# COMP3419 Assignment - PokéGAN

460299200

## Introduction

In this paper, we investigate Wasserstein Generative Adversarial Networks in relation to pokémon image generation, testing the effects of data augmentation on the performance of WGANs, as well as testing the performance of various discriminators inspired by recent developments in machine learning. We find that... INSERT RESULTS HERE!

### What are GANs?

Generative Adversarial Networks (GANs) are a special network made up of two neural nets, introduced in (Goodfellow et al. 2014). By training on a dataset of images, the network learns to mimic the dataset and generate similar images. The network consists of two neural networks working in concert: a generator, which learns to generate fake versions of the images fed into the network, and a discriminator, which learns to tell apart the real images and the generated images. In training both networks, with the discriminator's output being used to train the generator, the generator slowly learns to 'fool' the discriminator, while the discriminator still learns to distinguish the fake images of the improving generator. This results in the discriminator and generator playing a two-player minimax game, with the discriminator learning more and more features of the input dataset, and the generator learning to replicate these features. It has been shown[1] that this results in the loss function of a GAN being equivalent to the Jensen-Shannon divergence:

$$JS(\mathbb{P}_r, \mathbb{P}_g) = KL(\mathbb{P}_r || \mathbb{P}_m) + KL(\mathbb{P}_g || \mathbb{P}_m) - log(4) \tag{1}$$

where $\mathbb{P}_r$ is the real probability distribution (that is, the probability distribution of the input set of images) and $\mathbb{P}_g$ is the generated image's probability distribution[2]. $KL$ is the Kullback-Leibler divergence, and $\mathbb{P}_m$ is the mixture $(\mathbb{P}_r + \mathbb{P}_g)/2$. As the Jensen-Shannon divergence is always non-negative and only zero when the

[1] see (Goodfellow et al. 2014) for more details.

[2] This is a highly simplified explanation- please see (Arjovsky, Chintala, and Bottou 2017) for a more technical treatment.

two distributions are equal, minimising it is equivalent to the generator learning the real distribution over the image space - that is, learning how to produce images exactly similar to the real set of images. More intuitively, this distance is the difference between the probability distribution of images produced by the generator, and the probability distribution of the images within the input dataset. Therefore, by minimising this difference the network gets closer to representing the true distribution of images in the input dataset, thereby producing images closer to the input dataset.

This has achieved fantastic results - however, GANs can be famously unstable. The larger issues are mode collapse, where a generator learns to produce samples with low variety, due to it not fully learning a complex multimodal distribution[3], and the vanishing gradient problem, where the discriminator learns too well and the generator can not learn from it. Therfore, traditional GANs require that the generator and discriminator be trained slowly and at similar rates - lest the discriminator become too powerful and the generator fails to learn from it. This is obviously undesirable, and (Arjovsky, Chintala, and Bottou 2017) introduces a solution to this problem: WGANs.

### WGANs

In this report, we examine this particular recent innovation, as proposed in (Arjovsky, Chintala, and Bottou 2017): the Wasserstein Generative Adversarial Network (WGAN), which improves the performance and stability of GANs through its use of a novel loss function - the wasserstein distance. The use of this new loss function allows the 'discriminator' of a GAN to become a 'critic', giving a score to the generated images - the lower the score, the more obviously fake the image, and the better the score, the less fake the image seems. This provides a way for the generator to improve even if it cannot provide images that can fool the discriminator, eliminating the need to keep the discriminator weak enough that the generator can fool it, and eliminating mode collapse.

### The Wasserstein Distance

The core of the WGAN is its application of a new loss metric: the wasserstein distance. This distance, also known as the "earth mover's distance", can be expressed as:

$$W(\mathbb{P}_r, \mathbb{P}_g) = \inf_{\gamma \in \Pi(\mathbb{P}_r, \mathbb{P}_g)} \mathbb{E}_{(x,y)\sim\gamma}[||x-y||] \tag{2}$$

here $\Pi(\mathbb{P}_r, \mathbb{P}_g)$ denotes the set of all joint distributions $\gamma(x,y)$ whose marginals are respectively $\mathbb{P}_r$ (the real distribution of input images) and $\mathbb{P}_g$ (the generator's

[3] see this link for a more in-depth explanation

distribution). $\mathbb{E}$ simply means we take the average of the results from a infinite number of samples of (x,y) from the given joint distribution $\gamma$.

