Deep Learning - Final Course Project

Sapir David, Shoval Habas

Submitted as final project report for the DL course, BIU, Semester A, 2022

1 Introduction

Image inpainting is a process of reconstructing missing parts in the image, the purpose of this process is to fill a missing data in a designated region of the visual input (an image for example). Our goal is to produce the missing region in such a way that a casual observer will not detect the differences between the missing part we've produced and the original missing part. Image inpainting has many applications, for example restoration of photographs, paintings, marks removal and more. In this project we have tested and explored the ability of computer vision to predict missing regions in a given image using convolutional neural networks by implementing context pixel prediction-based algorithms. In other words, we will present an algorithm implementation for an unsupervised visual feature learning to construct missing regions in a given image. The implementation and the main architecture of our model is based on the architecture presented in the article: 'Context Encoders: Feature Learning by Inpainting' by Deepak Pathak, Philipp Krähenbühl, jeff Donahue, Trevor Darrell.
This report includes testing of different masks, different mask sizes, testing and training on different data sets and testing of different hyper parameters. In addition, in this report we present different trails on our model and investigate the effect of such changes.

1.1 Related Works

[1] Deepak Pathak, Philipp Krähenbühl, Jeff Donahue Trevor Darrell, and Alexei A. Efros. Context Encoders: Feature Learning by Inpainting.

[2] Omar Elharroussa , Noor Almaadeeda, Somaya Al-Maadeeda , Younes Akbaria Image inpainting: A review.

[3] Ayush Thakur, Sayak Paul Introduction to image inpainting with deep learning.

[4] HuBMAP: Keras Augmentation Layers.
https://www.kaggle.com/hiramcho/hubmap-keras-augmentation-layers

[5] Thushan Ganegedara , Intuitive Guide to Neural Style Transfer.

[6] TThe PASCAL Object Recognition Database Collection.
http://host.robots.ox.ac.uk/pascal/VOC/databases.html

2 Solution

2.1 General approach

Data preparation: As we've mentioned above, our goal in this project is to generate a patch to fill in the missing area of an image, we were instructed to use two datasets from Kaggle competition - 'Photos' and 'Monet'. To train and evaluate each of the datasets given, we will use masks. Masks are binary images consisting of 0 and 1 values (1 stands for a dropped pixel), corresponding to the missing part of the image. To test the ability of the model to reconstruct the missing area, we will test different types of masks:

- Center region
- Random block
- Random region

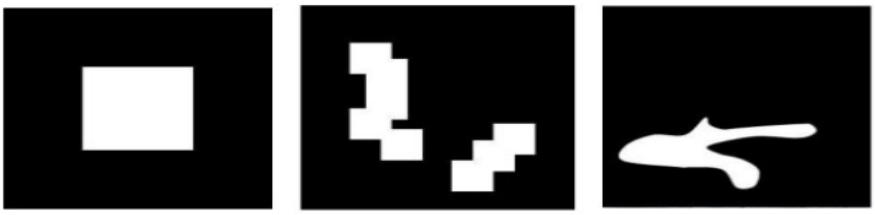


Figure 1: : Three types of masks - Center region, Random block, Random region

Architecture - Context Encoders: Context Encoders is an architecture used for image inpainting, which has proved to have a high potential, moreover it has been found that Context Encoders can learn not only data and appearances, but also the context of structures according to the paper above. Context Encoders is based on an autoencoder architecture (an encoder that maps the input into the code, and a decoder that maps the code to a reconstruction of the input) with an adversarial discriminator, while using convolutional layers with varying kernel sizes and channel counts.
Context Encoders architecture is a convolutional neural network trained to generate the contents of an arbitrary visual region conditioned on its surrounds and has to produce a reasonable hypothesis for the missing parts. As said above, the Context encoders model consists of a generator (the encoder-decoder) and an adversarial discriminator. A **generator** network takes an image with white pixels and the mask as input pairs, the generator initializes the model, which consists of encoder and decoder. The encoder consists of 2D convolutional operations, with a progressively increasing number of channels it captures the context of an image into a compact latent feature representation, and then the decoder uses this representation to provide the missing part (shown at figure 2 below).

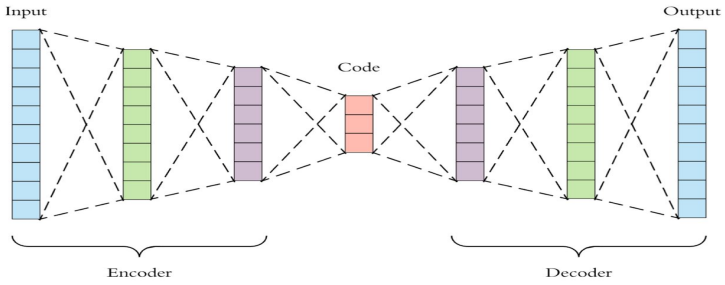


Figure 2: : Simple deep autoencoder architecture

At the training phase, we have trained the Context Encoders with reconstruction loss and adversarial loss (which creates sharper results) as recommended in the mentioned paper. There is an intermediate bridging layer between the encoder and the decoder, it is called the bottleneck layer and is meant to represent the encoded context of the image, hence the name Context Encoders. It is important to mention that the encoder and the decoder are evaluated independently. For the **discriminator** architecture we used patch loss as required, the discriminator has an image input and it is tasked with classifying it either as genuine data or an inpainting result, it tries to distinguish them, meaning the discriminator tries to distinguish real data and the data created from the generator, this is the networks output .

Augmentation: To train and evaluate the 'Monet' dataset we needed to add adjustments and also face a small dataset challenge. To solve the small dataset challenge we used augmentation to increase the amount of data by adding slightly modified copies of already existing data or newly created synthetic data from existing data (rotation, brightness, zoom, crop, flips and more).
For example:

Figure 3: Augmentation example.

2.2 Design

Platform and environment: Our implementation for the algorithm and the model is written in Keras. Keras is an open-source Python library that runs on top of tensorflow, it is used for developing and evaluating deep learning models. We ran our code with Google Colab Pro to use GPU performances. Using Google Colab allowed us to run our model with better GPU power than we have on our computers, this saved us time and gave us the opportunity to improve our model faster by viewing and evaluating the results faster. To learn and test the abilities of our algorithm, for each of the mask types we will train different models adjusted to the given mask so the prediction process will be as accurate as possible.
Architecture: Our top level architecture is based on the paper's architecture Context Encoders.
We used two different architectures for the datasets: For 'Monet' we created a similar architecture to the one in the paper. For 'Photos' dataset we used an adjusted version of 'AlexNet' for the encoder, the rest is similar to 'Monet' architecture.

