Github link: https://github.com/yiliu7724/Monet

Week 5 cycle GAN project deliverable 1

For this particular assignment I have to use a report because running the notebook in one go overloads my computer as well as the kaggle notebook.

We are provided with original Monet paintings, around 300 of them. We are also provided with 7038 photos that we have to convert to a style that resembles Monet style painting.

The purpose of this assignment is to apply the art style of Monet to new photos.

This is to be accomplished by a type of GAN known as a cycleGAN. Essentially a cycleGAN is 2 GANs forming a loop. We have one GAN that turns our photos into monet style. And we have another GAN that converts our monet style back into photo style. As a result, for model evaluation I will evaluate both the forward and backwards GAN to evaluate the strength of that particular cycle GAN.

For this assignment, for full disclosure I will be using the architecture suggested by this notebook:

https://www.kaggle.com/code/amyjang/monet-cyclegan-tutorial/notebook

Which in turn comes from:

https://arxiv.org/pdf/2203.02557

This particular UVCGAN architecture makes use of skip connections. Likely to prevent vanishing gradients.

Part 1: data loading

```
GCS_PATH = KaggleDatasets().get_gcs_path()

MONET_FILENAMES = tf.io.gfile.glob(str(GCS_PATH + '/monet_tfrec/*.tfrec'))
print('Monet TFRecord Files:', len(MONET_FILENAMES))

PHOTO_FILENAMES = tf.io.gfile.glob(str(GCS_PATH + '/photo_tfrec/*.tfrec'))
print('Photo TFRecord Files:', len(PHOTO_FILENAMES))
```

The images in this dataset are already RGB. so we have 3 channels.

Github link: https://github.com/yiliu7724/Monet

```
IMAGE_SIZE = [256, 256]
NO_CHANNELS = 3

def read_tfrecord(example):
    tfrecord_format = {
        "image_name": tf.io.FixedLenFeature([], tf.string),
        "image": tf.io.FixedLenFeature([], tf.string),
        "target": tf.io.FixedLenFeature([], tf.string)
    }
    example = tf.io.parse_single_example(example, tfrecord_format)
    image = tf.image.decode_jpeg(example['image'], channels=3)
    image = (tf.cast(image, tf.float32) / 127.5) - 1
    image = tf.reshape(image, [*IMAGE_SIZE, NO_CHANNELS])
    return image

def load_dataset(filenames, labeled=True, ordered=False):
    dataset = tf.data.TFRecordDataset(filenames)
    dataset = dataset.map(read_tfrecord, num_parallel_calls=AUTOTUNE)
    return dataset

monet_ds = load_dataset(MONET_FILENAMES, labeled=True).batch(1)
photo_ds = load_dataset(PHOTO_FILENAMES, labeled=True).batch(1)
```

Part 2, data cleaning and EDA.

We have already established that the images have 3 channels. The previous step also made our images the same size.

Our data seems decently clean otherwise just from visually inspecting the dataset images.

Github link: https://github.com/yiliu7724/Monet

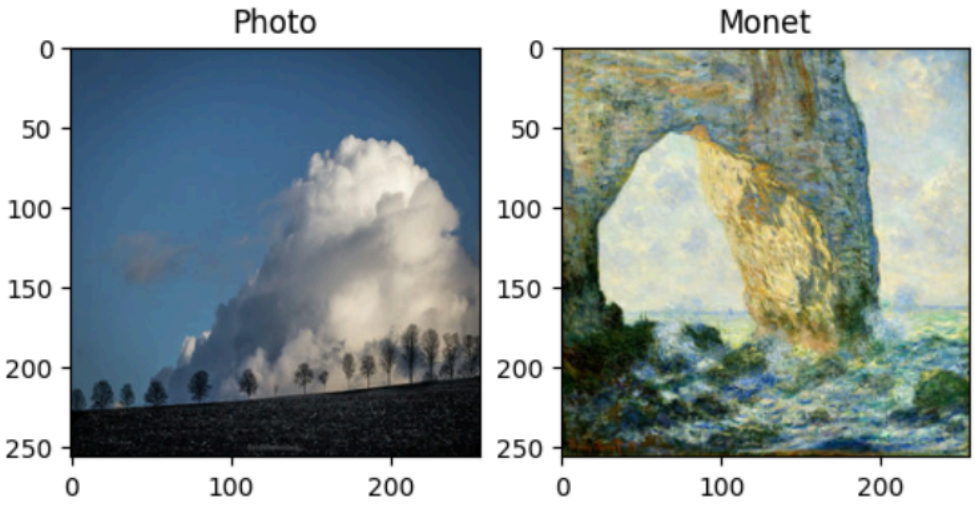
EDA: Visualize an example of picture and painting

```
example_monet = next(iter(monet_ds))
example_photo = next(iter(photo_ds))

plt.subplot(121)
plt.title('Photo')
plt.imshow(example_photo[0] * 0.5 + 0.5)

plt.subplot(122)
plt.title('Monet')
plt.imshow(example_monet[0] * 0.5 + 0.5)
```

```
<matplotlib.image.AxesImage at 0x1644610c990>
```



