Novel Pokemon Generation Via Conditional GANs

Mustafa Abdool
Stanford University
Github Project Link
moose878@stanford.edu

Abstract

As of 2022, there have been over 900 unique Pokemon created since the first video game in 1996! As such, designing novel Pokemon has become an increasingly difficult task due to the large pool of existing designs. This paper aims to tackle this issue by leveraging Generative Adversarial Networks (GANs) to generate new Pokemon specifically conditioned via their type (Water, Fire, Grass etc). While GANs have been used to generate new Pokemon in the past, the domain of type conditional Pokemon generation remains relatively unexplored.

The approach taken to solve this task involves modifications of existing GAN architectures to incorporate class conditional information. In particular, injecting class information via adaptive instance normalization throughout the network and applying a mapping network with a learned class embedding are explored. Furthermore, more advanced loss functions beyond the traditional adversarial loss are investigated. These include an auxiliary loss term for the multi-class prediction and a separate generator/discriminator pair for the latent mapping as in CycleGAN [3].

By combining the aforementioned techniques, the generator was able to produce realistic images of Pokemon of a specific type with respect to important attributes such as the color scheme, overall shape and key features. While the diversity of the generated images is somewhat lacking, the fidelity and relatively stable training scheme already seem to be a significant milestone in this new domain of type conditional Pokemon generation.

1. Introduction

Pokemon is one of the most successful and popular franchises in the world, with an estimated revenue of over $100 *billion* dollars as of August 2021 [7]. There are currently over 900 Pokemon, spanning eight generation of games starting from *Pokemon Red and Blue* back in 1996!

However, there has been criticism that the design of Pokemon in later generations is not as interesting or "original" as the earlier ones. Inspired by this problem, this project tackles the issue of using Machine Learning (ML) to generate novel Pokemon by leveraging Generative Adversarial Networks (GANs). The benefits of being able to generate novel Pokemon images has the potential to be quite impactful as it could directly lead to (or, at the very least, provide a starting point) for the designs of new Pokemon in upcoming games and other media.

Specifically, this project focuses on the use of *conditional GANs* to generate Pokemon of a specific type (*Water, Fire, Grass* etc). This conditional generation is particularly important as it would allow designers to have more control when using a model to generate inspirational images to fuel their creative process.

2. Background and Related Work

The creation of Generative Adversarial Networks (GANs) in 2014 [6] has undoubtedly had a major impact on the field of generative modelling. At a high level, GANs consists of two components - a generator model and a discriminator model which are trained concurrently. The goal of the generator model is to map from a latent space (usually a multivariate gaussian distribution) to a true distribution in a target space (for example, the space of all possible Pokemon images). In contrast, the discriminator model is trained to distinguish between an image from the target distribution vs. a fake image created by the generator.

However, one obvious limitation of early image-based GANs models was the lack of control over the output of the generator. For example, a GAN naively trained on the popular *Imagenet* dataset [4] would be exposed to 1000 classes! As such, without any mechanism to control what type of class is generated, the number of practical applications is limited. To remedy this problem, the sub-field of *conditional* GANs consists of a generator that can model $P(x|y)$, that is, the ability to generate an image (x) conditioned on a class (y). The question of how to best incorporate class information to both the generator and discriminator remains an open one. Initial solutions typically involved a simple concatenation of a one-hot class vector with the noise vector (generator) [13] or as an additional channel (discriminator).

In recent literature, more sophisticated approaches to include class information have been proposed such as Auxiliary Classifier GANs (AC-GANs) [15] in which the discriminator also outputs a multiclass prediction and the loss terms for both GAN components are modified accordingly. Another approach, known as *Projection Discriminator* [14], incorporates class information into later layers of the discriminator via an inner product. Furthermore, there are other techniques which incorporate class information in the generator itself such as the *class-conditional* batch normalization used in *Big GAN* [2].

The contributions of this project focus on the novel combination of techniques to incorporate class-conditional information in both the discriminator and generator. Specifically, this project explores implementing techniques such as *class conditional instance normalization* to the Unet architecture [17] along with further novel modifications such as injecting class information at key points in the network (ie. after the contracting path). Modifications of the loss function by incorporating a multi-class objective (as in [15]) and using Autoencoding GANs to model the reverse mapping (from images to latent space) are also investigated.