The architecture for 'Photos' dataset is as follows:

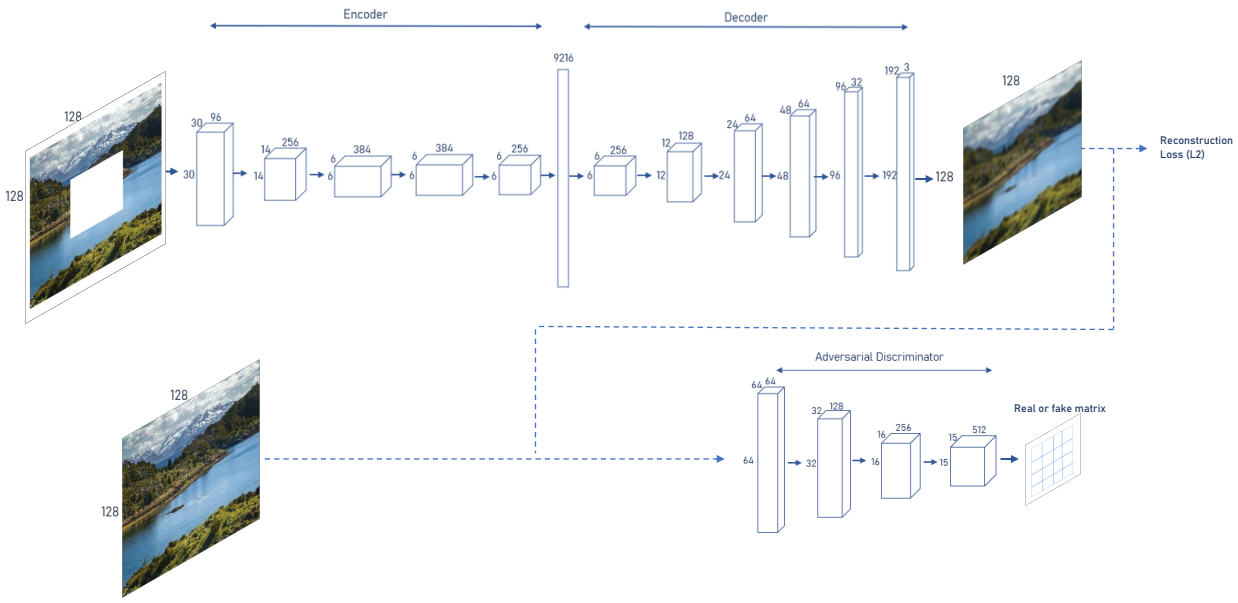


Figure 4: : Architecture for 'Photos' dataset.

Encoder:

Encoder Architecture				
Layer	**Output Shape**	**Kernel Size**	**Stride**	**Padding**
Convulsion layer - Conv2D	(30,30,96)	11x11	4	None
Normalization layer	BatchNormalizatio			
Activation layer	Relu			
Pooling layer- MaxPooling2D	(14,14,96)	pool size=(3, 3)	2	None
Convulsion layer - Conv2D	(14,14,256)	5x5	None	same
Normalization layer	BatchNormalization			
Activation layer- MaxPooling2D	Relu			
Pooling layer- MaxPooling2D	(6,6,256)	pool size=(3, 3)	2	None
Convulsion layer - Conv2D	(6,6,384)	3x3	None	same
Normalization layer	BatchNormalization			
Activation layer	Relu			
Convulsion layer - Conv2D	(6,6,384)	3x3	None	same
Normalization layer	BatchNormalization			
Activation layer	Relu			
Pooling layer- MaxPooling2D	(6,6,384)	pool size=(3, 3)	2	None
Convulsion layer - Conv2D	(6,6,256)	3x3	None	same
Normalization layer	BatchNormalization			
Activation layer	Relu			
Pooling layer- MaxPooling2D	(2,2,256)	pool size=(3, 3)	2	None

Bottleneck (FC):

Bottleneck Architecture				
Layer	**Output Shape**	**Kernel Size**	**Stride**	**Padding**
Core layers - Dense	9216	None	None	None
Activation layer	Relu			
Normalization layer	Dropout with rate=0.4			

Decoder:

Decoder Architecture				
Layer	**Output Shape**	**Kernel Size**	**Stride**	**Padding**
Reshaping layer - Reshape	(6,6,256)	None	None	None
Convulsion layer - Conv2DTranspose	(12,12,128)	4x4	2	same
Convulsion layer - Conv2DTranspose	(24,24,64)	4x4	2	same
Activation layer	Relu			
Normalization layer	BatchNormalization with momentum=0.8			
Convulsion layer - Conv2DTranspose	(48,48,64)	4x4	2	same
Activation layer	Relu			
Normalization layer	BatchNormalization with momentum=0.8			
Convulsion layer - Conv2DTranspose	(96,96,32)	4x4	2	same
Activation layer	Relu			
Normalization layer	BatchNormalization with momentum=0.8			
Convulsion layer - Conv2DTranspose	(192,192,3)	4x4	2	same
Preprocessing layer - Resize	(128,128,3)	4x4	2	same
Activation layer	tanh			

Discriminator Architecture:

Discriminator Architecture				
Layer	**Output Shape**	**Kernel Size**	**Stride**	**Padding**
Convulsion layer - Conv2D	(64,64,64)	4x4	2	same
Activation layer	LeakyRelu alpha=0.2			
Convulsion layer - Conv2D	(32,32,128)	4x4	2	same
Activation layer	LeakyRelu alpha=0.2			
Normalization layer	BatchNormalization with momentum=0.8			
Convulsion layer - Conv2D	(16,16,256)	4x4	2	same
Activation layer	LeakyRelu alpha=0.2			
Normalization layer	BatchNormalization with momentum=0.8			
Reshaping layers- ZeroPadding	(18,18,256)	None	None	None
Convulsion layer - Conv2D	(15,15,512)	4x4	1	valid
Activation layer	LeakyRelu alpha=0.2			
Normalization layer	BatchNormalization with momentum=0.8			
Reshaping layers- ZeroPadding	(17,17,512)	None	None	None
Convulsion layer - Conv2D	(14,14,1)	4x4	1	valid
Activation layer	Sigmoid			

