CSE-578 Computer Vision

Project Report
Context Encoders: Feature Learning by Inpainting

April 22 2020

1 Abstract

In this paper,the authors propose a novel unsupervised feature learning method using inpainting. Inpainting is the task of filling a potentially large region of an image (one fourth for example). The method is Context Encoder, a convolutional neural network trained for inpainting. Trained to create plausible content for the missing part of an image based on the surroundings, the network learns a representation of the input that captures the appearances and semantics of visual structures. The features learned by the network are shown to be effective for other tasks than inpainting that usually requires supervised learning for example segmentation, detection and classification.

1.1 Introduction

Convolutional Neural Networks (CNN) were originally found to be effective for the classification task under a supervised setting. A tremendous amount of later works further proved that they also excel for other computer vision tasks. In the present work the authors explore CNN trained in a unsupervised manner. The network is asked to completely restore a large part of the input image that is missing (for example a $64 * 64$ square in the center of an $128 * 128$ image). This enforces the network to holistically understand the semantics of the scene, learns high-level features over the whole spatial extent of the image in order to provide content for the missing part that semantically complies with the context of the scene. This is achieved thanks to an Auto-Encoder-like architecture. The network is composed of an **Encoder** that transforms the input image into a fixed small number of features followed by a **Decoder** that creates out of these features the content of the missing part. The parameters are learned with a reconstruction loss as well as an adversarial loss.

1.2 Context Encoder-Decoder:Network Architecture

The original architecture of Context encoders proposed by the authors was to use an autoencoder-like structure to code the context of an input image as a $6 * 6 * 256$ feature map (9126 hidden features). The encoder was to be derived from the **AlexNet** architecture and composed of five convolutional layers. The output of the encoder is connected to the input of the decoder by a channel-wise fully-connected layer (($6 * 6 * 256$=9126) activations layers), in order to propagate the information within each feature map. The decoder follows with five up-convolutional layers that augment the dimensions of the latent representation in-between the encoder and the decoder (at the interface of the encoder and the decoder).

1.2.1 Loss Function

The choice of the loss function is strongly related with the task at hand namely inpainting or put it differently filling a hole missing in the input image. The paper initially uses a regular $L2$ norm based loss function , but the resulting patches were blurry and were not sharp. To state in detail , we have x as our input image and $\hat{M}$ the binary mask , such that $\hat{M} * x$ (Hadamard product) selects the missing region.The formula is as follows :

$$L_{recx} = || \ \hat{M} * (x - F(1- \hat{\mathrm{M}}) *x \ ||^2$$

Note that the mask $\hat{M}$ can be of any size and shape, covering up to 1/4 of the image. This pixel-wise distance does not encourage the network to produce details with high precision but rather create content that overall capture the structure of the original region . To overcome the blurry result favored by the reconstruction loss the authors propose to incorporate an adversarial loss to the total loss.

1.3 GAN Framework : Adversarial Loss

The adversarial loss is based on Generative Adversarial Networks. GANs were introduced by [6] as deep generative models able to learn to represent an estimate, either explicitly or implicitly, of some distribution $pdata(X)$ on some space X, given a training set composed of samples x drawn from this distribution. As explained in [5], this is done through a two player game between two parameterized and differentiable (with regards to their inputs and their parameters) functions: a generator G and a discriminator D.

- The **Generator G : Z** $>$ **X**: takes as input some z drawn from a prior distribution $p(z)$ on a space Z and tries to produce an output $G(z)X$ drawn from a probability distribution p_{model} such that that p_{model} is close from p_{data}, that is to say such that G_z seems to have been drawn from the distribution $p_{data(X)}$.
- The **Discriminator D:Z** $>$ **X** : x=$[0, 1]$ on the other hand takes as input an element x $0X$ and outputs the probability D(x) that x comes from the training set, that is to say, has been drawn from the true distribution $p_{data}(X)$, rather than having been generated by G, and therefore drawn from the distribution $p_model(X)$. This translate by the maximization of the loss function L_{adv} given by the following log likelihood (or the minimization of the corresponding negative log likelihood):

$$\begin{aligned}\mathcal{L}_{adv}(G, D, Z, X) =& E_{x\sim p_{data(X)}}(logD(x)) \\ &+ E_{z\sim p_Z}(log(1-D(G(z))))\end{aligned}$$

The generator tries to fool the discriminator, and therefore tries to minimize this very value $L_{adv(}G, D, Z, X)$. The solution of this mini- max game (or zero-sum game, as the generator and the discriminator try to minimize two opposite cost functions) is given by $min_G\ max_D\ L_{adv(}G, D, Z, X)$.

