CSC 487 Final Project Proposal

Project:
PokéGAN: Generating Original Pokémon Using a GAN

Team:
Lucas Summers (lsumme01@calpoly.edu)
Braeden Alonge (balonge@calpoly.edu)

Problem Statement:
Our project aims to generate new sprite images of Pokémon-style creatures using a Generative Adversarial Network (GAN) trained from scratch. Rather than training a model to identify existing Pokémon given their image representation, our model will learn the different attributes that give Pokémon their unique style, including color schemes, outlines, shading, patterns, and body structure. Our goal is to generate images such that we will successfully be able to make real Pokémon indistinguishable from our originally generated Pokémon.

Dataset:
Our primary dataset, "1000 Pokémon Dataset" (Kaggle), contains over 26,000 images of 1000 different Pokémon. Each of the 1000 Pokémon has around 15–40 images containing different angles, crops, and color schemes. The 128x128 pixel images have transparent backgrounds, allowing our model to train more effectively on the Pokémon itself without a noisy background. If we find our model to be performing poorly, and we have tried several different methods to improve, we will add in more datasets. Such datasets we have found that we could append are the "Pokémon sprite archive" and the "Pokémon images dataset". All datasets are publicly released under permissive licenses (CC0 / noncommercial allowed). We plan to use an 80-20 train-validation split. For preprocessing, we will resize all images from 128x128 to 64x64 and convert RGBA to RGB for training efficiency. We will also normalize all pixel values to [-1,1] to match our generator's Tanh output activation. Depending on initial performance, we will also possibly choose to augment our data using methods such as horizontal flips, mild color jitter, and random crops to improve generalization. The links to the datasets are listed below:

- ⭐ **1000 Pokémon Dataset**
- **Pokémon sprite archive**
- **Pokémon images dataset**

Model Plan:
We will train a Deep Convolutional Generative Adversarial Network (DCGAN)-style architecture, fully implemented from scratch in PyTorch. The generator will take a

100-dimensional Gaussian noise vector as input and use 4-5 transposed convolutional blocks with BatchNorm and ReLU activations, outputting a 64×64×3 RGB image with Tanh activation. Our discriminator will take a 64×64×3 RGB image as input and use a CNN classifier with several convolutional blocks, LeakyReLU activations, strided convolutions instead of pooling, and spectral normalization for stability. The discriminator outputs a single real/fake probability score. Both networks will have approximately 2.5-3 million parameters each. We will use Binary Cross-Entropy loss with the standard adversarial objective and Adam optimizer. Training will use alternating generator and discriminator updates. We will incorporate regularization techniques including spectral normalization, label smoothing, and gradient clipping if needed. We plan to train for 200-300 epochs initially, adjusting based on convergence and compute budget. For Stage 1, we will implement a minimal baseline with a simple 2-block generator and 2-block discriminator, train on a smaller dataset (around 5k images), and demonstrate that our training loop, data loader, and loss functions work end-to-end with non-random outputs.

<u>Estimated Final Generator Network:</u>
Input: 100-dim Gaussian noise vector
1. Linear: 100 → 4×4×512 (reshape to 4×4×512 feature map)
2. TransposeConv2d: 4×4×512 → 8×8×256
   (kernel=4, stride=2, padding=1, BatchNorm2d, ReLU)
3. TransposeConv2d: 8×8×256 → 16×16×128
   (kernel=4, stride=2, padding=1, BatchNorm2d, ReLU)
4. TransposeConv2d: 16×16×128 → 32×32×64
   (kernel=4, stride=2, padding=1, BatchNorm2d, ReLU)
5. TransposeConv2d: 32×32×64 → 64×64×3
   (kernel=4, stride=2, padding=1, Tanh activation)

Output: 64×64×3 RGB image with values in [-1, 1]