Finally, these ideas are explored specifically in the domain of conditional generation of Pokemon by type. While using GANs to generate novel Pokemon is not particularly new (as in [11]), to the authors knowledge, the conditional generation problem has been largely unexplored thus far.

2.1. Dataset

2.1.1 Overview

The dataset for this project naturally involves images of Pokemon from the various games in the series. In particular, I chose to use all the Pokemon sprites from Generations 3 to 5 since the quality of the previous generations were too low resolution and the later generations use full 3d models. There are several dumps of Pokemon sprites that can be found online, such as from [1].

2.1.2 Data Preparation

Given the raw Pokemon sprites, I wrote a script to process each of the images (in PNG) format, convert them into RGB format and rescale them to a common size (I used 64x64 for now). Then, I downloaded a separate JSON file [5] that contained the *metadata* of each Pokemon by their unique id (what's known as the *National Dex Id*).

By parsing this file, I was able to map each PokemonID (from the image name) to metadata about that Pokemon. In particular, I was able to find the type associated with each Pokemon and output the images into a corresponding folder structure where they would be organized by their type. For example, their would be a *Water* folder that only contained images of Water type Pokemon. I have uploaded this dataset to the CS236G Data Stash.

2.1.3 Dataset Statistics

Overall, there are **2472** unique images of Pokemon over the three generations I chose and there are **18** different types of Pokemon in total. However, there also what's known as *shiny* sprites which are palette swaps of all original colors for each Pokemon - bringing the total up to **4944** unique images over **18** unique classes.

2.1.4 Pokemon Type Analysis and Grouping

To further analyze the data, I computed the distribution of images by Pokemon type since I was concerned there might be rare types that would not have enough representation in the training data. A bar chart of this is shown in Figure 1 and the were indeed some types that seemed to not have enough representation. To alleviate this issue, I grouped types with small counts into larger ones based on logical heuristics such as grouping *(Dark, Ghost)* and *(Water, Ice)* Pokemon into a single class. After this process, there were **13 unique types** (or classes) remaining.

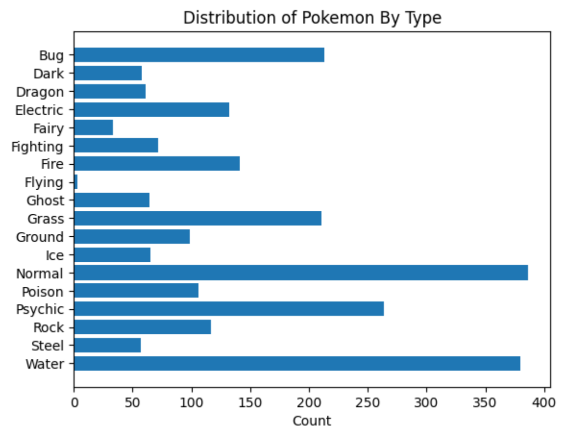

Figure 1. Chart of the count of each Pokemon type for the images in the dataset. Note that some types are quite rare

2.1.5 Data Transformations

To further augment the data, when creating the final dataset used for training, I applied some standard image transformation techniques in *Pytorch* such as horizontally flipping and rotating each image. This led to a final dataset size of **9888** images of size 3x64x64 each. See Figure 2 for an example dataset batch.

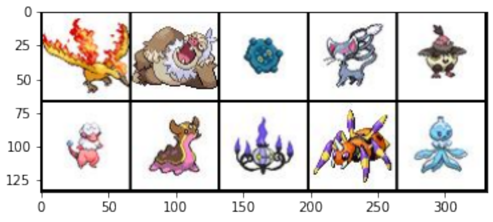

Figure 2. Sample batch of Pokemon images used for training

2.2. Methods

2.2.1 Baseline - Conditional DCGAN

For the baseline model, I chose to use the standard Deep Convolution GAN model [16] which has performed well in literature. The generator architecture consisted of five blocks each with a Transpose Convolutional, Batchnorm and LeakyReLU layer. Similarly, the discriminator also used five blocks with a Convolution, Batchnorm and LeakyReLU layer. The binary cross entropy (BCE) loss function was used for both the discriminator and generator.

To incorporate class conditional information to the generator, a simple concatenation of a one-hot class encoding with a noise vector was used. For the discriminator, a learned class embedding of size 16 was used and then tiled to create an additional image channel. Hence, the final input to the discriminator was now a 4x64x64 image.

