Inpainting landscapes

Introduction

#TODO

Installation

1. Clone the repository

```
git clone git@github.com:istepka/im-outpainting.git
```

2. Install the requirements

```
pip install -r requirements.txt
```

3. Download the data from the following url: https://1drv.ms/u/c/35ddce87939617c8/EWyelnH8qlJEgFNjAX_z3EABZiNsaDIGrOS8du5V52DSXA?e=mkizPd and uzip it into the `data/raw` folder.

Usage

1. Prepare the data

```
python src/preprocess.py [DEFAULT OPTIONS: --data_dir data/raw, --out_dir
data/processed, --num_workers CPU_COUNT, --size 256]
```

2. Train the model

```
python src/train.py [ OPTIONAL ARGUMENTS
    --optimizer ["adam", "SGD", "rmsprop", "adagrad", (default "adam")]
    --epochs [INT, (default 20)]
    --batch_size [INT, (default 32)
    --loss ["mse" "l1" "cross_entropy" "poisson" "kldiv", (default "mse")]
    --learning_rate [INT, (default 1e-3)]
    --model ["UNet", "Encoder-Decoder", (default "UNet")]
    --experiment_name [STR, (default $model)]
    ]
```

Data sources

The dataset is comprised of 16k images of landscapes. They were collected from the following sources:

1. Landscape Pictures (~4k images) https://www.kaggle.com/datasets/arnaud58/landscape-pictures?rvi=1
2. Landscape Recognition Image Dataset (~12k images) https://www.kaggle.com/datasets/utkarshsaxenadn/landscape-recognition-image-dataset-12k-images

We cut-out 256x256 squares from images to create a larger dataset comprising of ~50k images instead of 16k. For experimentation purposes we decided to use smaller dataset size, so it is faster to train (and cheaper).

Data augumentation:

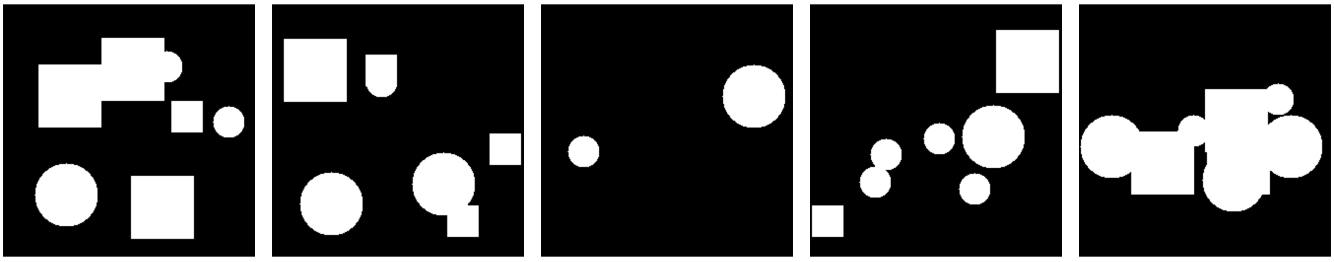
- Random left-right flips
- Rotations by 90 degrees only

Splits:

- Train - 10k
- Val - 2k
- Test - 10k

Example of images:

Masks:



Masks applied to images:

Architectures

An Encoder-Decoder architecture comprises two parts: an encoder to extract features and a decoder to generate output. It's commonly used for tasks like image translation or image generation. Implementation here has separate encoder and decoder parts. The encoder consists of convolutional layers followed by batch normalization and ReLU activation. The decoder reverses this process, ending with a non-linear activation function.

Specification:

- Parameters: ~744 K
- Weight: ~ 3 MB
- Default config:
    - Epochs: 20
    - LR: 1e-3
    - Optimizer: Adam

UNet is a convolutional neural network architecture designed for semantic segmentation tasks. It consists of an encoder-decoder structure with skip connections, enabling precise localization. Implementation provided here has an encoder with three blocks, a bottleneck layer, and a decoder with corresponding blocks. Skip connections concatenate encoder and decoder feature maps.

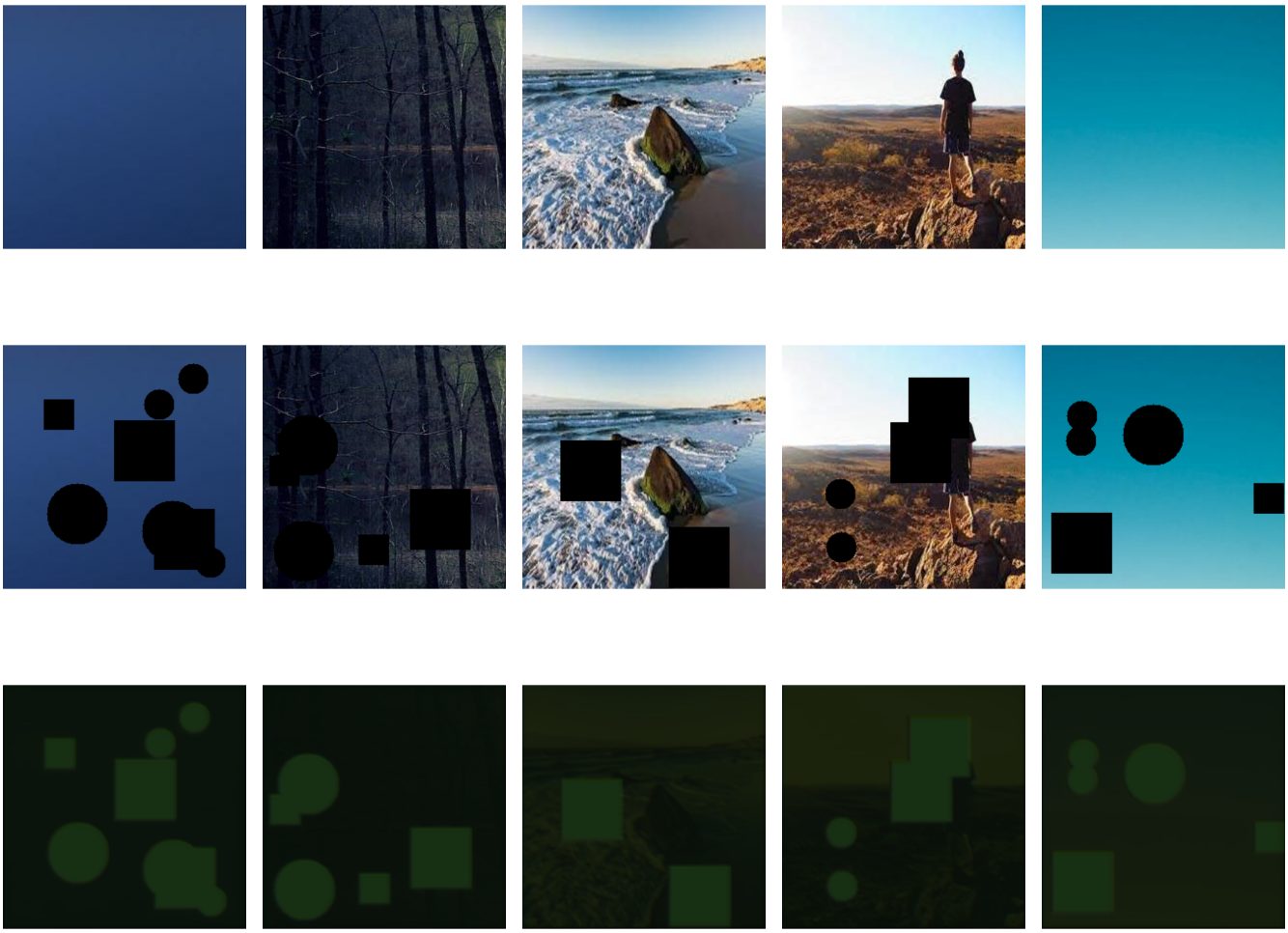
Specification:

- Parameters: 2.7 M
- Weight: ~ 10.6 MB
- Default config:
    - Epochs: 20
    - LR: 1e-3
    - Optimizer: Adam