The architecture for 'Monet' dataset is as follows:

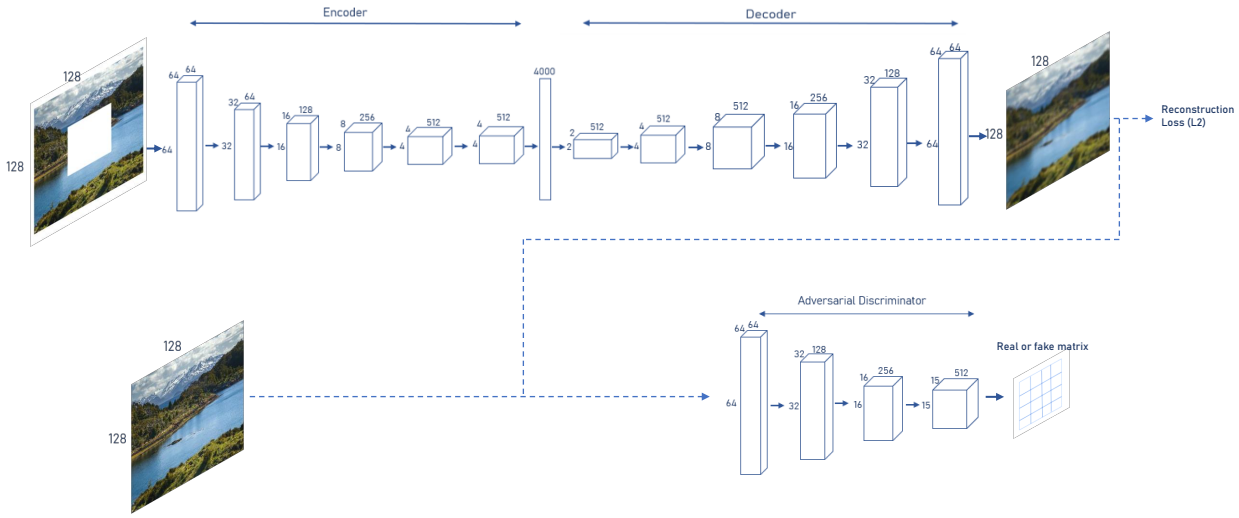


Figure 5: : Architecture for 'Monet' dataset.

Encoder:

Encoder Architecture				
Layer	**Output Shape**	**Kernel Size**	**Stride**	**Padding**
Convulsion layer - Conv2D	(64,64,64)	4x4	2	same
Activation layer	LeakyRelu alpha=0.2			
Normalization layer	BatchNormalization with momentum=0.8			
Convulsion layer - Conv2D	(32,32,64)	4x4	2	same
Activation layer	LeakyRelu alpha=0.2			
Normalization layer	BatchNormalization with momentum=0.8			
Convulsion layer - Conv2D	(16,16,128)	4x4	2	same
Activation layer	LeakyRelu alpha=0.2			
Normalization layer	BatchNormalization with momentum=0.8			
Convulsion layer - Conv2D	(8,8,256)	4x4	2	same
Activation layer	LeakyRelu alpha=0.2			
Normalization layer	BatchNormalization with momentum=0.8			
Normalization layer	Dropout with rate=0.5			
Convulsion layer - Conv2D	(4,4,512)	4x4	2	same
Activation layer	LeakyRelu alpha=0.2			
Normalization layer	BatchNormalization with momentum=0.8			
Normalization layer	Dropout with rate=0.5			
Convulsion layer - Conv2D	(4,4,512)	4x4	2	same
Activation layer	LeakyRelu alpha=0.2			
Normalization layer	Dropout with rate=0.5			

Bottleneck:

Bottleneck Architecture				
Layer	**Output Shape**	**Kernel Size**	**Stride**	**Padding**
Convulsion layer - Conv2D	(1,1,4000)	4x4	1	valid
Activation layer	LeakyRelu alpha=0.2			
Normalization layer	BatchNormalization with momentum=0.8			

Decoder:

Decoder Architecture				
Layer	**Output Shape**	**Kernel Size**	**Stride**	**Padding**
Convulsion layer - Conv2DTranspose	(2,2,512)	4x4	2	same
Convulsion layer - Conv2DTranspose	(4,4,512)	4x4	2	same
Activation layer	Relu			
Normalization layer	BatchNormalization with momentum=0.8			
Convulsion layer - Conv2DTranspose	(8,8,512)	4x4	2	same
Activation layer	Relu			
Normalization layer	BatchNormalization with momentum=0.8			
Convulsion layer - Conv2DTranspose	(16,16,256)	4x4	2	same
Activation layer	Relu			
Normalization layer	BatchNormalization with momentum=0.8			
Convulsion layer - Conv2DTranspose	(32,32,128)	4x4	2	same
Activation layer	Relu			
Normalization layer	BatchNormalization with momentum=0.8			
Convulsion layer - Conv2DTranspose	(64,64,64)	4x4	2	same
Activation layer	Relu			
Normalization layer	BatchNormalization with momentum=0.8			
Convulsion layer - Conv2DTranspose	(128,128,3)	4x4	2	same
Activation layer	tanh			

Discriminator:

Discriminator Architecture				
Layer	**Output Shape**	**Kernel Size**	**Stride**	**Padding**
Convulsion layer - Conv2D	(64,64,64)	4x4	2	same
Activation layer	LeakyRelu alpha=0.2			
Convulsion layer - Conv2D	(32,32,128)	4x4	2	same
Activation layer	LeakyRelu alpha=0.2			
Normalization layer	BatchNormalization with momentum=0.8			
Convulsion layer - Conv2D	(16,16,256)	4x4	2	same
Activation layer	LeakyRelu alpha=0.2			
Normalization layer	BatchNormalization with momentum=0.8			
Reshaping layers- ZeroPadding	(18,18,256)	None	None	None
Convulsion layer - Conv2D	(15,15,512)	4x4	1	valid
Activation layer	LeakyRelu alpha=0.2			
Normalization layer	BatchNormalization with momentum=0.8			
Reshaping layers- ZeroPadding	(17,17,512)	None	None	None
Convulsion layer - Conv2D	(14,14,1)	4x4	1	valid
Activation layer	Sigmoid			