Part 3, building the cycle GAN and tuning parameters to see what works

The UVCGAN architecture introduced in the paper makes use of an assumption of a shared and identical latent space. This is the same assumption as the other autoencoders and variational encoders we learned in class make.

For this project I am going to be making modifications to the original notebook architecture and experiment with changing the architecture around to see what works well.

Tweaking these will allow us to see what works well and what does not.

Github link: https://github.com/yiliu7724/Monet

Note that the model is called cycle GAN because we are training 2 GANs at the same time. One transforms photo to Monet, and another transform Monet to photo. Because of this we actually have 2 GANS, and we can look at how well each of them perform.

Because of that I will also make plots of Monet being transformed into photo as well as plots of photos being transformed to Monet and see how well each of these works.

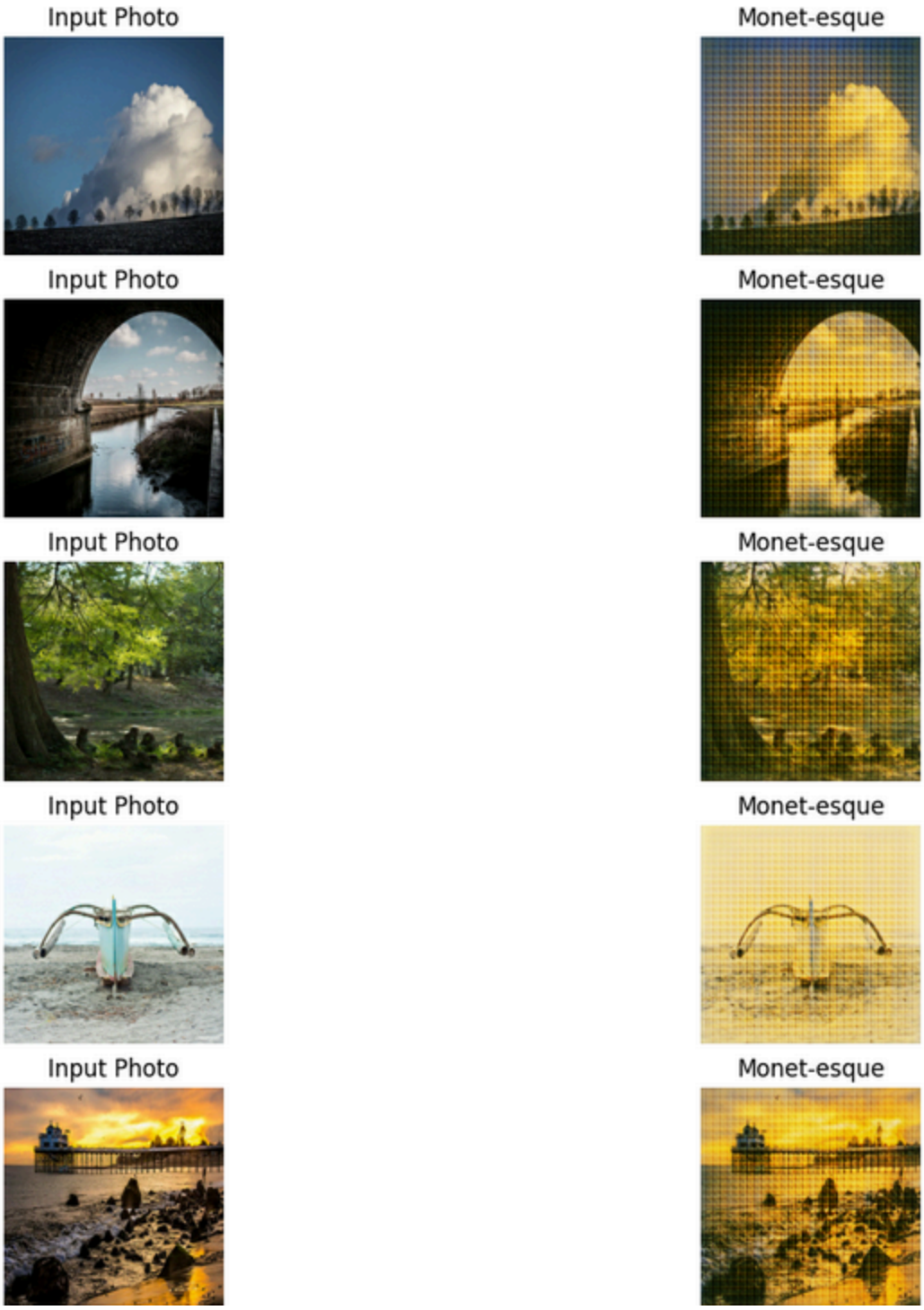
Tuning convolution layer activation function

