Disaster Image Inpainting And Classification Using GAN and CNNs

GROUP E4

Rajat Payghan

(2021A7PS2218P)

Rudra Goyal

(2021A7PS0708P)

Akshat Bajpai

(2021A7PS0573P)

In the fulfillment of Project Milestone #2 of CS F425: Deep Learning

Instructor : Prof. Pratik Narang

Department of CSIS, BITS Pilani Pilani,India

Table of Contents

Abstract

Natural disasters such as earthquakes, floods, hurricanes, wildfires, and landslides often leave behind a trail of destruction captured by drone footage. This project explores an approach to classify disaster images through a combined framework of Generative Adversarial Networks (GANs) for image inpainting and Convolutional Neural Networks (CNNs) for subsequent classification.

The first phase involves utilizing GANs to in-paint corrupted input images obtained from drone footage, effectively reconstructing missing or damaged regions. The GAN-based inpainting process enhances the visual fidelity of the images, allowing for more accurate and informative analysis. The inpainted images are then subjected to a CNN-based classification system in the second phase. The CNN model is trained to distinguish between different types of natural disasters, namely earthquakes, floods, hurricanes, wildfires, and landslides.

Inpainting Architecture

2.0 Overview

We have utilized a Generative Adversarial Network (GANs) to in-paint the corrupted image. The generator part of the GAN contains an encoder and a decoder which helps remove the corrupted parts of the image. The image with dimensions 256 x 256 x 3 is encoded into a latent space with 100 parameters and then is decoded back to a 256 x 256 x 3 image. We have trained a CNN with ReLU as activation function for the encoder and decoder. We have utilized batchnorm to normalize the outputs on each layer and increase the speed of training.

2.1 Architecture

Generator → Encoder

(0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))

(1): ReLU(inplace=True)

(2): BatchNorm2d(64, eps=1e-05, momentum=0.1)

(3): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))

(4): ReLU(inplace=True)

(5): BatchNorm2d(128, eps=1e-05, momentum=0.1)

(6): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))

(7): ReLU(inplace=True)

(8): BatchNorm2d(256, eps=1e-05, momentum=0.1)

(9): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))

(10): ReLU(inplace=True)

(11): BatchNorm2d(512, eps=1e-05, momentum=0.1)

(12): Flatten(start_dim=1, end_dim=-1)

Generator → Decoder

(0): Linear(in_features=100, out_features=131072, bias=True)

(1): ReLU(inplace=True)

(2): Unflatten(dim=1, unflattened_size=(512, 16, 16))

(3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))

(4): ReLU(inplace=True)

(5): BatchNorm2d(256, eps=1e-05, momentum=0.1)

(6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))

(7): ReLU(inplace=True)

(8): BatchNorm2d(128, eps=1e-05, momentum=0.1)

(9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))

(10): ReLU(inplace=True)

(11): BatchNorm2d(64, eps=1e-05, momentum=0.1)

(12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))

(13): Tanh()

Discriminator → Global

(0): Conv2d(6, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))

(1): LeakyReLU(negative_slope=0.2, inplace=True)

(2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))

(3): LeakyReLU(negative_slope=0.2, inplace=True)

(4): BatchNorm2d(128, eps=1e-05, momentum=0.1)

(5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))

(6): LeakyReLU(negative_slope=0.2, inplace=True)

(7): BatchNorm2d(256, eps=1e-05, momentum=0.1)

(8): Flatten(start_dim=1, end_dim=-1)

(9): Linear(in_features=262144, out_features=1, bias=True)

(10): Sigmoid()

Discriminator → Local 64 x 64

(0): Conv2d(6, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))

(1): LeakyReLU(negative_slope=0.2, inplace=True)

(2): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))

(3): LeakyReLU(negative_slope=0.2, inplace=True)

(4): BatchNorm2d(128, eps=1e-05, momentum=0.1)

(5): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))

(6): LeakyReLU(negative_slope=0.2, inplace=True)

(7): BatchNorm2d(256, eps=1e-05, momentum=0.1)

(8): Flatten(start_dim=1, end_dim=-1)

(9): Linear(in_features=16384, out_features=1, bias=True)

(10): Sigmoid()

Loss Functions

1. Reconstruction Loss

   For the reconstruction loss we have applied L1 loss which is the mean absolute difference between corresponding pixels in the in painted and original images. The reconstruction loss measures the pixel-wise difference between the inpainted images (generated by the generator) and the original images.

