Deep Learning - Final Course Project

Roy Mechany (ID. 316383504),Gavriel Sorek (ID. 318525185)

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

1 Introduction

Image inpainting is the task of reconstructing missing regions in the image. our main goal in this task is to generate an image with the missing regions so that it will not be possible to discern from human sight that this is a processed image. In our final project in the course Deep Learning we will try to use all of our skills that we learned in this course and combine them into a project that will inpaint the missing region in images and involved many techniques that we learned.

Image inpainting is a classic problem in computer vision , this section has a lot of uses in our world such as remove an object from the picture, image completion and more.

The task of image inpainting has significant progress in the last years due to the evolution of the neural and deep learning.

In our project we will handle 3 major cases of image inpainting,
1. Center block
2. Arbitrary region
3. Random blocks

The wholes will be presented in masks, here is an example of masks of each type:

We will represent our high level architecture, the challenges and the difficulties in the development to reach our model.

1.1 Related Works

Image inpainting is a common and classic problem in computer vision, there is a lot of works and articles about this issue:

1. Guilin Liu, Fitsum A. Reda, Kevin J. Shih, Ting-Chun Wang, Andrew Tao, Bryan Catanzaro - Image Inpainting for Irregular Holes Using Partial Convolutions

2. Jiahui Yu, Zhe Lin, Jimei Yang, Xiaohui Shen, Xin Lu, Thomas S. Huang - Generative Image Inpainting with Contextual Attention

3. AYUSH THAKUR, SAYAK PAUL-Introduction to image inpainting with deep learning

2 Solution

2.1 General approach

In our project, we used the PyTorch environment.
PyTorch is an open source machine learning framework based on the Torch library, used for applications such as computer vision and natural language processing, This environment developed by Facebook AI Research lab.

Our solution is based on the article "Context Encoders: Feature Learning by Inpainting" by Deepak Pathak ,Philipp Krahenbuhl ,Jeff Donahue, Trevor Darrell. Alexei A. Efros, from University of California, Berkeley
https://arxiv.org/pdf/1604.07379.pdf

We will describe and represent our progress in implementing the problem-solving model.

Along the way we have defined 3 main models, each of which comes to improve and address a different problem in our solution.

1. The first model goal is to handle and inpaint only in cases of task of inpatient center block.
2. The second model goal is to handle tasks of inpaint multi random blocks and arbitrary region in the photo.
3. The third model is essentially similar to the second model but we implement new technique of transfer learning, and we used the encoder as the Alex-Net feature extraction. Each of the models has 2 main parts, the primary model and discriminator model.

The main consideration of configure,train and used the second model (instead of the first model) was that the first model is using specific mask size and not compatible to use multi size on center blocks, we wants to create and use more generic model that will handle in many cases of mask, the second model (will more details in the next section) is more generic and get mask

and picture.

On the second model we got satisfactory results for the inpainting task with different masks,but we thought we can get better results and faster training cycle time if we will switch our encoder implementation and will use the leaned features extraction from the AlexNet net. Also as we will present in the next paragraph some adjustment we did for this net.
We will describe briefly the architecture of the first model that was not included in the final solution:

First model high level architecture:
Encoder architecture :

Conv Layer	**Input channel**	**Output channel**	**Kernel size**	**Padding**	**Stride**
Encoder layer 1	3 (RGB)	64	(11X11)	0	(4X4)
Batch normalisation 64 , Activation function:ReLU					
Encoder layer 2	64	128	(5X5)	2	(1X1)
Batch normalisation 128, Activation function: ReLU Max pool (kernel size = 3 , stride = 3)					
Encoder layer 3	128	256	(3X3)	1	(1X1)
Batch normalisation 256, Activation function: ReLU Max pool (kernel size = 3 , stride = 2)					
Encoder layer 4	256	512	(3X3)	1	(1X1)
Max pool (kernel size = 3 , stride = 2), Flatten # out					

Decoder architecture:

Conv Layer	**Operation**	**Input channel**	**Output channel**	**Kernel size**	**Stride**
Decoder layer 1	conv transpose	512	256	(3X3)	(2X2)
Batch normalisation 256, Activation function: ReLU					
Decoder layer 2	conv transpose	256	128	(4X4)	(1X1)
Batch normalisation 128, Activation function: ReLU					
Decoder layer 3	conv transpose	128	64	(5X5)	(2X2)
Batch normalisation 64, Activation function: ReLU					
Decoder layer 4	conv transpose	64	32	(4X4)	(2X2)
Batch normalisation 32, Activation function: ReLU					
Decoder layer 5	conv transpose	32	3 (RGB)	(1X1)	(1X1)
Activation function: ReLU, Activation function: P-ReLU, Batch normalisation 8192, Batch normalisation 8196					

Discriminator model:

Conv Layer	Input channel	Output channel	Kernel size	Padding	Stride
Dis layer 1	3 (RGB)	64	(3X3)	1	(1X1)
Batch normalisation 64 , Activation function:ReLU Max pool (kernel size = 3 , stride = 3)					
Dis layer 2	64	128	(3X3)	1	(1X1)
Batch normalisation 128, Activation function: ReLU Max pool (kernel size = 2 , stride = 2)					
Dis layer 3	128	256	(3X3)	1	(1X1)
Batch normalisation 256, Activation function: ReLU Max pool (kernel size = 2 , stride = 2)					
Dis layer 4	256	512	(3X3)	1	(1X1)
Batch normalisation 512, Activation function: ReLU					
Flatten					
Flatten -normalization 8196					
FC layer 1	8192	64			
FC layer 1 - normalization - 64					
FC layer 2	64	1 (real or fake)			
Activation function Sigmoid Activation function ReLU					

2.2 Design

In this section, we will describe and explain our model and our decisions to make better results and predictions.

2.2.1 Parameters and architecture

In this section we will present our architecture of the second model and the architecture of the final model.

The second model architecture:
Encoder architecture :

Conv Layer	Input channel	Output channel	Kernel size	Padding	Stride
Encoder layer 1	3 (RGB)	96	(11X11)	0	(4X4)
Activation function:ReLU, Max pool (kernel size = 3 , stride = 2)					
Encoder layer 2	96	256	(5X5)	2	(1X1)
Activation function: ReLU Max pool (kernel size = 3 , stride = 2)					
Encoder layer 3	256	384	(3X3)	1	(1X1)
Activation function: ReLU					
Encoder layer 4	384	384	(3X3)	1	(1X1)
Activation function: ReLU					
Encoder layer 5	384	384	(3X3)	1	(1X1)
Activation function: ReLU Max pool (kernel size = 3 , stride = 2), Flatten channel-wise # out					

