Deep Learning- Final Course Project

Gili Gutfeld (ID. 209284512), Osher Elhadad ID. 318969748)

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

1 Introduction

Our project involves utilizing deep learning techniques to create Monet-inspired images from photographs. To achieve this, we acquired a dataset from 'kaggle' containing both Monet's artworks and regular photographs. We then proceeded to train multiple models using this dataset. The project is composed of two phases: training and testing, and it explores two distinct image style transfer architectures: cycleGAN and neural style transfer.

1.1 Related Works

We explored a more specific cycleGAN architecture proposed by Amy Jang, a contributor on Kaggle, which yielded promising results -

https://www.kaggle.com/code/amyjang/monet-cyclegan-tutorial. Also we read a tutorial of CycleGan -

https://www.tensorflow.org/tutorials/generative/cyclegan

2 First Solution - CycleGan Model

2.1 General approach

To generate Monet-style images from photographs, we employed a cycleGAN as our primary approach. This architecture incorporates two generators and two discriminators. The first generator converts photographs into Monet-style images, while the second generator performs the opposite transformation. Meanwhile, the two discriminators strive to differentiate between genuine and synthetic images within their respective domains.

2.2 Design

We utilized advanced techniques such as residual blocks used two convolutional layers with instance normalization and a ReLU activation function, and a shortcut connection that adds the input feature map to the output of the second convolutional layer. This allows the network to learn identity mappings and

improve the performance of deep neural networks, Used in the Generator architecture. We also applied normalization techniques like batch and instance norm during the forward step. We tested different epochs, learning rates, weight decay and batch to improve our model and find the best hyperparameters.

Regarding the discriminator architecture, we used a PatchGAN architecture that is a variant of the standard GAN architecture that operates on small patches of the input image instead of the entire image. More specifically, it takes as input an RGB image of size 320x320x3 and outputs a patch of size 19x19x1. The receptive field of the PatchGAN discriminator is 94x94 (different from the usual 70x70 as requested). The discriminator consists of four convolutional layers with increasing depth, followed by a patch-based output layer. After passing through the four convolutional layers, the feature maps are fed into a patch-based output layer that consists of a single 1x1 convolutional layer with a stride of 1. The output of this layer is then passed through a sigmoid activation function to produce a probability between 0 and 1 for each patch. Finally, leaky ReLU activation function with a negative slope of 0.2 is applied to all convolutional layers except the last one, which uses the sigmoid activation function.

The generator takes an RGB image of size (320x320x3) as input and generates an RGB image of the same size as output, and consists of three main parts: an encoder, a series of residual blocks, and a decoder.

The encoder downsamples the input image and extracts features from it. It consists of four convolutional layers with a kernel size of 4, stride of 2, and padding of 1. The number of output channels for each layer is doubled compared to the previous layer, starting from the initial number of channels set by the "conv-dim" parameter (which is set to 64 by default). Instance normalization is applied after the second, third, and fourth convolutional layers. After the last convolutional layer in the encoder, the output feature map size is (20, 20, 512).

The series of residual blocks is used to refine the features extracted by the encoder. Each residual block has two convolutional layers with a kernel size of 3, stride of 1, and padding of 1. Instance normalization is applied after each convolutional layer. The residual block takes the output of the previous block as input and adds it to the output of the second convolutional layer before applying the ReLU activation function.

The decoder upsamples the refined features to generate the output image. It consists of four transposed convolutional layers with a kernel size of 4, stride of 2, and padding of 1. Instance normalization is applied after each transposed convolutional layer, except for the last one. The output of the last transposed convolutional layer is passed through a hyperbolic tangent activation function to generate the final output image. Overall, this architecture follows the U-Net architecture which combines the contracting path (encoder) and the expanding path (decoder) of a neural network.

The generator architecture posed major challenges for us. Limited GPU RAM meant we had to select architectures that were expressive but not too complex, avoiding tensors that exceeded the capacity of the RAM. Furthermore, some of the chosen architectures generated images that did not closely resemble

the content image or had lighting issues in some white or black areas.