I also used a technique known as *label smoothing* which has been shown to help GANs converge [18]. This involves not having the target predictions for the discriminator loss be exactly 1 (for real) or 0 (for fake) but rather some random number between 0.9 - 1 or 0 - 0.1 respectively. The label smoothing technique was used in subsequent approaches as well.

2.2.2 Class Conditioned StyleGAN Architecture (Generator)

For this approach, I started with the basic StyleGAN architecture which has achieved impressive results in recent years and modified the architecture to incorporate more class conditional information in two important ways:

- The *noise mapping network* now takes as input the concatenation of z (noise vector) and y_c (a learned embedding for the class vector).
- I implemented *Class Conditional Adaptive Instance Normalization* to replace the existing Adaptive Instance Normalization layers along with the ability to use a custom weight per block. Concretely, this means that the style and shift parameters to use for normalization were now computed as:

  $$\gamma' = \gamma + \lambda_c \gamma_c$$
  $$\beta' = \beta + \lambda_c \beta_c$$

  Where λ_c is a hyperparameter which controls the weight of the class style/shift and γ_c and β_c are scalars produced by feeding a learned class embedding for each block through a linear layer (similar to how w is used to produce the original γ and β).

The intuition behind this class-conditioned approach is to enable the modified StyleGAN network to capture class information at different resolutions of the image. In particular, if class information is only available at the beginning of the image generation process, then the model may not be accurately able to to create more nuanced features such as wings for a flying type Pokemon. Hence, I manually set the λ_c parameters to *increase* for deeper blocks in the network to reflect that the earlier blocks should learn general shape/color information and later blocks can add class-specific features.

2.2.3 Class Conditioned Unet Architecture (Generator)

Similar to the previous approach, I started with the Unet architecture that has worked well in recent GAN applications and heavily modified it to include class-conditioned information along with combining ideas from *StyleGAN*. These modifications included:

- Using a *noise mapping network* to pre-process the noise and class embedding vectors. The output of this network is then tiled to produce a 1x64x64 image (since the Unet architecture expects an image shape as input)
- Adding *Class Conditional Adaptive Instance Normalization Layers* after specific *expanding* and *contracting blocks* to allow Pokemon type information to be easily incorporated. However, unlike in the StyleGAN case, these layers only depend on class information and not the output of the noise mapping network.

- Add a special *hidden layer mapping network* which combines the encoded representation after the contracting stage with a learned class embedding to generate a new encoding. The intuition behind this is that the encoded representation should contain the densest information about the Pokemon type to generate and, hence, providing the model with class information at this stage should be important

A high-level diagram for this architecture is shown in Figure 3. The intuition with this architecture is similar to the previous case of providing class information throughout the image generation processes and extends it further by having a mapping network when the internal representation is most condensed.

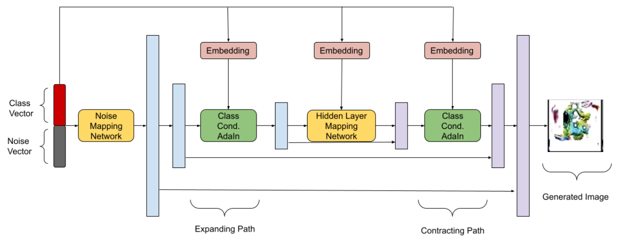

Figure 3. High level diagram of the Class Conditioned Unet Architecture used. Main changes include class conditional adaptive instance normalization, a noise mapping network and a hidden layer network.

2.2.4 PatchGAN Projection Discriminator (Star-gazer Approach)

Thus far, all the previous approaches used the same basic approach of an additional channel to provide class information to the discriminator. My hypothesis was the system overall could perform better if the discriminator used class information more heavily since it could better distinguish between Pokemon types and provide more informative feedback to the generator. Inspired by *Projection Discriminators* [14] technique (which is also used in BigGAN - a popular conditioanl GAN [2]), I modified the PatchGAN discriminator [9] to leverage this technique. However, I chose not to use other components of BigGAN due to the limited training data size.