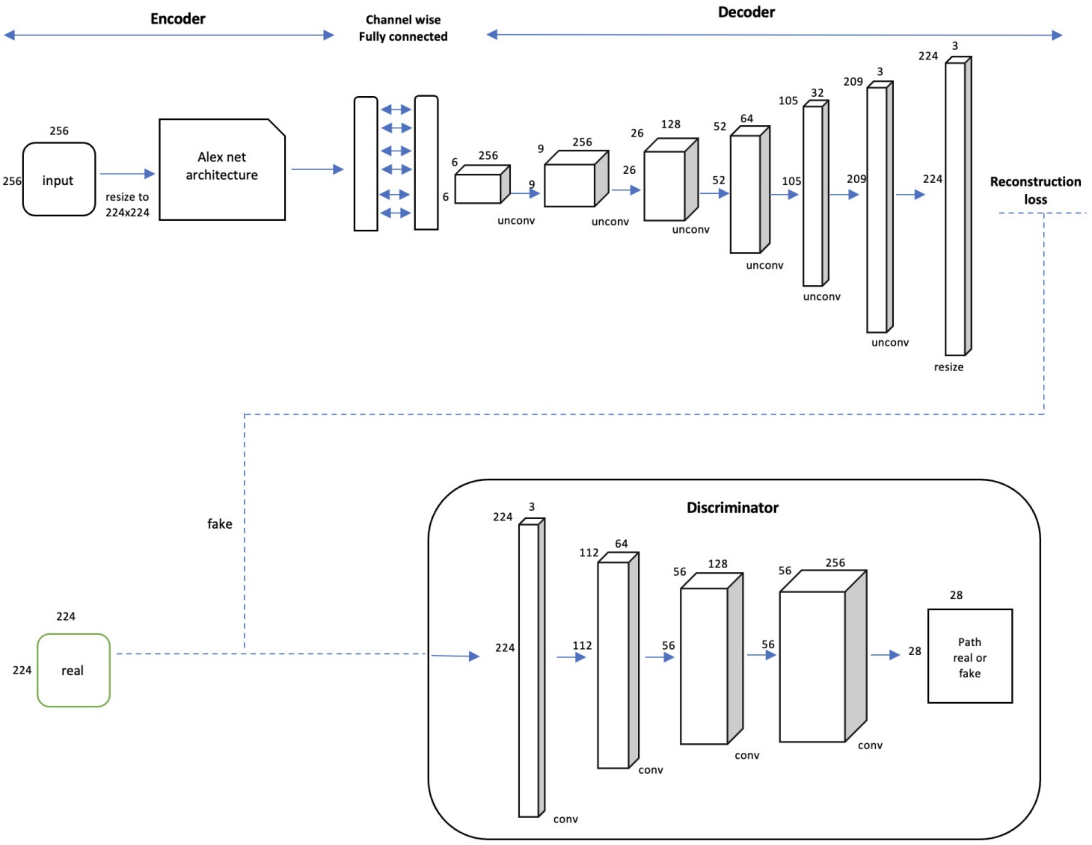
Decoder architecture :

Conv Layer	Operation	Input channel	Output channel	Kernel size	Stride
Decoder layer 1	conv transpose	512	256	(4X4)	(2X2)
Batch normalisation 256, Activation function: ReLU					
Decoder layer 2	conv transpose	256	128	(4X4)	(2X2)
Batch normalisation 128, Activation function: ReLU					
Decoder layer 3	conv transpose	128	64	(4X4)	(2X2)
Batch normalisation 64, Activation function: ReLU					
Decoder layer 4	conv transpose	64	32	(5X5)	(2X2)
Batch normalisation 32, Activation function: ReLU					
Decoder layer 5	conv transpose	32	3 (RGB)	(4X4)	(2X2)

Discriminator architecture:

Conv Layer	**Input channel**	**Output channel**	**Kernel size**	**Padding**	**Stride**
Dis layer 1	3 (RGB)	64	(3X3)	1	(2X2)
Batch normalisation 64 , Activation function:ReLU					
Dis layer 2	64	128	(3X3)	1	(2X2)
Batch normalisation 128, Activation function: ReLU					
Dis layer 3	128	256	(3X3)	1	(1X1)
Batch normalisation 256, Activation function: ReLU					
Dis layer 4	256	1 (real or fake)	(3X3)	1	(2X2)

The final model architecture:



Encoder architecture:
In our implementation we configured the encoder to use the AlexNet architecture.
Because we selected this net we needed to normalized out input photos to the alexNet **format, we resized our input size to 224x224 and normalized the input with the parameters: (mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225].

After this part we configured the channel-wise fully-connected layer (as suggested in the article) which allows each unit in the decoder to reason about the entire image content.
This layer is essentially a fully-connected layer with groups, intended to propagate information within activation's of each feature map. If the input layer has m feature maps of size n $\times$ n, this layer will output m feature maps of dimension n $\times$ n. However, unlike a fully-connected layer, it has no parameters connecting different feature maps and only propagates information within feature maps. The number of parameters in this channel-wise fully-connected layer is mn^4.

