Dataset Distillation with Deep Learning

by

Herbie Bradley

Supervisor: Professor Nathan Griffiths

Department of Computer Science

University of Warwick

September 2020

Abstract

In deep learning, dataset reduction is a technique which aims to speed up training while maintaining accuracy by identifying the most representative or important examples from a dataset while discarding the rest. Recent work with convolutional neural networks extends the dataset reduction concept to dataset distillation - distilling a large dataset into a small number of *synthetic* examples. This distillation is achieved by optimising a loss function parameterised by both the network weights and the synthetic training data (which starts as random noise). This technique can be used with any model as long as the loss function is twice-differentiable and the gradient can be calculated with respect to the input data.

Distillation aims to generate a set of synthetic data which can train a machine learning model to a given accuracy faster or in fewer optimisation steps than the original dataset - the most promising application for this is to speed up hyper-parameter or architecture searches. In this project, we first attempted to explore the application of dataset distillation as a way of accelerating the training of architecture search algorithms. However, using distilled data for this application gave poor results and showed that the distillation algorithm is inherently too architecture-dependent to use well with architecture search.

We then moved onto attempting to find a novel dataset distillation algorithm. Since generative adversarial networks (GANs) are the most popular class of generative image model, they seem like a good candidate for this problem. In this project, we looked at both of the two possible ways of generating images using GANs: modifying the loss function to train a GAN to *generate* distilled images, and attempting to use gradient descent in the latent space of an already trained GAN to *find* distilled images.

The results showed that although changing the GAN loss function can increase the final test accuracy above that obtained with data from a normal GAN, none of the experiments achieved accuracy with GAN generated data that is above the baseline accuracy at any point. This indicates that the GAN generated data fails to effectively train the network faster.

The results from the latent space experiments also failed to produce useable data but showed that the optimisation procedure quickly strayed into areas of the latent space which produced incoherent generated images. Therefore, further work on constraining the gradient descent in the latent space may yield distilled data. Overall, this project's most important result is that distilled data can most likely *not* be generated by modifying a GAN loss function, but that performing gradient descent in the latent space of a GAN may be a useful direction for further work.

Acknowledgements

I would like to thank my supervisor Professor Nathan Griffiths for his invaluable advice and support with this project.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Artificial neural networks are statistical models inspired by the brain, commonly used in machine learning as universal function approximators [11]. In deep learning - the study of neural networks - dataset reduction [29], [55] is a technique which aims to speed up training while maintaining accuracy by identifying the most representative or important examples from a dataset while discarding the rest.

Recent work with convolutional neural networks (CNNs) - a network architecture specialised for processing images - extends the dataset reduction concept to **dataset distillation** [57], [64] - distilling a large dataset into a small number of *synthetic* examples. More precisely, for this dissertation, we shall define 'distilled data' to mean any data generated using some optimisation procedure on a dataset, which trains a machine learning model to a given accuracy faster or in fewer optimisation steps than the original dataset. The distillation in [64] is achieved by optimising a loss function parameterised by both the network weights and the synthetic training data (which starts as random noise). This technique can be used with any model as long as the loss function is twice-differentiable and the gradient can be calculated with respect to the input data.

This project has the following objectives or research goals:

1. To investigate whether dataset distillation can help speed up Neural Architecture Search algorithms, and to experiment with different ways of initialising the distillation algorithm to gain more insight.
2. To investigate whether generative adversarial networks - a type of generative deep learning model - can be used to generate distilled data that can train a network to a given accuracy faster than the original dataset.

In Chapter 2, we introduce the dataset distillation algorithm in detail, along with the necessary background material to understand the experiments in Chapter 4 . This chapter also serves as a comprehensive literature review of both distillation algorithms and the state of research into generative adversarial networks.

Next, we look more deeply at the distillation algorithm [64] in Chapter 3 and carry out several experiments to understand its properties more deeply. We also attempt to apply it to Neural Architecture Search - a class of machine learning algorithms that are designed to automatically find the optimal neural network architecture for solving a particular problem. These experiments fulfil the first objective listed above.

Since generative adversarial networks (GANs) are the most popular class of generative image model, they seem like a good candidate for the problem of generating distilled data. Therefore, in Chapter 4 we look at both of the two possible ways of generating images using GANs: modifying the loss function to train a GAN to *generate* distilled images, and attempting to use gradient descent in the latent space of an already trained GAN to *find* distilled images.

A detailed description of the management of this project is provided in Chapter 5 - how it was planned, and how the project objectives have changed over time. Finally, the dissertation is wrapped up with Chapter 6, which examines possible reasons for the main results of the experiments, provides a direction for future work, and summarises the project in the conclusion.

