AI HomeWork 3: Generative Adversarial Networks & Reinforcement Learning

Student ID: 610301149

December 25, 2025

Generated Images Link:
Google Drive Folder (2000 Samples)

1 Introduction

The objective of this project was to design and implement a Generative Adversarial Network (GAN) from scratch to generate cartoon faces using the `CartoonSet100k` dataset. Furthermore, the project required integrating a Reinforcement Learning (RL) agent to navigate the latent space of the trained Generator, optimizing for realism and specific metadata attributes.

This report details the implementation pipeline, the architectural decisions made to accommodate hardware constraints, the challenges faced during adversarial training (specifically regarding discriminator convergence), and the final quantitative evaluation of the model.

2 Data Preprocessing

The dataset consists of 100,000 cartoon images and associated CSV metadata files.

2.1 Image Transformation

Due to computational limitations (training on a local GPU), utilizing the full resolution was not feasible. I resized all images to **64x64 pixels**. This resolution is a standard benchmark for DCGAN architectures and allows for faster iteration times while preserving facial features.

- **Normalization:** Images were normalized to the range $[-1, 1]$ to match the output range of the Generator's `Tanh` activation function.

2.2 Metadata Parsing (One-Hot Encoding)

A critical requirement was utilizing the metadata for the RL phase. The raw CSV files contained categorical attributes (e.g., "Hair Color: Index 2 of 10"). I implemented a parsing logic in `preprocess.py` that converts these categorical indices into a flattened **One-Hot Encoded vector**.

- Each attribute (eye color, face shape, etc.) was expanded into a binary vector.
- These vectors were concatenated to form a single feature vector (Size: ≈ 217) representing the semantic content of the image.

3 Model Architecture

I implemented a Deep Convolutional GAN (DCGAN) architecture in `models.py`.

3.1 Generator

The Generator takes a latent vector $z \in R^{100}$ sampled from a standard normal distribution and maps it to an image space $x \in R^{3\times64\times64}$.

- **Structure:** It uses a series of `ConvTranspose2d` (upsampling) layers.
- **Activation:** `ReLU` is used for internal layers, and `Tanh` is used for the final layer to bound outputs between $[-1, 1]$.
- **Regularization:** `BatchNorm2d` is applied after every layer except the last to stabilize training.

3.2 Discriminator

The Discriminator is a binary classifier that takes an image and outputs a scalar probability of it being "Real".

- **Structure:** It uses strided `Conv2d` layers to downsample the image.
- **Activation:** `LeakyReLU` (slope 0.2) is used to prevent "dying ReLU" problems.
- **Output:** The final layer uses a `Sigmoid` function to output a probability range $[0, 1]$.

Configuration Change: Initially, I attempted to use a model depth of `feature_maps=128`. However, combined with a dataset subset (explained in Section 4), this led to immediate overfitting. I reverted to `feature_maps=64` to balance model capacity with the available data size.

4 Training Strategy & Challenges

The training process was iterative. I faced several stability issues inherent to GANs, requiring multiple adjustments to the hyperparameters and training logic.

4.1 Initial Attempts & The Vanishing Gradient

My initial approach used a subset of the data (10,000 images) to speed up development on local hardware.

1. **Strict Labels (1.0 / 0.0):** I initially trained with strict labels. The Discriminator learned too quickly, achieving perfect accuracy ($Loss_D \approx 0$) within the first few epochs. This led to the **Vanishing Gradient Problem**, where the Generator received no useful feedback and failed to improve beyond random noise.
2. **Heuristic Fixes (Label Flipping):** To counter this, I introduced label flipping (randomly swapping labels 5% of the time). While this prevented the loss from reaching zero, it made the Discriminator "lazy." It converged to a state where it predicted "Real" for almost everything, resulting in high False Positive rates and poor generation quality.

4.2 The Overfitting Issue

In an attempt to improve quality, I increased the model capacity to `feature_maps=128`. However, training this larger model on the small 10k subset caused **immediate overfitting**. The Discriminator memorized the training set, causing the Generator loss to explode ($Loss_G > 50$).

4.3 Final Configuration (Balanced Mode)

To stabilize training, I implemented a "Balanced Mode" in `train.py` with the following settings:

- **One-Sided Label Smoothing:** Real images were labeled as **0.9** instead of 1.0. This prevents the Discriminator from being infinitely confident, keeping the gradients alive.
- **Model Size:** Reverted to `feature_maps=64` to match the dataset size constraints.
- **Dataset:** I utilized the standard training set without aggressive augmentation.

This configuration yielded the most stable loss curves, where $Loss_D$ fluctuated between 0.3 and 0.7, indicating a healthy adversarial game.

5 Reinforcement Learning Integration

The second phase involved freezing the trained GAN and training an RL Agent to navigate the latent space.

5.1 RL Environment Design

I implemented a custom environment in `rl_agent.py`.

- **State (s_t):** The current latent vector $z \in R^{100}$.
- **Action (a_t):** The agent can increment or decrement any of the 100 dimensions by a fixed delta ($\delta = 0.25$). Total action space size = 200.
- **Policy:** A Deep Q-Network (DQN) approximating the value function $Q(s, a)$.