1.4 Adversarial Loss

In the case of our inpainting task, the target domain X is the space of complete(d) images, with p_{data} represented by the set of training images. The source domain Z is the space of context images (image with a missing masked region). The encoder-decoder pipeline F aims at generating a complete image out of an incomplete one and therefore can be seen as a generator. The authors hence introduce a corresponding discriminator network D that aims at predicting whether the input fed is a real sample or is a sample coming from the distribution of the content generated by the encoder-decoder pipeline. The discriminator is a five layer convolutional network that takes as input the context either completed with the ouptut of the encoder-decoder pipeline (which is the generated missing area) or the ground truth missing area. The adversarial loss is then :

$$\begin{aligned}\mathcal{L}_{adv}(F, D) =& \mathbb{E}_{X\in\mathcal{X}}[\log D(x)] \\ &+ \log(1-D(F((1-\hat{M})*x)))\end{aligned}$$

1.4.1 Joint Loss

The joint loss that the encoder-decoder generator tries to minimize is a weighted combination of the reconstruction loss and the adversarial loss .

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

Here, the $\lambda 1$ value is set to 0.1 and $\lambda 2$ value is set to 0.9. Note that unlike the base GAN framework, the loss that the generator of the context encoder tries to minimize is not strictly equal to the opposite.

2 Model Diagram : Fixed Size Patch

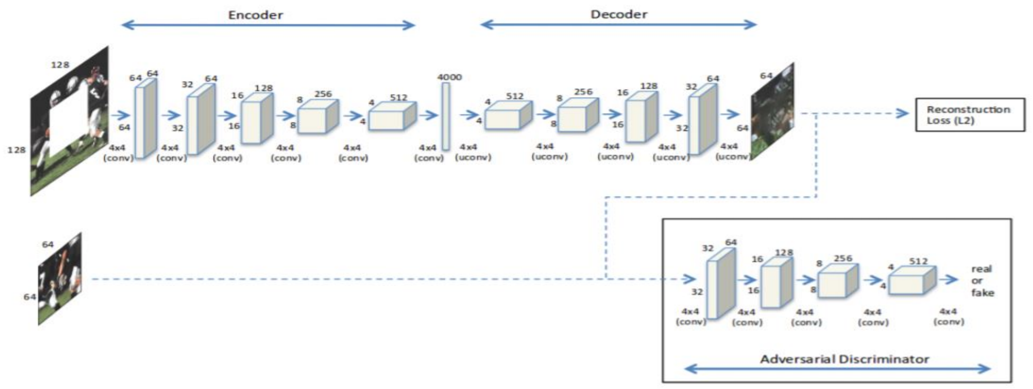

Figure 1. Context Encoder trained with joint reconstruction and adversarial loss for **semantic inpainting**. This illustration is shown for center region dropout. Similar architecture holds for arbitrary region dropout as well

Figure 1: Model Architecture

3 Our Code Results

For producing the following results , we have taken up Paris Building dataset provided by the authors which consists of 14000 of variations. The network has undergone 3000 epochs while training.

3.1 Generator Module

```
from torch import nn

class generator(nn.Module):
    '''
    Generator
    encoder-decoder setup
    '''
    def __init__(self, img_x=128, img_y=128, channels=3):
        super().__init__() #for creating a instance of nn.module for inheritence
        self.img_x, self.img_y = img_x, img_y
        self.channels = channels

        def down_sampling(input_channels=None, output_channels=None, normalize=True):
            layers = list()

            #Conv2d Inputs
            args = {
                        'in_channels':input_channels,
                        'out_channels':output_channels,
                        'stride':2,
                        'padding':1,
                        'kernel_size':(4 ,4)
                   }

            layers.append(nn.Conv2d(**args))

            #normalize flag
            if normalize:
                layers.append(nn.BatchNorm2d(output_channels))

            #activation
            layers.append(nn.LeakyReLU(.2))
            return layers

        def up_sampling(input_channels=None, output_channels=None, normalize=True):
            layers = list()

            #Conv2dTranspose Inputs
            args = {
                        'in_channels':input_channels,
                        'out_channels': output_channels,
                        'kernel_size':(4, 4),
                        'stride':2,
                        'padding':1
                   }

            layers.append(nn.ConvTranspose2d(**args))

            #normalize flag
            if normalize:
                layers.append(nn.BatchNorm2d(output_channels))

            #activation
            layers.append(nn.ReLU())
            return layers
        def get_layers(layer_config_list, transpose=False):
            layers = list()

            for layer_config in layer_config_list:
                if not transpose:
                    layers += down_sampling(**layer_config)
                else:
                    layers += up_sampling(**layer_config)

            return layers

        def config_layer(input_channels=3, output_channels=3, normalize=True):
            return locals()

        self.encoder_layers = list()
        #encoder_layers
        self.encoder_layers.append(config_layer(self.channels, 64, False)) #-->layer 1
        self.encoder_layers.append(config_layer(64, 64, True))              #-->layer 2
        self.encoder_layers.append(config_layer(64, 128, True))             #-->layer 3
        self.encoder_layers.append(config_layer(128, 256, True))            #-->layer 4
        self.encoder_layers.append(config_layer(256, 512, True))            #-->layer 5
        self.decoder_layers = list()
        #decoder_layers
        self.decoder_layers.append(config_layer(4000, 512, True))           #-->layer 1
        self.decoder_layers.append(config_layer(512, 256, True))            #-->layer 2
        self.decoder_layers.append(config_layer(256, 128, True))            #-->layer 3
        self.decoder_layers.append(config_layer(128, 64, True))             #-->layer 4
        #model
        self.model = nn.Sequential(
                                        #encode Layers
                                        *get_layers(self.encoder_layers, transpose=False),

                                        #bootleneck
                                        nn.Conv2d(in_channels=512, out_channels=4000, kernel_size
    =(1,1)),

                                        #decode Layers
                                        *get_layers(self.decoder_layers, transpose=True),

                                        nn.Conv2d(in_channels=64, out_channels=self.channels,
    kernel_size=(3, 3), stride=1, padding=1),nn.Tanh)
    def forward(self, x):
        return self.model(x)
```

