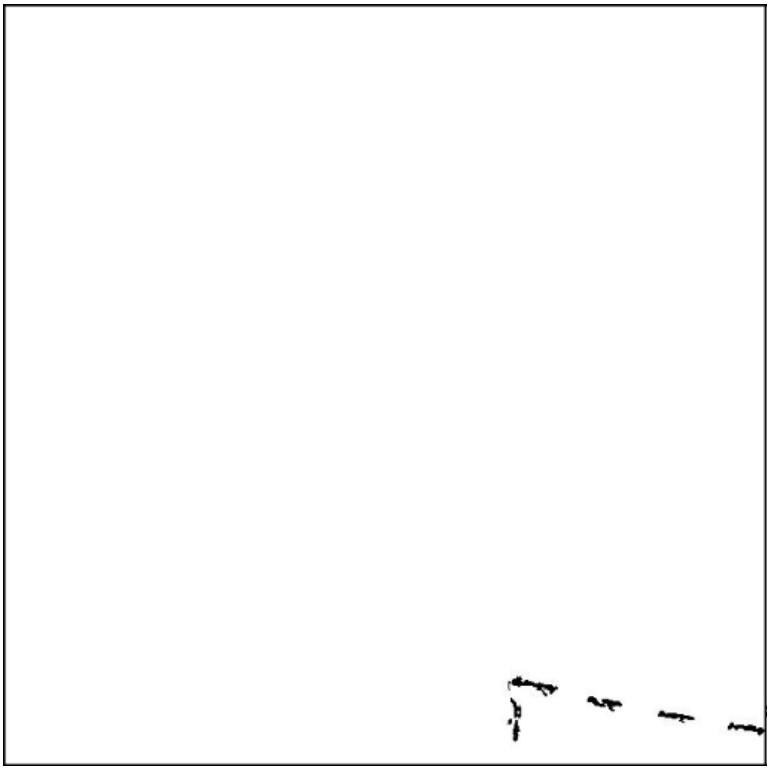
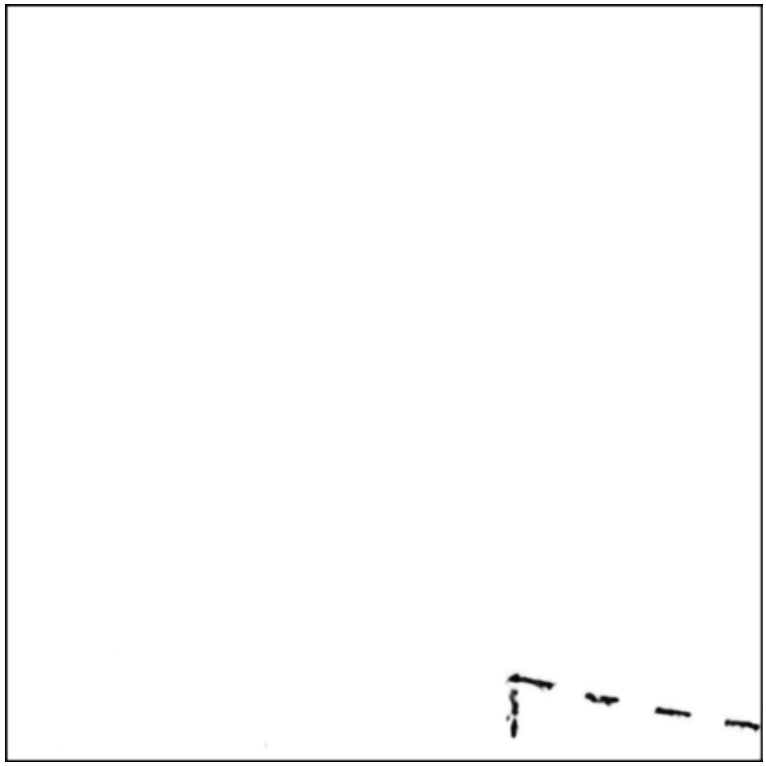
Solution

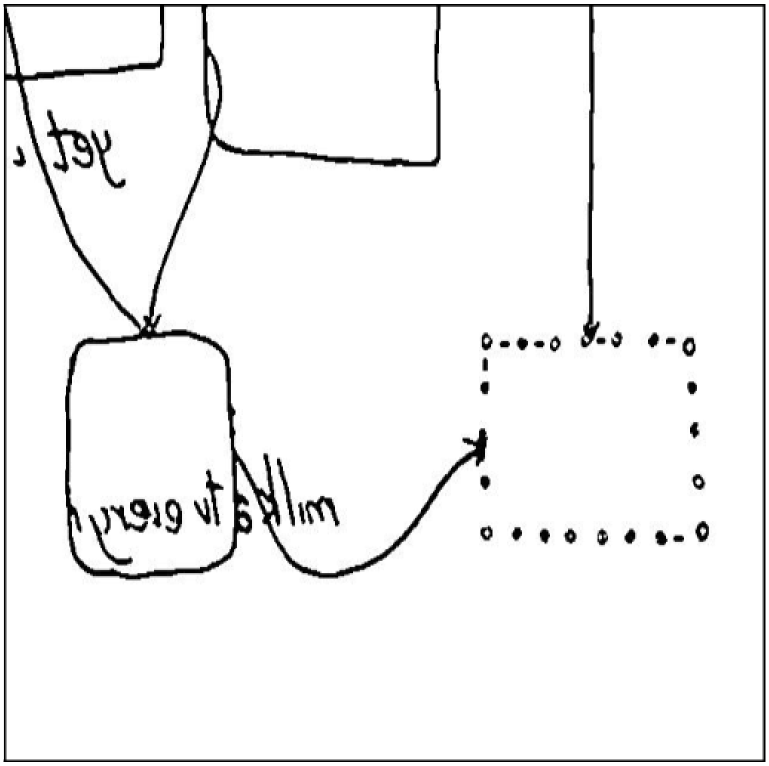
Looking ahead, if you do not want to scroll more than 40 pages of this report, I would like to show a brief summary of my work: the model graph and some examples of the validation outputs.

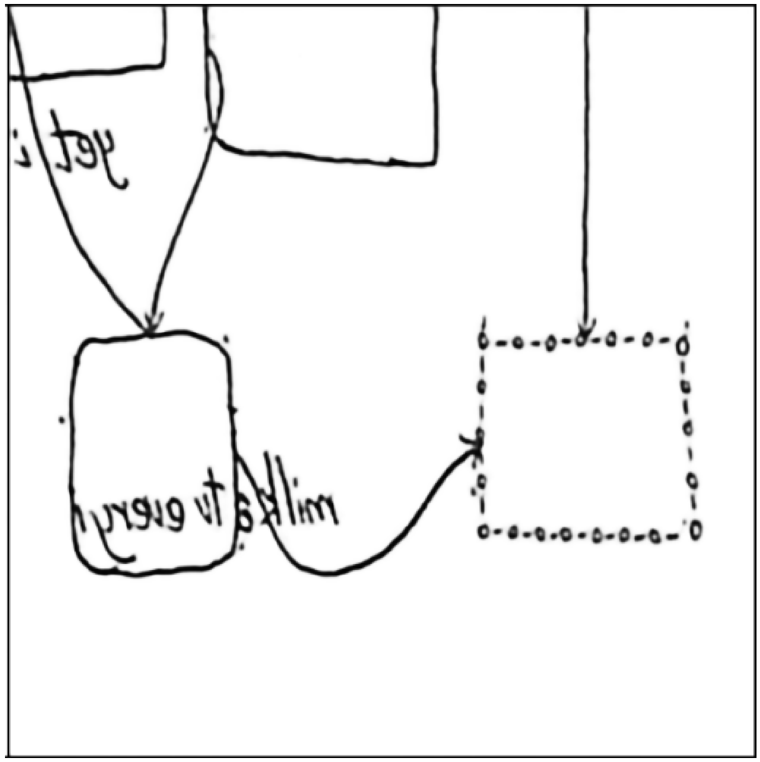
model scheme

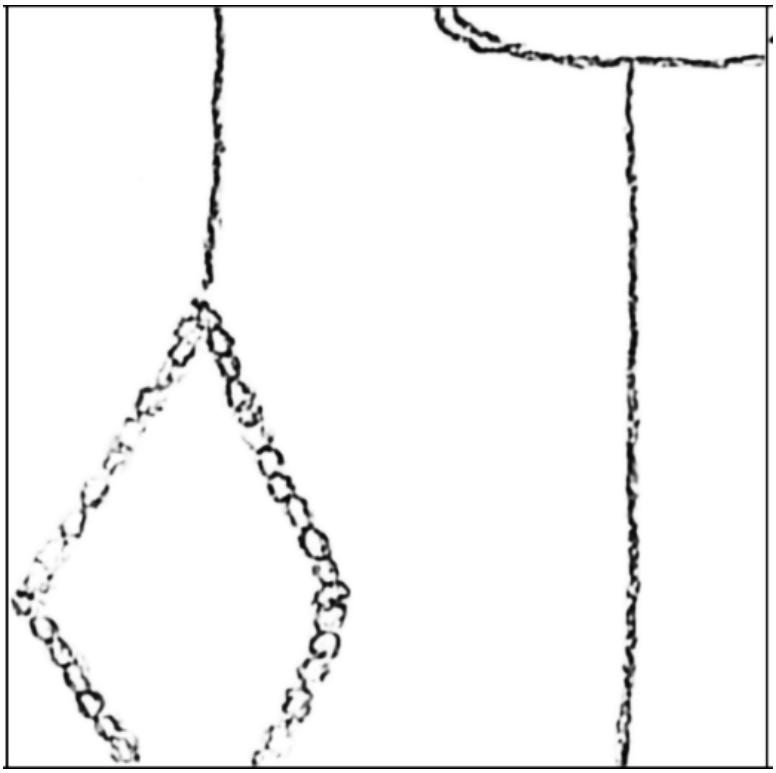


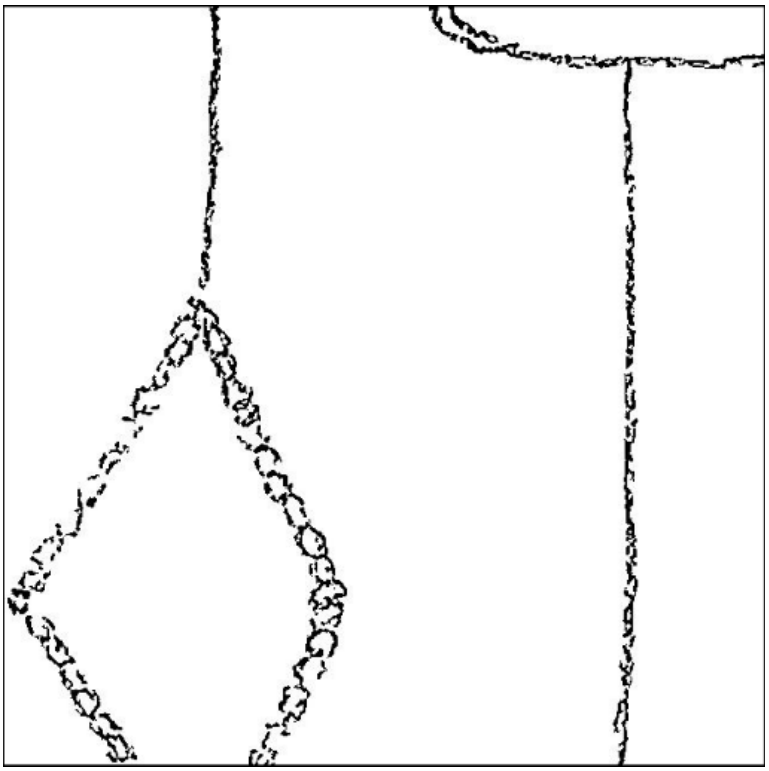
Model's validation outputs

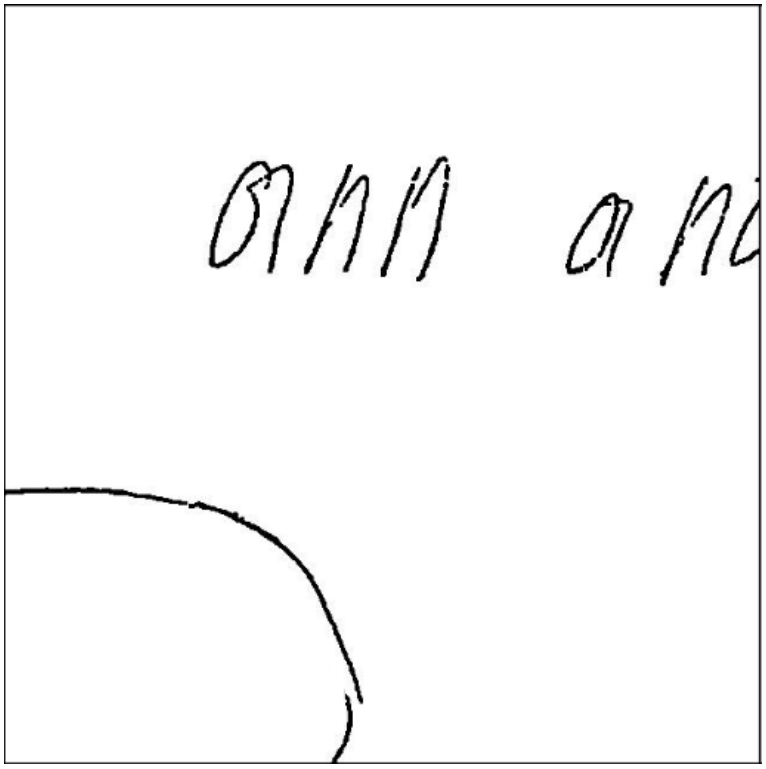
True	Predicted
	
	