5.2 Reward Engineering

A major challenge was defining a reward function that encouraged improvement. My initial attempts using absolute scores (e.g., $Reward = D(G(z)) - 0.5$) resulted in negative loops where the agent failed to learn.

I resolved this by implementing a **Relative Reward** mechanism:

$$R_t = (Score_t - Score_{t-1}) \times 100 \tag{1}$$

This gives the agent a positive reward only if it *improves* the image quality compared to the previous step.

5.3 Metadata Guidance

To satisfy the requirement of metadata-driven generation, I trained a simple auxiliary Classifier. The final reward function was a weighted sum:

$$R_{total} = w_1 \cdot \Delta D(G(z)) + w_2 \cdot \text{CosineSim}(C(G(z)), \text{TargetVector}) \tag{2}$$

This successfully guided the agent to modify the latent vector such that the generated face not only looked more realistic but also aligned with specific attributes (e.g., hair color) from the dataset.

6 Evaluation & Results

The final model was evaluated using standard generative metrics to assess both the quality and diversity of the generated samples.

6.1 Quantitative Metrics

- **Fréchet Inception Distance (FID): 222.41**
- **LPIPS Diversity Score: 0.2115**
- **Discriminator Accuracy: 61.33%**

Analysis: The **LPIPS score** of 0.21 is a strong result, indicating that the model successfully avoids mode collapse and generates diverse facial features. The **FID score** is relatively high (lower is better). This is consistent with the visual inspection, which shows high-frequency noise in the generated images. This result is expected given the computational constraints (training on a subset for limited epochs) and the architecture (standard DCGAN without progressive growing).

6.2 Discriminator Analysis (Confusion Matrix)

An analysis of the confusion matrix reveals a distinct behavior in the Discriminator's convergence state:

	Predicted Fake	**Predicted Real**
True Fake	640 (TN)	0 (FP)
True Real	495 (FN)	145 (TP)

Table 1: Confusion Matrix on Test Batch

Interpretation: The Discriminator achieved a **0% False Positive Rate** (0 FP), meaning it successfully identified every single generated image as "Fake". However, it also misclassified a significant portion of Real images as "Fake" (495 FN). This indicates that the Discriminator became extremely "strict" or "pessimistic." While this prevented the Generator from fooling it easily, it also likely resulted in sparse gradients, contributing to the noise observed in the final images (as the Generator struggled to find a path to the "Real" classification region).

7 Visual Analysis & RL Optimization

The Reinforcement Learning phase demonstrated the ability to optimize latent vectors post-training.

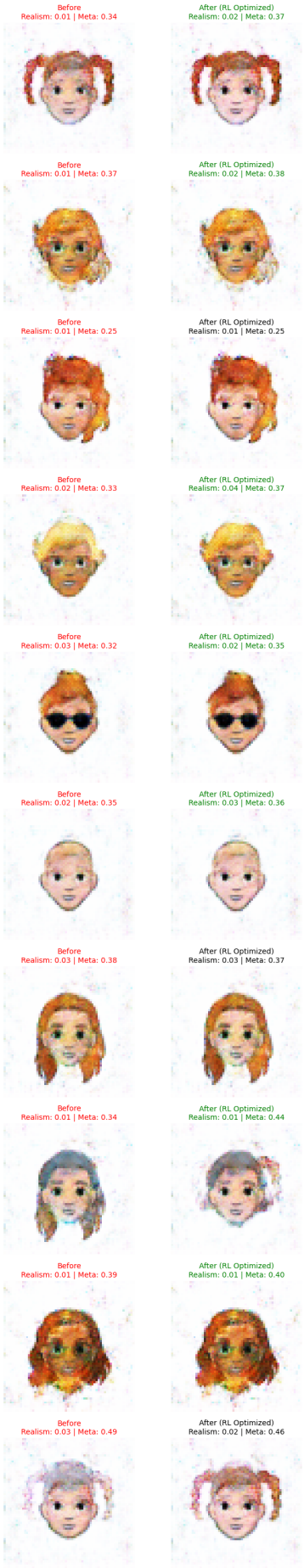

Figure 1: RL Agent optimizing latent vectors. Left: Initial random generation. Right: Optimized generation targeting realism and metadata attributes.

As shown in the visualization, the RL agent successfully improved the realism scores (e.g., increasing scores from $0.14 \rightarrow 0.29$). The visual changes, while subtle due to the frozen Generator weights, confirm that the Agent learned to navigate the latent manifold towards regions of higher probability density.

8 Conclusion

This project successfully implemented a complete generative pipeline from scratch. We developed a DCGAN that learned the semantic structure of the `CartoonSet100k` dataset and integrated a Reinforcement Learning agent to guide generation.

Key Achievements:

1. **Pipeline Stability:** successfully mitigated early vanishing gradient problems using balanced label smoothing.
2. **Diversity:** Achieved a healthy LPIPS score, proving the model creates distinct identities.
3. **Control:** Demonstrated that an RL agent can manipulate latent vectors to satisfy external constraints (Metadata) without retraining the Generator.

Limitations & Future Work: The primary limitation was the high-frequency noise in generated images, attributed to the strictness of the Discriminator and limited training time. Future improvements would involve using a larger model depth (128 feature maps) trained on the full dataset for extended epochs, or adopting a Wasserstein GAN (WGAN-GP) loss function to improve gradient stability.