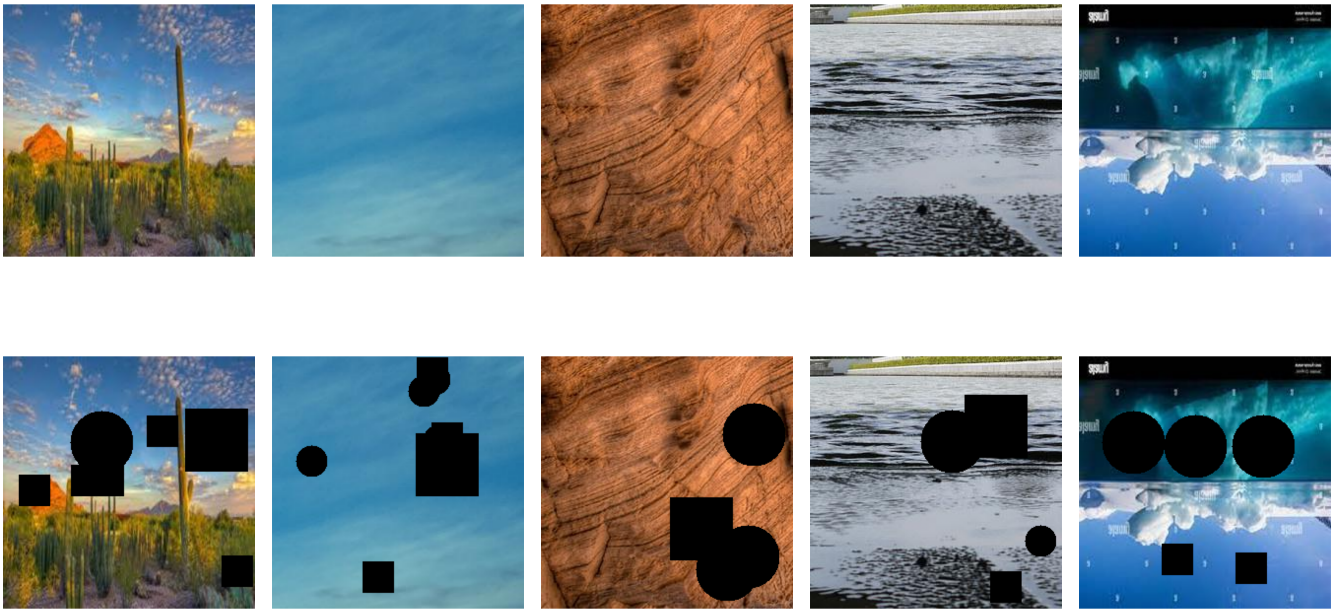
Training

Images on first epoch:

Original images

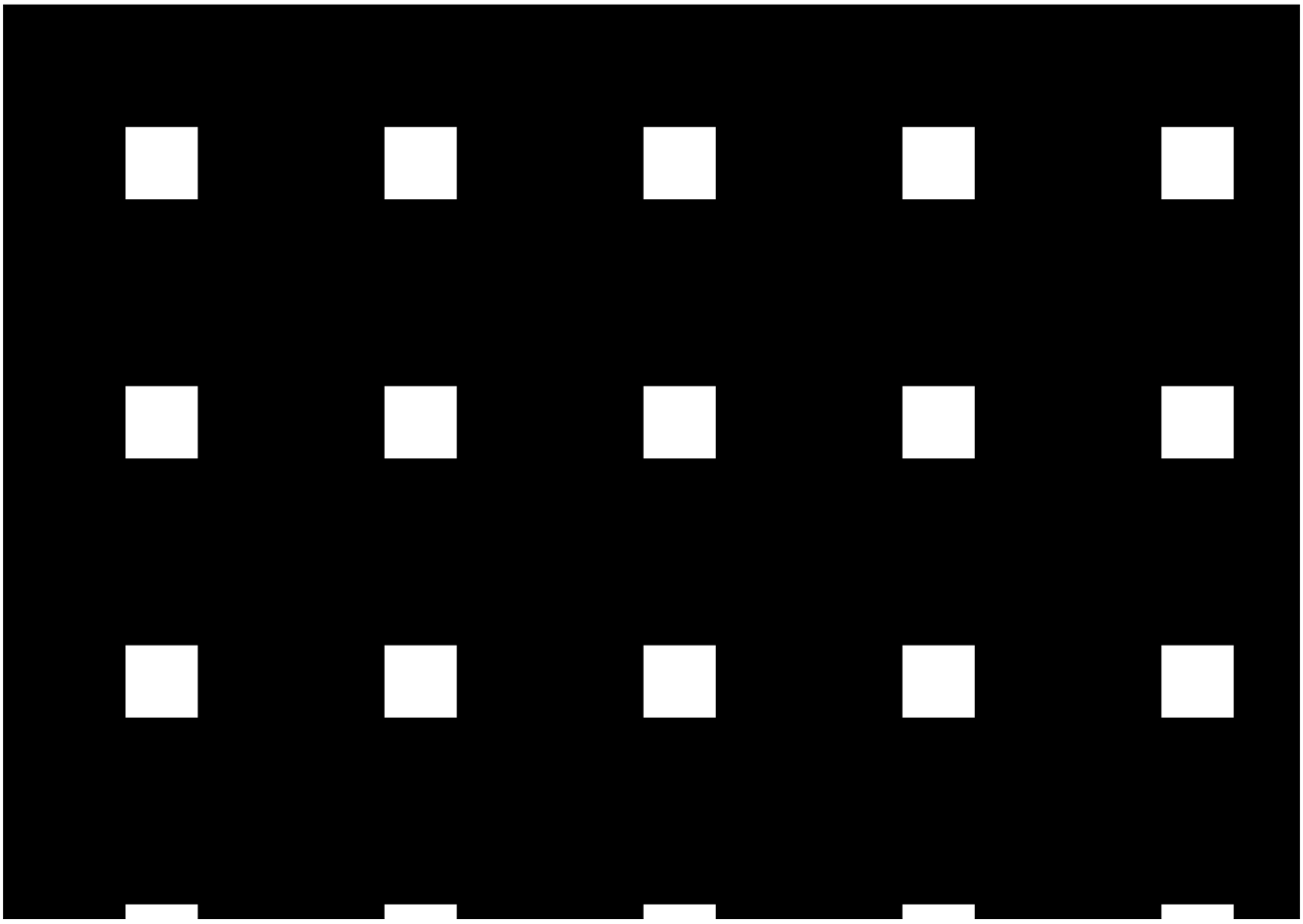


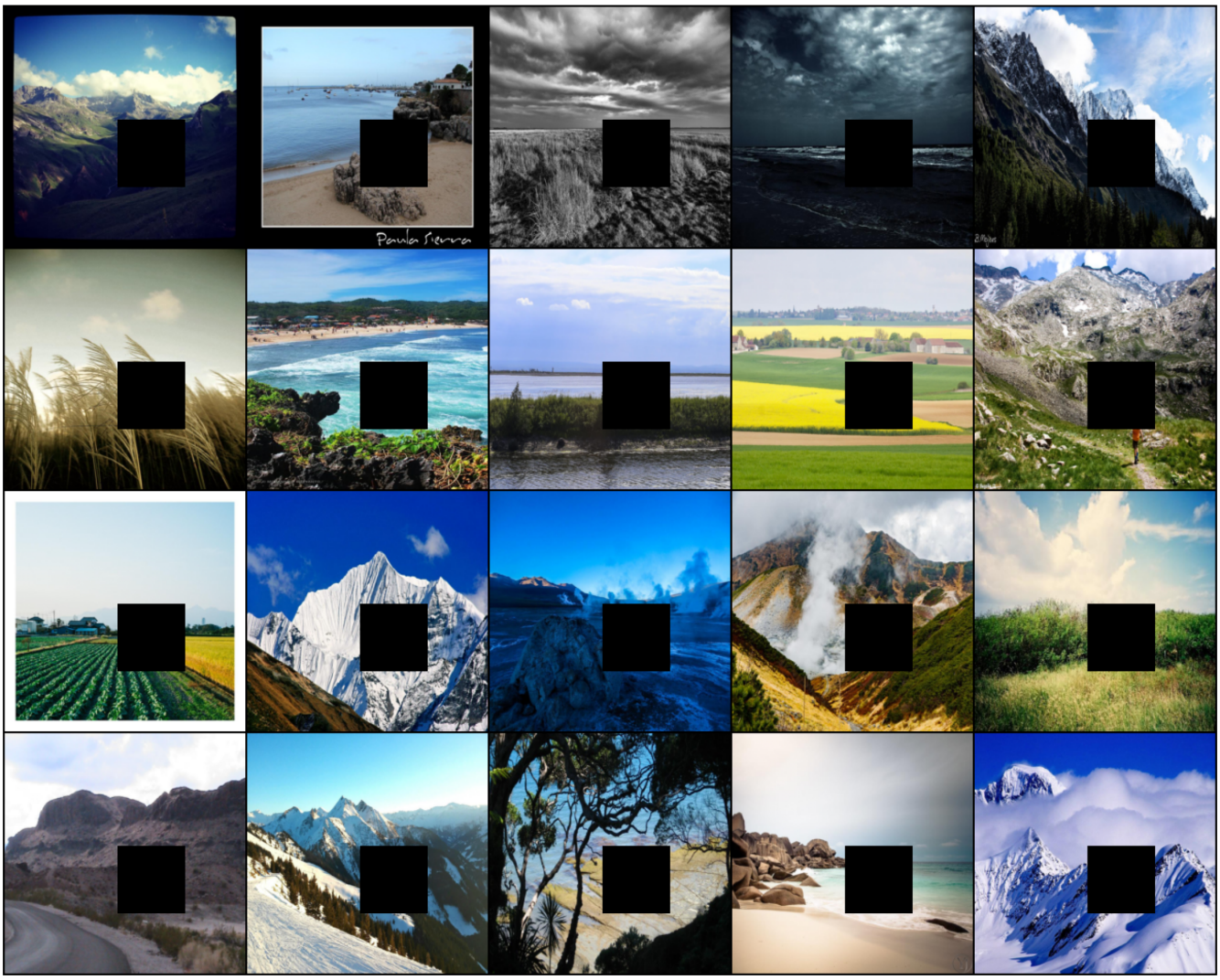
Images on the last epoch:

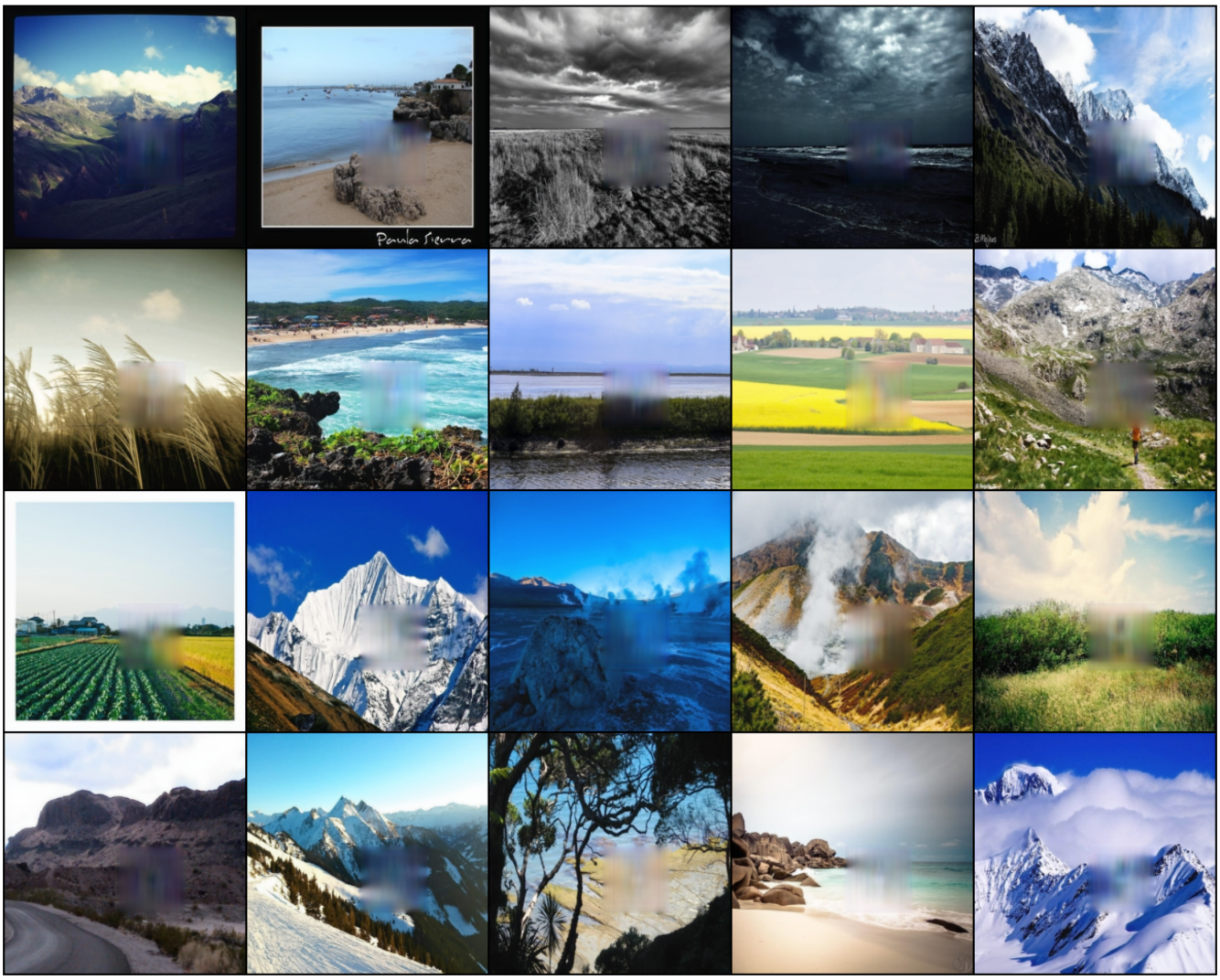


I think the results are pretty good. We can see that the model is able to inpaint the images and fill the missing parts with plausible content. However, the input mask shapes are often pretty visible as blurry shapes. I believe being able to remove them would be a good next step for this project, but not an easy one. I would guess that we would have to use some sort of GAN architecture to achieve that or borrow some ideas from generative AI like loss function based on perceptual similarity or style transfer. Nevertheless, it is clear that model is able to kind of understand the context of the image and fill the missing parts with blurred, but very much plausible content (e.g. sky, grass, trees, edges, etc.).