Decoder architecture:

Conv Layer	Operation	Input channel	Output channel	Kernel size	Stride	Padding
Decoder layer 1	conv transpose	256	128	(4X4)	(1X1)	
Batch normalisation 128, Activation function: ReLU						
Decoder layer 2	conv transpose	128	64	(4X4)	(3X3)	(1X1)
Batch normalisation 64, Activation function: ReLU						
Decoder layer 3	conv transpose	64	64	(4X4)	(2X2)	(1X1)
Batch normalisation 64, Activation function: ReLU						
Decoder layer 4	conv transpose	64	32	(5X5)	(2X2)	(1X1)
Batch normalisation 32, Activation function: ReLU						
Decoder layer 5	conv transpose	32	3 (RGB)	(3X3)	(2X2)	(1X1)

Discriminator architecture:
The discriminator architecture is the same architecture of the second model that we present in this section.

2.2.2 Loss functions

In this section we will provide information about the evaluation of the loss function in our model.

The first loss that we calculated is the reconstruction loss, this loss calculate the mean squared error (MSE) between the original image to the generated image.
The second loss is the discriminator loss, which learns to identify features that are specific only to generated patches. For this loss we used "patch loss" , we calculate for each patch if this patch is real or fake, the patch size is defined following to the receptive filed. the calculation of the receptive field is describe in the next section). The calculation for this loss is based on binary cross entropy (BSE) loss between the images.

We combined these two losses to "context encoder loss" for our model, as suggested at the article we used the formula: (0.999 * reconstruction loss + 0.001 * discriminator loss).

For the "monet" data-set we added additionally loss, the style loss. This loss purpose is to ensure and validate that the original style picture and the generated image have as equal as possible correlations across all layers, this mean that we want to keep the style and the texture of the photos.
for this loss we used the gram matrix, we calculate the style loss for each layer of the encoder with different weights,
For the photos for the "monet" data-set we used this formula to calculate the combined loss:

(0.999 * reconstruction loss + 0.001 * discriminator loss + 0.01 * style loss)
We tried to change the style loss wight with different values but we found that 0.01 gave us the best result on the data-set.

2.2.3 Normalization

For our data we use the Alex-Net normalization weights as described in the design section.
This normalization works better on our model than the min-max or others classics normalization functions.

2.2.4 Receptive field

the Receptive Field is defined as the size of the region in the input that produces the feature.
Basically, it is a measure of association of an output feature (of any layer) to the input region (patch).
The formula to calculate the receptive field is : $\mathrm{K}_{L-1} = K + S(K_L - 1)$

$K = kernelsize, S = stride.$

$Output = 28x28x1$
$k4 = 3 + 2(1-1) = 3x3$
$k3 = 3 + 1(3-1) = 5x5$
$k2 = 3 + 2(5-1) = 11x11$
$k1 = 3 + 2(11-1) = 23x23$
$image = 3 + 2(23-1) = 47x47$

As we calculate below, we saw that every pixel in the discriminator sees 47x47 in the original picture.
Clearly, we can see that there is intersections between the receptive field of every neuron in the result.
The loss value is influenced by every patch and it's helps our model to change the wights accordingly to the problematic region.

2.2.5 Augmentation

In our project we handled 2 data-sets , the first one is a large one and has about 7000 photos , and the second data-set , "monet" is smaller and have only 300 photos. our main challenge in this scope regard's the second data-set is that we don't have a lot of photos to train and "learn" from them to get and inpaint new photos.

To improve our results and expand out data-set we used augmentation to generate and do some manipulations on our photos, we used the transform package from pythroch and used the AutoAugment function. this method generates photos with randomize algorithm such as : RandomRotation, GaussianBlur,RandomHorizontalFlip and more.

We generated many photos from each image in the train data-set and total we amplified out train data to 3576 photos.
full details about the available transforms can be found at the following link
https://pytorch.org/vision/stable/transforms.html

2.2.6 Optimizers

In our project we used 2 different optimizer , Adam + SGD , each of them was adjusted according to the purpose of that logical part.