2. Global Discriminator Loss

   The global discriminator loss is calculated using binary cross-entropy loss. It measures how well the global discriminator can distinguish between real and generated (inpainted) images.

3. Local Discriminator Loss

   For each patch, the local discriminator evaluates its ability to discriminate between real and generated patches. The binary cross-entropy loss is computed for each

patch. The local discriminator loss is computed by iterating over local patches of the images.

4. Total GAN Loss
   The total GAN loss is computed as the sum of the above losses.

Training

We have used three lambda hyperparameters

1. lambda_recon : This hyperparameter scales the contribution of the reconstruction loss in the total GAN loss.
2. lambda_global : Scales the contribution of the global discriminator loss in the total GAN loss.
3. lambda_local : Scales the contribution of the local discriminator loss in the total GAN loss.

We have used Adam Optimiser to dramatically improve the model and train it quickly. The hyperparameters used are

1. For Generator: lr=0.0002, betas=(0.5, 0.999)
2. For Discriminator: lr=0.00001, betas=(0.5, 0.999)
3. For Local Discriminator: lr=0.0002, betas=(0.5, 0.999)

2.2 Design Choices

Encoder and Decoder

1. Convolutional Neural Network
   We have used convolution operations on the input image. In this case, the kernel size is (4, 4) with a stride of (2, 2), which means the convolutional operation uses a 4x4 filter and moves with a step of 2 pixels in both the horizontal and vertical

directions. This design choice helps in downsampling the spatial dimensions of the input, reducing the computational load and extracting higher-level features.

2. ReLU activation

   We have used ReLU as the activation function because it introduces non-linearity to the model and helps improve the convergence of the model.
3. Batch Normalization

   Batch Normalization normalizes the inputs of a layer, reducing internal covariate shift and accelerating the training process. It helps stabilize and speed up the training by normalizing the activations of each layer.

4. Number of Channels

   The number of output channels progressively increases from 64 to 512. This allows the network to capture increasingly complex and abstract features as it goes deeper into the encoder.

5. Padding

   Padding=(1, 1) is employed, maintaining the spatial dimensions by adding a border of zeros around the input. This helps preserve information at the borders during convolution.

Local and Global Discriminator

1. Convolution 2d layers

   The Conv2d layers are used for spatial feature extraction. The kernel size is (4, 4) with a stride of (2, 2), and padding=(1, 1) for global. This configuration reduces the spatial dimensions while increasing the number of channels, enabling the discriminator to capture hierarchical features.

2. LeakyReLU activation

LeakyReLU introduces a small slope for negative input values, preventing the complete suppression of information during training. A negative slope of 0.2 is applied here.

3. Batch Normalization
   Batch Normalization is applied after the second convolutional layer (index 4). It helps stabilize and accelerate the training process by normalizing the activations.
4. Sigmoid activation
   The Sigmoid activation function is applied to squash the output of the linear layer to the range [0, 1], producing a probability score.