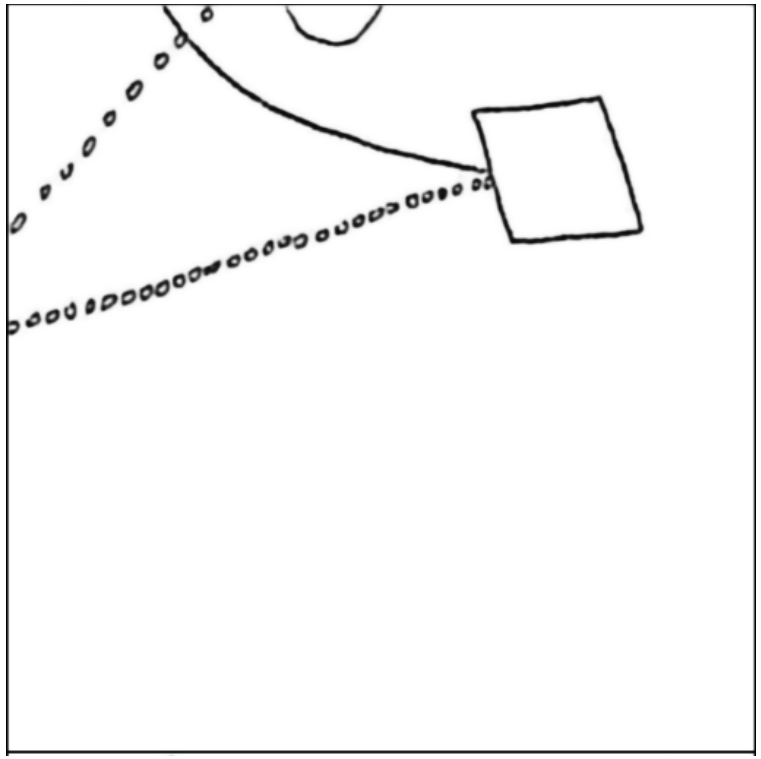




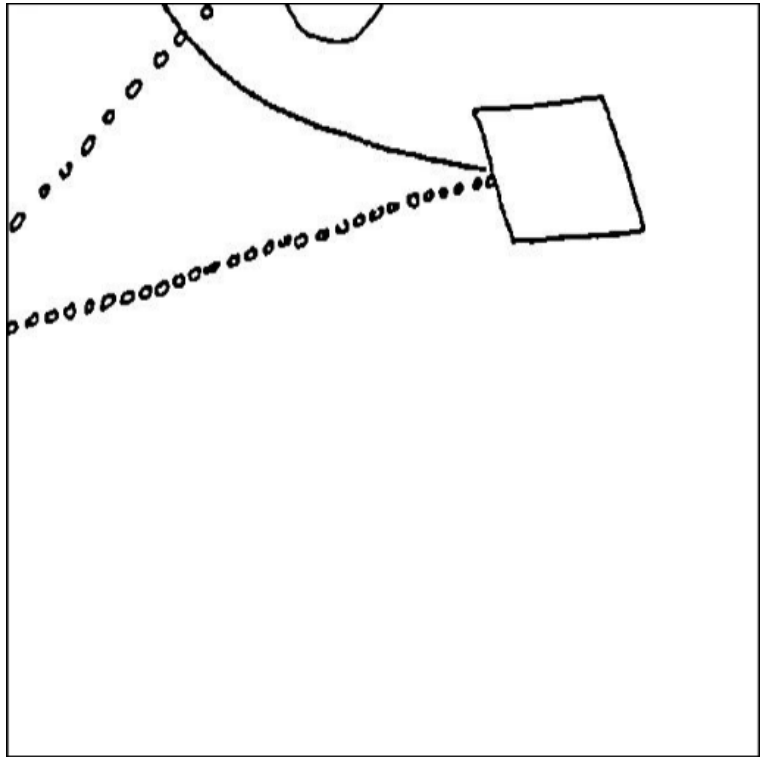


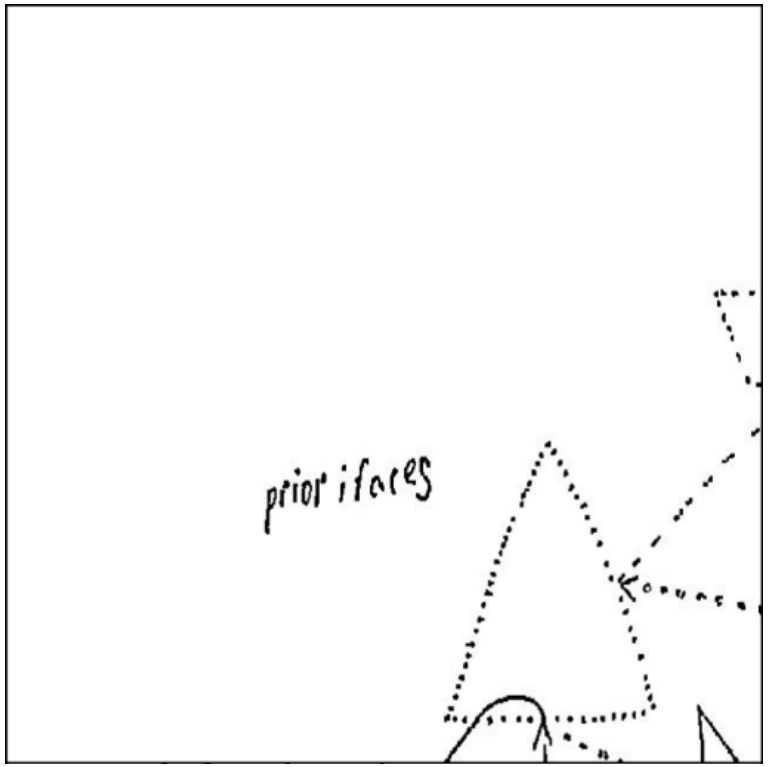


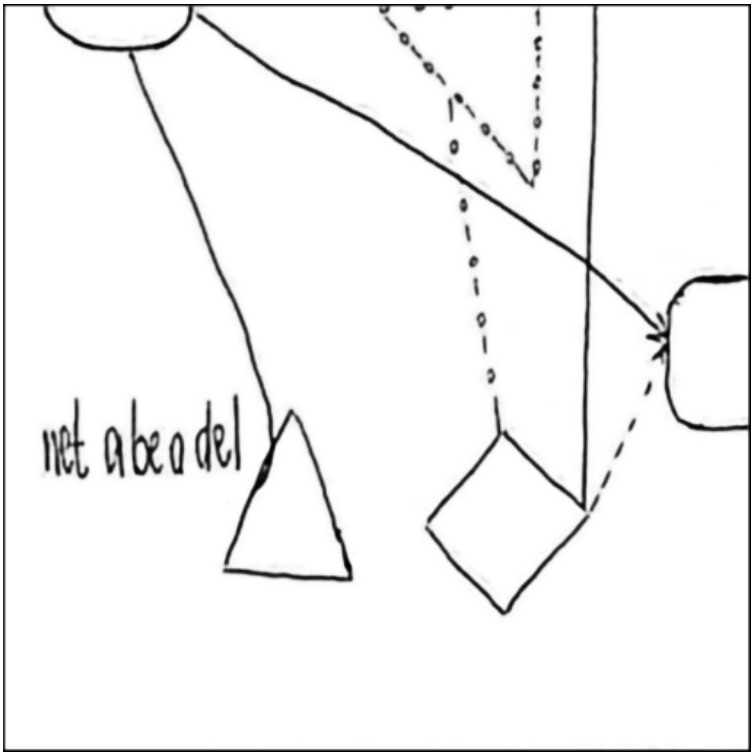


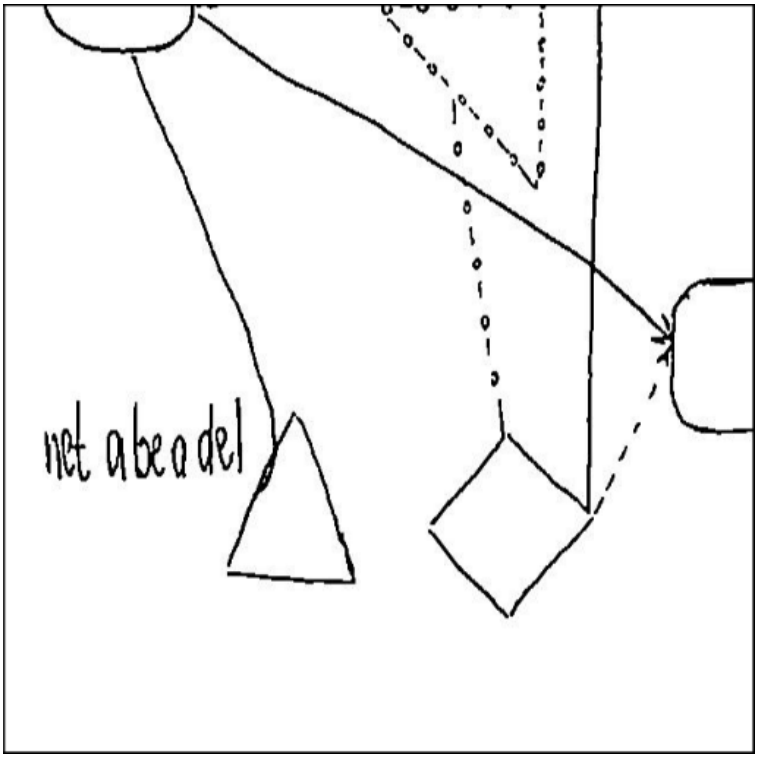


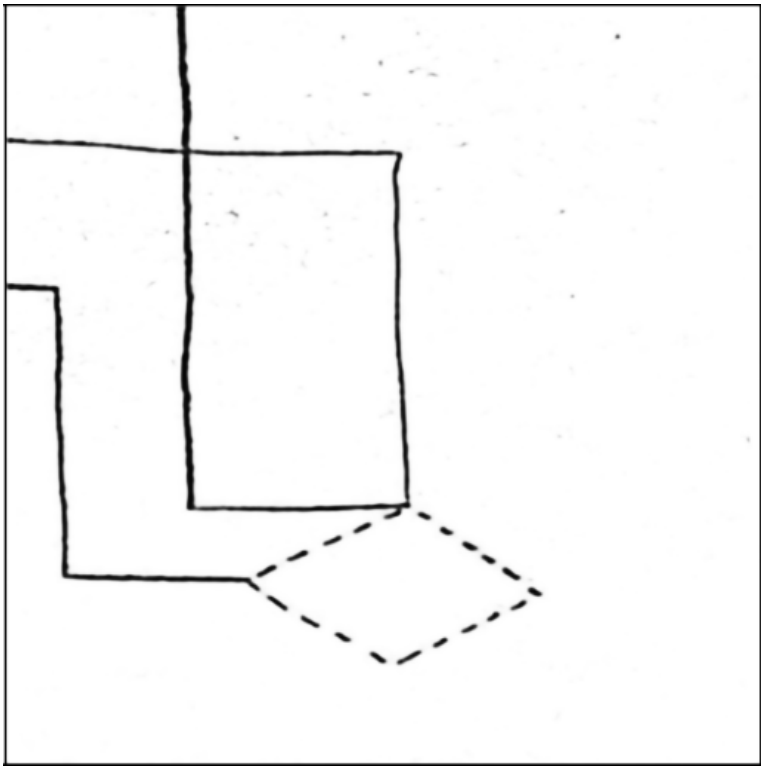

prior i fores

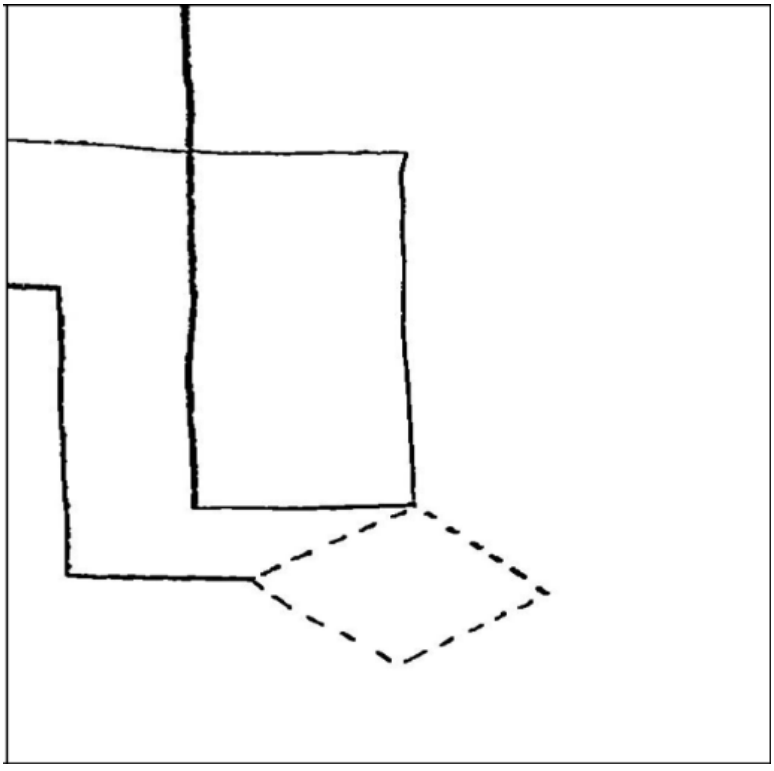


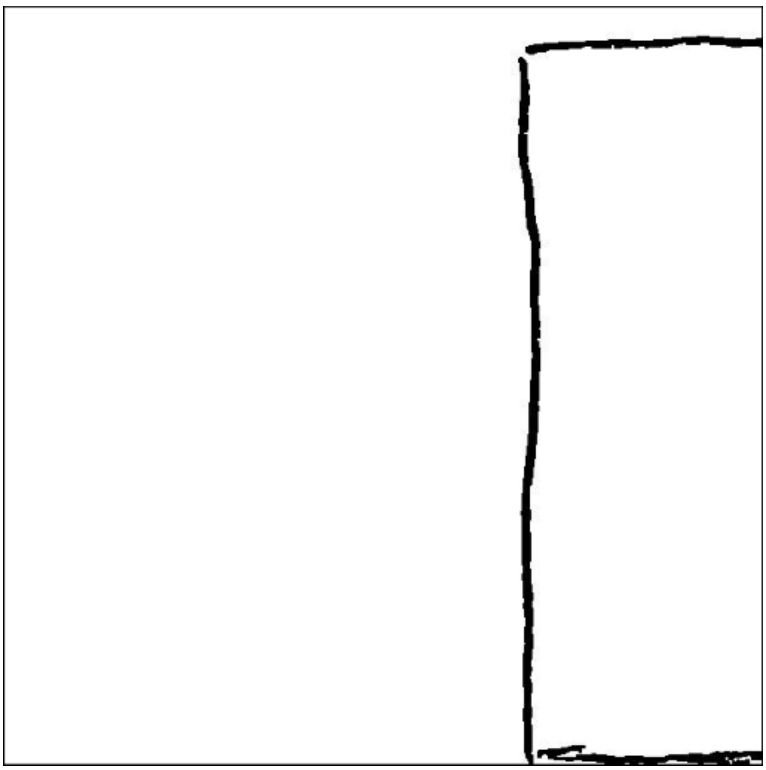

prior i fores

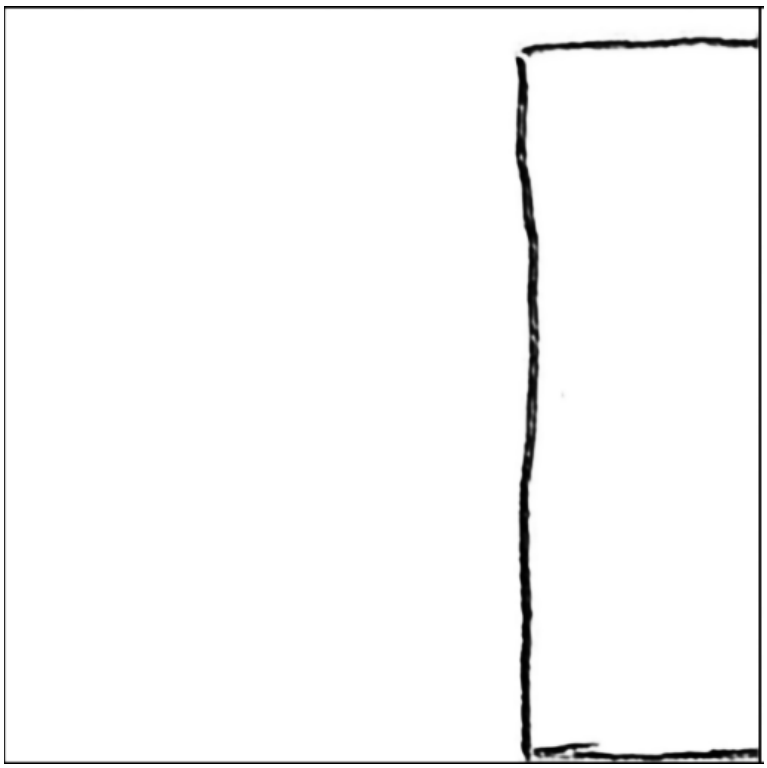

net a be a del

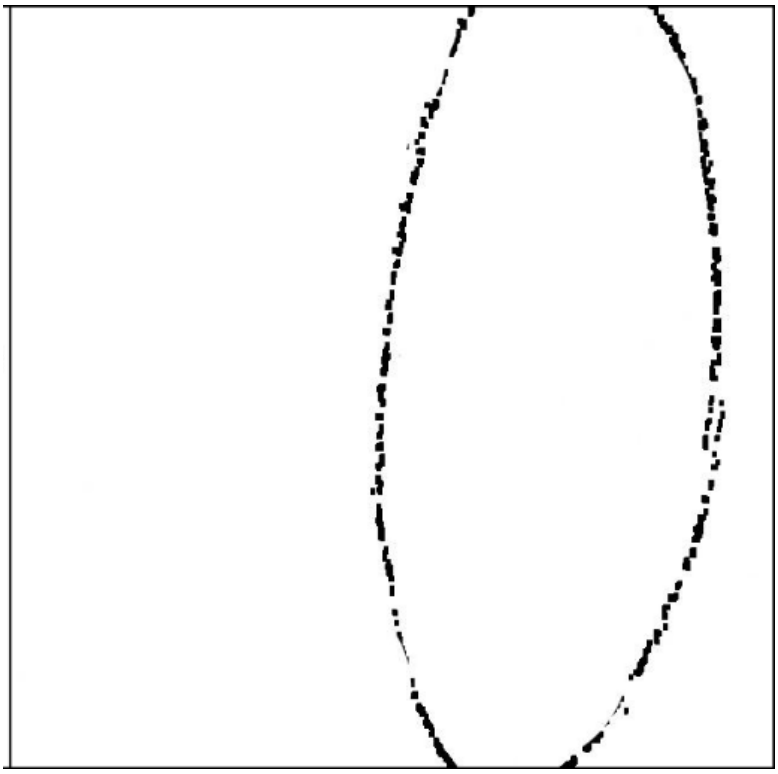

net a be a del

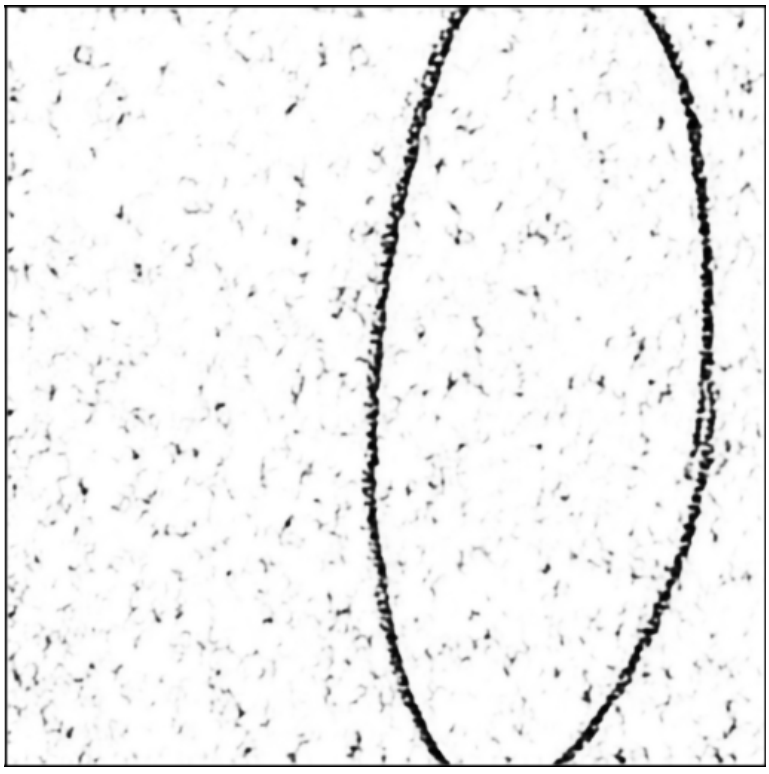


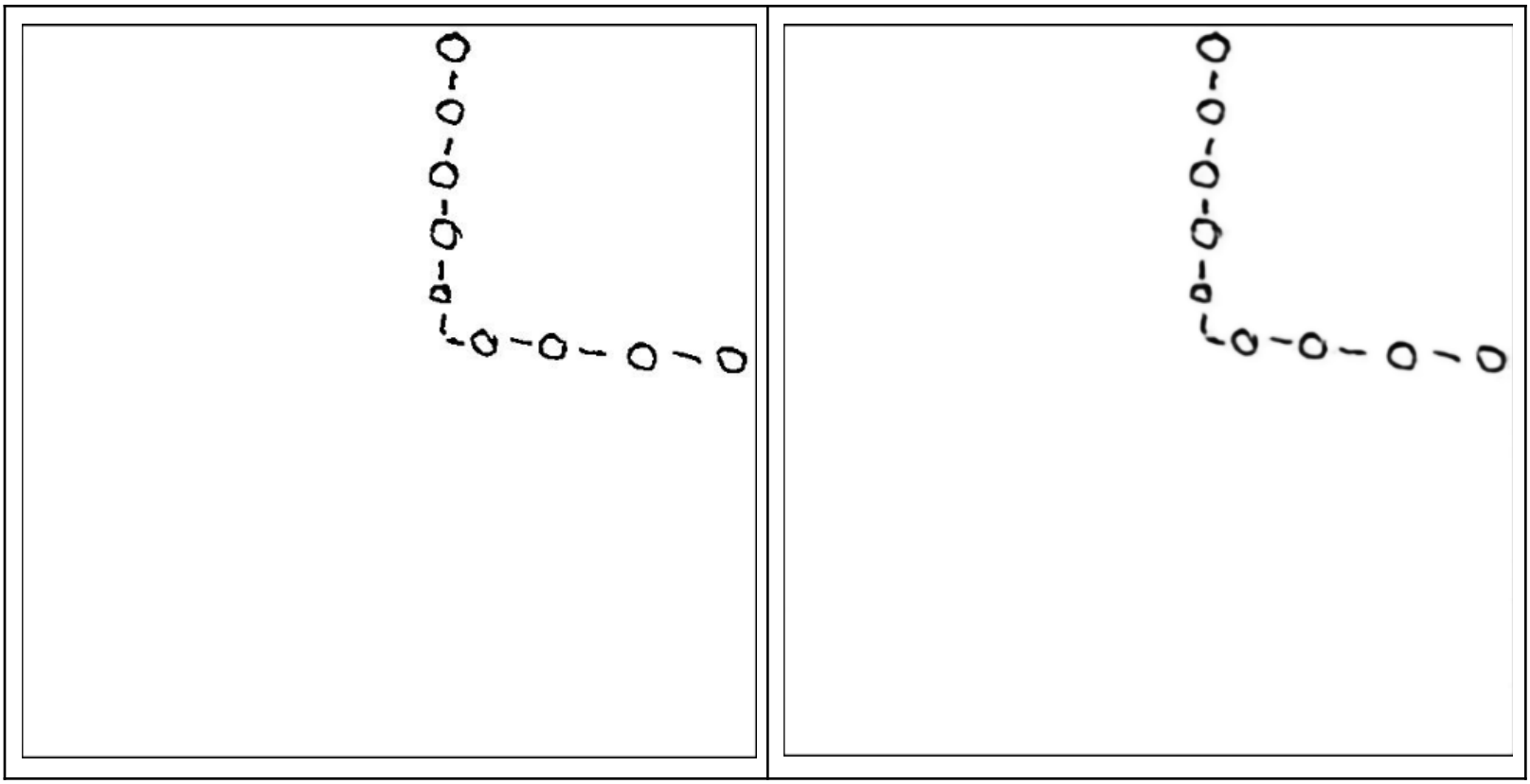








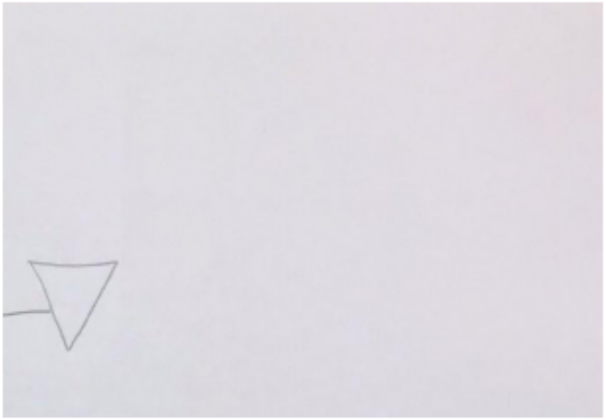




EDA

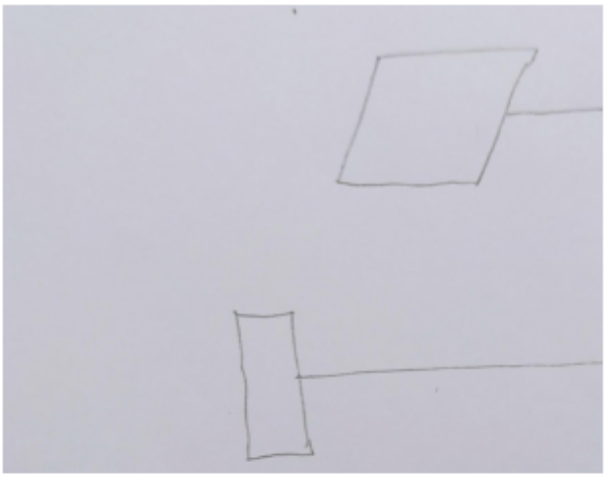
In this task we do not have a familiar bunch of classes as labels but denoised pictures instead. Firstly, let us take a look at what our source and label pictures look like.

020003.jpg

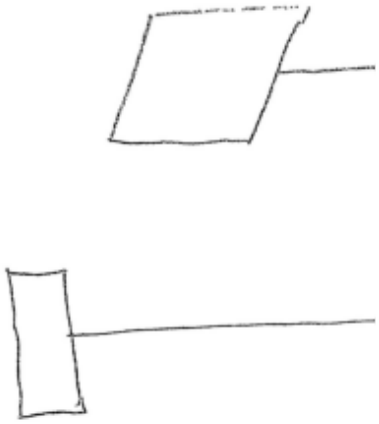


020003.jpg

010879.jpg



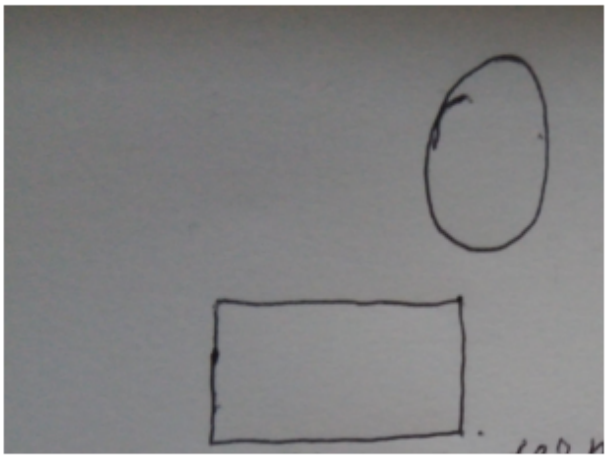
010879.jpg



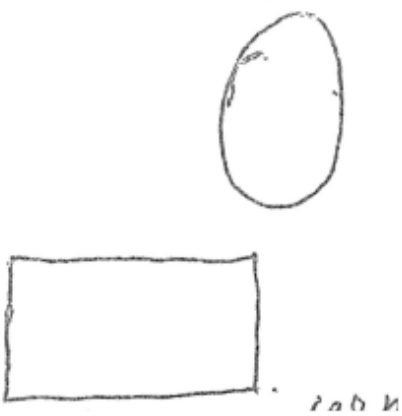
012391.jpg

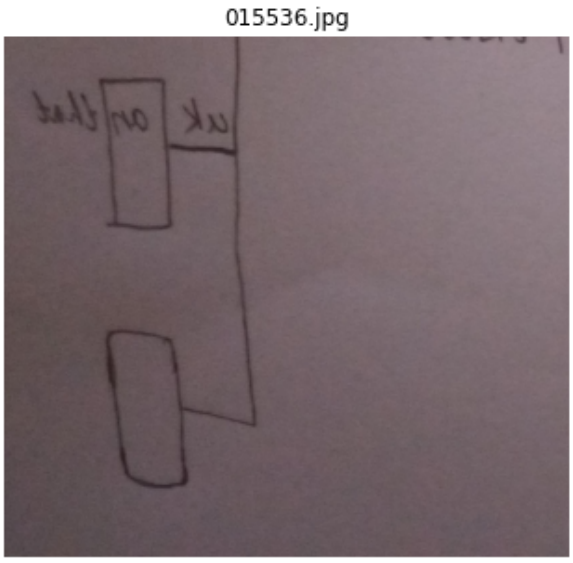
012391.jpg

007990.jpg



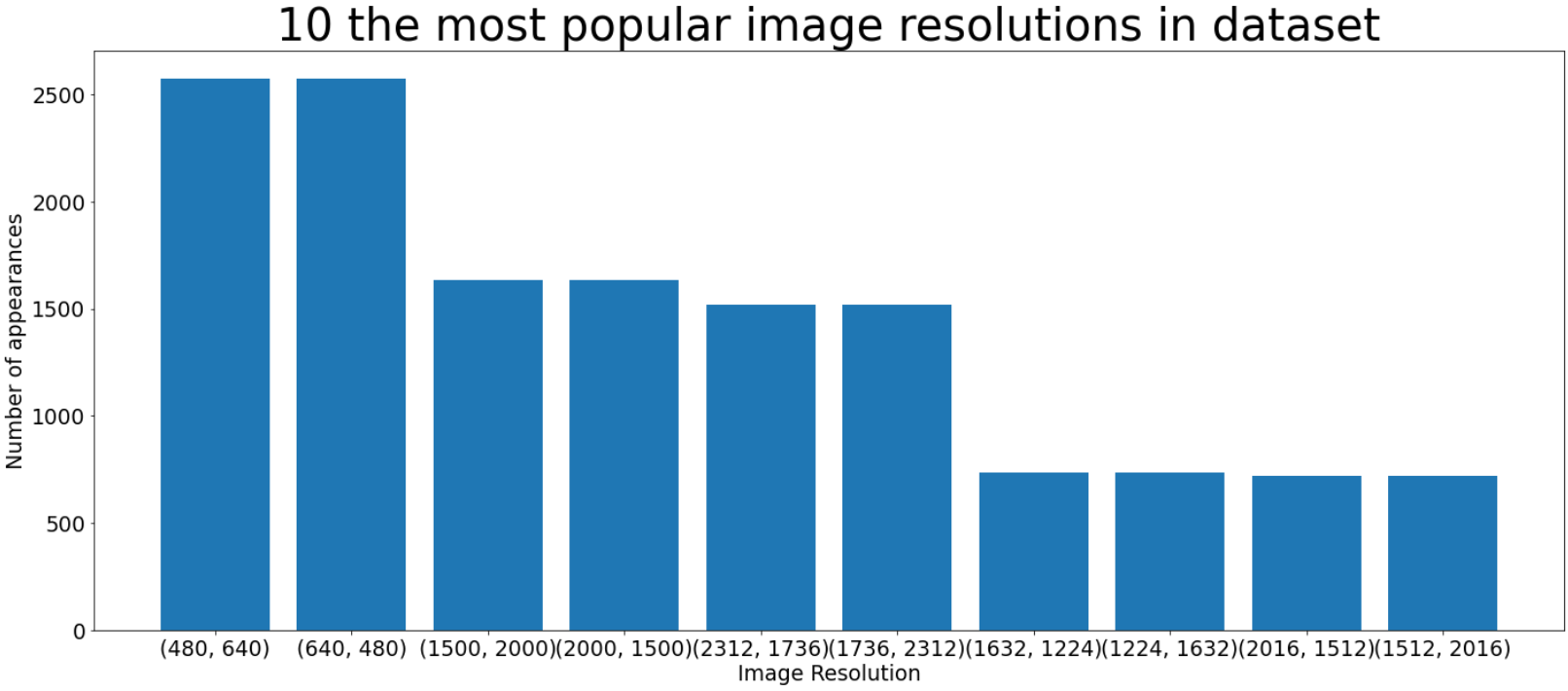
007990.jpg

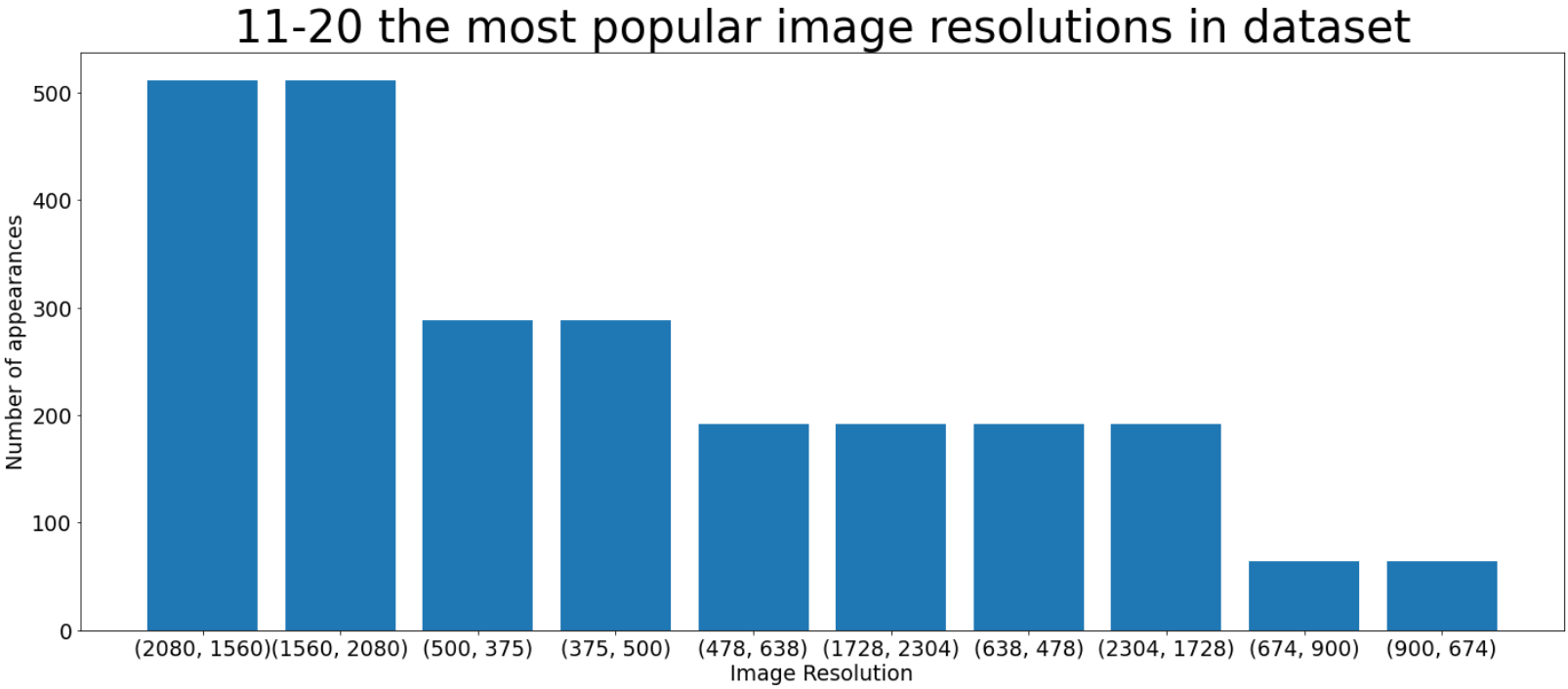




Then I should find out how many pictures are in the dataset, what their sizes are and whether their names and sizes match with label pictures.