<u>Estimated Final Discriminator Network:</u>
Input: 64×64×3 RGB image
1. Conv2d: 64×64×3 → 32×32×64
   (kernel=4, stride=2, padding=1, LeakyReLU 0.2)
2. Conv2d: 32×32×64 → 16×16×128
   (kernel=4, stride=2, padding=1, BatchNorm2d, LeakyReLU 0.2)
3. Conv2d: 16×16×128 → 8×8×256
   (kernel=4, stride=2, padding=1, BatchNorm2d, LeakyReLU 0.2)
4. Conv2d: 8×8×256 → 4×4×512
   (kernel=4, stride=2, padding=1, BatchNorm2d, LeakyReLU 0.2)
5. Flatten → Linear: 4×4×512 → 1 (Sigmoid activation)

Output: Probability score (real vs. fake)

Evaluation:
To evaluate our model, we will use both qualitative and quantitative measures. As our primary metric, we will use Fréchet Inception Distance (FID), which is a metric used to evaluate the quality of images generated by AI models, such as GANs, by comparing the statistical similarity between a set of real images and a set of generated images. A lower FID will indicate a closer similarity to real Pokémon artwork. FID is a standard metric for GAN research and is measurable across epochs. Our secondary metrics will include Inception Score (IS), which measures quality of images, and Diversity Score (with pairwise LPIPS, or Learned Perceptual Image Patch Similarity), a way to measure the variety of generated images by averaging the perceptual distance between many random pairs of images. Qualitatively, we will visually inspect training progression by saving samples periodically and comparing them to actual Pokémon sprites.

Milestones:
1. Finalize dataset collection & preprocessing
2. Build PyTorch data pipeline
3. Implement minimal baseline GAN
4. Implement full DCGAN-style model
5. Train first stable version on 64×64 resolution
6. Track FID/IS and visualize outputs
7. Improve training stability (learning rate schedules, label smoothing)
8. Final experiments, sample generation, and evaluation
9. Write report and document reproducible code
10. Final presentation preparation

Risks:
One risk that we have for this project is mode collapse, a problem particularly in GANs, where the model fails to produce a wide variety of outputs and instead generates only a few repetitive examples. This happens when the generator finds a few outputs that can easily fool the discriminator and over-optimizes on them, neglecting to capture the full diversity of the training data. To mitigate this, we will use spectral norm, lower LR, and mini-batch discrimination. Another risk we may have is GAN/training instability, resulting in non-convergence and/or vanishing gradients. To address this, we can use techniques such as label smoothing, gradient clipping, or use lower learning rates. If these fail to succeed, we may have to fall back to a WGAN-GP, where we add Wasserstein distance and a gradient penalty.

Ethics: Our model will generate fictional, stylized fantasy artwork rather than real creatures or human subjects. The dataset that we have selected is properly licensed and documented, and no personal data is being used. We are also not reproducing proprietary Pokémon characters, but instead creating new creatures inspired by style only. Pokémon sprites are copyrighted by

Nintendo/Game Freak, but our use is educational/non-commercial under fair use doctrine, with no redistribution of generated outputs beyond academic evaluation.

Compute Plan:
All training and experiments will be run on Google Colab, using the built-in NVIDIA T4 (16GB VRAM) or A100 GPUs (40GB VRAM) if available. Training will be executed using Colab Pro's background execution feature for continuous training sessions up to 24 hours and save checkpoints every 25 epochs in Google Drive to prevent data loss from disconnections. We also might monitor training remotely using Weights & Biases (wandb) for real-time loss curves and generated sample visualization. For early development and baseline GAN training (64×64 resolution), a single T4 GPU should be sufficient. For full experiments or higher resolution models, we will schedule longer training sessions on A100-enabled Colab Pro+. We will manage runtime limits by saving intermediate model weights regularly. This setup provides reliable GPU access while keeping the project fully reproducible.

Reproducibility Plan:
For reproducibility purposes, we will use a fixed seed and deterministic PyTorch operations where possible. We will also have a *requirements.txt* contained in our project structure with detailed instructions and data preprocessing scripts. Lastly, we will have checkpointing, where we save model weights every ~25 epochs. The best model will be selected by the lowest FID on the validation set.