Deep Learning- Final Course Project

Yuval Uner (ID. 322558842),Omri Ben Hemo ID. 313255242)

Submitted as final project report for the DL course, BIU, 2023

1 Introduction

As part of the DL course, we have learnt about various aspects of machine learning and deep learning in particular, both the theoretical aspects of deep learning and the practical aspects of it.
In this project, we explored a part of the practical aspect that we have learnt.
In particular, in this project we explore the architecture and usage of GANs - Generative Adversarial Networks, by taking part in this Kaggle competition, where the goal is to succesfully generate Monet style paintings using a neural network.
As such, we explored 2 different methodologies for GAN architectures - simple GAN and cycle GAN.

2 Solution

2.1 General approach

The given problem is a generation problem - generating Monet style paintings using a neural network. Our approach to this problem was using GANs - Adversarial Neural Networks, and we focused on implementing 2 different methodologies for our GAN architectures: cycle GAN and simple GAN.

2.2 Design

2.2.1 Picking the data

The feature that we decided to use for choosing our paintings was a painting's vibrancy - we aimed to take paintings from the dataset with varying vibrancy, so that the model will learn from both paintings with more subdued colors as well as paintings with more vibrant and lively colors.
Our selection algorithm is as follows:

1. Load paintings
2. For every painting, calculate painting vibrancy
3. Sort paintings by their vibrancy
4. Select paintings at index 0, 10, 20, and so on

With the vibrancy calculation algorithm being the following (based on this stackoverflow answer):
The find dominant color function used in the algorithm above was not specified, as we have mostly used existing library functions to do it.
It is done by loading the image, getting the colors from it, then sorting each color by the amount of

Algorithm 1 CalculateImageVibrancy(image)

$dominantColors \leftarrow FindDominantColor(image, 3)$
$vibrancyValues \leftarrow \emptyset$
for all $color \in dominantColors$ **do**
  $minVal \leftarrow min(color)$
  $maxVal \leftarrow max(color)$
  $vibrancy \leftarrow \frac{(maxVal+minVal)(maxVal-minVal)}{maxVal}$
  $vibrancyValues \leftarrow vibrancyValues \cup vibrancy$
end for
return $max(vibrancyValues)$

pixels it appears in and returning the top 3 colors.
By doing this, we are able to pick images with varying levels of vibrancy, as we take one every 10 images in the sorted array, starting from the most vibrant image to the least vibrant one, thereby creating some variance in the dataset our models use.

2.2.2 Cycle GAN design

We utilized the UNET architecture to construct the generator in our cycleGAN model. The generator takes as input a $320 \times 320 \times 3$ image, and outputs a $320 \times 320 \times 3$ image.

The model consists of a series of downsampling and upsampling layers, with skip connections between corresponding layers of the downsampling and upsampling stacks. Each downsampling layer reduces the spatial resolution by a factor of 2 while doubling the number of channels, using a 4x4 convolutional filter with stride 2 and applying instance normalization and leaky ReLU activation. Each upsampling layer performs the opposite operation, using a 4x4 transposed convolutional filter with stride 2, concatenating the output with the corresponding skip connection from the downsampling stack, and applying batch normalization and ReLU activation.
The final layer of the model is a 4x4 transposed convolutional filter with stride 2, a tanh activation function, and 3 output channels.

The discriminator takes as input a 320x320x3 image and applies downscaling three times, with 64, 128, and 256 filters respectively, each using a 4x4 kernel size. It then applies a zero-padding layer, followed by a 512-filter 4x4 convolutional layer, and an instance normalization layer. The output is then passed through a leaky ReLU activation function and another zero-padding layer before finally being processed by a 4x4 convolutional layer to produce a single-channel output with dimensions of 30x30x1.

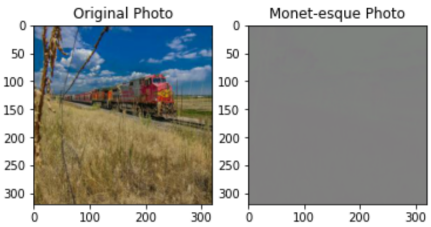
The cycleGAN model includes two generators and two discriminators. During training, the model takes in a batch of real images from both domains (monet and photo) and generates fake images in both domains using the generators. These generated fake images are then fed into their corresponding discriminators to determine if they are real or fake. This process is repeated until the generators are able to generate realistic images in both domains.

