DEEP LEARNING FINAL PROJECT

Kashyap Shekhada: 23PGAI0102

Lakshya Soni: 23PGAI00117

Minal Sheth: 23PGAI0079

Solomon Odum: 23PGAI00119

Pokémon generation using GAN (Generative Adversarial Network)

We have tried to generate Pokémon by using a generative adversarial network (GAN) trained on a dataset of the 809 existing Pokémon species. By using this deep learning technique, we hope to create unique and diverse Pokémon**.**

Here is the link of the research paper we refer https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf

We have refered many codes for how to applying GAN properly and for hyper parameter optimization.

We have tried to change different hyperparameter to check the effects on the result. Some common hyperparameters in GAN that we have tried to change are learning rate, batch size, and the number of training epochs.

Introduction: -

Pokemon, short for "Pocket Monsters," are fictional creatures that have been popularized through video games, trading cards, and other media. There are over 800 known Pokemon species, each with its own unique characteristics and abilities.

Our goal was to build a GAN model that could generate new Pokemon characters that are similar to those in the official Pokemon franchise. We collected a dataset of Pokemon sprites and trained a GAN model on this dataset. The model learned to generate synthetic Pokemon sprites that capture the style and appearance of the original characters.

Dateset:-https://www.kaggle.com/datasets/vishalsubbiah/pokemon-images-and-types?datasetId=92703&searchQuery=gan

Architecture

We have used following architecture for Generator and Discriminator.

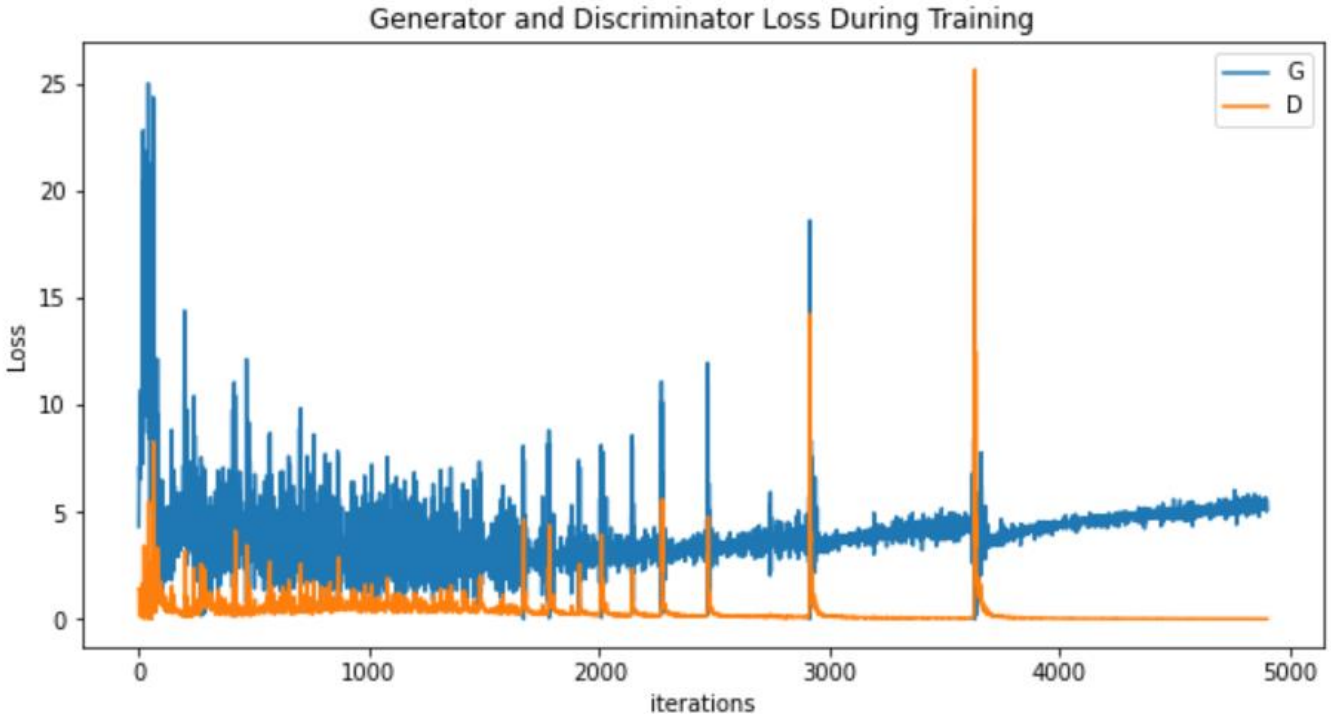
```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
```

```
 Discriminator(
   (main): Sequential(
     (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
     (1): LeakyReLU(negative_slope=0.2, inplace=True)
     (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
     (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
     (4): LeakyReLU(negative_slope=0.2, inplace=True)
     (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
     (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
     (7): LeakyReLU(negative_slope=0.2, inplace=True)
     (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
     (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
     (10): LeakyReLU(negative_slope=0.2, inplace=True)
     (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
     (12): Sigmoid()
   )
 )
```