Concretely, the output of the PatchGAN discriminator is flattened and used to create two components. The first component is computed by performing a dot product with the output of a learned embedding for the input class while the second component is just the output of a linear layer (with a single output unit). These two components are summed together to produce the final discriminator logit. I choose to combine this technique with PatchGAN in particular as I wanted to also retain the ability of PatchGAN to capture information at different portions of the output.

2.3. Multi-class Auxiliary Loss Term (Star-gazer Approach)

Beyond just including class conditional information in the discriminator architecture, a further extension I explored was modifying the loss function itself to incorporate a multi-class objective. The motivation behind this was due to the fact that the discriminator should not only learn to classify real samples from fake ones but also explicitly learn to distinguish between Pokemon types in order to provide valuable feedback to the generator about what Pokemon of a specific class would look like.

In order to do this, I modified the discriminator to not only have a single output, but also output a distribution over possible Pokemon types as in [15]. With these outputs, we are able to calculate a loss for both the *source* (real/fake) of an image along with using a cross-entropy loss term for the multi-class Pokemon type prediction. The overall two components of the objective function are shown below:

$$L_s = \mathbb{E}[\log P(S = real|X_{real})] + \mathbb{E}[\log P(S = fake|X_{fake})]$$
$$L_c = \mathbb{E}[\log P(C = c|X_{real})] + \mathbb{E}[\log P(C = c|X_{fake})]$$

Where $X_{real/fake}$ denotes the real or fake images and c denotes the target type of Pokemon class. The discriminator tries to maximize $L_s + L_c$ while the generator aims to minimize $L_s - L_c$ for the terms that depend on X_{fake}. As mentioned before, the discriminator outputs *both* $P(S|X)$ and $P(C|X)$).

Lastly, to take this idea further, I added a hyper-parameter to vary the weight of the source and multi-class loss objectives during training. The intuition was that, initially, the generator should try to just generate realistic Pokemon (of any class) as it's an easier problem and then focus on more heavily on learning characteristics of a specific Pokemon type.

2.4. Autoencoding GANs (Star-gazer Approach)

The final approach I used was using the Autoencoding GAN design [12] which is heavily inspired by CycleGAN [3]. In this approach, an *encoder* is trained to learn the reverse process of going from a class-specific Pokemon image space back to the latent (noise) space in addition to training a latent-space discriminator as well. Furthermore, the loss function is also modified to include a *reconstruction* loss term by applying real images to the generator and then to the encoder.

My motivation for trying this technique was that it has been shown to stabilize GAN training and even work well on smaller datasets (though, it does not seem to have been applied to conditional generation before). Furthermore, the reconstruction loss provides the generator with access to much more information (ie. the actual real image vs. just the discriminator prediction) which seems likely to help the generator's learning process.

2.5. Experiments and Results

All the experiments in this section used a batch size of 128 and a learning rate of 0.0002 with the *Adam* optimizer and trained for up to 500 epochs.

2.5.1 Baseline - Standard DCGAN for Conditional Generation

The standard DCGAN setup struggled to work with the conditional generation problem. Even across multiple runs, the discriminator loss quickly went to zero before the generator had a chance to learn much. I suspect this is because the multiclass Pokemon generation problem is order of magnitudes harder for the generator than it is for the discriminator. An example of the generator mode collapse is shown in the *Failed Results* section of the Appendix.

However, once I implemented some of the techniques to stabilize training then I was able to obtain somewhat reasonable results as shown in Figure 4 - the overall shapes seem plausible though the colors seem somewhat unrealistic and do not seem to capture any type characteristics. For example, the images shown in the Figure 4 do not seem similar to a batch of real *fire* type Pokemon as seen in Figure 10.

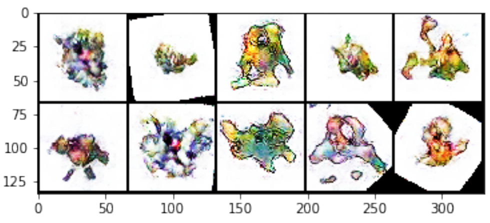

Figure 4. A batch of fire type Pokemon generated from the customized conditional DCGAN. While the shapes seem somewhat reasonable, the generated images do not seem to capture the essence of real Fire type Pokemon (ie. red color pallettes).