Demo / some more images



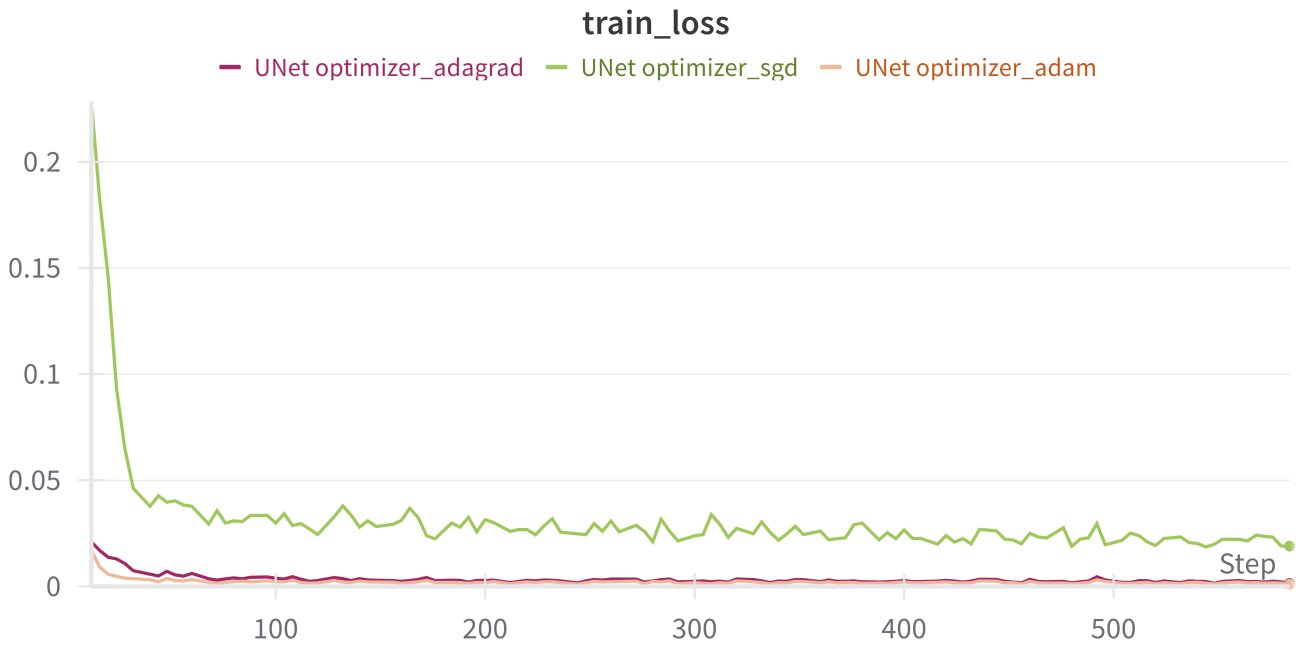


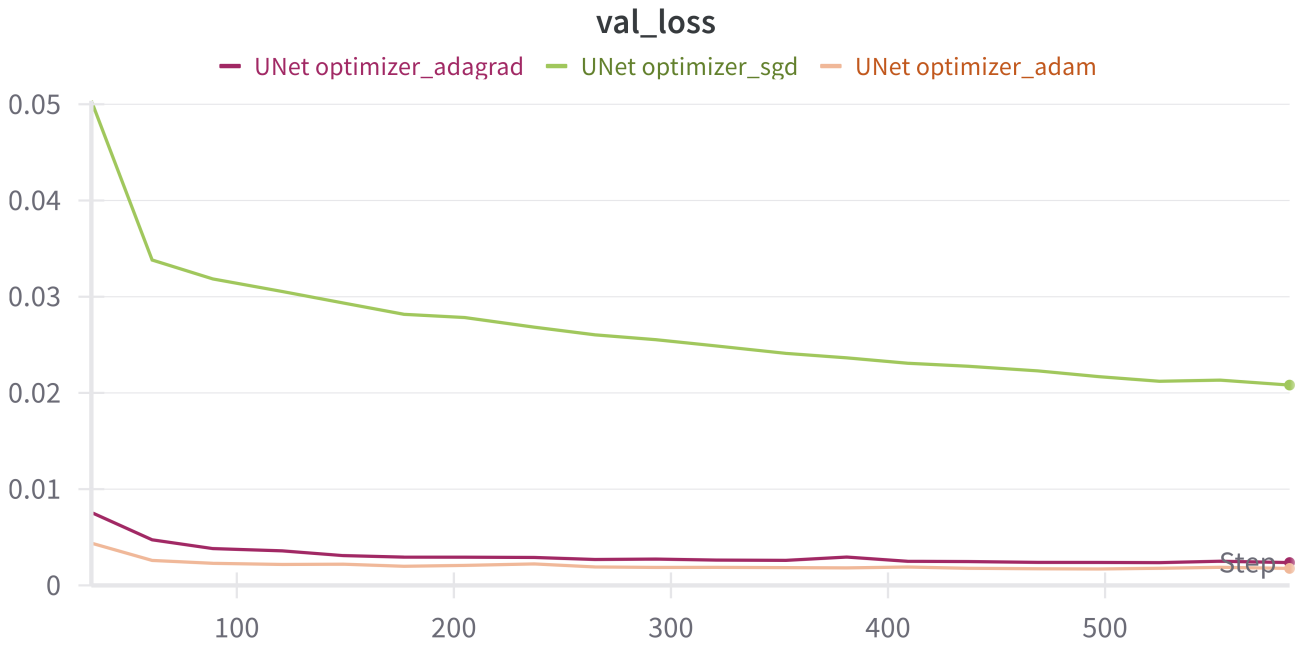


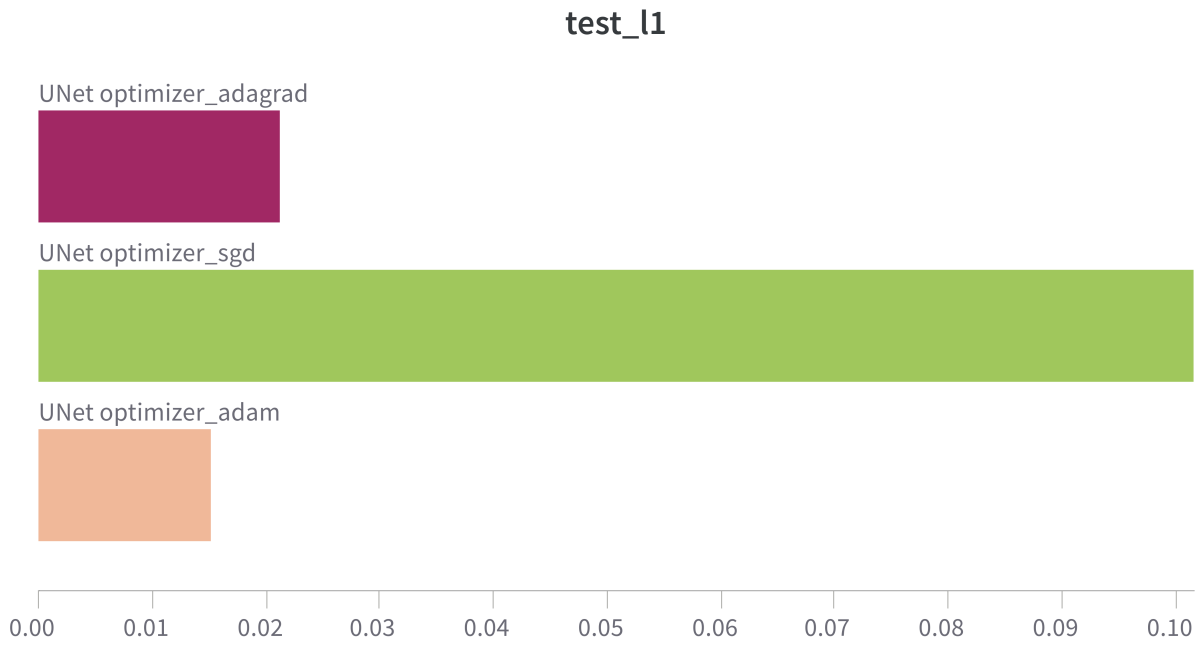
Benchmarking

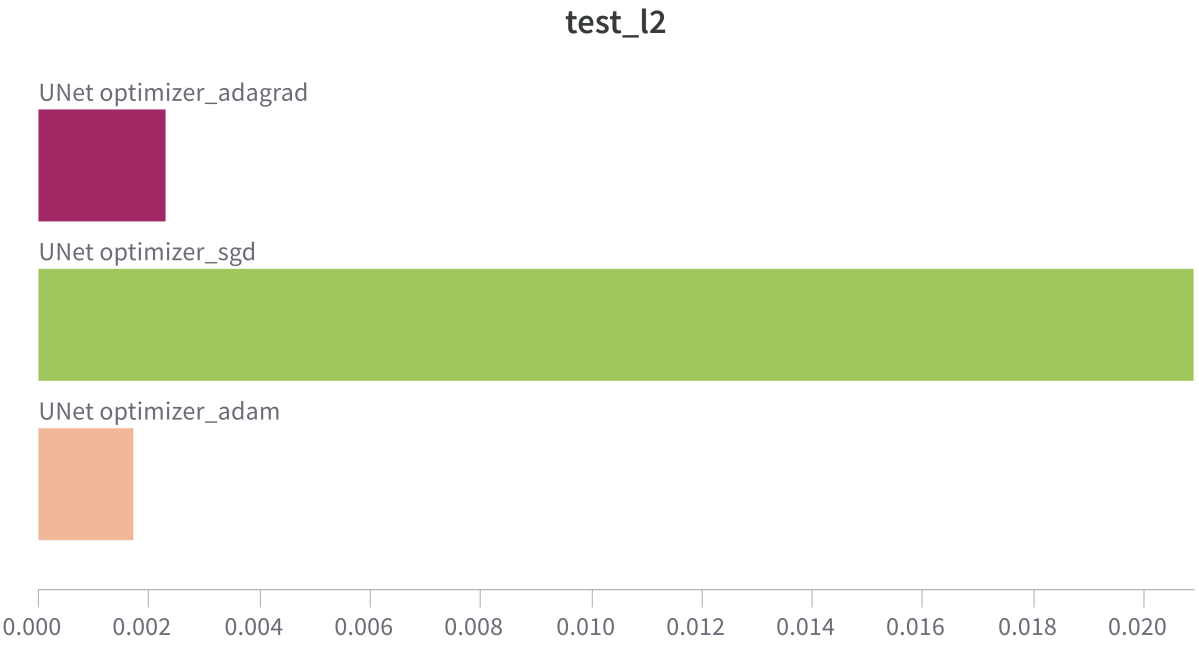
Optimizers

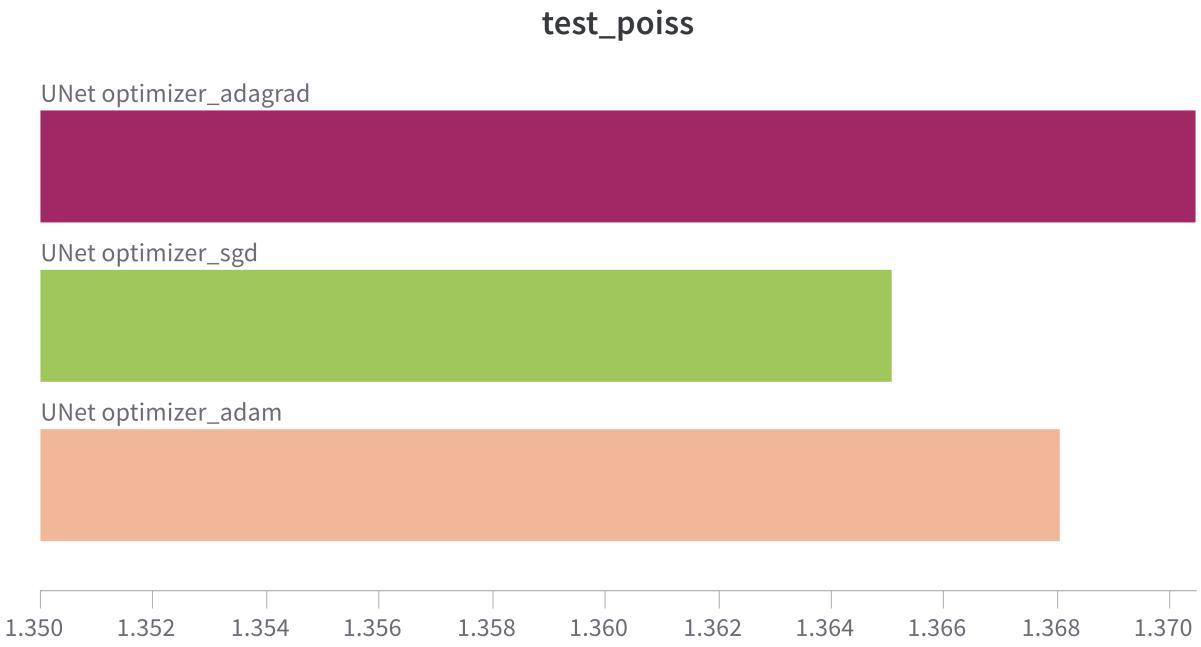
Adam, SGD, RMSprop, Adagrad
*rmsprop is omitted in this visualizations because it achieved horrible results and makes it impossible to see anything useful on the plots.
The most interesting takeaway from this benchmark is that SGD is consistently much worse than Adam and Adagrad. Adam and Adagrad are very similar in terms of performance, but Adam is just a bit better.

