ML4B

Inpainting Marine Images (Prisilla & Frederick)

> It is highly recommended to read the fundations on the page "Image inpainting using Wasserstein GAN" to gain a basic understanding and insight into gan

---

Abstract

In this work, we present a high-resolution generative adversarial network for object removal with free-form mask and guidance, and a generative multicolumn network to recover the corrupted images using a dataset "deepfish", which is a collection of different images with and without objects. The goal was to answer the question, "Which is the most effective and efficient method to achieve sufficient results in two different application domains and what are the challenges?".

Introduction

"No system is secure" this phrase (subtitle to the movie "Who am I") is fundamental to the discipline of cybersecurity. Even random hardware failures, such as corrupted memory or pixels in images or cameras, can be compensated.The question then is what to do. Either recreate the image, with all the vastness that it entails, for example, on an expedition to the deep sea, or use artificial intelligence to reconstruct the image. One of the widely used techniques is image inpainting. Image inpainting can estimate the suitable pixel informations to repair the defects. It can be applied for image restoration, object removal, image denoising, etc.

In this work, we aim to analyze two different image inpainting applications, namely image restoration and object removal, on our dataset "DeepFish". For this purpose, we use image inpainting methods, namely a generative multi-column convolutional neural network

(GMCNN) for reconstructing lost or degraded parts of marine images (image restoration) and a high-resolution generative adversarial network (GAN) for object removal.

This thesis is divided into the following chapters. First, an overview of related work is given and the related work is compared to the model/project created. This is followed by a brief description of how the project was created. The next part of this thesis is a description of the project itself and how it works. This is followed by the evaluation of the created model, divided into two parts. The first part describes the data set and the second part presents our results.

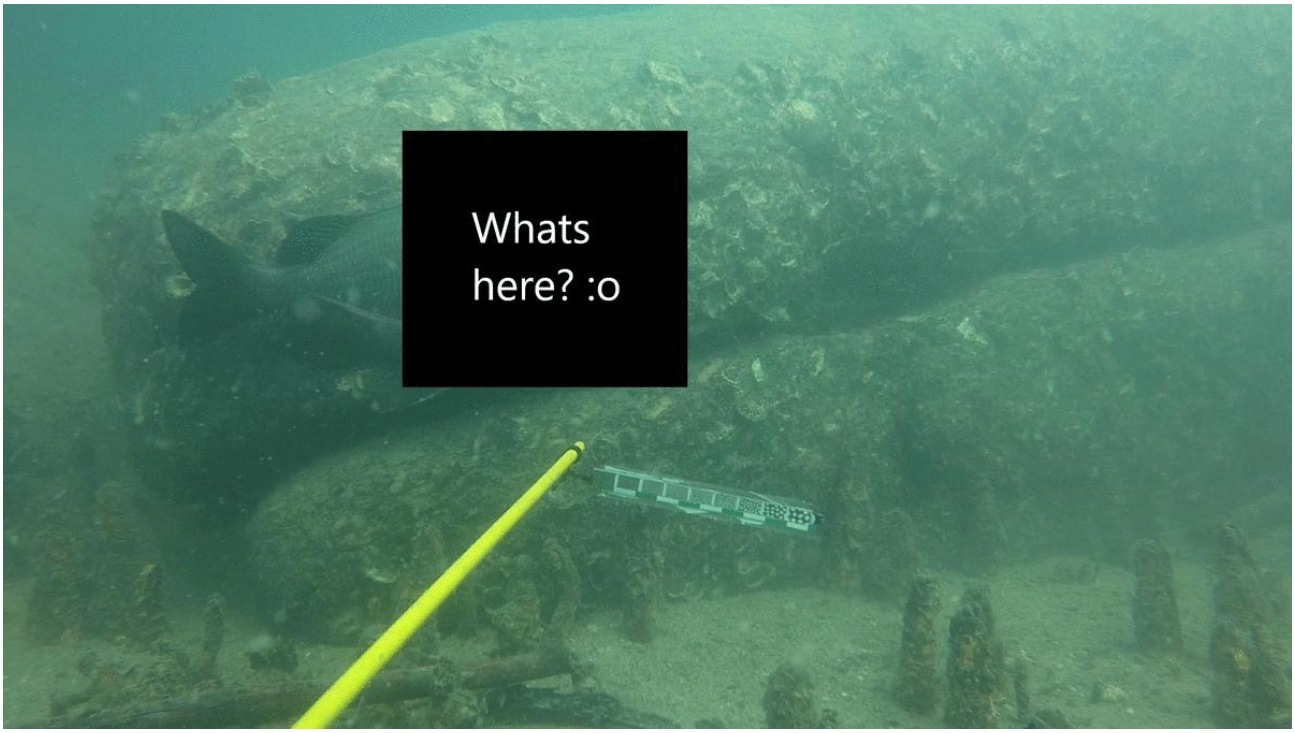


Figure 1: this is how the mask should look like

Related Works

Exemplar-based Inpainting:

This method is basically a traditional method that copies and pastes the pixels in pre-defined order. Using exemplar-based inpainting enable us to achieve speed efficiency and accuracy in the synthesis of texture. At the same time, It often produces a low quality semantic images because of its low-level information, high dependence of order by filling

proceeds. It also can't detect big ambiguities or curve structures and textures, such as faces or facades.

Convolutional Neural Network(CNN):

The algorithms uses the neural network with automatically deep learning. Image inpainting with large-scaling training datasets has been used convolutional neural networks (CNNs) or encoder-decoder network based on CNN.