To overcome these issues, we tried adjusting variables such as the receptive field of neurons in deeper layers, the model's expressivity, and generalization ability by altering the number of epochs, architecture, learning rates, weight decay and batch, as described in the experiments themselves.

2.3 Smart Select Paintings

We should choose only 30 monet paintings to train our model. We chose an algorithm that calculate the colorfulness score for each image using the saturation and hue values of each pixel. We first convert the image to RGB and normalize the pixel values to be between 0 and 1. Then we calculate the saturation as the average of the difference between the maximum and minimum pixel values across all three color channels. We calculate the hue by dividing the pixel values by their sum across the three color channels and taking the standard deviation of these normalized values. Finally, we calculate the colorfulness score as the product of saturation and hue.

We then sort the images by their colorfulness scores and select the top 30 most colorful paintings. We return a dictionary of selected filenames with keys from 0 to 29.

We chose these 30 images because we wanted the most colorful paintings, so then we can do augmentations like brightness or rotations (0, 90, 180, 270) to generate more versatile images and to generalize better. We generated 16 augmentations for every painting of the top 30 most colorful paintings.

3 Second Solution - NST Model

3.1 General approach

Neural style transfer is our second solution, and it differs from other architectures in that there is no actual training phase. Instead, the process involves gradient ascent of the combination image to match the results of the base and style images from forward propagation through the model.

Therefore, the training phase in this architecture is focused on conducting experiments with various architectures and hyperparameters to determine the optimal formula and identify one typical style image. By doing this, we can apply neural style transfer on-the-fly during the test phase, enabling us to implement the process on any image with ease.

Our approach employs a pre-trained VGG-19 model to perform neural style transfer, a technique that blends the content of one image with the style of another to generate new images. VGG-19, a convolutional neural network with 19 layers, serves as the backbone of our model architecture. To achieve our objective, we utilize an iterative process wherein we receive two input images, a content image and a style image, at each iteration. We then adjust the pixel values of the content image iteratively while minimizing the loss function.

3.2 Design

The neural style transfer model utilized in this experiment is built on the convolutional neural network architecture of the VGG19 classification model. Consisting of two key components, the model involves the content loss and style loss, both of which operate during each iteration.

The trained network processes the base, style, and combination images, and then assesses the differences between the tensors in the deeper layers. The content loss is responsible for measuring the difference between the feature representations of the combination image and the content image, observed in the last tensor. On the other hand, the style loss measures the difference between the Gram matrices of the feature representations of the combination image and the style image using Mean Squared Error (MSE) loss.

The loss function is a weighted sum of both the content loss and style loss. Utilizing Python with Keras and TensorFlow libraries, we crafted our code in Google Colab notebooks. As with any technical project, we encountered several challenges, such as the extended training time for our model. Additionally, we addressed the need to resize our images from [256, 256, 3] to [320, 320, 3] while also ensuring proper normalization.

4 Experimental results - CycleGan

4.1 Base Model- Experiment 1

learning rate= 0.0002, weight decay= 0 (default), Adam optimizer with beta1=0.5 and beta2=0.999, 300 epochs, batch size= 16.

The rationale behind this choice of hyperparameters is as follows:

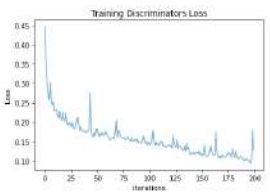
Learning rate: The learning rate determines the step size of the optimization algorithm during training. A larger learning rate can lead to faster convergence, but may also cause the optimization to overshoot or oscillate around the optimal solution. A smaller learning rate can ensure more stable and accurate updates, but may also require more training time. In this case, a learning rate of 0.0002 is chosen, which is a common default value for GANs.

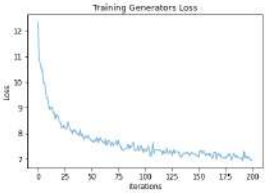
Beta coefficients: The beta coefficients control the decay rates of the moving averages of the gradient and its squared gradient used by the Adam optimizer to compute the adaptive learning rates. A smaller beta1 value, typically between 0.1 and 0.3, can help to smooth out the gradient estimates and reduce the sensitivity to noise, while a larger beta2 value, typically close to 1, can help to estimate the variance of the gradients and improve the convergence. In this case, a beta1 of 0.5 and a beta2 of 0.999 are chosen, which are also common default values for Adam.