To train the model, four different loss functions are used. These losses work together to train the generators and discriminators and ensure that the generated images are both realistic and similar to the input images.

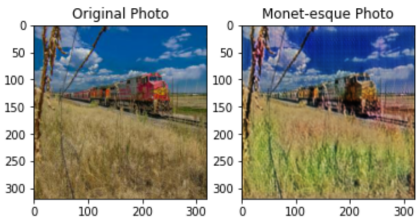
The generator loss function is used to train the generators to generate realistic images. This loss function is calculated by comparing the output of the discriminators when fed with the generated images to the label of "real" images. The generator is updated based on the gradients of this loss function, thereby improving its ability to create realistic images.
The discriminator loss function is used to train the discriminators to distinguish between real and fake images. This loss function is calculated by comparing the output of the discriminator when fed with real images and the output of the discriminator when fed with generated images. The discriminator is updated based on the gradients of this loss function, improving its ability to distinguish between real and fake images.
The calculation cycle loss function is used to ensure that the generated images are similar to the input images in both domains. This loss function calculates the difference between the input image and the image generated by passing the generated image through the other generator. This loss is then used to update the generators, ensuring that they generate images that are similar to the input images.
The identity loss function is used to ensure that the generator does not change the input image too much when generating the output image. This loss function calculates the difference between the input image and the image generated by passing the input image through the generator. The generator is updated based on the gradients of this loss function, ensuring that the output images are not too different from the input images.

The train step method executes one training step and returns the losses for the various components of the model. During each training step, the losses for each component are calculated and used to update the weights of the generators and discriminators using the corresponding optimizers.

Before training the network, its results looked like this:



After training the network, its results looked like this:



2.2.3 Simple GAN design

The simple GAN is composed of 1 generator and 1 discriminator, and was implemented using Keras (this was done long before the usage of NST was approved).

While the generator's architecture varied depending on the experiment, the discriminator's architecture was always that of a convolutional neural network.
In depth details about the architectures will be provided in section 3.2.

The training of this model was lengthy, depending somewhat on the architecture and also heavily on the number of epochs used and the batch size.
As the batch size was kept to a constant 32 during the training, the main effect was from the number of epochs and the GPU used, resulting in training taking between 3 minutes to 45 minutes.

One of the major technical challenges was running out of memory - the models required a lot of GPU RAM, which made them challenging to design and train.
Due to this issue, one of the models was trained on a premium GPU offered by Google Colab.
The loss function used was always binary cross-entropy, as the discriminator was always differentiating between 2 classes.
The optimizers used were RMSProp and Adam - once again, more details will be provided in section 3.2.

3 Experimental results

3.1 Cycle GAN experiments

3.1.1 First improvement attempt - optimizers

Initially, we designed an experiment to see whether the SGD optimizer would outperform the Adam optimizer.
To achieve this, the code looped through common parameter combinations for the SGD optimizer and trained the CycleGAN model for 5 epochs using each combination. After training, the best combination of parameters was selected based on the quality of generated images and convergence speed of training.
Once the best combination of parameters was identified, the CycleGAN model was trained for 20 epochs using this combination of parameters and compared to the results obtained using the Adam optimizer.
The resulting losses after training the model with the SGD optimizer:

```
Epoch 20/20
30/30 [==============================] - 11s 375ms/step - monet_gen_loss: 6.6756 - photo_gen_loss: 6.7790 - monet_disc_loss: 0.6532 - photo_disc_loss: 0.6647
```

The resulting losses after training the model with the ADAM optimizer:

```
Epoch 20/50
30/30 [==============================] - 12s 392ms/step - monet_gen_loss: 3.3971 - photo_gen_loss: 3.3934 - monet_disc_loss: 0.5758 - photo_disc_loss: 0.5758
```

The experiment showed that the Adam optimizer performed better than the SGD optimizer with momentum for training the CycleGAN model. As a result, the Adam optimizer was chosen for the final training of the model.