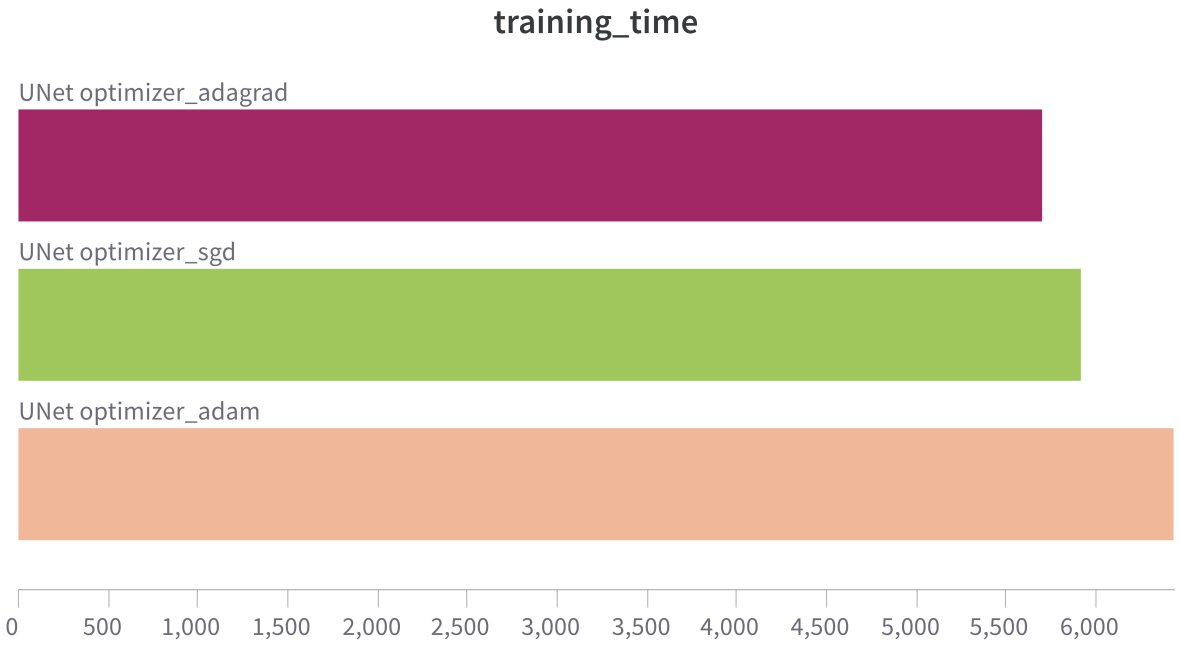

train_loss
UNet optimizer_adagrad
UNet optimizer_sgd
UNet optimizer_adam
0.2
0.15
0.1
0.05
0
Step
100
200
300
400
500


val_loss
UNet optimizer_adagrad
UNet optimizer_sgd
UNet optimizer_adam
0.05
0.04
0.03
0.02
0.01
0
Step
100
200
300
400
500


test_l1
UNet optimizer_adagrad
UNet optimizer_sgd
UNet optimizer_adam
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.10

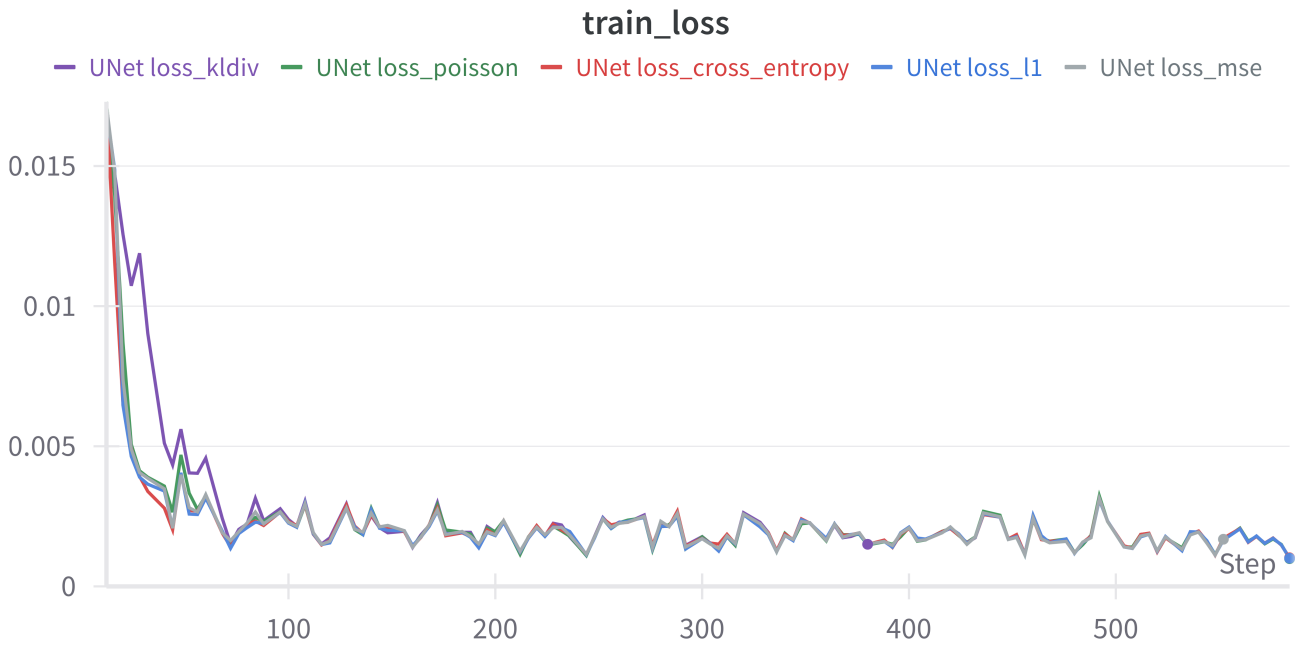

test_l2
UNet optimizer_adagrad
UNet optimizer_sgd
UNet optimizer_adam
0.000
0.002
0.004
0.006
0.008
0.010
0.012
0.014
0.016
0.018
0.020