- PatchGAN: is a type of discriminator for adversarial networks, it tries to classify if each patch in an image is real or fake. The difference between PatchGAN and regular discriminator is that rather the regular discriminator maps from a 128x128 image to a single scalar output which denotes "real" or "fake", where as the PatchGAN maps from 128x128 to NxN array of outputs. In the output array X, each X_{ij} signifies whereas the image is real or fake. Each X_{ij} in the output is a neuron in the convolution network and we can trace back its receptive fields to see which input pixels it is sensitive to. The receptive fields are defined portions of space or spatial construct containing units that provide input to a set of units within a corresponding layer. For example:

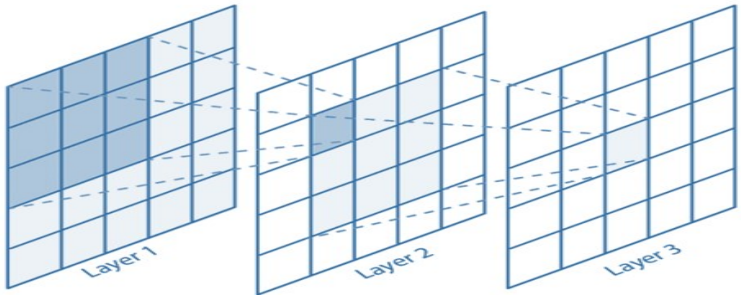


Figure 6: Receptive field visualization.

To calculate the size of the receptive field of the l-1 layer we use the following formula :

$$K_{l-1} = K + S(K_l - 1)$$

Where K=kernel size, S= stride, K_l= current layers receptive field,K_{l-1}= previous layers receptive field. Meaning we backtrace from each layer to its previous layer for the calculation.
For example, we will calculate the receptive field of the last layer before the output. The output size is 14x14, Kernel size is 4x4, stride is 1x1, K_l=1x1. Then we get according to the formula:

$$\text{rows: } K_{l-1} = K + S(K_l - 1) = 4 + 1(1 - 1) = 4$$
$$\text{columns: } K_{l-1} = K + S(K_l - 1) = 4 + 1(1 - 1) = 4$$

We've received that the receptive field of that layer is 4x4.

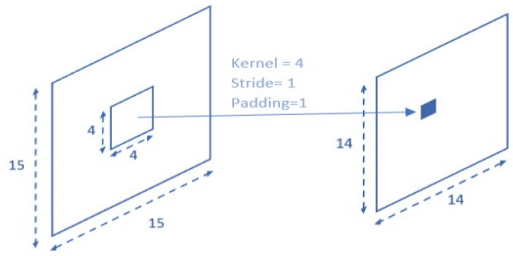


Figure 7: Demonstration of receptive field - part 1.

If we backtrace another layer back, according to the formula above, we'll get:

rows: $K_{l-1} = K + S(K_l - 1) = 4 + 1(4 - 1) = 7$

columns: $K_{l-1} = K + S(K_l - 1) = 4 + 1(4 - 1) = 7$

We've received that the receptive field of that layer is 7x7.

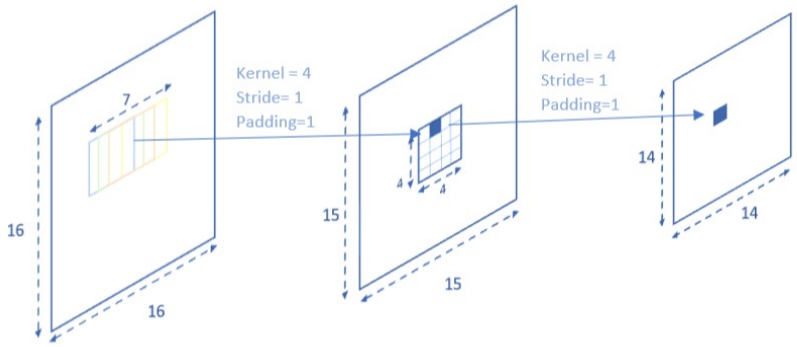


Figure 8: Demonstration of receptive field - part 2.

We have tried different architecture for PatchGAN discriminator which led to different receptive field sizes, eventually we chose as described above.

Training: We ran our model on 2 accounts of Google Colab Pro. Each epoch lasted about 1 minute, we trained each model for about 200 epochs (at the results section you will be able to see graphs. Those graphs represent train and validation, you can see that learning stops after 40-60 epochs) .
Optimizer: In the encoder and the decoder we use Adam optimizer with learning rate of 0.0002 and beta of 0.5.
Validation: We shuffled and divided our data set into 2 categories – train consists 80% of the data and the rest is validation. All training is performed in an unsupervised manner, meaning no class label information is supplied to the model and there is no conditioning for the evaluation. The process of validation assures that the model has achieved the knowledge it needs for the testing phase, in this process we compare the model output to the real outputs to examine our model output quality.

Loss Functions: To evaluate our model results, we used different loss function:

- Joint loss (reconstruction L2 and adversarial loss):
  We used the reconstruction L2 loss between the original data and the unpainted result, the results are blurry and very easily discarded as fake. This issue is alleviated by the use of the adversarial discriminator loss, which learns to identify features that are specific only to generated patches. For the loss formulas below we denote: x - ground truth image, $F(x)$ - output of context encoders, $\hat{M}$ - binary mask corresponding to the dropped image region with a value of 1 wherever a pixel was dropped and 0 for input pixels,$\odot$ is the element-wise product operation,$D(x)$ - output of discriminator.

$$\mathcal{L}_{rec}(x) = \|\hat{M} \odot (x - F((1 - \hat{M}) \odot x))\|_2^2$$

Figure 9: Reconstruction loss L2.

$$\mathcal{L}_{adv} = \max_{D} \quad \mathbb{E}_{x \in \mathcal{X}}[\log(D(x)) + \log(1 - D(F((1 - \hat{M}) \odot x)))]$$

Figure 10: Adversarial loss.

These two loss functions are joined together, meaning we use a joint loss composed of reconstruction L2 and adversarial loss (using only reconstruction L2 will produce a blurry missing part, in addition to L2 loss we use the adversarial loss which creates sharper results).
In addition, we use coefficients for each of the loss functions for tuning the influence of each of the losses.

$$\mathcal{L} = \lambda_{rec}\mathcal{L}_{rec} + \lambda_{adv}\mathcal{L}_{adv}$$

Figure 11: Joint loss.