As the ‘os’ method showed, we have both **23104** pictures as in source as in label directories but there are **526 different** image **resolutions**. Fortunately, all source-label pairs have the same size but the issue is that pictures in general have so many different resolutions.





Due to the charts above it seems that pictures in our dataset were rotated to double (or quadruple) their amount.

As for solving I could do several things:

- shrink/expand images to the chosen resolution
- crop random image part with the chosen resolution
- write custom collate function for the dataloader which helps process images with different sizes in the batch

First option has an obvious disadvantage: I will use either square or rectangle resolution form and in both cases some pictures will be distorted since some of them are closer to rectangle, and some - to square form.
But since our data consists of pictures of hand-written graphs, their distortion will not affect the nature of the data drastically.

Second one could be not the best option because of the aforementioned nature of our data: most of the pictures' area is just a white background. To avoid conflicts when the source picture is smaller than the cropping area I should use the smallest available resolution but most of our pictures have size with more than 1000 px on each side.
The smallest resolutions in the dataset are 205x167 and, respectively, 167x205. Based on this, we should use a 160x160 crop (not 167x167 to have the possibility to shrink the image in the model without workarounds).
This means that, cropping a 160x160 piece from the 1500x2000 picture, we would lose 99.1% of pixels of the original image.

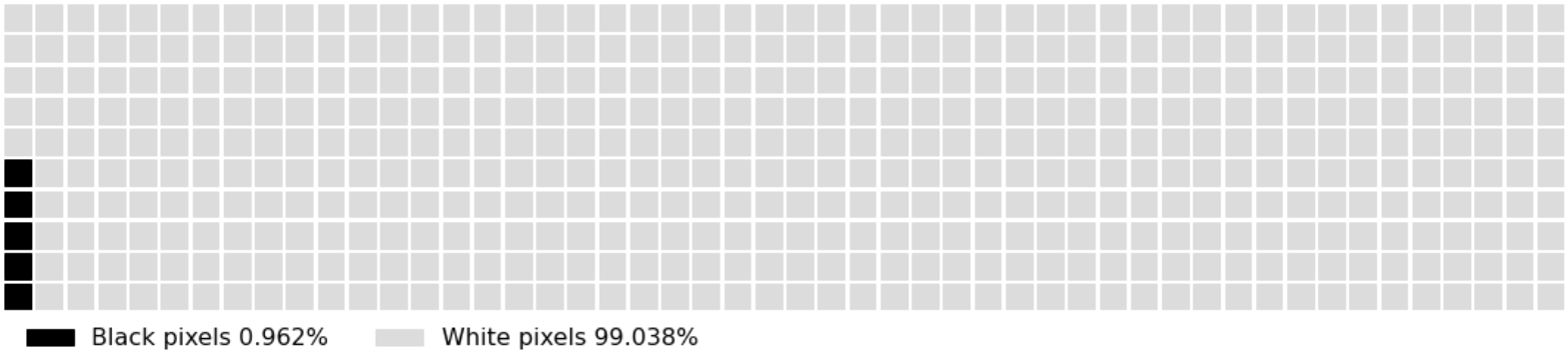
With the third option I can create a custom function that will add padding to the smaller pictures in the batch or resize all pictures to some size.
It would be useful to not crop the big images but also it will be harder to estimate the model quality since all batches have different picture sizes.
I decided to use the hybrid approach: expand images to the 640x640 resolution if they are smaller and crop 640x640 piece if they are bigger.

Another issue is that **1760** source-label pairs have **different file names**. As it turned out, the reason for that is 1760 source pictures with **capital letters in the file extension** ('.JPG' instead of '.jpg'). This would cause errors while data loading so I have to handle it.
To avoid problems, I use 2 sorted file lists in the custom MyDataset class: for source and label data. At the same time I use the same indexer to both of them so I get a needed label file that can have a slightly different name to the source file (file format in our case) since file names in source-label pairs excluding the format remain the same.

As we can know just from seeing the label pictures, our data is highly imbalanced: most of the pictures are just white background with not so many black pixels on it. Model goes the easiest way to minimize the loss: just imitates the background color on all picture's area and does not even try to reproduce graphs from the source picture.

A possible way to avoid it is to change the loss function which can consider class imbalance.
During the first tries I used Mean Absolute Error which did not do it.
From all loss functions available in PyTorch there is one that has such a possibility: "BCEWithLogits Loss".
I just need to know the ratio between the negative (black pixels) and positive (white or gray background) examples on each picture.

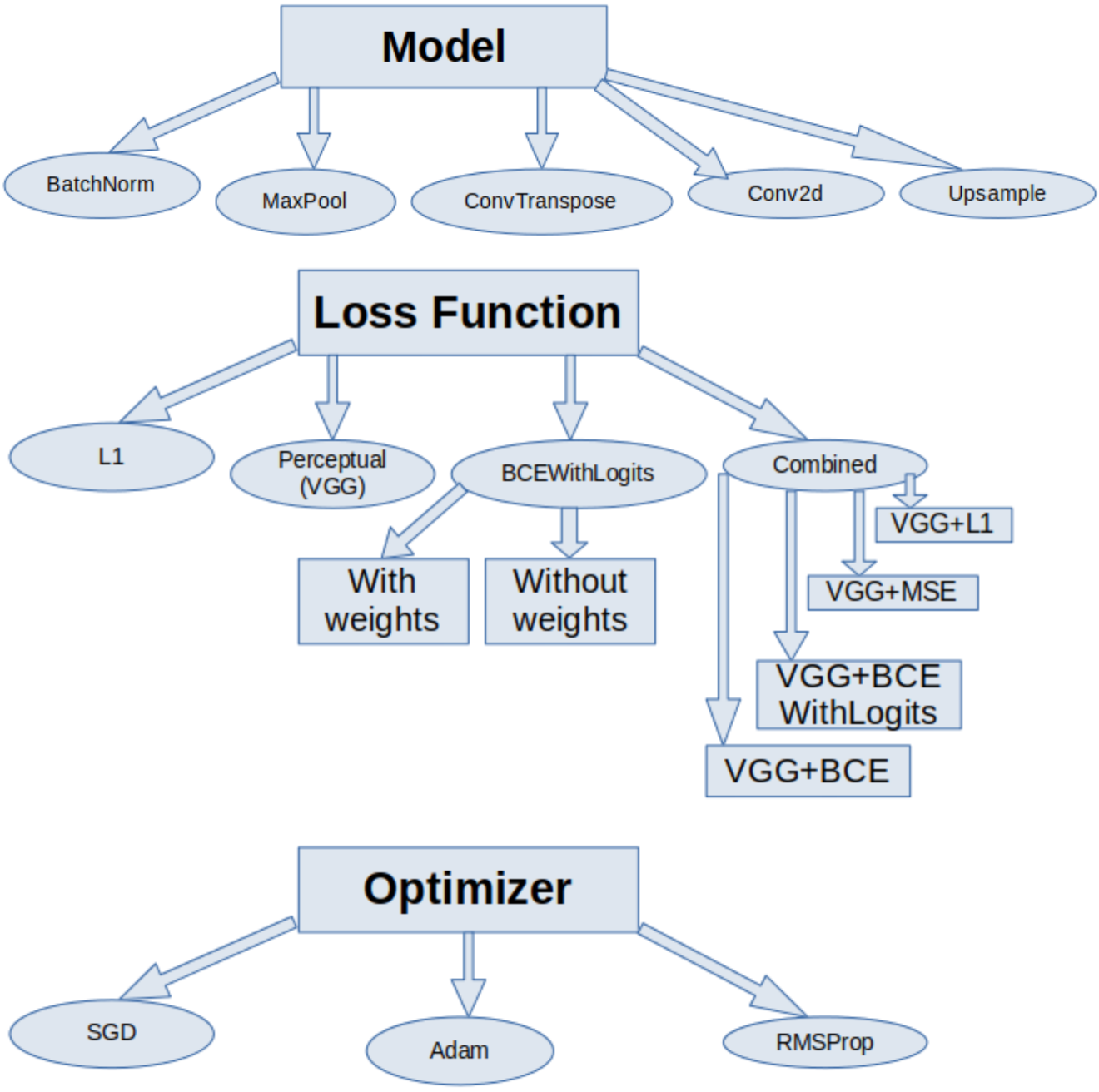
After some calculation I got the exact percentage of non-white pixels on all images in the dataset:

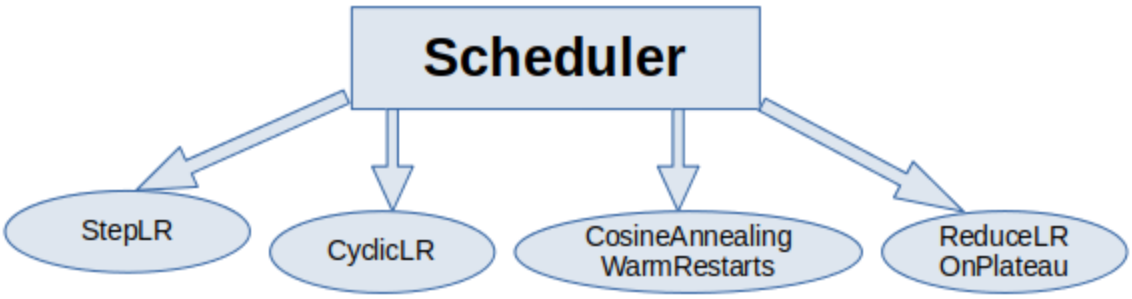


This ratio will help us further during the modeling.

Modeling

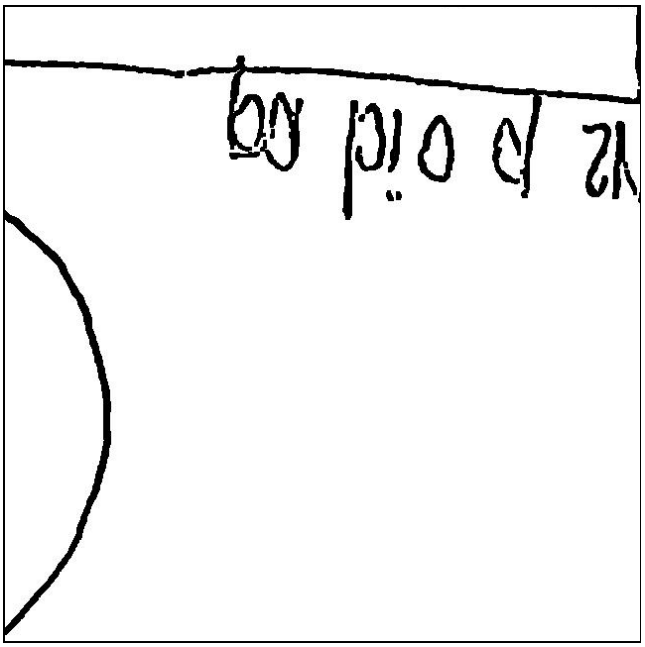
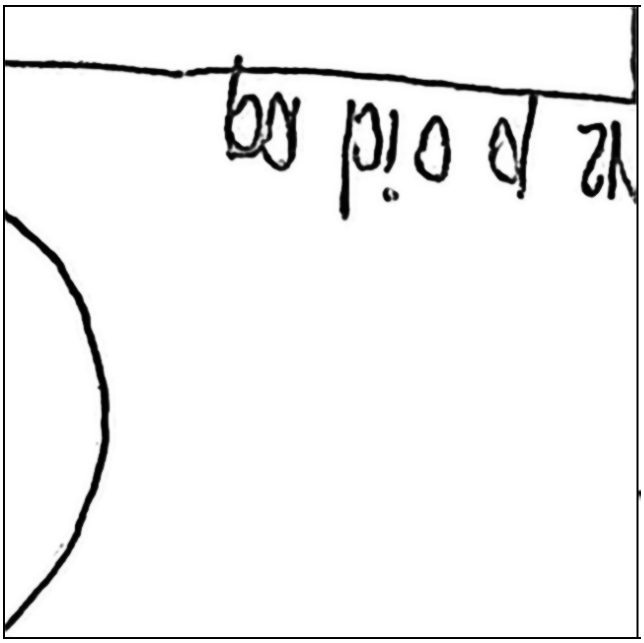
In general, during modeling I used to try many different combinations of layers, losses, optimizers and schedulers.
Schematically I can represent them like this:

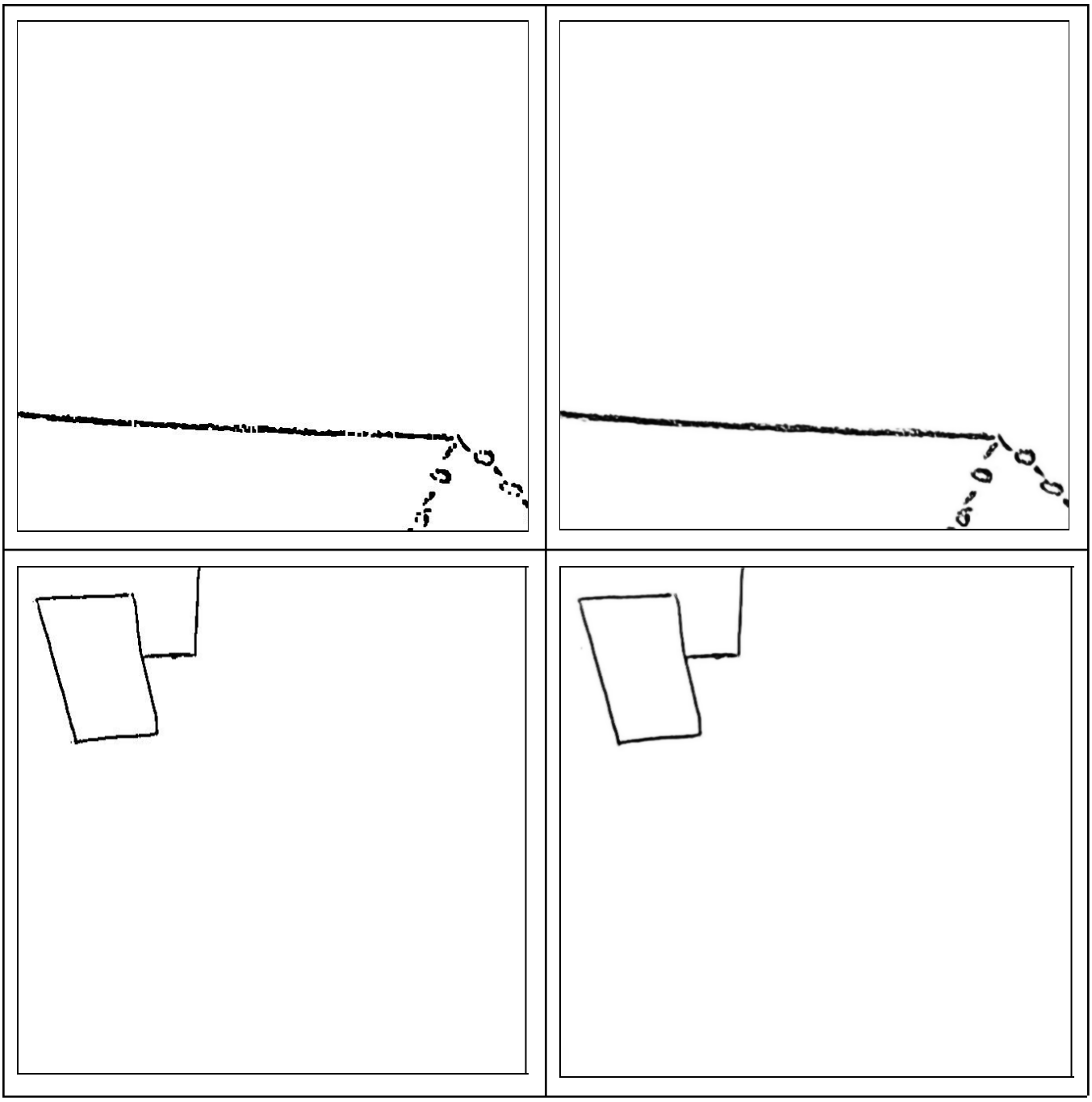




U-Net

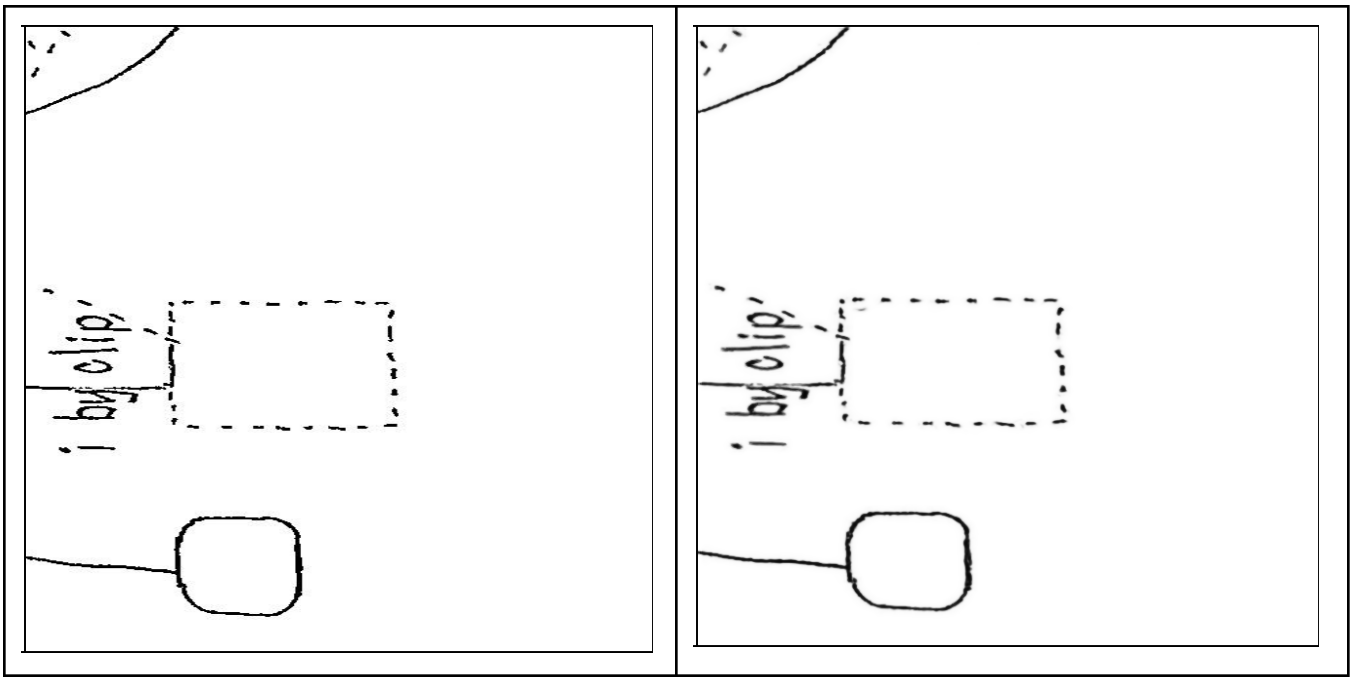
To have some baseline to compare results with, I ran the original U-Net model. We can consider results as even better than our labels in terms of line smoothness:

True	Predicted
	



a one he last

a one he last

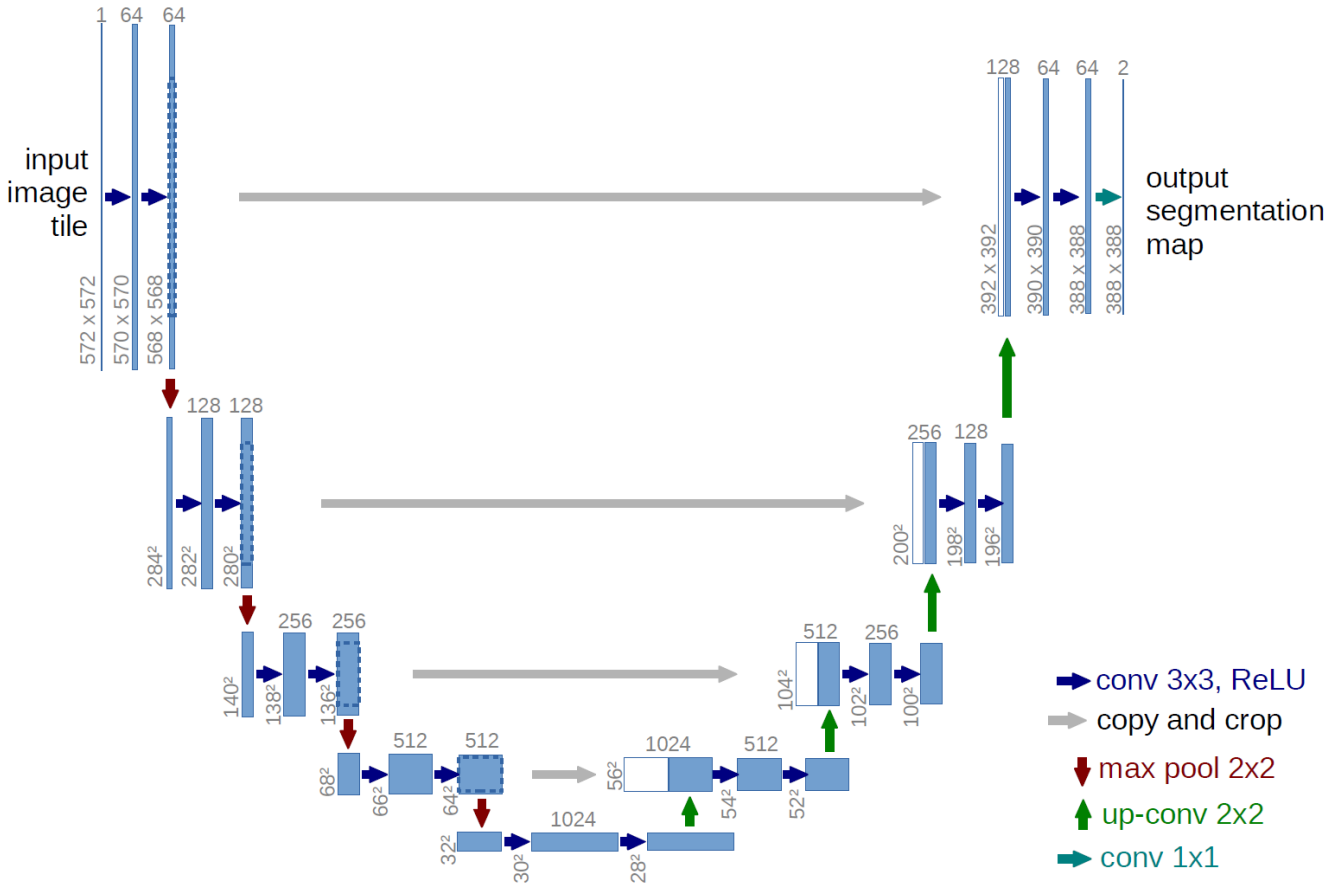


Model

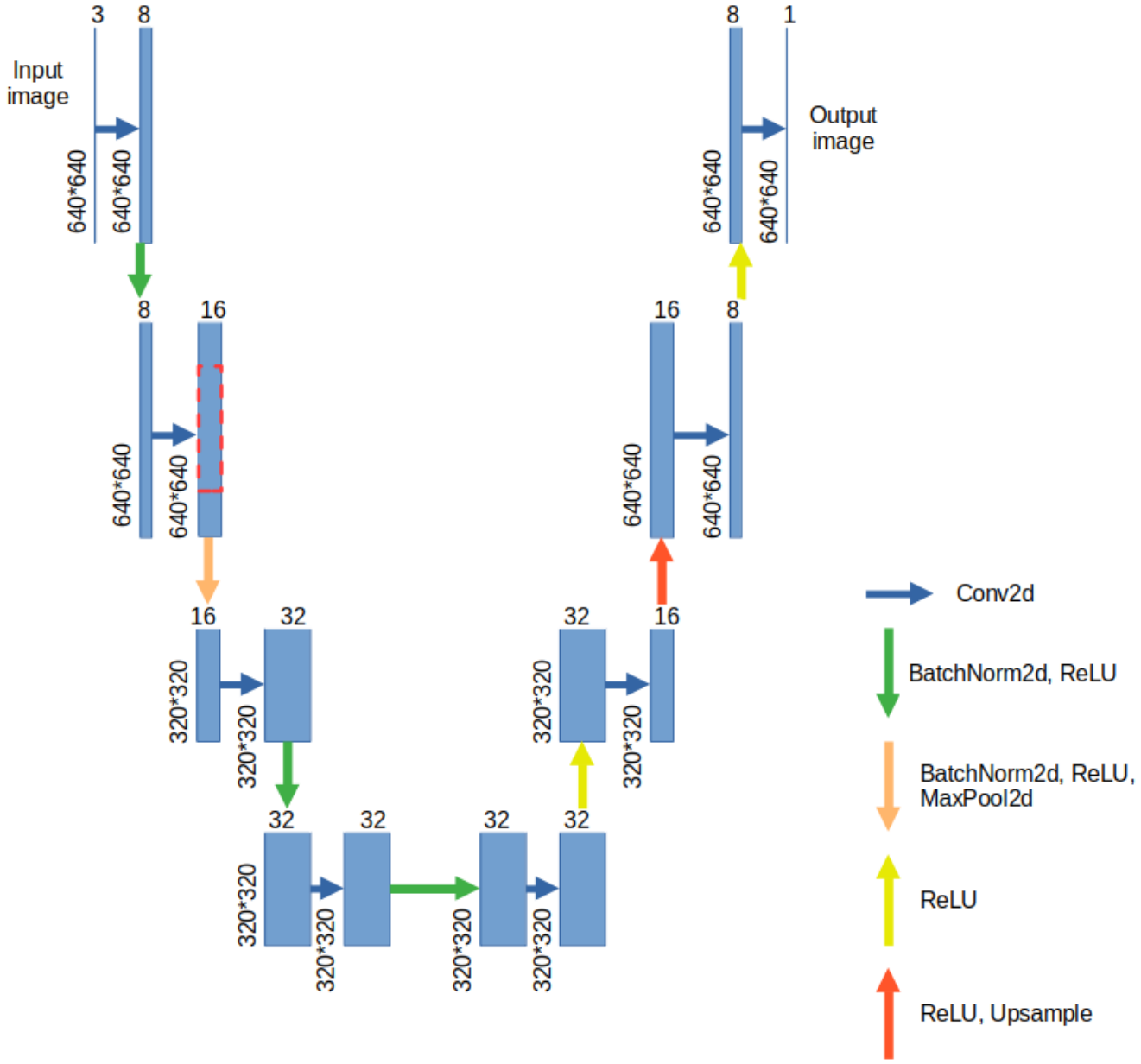
In this work I will use the U-Net-like model with encoder and decoder:

- `Encoder: picture resolution ⬇, channels number ⬆`
- `Decoder: picture resolution ⬆, channels number ⬇`

While the actual U-Net model architecture looks like this:



The main difference between this and my model will be in the absence of residual connections (copy and crop grey arrows) and channel numbers.



But, jumping ahead, I will try residual connections as well.

Let us stop in more detail on each layer I used to try for my model.

BatchNorm computes the mean and the standard deviation of the pixel values (x_{nchw}) separately in each channel (c):

$$\mu_c = \frac{1}{BHW} \sum_{n=1}^{B} \sum_{h=1}^{H} \sum_{w=1}^{W} x_{nchw}$$

$$\sigma_c = \sqrt{\frac{1}{BHW} \sum_{n=1}^{B} \sum_{h=1}^{H} \sum_{w=1}^{W} (x_{nchw} - \mu_c)^2 + \epsilon}$$

And replaces each pixel value in channel c with the following value:

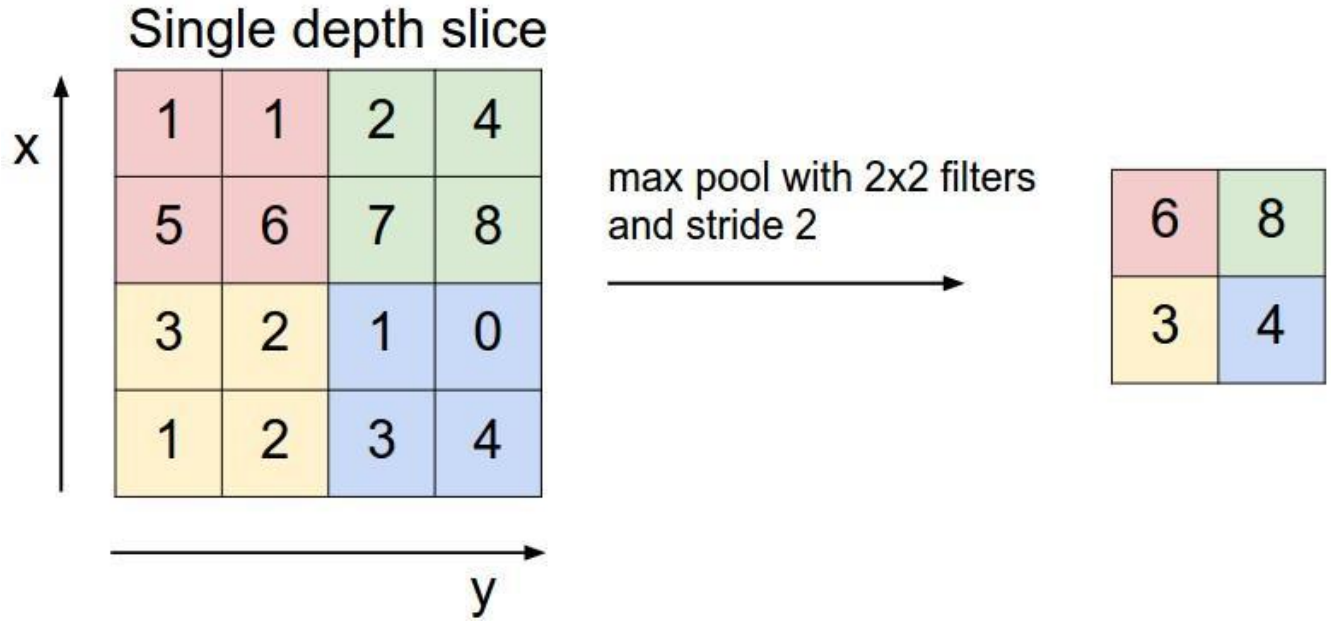
$$BN(x_{nchw}) = \gamma_c \frac{x_{nchw} - \mu_c}{\sigma_c} + \beta_c$$

Where γ_c and β_c are trainable parameters.

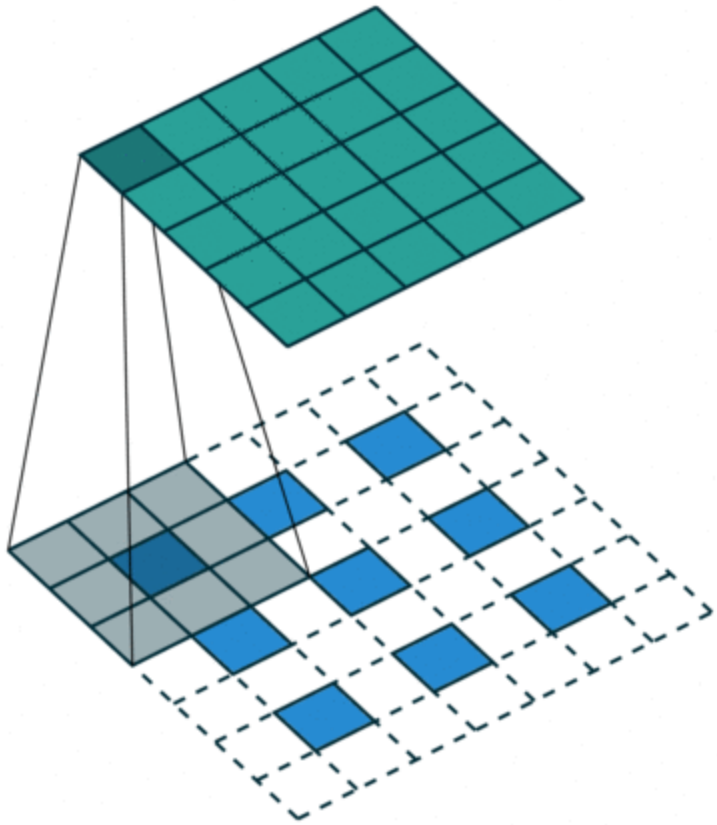
Batch normalization has been shown to be the most needed layer in the model since without it I could not get any real result except for the monochromatic or checkered image.

Conv2d is the layer that changes (in my case increases) channel number:

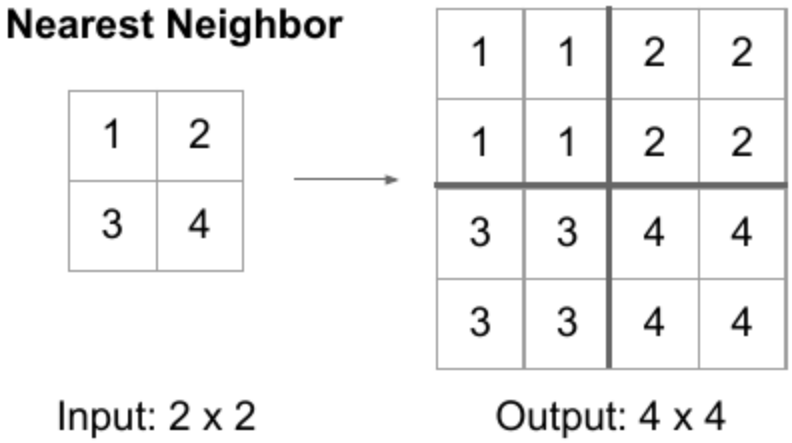
MaxPool2d layer remains channel number the same but, taking 1 maximum value from the kernel, shrinks image:



For the decoder there was an alternative between **ConvTranspose2d:**

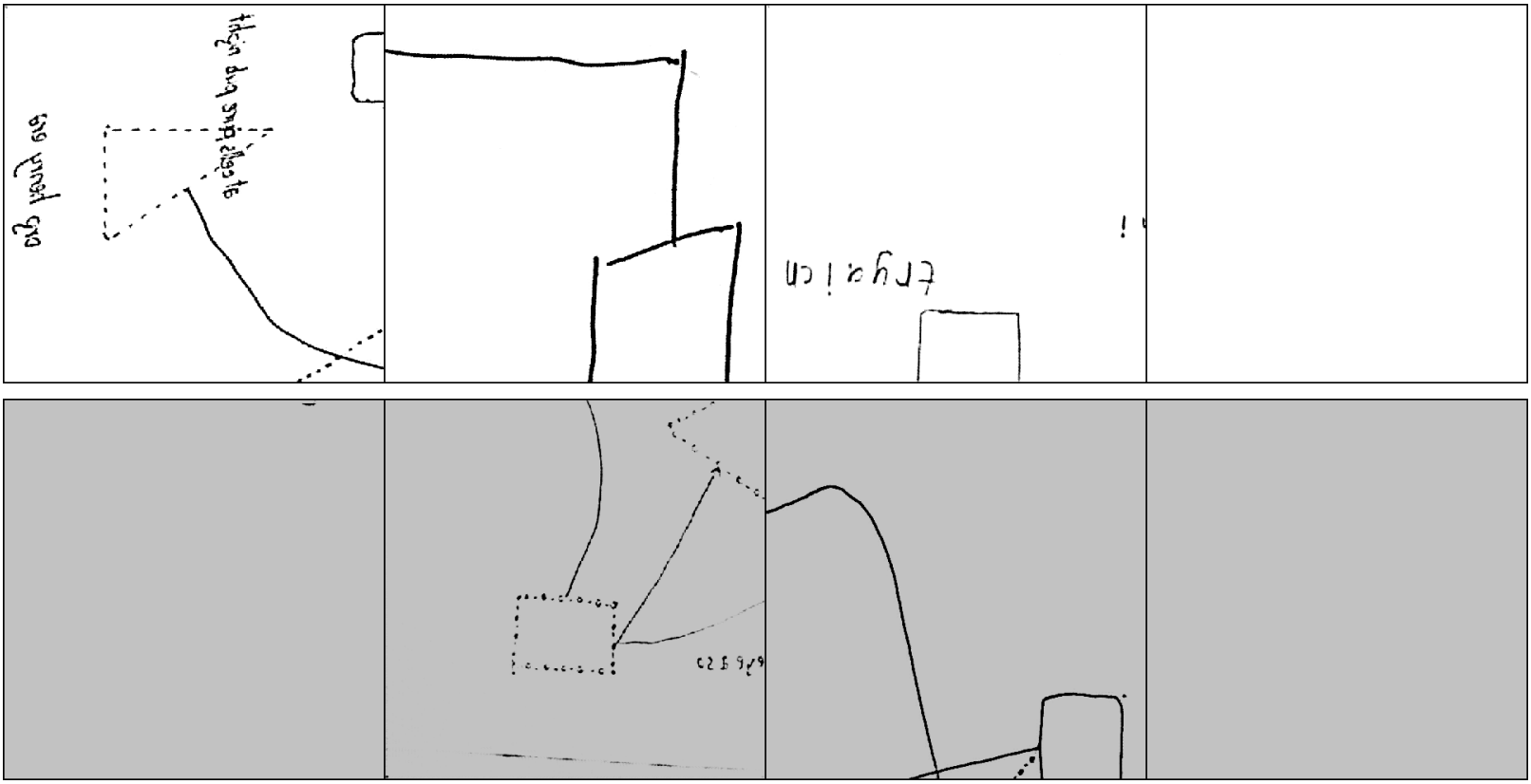


and **Upsample**:



layers.
Both of them can increase image size but in practice models with ConvTranspose was giving unstable outputs: one result could be almost ideal but the other could have a grey background:

validation output batch after 2 model trainings with all the same parameters

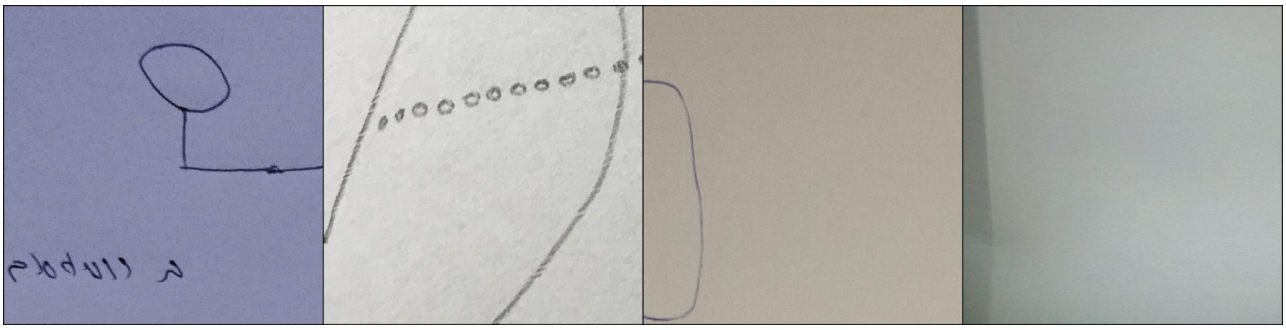
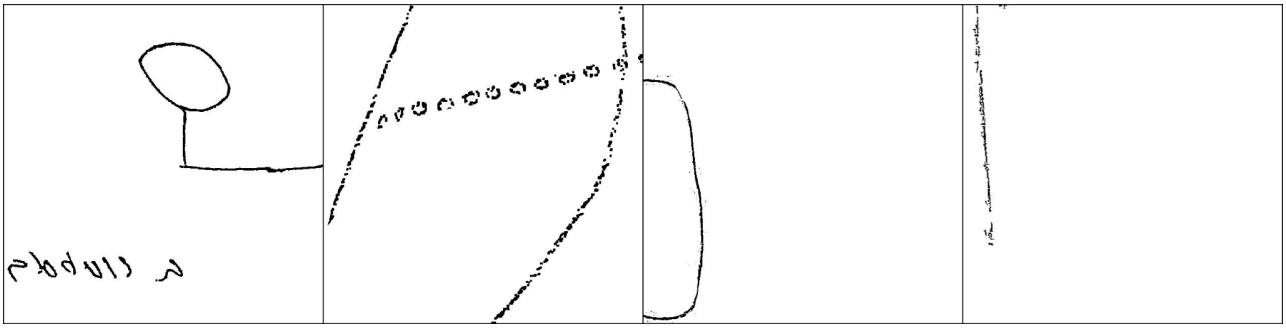
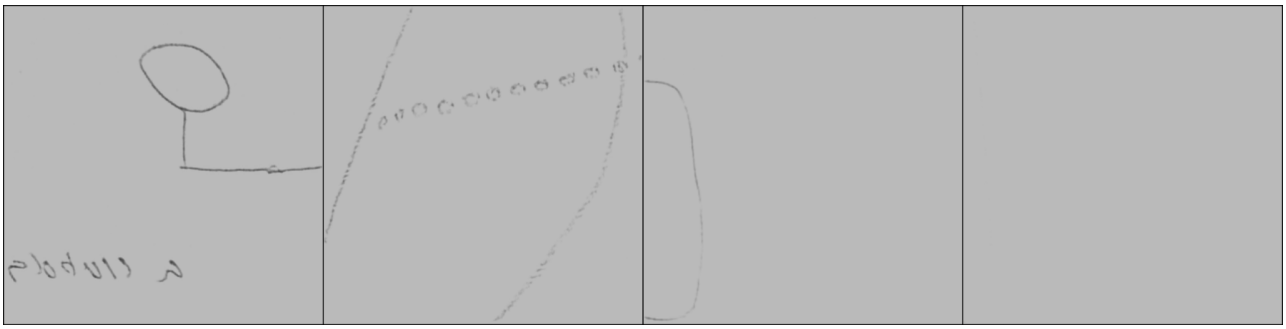