Optimizer: The Adam optimizer is a popular choice for GANs due to its adaptive learning rates, momentum, and second-order moment estimation. It can handle sparse gradients, non-stationary objectives, and noisy or large datasets. The optimizer is applied separately to the generator and discriminator networks, with different learning rates and beta coefficients.

Epochs and batches: The total number of epochs and batches is determined by the size of the training dataset and the batch size used during training. The epochs are multiplied by the number of batches that is set to 16 per epoch to obtain the total number of iterations. In this case, the number of epochs is set to 300, which is a common value for CycleGANs.

We can see that the generated paintings are not as good as we wanted (bright or black and red spots) and the losses are- discriminator total loss avg: 0.1265, generator total loss: 6.9434.





4.2 Experiment 2

learning rate= 0.0001, weight decay= 0.0001, Adam optimizer with beta1=0.5 and beta2=0.999, 200 epochs, batch size= 16

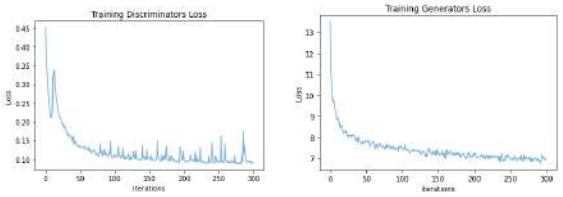
The rationale behind this choice of hyperparameters is as follows:

Learning rate: A lower learning rate helps to prevent the optimizer from taking large steps that may overshoot the optimal solution or get stuck in a suboptimal one. This is particularly important in GANs, where the generator and discriminator are competing against each other and the optimization landscape can be highly non-convex and difficult to navigate.

Weight decay: A higher weight decay, or L2 regularization, penalizes large weights in the model and encourages it to have a simpler and smoother solution. This can help to prevent overfitting and improve generalization, especially when the number of training samples is limited.

Number of epochs: Increasing the number of epochs can help to further improve the quality of the generated images, but may also lead to overfitting if the model is too complex or the dataset is too small.

We can see that the generated paintings are better than experiment 1 but not as good as we wanted (still some bright spots) and the losses are similar- discriminator total loss avg: 0.0892, generator total loss: 6.9602.



4.3 Experiment 3

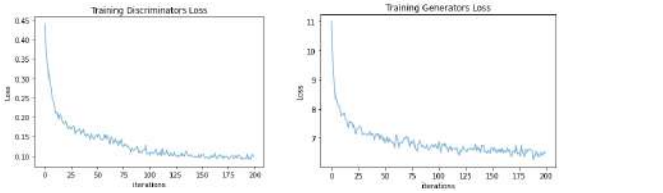
learning rate= 0.0002, weight decay= 0.0001, Adam optimizer with beta1=0.5 and beta2=0.999, 200 epochs, batch size= 8.

Changed the generator model to that architecture: The generator takes an RGB image of size (320x320x3) as input and generates an RGB image of the same size as output. The generator still consists of three main parts: an encoder, a series of residual blocks, and a decoder. The encoder downsamples the input image and extracts features from it. It consists of four convolutional layers with a kernel size of 9, 6 and 4, stride of 1, 2 and 2, and padding of 1. The number of output channels for each layer is doubled compared to the previous layer, starting from the initial number of channels set by the "conv-dim" parameter (which is set to 64 by default). Instance normalization is applied after the second, third, and fourth convolutional layers. After the last convolutional layer in the encoder, the output feature map size is (78, 78, 256). The number of residual blocks is set by the "n-res-blocks" parameter, which is set to 6 by

default. The decoder upsamples the refined features to generate the output image. It consists of four transposed convolutional layers with a kernel size of 4, 6 and 9, stride of 2, 2 and 1, and padding of 1.