Generative adversarial network(GAN):

Goodfellow et al. (2014) *Generative Adversarial Networks* retrieved from **https://arxiv.org/abs/1406.2661**

Generative adversarial network (GAN) are a framework which contains two feed-forward networks, a generator G and a discriminator D. This method or known as progressive inpainting, where a pyramid strategy from a low-resolution image to a higher quality images is performed for repairing the image. It's designed to repairing face images. With this method, we can improve performance of image inpainting, but at the same time it needs good performance devices and longer training time.

Dataset Description

The dataset we used is a collection of different fish images consisting of 37322 images of 50 fish classes with pre-extracted feature representations for each image.

The original .zip data contains SummaryOfdatset_190915.csv, the classification, localization, and segmentation folder, which contains subfolders with different images for each folder. The original size of the dataset is 7.06 GB.

In this work we focus only on the segmentation folder. The various data preparation steps are described in the section of each model.

```
DeepFish
├──...
```

```
├──segmentation
│   ├──val.csv
│   ├──train.csv
│   ├──test.csv
│   ├──...
│   ├──images
│   │   ├──9908_Epinephelus_f000180.jpg
│   │   ├──...
├──Localization
├──Classification
```

The image data was collected from public sources, such as Flickr, in 2016. All Images are license free for use.

Image Inpainting using GMCNN

In this chapter we present a method to restore damaged images using a Generative Multi-columend Convolutional Neural Network (GMCNN).

Methodology

GMCNN method is new CNN-based method with multiple layers for image restoration. As well as the previous CNN method, GMCNN is designed for linear and curved structures which is suitable for our dataset. It expands on the importance of sufficient receptive fields for image inpainting and proposes new loss functions to further enhance local texture details of generated content. It consists of three sub-networks: a generator to produce results, global and local discriminators for adversarial training, and a pretrained VGG network to calculate Implicit Diversified Markov Random Field (ID-MRF) loss.

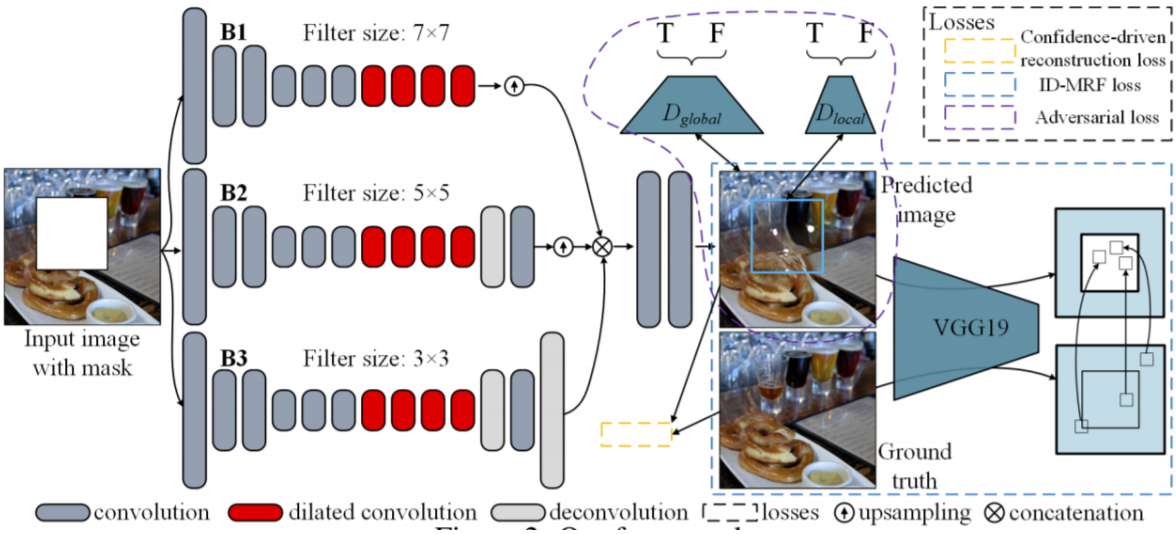


Figure 2: GMCNN Framework from paper "Image inpainting via GMCNN"

Requirements

GMCNN

Project:

https://github.com/tlatkowski/inpainting-gmcnn-keras

Required dependencies:

Keras==2.2.4; pydot==1.4.1; matplotlib==3.0.2; h5py==2.9.0; pillow==6.2.0; opencv-python==3.4.3.18; tqdm==4.31.1; scikit-image==0.14.2

Dataset Preparation

Dataset for this work

In this work, we want to focus only on original images and fish masks in the segmentation folder.

Image dataset

Model was trained with usage of high-resolution images from **DeepFish** dataset. The image dataset consists of 310 images. It can be found here.

Mask dataset

The mask dataset used for model training comes from **Deepfish** dataset. The image dataset consists of 310 images. It can be found here.

Structure of the "data" directory

```
.../Segmentation/
   ||
   |->/Segmentation/training/
   |          |
   |          ->/valid/images/
   |
   -> /Segmentation/evaluation/
             | ->/valid/masks/
```

Structure of the data directory split up in data for training and data for evaluation. In each sub directory there is a directory for the noise data going into the generator and for the real data going into the discriminator. In each of these directories there is also a directory for the masks and for the images (Maybe use the filename to distinguish between mask and image).

Data Cleaning

Eliminating all of the .csv data, folder "localization" and also folder "validation" from the dataset "Deep Fish". For the GMCNN project, the Segmentation folder will be our focus, to be exact the from the images and masks folder, we use only the valid folder which means for each folder there are 310 images available.