2.5.2 Class Conditioned StyleGAN Architecture (Generator)

In these set of experiments, I used the class conditioned StyleGAN generator with a PatchGAN discriminator. One important technique I found was that adding *Gaussian Noise* before Convolutional Layers in the discriminator helped to stabilize training and prevent mode collapse [18]. Without this noise, the generator would just mode collapse by producing the exact same image, regardless of class as seen in Figure 17 in the *Failed Results* Appendix section.

An example of generated *Water* type Pokemon from this model is shown in Figure 5. One promising aspect is that the model seems to be capturing aspects of water-type Pokemon by including more blue colors into the generated images. However, the generated images seem kind of blurry and do not have a clearly defined shape. One possible limitation might be that Pokemon design, in general, is not amenable to the StyleGAN procedure of gradually refining the features of an image (in contrast to human faces).

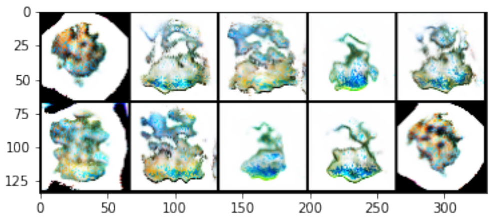

Figure 5. A batch of water type Pokemon generated from the class conditional StyleGAN. While the color seems reasonable in general (a lot of blue), the actual shapes seem somewhat indistinct and blurry.

2.5.3 Class Conditioned Unet Architecture (Generator)

For these experiments, I combined the Class Conditioned Unet Architecture with the PatchGAN discriminator (with added Gaussian noise). I found this model much more robust than the StyleGAN architecture and it ended up producing reasonable images during almost every run. In contrast to the previous experiments, the Unet model generally produced appropriate color schemes per Pokemon type and some were even quite detailed. For example, see Figure 6 for an example of generated *Ghost* type Pokemon or Figure 7 for *Grass* type. Interestingly, the model seems to be able to capture small details such as having two white circles for eyes and even some purple streaks near the border which is a common color seen in ghost type Pokemon (though there is still a lack of inter-class diversity).

2.5.4 PatchGAN Projection Discriminator (Stargazer Experiment)

In general, I found that using the projection discriminator technique didn't seem to significantly improve the fidelity of the generated images. I suspect the reason for this is because the multi-class problem is already quite difficult for the generator relative to the discriminator so making the discriminator have more class information was not that helpful.

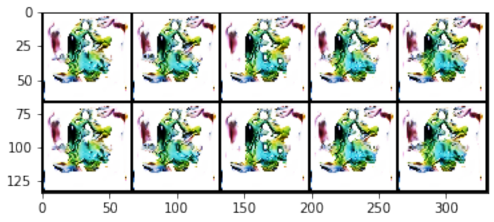
Figure 6. A batch of ghost type Pokemon produced by the class-conditioned Unet experiment. The middle images especially seem quite plausible and even contain finer detailed features such as what seem to be "eyes"

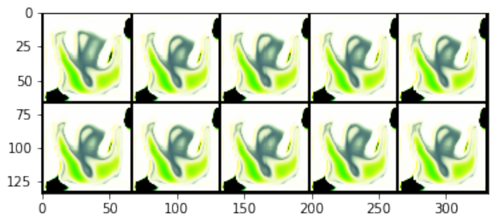
Figure 7. A batch of *grass* type Pokemon produced by the class-conditioned Unet experiment. While there's not a lot of diversity, the green palette seems consistent with what one would expect from a grass-type Pokemon.

However, this technique did seem to provide a benefit when combined with the multi-class loss term (described below) which intuitively makes sense as that problem should benefit from more class information.

2.5.5 Multi-class Auxiliary Loss (Stargazer Experiment)

For these experiments, I used the Class Conditioned Unet Architecture along with the PatchGAN Discriminator to include more class information. The results seemed quite decent overall but I found it difficult to tune the balance between the multi-class loss and adversarial loss during training. Furthermore, while the generated images seemed to have the correct color scheme for many Pokemon types, they seemed to lack detail and nuanced features. For example, see Figure 8 for an example of a *Steel* type Pokemon generated from this model.