The original notebook used in the kaggle tutorial makes use of instance normalization. But batch normalization works as well. So for simplicity we can use batch normalization to see if it works well.

Trial 1: batch normalization with ReLU activation functions

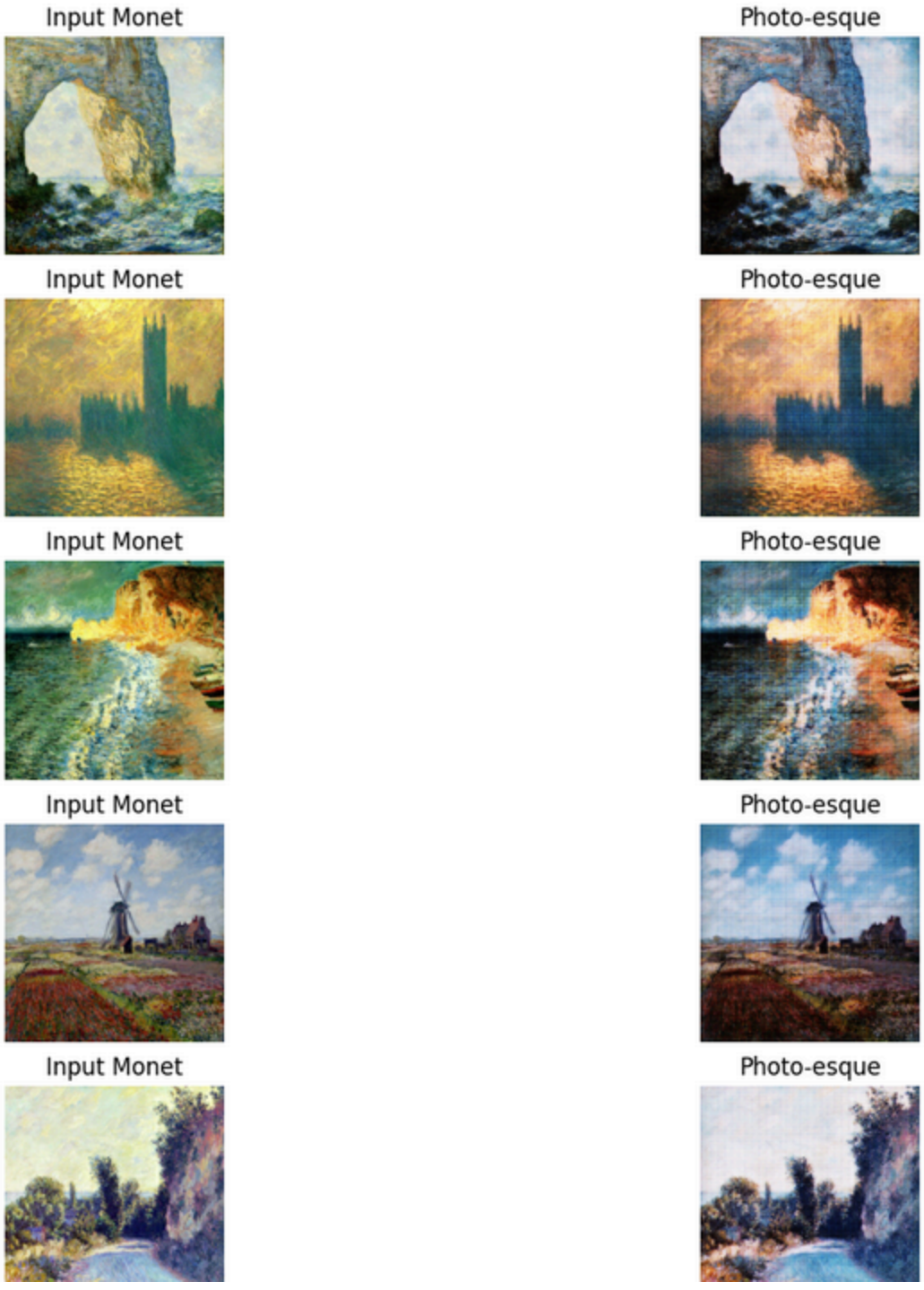
The original notebook makes use of leaky ReLU activation. However, ReLU and PReLU will work as well. So I will begin by trying what happens if we use the ReLU activation function.