Training

Parameter	Value
Epoch	10-50
Batch size	4
Learning rate Generator	0,0001
Image size	256x256
Training time	15h 11m 4s
The total number of parameters	12.562M

Evaluation

We tried to train using different dataset size, but it came out that the large-scale data produced better result than less data. We also trained and tested using few epochs (10, 20, 31, 40 and 50). For each epoch, we trained the data four times. So far, the best result's shown by epoch 50.

Figure 3: left: masked image middle: inpainted image right: original image

Figure 4: left: masked image middle: inpainted image right: original image

Figure 5: left: masked image middle: inpainted image right: original image

Figure 7: left: masked image middle: inpainted image right: original image

Figure 6: left: masked image middle: inpainted image right: original image

From those prediction images, we can see that the GMCNN model filled the fish masks with the informations that the generator gathered based on original images during the training time.

Visualization of the loss functions using Tensorboard

Scalar

Each scalar displays generator, global discriminator, and local losses.
Those losses bring more attention to the image structure at coarse network and restore more detailed image textures at refinement network. They are also helpful to focus on different scales of regions and produce a more realistic structure and details in a restored image.

ID-MRF losses are used to guide the generated feature patches to find their nearest neighbours outside the missing areas as references and these nearest neighbours should be sufficiently diverse such that more local texture details can be simulated.

Figure 8: Visualization of generator and global discriminator losses

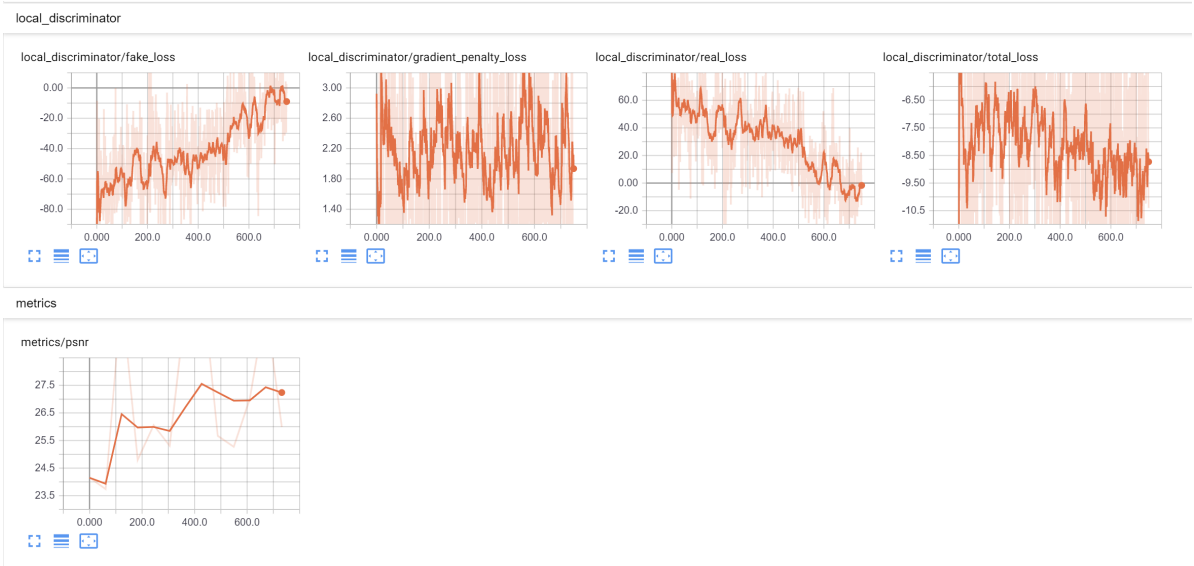
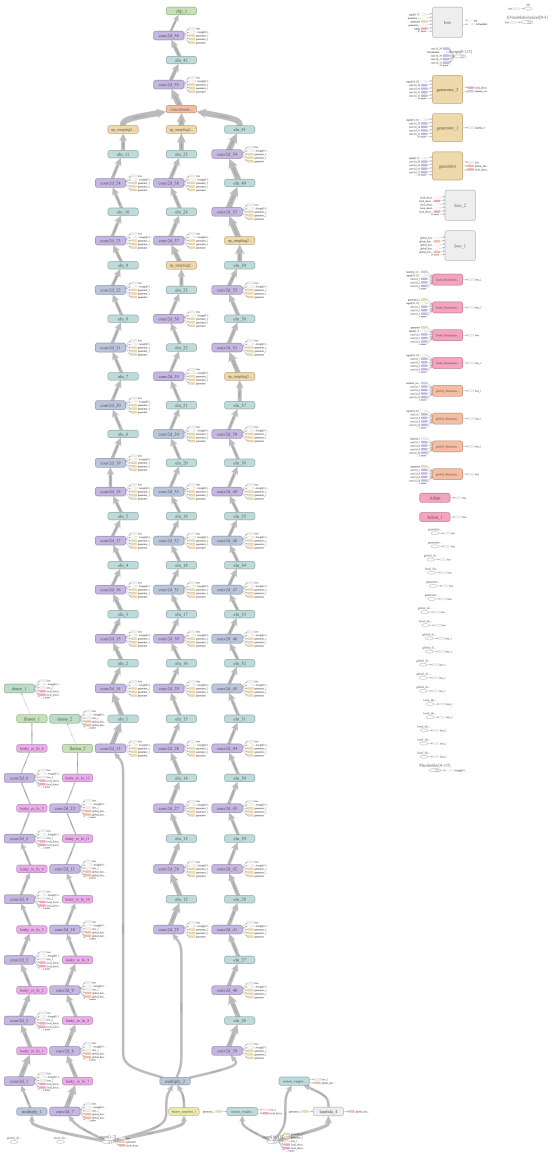


Figure 10: Visualization of local discriminator losses and metric

Graph

This graph shows us the training process and how the convolutional layers are applied for each step.