3.1.2 Second improvement attempt - changing batch size

Initially, we evaluated the model using a batch size of 1, which provided a good basis for our analysis. However, we aimed to study the impact of different batch sizes on the model's performance. we attempted to adjust the batch size to 30 and later 10. Unfortunately, both of these attempts caused RAM issues on the Colab machine. As a result, we settled on a batch size of 5 for further analysis. The result of changing the batch size:

The resulting losses after changing the batch size to 5:

```
Epoch 20/20
6/6 [==============================] - 7s 1s/step - monet_gen_loss: 4.2014 - photo_gen_loss: 4.3692 - monet_disc_loss: 0.6427 - photo_disc_loss: 0.5973
```

Although the result looked good when we generate monet picture, we observed that using a batch size of one resulted in lower loss. Consequently, we attempted to repeat the experiment with a batch size of two, but even with this larger batch size, the best performance was still achieved with a batch size of one. Therefore, we decided to stick with the latter.

3.1.3 third improvement attempt - learning rate

After defining the batch size and selecting the ADAM optimizer, our next step was to determine the optimal parameters for the learning rate and beta of the optimizer. To accomplish this, we conducted an experiment where we used a combination of recommended parameters from the internet along with some of our own. We systematically tested all possible combinations of learning rate and beta for 5 epochs and identified the best pair. Once the optimal pair was identified, we trained the model and obtained our results.
Thus, the resultingloss when the learning rate is changed:

```
Epoch 20/50
30/30 [==============================] - 12s 389ms/step - monet_gen_loss: 3.3361 - photo_gen_loss: 3.3826 - monet_disc_loss: 0.5635 - photo_disc_loss: 0.5535
```

Here is the generated monet picture after changing the learning rate:

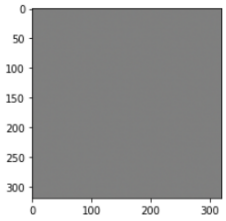
The experiment's findings clearly indicate that the ideal values for the learning rate and beta parameters are 0.0002 and 0.6, respectively. Employing these values resulted in the best training outcomes for the model.

3.2 Simple GAN experiments

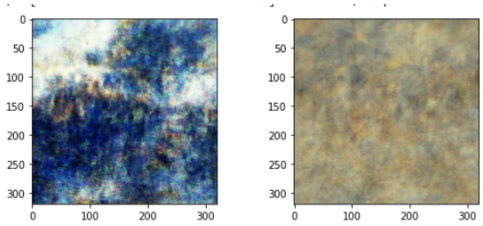
3.2.1 Base model

The first model that we created is our base model - a fairly simple model meant to be improved on in the future.
The model accepts as input 320x320 images (images not of that size before being input into the model are re-sized via our code), and returns as output 320x320 images.
The base model's generator features some fully connected layers, followed by a few convolution layers, with leaky ReLU as the activation function between each layer.
Its discriminator is a very simple convolutional network, with few convolution layers and a single dense layer, without any pooling layers.

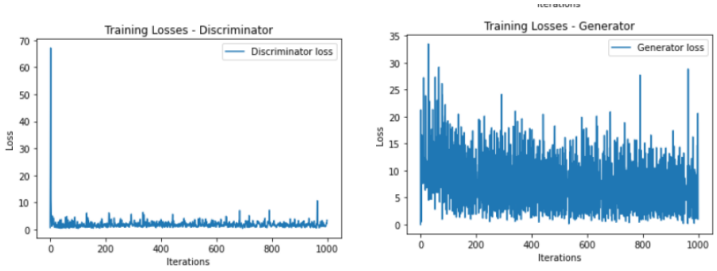
To decide on the optimizer and hyper-parameters of the base model, we let Github Copilot decided- as it is likely to use values used commonly by others, and ones that should fit the given task.
As such, we ended up using the RMSProp optimizer, a learning rate of 0.0008 for the discriminator and a learning rate 0.0004 for the generator.
The other hyper-parameters, the number of epochs and batch size, are ones we set ourselves.
We set the epoch count to 1000 epochs, due to GANs being notoriously hard to train as well as us wanting to see if the loss graph will show a an epoch count that could help with future models.
The batch size is a standard batch size of 32.
Before training the network, its results looked like this:



The training method used for the model first creates noise from a uniform distribution, between -1 and 1, and then uses that noise as input to generate a batch of images from the generator.
Then, the discriminator is trained on these fake images as well as real images, where their labels are "soft" labels instead of hard labels.
Afterwards, noise is generated again, and this time given to the generator model to train on, this time the labels all being 1 and the discriminator used but not trained.
Both the generator and discriminator are trained on a single batch with every iteration.
After training the model, it produced images that looked like this (full set of 32 images can be seen in the colab notebook):



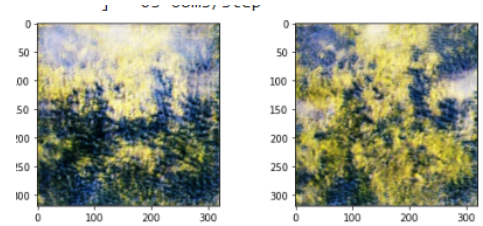
As can be seen, while it does not look like more than blotches of color, the style of the Monet paintings is there - indicating that the model is able to pick up on the style, at the very least.
Its loss graph was as follows:



As can be seen, the models were both seeing large fluctuations in their loss values.
This is likely (or at-least partly so) to the discriminator not being able to pick up on features due to it being too simple, making it so neither model was able to learn what it should pick up on.
In addition, from both models it seemed like mostly the same pattern was repeated with their losses from around the 200th iteration, and so we lowered the number of epochs to 300 (200 + safety margin) for future experiments.

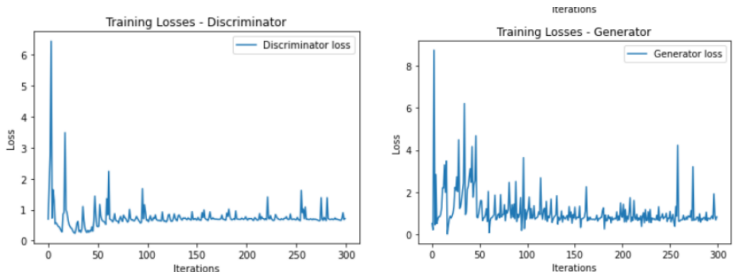
3.2.2 First improvement attempt - better discriminator

The base model's discriminator is an incredibly weak one - it only uses 2 convolution layers and a single dense layer, without using pooling of any sort.
As such, for this attempt, we improved the discriminator's architecture by adding more convolution layers with more filters, as well as max pooling layers.
And indeed, the results show:



This time, not only is the style there, but the results look much more like paintings of flowers, ones that can possibly pass as paintings drawn by a human, which is a big improvement.
However, when looking at the image set as a whole, it can be seen that the model generally only

generated a single type of photo - which could be a result of overfitting, a discriminator overpowering the generator, not enough data, or any other reason.
The loss graph of this model is:

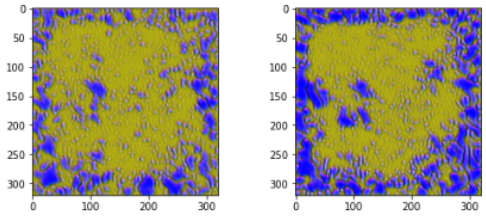


Showing that once again, the models did not converge.
However, there is a lot less noise this time - both the generator and discriminator have much less fluctuation to them, suggesting an improvement in this regard as well.

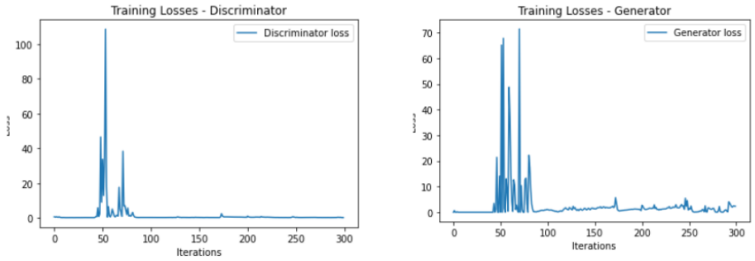
Overall, this experiment makes things seem like they are on the right track.