Intuitively, if we imagine both probability distributions $\mathbb{P}_r$ and $\mathbb{P}_g$ as histograms, this distance is the amount of 'earth' (or 'mass') required to move to make $\mathbb{P}_r$ resemble $\mathbb{P}_g$, with an optimal movement plan. And this serves as the cost function for the WGAN, providing a meaningful representation of the divergence even in difficult scenarios, thus providing better performance than the Jensen-Shannon distance[4]. By assuming K-lipschitz continuity for some K, this cost function can be transformed into a more usable form, providing a suitable cost function for the network[5].

In order to use this new cost function, we transform the 'discriminator' in a GAN to an estimator of the Wasserstein distance. This has the effect of turning the 'discriminator' into a critic - when the generator is worse, its distribution will be further from the real distribution, and so the wasserstein distance will be large, while if the generator is closer to the real distribution, the distance will be smaller. This prevents the vanishing gradient and mode collapse problems present in prior GANs, as now the generator always gains useful information from the critic regardless of how far off it is (assuming the critic has been trained to optimality). Hence, we actually train the critic in a WGAN much more at the start in order to ensure it provides meaningful data to the generator from the beginning, unlike previous GAN systems (which tried to keep the discriminator weak so it could be partially fooled by the generator, preventing the vanishing gradient problem). By using the wasserstein distance as the cost function instead of the Jensen-Shannon divergence, WGANs completely remove the chance of model collapse or vanishing gradients occurring, and remove the need for the discriminator and generator to be trained at a roughly equal rate, further stabilising GAN performance.

### Weight Clipping

However, there is one large issue with using the Wasserstein distance: in order to approximate the wasserstein distance, the weights of the critic have to lie in a compact space, as then K-lipschitz continuity is upheld. In order to ensure this, we simply clip the weights to be within a fixed range (usually [-0.01, 0.01]) after every gradient update. This is noted by the creators of the WGAN to be an awful way to enforce the lipschitz condition, and new networks, such as WGAN-GP, have managed to find better solutions to this problem[6]. However, for this assignment we chose to focus on the normal WGAN setup, and so used weight clipping to ensure lipschitz continuity. Without this, the discriminator

[4] Arjovsky, Chintala, and Bottou (2017), p4-6

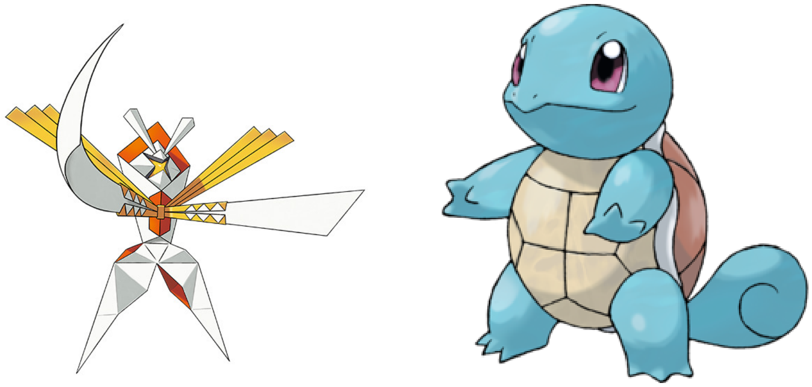
[5] If you wish to know more about the mathematics, please read (Arjovsky, Chintala, and Bottou 2017) for details.

[6] see Gulrajani et al. (2017)

ceases to be a good estimator of the wasserstein distance and so does not provide useful information to the generator.