Github link: https://github.com/yiliu7724/Monet



We do see those rather unsightly blocks in the generated painting. But color wise this is disappointing.

Github link: https://github.com/yiliu7724/Monet



On this direction things do seem to work okay,

Trial 2: Batch normalization with PReLU activation function

Github link: https://github.com/yiliu7724/Monet

Input Photo

Monet-esque

Input Photo

Monet-esque

Input Photo

Monet-esque

Input Photo

Monet-esque

Input Photo

Monet-esque

Github link: https://github.com/yiliu7724/Monet

Input Monet

photo-esque

Input Monet

photo-esque

Input Monet

photo-esque

Input Monet

photo-esque

Input Monet

photo-esque

We can see here that the result of the PReLU model is rather disappointing. That being said, in an earlier run of this model the results actually were not bad. Perhaps the alpha value of the PReLU activation function can use some regularization.

Github link: https://github.com/yiliu7724/Monet

Tuning regularization

The original model in the tutorial called for instance normalization and dropout. Let’s create a model without normalization and dropout and see what happens with our model.

```
Epoch 21/25
300/300 ━━━━━━━━━━━━━━━━━━━━ 374s 1s/step - monet_disc_loss: 0.6269 - monet_gen_loss: 2.5462 - photo_disc_loss: 0.6010 - photo_gen_loss: 2.5025
Epoch 22/25
300/300 ━━━━━━━━━━━━━━━━━━━━ 376s 1s/step - monet_disc_loss: 0.6259 - monet_gen_loss: 2.5275 - photo_disc_loss: 0.5979 - photo_gen_loss: 2.5068
Epoch 23/25
300/300 ━━━━━━━━━━━━━━━━━━━━ 375s 1s/step - monet_disc_loss: 0.6270 - monet_gen_loss: 2.5116 - photo_disc_loss: 0.5959 - photo_gen_loss: 2.4705
Epoch 24/25
300/300 ━━━━━━━━━━━━━━━━━━━━ 377s 1s/step - monet_disc_loss: 0.6168 - monet_gen_loss: 2.4556 - photo_disc_loss: 0.5953 - photo_gen_loss: 2.4175
Epoch 25/25
300/300 ━━━━━━━━━━━━━━━━━━━━ 375s 1s/step - monet_disc_loss: 0.6359 - monet_gen_loss: 2.4278 - photo_disc_loss: 0.5979 - photo_gen_loss: 2.4159
```