3.2.3 Second improvement attempt - different generator architecture and optimizer

After improving the discriminator, we next aimed to improve the generator.
We changed its architecture to include transpose convolution layers and batch normalization layers as well, as well as generally having more convolution layers and less dense layers.
Due to memory constraints, after every upsampling of the image, we had to downsample it back down (any attempt to upsample further caused the code and possibly the session to crash).
In addition, we also switched to using the Adam optimizer, as Adam should simply be a better optimizer overall.
Moreover, due to the memory requirements for this, we had to change the training process - we could no longer feed the model batches of 32 each time.
For creating fake images for the discriminator, we simply had the model predict twice then appended those to a single batch.
For training the model, we had to cut the batch size of it in half, making it so the model trains on 16 inputs per batch instead of 32.
This was kept this way instead of ran twice to try and also train the discriminator more, as seen in the advice in the lecture slides.
After 300 epochs, the results of the model are:



In a word - not good.
The current results look nothing like a Monet painting, and seem more like noise.
The loss graph is:



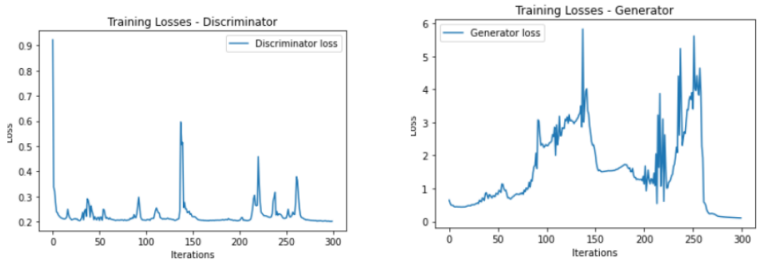
As can be seen, while both the generator and discriminator started with their loss values fluctuating heavily, they seemed to almost converge later on.
Seeing this and the previous results, we decided to train the model some more, the results of which can be seen in the notebook.
However, no real improvement was achieved from doing so.

Overall, this architecture was a failure.

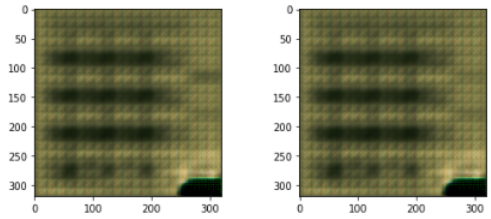
3.2.4 Third improvement attempt - patch GAN and emulating U-net

Note: There are more improvement attempts in the Colab notebook, and this is in fact the 6th one in the notebook, and those mostly focused on attempting regulurization methods to improve the model's possibly severe case of overfitting.
However, due to space constrictions (10 page limit), these were cut out from the report.

Our own attempts at making a good architecture did not work out too well.
Therefore, we have decided to follow guides from here and here, detailing how to use and create a patch GAN, as well as creating an architecture resembling U-net - a known and very good network architecture for generating images, where we first downsample the images then upsample them.
The discriminator's architecture is of a patch GAN, with a receptive field of 96×96.
It produced the following loss graphs:



Showing that there were many fluctuations and a lot of variance in the training process, but with overall fairly low loss values compared to some of the other models, which made it look good and like a potential improvement.
However, its results are:



Overall, not only do the results look far from even resembling the Monet style, the model generated an entire batch of images that look nearly identical.

Overall, this model too was a complete failure for us.

4 Discussion

From this project, we have learnt to implement GAN architectures, as well as learnt about the importance of the amount of data we have when training a model.
As can be seen, while our cycle GAN which was trained using 30 paintings and about 7000 photos produces good looking results, our simple GAN which only used 30 paintings produced results which were not satisfactory in the slightest.
In addition, we have also learnt about dealing with challenging restrictions - namely the amount of memory we can use and how to work alongside it.
From our experiments, we have learnt about the impact that the architecture and hyper-parameters used can have on the final results of the model.
Overall, we were able to learn a lot about both the practical aspects of implementing a GAN as well as how the theory behind neural networks can be implemented in high level usages such as these.

5 Code

Link to the colab notebook for the training environment.

Link to the colab notebook for the test environment.