2 Background

In this chapter, we will explain the necessary background material to understand the dataset distillation experiments described later in this dissertation. We begin with a brief description of the foundational concept of **convolutional networks** (CNNs), before looking at two popular improvements to convolutional networks: **batch normalisation** and **residual connections**. Together, these architecture changes greatly improve the ability of deep convolutional networks to converge to a solution efficiently.

Next, we give an overview of **generative adversarial networks** (GANs), a type of generative model that will be used in the experiments in Chapter 4. Finally, we explain the basic **dataset distillation** algorithm that we will investigate in Chapter 3.

2.1 Convolutional Networks

A convolutional neural network (CNN) is a network composed of multiple convolutional layers, often followed by one or more fully-connected layers if the task requires a scalar output. These layers are optimised for processing data in the form of a grid, where it is useful to take advantage of the fundamental structure of data, such as the correlations between local groups of values found in image data (a 2D grid of pixels) and time series data (a 1D grid) [31].

A convolutional layer for image data consists of weights arranged in three dimensions: width, height, and depth (a convolutional layer for 1D data would have two dimensions). The parameters of the layer are a set of filters - each filter is a small grid of weights that extends through the full depth of the input volume. During the forward pass, the filters slide across the input and at each point take the dot product of the region under the filter with the weights of the filter. This creates the width and height of the convolutional layer's output, and this is repeated with any number of filters to produce an output volume with the same depth as the number of filters [11].

There are two key properties of image data that convolutional layers can exploit. First, local regions of values are usually closely related; second, a pattern that appears in one part of an image could appear in an entirely different part of the same image - this is called translational invariance [5]. Convolutional layers can exploit the first property (local connectivity) because each weight in the convolutional layer only depends on a small local region in the input image. The weights of convolutional filters are also shared across each depth slice in the output volume, which allows the layer to capitalise on the translational invariance of images to extract features. Both properties also give the convolutional layer many times fewer weights than a similar fully connected layer, greatly speeding up learning [11].

2.1.1 Batch Normalisation

One common issue with training deep networks is the vanishing gradient problem [18]. Backpropagation calculates gradients via the chain rule, which requires the derivatives of each layer to be multiplied together to calculate the gradient of the first layers. When each layer has an activation function that saturates (a function with zero gradient when the input tends to ∞ or $-\infty$) such as the tanh or sigmoid functions, these small derivatives get multiplied together and eventually the derivatives of the layers will be driven towards zero. As more gradients get close to zero the network trains slower, which can cause the network to converge to lower accuracies when more layers are added despite the increase in model capacity.

This problem is one of the reasons ReLU ($f(x) = \max(0, x)$) is the most popular activation function. Since it has a slope of either 0 or 1 if the input to the ReLU is greater than 0 the gradients will remain the same.

Batch Normalisation is a technique for deep neural networks designed to solve the issue of internal covariate shift. This refers to the change in the distribution of the inputs to different layers during training. When the weights of the preceding layers in a network change, it causes the distribution of inputs to the current layer to also change, resulting in the current layer needing to continually adjust to the new distributions. This can greatly slow down the convergence speed of the network [20], and particularly affects convolutional networks.

Batch normalisation normalises the output of each layer to have zero mean and unit variance - these mean and variance statistics are calculated across each mini-batch of training data, as the name implies. This procedure is applied after every layer, but before the activation function to produce activations with normalised distributions. By ensuring that the distributions of each layer's weights are kept roughly the same throughout training, Batch Normalisation can speed up the convergence of deep neural networks, allow the use of higher learning rates without vanishing gradients, and also act as a form of regularisation. As a result, Batch Normalisation is now used by default in deep convolutional networks and has even been shown to make the optimisation landscape much smoother [53].

2.1.2 Residual Networks

Residual networks [15] are a standard addition to convolutional networks designed to fix the problem of slow training and vanishing gradients in very deep networks. They have shown successful training of CNNs up to 1000 layers deep. The key ingredient is the residual block: two convolutional layers in which the input to the first layer is added to the output of the second layer, forming a skip (or residual) connection. Modern CNNs are usually composed of a series of residual blocks [50], [58].

When we add a residual block to a network we can be certain that in the worst case it will not reduce the network performance, since the layers inside the residual block can learn to drive their weights towards zero - if this happens the block becomes an identity mapping thanks to the skip connection. This has the effect of reducing overfitting since the network as a whole becomes simpler - which reduces variance. The identity connections create shorter paths through the network, allowing the gradient to propagate efficiently [16]. Residual networks also reduce the vanishing gradient problem, although that issue is mostly addressed by the use of batch normalisation and ReLU activation functions [14].