### Pokémon and WGANs

Pokémon provide a fantastic dataset for investigating the behaviour of GANs: large datasets are easy to find, and the pokémon themselves are quite varied, especially with the introduction of ultra beasts in Pokémon Sun and Moon, which break many traditional pokémon design norms. Hence, we used a dataset collated from veekun (a great resource for pokémon images) for testing.



(a) Kartana

(b) Squirtle

Figure 1: Kartana and Squirtle, examples of ultra beast and traditional pokémon design respectively

### Our Implementation

For this assignment, we examined two main additions to a base WGAN: using upsampling combined with convolutional layers to reduce checkboarding in the generator, and using residual layers in the discriminator. This was based off of the work done in (Odena, Dumoulin, and Olah 2016) and (He et al. 2015) respectively, though the critic architecture used is not a full residual net. Rather, shallower version of the architecture was used. A baseline WGAN, based off the design described in the assignment specification, was also implemented. Both implementations are discussed below.

# Method

### Data Augmentation

Despite the thousands of pokémon images, this is still not enough to train a WGAN - rather, successful GANs need millions and millions of images to properly train. Hence, we use data augmentation techniques to pad our dataset. Several data augmentation techniques have been applied:

- random cropping
- random transforms (padding random edges and then cropping back to the usual size)
- horizontal flips
- vertical flips

Applying these not only augments our dataset with more images, but also will ideally result in WGAN learning features of pokémon invariant of translation or rotation, since more pokémon images feature the pokémon facing left or right in the centre of a white background. By randomly shifting the pokémon and flipping it, the WGAN will have to learn how to recognise and produce such features even when they are flipped or moved around, rather than learning them in a particular place in a particular orientation. This serves to make the WGAN more robust on new data, helping the network generalise better about features (as opposed to 'overfitting' on features in the middle of the image).

Data augmentation was implemented through the Keras *ImageDataGenerator*[7] class, which allows the user to set what augmentations they wish, then apply them randomly to a dataset stored within a folder. This creates a generator which produces batches of image data with the augmentations randomly applied. To demonstrate this, run `data_prep.py` from my codebase. This will generate some images with the data augmentations randomly applied.

The base WGAN used the following data augmentation parameters:

```
{
    rescale=1./255,
    zoom_range=0.2,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True,
    vertical_flip=True,
}
```

The rescale was required to bring images color values to a [0,1] range. Zoom refers to cropping (selecting random sections of the image and zooming into them), while shifting refers to random padding. Horizontal and vertical flip refer to flipping the images horizontally and vertically respectively.

[7] see the keras documentation for more details.

For the data augmentation experiment, these parameters were removed or tinkered with. More details can be found below, within the experiment section.

### Generator

Two generator architectures were used: *(i)* a base architecture utilising repeated layers of deconvolution, batch normalisation, and activation layers, and *(ii)* an augmented architecture consisting of repeated convolution, batch normalisation, activation, and upsampling layers.

The augmentation was modelled after work done in (Odena, Dumoulin, and Olah 2016), which suggested that deconvolution layers caused checkerboard artifacts due to uneven patterning in architectures, and suggesting using convolution and upsampling to avoid this issue. This is due to the deconvolutional layers operating on overlapping image segments, thereby causing the generator to 'care' more about certain pixels. This overlap occurs in a checkerboard fashion, thus causing the checkboard artifacts. Hence, applying upsampling and then convolution still allows recognition and generation of features, but while avoiding this overlapping issue and thus reducing artifacts.

In order to test the generator architectures, an image segmentation task was performed using the AD20K dataset.

### Base Generator Architecture

Below is a diagram of the baseline generator architecture. Rather than show the full architecture, I instead show the first few layers. 6 deconvolutional blocks were used in the architecture, along with input and output dense layers.