We added more parameters (bigger kernel size), and added stride 1 to the first conv layer, in order to add expressivity to the model by having more parameters to learn and less loss of information from the input image. We also change the number of batches to be 8 (lower than 16) to update the weights more frequently and to converge faster.

We can see that the generated paintings are better than experiment 1 and 2, and the losses better- discriminator total loss avg: 0.0990, generator total loss: 6.4792.

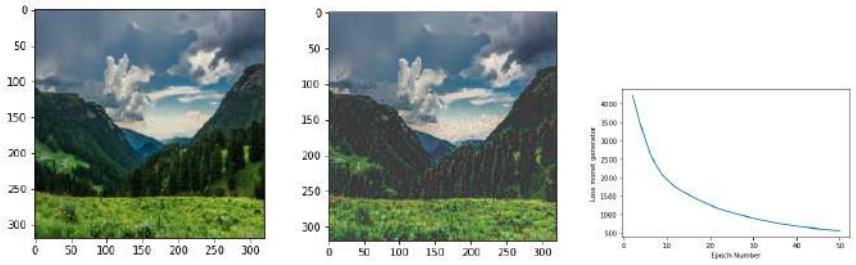


5 Experimental results - Neural Style Transfer

To assess the efficacy of each model, we employed the FID score, typically utilized to evaluate the caliber of images generated by generative adversarial networks. The lower the FID, the higher the image quality. Additionally, we visually inspected several generated images and selected the most exceptional one. After several experiments, the optimal model emerged from experiment 4, and we concluded that it outperforms the cycleGAN model. However, this model necessitates learning in each generation, whereas the cycleGAN produces images via a single forward phase, leading to faster processing times.

5.1 Base model

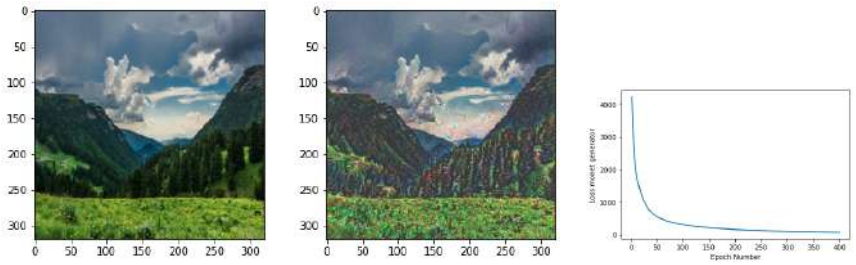
We performed several experiments and each time we changed one of the parameters in order to find the optimal value for it and finally we chose the model according to the best parameters we found. In the first experiment, we built a base model in which the number of epochs is 50, the learning rate is 0.01, the weight is 0.01, the weight decay is 0.01 and the beta1 of Adam Optimizer is 0.99.



The loss function demonstrates a smooth, steady decrease as the number of epochs increases. Moreover, the outcome image shows good quality. The loss is 555.80.

5.2 Experiment 1

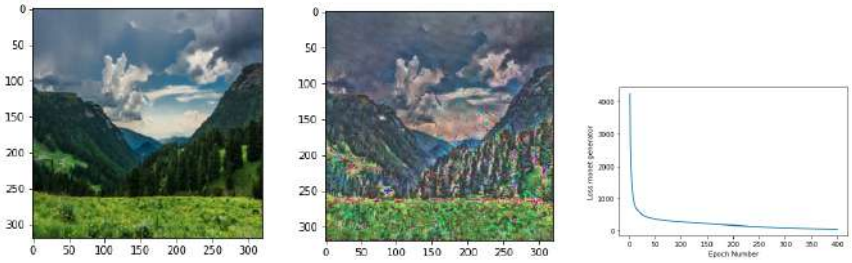
In the next experiment we tried to increase the number of epochs and there are the results.



The loss function demonstrates a smooth, steady decrease as the number of epochs increases. Moreover, the outcome image shows better quality as expected. It is evident that a significant augmentation in the number of epochs has a beneficial impact on the results. The loss is 76.31.