3.2 Discriminator Module

```python
from torch import nn
class discriminator(nn.Module):
    '''
    Discriminator setup
    '''
    def __init__(self, channels=3):
        super().__init__()
        self.channels = channels

        def down_sampling(input_channels=None, output_channels=None, stride=None, kernel_size=None, normalize=None, activation=True):
            layers = list()

            #Conv2d inputs
            args = {
                        'in_channels':input_channels,
                        'out_channels':output_channels,
                        'stride':stride,
                        'kernel_size':kernel_size,
                        'padding':1
                   }

            layers.append(nn.Conv2d(**args))


            #normalize flag

            if normalize:
                layers.append(nn.InstanceNorm2d(output_channels))
            if activation:
                layers.append(nn.LeakyReLU(.2, inplace=True))

            return layers

        def get_layers(layer_config_list):
            layers = list()

            for layer_config in layer_config_list:
                layers += down_sampling(**layer_config)

            return layers

        def config_layer(input_channels=3, output_channels=3, stride=2, normalize=True, kernel_size=(4, 4), activation=True):
            return locals()

        self.discriminator_layers = list()
        #discriminator layers

        self.discriminator_layers.append(config_layer(self.channels, 64, 2, False))     #layer-->1

        self.discriminator_layers.append(config_layer(64, 128, 2, True))                #layer-->2

        self.discriminator_layers.append(config_layer(128, 256, 2, True))               #layer-->3

        self.discriminator_layers.append(config_layer(256, 512, 2, True))               #layer-->4

        self.discriminator_layers.append(config_layer(512, 1, 1, True))                #layer-->5

        #model
        self.model = nn.Sequential(*get_layers(self.discriminator_layers),nn.Sigmoid())


    def forward(self, x):

        return self.model(x)
```

4 Qualitative Results

The following are few results of the test images , generated by our model. The first row consists of the patch in the cropped region , the second row is our model's result and last row is the corresponding ground truth image.