As the paper suggested we gave weights to each loss function- 0.999 to the reconstruction function and 0.001 to adversarial loss, similarly here we tried to change the weights, but it did not produce better results.

- Style loss: In the 'Monet' dataset we need to take under consideration the style of the image, to do so we use style loss. The style loss ensures that the style picture and the generated image have equal activation correlations across all layers, meaning it keeps the generated image close to the local textures of the style reference image. In order to compute the style loss we use the gram matrix of image tensor, a gram matrix defines a style with respect to specific content. By calculating the gram matrix for each feature activation in the target/ground truth image, it allows the style of that feature to be defined.

$$L_{style} = \sum_l w^l L^l_{style} \text{ where,}$$

$$L^l_{style} = \frac{1}{M^l} \sum_{ij} (G^l_{ij}(s) - G^l_{ij}g)^2 \text{ where,}$$

$$G^l_{ij}(I) = \sum_k A^l_{ik}(I) A^l_{jk}(I).$$

Figure 12: Style loss and gram matrix.

2.3 Experimental results

2.3.1 Trails

- **Different image size:** the image input we receive is in size of 256 X 256. We have resized the image to 128X128 and then trained our joint loss with resized images, so the number of learned parameters will be smaller and will converge. We tried to stay with an image size of 256 X 256 to keep the original pixels the same, but it resulted in an enormous number of parameters. Due to this number of parameters, we did not succeed to run our model in Google Colab Pro.

- **Transfer learning:** when a model uses the knowledge learned from a prior assignment to increase prediction about a new task. Meaning that elements of a pre-trained model are reused in a new model. If the two models are developed to perform similar tasks, then generalised knowledge can be shared between them. In our project we investigated the option to use a famous neurons network called VGG16. This network was trained on ImageNet (a large visual database project used in visual object recognition) containing more than 14 million different images, its task is classification. We tried to use this network and its weights, meaning we tried to use the networks knowledge for a similar task to get better results in our model. Unfortunately this network contains too many parameters for us to use, we tried to run and adjust the network to our networks requirements and architecture but we were not able to run this model due to a large number of parameters (we tried with 2 different Google Colab Pro accounts).

- **Small data set in 'Monet':** in the 'Monet' directory there exist only 300 images. To solve this challenge we thought about 2 techniques - the first is augmentation as described in 'General Approach' paragraph, for augmentation options we randomize a value in the given range (as input) and in total generate a large amount of slightly different images to increase the data set (we created over 3000 images). The second technique is to use Neural Style Transfer (NST) with the 'Photos' directory to convert them to images in 'Monet' style. We have a large dataset of images in the 'Photos' directory and with NST we can transfer them into a 'Monet' style image. To do so we thought that for each photo from 'Photos' we will randomly choose an image from 'Monet' and create an image in 'Monet' style, by doing so we will increase the data set. Eventually we chose the first option due to time limitations.

- **Joint loss weights:** in the article the researchers used 0.999 and 0.001 for reconstruction loss and adversarial loss correspondingly. We have tested different weights, for example 0.994 for the reconstruction loss and 0.006 for the adversarial loss. We did not achieve a better result, thus we decided to stay with those weights.

- **Hyper parameters:** we've tried to increase and decrease the learning rate between the range of 0.0001 and 0.002 but the best results (in relation to time and quality) were as we mentioned in the design section. We've tested other optimizers such as Adaelta, RMSprop and more, but we learned that Adam is more efficient and works well with noise.

- **Dropout:** Dropout is a regularization technique. We used a dropout layer which randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting. we have tested different rates for the dropout layer, in range of 0.2 to 0.5.

- **Number of layers:** at the beginning we build our network as the article described, after testing various numbers of layers and its sizes, under the understanding that we want to improve the learning and its results but also, we don't want a "heavy" network with a lot of parameters.

- **Normalization:** we have tested two normalization techniques, the first is pixel normalization by 255, and the second is Min-Max normalization, due to no significant change in the results we decided to stay with the first.

2.3.2 Results

Center region - 'Photos' dataset:

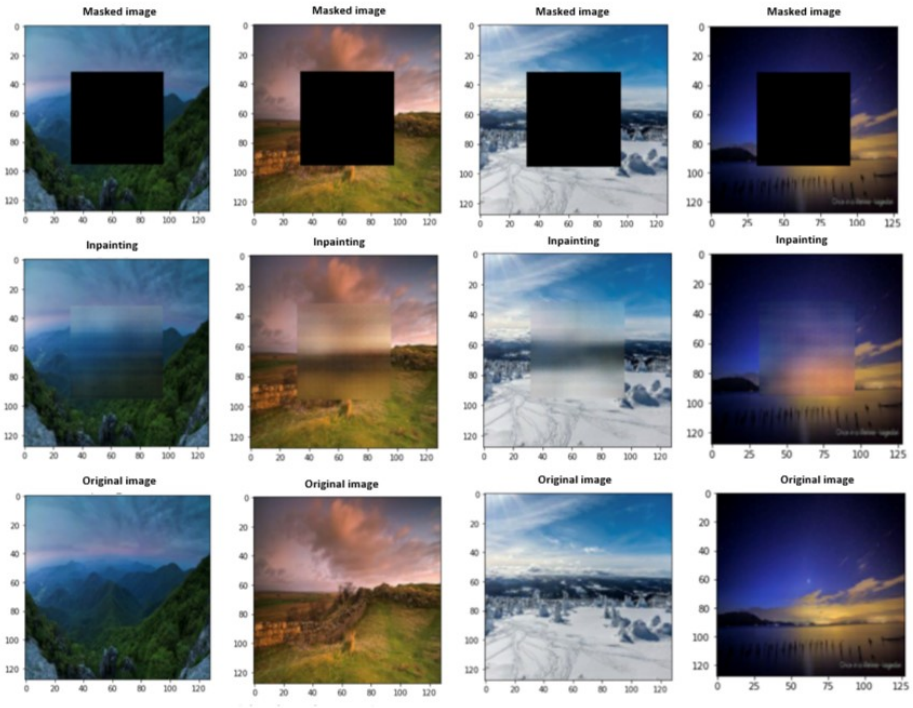


Figure 13: Visual results of Center region.