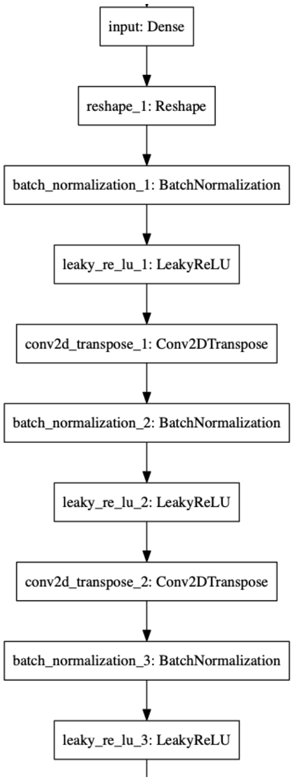

input: Dense
reshape_1: Reshape
batch_normalization_1: BatchNormalization
leaky_re_lu_1: LeakyReLU
conv2d_transpose_1: Conv2DTranspose
batch_normalization_2: BatchNormalization
leaky_re_lu_2: LeakyReLU
conv2d_transpose_2: Conv2DTranspose
batch_normalization_3: BatchNormalization
leaky_re_lu_3: LeakyReLU

Figure 2: Baseline generator architecture

### Augmented Generator Architecture

Below is a diagram of the augmented generator architecture. Rather than show the full architecture, I instead show the first few layers. 4 upsampling/convolutional blocks were used in the architecture, along with input and output dense layers.

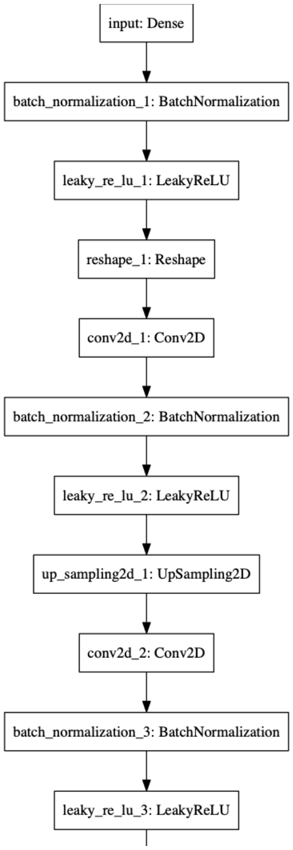

input: Dense
batch_normalization_1: BatchNormalization
leaky_re_lu_1: LeakyReLU
reshape_1: Reshape
conv2d_1: Conv2D
batch_normalization_2: BatchNormalization
leaky_re_lu_2: LeakyReLU
up_sampling2d_1: UpSampling2D
conv2d_2: Conv2D
batch_normalization_3: BatchNormalization
leaky_re_lu_3: LeakyReLU

Figure 3: Baseline generator architecture

### Discriminator

Two discriminator architectures were used: *(i)* a base architecture consisting of repeated convolution, batch normalisation, and activation layers, and *(ii)* an augmented architecture based off resnet[8]. The augmented architecture uses residual blocks, which allow the network to 'skip' convolutional layers. The intuition here is that the network is simultaneously very simple and quite deep, depending on if you take these shortcuts or not. Hence, the network can learn simple features fast and easily using only a few layers, and then slowly expand to use its entire depth when determining more complex features. As my computer GPU's memory was limited, I reduced the full resnet architecture to a few less layers, but still using the residual blocks. Each residual block is simply a set of standard convolutional blocks (repeated convolution, batch normalisation, and activation layers), with a shortcut from the first convolutional block to the end

[8] The architecture proposed in (He et al. 2015)

of the residual block. These residual blocks either have their own convolutions in the shortcut path (in order match the longer path's downsampling), or no convolution.

To test the discriminator architectures, a simple digit recognition task using the MNIST dataset was performed.

### Base Discriminator Architecture

Below is a diagram of the base discriminator architecture. Again, rather than showing the full network, I instead show the first few layers. In the full archiecture, 6 convolution blocks were used, with an output dense layer.