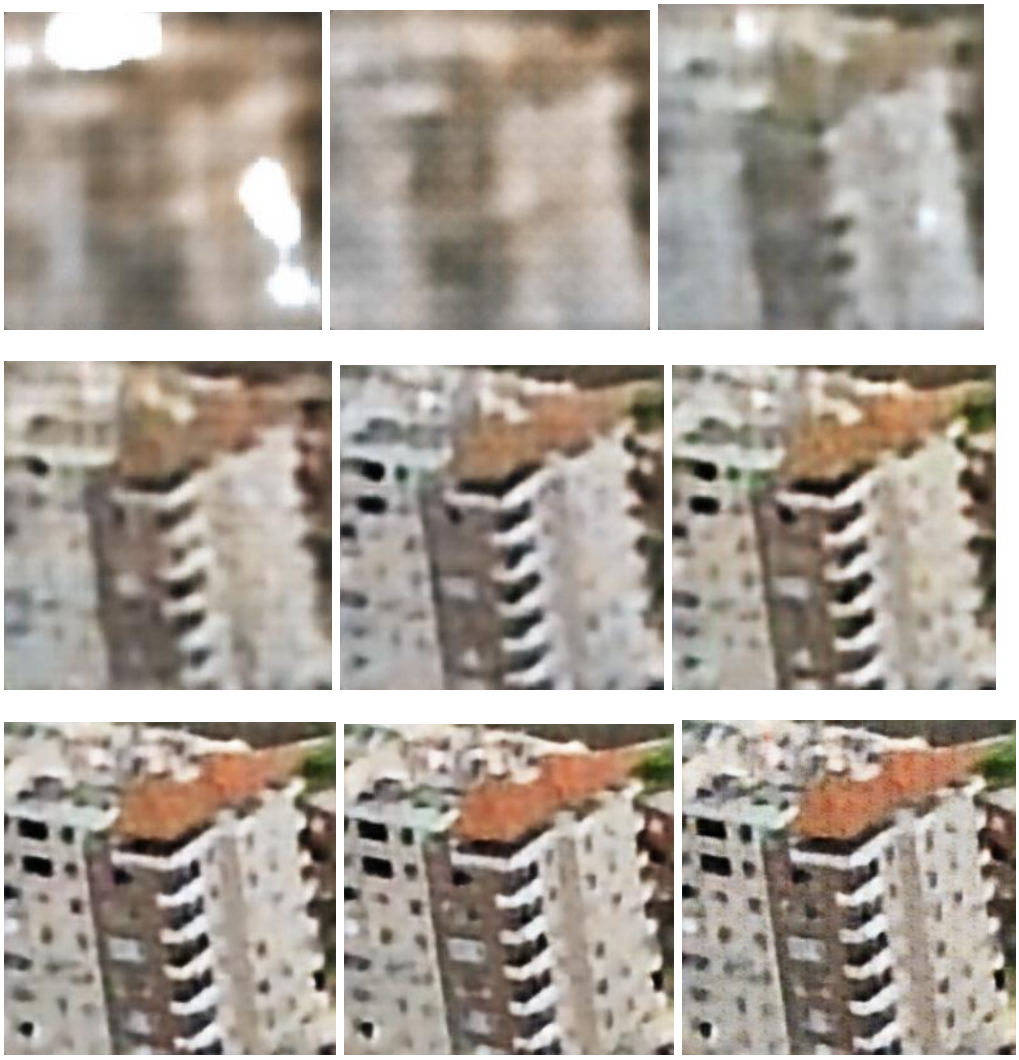
2.3 Experiments

We experimented with different discriminator architecture to come up with the best possible in-painted output image.

Firstly, we had not used batch norm in the generator in the earlier versions of our training. This caused the values to reach too high / low values which caused the activation function tanh() to output extreme values. This caused the first versions to output a pure black image.

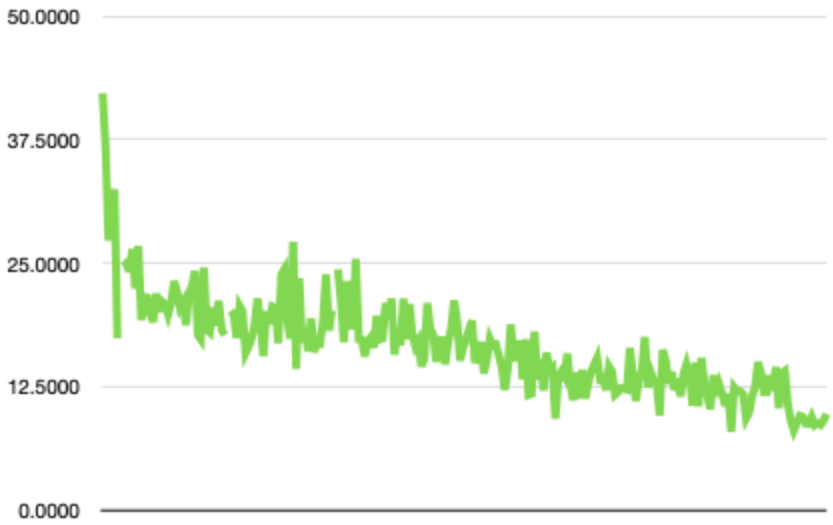
Next, we had used a local discriminator for a 16 x 16 patch. This caused 256 patches per image which had to be trained, which turned out to be exponentially slow. We reduced the 64 x 64 patch, causing an increase in training speed. But, the image quality degraded and did not improve significantly even after 100 epochs. The following images show the progress of the image when we added 64 x 64 patch for the local discriminator.