Limitations

- Each training for those images has long computational time.
- Difficulty dealing with less informations.
- It still suffers from the structural or textural inconsistency problem.
- The inpainted images are still blurred, even after few trainings with the same epoch.

Image Inpainting using High-Resolution GAN

Methodology

Another approach we use for image inpainting is a contextual high-resolution image inpainting algorithm based on a generative adversarial network. The goal of this GAN is to remove objects and replace them with an solid background. To achieve the goal, the GAN begins by applying a mask to the original image and then generates what might be in that region based on the trained ground truth.

Figure 11: left: original image middle: masked image right: inpainted image

Requirements

High-Resolution GAN

Project

https://github.com/wangyx240/High-Resolution-Image-Inpainting-GAN

Requirements

Package	Version
python	3.6
pytorch	1.2.0

References:

https://github.com/zhaoyuzhi/deepfillv2

Dataset Preparation

First, we had to adjust the size of the images, since the high-resolution GAN only worked with images in 512 x 512 pixel format.

Training

The training of the High-Resolution GAN was achieved, by generate a random mask and laying it over the picture. In the next step a picture was generated from noise and given into the generator as noise again. The result of the second generation was applied at the mask resulting in the filled image.

The selected results shown in Evaluation were obtained with the following parameters:

Parameter	Value

Epoch	80
Batchsize	2
Learning rate Generator	0,0001
Image size	512
Worker	0

Time for training was 2:05:45h in a NVIDIA GTX 1660 SUPER with 6GB GDDR6 and Cuda 10.3

Evaluation

Epoch 10

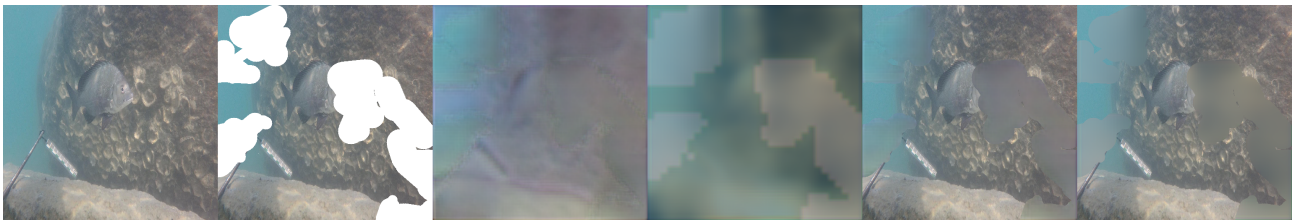


Figure 12: from left: original, masked, step 1, step 2, unknown, original with generated mask

In epoch 10 as it is seen the object removal has a very blurry effect on it and the colorization has only the dominant color of the given grountruth.

Epoch 30

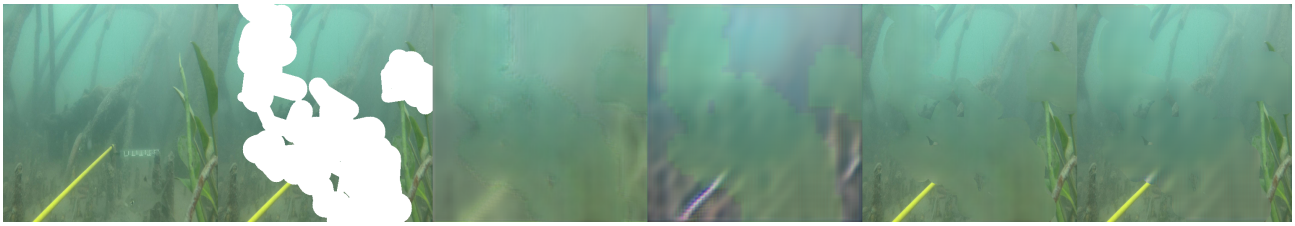


Figure 13: from left: original, masked, step 1, step 2, unknown, original with generated mask

In epoch 30 the blurry effect is less then in epoch 10, but still very visable. The colors of the generated part has a higher variation then as in the 10th epoch.

Epoch 50

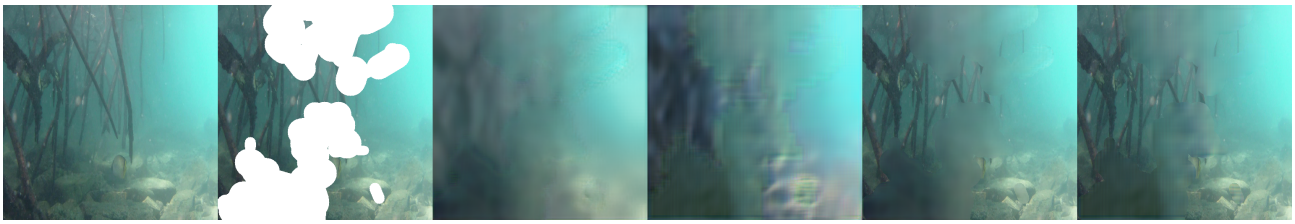


Figure 14: from left: original, masked, step 1, step 2, unknown, original with generated mask

In epoch 50 the blurry effect is less then in epoch 30, but still visable. The colors of the generated part has a higher variation then as in the 30th epoch.