After many tries I found such combination of layers that gave pretty decent result:

```
        self.encoder = nn.Sequential(
            nn.Conv2d(3, 8, 3, 1, 1),
            nn.BatchNorm2d(8),
            nn.ReLU(),
            nn.Conv2d(8, 16, 3, 1, 1),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(16, 32, 3, 1, 1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.Conv2d(32, 32, 3, 1, 1),
            nn.BatchNorm2d(32),
            nn.ReLU()
            )
        self.decoder = nn.Sequential(
            nn.Conv2d(32, 32, 3, 1, 1),
            nn.ReLU(),
            nn.Conv2d(32, 16, 3, 1, 1),
            nn.ReLU(),
            nn.Upsample(scale_factor=2),
            nn.Conv2d(16, 8, 3, 1, 1),
            nn.ReLU(),
            nn.Conv2d(8, 1, 3, 1, 1)
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

As can be seen from the code, there is no activation function in the end of the model but, going forward, the loss function that will be used already has Sigmoid function in it so there is no need for it in the model code.

Source batch	
Label batch	
Validation output batch (3 epochs, L1 loss, SGD optimizer, learning rate = 0.01)	

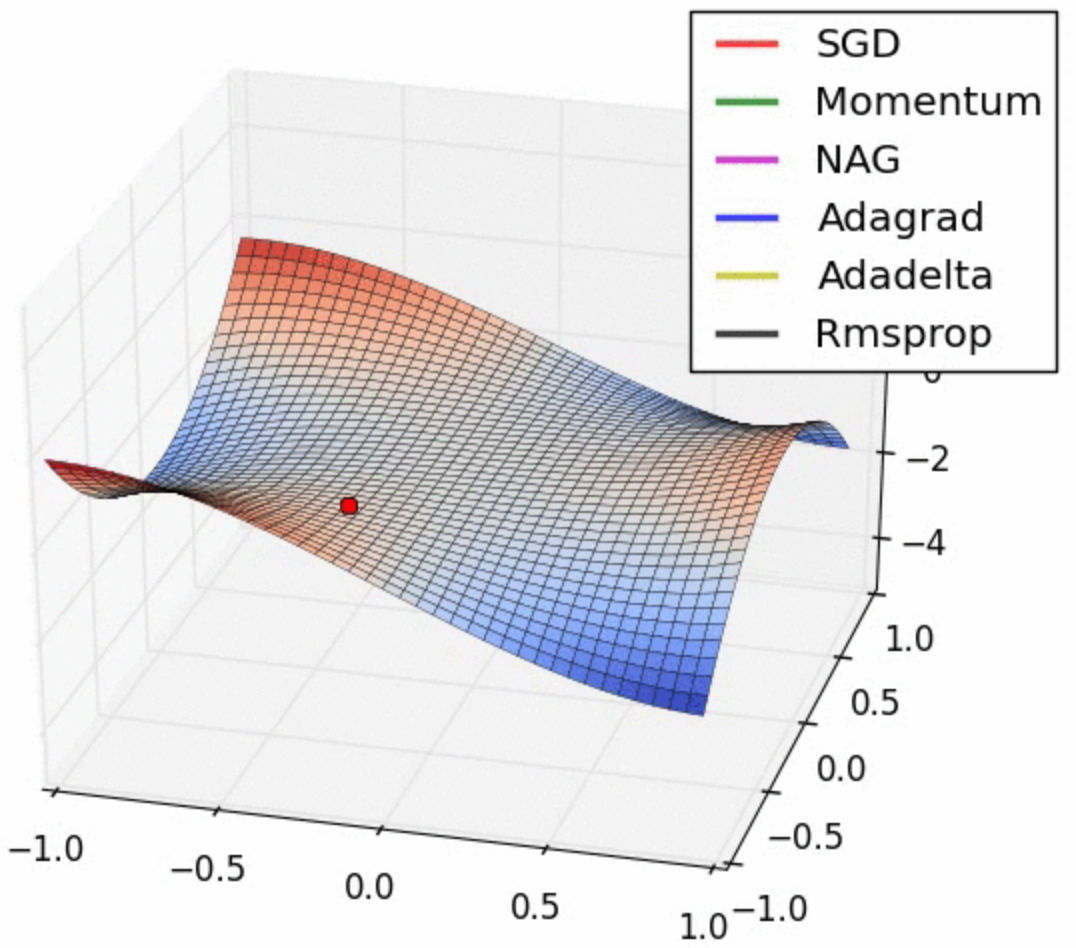
If, looking at the last picture, you would disagree, I would give you a little spoiler: after optimizer and loss tuning it will become much prettier.

Optimizer

I have chosen 3 optimizers tu try on my model:

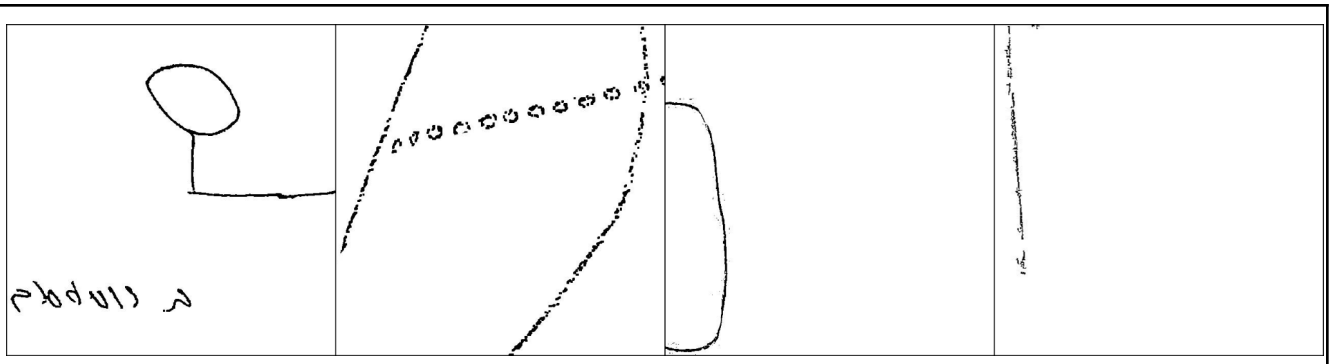
- SGD:
    - With momentum
    - Without momentum
- Adam
- RMSProp

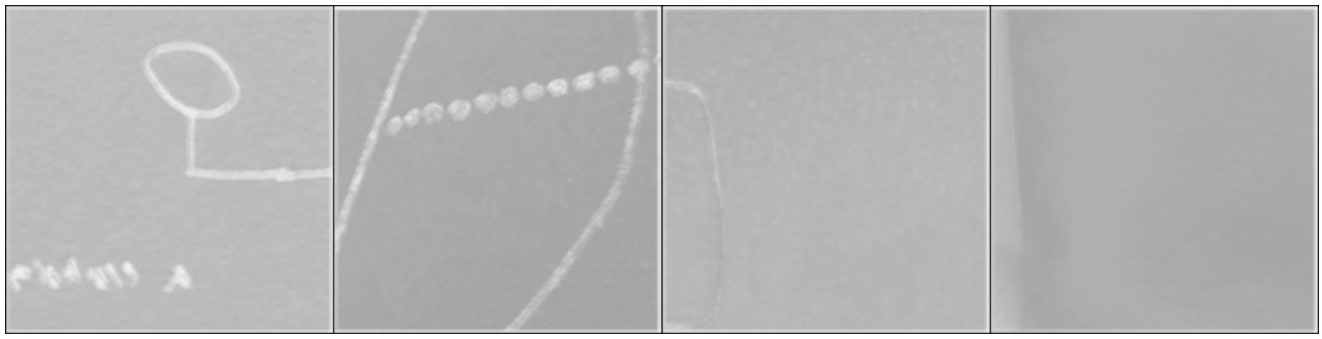
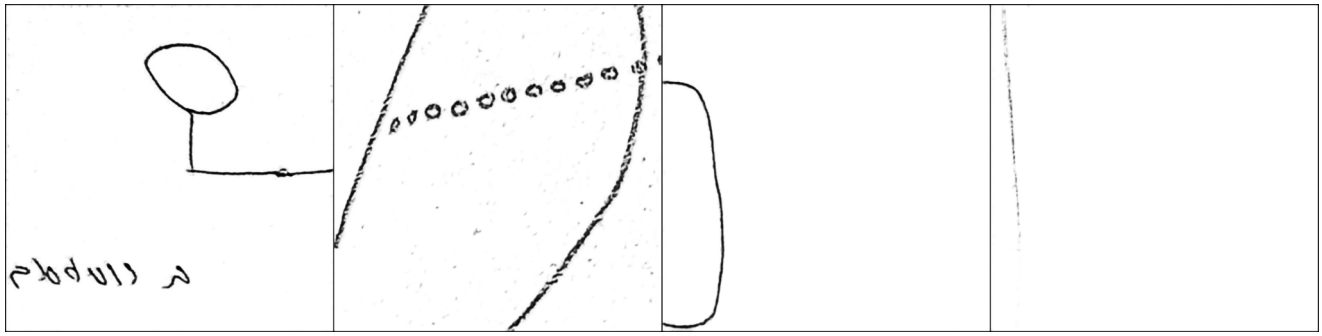
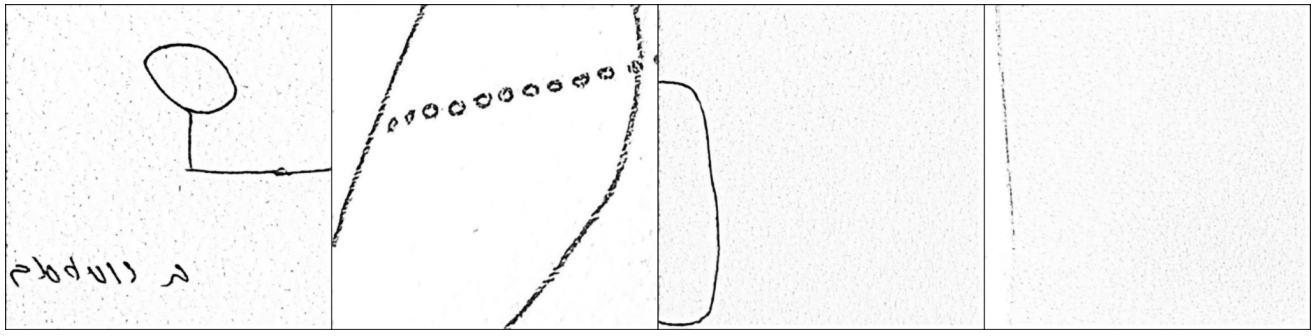
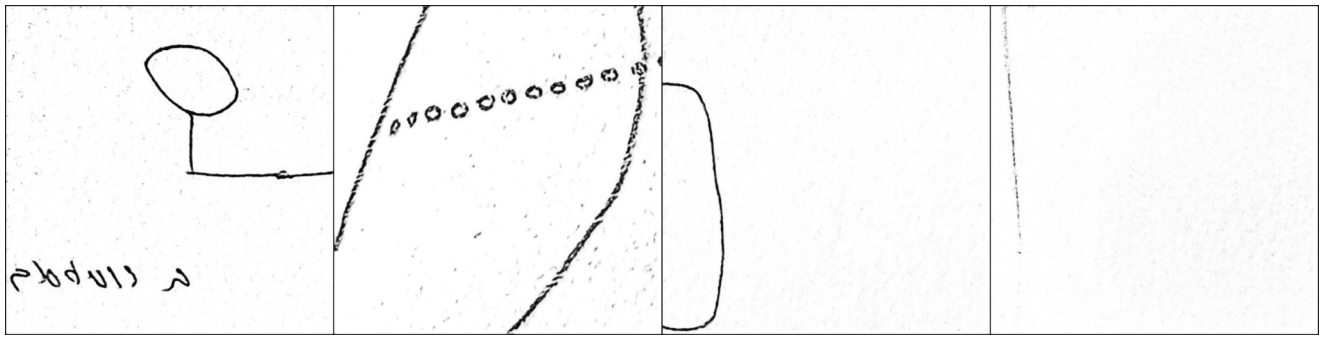
We can approximately estimate their speed just looking at the picture below but their efficiency in our specific problem can be proved only after practical tests.

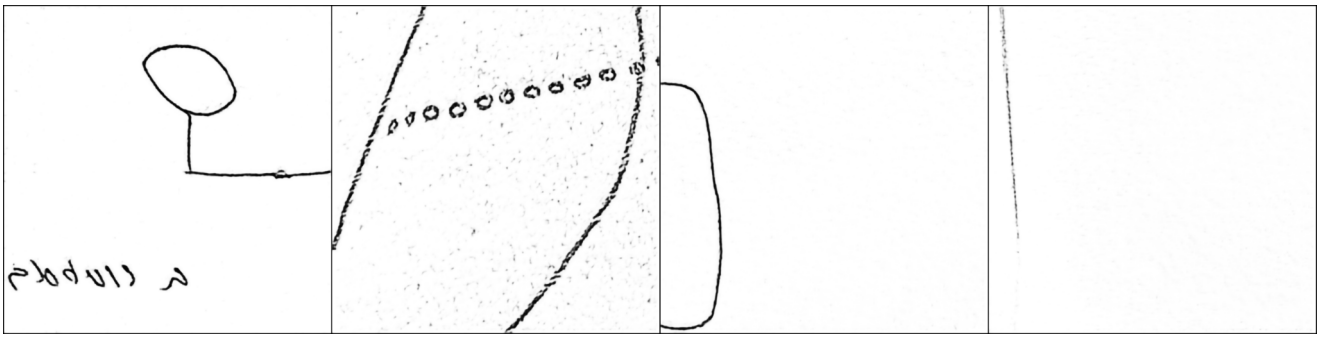
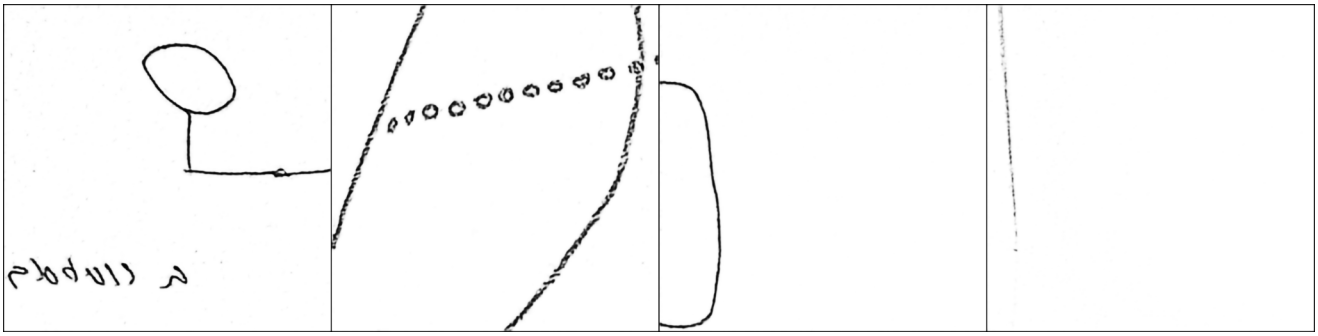
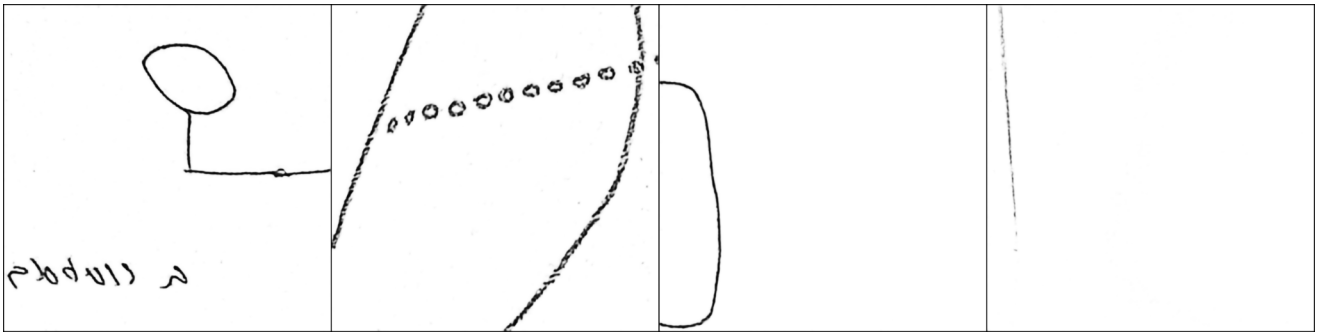
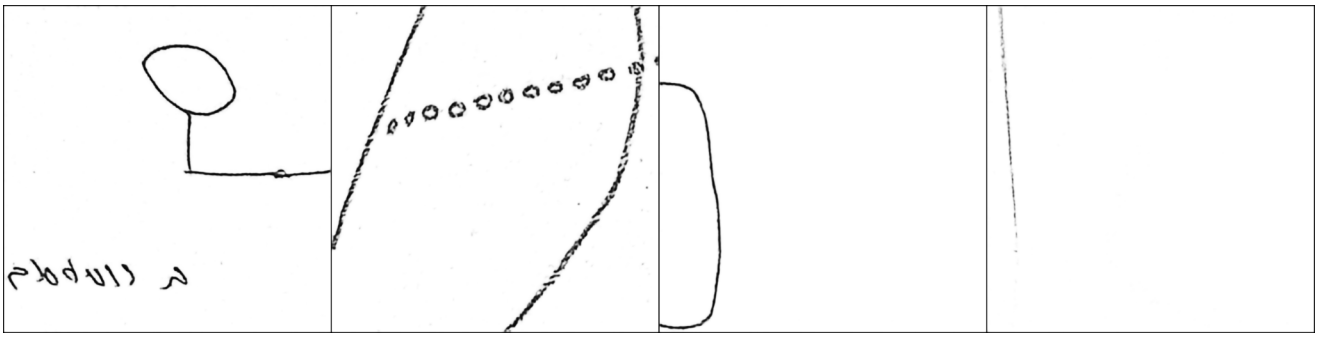


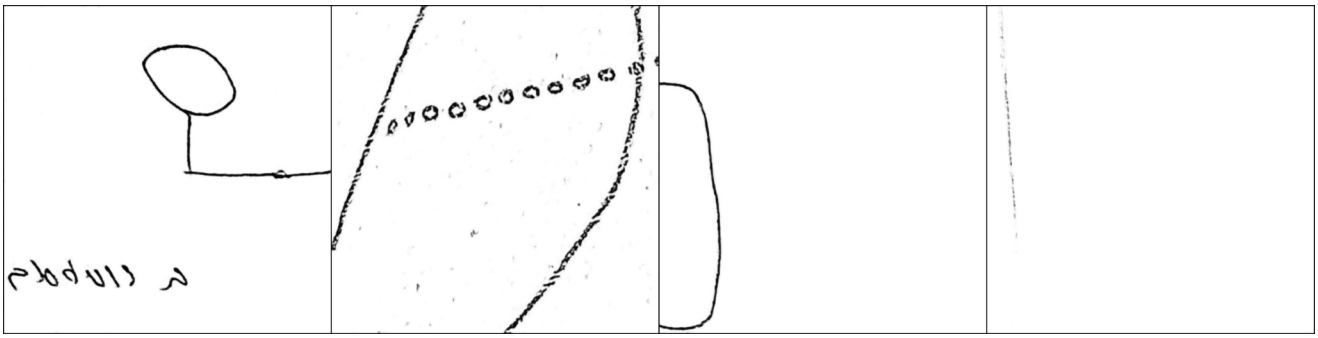
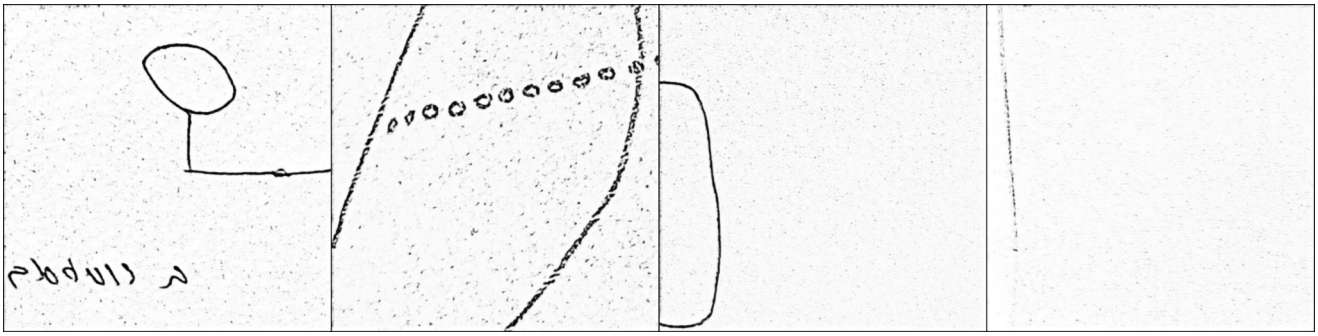
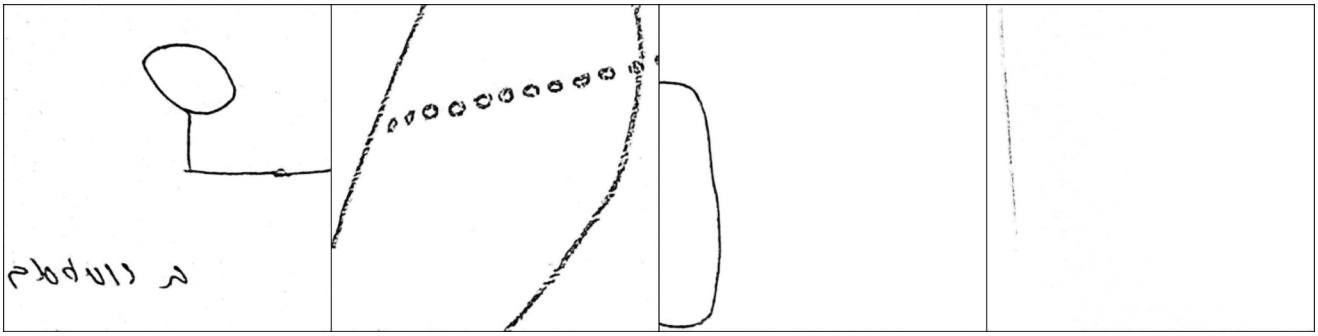
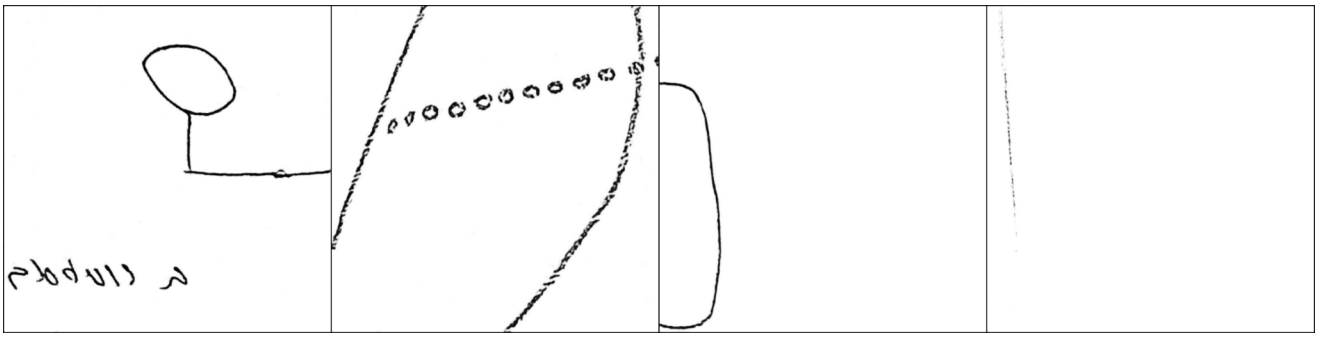
In the table below we can compare model's outputs between each other, analyzing such parameters as noisiness, lines thickness and tracing, and presence of unwanted artifacts.

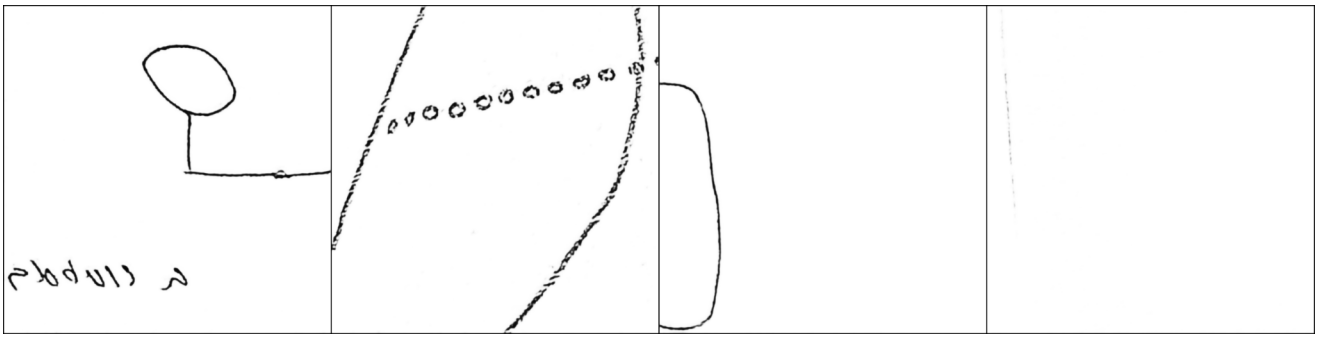
** 3 epochs, BCEWithLogits Loss without weights:*

Label	
Optimizer	Validation output

SGD (lr=0.0001)	
SGD (lr=0.1)	
SGD (lr=0.1, momentum=0.9)	
SGD (lr=0.1, momentum=0.7)	

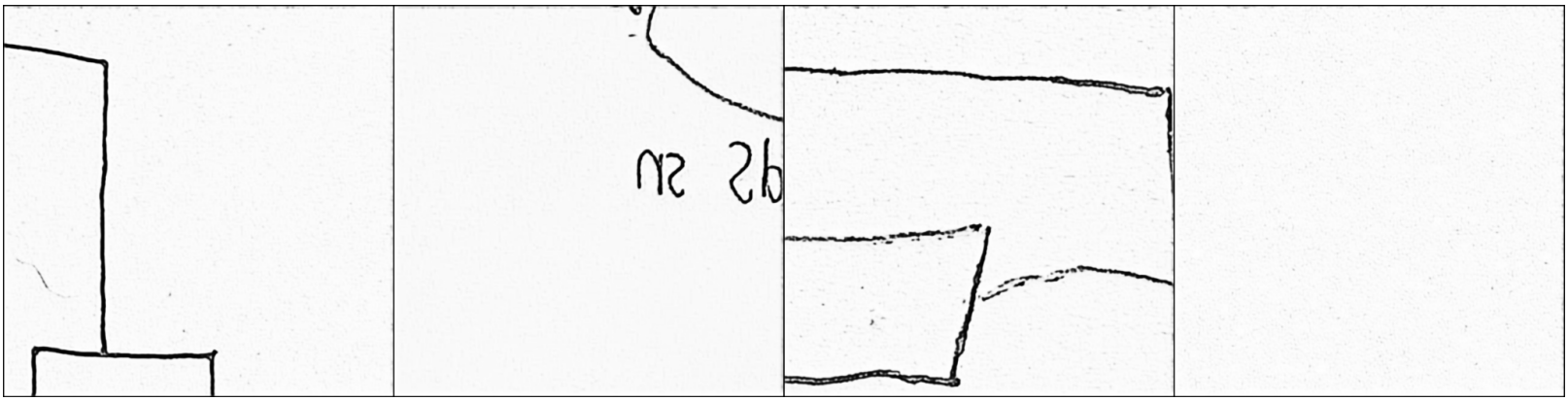
SGD (lr=0.1, momentum=0.5)	
Adam (lr=0.0001, betas=(0.9, 0.999))	
Adam (lr=0.0001, betas=(0.7, 0.999))	
Adam (lr=0.0001, betas=(0.5, 0.999))	

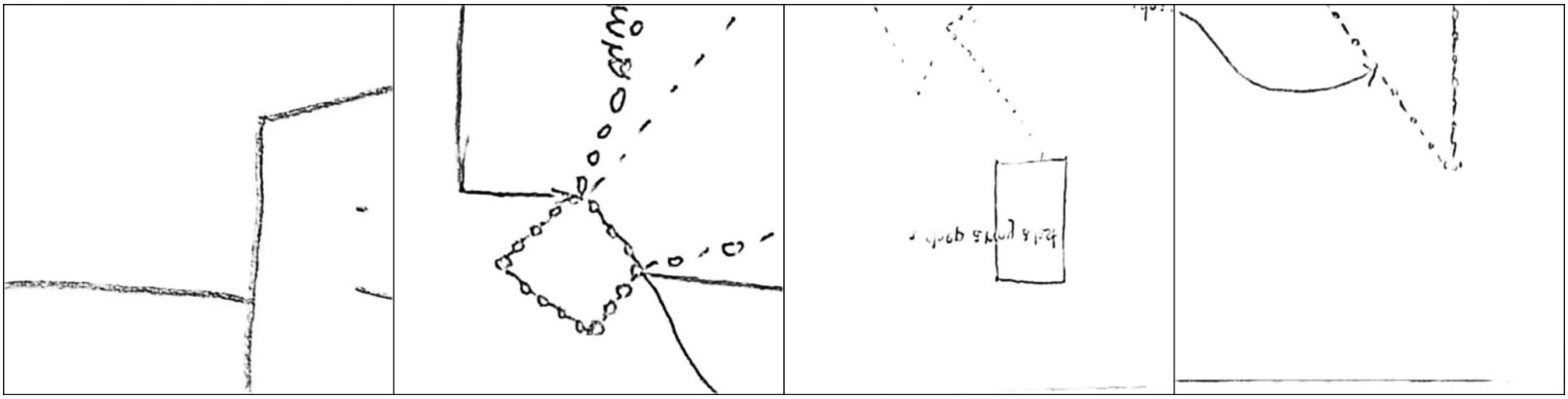
RMSProp (lr=0.0001, decay=1e-10, momentum=0.9)	
RMSProp (lr=0.0001, decay=1e-8, momentum=0.9)	
RMSProp (lr=0.0001, decay=1e-6, momentum=0.9)	
RMSProp (lr=0.0001, decay=1e-6, momentum=0.7)	

RMSProp (lr=0.0001, decay=1e-4, momentum=0.9)	

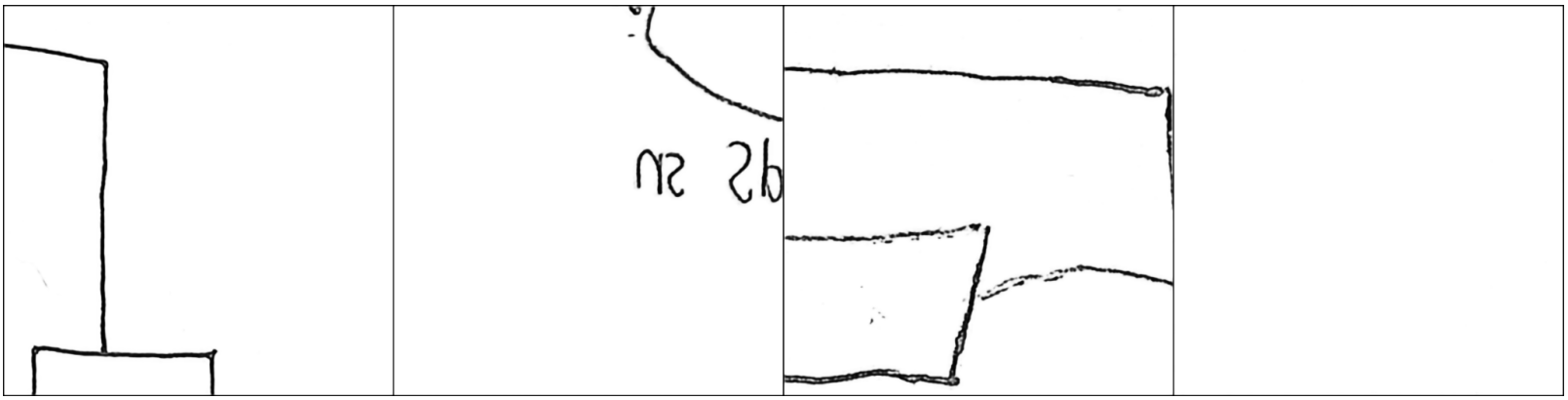
We can conclude from the table above that SGD optimizer by its nature needs a high learning rate. I will solve this problem further using a scheduler but we can see that even with a high learning rate SGD provides an output with a relatively high noise level.
Using SGD with momentum decreases an amount of noise but the best result looks worse than the best one with simple SGD.
As for Adam optimizer, it brings us not as great results at first, but if we tune its parameters, we get a very good combination of noise level and line thickness among all optimizers.
Finally, RMSProp optimizer with the last parameter combination gives us the very best result with almost no noise line on the 4th picture and should be our first choice on the first sight.

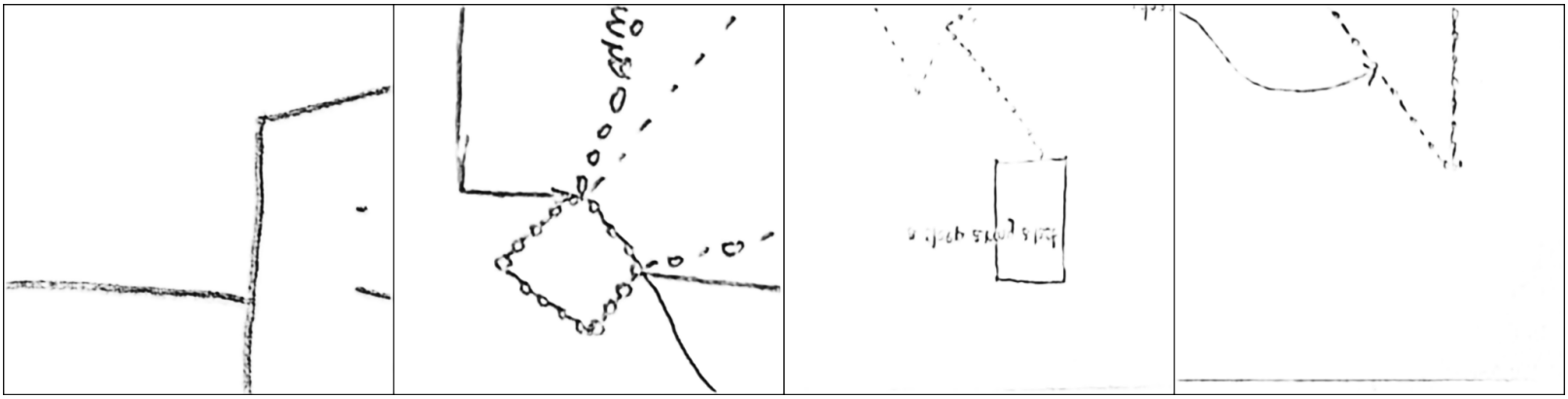
But, if train model with all the same parameters but with different random seeds we can notice that on some runs noise appears and on some - lines become more subtle and poorly visible:





In comparison, using Adam optimizer on the same random seeds gives these results:





Adam also gives us poorly visible lines on other runs but they are still a little better than RMSProp, and we almost do not get any noise.

Conclusions:

Based on this, I decided to use Adam optimizer with betas of 0.7 and 0.999 in the final model version because of the greater stability of results.

Loss Function

As for loss function, I used such as those:

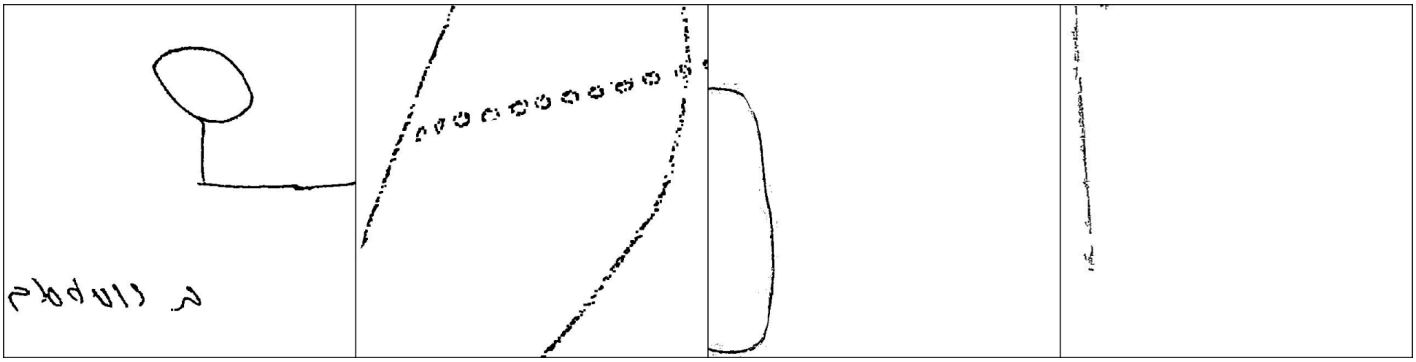
- L1 (MAE) Loss
- BCEWithLogits Loss:
    - With weights
    - Without weights
- Perceptual (VGG) Loss
- Combined Loss:
    - VGG + L1 Loss
    - VGG + MSE Loss
    - VGG + BCE Loss
    - VGG + BCEWithLogits Loss (with weights)

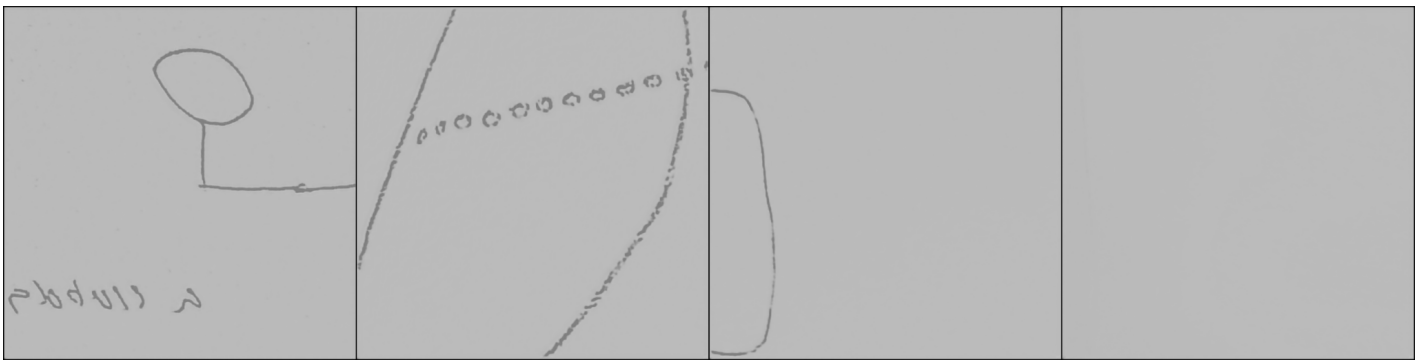
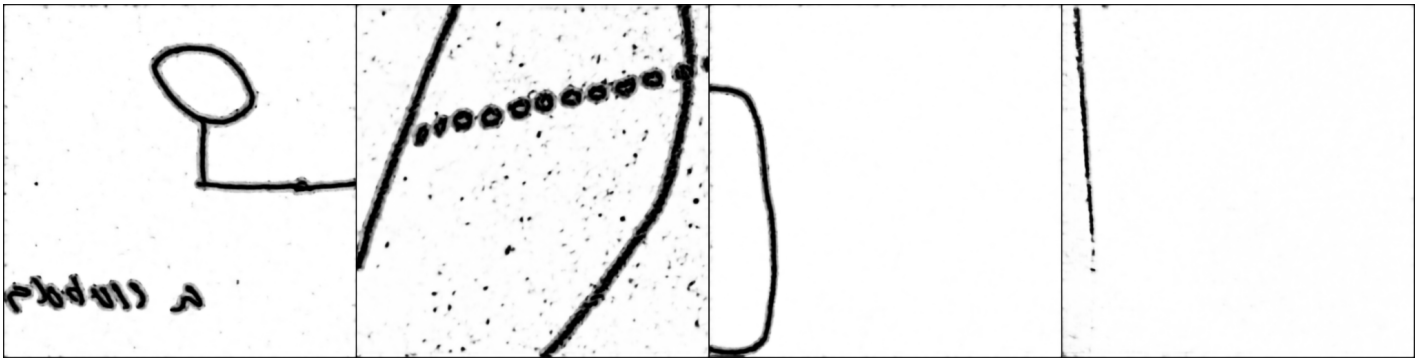
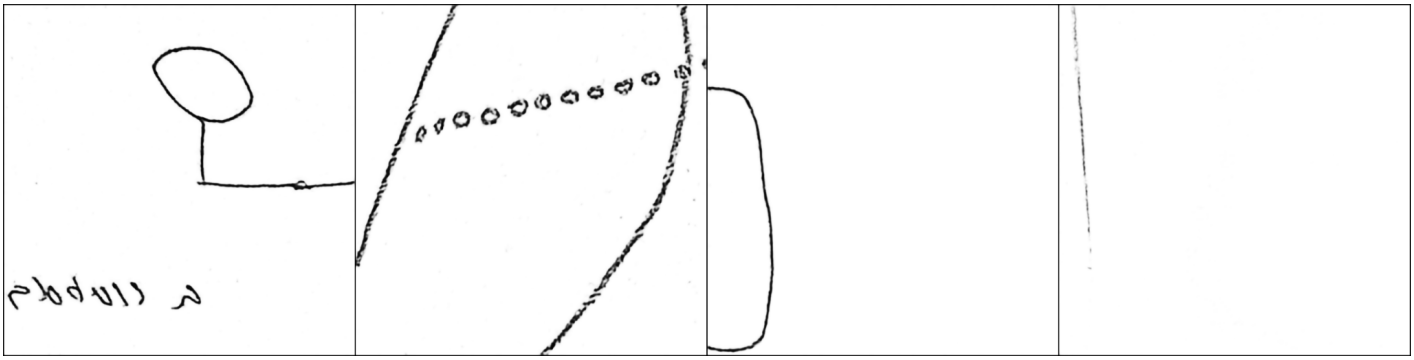
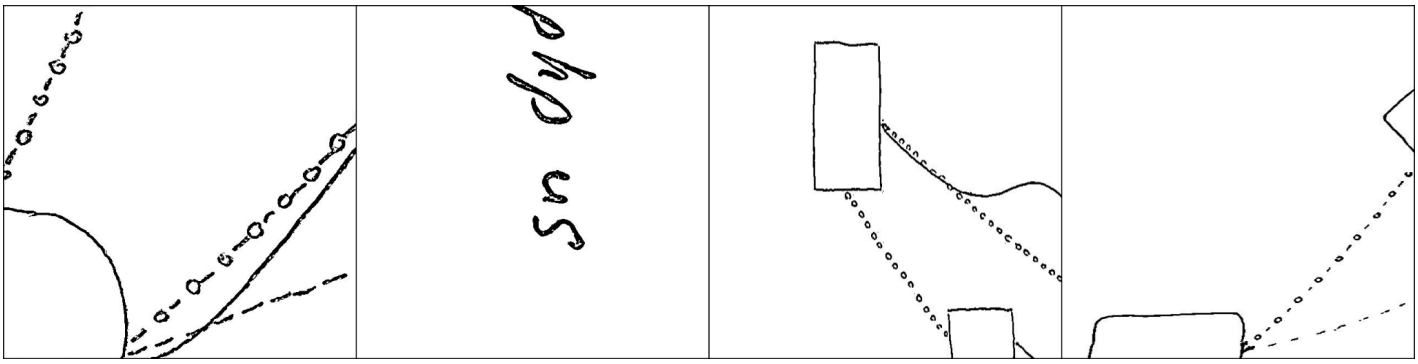
Our task specifics is that loss is calculated pixel-wise in a greyscale. That is why there is no loss graphics with dependence on epochs: it is very hard to see the difference between 2 graphics even when we have much noise in one output and clean images in another.
Apart from the obvious MAE loss I tried 2 versions of the BCEWithLogits loss: with and without black pixel weight (0.962%).
BCEWithLogits loss is the same as usual Binary Cross-Entropy loss but with a Sigmoid layer:

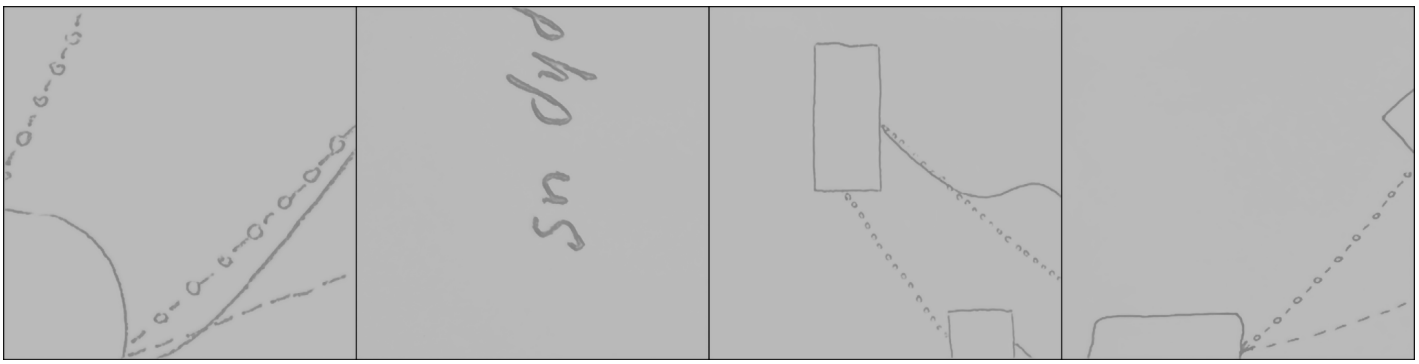
$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log \sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n)) \right],$$

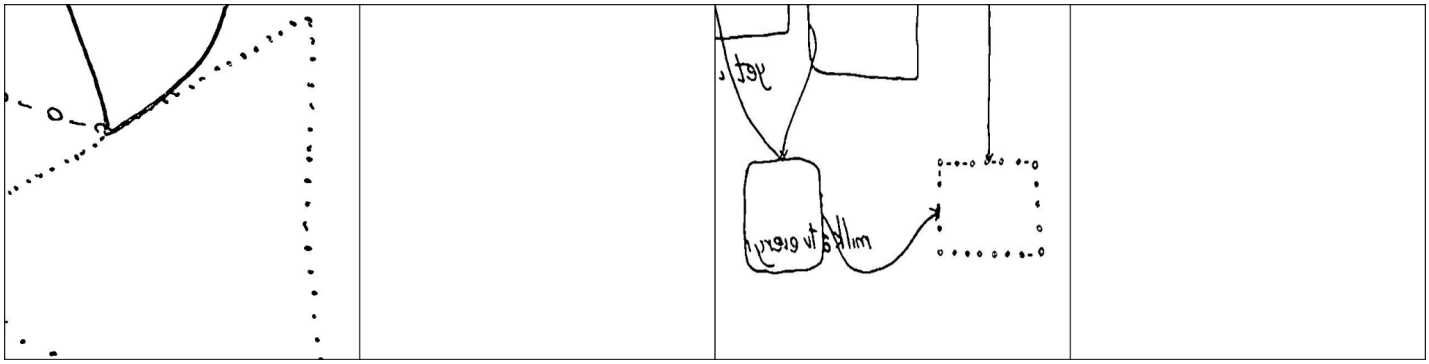
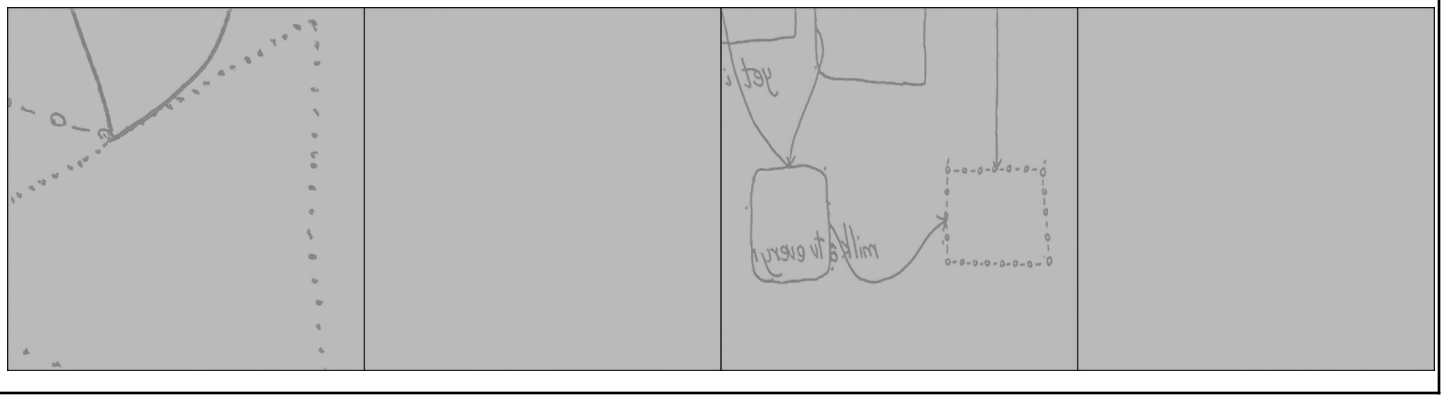
Where N is the batch size.
Also I used the VGG or Perceptual loss which compares not only model's output but also intermediate outputs with the ones that VGG16 model would produce with our source images.
In addition, I combined VGG loss with some simpler losses.

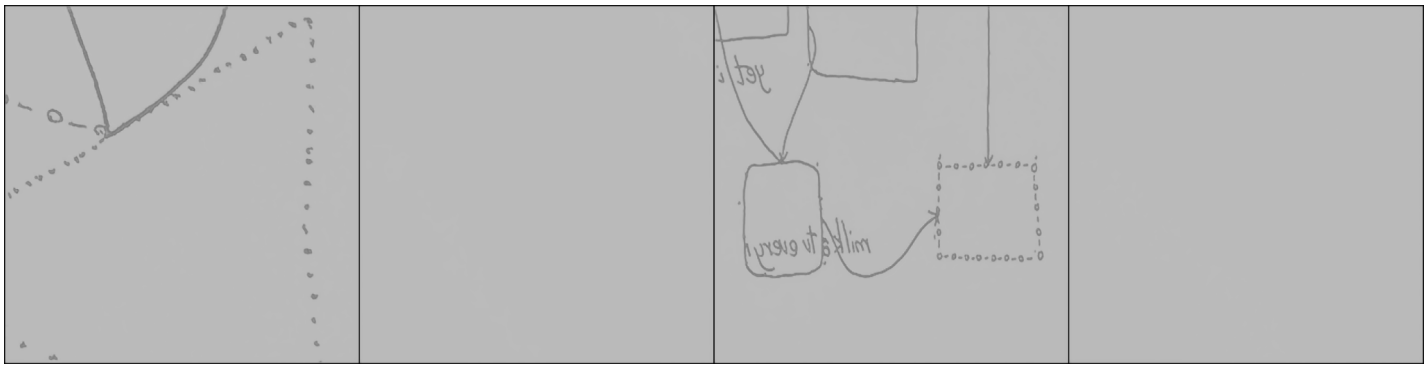
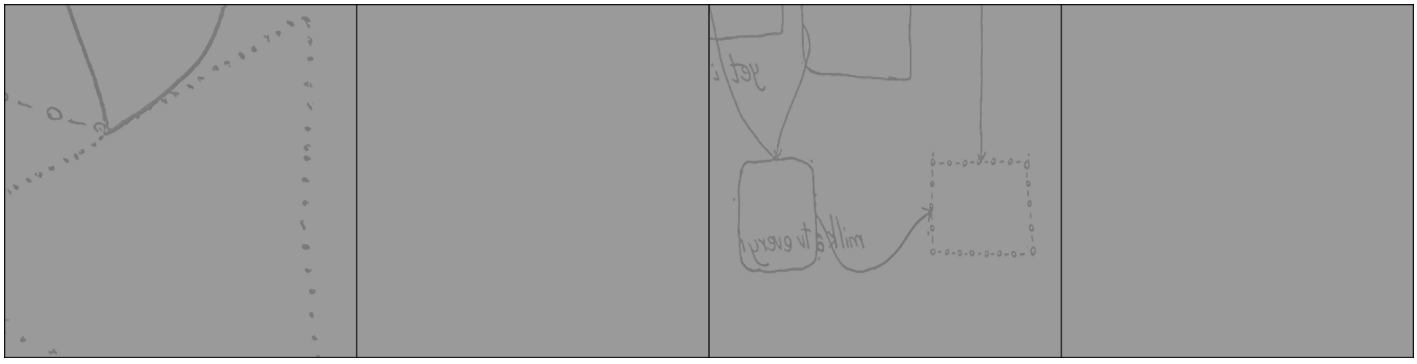
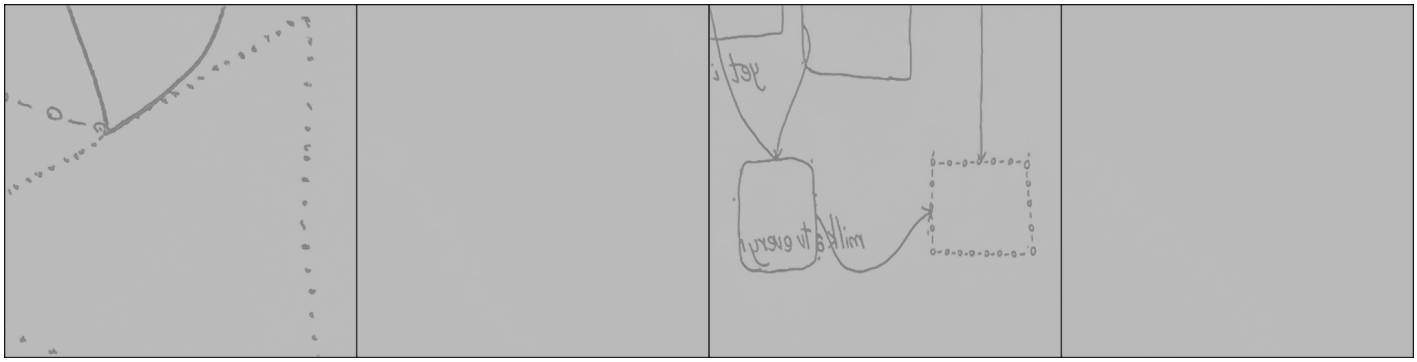
In the table below Adam optimizer with learning rate 0.0001, betas 0.7 and 0.999 was used. All these losses showed such results:

Label	
Loss	Validation output

L1	
BCEWith Logits With weights	
BCEWith Logits Without weights	
Perceptual	

Label	
Loss	Validation output
VGG + L1	

VGG + MSE	
VGG + BCE	
VGG + BCEWith Logits (without weights)	

It is interesting that BCEWithLogits loss in general worked better without black pixel weights: lines were becoming much more blurred and bold.
Perceptual loss also did not work well, even though I tried many things with it (it is not shown in the table but I also used to change the loss that used in the VGG loss itself from L1 to the ones from the list above but it also produced nothing.

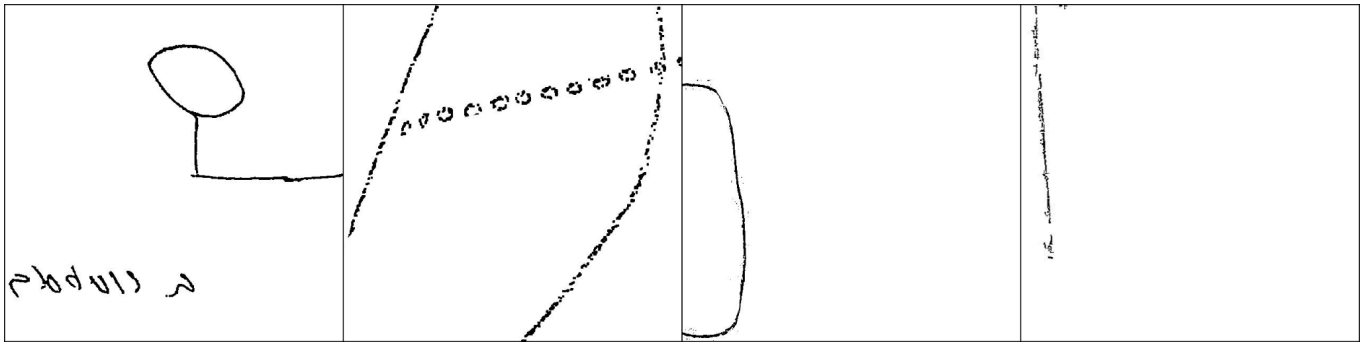
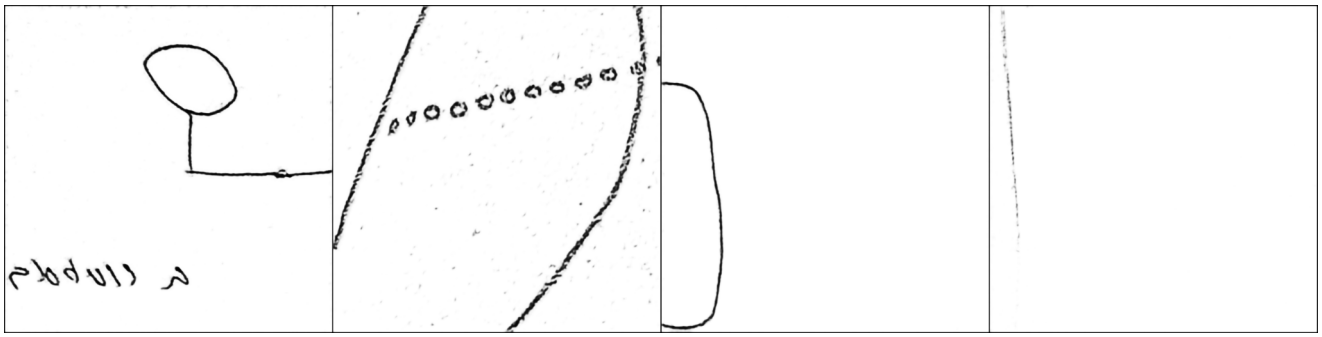
Conclusions:
I chose BCEWithLogits loss without weights to use with my model since it provided the best output results.

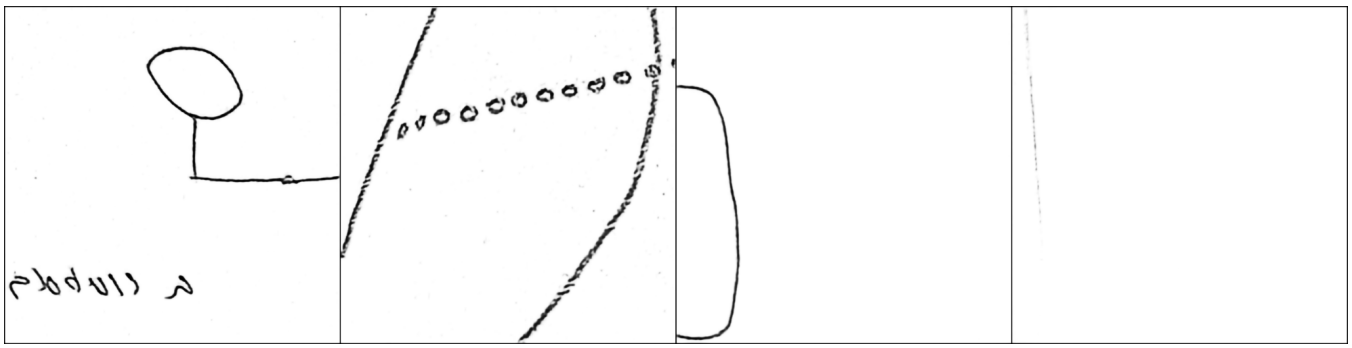
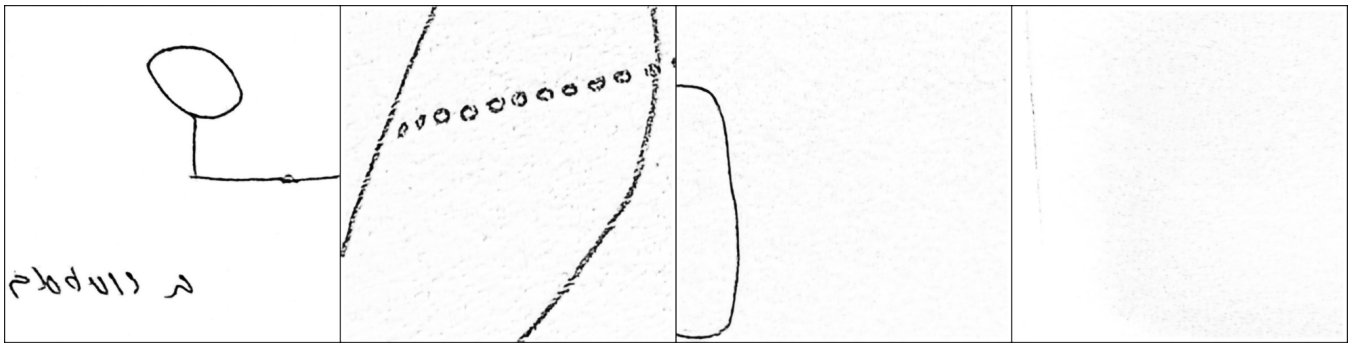
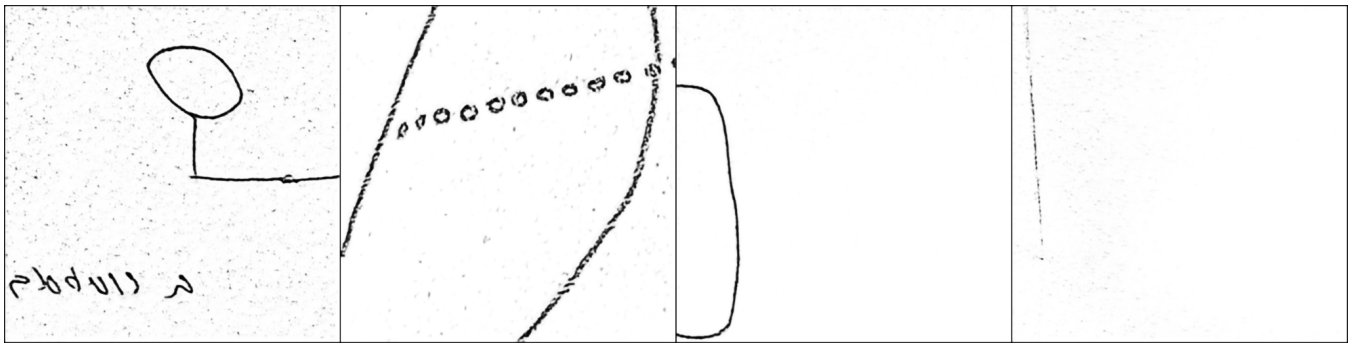
Schedulers

Schedulers are needed to tune the learning rate not to be too high or low.
In the table below are showed experiment results using a SGD optimizer since
I used such of them:

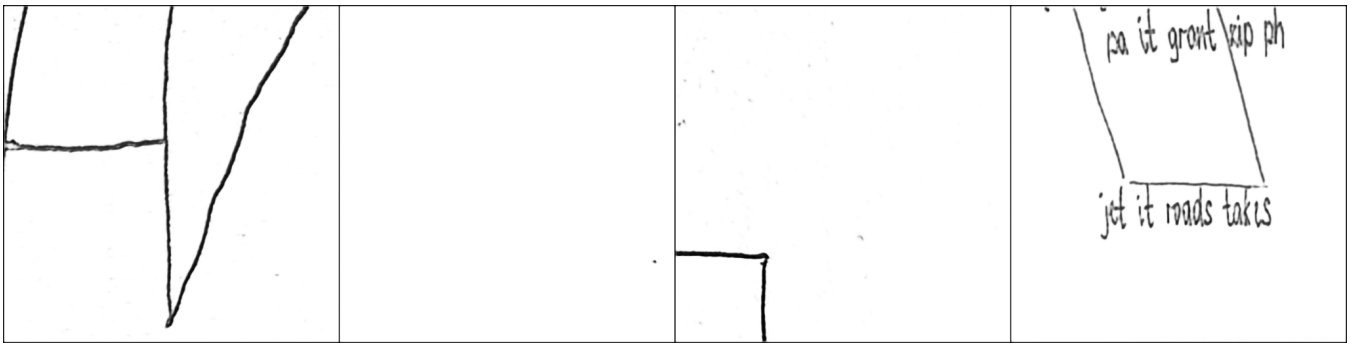
- StepLR: multiplies initial LR by some value each N steps. In my specific case I multiply initial LR=0.1 by 0.1 each 1000 steps (twice an epoch)
- CyclicLR: cycles the LR between two boundaries (0.0001, 0.1) with a constant frequency (twice an epoch) increasing half of a cycle, no amplitude scaling was used
- CosineAnnealingWarmRestarts: sets the LR using a cosine annealing schedule, with minimum value of LR=0, maximum LR=0.1 and 1000 steps between warm restarts (half an epoch)
- ReduceLROnPlateau: reduces the LR by a factor of 10 when the quantity metrics monitored has stopped decreasing for *patience* amount of epochs

3 epochs for all trains, BCEWithLogits Loss without weights

Label	
Scheduler	Validation output
Without scheduler, SGD optimizer, lr=0.1	

StepLR (step=1000, gamma=0.1)	
CyclicLR (base_lr=0.0001, max_lr=0.1, step=1000)	
CosineAnnealingWarmRestarts (T_0=1000)	

SGD (8 epochs, lr=0.001, momentum=0.7)

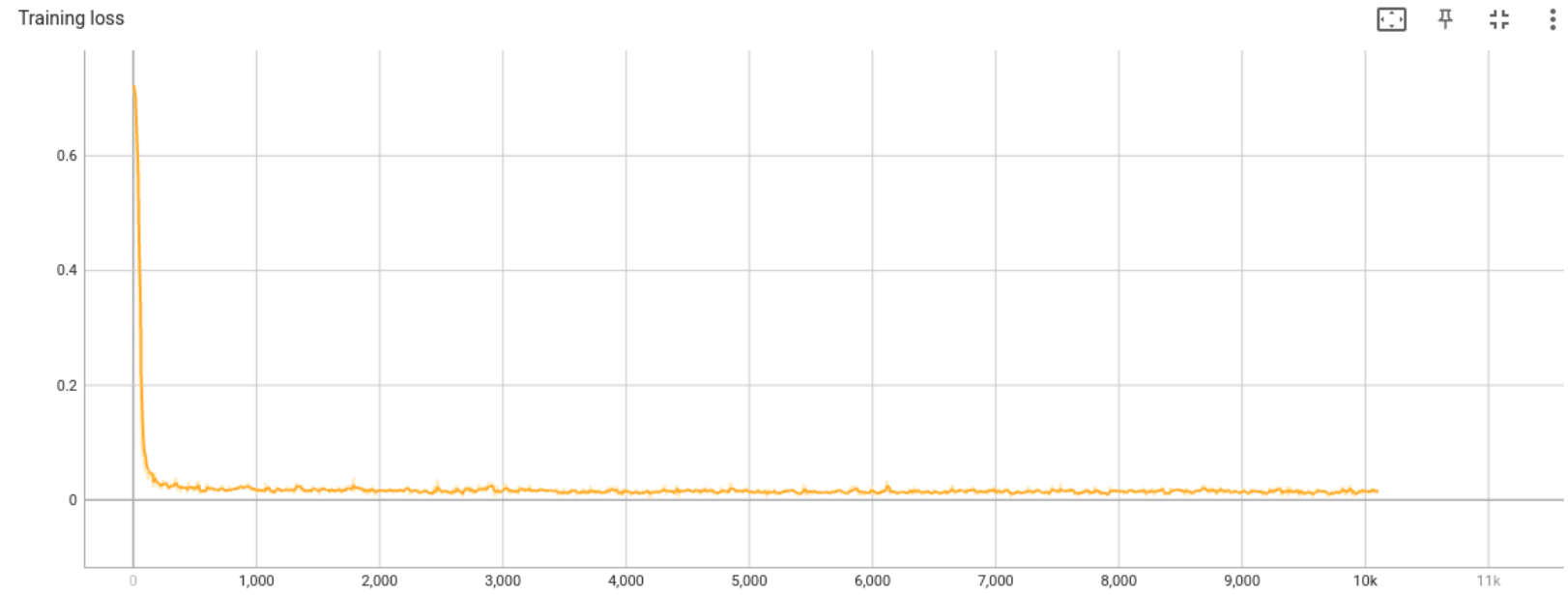
ReduceLROnPlateau (patience=2)	

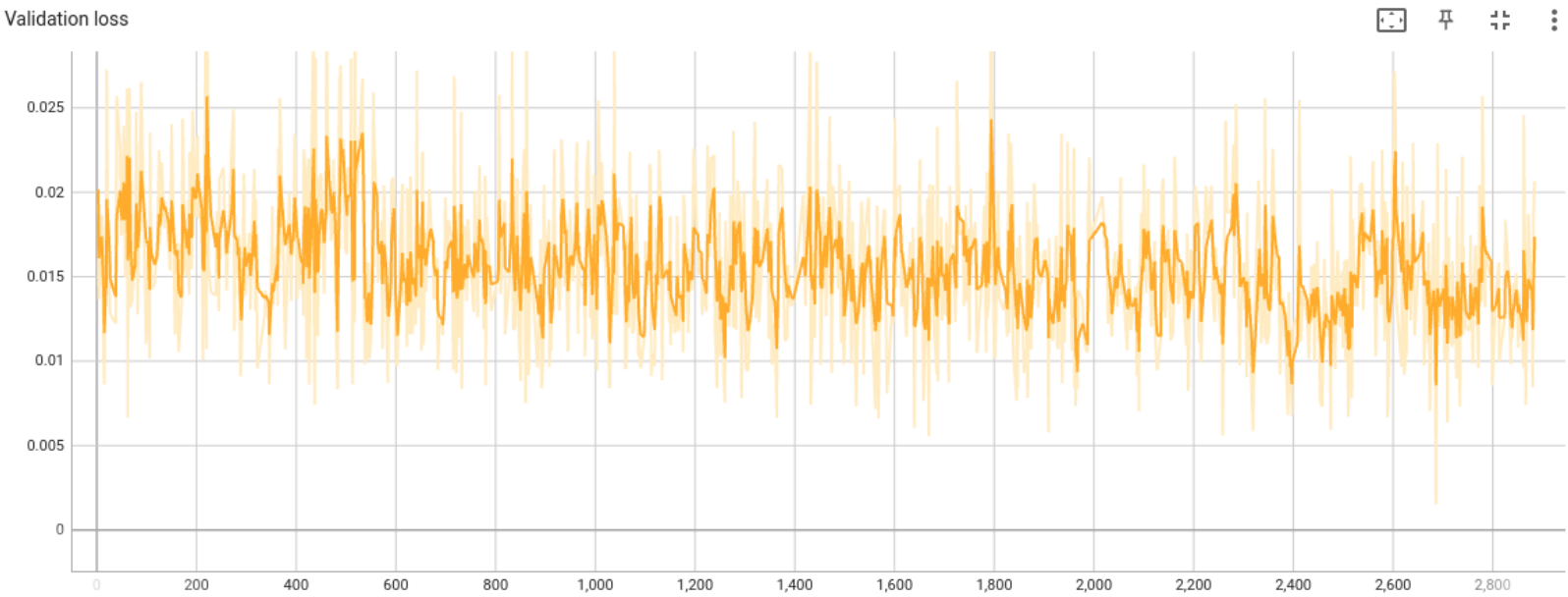