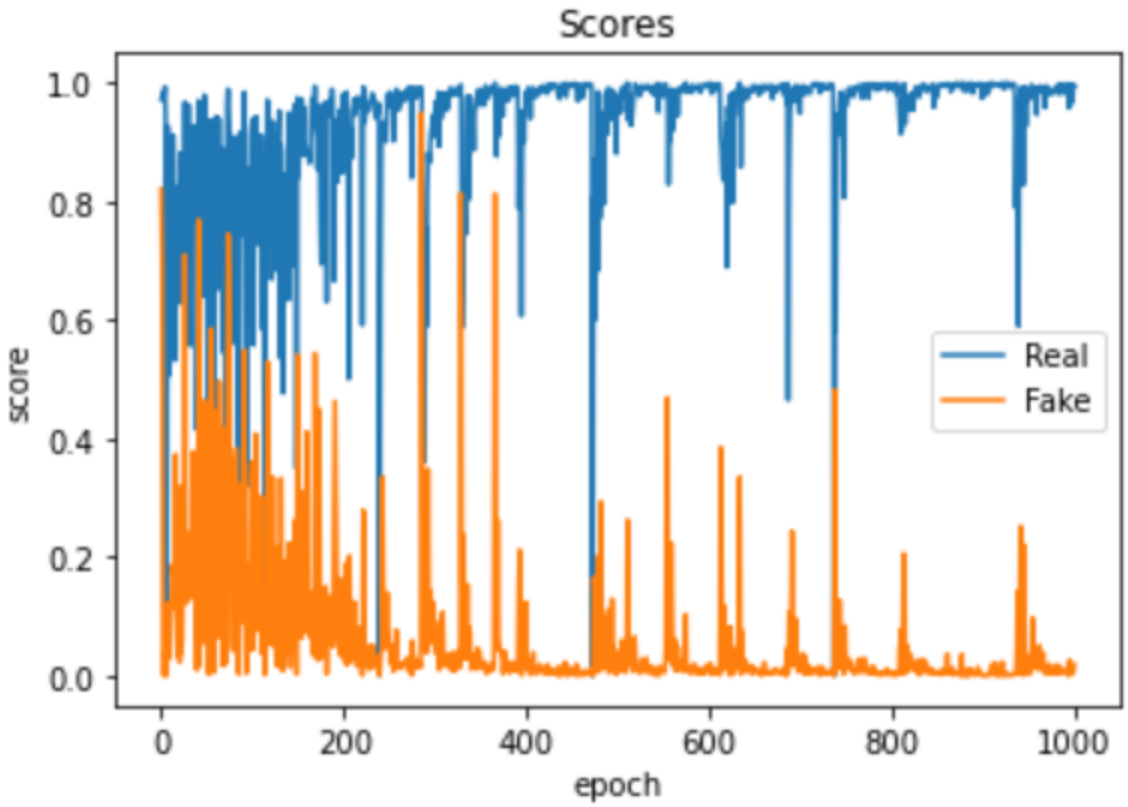
- Use batch norm in both the generator and the discriminator.
- Use ReLU activation in generator for all layers except the output which uses Tanh.
- Use LeakyReLU activation in the discriminator for all layers except the output which uses Sigmoid.
- Use Adam optimizer with a learning rate of 0.0002.

Loss during training

Generator and Discriminator loss plotted below.



Real VS Fake score graph plotted below.