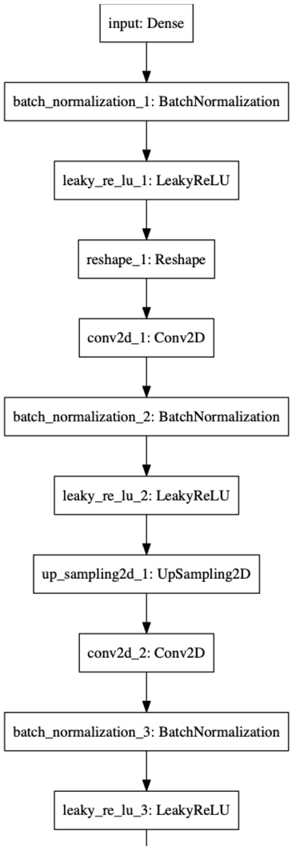

input: Dense
batch_normalization_1: BatchNormalization
leaky_re_lu_1: LeakyReLU
reshape_1: Reshape
conv2d_1: Conv2D
batch_normalization_2: BatchNormalization
leaky_re_lu_2: LeakyReLU
up_sampling2d_1: UpSampling2D
conv2d_2: Conv2D
batch_normalization_3: BatchNormalization
leaky_re_lu_3: LeakyReLU

Figure 4: Baseline generator architecture

### Augmented Discriminator Architecture

As the full architecture is quite deep, again, instead of showing the full network, below there is a diagram of the two main blocks of this network. Four of these blocks, as well as an input convolutional layer, and an output dense layer, made up the entire network.

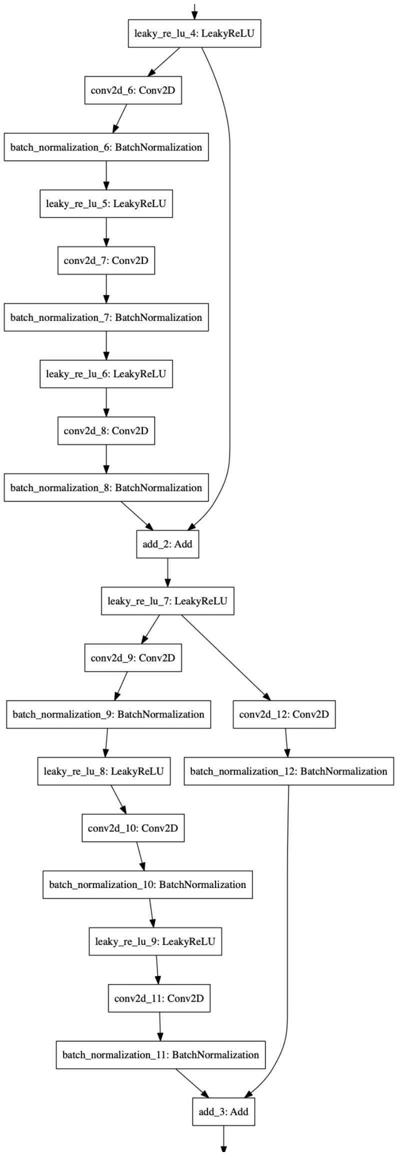

leaky_re_lu_4: LeakyReLU
conv2d_6: Conv2D
batch_normalization_6: BatchNormalization
leaky_re_lu_5: LeakyReLU
conv2d_7: Conv2D
batch_normalization_7: BatchNormalization
leaky_re_lu_6: LeakyReLU
conv2d_8: Conv2D
batch_normalization_8: BatchNormalization
add_2: Add
leaky_re_lu_7: LeakyReLU
conv2d_9: Conv2D
batch_normalization_9: BatchNormalization
conv2d_12: Conv2D
leaky_re_lu_8: LeakyReLU
batch_normalization_12: BatchNormalization
conv2d_10: Conv2D
batch_normalization_10: BatchNormalization
leaky_re_lu_9: LeakyReLU
conv2d_11: Conv2D
batch_normalization_11: BatchNormalization
add_3: Add

Figure 5: Resnet-inspired discriminator architecture

### Overall WGAN Setup

I implemented a custom training program, as the Keras library did not natively support WGAN training. As based on advice in (Arjovsky, Chintala, and Bottou 2017), the discriminator was trained on 5 batches of data for every 1 batch on the generator. The exception was the first 25 epochs (or every 500 epochs thereafter), in which the discriminator was trained on 100 batches for every one batch on the generator. Note that here, an epoch is not a full run through the dataset, but rather a single batch being trained on the generator. Weights were clipped to the range [-0.01, 0.01]. Images were also rescaled so that RGB values were in the range [-0.5, 0.5], rather than [0,1]. This is all based off the original WGAN code, found here.