For the last scheduler I trained the model for 8 epochs and used a momentum because of the scheduler's peculiarity: it needs many epochs and not relatively low initial learning rate so if I used simple SGD with a low LR, even with this scheduler results would be poor. That is why I had to use the most efficient SGD parameters combination.

Conclusions:
We can see from the table that the simplest StepLR scheduler has shown us the best result among all of them, cleaning the most of the noise and, in fact, providing even better result than the target one on the 4th picture which is a noise actually and should not be on the output image.
ReduceLROnPlateu did a good job too but it worked with a much better baseline of momentum SGD with much more epochs.

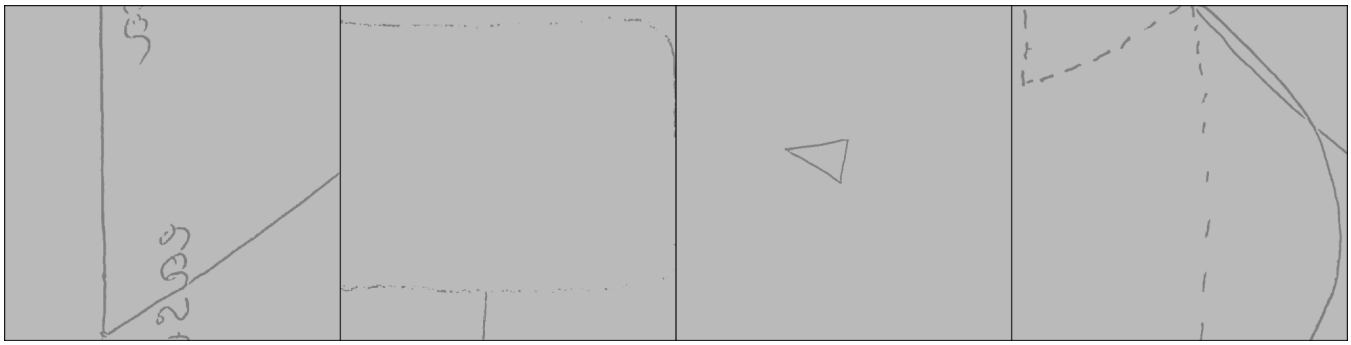
Loss plots

As it is said above, the difference between bad and good models counts literally in the bunch of pixels so the loss plots will not be so representative.
I will provide plots below to show that I got the experience working with the TensorBoard.
On all plots Y scale is the loss value, X scale - iteration number (2021 per epoch for training, 577 per epoch for validation), smoothing=0.6 in the TensorBoard.
For example, loss plots for the final model look like this:

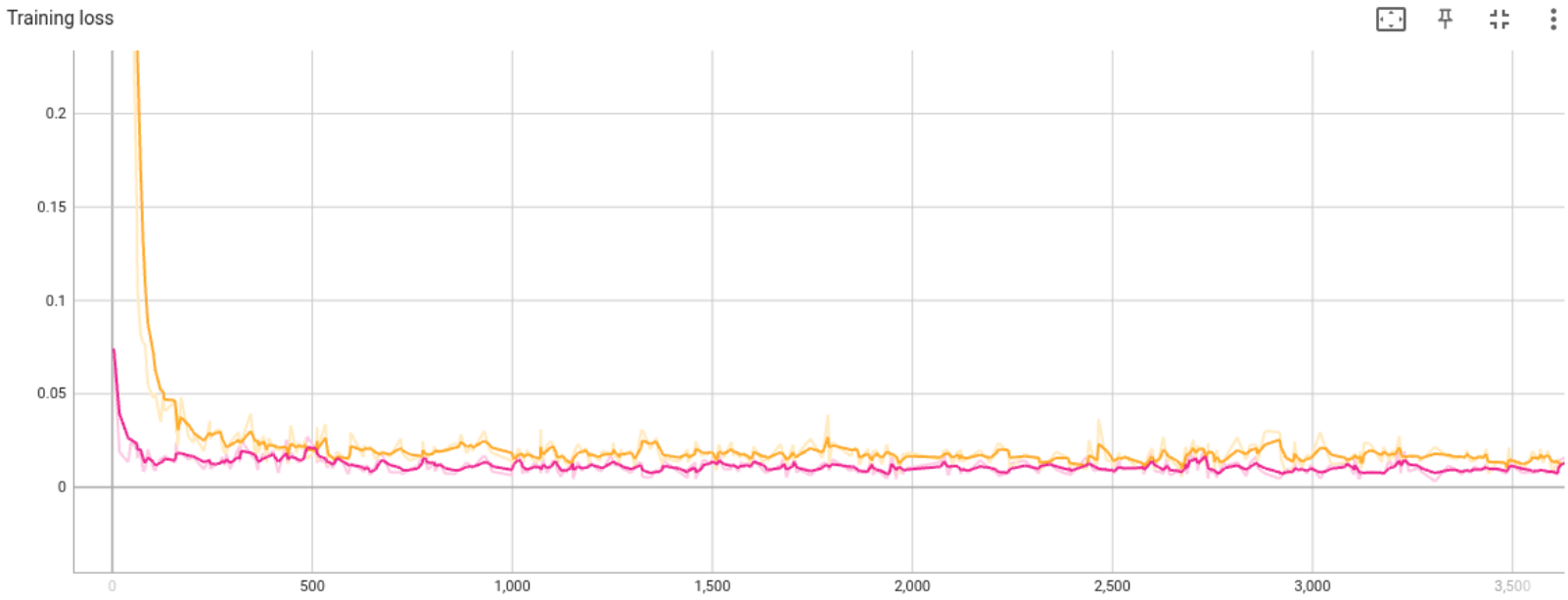




For comparison's sake: below are the plots with the final model from above graphs with the model that provided a grey-background pictures like this:

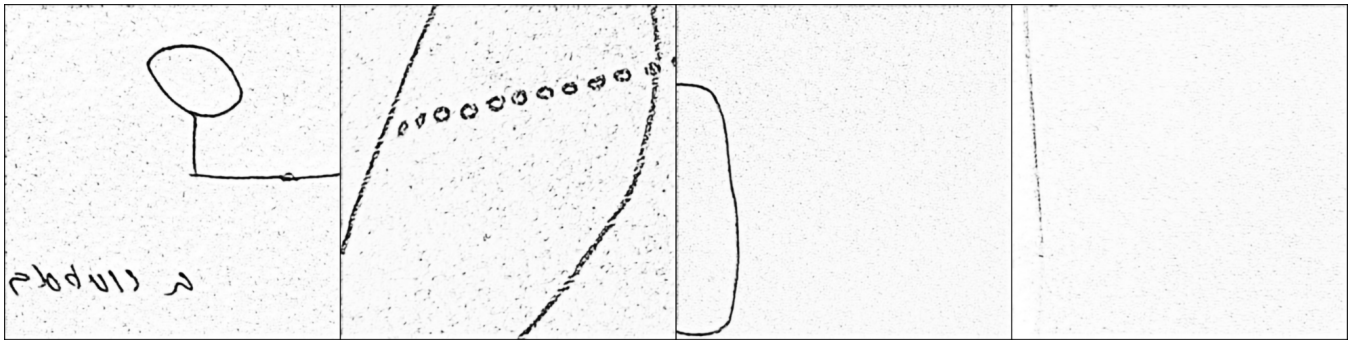


("bad" model's loss is pink-colored)

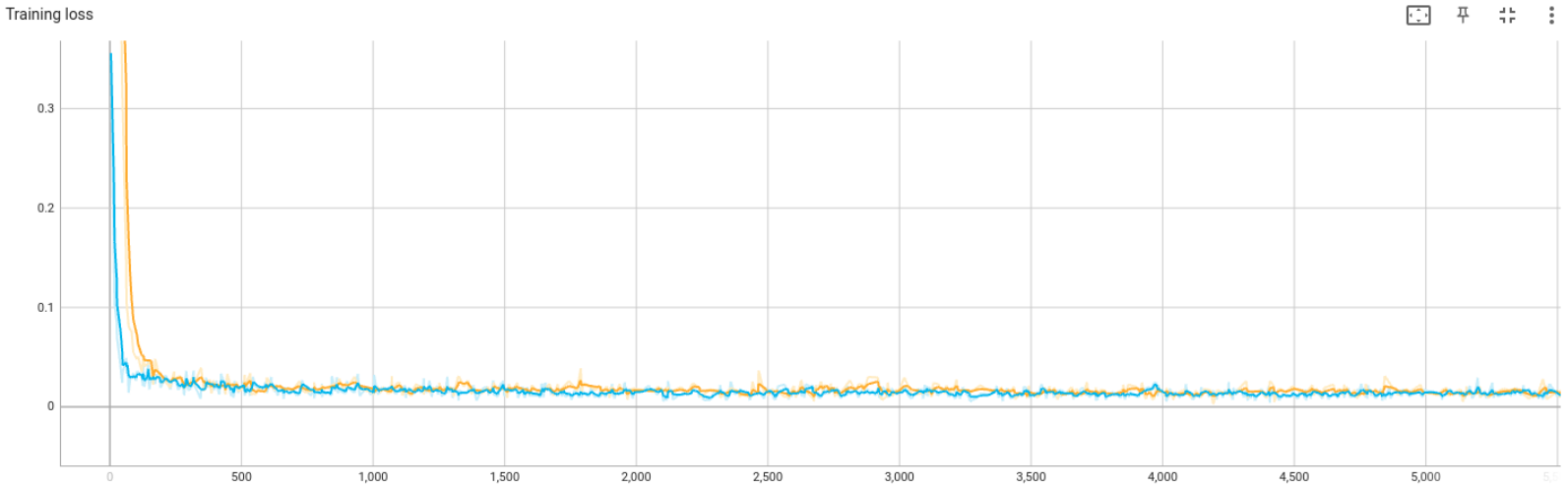


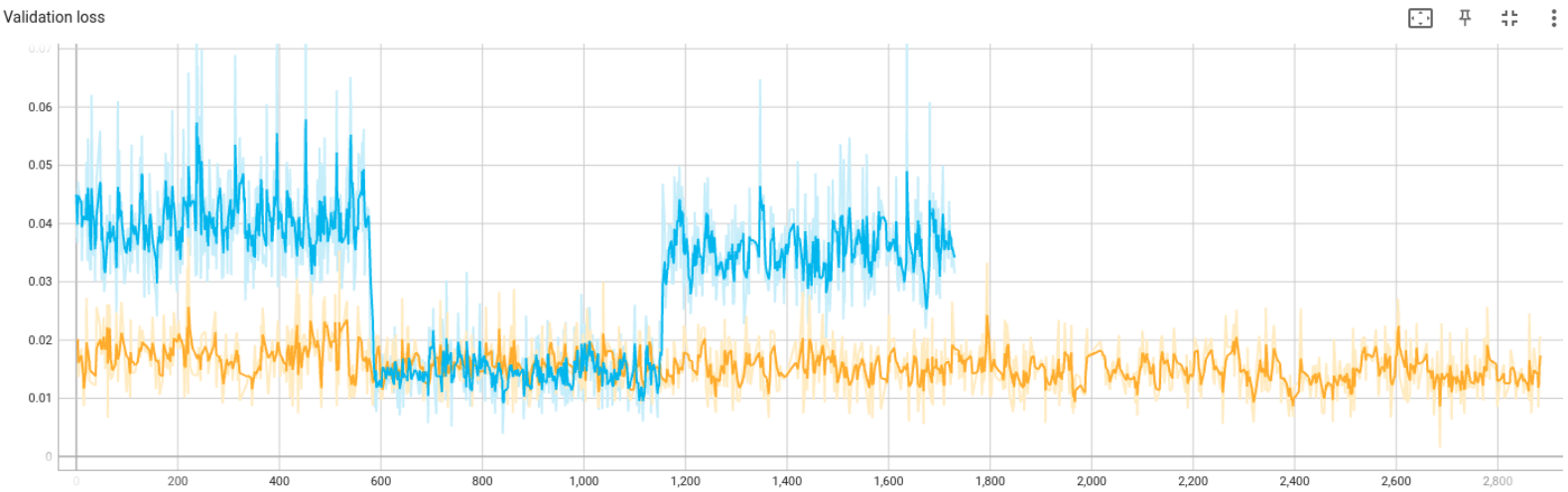
As to models that work well but leave the noise or draw subtle lines, it is almost impossible to find the difference between them and good version of the model on the train loss plots but losses differ much more on the validation plots:

Model's output



Loss plots, “bad” model’s color is blue





Conclusions:

In our specific case, loss plots are not so indicative because the difference between good and bad results could be measured just in 1% of all pixels on the image.

That is why it is better to check outputs after each model run manually since the human eye can notice little differences between images better than the loss function that is used in our task.

Speed/Memory usage

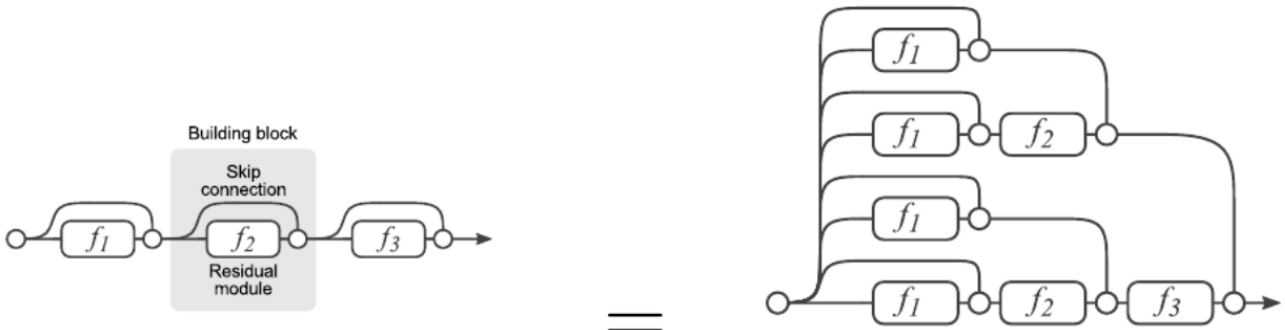
After time and video memory usage for 1000 pictures on inference measuring I got these results:

Model	Inference for 1000 images on GPU, seconds	Inference for 1000 images on CPU, seconds	Video memory usage on GPU, MiB	Video memory usage on CPU, MiB
Baseline (U-Net)	38.480303	1328.83265	2165 MiB	1322 MiB
Solution	6.945951	66.545736	1327 MiB	1176 MiB

As we can see from above, my custom model is 5.5 times faster than standard U-Net on GPU and 20 times faster while using CPU.
Also, U-Net model uses 63% more video memory while running on GPU and 12% more while running on CPU.

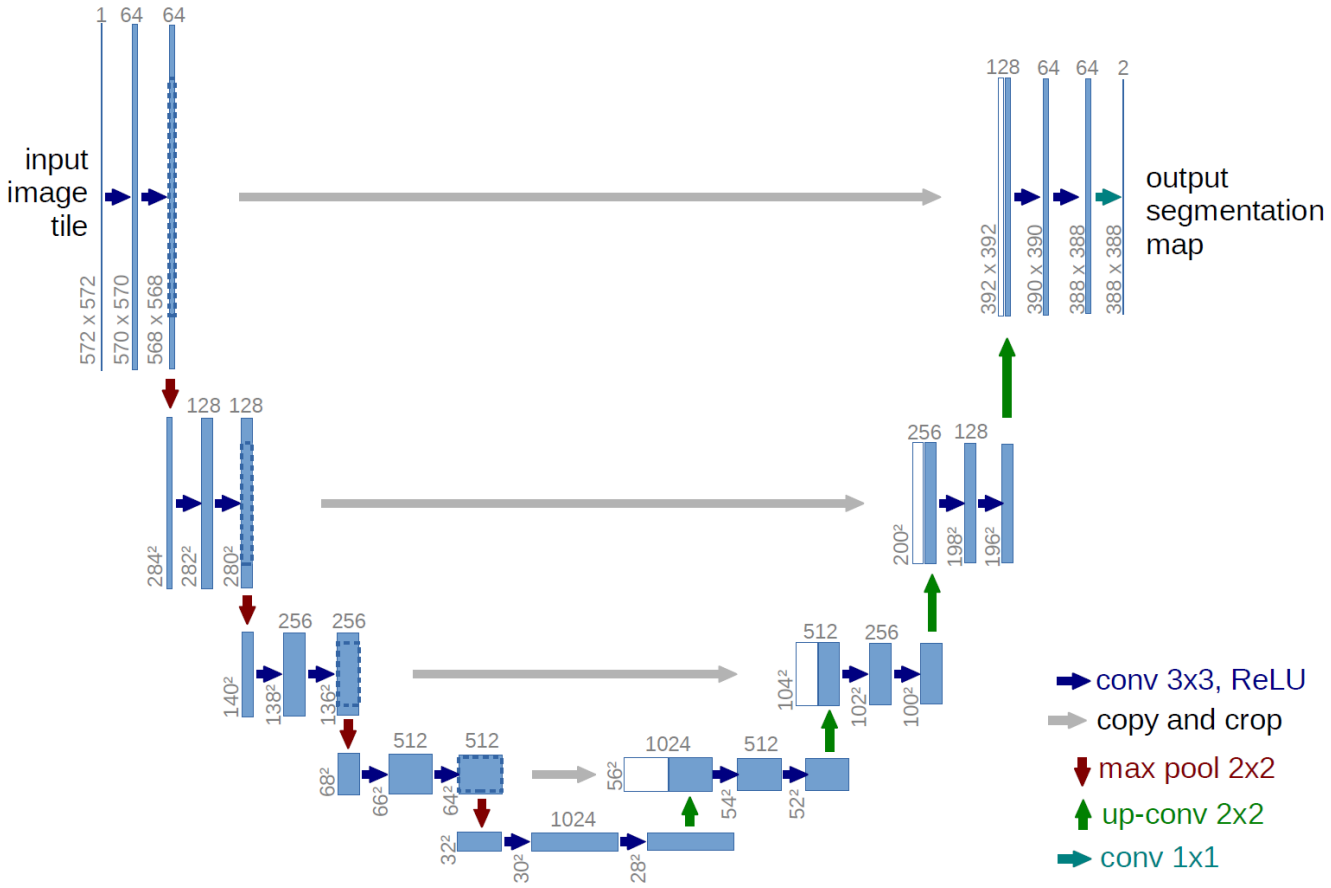
Skip connection

Skipped connections is one of the U-Net model features.
The main difference between the model without them is that we write model as several blocks that contain Conv2d, BatchNorm2d, ReLU and MaxPool2d layers in encoder and Conv2d or ConvTranspose2d with ReLU layers in decoder. Then we concatenate or sum up the output of the first block in the decoder with the corresponding output from the encoder.
We can see skipped or residual connections on the U-Net architecture scheme as grey arrows.



(a) Conventional 3-block residual network

(b) Unraveled view of (a)



In code skipped connections model looks like this:

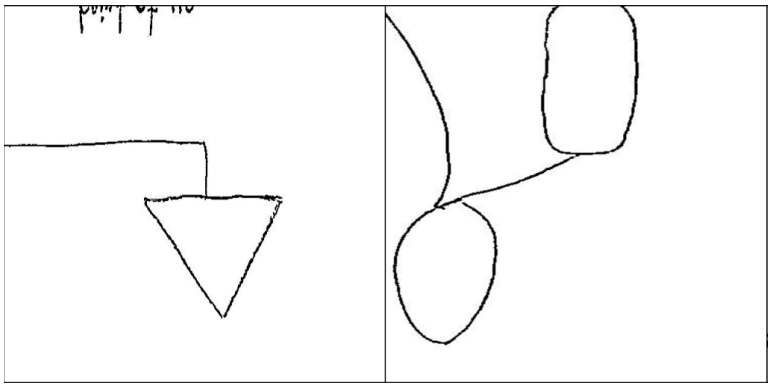
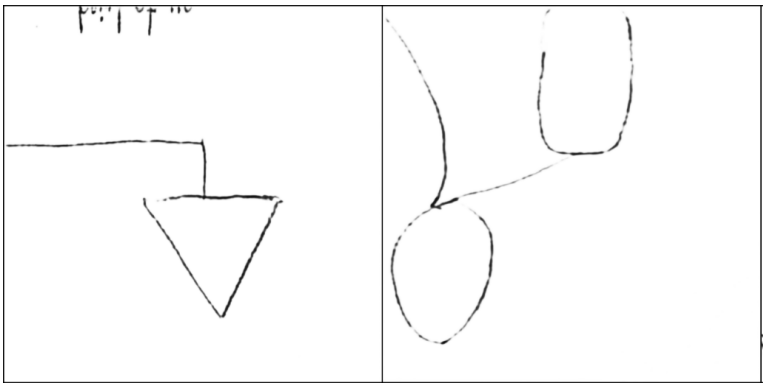
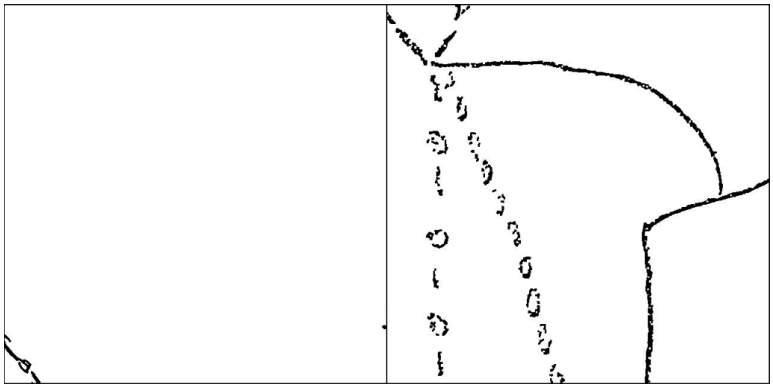
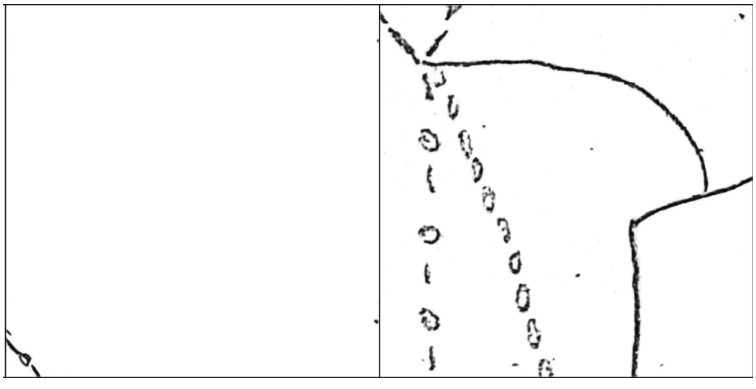
```
self.down_block1 = nn.Sequential(
            nn.Conv2d(3, 8, 3, 1, 1),
            nn.BatchNorm2d(8),
            nn.ReLU()
        )
        self.down_block2 = nn.Sequential(
            nn.Conv2d(8, 16, 3, 1, 1),
            nn.BatchNorm2d(16),
            nn.ReLU()
        )
        self.down_block3 = nn.Sequential(
            nn.Conv2d(16, 32, 3, 1, 1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(2, 2)
        )
```