Github link: https://github.com/yiliu7724/Monet

Input Photo

Monet-esque

Input Photo

Monet-esque

Input Photo

Monet-esque

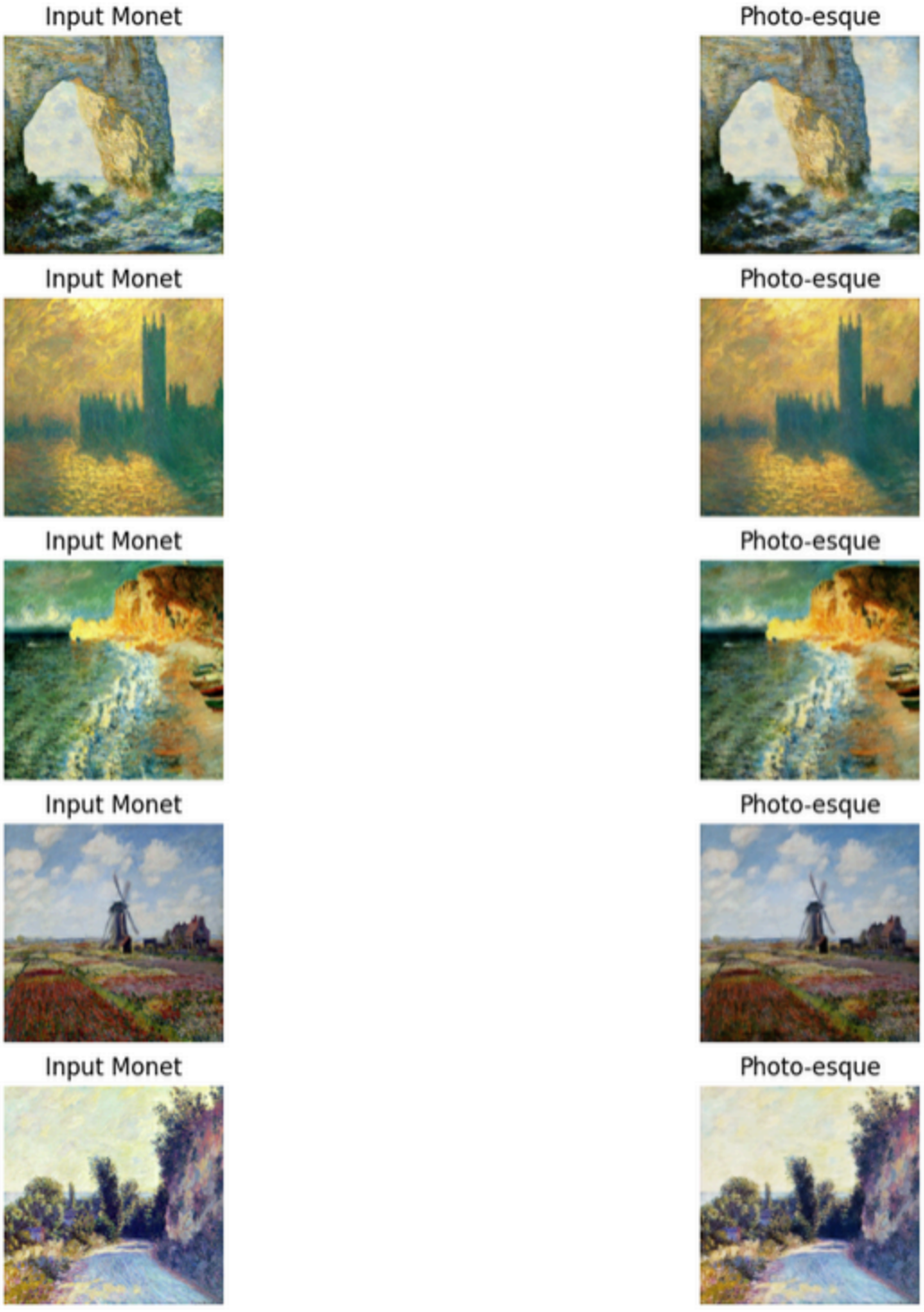
Input Photo

Monet-esque

Input Photo

Monet-esque

Github link: https://github.com/yiliu7724/Monet



What we see here is that the training seems to be able to converge further evidenced by lower loss. However, the resulting Monet style pictures we generated with the model looked too much like the original pictures and do not resemble paintings. As a result, I would say regularization is needed.

Github link: https://github.com/yiliu7724/Monet

Tuning filter sizes and padding sizes

The original tutorial notebook had a filter size per layer of 4. For the preceeding trials I had fiter size set to 4.

However, the architecture of the convolution layers also means that if we have too big of filter sizes we may very well exceed the maximum memory allowed. As a result the amount of tuning I can do is limited in this regard.

Filter size 5

Github link: https://github.com/yiliu7724/Monet

Input Photo

Monet-esque

Input Photo

Monet-esque

Input Photo

Monet-esque

Input Photo

Monet-esque

Input Photo

Monet-esque

Github link: https://github.com/yiliu7724/Monet

Input Monet

Photo-esque

Input Monet

Photo-esque

Input Monet

Photo-esque

Input Monet

Photo-esque

Input Monet

Photo-esque

This does seem to be better than filter size 4. But is it because its larger or is it because the filter size is odd? Let’s find out