Epoch 70

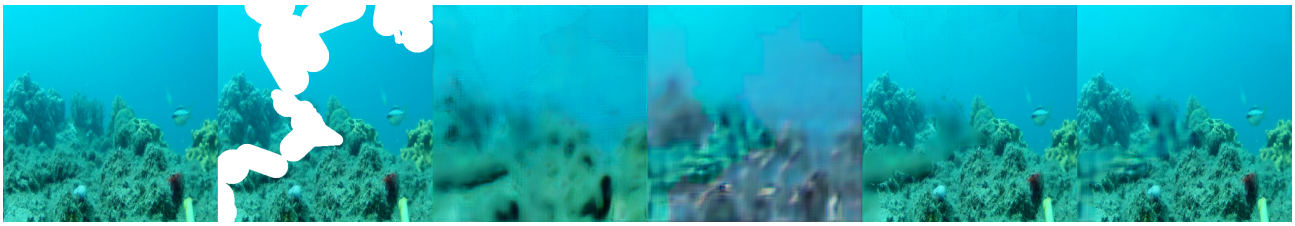


Figure 15: from left: original, masked, step 1, step 2, unknown, original with generated mask

In epoch 70, compared to epoch 50 the structure of objects in the background are clearly visable in the masked image (first of the right).

Epoch 80

Figure 15: from left: original, masked, step 1, step 2, unknown, original with generated mask

In epoch 80 we got the best result, although the generated mask only covers a little area of the object (see limitations tab "Random mask generation").

Limitations

Image size

One of the problems with the high-resolution GAN was that it only accepted images with a size of 512x512 pixels. We tried to modify the project, but errors were displayed as output from pytorch.

Graphic memory

Another problem was that a Nvida GTX 1660 SUPER can apparently only handle a batch_size of 2 due to its 6GB memory.

Random mask generation

However, the real problem with the high-resolution GAN method was that there was no way to apply a custom mask. It uses a free-form mask that is applied to random locations within the image. We tried to modify the project to accept our preset masks. With the result that the pytorch code broke down and produced errors.

Discussion

We showed new version of CNN application with multiple convolution layers and delivered the better ability to inpaint the model with large-scale dataset. We still faced some

problems, like long computational time, difficulty dealing with less information and also the structural or textural inconsistency problem. For the future research, increasing the epoch or batch size to perform better. So far, this method perform well with large-scale dataset. We also demonstrated the high-resolution GAN. The problems, such as limited options of image size, unable to apply the custom masks, are arisen during the training time.

Due to flexibilty and efficiency, it would be better to create new model from scratch or using the other method that aren't tested in this paper. Using high performance engines could improve the computational time that leads to better prediction result.

Conclusion

We presented two different methods which serve different purposes, namely GMCNN for the image restoration and high-resolution GAN for object removal. We have addressed the important problems of representing visual context and using it to generate unknown regions in inpainting using both methods.

In GMCNN, we cleaned the dataset and reduced it to 310 training images and 310 masks. We trained the dataset using different epoch to get the optimal result which in our case epoch 50 and different data size. The inpainted images with large-scale data were still suitable using GMCNN, but not for small-scale data.

In the high-resolution GAN, we trained the data with epoch 80 and also created random masks during the training period. This method has better performance than the previous GAN and gave better prediction result.

References

Paper references

Criminisi, A., Pérez, P., & Toyama, K. (2004). Region filling and object removal by exemplar-based image inpainting. IEEE Transactions on image processing, 13(9), 1200-1212.

Wang, Y., Tao, X., Qi, X., Shen, X., & Jia, J. (2018). Image inpainting via generative multi-column convolutional neural networks. arXiv preprint arXiv:1810.08771.

Yu, J., Lin, Z., Yang, J., Shen, X., Lu, X., & Huang, T. S. (2019). Free-form image inpainting with gated convolution. In Proceedings of the IEEE/CVF International Conference on Computer Vision (pp. 4471-4480).

Elharrouss, O., Almaadeed, N., Al-Maadeed, S., & Akbari, Y. (2020). Image inpainting: A review. Neural Processing Letters, 51(2), 2007-2028.

Laube, P., Grunwald, M., Franz, M. O., & Umlauf, G. (2017). Image inpainting for high-resolution textures using CNN texture synthesis. arXiv preprint arXiv:1712.03111.

Google course for machine-learning especialy GAN (Google Developers)

Introduction to image inpainting with deep learning

Github Links

Generative Multi-column Convolutional Neural Networks inpainting model in Keras

High Resolution Image Inpainting Based on GAN

Image inpainting using Wasserstein GAN

Due to the lack of compareable results in the aspect of quality this implementation of a gan for impainting marine images was not added to the paper itself. But we did not wated our work to go silently into the depths of our harddrives, so we present it in this blog entry alongside some basic informations about gan and wgan.

Its highly recommended to read this page before advancing to the paper itself.

Thought or good to know

Thought or things, that are good to know are shown as an information.

References to the code

References to the code are shown as a warning, you will understand why, when you read the codebase carefully.

Importent understanding

Important understanding are shown as a success hint.

You can find the written code with further information in the page "Codebase".

GAN, Wasserstein - What Is It And What Can It Do?

What is a gan and what can it do and what is its connection to a german word? To answer this questions, we first show what are the elements of a gan. On the one hand a gan has a generator, that generates results from noise.

You can only transform data in the datascience, but not create new one we need a base.

On the other hand we have the discriminator model, its purpose is to distinguise between real images and fake images.

real images are also known as ground truth and the fake images are created by the generator of the gan.

in the code we use -1 as a real image and 1 as a fake image.

As for the training of a gain, there are two steps. One for the discriminator and one for the generator.