```
        self.my_equal = nn.Sequential(
            nn.Conv2d(32, 32, 3, 1 ,1),
            nn.BatchNorm2d(32),
            nn.ReLU()
        )
        self.up_block1 = nn.Sequential(
            nn.Conv2d(32, 32, 3, 1, 1),
            nn.ReLU()
        )
        self.up_block2 = nn.Sequential(
            nn.ConvTranspose2d(64, 32, 1, 1),
            nn.ReLU(),
            nn.ConvTranspose2d(32, 16, 2, 2),
            nn.ReLU()
        )
        self.up_block3 = nn.Sequential(
            nn.Conv2d(32, 16, 3, 1, 1),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.Conv2d(16, 8, 3, 1, 1),
            nn.BatchNorm2d(8),
            nn.ReLU()
        )
        self.up_block4 = nn.Sequential(
            nn.Conv2d(16, 8, 3, 1, 1),
            nn.BatchNorm2d(8),
            nn.ReLU(),
            nn.Conv2d(8, 1, 3, 1, 1)
        )

    def forward(self, x):
        x1 = self.down_block1(x)
        x2 = self.down_block2(x1)
        x3 = self.down_block3(x2)
        x4 = self.my_equal(x3)
        x = self.up_block1(x4)
        x3 = torch.cat([x, x3], dim=1)
```

```
        x = self.up_block2(x3)
        x2 = torch.cat([x, x2], dim=1)
        x = self.up_block3(x2)
        x1 = torch.cat([x, x1], dim=1)
        x = self.up_block4(x1)
        return x
```

After adding skipped connections my model became even more similar to the original U-Net one but, interestingly, after 3 versions of this model I could not get close to results even my base model provides, not to mention outputs that U-Net produces:

True	Predicted
	
	

Conclusions

As this work showed, it is possible to create an effective autoencoder that will be relatively light and fast and will give a decent result to work with it further.
Because of the big amount of things to tune, like optimizer, loss function or schedulers, the modeling itself takes not that much time compared to the full problem solving process.
But some things that did work well in U-Net, did not in my model, like perceptual loss and skipped connections.
The biggest influence on the outputs was exerted by the model architecture: only two different “branches” could give the acceptable results and the rest of the work was just to set up the layers amount, order and channels number.
If you would like to see many-many words and pictures, you can follow this link and get acquainted with the full modeling process with corresponding outputs.

Further Improvement

As for the next steps for improvement we can consider working more with perceptual loss and different versions of skipped connections. The key is not to replicate the original U-Net model during this process.
The other possible field that could be improved - data preprocessing: while I use hybrid approach with