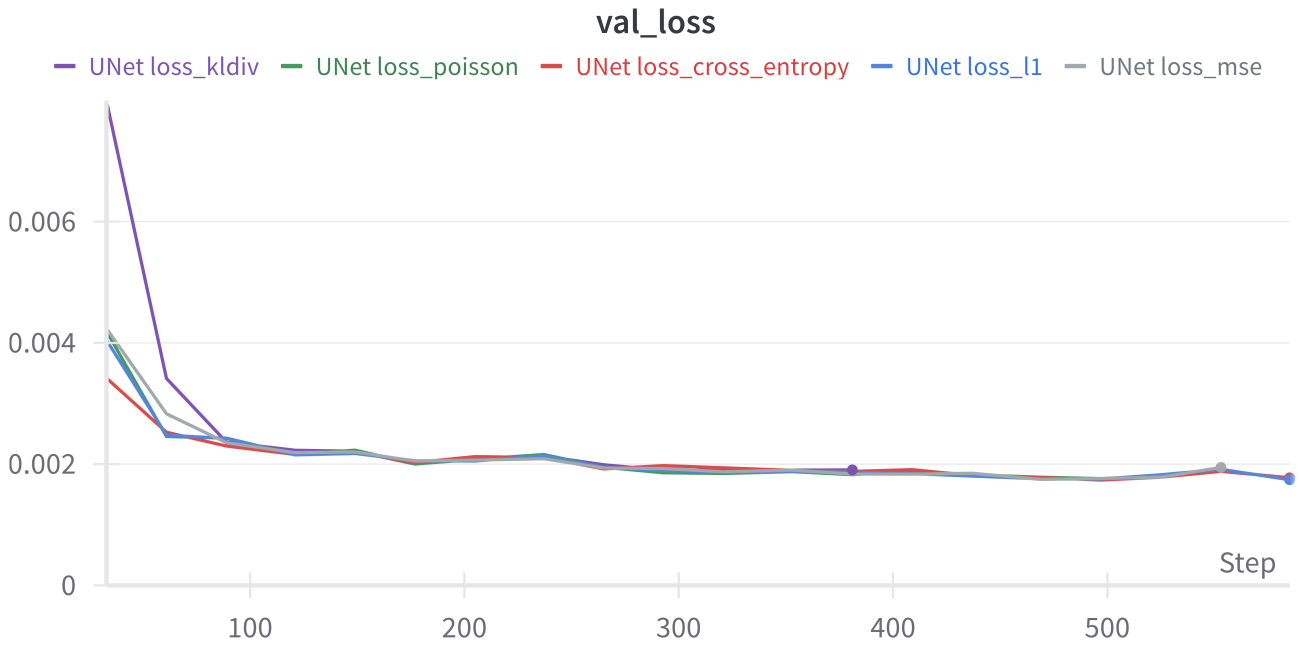


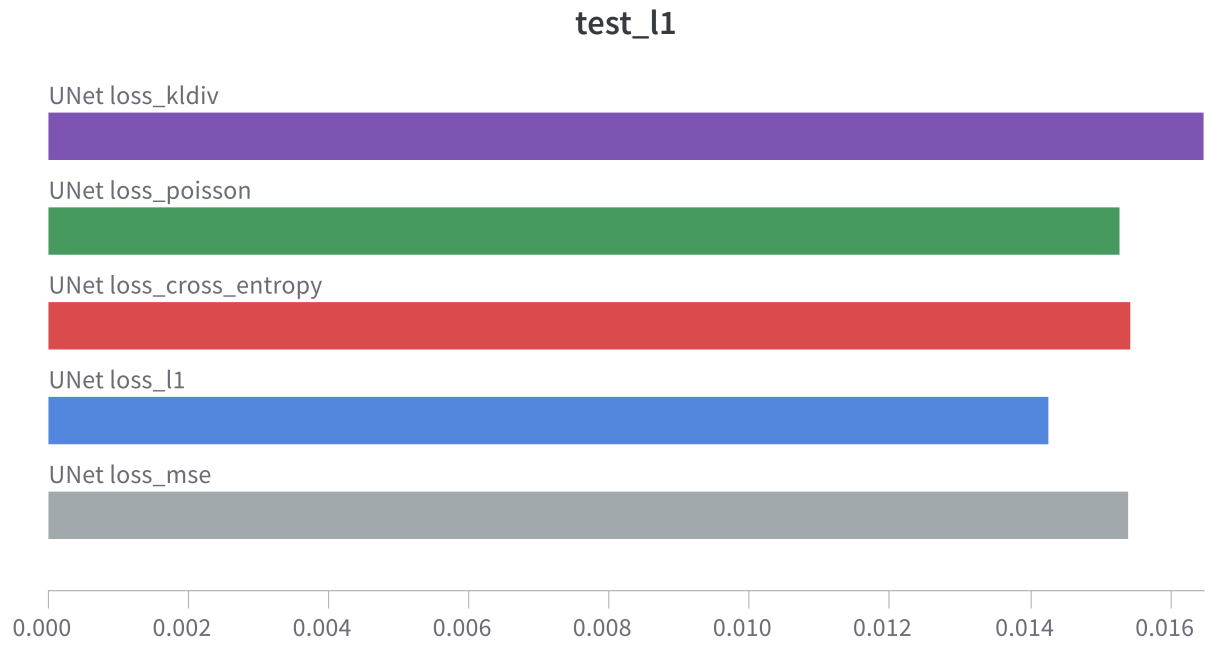


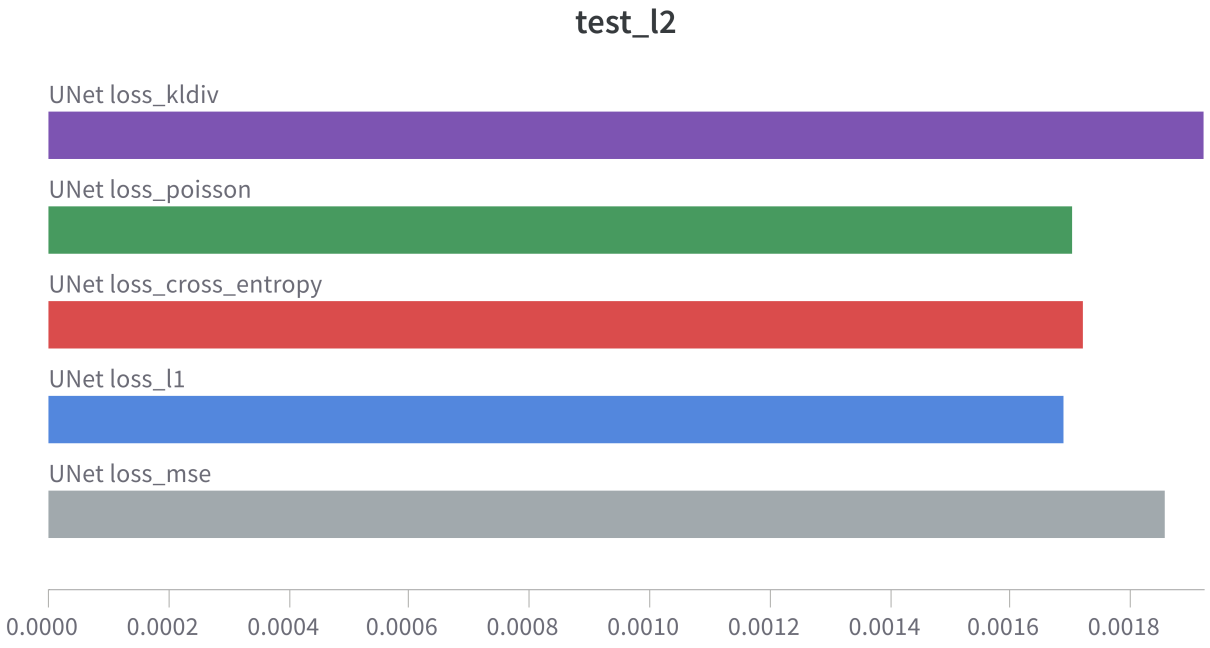
Loss functions

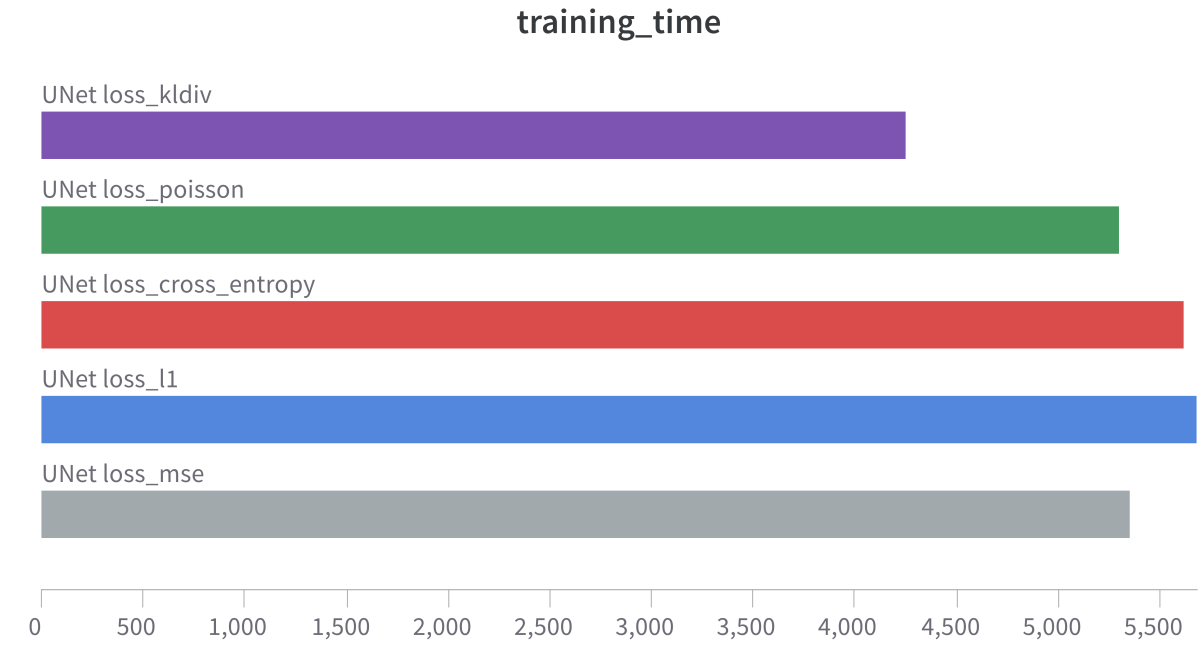
Lossess used: MSE, L1, PoissonNLLoss, KLDivLoss, CrossEntropyLoss Now, analysing loss functions. From train/val loss curves we see that all loss functions perform similarly. In terms of test MAE (L1) and MSE (L2) we see that L1 is the best choice and is closely followed by poisson loss. KLDiv is the worst choice.


train_loss
UNet loss_kldiv
UNet loss_poisson
UNet loss_cross_entropy
UNet loss_l1
UNet loss_mse
0.015
0.01
0.005
0
100
200
300
400
500
Step


val_loss
UNet loss_kldiv
UNet loss_poisson
UNet loss_cross_entropy