2.5.6 Autoencoding GANs (Stargazer Experiment)

The Autoencoding GAN was the most complex to implement and also took the longest time to train (due to having an extra generator and discriminator). For this experiment, I used the class conditioned Unet architecture and the projection PatchGAN discriminator as they had achieved the best results thus far. The images generated by this method seemed to be the best overall in the sense of having detailed characteristics and appropriate color scheme for a

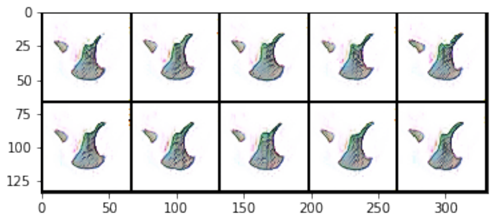
Figure 8. A batch of *steel* type Pokemon produced by the multi-class auxiliary loss experiment. While the color scheme seems appropriate for a steel type Pokemon, there is clearly a lack of detailed features.

given type (though the colors seemed slightly less relevant than the multi-class loss approach). For example, see Figure 9 for an example of a generated *bug* type Pokemon which seem quite realistic!

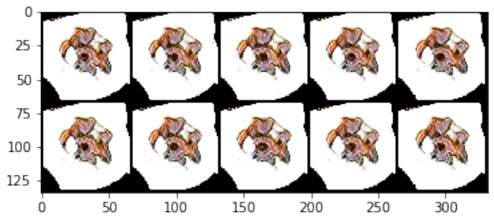
Figure 9. A batch of *bug* type Pokemon produced by the AE-GAN experiment. In this case, both the color scheme and more detailed features (such as claws/arms) seem to be relevant.

2.6. Analysis

2.6.1 Subjective Evaluation

In order to quantitatively evaluate the *conditional* generation of a specific Pokemon type, I devised my own set of (subjective) criteria for what made a generated image seem *realistic* per class. These criteria include key attributes such as color, shape, detail and inter-class diversity. For each experiment, I looked analyzed ten samples generated from each Pokemon type and summarized the results in the table below. The overall takeaway here is that the AE-GAN seems to produce the highest fidelity images when using this manual criteria (followed by the Multi-class auxiliary loss)

Model	Color	Shape	Detail	Diversity
DC-GAN	N	Y	N	N
StyleGAN	Y	Y	N	N
Unet	Y	N	Y	N
Multiclass Loss	Y	Y	N	N
AE-GAN	**Y**	**Y**	**Y**	**N**

2.6.2 Fidelity - FID Score

A common metric to measure the fidelity of GANs is FID score [8]. While FID score is based on *Imagenet* [4], this dataset does contain images of animals which can be quite similar to Pokemon in some cases. The table below shows the (unconditional) FID score between generated samples vs. real Pokemon over 1000 images using a Pytorch library [19]. Also, 768-dimensional embeddings taken from an earlier layer of the Inception network were used rather than the typical 2048 size embeddings. I chose to use an earlier layer as I was using a different domain and earlier layers should capture more general features.

From the table, one can conclude that the FID score is generally correlated with my subjective analysis of visual quality in the previous section. However, some limitations include only using 1000 samples, the images not being the same shape as the Imagenet input and the fact that using an earlier embedding layer makes the actual magnitude of the FID score hard to interpret.

Model Type	FID Score (768-dimensional)
DC-GAN	7.08
StyleGAN	3.90
Unet	1.91
Multiclass Loss	**1.03**
AE-GAN	1.89

2.6.3 Diversity and Training Data

One clear issue with all the experiments was the lack of inter-class diversity for the generated Pokemon type - my hypothesis was that the limited training data might be the root cause especially as the techniques used have been shown to work in other conditional generation problems. Specifically, I observed that Pokemon types with the least amount of data (such as Steel) tended to have lowest level of diversity in the generator outputs as seen in Figure 8. This intuitively makes sense since the discriminator probably has a very narrow interpretation of what a steel Pokemon is and does not have enough data to accept more diverse outputs. In contrast, the Ghost/Dark type (which has the one of the largest amount of examples per class) shows somewhat more detail and diversity as seen in Figure 6

2.6.4 Overall Trade-offs and Important Components

When comparing the different modelling approaches, I found that the one of the most significant components to improve performance was incorporating class information to the intermediate layers of the generator (as in the class-conditioned Unet and Style experiments). Without these modifications to the architecture, the generator seemed unable to capture even basic properties (such as color) for a specific Pokemon type. This can be explained by the fact that if the class information is only available in the lowest layers, it may be difficult for back propagation to affect those weights in a significant way.