Now we had three hyperparameters to work with, lambda_recon, lambda_global, lambda_local. So as the local discriminator was messing up the learning, we reduced lambda_local to 0.0000001, let the model learn for over 100 epochs. Then we increased the lambda_local to 1 and let the model train for 300 more epochs. We noticed that not focussing on local patches early on helped us get a jumpstart on reconstructing the general shape and then improve upon the localized details later on. The following images show the training of the model over a span of 300 epochs.



2.4 Performance Overview

The loss during training for the generator exhibits a decreasing trend over epochs, indicating successful model convergence. The initial loss at epoch 0 is significantly higher (45.96), but it steadily decreases to lower values, reaching 9.41 at epoch 200. The fluctuations within each epoch's steps show the model's adaptability. Key Performance Indicators (KPIs) such as convergence stability and steady loss reduction are positive.



This graph shows the declining loss over the span of 200 epochs

Average PSNR : ~25

SSIM : 0.43

Classification Architecture

3.0 Overview

As the task was image classification based, we have used a simplified implementation of the ResNet-50 architecture. The BasicBlock class defines a basic residual block used in the ResNet architecture. It consists of two convolutional layers with batch normalization and ReLU activation functions.The ResNet class defines the overall ResNet model. The model consists of four stages (layer1 to layer4), each containing multiple residual blocks. The channels in these layers double every layer, starting from 64 and ending at 512

3.1 Architecture

BasicBlock

(0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)

(1): BatchNorm2d(64, eps=1e-05, momentum=0.1)

(2): ReLU(inplace=True)

(3): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)

(4): BatchNorm2d(64, eps=1e-05, momentum=0.1)

ResNet Architecture

(0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)

(1): BatchNorm2d(64, eps=1e-05, momentum=0.1)

(2): ReLU(inplace=True)

(3): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)

(4): LayerOne(block, 64, layers[0])

(5): LayerTwo(block, 128, layers[0])

(6): LayerThree(block, 256, layers[0])

(7): LayerFour(block, 512, layers[0])

(8): AveragePool

(9): Fully Connected Layer

3.2 Design Choices

1. Residual Blocks: The use of residual blocks allows for the training of very deep networks by mitigating the vanishing gradient problem.
2. Batch Normalization: Batch normalization is applied after convolutional layers to stabilize and accelerate training.
3. Adaptive Average Pooling: Global average pooling is used to reduce spatial dimensions before the fully connected layer, providing a fixed-size output irrespective of the input size.

We chose to use four layers as it's a common architecture used in ResNet-50 architecture.

2.3 Experiments

Previously, the VGG net architecture was employed in our project. In the architecture, we noticed that we were not getting good performance in terms of both time efficiency and convergence over epochs. Looking further into the problem, we realized that there were possible vanishing gradients, which may have caused stagnation of performance. We experimented with changing the model and tuning the hyperparameters, but to no avail.

Subsequently, a decision was made to transition the architecture to ResNet-50, an established architecture for image classification to circumvent the challenges associated with vanishing gradients. This provided a much needed boost in performance of the

model. Due to residual connections, we got a faster training model and better efficiency in training.

2.3 Performance Overview

We reached a training accuracy of 0.38 for the validation set.