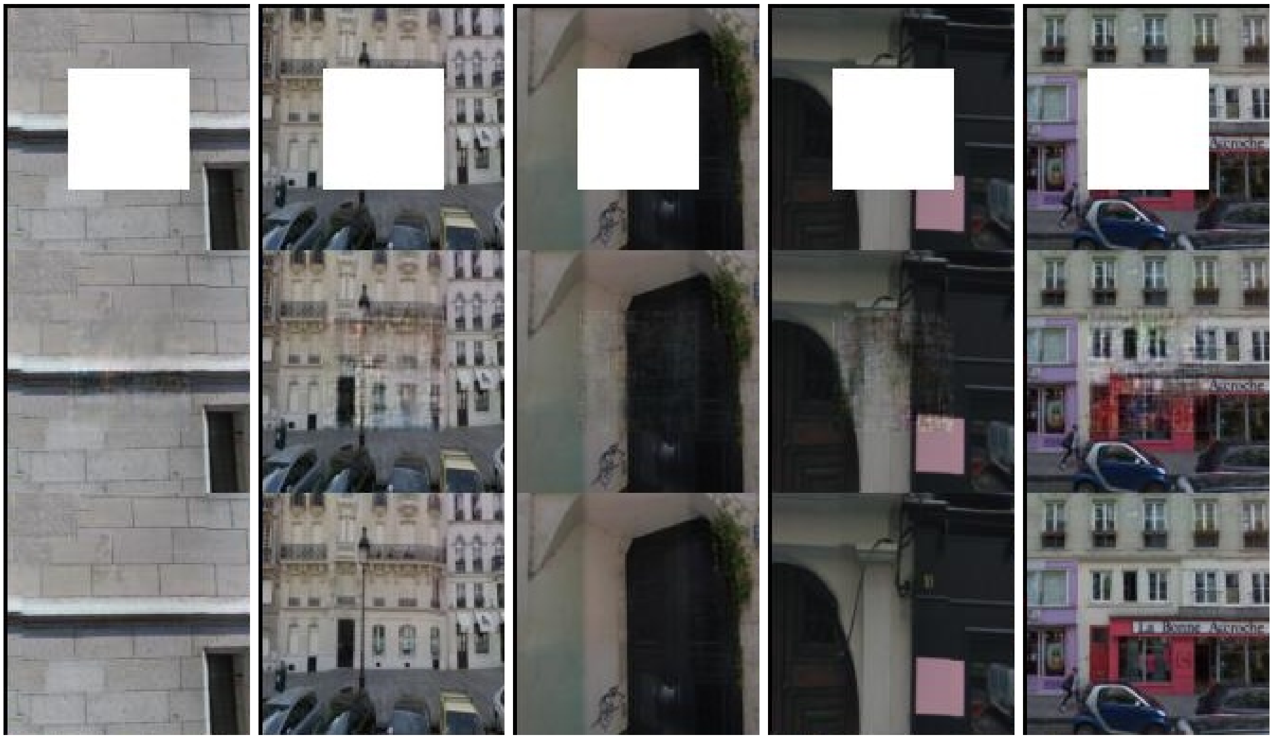

Figure 2: Test Instance -1

5 Model Settings

We have tweaked certain parameters of the model and its training procedure to match the results according to our available resources . The following are parameters that we have used :

- Learning Rate : 0.0002
- Optimiser : Adam (Discriminator and Generator)
- Batch Size : 256
- Training Size Set : 14000
- Training Epochs for Random Crop : 3000
- Training Epochs for Center Crop : 200
- Total Trainable Parameters (Generator) : 71124544
- Total Trainable Parameters (Discriminator) :2675568

6 Plots : Training Loss

In the following graph , we can notice the Generator and Discriminator's losses decreasing across the epochs . The red line signifies the model's training error .

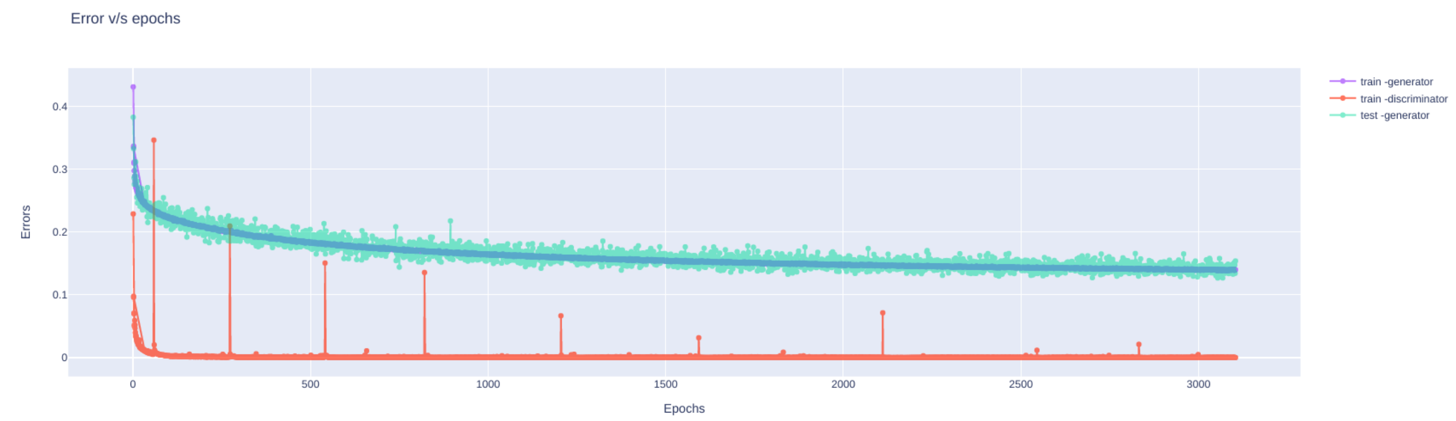

Figure 3: Training Loss for Discriminator and Generator

7 Relevant Links

To find the entire model code and other detailed results, you can visit our GitHub repository .
Link: `https://github.com/thesigmaguy/context_encoder`

8 References

- `https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf`
- `https://people.eecs.berkeley.edu/~pathak/papers/cvpr16.pdf`
- `http://ufldl.stanford.edu/tutorial/unsupervised/Autoencoders/`

9 Team

- S.Dhawal - 2019201089
- Sravya.S - 2019702008
- Niharika.V - 2018122008