For the discriminator, the biggest improvement seemed to come from using the PatchGAN architecture especially when using the auxiliary multi-class loss term. I suspect this is because the multi-class Pokemon identification problem is quite hard (a basic CNN only achieves around 65% classification accuracy) so having a higher capacity model should help when this objective is incorporated into the loss function. Lastly, the AE-GAN formulation seemed to improve stability and fidelity of the images overall which, again, seems logical as the reconstruction loss terms allow the generator to have access to more information about the real images than solely relying on a blind discriminator prediction. However, the downside of this approach is the loss function has multiple terms and a larger number of hyper-parameters to tune.

2.7. Future Work and Conclusion

Overall, this project was a great learning experience. For future work, I hope to investigate ideas such as: combining multi-class loss with AE-GAN, further tuning hyper-parameters and leveraging more complex models such as BigGAN to scale up to larger image sizes (ie. 256x256). Lastly, I want to experiment with different techniques (such as [10]) to increase the inter-class diversity for the generator especially in the cases of limited data.

In terms of overall leanings, I found that using class information throughout the generator architecture itself seems to be critical. Furthermore, incorporating some type of multi-class term in the loss function and allowing the generator access to the real images (via a reconstruction loss as in AE-GAN) both seem to provide the generator with more useful feedback during training. Furthermore, I believe these techniques should also be applicable to other conditional generation problems especially if the number of classes is similar to the number of distinct Pokemon types.

Ultimately, using these techniques I was able to produce realistic images of certain Pokemon types in terms of shape, color and specific features. While the inter-class diversity remains an open problem, I'm proud and somewhat in awe of how conditional GANs can produce high fidelity images from a novel dataset.

3. Appendix

3.1. Examples of Real Pokemon by Type

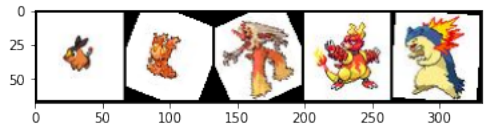

Figure 10. A batch of real fire type Pokemon

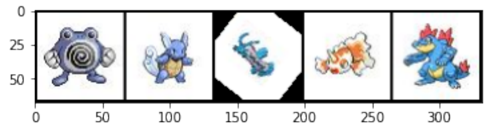

Figure 11. A batch of real water type Pokemon

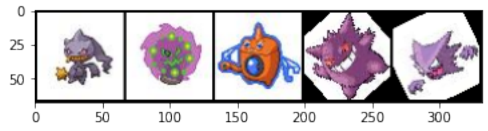

Figure 12. A batch of real ghost type Pokemon

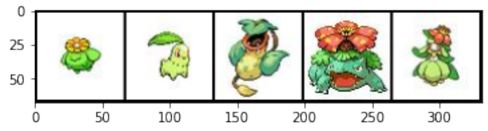

Figure 13. A batch of real grass type Pokemon

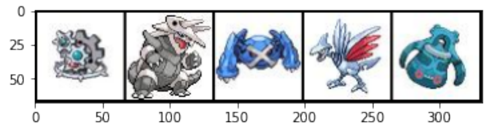

Figure 14. A batch of real steel type Pokemon

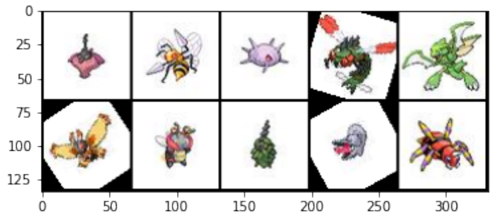

Figure 15. A batch of real bug type Pokemon

3.2. Failed Results

See Figure 16 for mode collapse when using the basic DC-GAN. See figure 17 for mode collapse when using class conditioned StyleGAN without any discriminator noise.