For our main model we used the Adam optimizer to get fastest and reliable results depending on the general direction.
Parameters for Adam: lr=0.0006, betas=(0.5, 0.999)

The second optimizer we uses is SGD , we used this optimizer for the discriminator with learning rate 0.001. we decide to configure this optimizer to get accurate results as possible although this optimizer is not the efficient one, the purpose is to go ahead in the right direction.

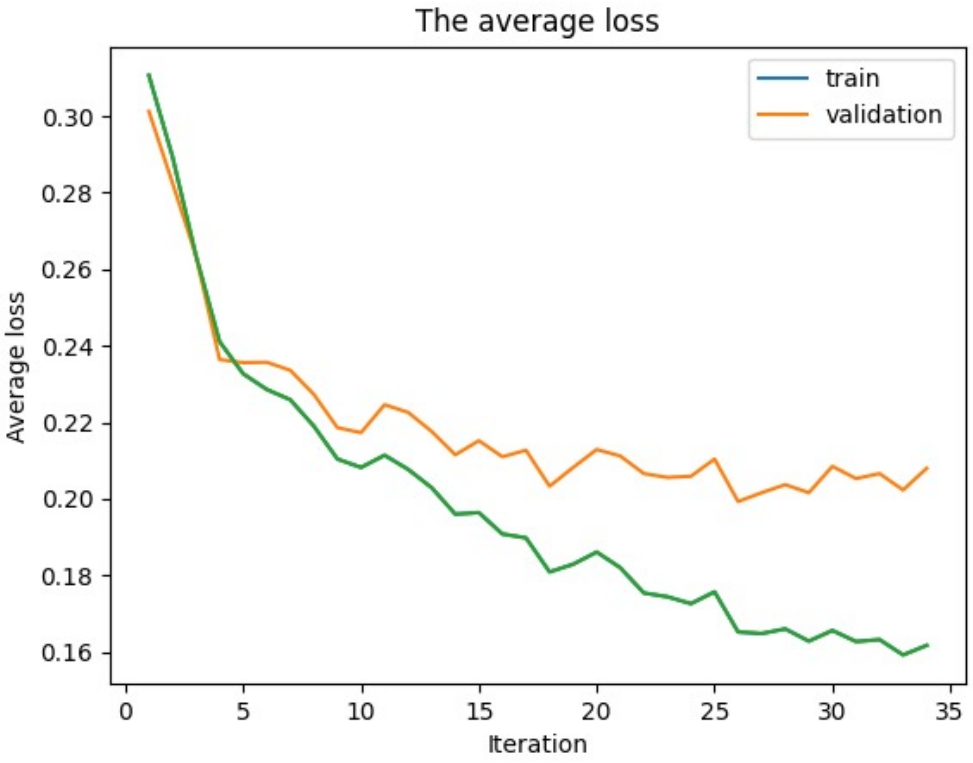
2.2.7 Train and validation

We divided our data into 2 groups, train and validation photos in ratio on 80 percentage train photos and 20 percentage for the validation.