Github link: https://github.com/yiliu7724/Monet

FIlter size 3

Input Photo

Monet-esque

Input Photo

Monet-esque

Input Photo

Monet-esque

Input Photo

Monet-esque

Input Photo

Monet-esque

Github link: https://github.com/yiliu7724/Monet

Input Monet

Photo-esque

Input Monet

Photo-esque

Input Monet

Photo-esque

Input Monet

Photo-esque

Input Monet

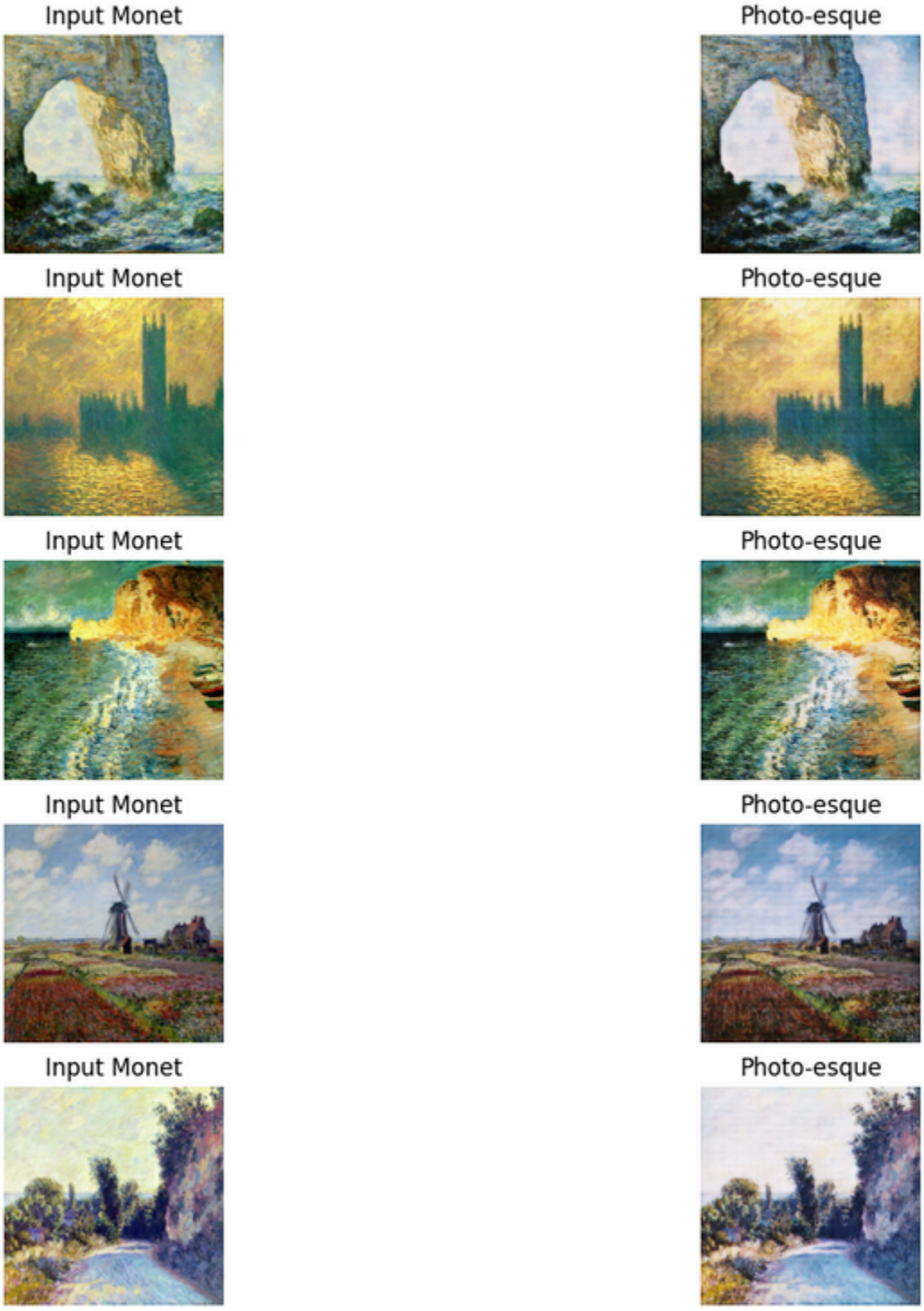
Photo-esque

What we see with filter size 3 is that the model converges faster. At the cost of more grainy generated images.

So it looks like an even sized filter works better. Let's try a larger even sized filter then.