UNet loss_l1
UNet loss_mse
0.006
0.004
0.002
0
100
200
300
400
500
Step


test_l1
UNet loss_kldiv
UNet loss_poisson
UNet loss_cross_entropy
UNet loss_l1
UNet loss_mse
0.000
0.002
0.004
0.006
0.008
0.010
0.012
0.014
0.016

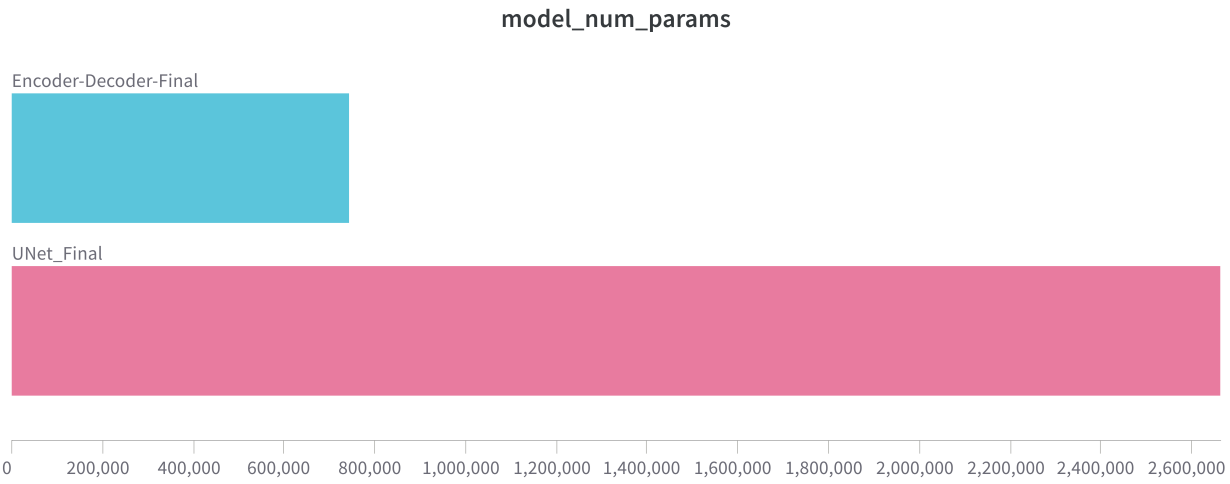

test_l2
UNet loss_kldiv
UNet loss_poisson
UNet loss_cross_entropy
UNet loss_l1
UNet loss_mse
0.0000
0.0002
0.0004
0.0006
0.0008
0.0010
0.0012
0.0014
0.0016
0.0018

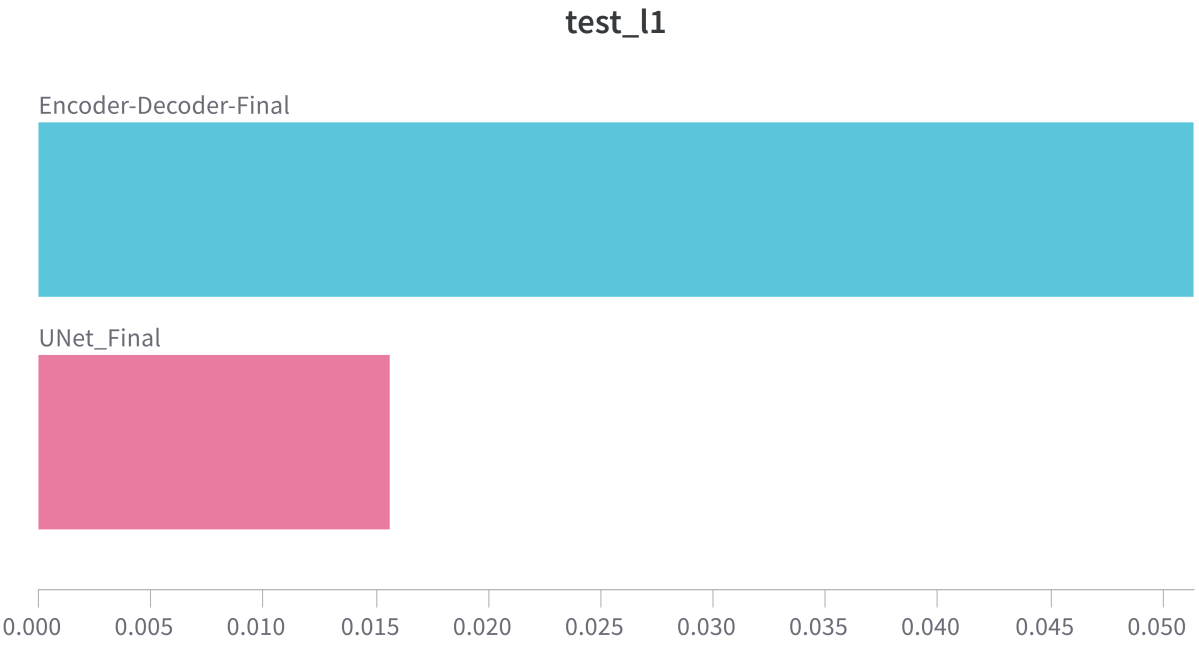


UNet vs Encoder-Decoder

Encoder-Decoder architecure results clearly show that UNet is a better choice for this task. However, I have arbitrarly made the decision to have Encoder-Decoder as sort of a smaller model for faster training. But, as