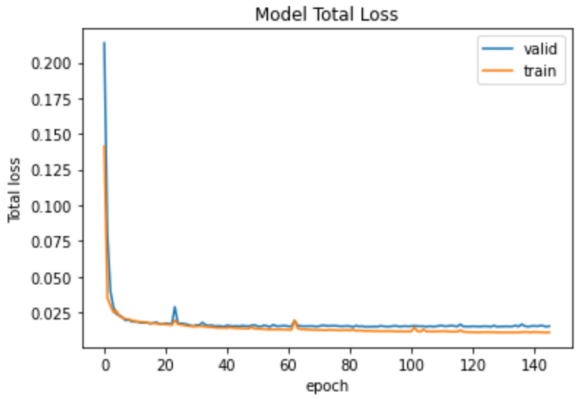


Figure 14: Graphic results of Center region.

Random block - 'Photos' dataset:

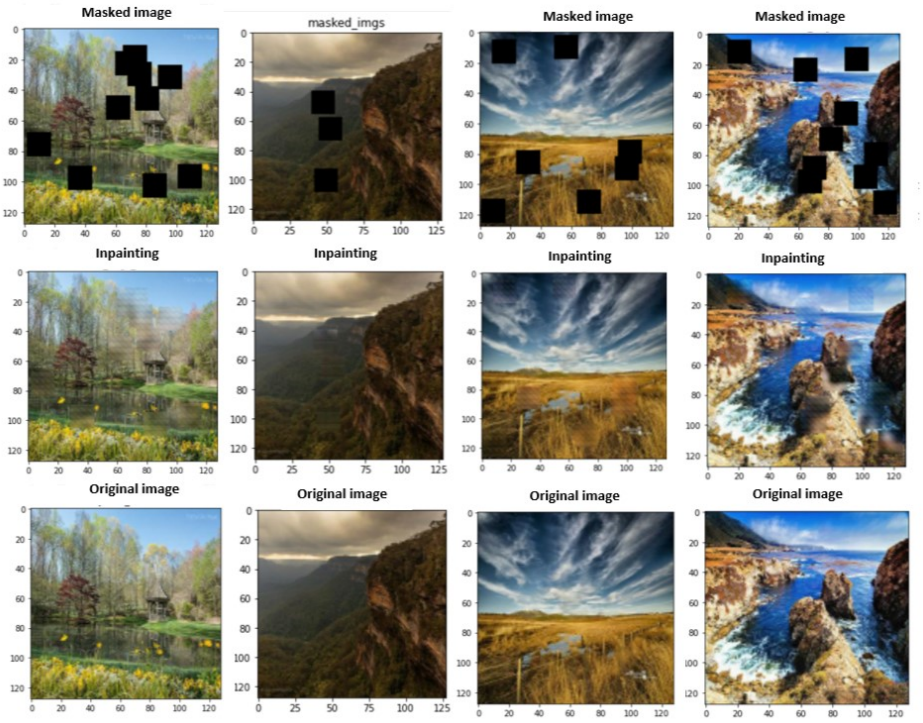


Figure 15: Visual results of Random block.

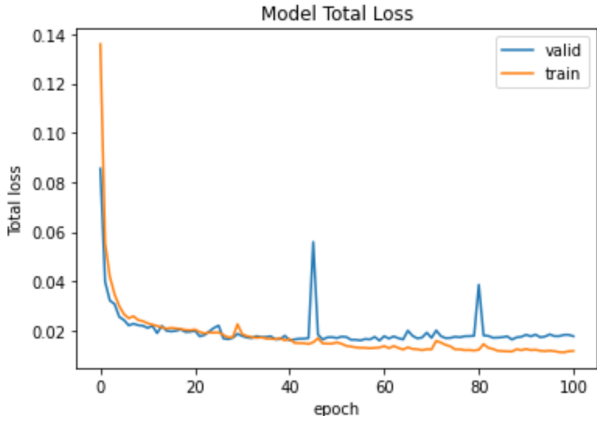


Figure 16: Graphic results of Random block.

Random region - 'Photos' dataset:

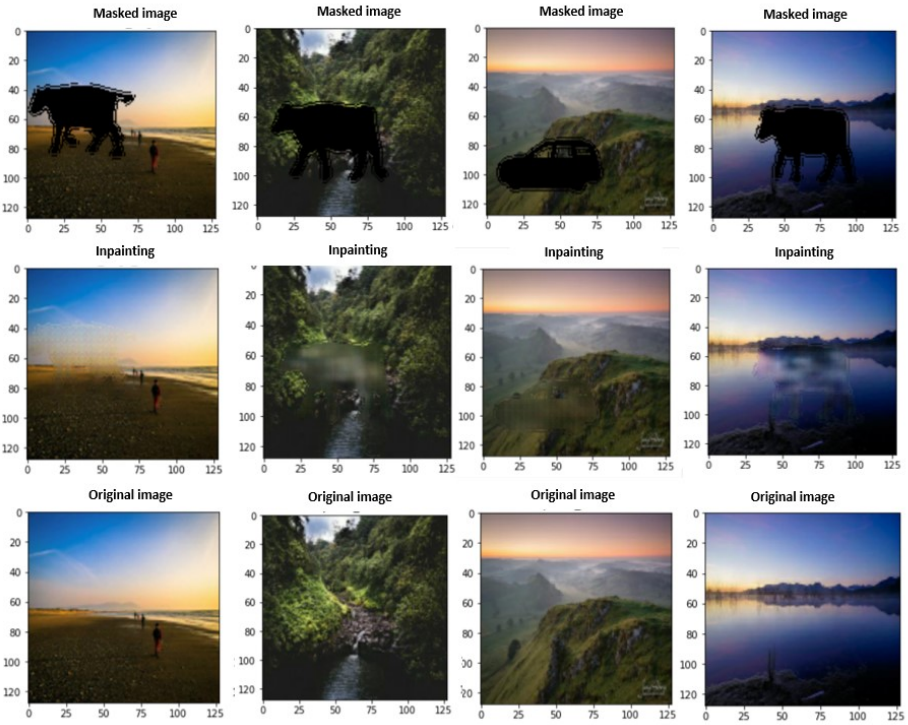


Figure 17: Visual results of Random region.