## Experiment

### Generator and Discriminator Experiments

In order to determine the best setup, I ran four different WGAN configurations for 20,000 epochs each:

- baseline discriminator and baseline generator (the baseline WGAN)
- baseline discriminator and improved generator (the improved generator WGAN)
- resnet discriminator and baseline generator (the improved discriminator WGAN)
- resnet discriminator and improved generator (the improved WGAN)

Below I graph the loss values for the generators of each WGAN:

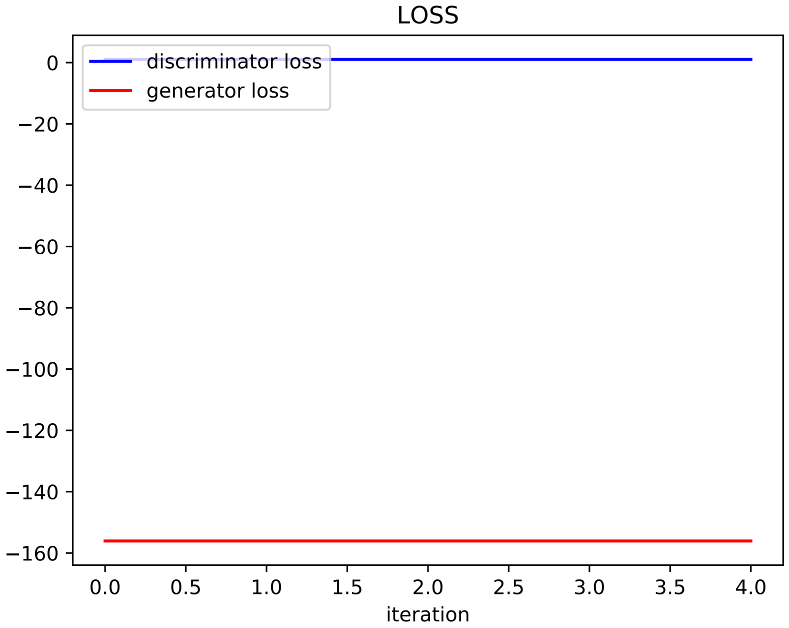


Figure 6: Example figure proudly made with matplotlib's PGF backend.

Below I graph the loss values for the discriminators of each WGAN:

Below are the images produced by each WGAN after 20,000 iterations:

As we can see…

### Data Augmentation Experiments

In order to determine the effect of data augmentation on WGAN performance, I used the WGAN setup INSET HERE on the following data augmentation setups for 20,000 iterations:

- no data augmentation
- flipping only
- padding only
- random zoom only

Below I graph the loss values for the generators of each setup:

Below I graph the loss values for the discriminators of each setup:

Below are the images produced by each setup after 20,000 iterations:

As we can see…

## Conclusion

conclude report and summarise results

## Bibliography

Arjovsky, M., S. Chintala, and L. Bottou. 2017. "Wasserstein GAN." *ArXiv E-Prints*, January.

Goodfellow, I. J., J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. 2014. "Generative Adversarial Networks." *ArXiv E-Prints*, June.

Gulrajani, I., F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville. 2017. "Improved Training of Wasserstein GANs." *ArXiv E-Prints*, March.

He, K., X. Zhang, S. Ren, and J. Sun. 2015. "Deep Residual Learning for Image Recognition." *ArXiv E-Prints*, December.

Odena, Augustus, Vincent Dumoulin, and Chris Olah. 2016. "Deconvolution and Checkerboard Artifacts." *Distill.* doi:10.23915/distill.00003.