Final Results: -

Input Image

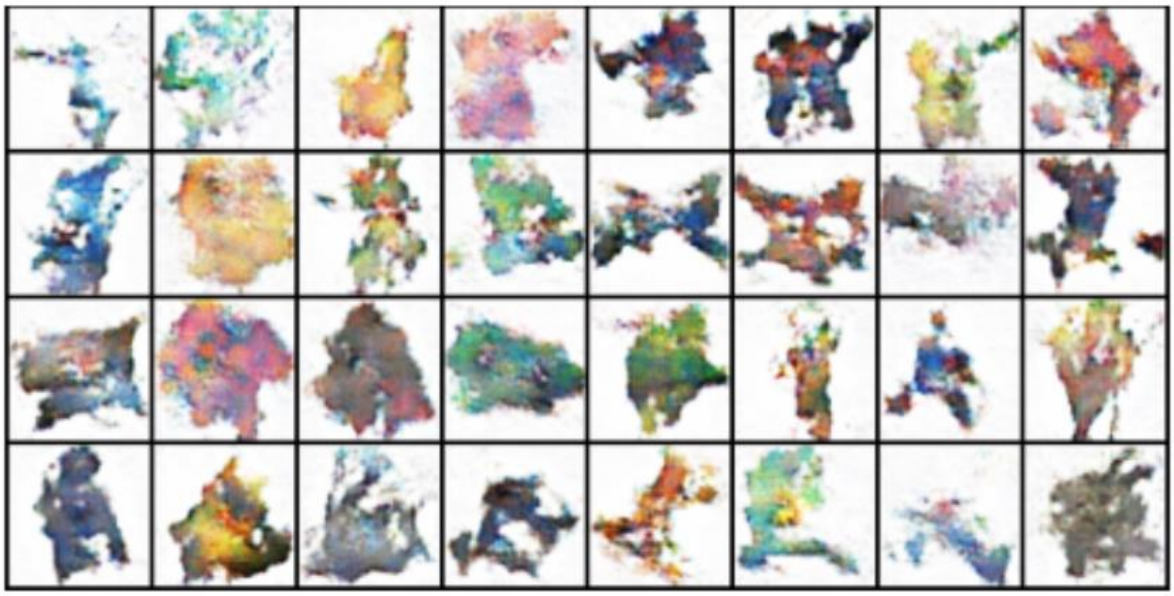
The following result was generated on running for 1000 epochs with 128 minibatch size using 64x64 images.

Real Image

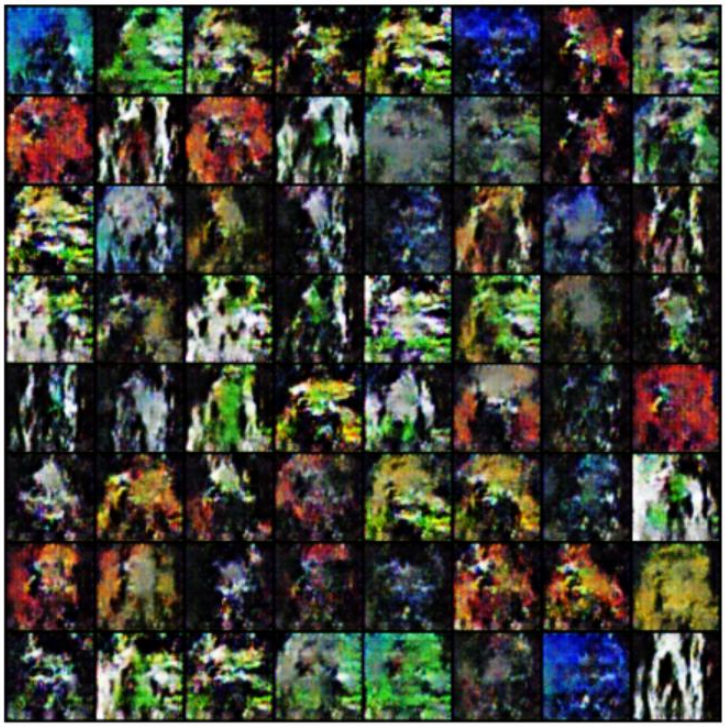
Fake Image

Input Image

Result was generated on running for 700 epochs with 64 minibatch size using 64x32 images.



We have also tried to apply the same code for the Ben 10 alien's dataset and followings are the result.



One probable reason for the poor performance on the Ben-10 dataset could be the inconsistency of the background in the images. In Pokémon dataset each image has white background, whereas the lack of consistent background in the Ben-10 dataset may make it more challenging for a CNN model to identify and extract relevant features. Pre-processing the data by standardizing the background of the images in the Ben-10 dataset could potentially improve the results of the algorithms.

Conclusion: -

Optimizing the values of these hyperparameters can help improve the performance of the GAN in terms of both training speed and quality of the generated samples. For example, setting a higher learning rate may allow the model to converge faster, but it may also make the training process more unstable. Similarly, increasing the batch size can improve the stability of the training process, but it may also require more computational resources.