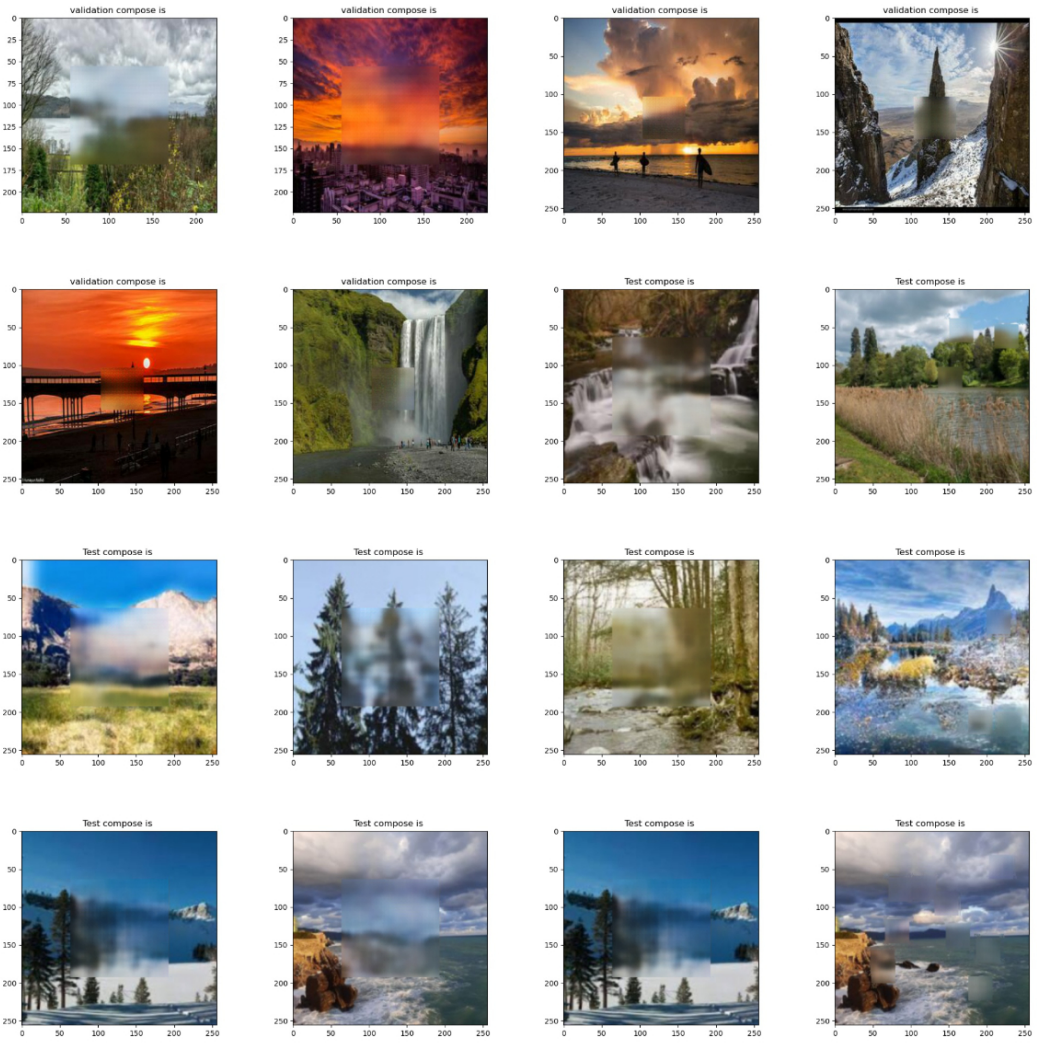
We also calculated the loss on the validation set and saved the best model with the lowest loss value.
Finally, we decided to configure the number of epochs for train to 50.

2.3 Experimental results

This graph is present the average loss value for the training and the validation data for the photos dataset after each iteration, as you can see in the graph, the loss value is in a declining trend.
After 34 epochs we stooped to training the model because we saw the the loss of the validation change the declining trend and start to raise up.



Also, we will provide some results from our model. The results we will present is from our validation set + test set after we finish to train our train Photos.



3 Discussion

In this project we tried to implement and use many techniques that we learned in this course, such as :

* data augmentation
* transfer learning
* generator discriminator architecture
* patch loss
* CNN (convectional neural network)
* Activation functions and optimizers

Also as part of this project we needed to search and read academic papers to learn ,investigate and enrich our knowledge in purpose to improve solution to the relate problem.
We learned a lot about machine learning and deep learning techniques and technology's that will help us in our careers.

4 Code

You can see the full implementation in the GitHub project (with full instructions to run the project). and in the google colab notebook.

`https://github.com/gavrielSorek/deep_learning_inpainting`

`https://colab.research.google.com/drive/1UBU2IAjb1yj_JWgHZ3mYbLEqw0MG-y0W?usp=sharing`