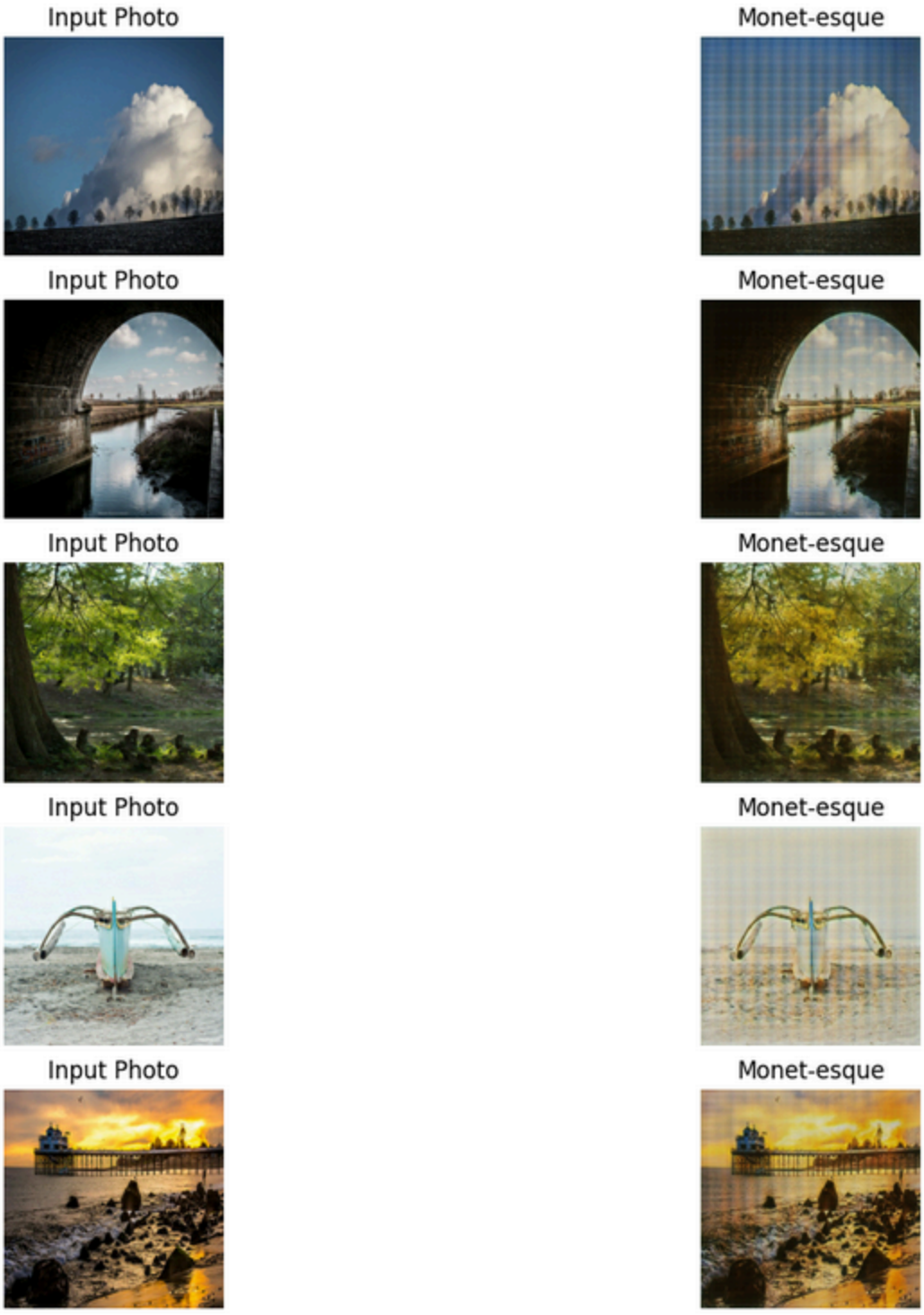
Github link: https://github.com/yiliu7724/Monet

Filter size 6



We can still see graininess, but it is not as severe in that the grains are larger.

Github link: https://github.com/yiliu7724/Monet



Conclusions and discussions

Github link: https://github.com/yiliu7724/Monet

1. Using PReLU rather than ReLU helps with making the generated colors better, at the cost of longer training time (due to more parameters) and more instability between training runs.

2. Using larger filter sizes seems to work well. Using even number sized filters seems to eliminate the grains in the generated images more. In the run that the images in this report are collected this does not seem to be true though, which is why I noted the instability between runs.

3. Regularization is necessary for the model to actually change the images. Not using regularization results in images that aren't really transformed. (true for all runs)

4. The cycle GAN model seems better at matching colors that are Monet style than at matching shapes and brush strokes. The resulting pictures do take on the color scheme that resembles a painting, but don't yet have the texture and stuff of a painting.

5. THe results between different trials are a bit unstable. The way the images look each time I run this notebook is different. This is especially true of PReLU activated models.