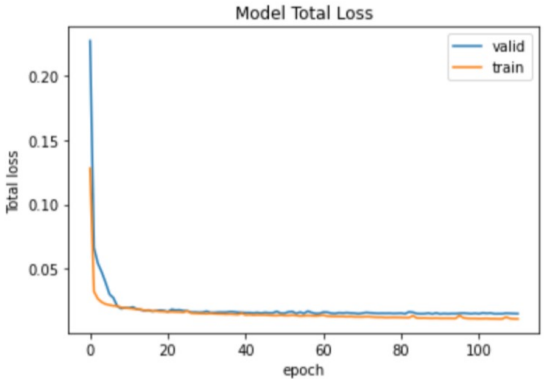


Figure 18: Graphic results of random region.

Center region, Random block, Random region - 'Monet' dataset:

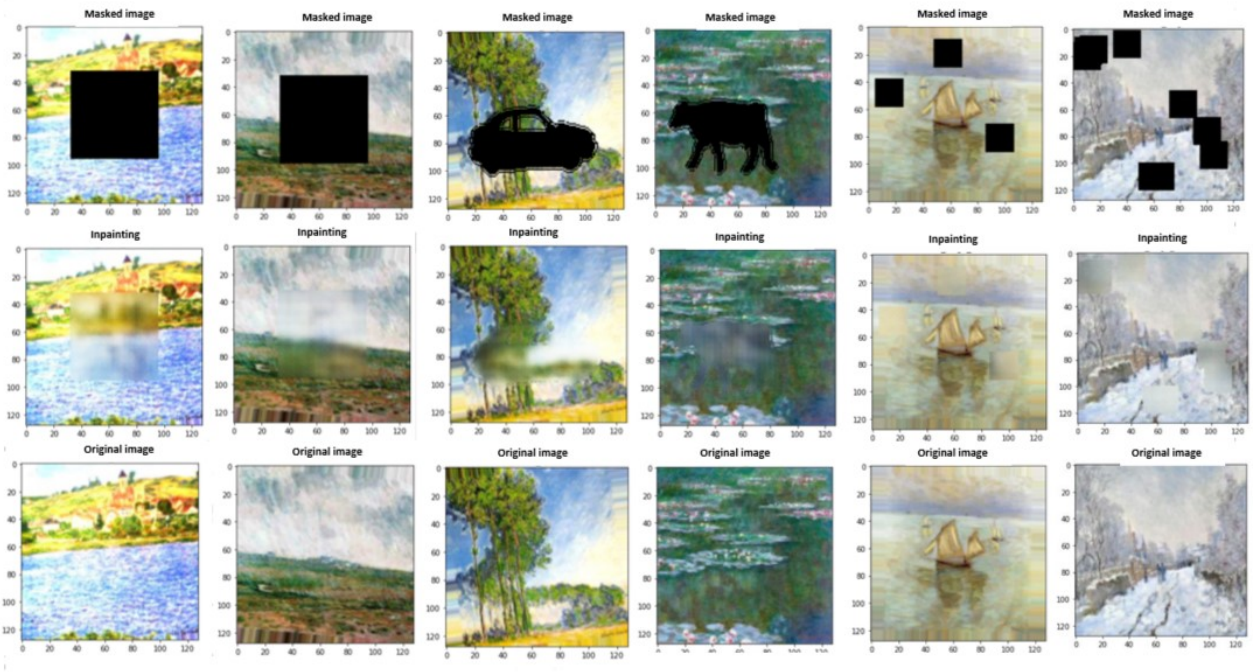


Figure 19: Visual results of 'Monet' dataset.

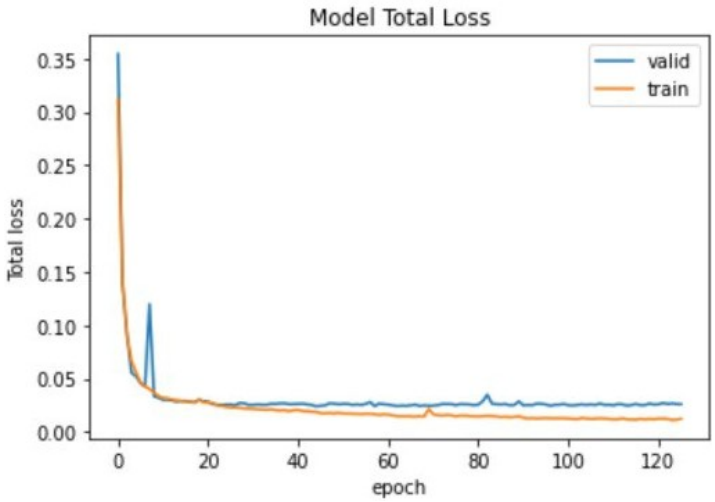


Figure 20: Graphic results of 'Monet' dataset.

3 Discussion

In this project we have tested and explored different aspects of deep learning networks and image inpainting techniques.
We invested a lot of time and effort to understand how our networks architecture should composed of and how we keep the top level architecture presented in the paper. With that said, we wanted a network which has a high learning ability, but also we needed to take under consideration the number of parameters.
Through discussion and research we found many different ways to face the challenges we had, for example for small dataset we thought about 2 ways (augmentation, manipulation 'Photos' dataset to 'Monet' style) to face this challenge. Moreover, we understood the importance of experiments and understating how the network learns. We conducted many trails until we found the best model and best parameters by following the effect created by each change we've preformed. In order to find the best model we had to pay attention to the values of the loss values of the train and validation to the point where learning stops and overfitting begins. The best results we achieved were in the random block, this makes sense due to the fact that it has small regions to learn. To summarize, we wanted to try more architectures, but due to limited resources we were not able to do so.

4 Code

Two options for a trained model: At the link for Google Drive you may find the following models:

- "Separate models": trained model for each mask type in each dataset: photos+center, photos+random, photos+region, monet+center, monet+random, monet+region.
- "Combined models": trained model for each dataset for all mask types: photos+center/random/region, monet+center/random/region.

Pay attention please:

- Trained model for each mask type in each dataset (first option) has proven to have better results, please take it under consideration while running.
- We've trained our model with masked image, while the missing parts are in black. So, in the test given we have adjusted the images for that reason.

Running instructions: You may find detailed running instructions in the Google Colab notebook.

Datasets: At the link for Google Drive you may find the following datasets:

- For 'Photos' dataset: train_photos, valid_photos - photos for training and validation.
- For 'Monet' dataset: train_preview , valid_preview - photos for training and validation.