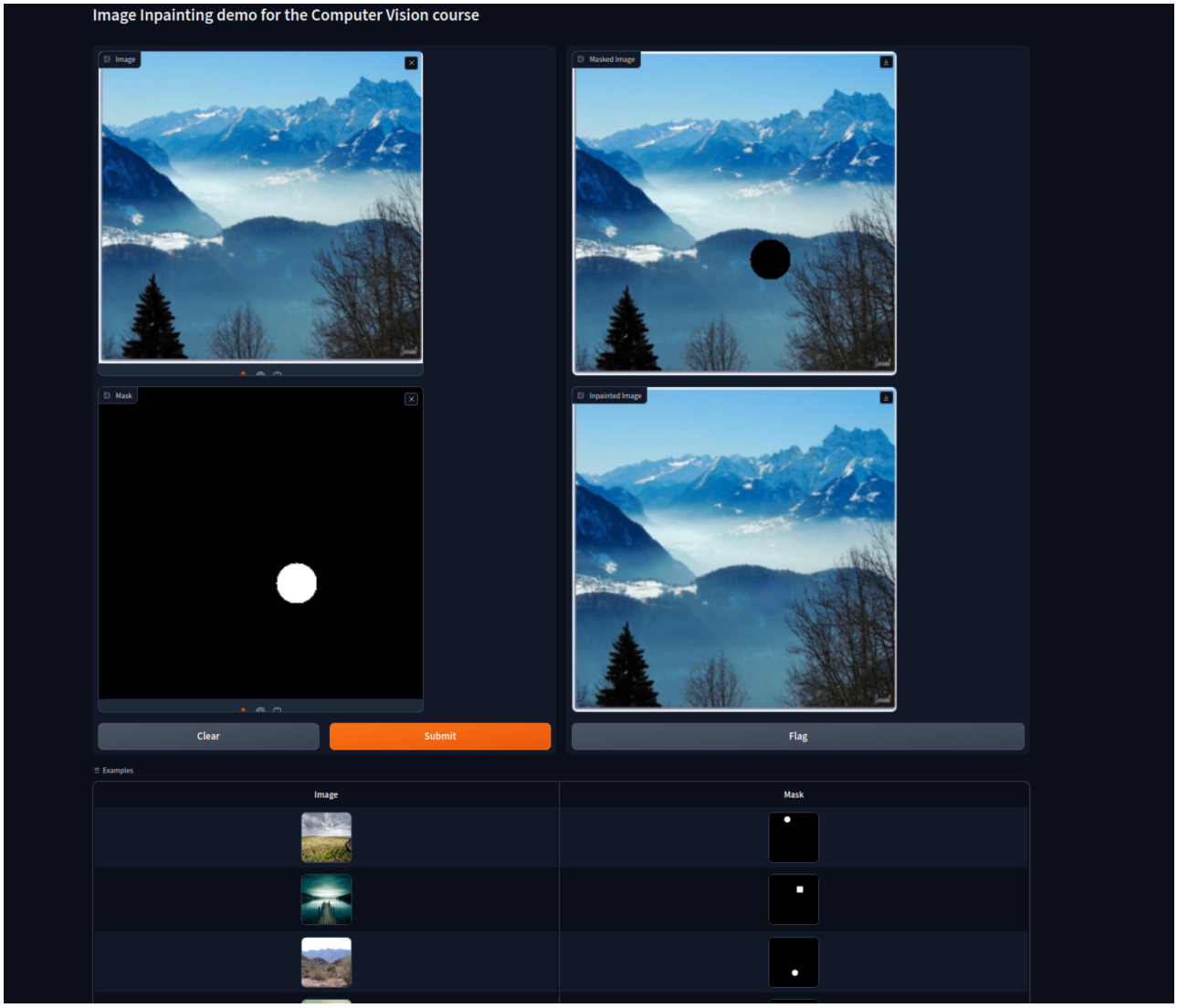
we can see from the results, it's being outperformed by UNet by landslide.





Other

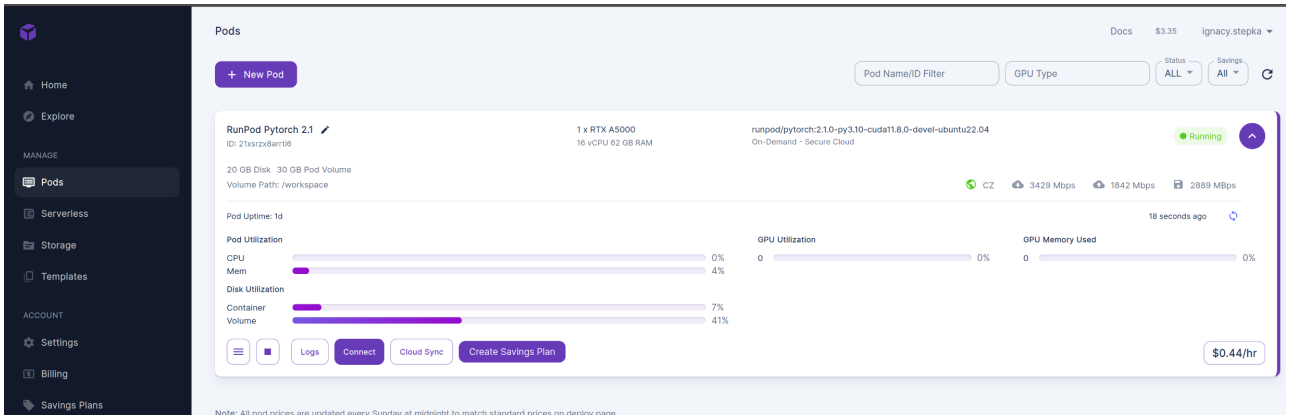
- **UI with Gradio**



To run the UI, run the following command:

```
python gradio_app.py
```

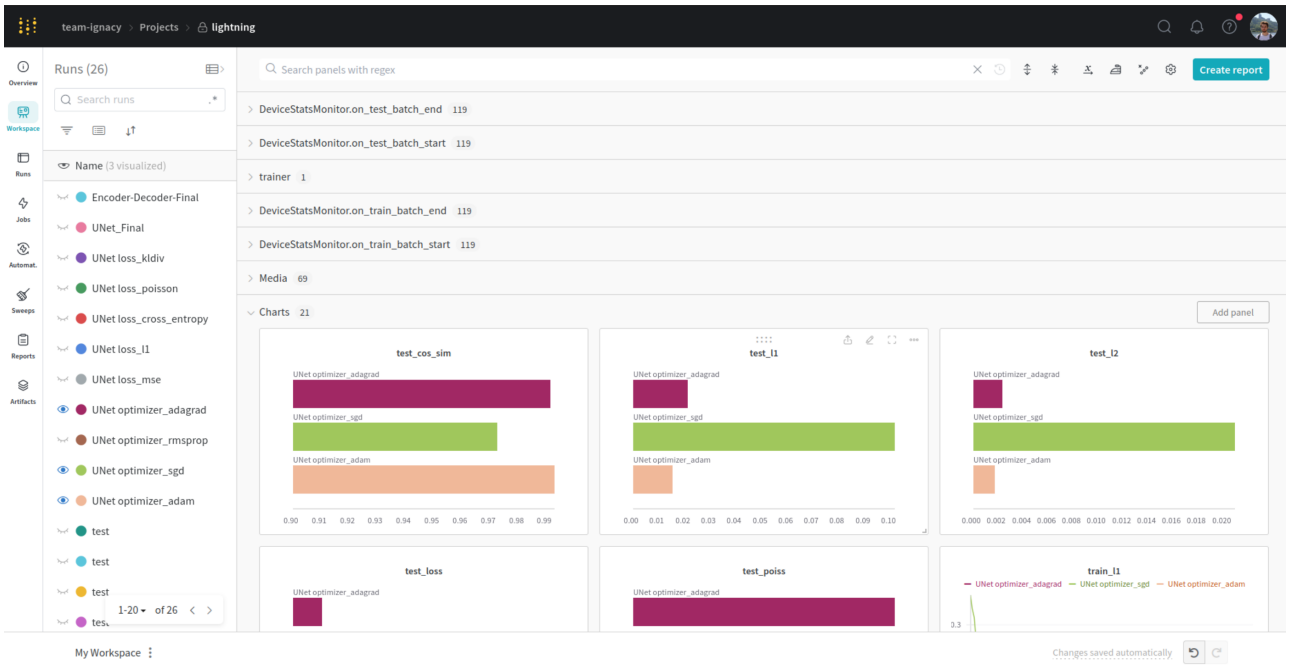
- **Deployment for training in cloud (RunPod)**



- **Docker image for Gradio App**

```
# Build the image
docker build -t inpainting .
# Run the image
docker run -p 8087:8087 inpainting
# Now you can access the app at http://localhost:8087
```

- **WanDB tracking**



Grading

What	Points	How
Image inpainting	3	The model is able to inpaint images
Own architecture (>50% own layers)	2	Encoder-Decoder CNN architecture
2nd own architecture with >50% own layers	2	Own UNet implementation
Evaluation on a test set >= 10k	1	Performed evaluation on 10,000 images in the test set
Testing various loss functions	1	MSE, L1, PoissonNLLoss, KLDivLoss, CrossEntropyLoss
Testing various optimizers	1	Adam, SGD, RMSprop, Adagrad
Image augmentations	1	Running random transformations on the input image before feeding it to the network
Weights & Biases	1	Wandb properly set up -- everything was tracked in it and I used data I saved to wandb for analysis

What	Points	How
Run as Docker	1	Dockerfile for building the docker image containing the Gradio UI
Gradio GUI	1	Gradio UI for inpainting
RunPod (DevOps)	1	Set up development environment with RunPod

Total: 15

References

Related stuff that I used to understand the topic and implement the project: [1] UNet: https://arxiv.org/abs/1505.04597s
[2] UNet + Attention: https://arxiv.org/pdf/2209.08850.pdf
[3] II-GAN: https://ieeexplore.ieee.org/document/8322859

License