6. This points to a likely best model that uses regularization, has filter size of 6, uses PReLU. So let's create a final model with these:

Creating the final model and submitting

This final model uses a filter size of 6, PReLU activation, and batch normalization and dropout.

Github link: https://github.com/yiliu7724/Monet

Input Photo

Monet-esque

Input Photo

Monet-esque

Input Photo

Monet-esque

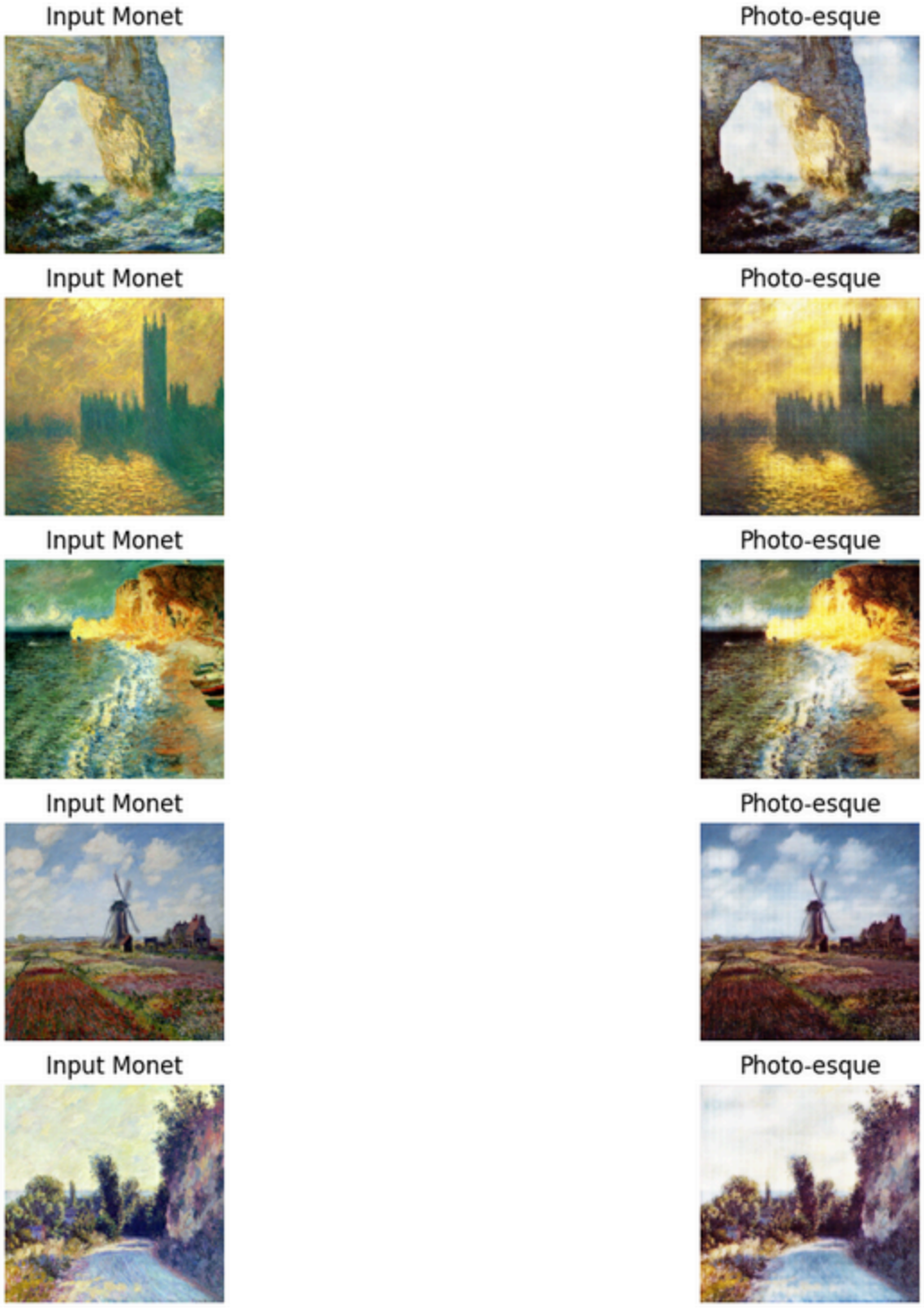
Input Photo

Monet-esque

Input Photo

Monet-esque

Github link: https://github.com/yiliu7724/Monet



We can see here that this final model is actually decent, not bad at all.

I have mentioned that in the iteration that the previous images were created in that it was an outlier (especially the PReLU). This is why I said that. This by all accounts look decent.