Discriminator

First the discriminator is trained using an equal amount of real and fake images.

we didnt trained the discriminator and generator equaly, you will see how we did it when you continue to read.

Generator

Second the generator is trained, its goal is to beat the discriminator.

We used a little trick by creating a combined model looking like this Noise->Generator->Discriminator->Output and for the training we set the discriminator non-trainable and gave in the value pair noise, -1. So the framework adjusts the generators weights to beat the discriminator.

Now that we know what a gan is and how its is trained. We still have to answer what you can do with it. Well you can do a lot from animating animes over letting a president say what you want him to say to repair images or remove objects from them.

> We focus on the last point. Having a member of the politics as a front man is something for lobbyists not us.

Wasserstein sounds fancy and is german, but what has it to do with gan? Well there is a special implementation of gan called wasserstein gan. We can't tell you where the name came from but we can tell you how it is trained.

Before we can show you how it is trained, we need to add, that the WGAN has a different loss function and uses another optimizer.

> The specific optimizer and loss function for wgan can be seen in the codebase. We don't go into detail about loss functions and optimizers or else we could write a book instead of a blog entry.

Training a wgan is basiclly the same as training a normal gan, except for the fact, that a wgan trains the discriminator multiple times before the generator. Combined with the different loss function and optimizer its more efficient.

> Atleast thats how we interpretated the original wgan paper.

> So it can be said, that the generator has a better trained discriminator as oppent in the wgan implementation.

Inpainting Images Using WGAN - (How Good) Does It Work?

Now we want to see wgan in action. For this we are using the segmentation part of the deepfish dataset. It contains images extracted from variouse underwater videos. Split into

original images, maskes from the fishes location and empty images. We prepared the dataset by downscaling every image to a size of 512x512.

> To train our wgan we first decided if we want to remove or insert the fish, based on the decision we used the empty or original image as groundtruth.

We choose to try and remove the fish from the image. To have a visable result we composit the original image with the gernerated image based on the mask.

Epoch 600

Masked images at epoch 600

You clearly can the, that the noise is random. Especially at the fact, taht the colors like red and green jump into the eye with this background.

Epoch 1000

Masked image at epoch 1000

Compared to the previous image it can be said, that the noise was processed to look more like the background. Although you still can clearly see the shape of the fish.

Training a plain (w)gan takes a lot of time.

We know the question, that comes up now is:

> Is there a way to improve the wgan?

The short answer is yes there are several.

For example by adding another discriminator, that checks the context of the composited image. But we wanted to stick with the basic implementation here to give you an overview on gan especially wgan, with the focus on what they are and how they are trained.

Another example would be using two levels of generation. One from noise and the other from the result of the first level.

Codebase - Like Github But You Have To Work

For The Brave - The Requirements

We hope you are ready for your journey through the forest of dispair and versions.

> We do not take any kind of responsibility for mental, hardware or software damage you cause by completing or trying to install or use the programs below.

Program	Version	Link
NVIDIA Drivers	most recent	Here
NVIDIA CUDA Toolkit	v11.3	Here
CuDNN	8.1.1.33	Here

> Don't forget to add all the libaries you need to your path variable.

Congratulation you have survived the non-python obsticals.

Module	Version
Anaconda (including science stack)	most recent
Python	3.8.5
Pip	20.2.4
Tensorflow GPU	2.4.1

Pillow	8.0.1

> **Congratulation from now on, if there is a problem all you need to do is to consult stackoverflow or channel your inner geek.**

For The Organized - The Project Structure

Main structure of the Wasserstein-GAN implementation for the inpainting of marine images.

```
├──data                     /* Preprcessed data
│   ├──empty                   Same structure as the in-directory*/
│   ├──mask
│   └──valid
├──in                       /* Input directory for the raw images */
│   ├──empty
│   ├──mask
│   └──valid
├──models                   /* Location to store/load the trained models *
│   ├──Discriminator
│   └──Generator
├──out                      /* Output directory for the metrics and sample
    ├──logs                    including the
    └──samples
```

For The Prepared - Preprocessing

After an informativ message is displayed, the preprocessing walks each of the three directories in the input folder and saves a rescaled (512x512) version in the encapsulated data folder. The main paradigm for this function was, that it would not alter the base dataset, but create a new object for each corresponding object in the dataset.

```python
from PIL import Image
import os


def pre_processing():
    print("Preprocessing input")
    in_dir = "in/valid"
    data_dir = "data/valid"
    counter = 0
    if not os.path.exists(data_dir):
        os.makedirs(data_dir)
        for file in os.listdir(in_dir):
            img = Image.open(os.path.join(in_dir, file))
            img = img.resize((512, 512), Image.ANTIALIAS)
            img.save(os.path.join(data_dir, str(counter) + ".jpg"), "JPEG"
            counter += 1
    in_dir = "in/empty"
    data_dir = "data/empty"
    counter = 0
    if not os.path.exists(data_dir):
        os.makedirs(data_dir)
        for file in os.listdir(in_dir):
            img = Image.open(os.path.join(in_dir, file))
            img = img.resize((512, 512), Image.ANTIALIAS)
            img.save(os.path.join(data_dir, str(counter) + ".jpg"), "JPEG"
            counter += 1
    counter = 0
    in_dir = "in/mask"
    data_dir = "data/mask"
    if not os.path.exists(data_dir):
        os.makedirs(data_dir)
        for file in os.listdir(in_dir):
            img = Image.open(os.path.join(in_dir, file))
            img = img.resize((512, 512), Image.ANTIALIAS)
            img.save(os.path.join(data_dir, str(counter) + ".png"), "PNG")
            counter += 1
```