5.3 Experiment 2

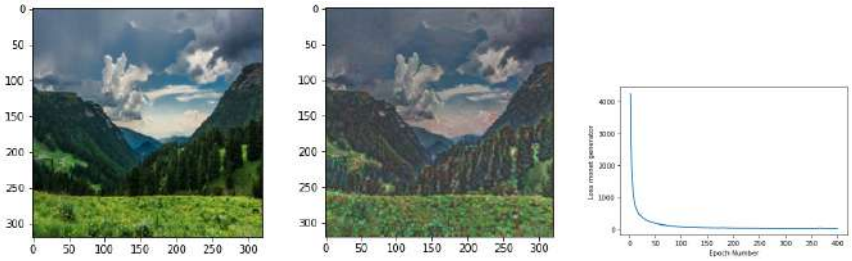
In the second experiment we tried to find the ideal learning rate. First, we decreased it and found that the loss was going down too slowly and when we raised it, we ended up choosing 0.08 in order to avoid distraction.



The loss function demonstrates a smooth, steady decrease as the number of epochs increases. Moreover, the outcome image shows excellent quality as expected. The change in the learning rate has a beneficial impact on the results. The loss is 53.89.

5.4 Experiment 3

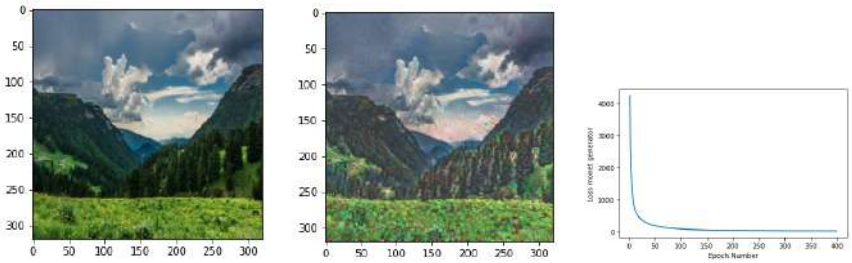
Next, we tried to find the ideal beta1. Finally we chose 0.95.



The function is less smooth now, as we give a higher weight to each new iteration than to the history, but the result looks even better. The change in the beta has a beneficial impact on the results. The loss is 27.35.

5.5 Experiment 4

Finally, we tried to find the ideal weight decay and we decreased it to 0.001.



The loss function now has a smooth, steady decrease as the number of epochs increases. Moreover, the outcome image shows excellent quality as expected. The change in the weight decay has a beneficial impact on the results. The loss is 27.82.

6 Discussion

Throughout the implementation of the cycleGAN and neural style transfer architectures, we gained valuable insights regarding the hyperparameter selection process and model architecture specifications. One key takeaway is the substantial difference in output quality between the two architectures. The cycleGAN, with its need for generalization and operation on a single content image during prediction, faces many challenges and is potentially sensitive to noise. Conversely, the neural style transfer technique is slower during prediction but produces more accurate results, as it can "overfit" the input images to the definition of the loss functions without the need for generalization.

Furthermore, we discovered that the cycleGAN's generator struggled to produce satisfactory results due to its limited selection of only 30 images and GPU limitations. To address this issue, we found that increasing the generator's receptive field and kernel size resulted in more detailed and expressive images. Additionally, we observed that state-of-the-art optimizers performed the best and should not be significantly altered when searching for better models.

We conducted a comprehensive investigation of two models to tackle the problem at hand. We performed a series of experiments to determine the optimal hyperparameters for achieving the best results. The best model was the second architecture (using VGG19) with 400 iterations, a learning rate of 0.08, weight decay of 0.001, beta1 of 0.95 and the Adam optimizer.

Overall, this project provided valuable insights into the selection and implementation of deep learning models, highlighting the importance of careful architecture and hyperparameter selection to achieve optimal results. We solved the problem using both approaches, but for the competition, we will present the second model since its results were superior, as we have illustrated.

7 Code

1. Train - https://colab.research.google.com/drive/1eo1SebGTWmZGYwd7NeUekpKRtKbnLta_?usp=sharing
2. Test - https://colab.research.google.com/drive/1Rxzt4qTrm6R4wO-U5xuPwPYqasfUy_Qz?usp=sharing