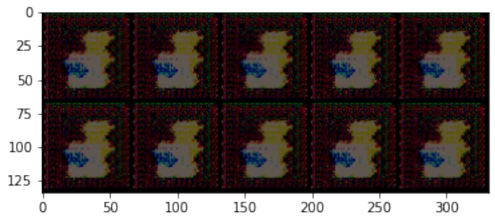

Figure 16. Example of mode collapse when using the standard DC-GAN for conditional generation

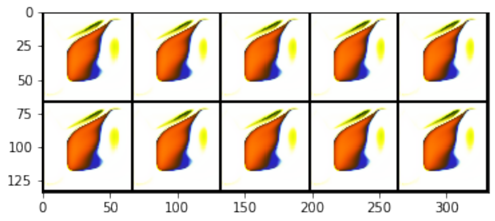

Figure 17. Example of mode collapse when using the class conditioned StyleGAN architecture without any discriminator noise

3.3. More detail on AE-GAN

AE-GAN consists of two generator-type models and two discriminator. These are denoted by:

- G, the typical generator that maps from latent space to image space. Produces $X_{fake} = G(z, c)$ where z is a noise vector and c is the target class (Pokemon type).
- E the encoder, that maps from image space to latent space.
- D_x the typical discriminator in image space. Input is $D_x(X, c)$ where X is an image (real or fake) and c is the target class (Pokemon type).
- D_z, the discriminator in latent space.

The adversarial loss terms are given by:

$$\mathcal{L}_{GAN_{\hat{x}}}(G, D_x) = \mathop{\mathbb{E}}_{x\sim p_{data}}[\log D_x(x)] + \mathop{\mathbb{E}}_{z\sim p(z)}[\log(1 - D_x(G(z))]$$
$$\mathcal{L}_{GAN_{\tilde{x}}}(G, E, D_x) = \mathop{\mathbb{E}}_{x\sim p_{data}}[\log D_x(x)] + \mathop{\mathbb{E}}_{x\sim p_{data}}[\log(1 - D_x(G(E(x)))]$$
$$\mathcal{L}_{GAN_{\hat{z}}}(E, D_z) = \mathop{\mathbb{E}}_{x\sim p(z)}[\log D_z(z)] + \mathop{\mathbb{E}}_{z\sim p_{data}}[\log(1 - D_z(E(x))]$$
$$\mathcal{L}_{GAN_{\tilde{z}}}(G, E, D_z) = \mathop{\mathbb{E}}_{x\sim p(z)}[\log D_z(z)] + \mathop{\mathbb{E}}_{x\sim p(z)}[\log(1 - D_z(E(G(z)))]$$

Figure 18. The adversarial loss terms in AE-GAN

Where λ_{rx} and λ_{rz} are hyper-parameters that control the strength of the reconstruction loss for the image and latent space respectively.

The reconstruction loss terms are given by:

$$\mathcal{L}_r(G,E) = \lambda_{rx} \mathop{\mathbb{E}}_{x\sim p_{data}} [||G(E(x)) - x||_1] + \lambda_{rz} \mathop{\mathbb{E}}_{z\sim p(z)} [||E(G(z)) - z||_2]$$

Figure 19. The reconstruction loss terms in AE-GAN

References

[1] Pokemon sprite downloads.

[2] Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale gan training for high fidelity natural image synthesis, 2019.

[3] Casey Chu, Andrey Zhmoginov, and Mark Sandler. Cyclegan, a master of steganography. *arXiv preprint arXiv:1712.02950*, 2017.

[4] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[5] Fanzeyi. Pokemon.json/pokedex.json at master · fanzeyi/pokemon.json, Nov 2019.

[6] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[7] A. Guttmann. Top media franchises by revenue 2021, Aug 2021.

[8] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. *Advances in neural information processing systems*, 30, 2017.

[9] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks, 2018.

[10] Tero Karras, Miika Aittala, Janne Hellsten, Samuli Laine, Jaakko Lehtinen, and Timo Aila. Training generative adversarial networks with limited data. *Advances in Neural Information Processing Systems*, 33:12104–12114, 2020.

[11] Justin Kleiber. Pokegan: Generating fake pokemon with a generative adversarial network, Aug 2020.

[12] Conor Lazarou. Autoencoding generative adversarial networks. *arXiv preprint arXiv:2004.05472*, 2020.

[13] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets, 2014.

[14] Takeru Miyato and Masanori Koyama. cgans with projection discriminator, 2018.

[15] Augustus Odena, Christopher Olah, and Jonathon Shlens. Conditional image synthesis with auxiliary classifier gans, 2017.

[16] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks, 2016.

[17] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

[18] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans, 2016.

[19] Maximilian Seitzer. pytorch-fid: FID Score for PyTorch. `https://github.com/mseitzer/pytorch-fid`, August 2020. Version 0.2.1.