For The Main Event - The Model & Its Constraint

Clip Constraint

Based on the Wasserstein-Paper each weight in the discriminator, except for weights in the batch normalization, has to be clipped during the training process. To achive this we created a class, which inherits the methods of the contraint class. Its constructor takes in a clip value in each call the class object uses the clip function from keras backend to clip each weight of an layer.

```
from tensorflow.keras.constraints import Constraint
from tensorflow.keras.backend import clip,


class ClipConstraint(Constraint):

    def __init__(self, clip_value):
        self.clip_value = clip_value

    def __call__(self, weights):
        return clip(weights, -self.clip_value, self.clip_value)

    def get_config(self):
        return {"clip_value": self.clip_value}
```

WGAN Model

You reached the heart, or more likely the brain, of the project the model for our wasserstein GAN implementation.

The WGAN model is a composit of a generator and a discriminator. We created a basic model for each of these two (*create_generator* and *create_discriminator*). And compiled them with the optimizer and loss function presented in the Wasserstein GAN paper.

A side from the basic generate method, which prepares noise and encapulates the predict method of the tensorflow Model object we are using as a generator. We implemented the training method. For each of the epochs we iterate over the batches of given images. First we train the discriminator, for that we created an equal amount of images with the generator compared to the amount of real images. After this the discriminator is trained 5 times as mentioned in the paper.

Following the distrciminator we trained the generator with a simple trick. We created a combined model, in which we set the submodel of the discriminator not trainable. So the train method provided by tensorflow will only adjust the weights of the generator if the combined model is trained.

The means of the metrics are stored in a list of dictionaries in the format:

```
"epoch": 1, "g_loss": 35957141.2195036, "d_loss": -17825683.26635361}
```

For The Final Step - Postprocessing

In the postprocessing process the original image and the result are composite using the mask. The white spaces in the mask are filled with the result and the black ones are filled with the original picture.

```python
from PIL import Image


def post_processing(result, original, mask):
    return Image.composite(result, original, mask)
```

The Whole Code

preocessing.py

```
from PIL import Image
import os
import random


def pre_processing():
    print("Preprocessing input")
    in_dir = "in/valid"
    data_dir = "data/valid"
    counter = 0
    if not os.path.exists(data_dir):
        os.makedirs(data_dir)
        for file in os.listdir(in_dir):
            img = Image.open(os.path.join(in_dir, file))
            img = img.resize((512, 512), Image.ANTIALIAS)
            img.save(os.path.join(data_dir, str(counter) + ".jpg"),
            counter += 1
    in_dir = "in/empty"
    data_dir = "data/empty"
    counter = 0
    if not os.path.exists(data_dir):
        os.makedirs(data_dir)
        for file in os.listdir(in_dir):
            img = Image.open(os.path.join(in_dir, file))
            img = img.resize((512, 512), Image.ANTIALIAS)
            img.save(os.path.join(data_dir, str(counter) + ".jpg"),
            counter += 1
    counter = 0
    in_dir = "in/mask"
    data_dir = "data/mask"
    if not os.path.exists(data_dir):
        os.makedirs(data_dir)
        for file in os.listdir(in_dir):
            img = Image.open(os.path.join(in_dir, file))
            img = img.resize((512, 512), Image.ANTIALIAS)
            img.save(os.path.join(data_dir, str(counter) + ".png"),
            counter += 1


def one_image(img_path="data/valid", mask_path="data/mask", valid_pa
    files = os.listdir(img_path)
    file = random.choice(files)
    img = Image.open(os.path.join(img_path, file)).convert("RGB")
```

```
    mask = Image.open(os.path.join(mask_path, file.replace(".jpg", "
    return mask, img


def post_processing(result, original, mask):
    return Image.composite(result, original, mask)


if __name__ == "__main__":
    pre_processing()