Masks:

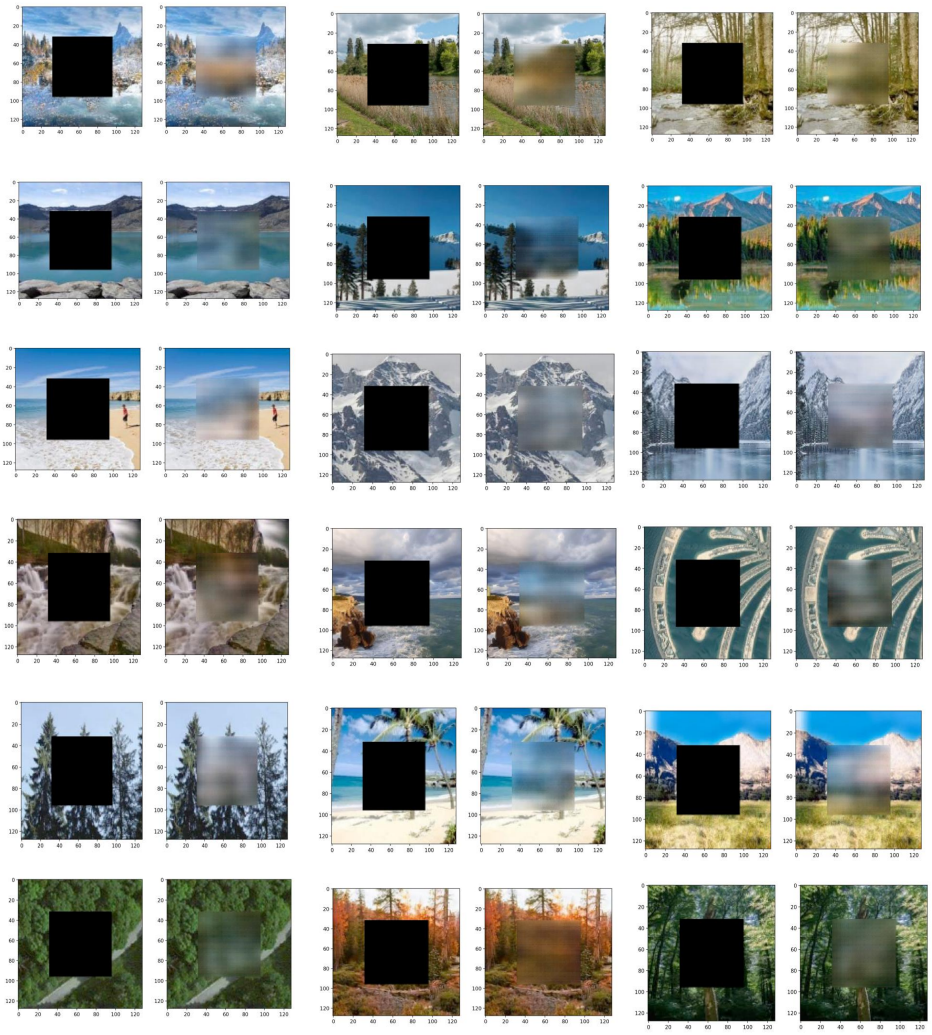
- For center block and random blocks we generated the masks in our code.
- For random region mask - you may find the masks in our Google Drive folder: Datasets/arbi.

Links:

- Google Drive link: https://drive.google.com/drive/folders/1BOenkvTtV_AvLC7xSY-6Ggra7RMIr6CH
- Google colab link: https://colab.research.google.com/drive/16dKK276PeEEx1KEl6x3S52WkkFA8NJSu?usp=sharing

5 Test

'Photos' - center block

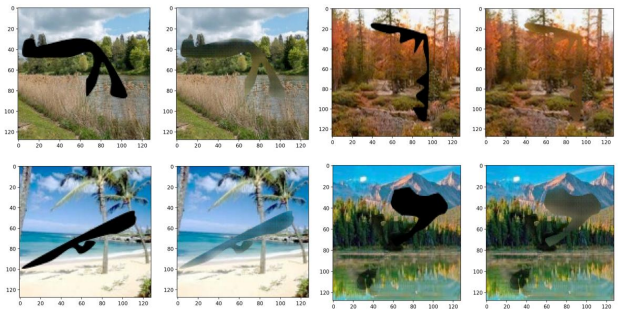


6 Test

'Photos' - random block

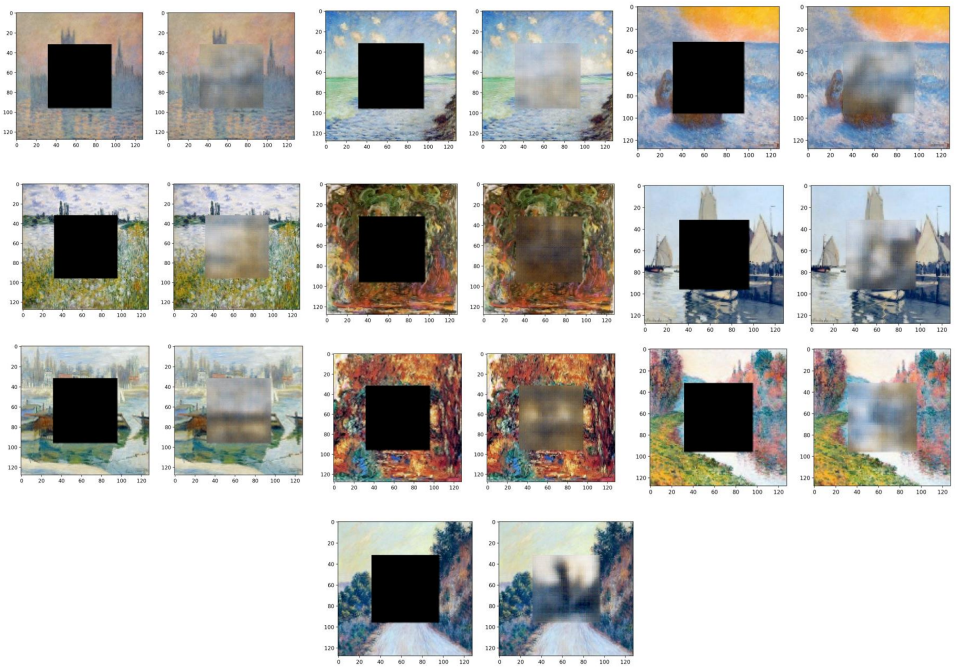
7 Test

'Photos' - random region



8 Test

'Monet' - center block



9 Test

'Monet' - random block