```

model.py

```
from tensorflow.keras.constraints import Constraint
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import Dense, BatchNormalization, Conv2
from tensorflow.keras.backend import clip, mean
from tensorflow.keras.initializers import RandomNormal
from tensorflow.keras.preprocessing.image import array_to_img, img_t

from processing import post_processing, one_image
from PIL import Image

import numpy as np
import os
import json


class ClipConstraint(Constraint):

    def __init__(self, clip_value):
        self.clip_value = clip_value

    def __call__(self, weights):
        return clip(weights, -self.clip_value, self.clip_value)

    def get_config(self):
        return {"clip_value": self.clip_value}


class WGAN:

    def __init__(self):
        self.img_height = 512
        self.img_widht = 512
        self.num_channels = 3
        self.n_critics = 5
        self.clip_value = 0.01

        self.generator = self.create_generator()
        self.discriminator = self.create_discriminator()
        self.combined = self.create_combined()
        self.evaluator = list()

    def wasserstein_loss(self, y_true, y_pred):
```

```
        return mean(y_true * y_pred)

    def summary(self):
        print("Generator")
        self.generator.summary()
        print("\nDiscriminator")
        self.discriminator.summary()
        print("\nCombined")
        self.combined.summary()

    def create_combined(self):
        for layer in self.discriminator.layers:
            if not isinstance(layer, BatchNormalization):
                layer.trainable = False
        model = Sequential(name="Combined")
        model.add(self.generator)
        model.add(self.discriminator)

        opt = RMSprop(learning_rate=0.000005)
        model.compile(optimizer=opt, loss=self.wasserstein_loss, met
        return model

    def create_generator(self):
        if os.path.exists("models/Generator"):
            model = load_model("models/Generator", compile=False)
            opt = RMSprop(learning_rate=0.000005)
            model.compile(optimizer=opt, loss=self.wasserstein_loss,
            return model

        init = RandomNormal(stddev=0.2)

        model = Sequential(name="Generator")
        model.add(Dense(3, kernel_initializer=init, input_shape=(512
                        activation="relu"))
        model.add(Conv2D(256, kernel_size=4, padding="same"))
        model.add(BatchNormalization(momentum=0.8))
        model.add(AveragePooling2D())
        model.add(Activation("relu"))
        model.add(Conv2D(128, kernel_size=4, padding="same"))
        model.add(BatchNormalization(momentum=0.8))
        model.add(Activation("relu"))
        model.add(UpSampling2D())
        model.add(Conv2D(3, kernel_size=4, padding="same"))
        model.add(Activation("tanh"))

        opt = RMSprop(learning_rate=0.00005)
        model.compile(optimizer=opt, loss=self.wasserstein_loss, met
        return model

    def create_discriminator(self):
        if os.path.exists("models/Discriminator"):
```

```
            model = load_model("models/Discriminator", compile=False
            opt = RMSprop(learning_rate=0.000005)
            model.compile(optimizer=opt, loss=self.wasserstein_loss,
            return model
        const = ClipConstraint(self.clip_value)
        init = RandomNormal(stddev=0.2)

        model = Sequential(name="Discriminator")
        model.add(Conv2D(filters=64, kernel_size=5, strides=(2, 2),
        model.add(LeakyReLU())
        model.add(Conv2D(filters=128, kernel_size=5, strides=(2, 2),
        model.add(LeakyReLU())
        model.add(Conv2D(filters=256, kernel_size=5, strides=(2, 2),
        model.add(LeakyReLU())
        model.add(Conv2D(filters=512, kernel_size=5, strides=(2, 2),
        model.add(LeakyReLU())
        model.add(Conv2D(filters=256, kernel_size=5, strides=(2, 2),
        model.add(LeakyReLU())
        model.add(Conv2D(filters=128, kernel_size=5, strides=(2, 2),
        model.add(LeakyReLU())
        model.add(Flatten())
        model.add(Dense(units=1))

        opt = RMSprop(learning_rate=0.00005)
        model.compile(optimizer=opt, loss=self.wasserstein_loss, met
        return model

    def generate(self):
        noise = np.random.uniform(-1, 1, (1, 512, 512, 3))
        result = self.generator.predict(noise)
        result = np.multiply(result, 127.5)
        result = np.add(result, 127.5)
        result = result[0]
        return array_to_img(result)

    def train(self, batch_size=0, epochs=50):
        print("Start Training")
        valid_path = "data/empty"
        data = np.array([img_to_array(Image.open(os.path.join(valid_
                         for file in os.listdir(valid_path)])
        data = np.subtract(data, 127.5)
        data = np.divide(data, 127.5)
        batch_size = batch_size if batch_size > 0 else data.shape[0]

        for epoch in range(epochs):
            real_epoch = epoch+1
            print(f"Epoch {real_epoch}/{epochs}")
            dis_curr = list()
            gen_curr = list()
            iterations = range(int(data.shape[0]/batch_size))
            for index in iterations:
```

```
                print(f"\tIteration {index + 1}/{max(iterations) + 1
                noise = np.random.uniform(-1, 1, (batch_size, 512, 5
                d_curr = list()
                for _ in range(self.n_critics):
                    self.discriminator.trainable = True
                    img_batch = data[index * batch_size:(index+1) *
                    generated = self.generator.predict(noise, verbos
                    x = np.concatenate((img_batch, generated))
                    y = np.concatenate((-np.ones((batch_size, 1)), r
                    dis_loss = self.discriminator.train_on_batch(x,
                    d_curr.append(sum(dis_loss) / len(dis_loss))
                d_loss_curr = sum(d_curr) / len(d_curr)

                real = -np.ones((batch_size, 1))
                self.discriminator.trainable = False
                gen_loss = self.combined.train_on_batch(noise, real)
                self.discriminator.trainable = True
                g_loss_curr = sum(gen_loss) / len(gen_loss)

                print(f"\t[g_loss]: {g_loss_curr} [d_loss] {d_loss_c
                dis_curr.append(d_loss_curr)
                gen_curr.append(g_loss_curr)

            g_loss = sum(gen_curr) / len(gen_curr)
            d_loss = sum(dis_curr) / len(dis_curr)
            print(f"[g_loss]: {g_loss} [d_loss] {d_loss}\n")
            metrics = {"epoch": real_epoch, "g_loss": g_loss, "d_los
            self.evaluator.append(metrics)

            if (epoch + 1) % 10 == 0:
                mask, img = one_image()
                generated = self.generate()
                result = post_processing(generated, img, mask)
                sample = Image.new("RGB", (img.width + generated.wid
                sample.paste(img, (0, 0))
                sample.paste(generated, (img.width, 0))
                sample.paste(result, (img.width+generated.width, 0))
                sample.save(f"out/samples/SampleEpoch{real_epoch}.pn
                self.save()

    def save(self):
        if not os.path.exists("models"):
            os.makedirs("models")
            os.makedirs("models/Generator")
            os.makedirs("models/Discriminator")
        self.generator.save("models/Generator")
        self.discriminator.save("models/Discriminator")
        json.dump(self.evaluator, open("out/logs/metrics.json", "w")


if __name__ == "__main__":
```

```
197     wgan = WGAN()
198     wgan.train(epochs=50000, batch_size=2)
199
```