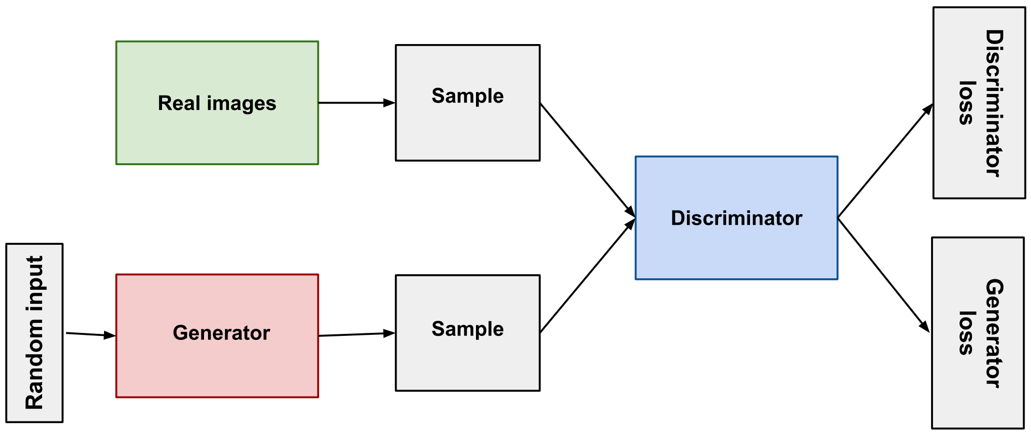

Figure 2.1: The GAN framework. Figure reproduced from [43].

2.2 Generative Adversarial Networks

Generative adversarial networks (GANs) [12] are a type of generative deep learning model that can generate realistic data in many domains, such as images [3], language [38], and music [7]. Since GANs are predominantly used for computer vision tasks, we shall only deal with image-based GANs in this dissertation.

A GAN consists of two neural networks: a discriminator network and a generator network. The generator takes as input a vector $\boldsymbol{z}$ (sampled from an n-dimensional distribution of Gaussian noise) and outputs an image. The space that $\boldsymbol{z}$ samples from is called the latent space, and it is usually relatively low-dimensional compared to the images the GAN is trying to generate. The discriminator alternates between evaluating the generated data and real data from the distribution we want the generator to approximate. The discriminator acts as a binary classifier, estimating the probability that its input is from the real dataset (Figure 2.1).

This is an unsupervised learning framework that takes its motivation from game theory, since it can be viewed as analogous to a game between two adversaries: a forger, trying to make fake currency, and the police, trying to differentiate between real and fake currency [10].

To talk about the training procedure in detail, we must first define some terms. Let $p_{\boldsymbol{z}}(\boldsymbol{z})$ be the prior distribution of noise (almost always Gaussian), let $p_{model}(\boldsymbol{x})$ be the generator's distribution over data $\boldsymbol{x}$, and let $p_{data}(\boldsymbol{x})$ be the distribution over the real data $\boldsymbol{x}$. Then let $G(\boldsymbol{z};\theta_g)$ and $D(\boldsymbol{x};\theta_d)$ be differentiable functions represented by neural networks, where $G(\boldsymbol{z})$, $\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})$ represents a generated (fake) sample. Note that a sigmoid function is applied to the output of D so that $D(\boldsymbol{x})$ represents the probability that $\boldsymbol{x}$ came from the real data rather than from p_{model}.

We want to train D to maximise the probability of assigning the correct label to real samples – in other words to maximise $\mathbb{E}_{\boldsymbol{x}\sim p_{data(\boldsymbol{x})}} \log D(\boldsymbol{x})$. This is equivalent to maximising $\mathbb{E}_{\boldsymbol{x}\sim p_{data(\boldsymbol{x})}} D(\boldsymbol{x})$ because the logarithm is a monotonically increasing function - therefore applying it does not change the location of the maximum. When given a fake sample, we also want to train D to output a value for $D(G(\boldsymbol{z}))$ that is close to zero, and we can do this by maximising $\mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}(\boldsymbol{z})}} \log(1 - D(G(\boldsymbol{z})))$.

Combining these two terms gives a standard binary cross-entropy loss for the discriminator [10] - the same loss that is minimised when training a binary classifier.

Meanwhile, G should be trained to maximise $D(G(\boldsymbol{z}))$, and we can do this by minimising $\mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}(\boldsymbol{z})}} - \log D(G(\boldsymbol{z}))$. Intuitively, this makes sense since we want the generator to fool the discriminator as much as possible - this is achieved if $D(G(\boldsymbol{z}))$ is large. If we combine all of these functions, D and G are playing a *minimax game* to optimise the following loss function:

$$\min_G \max_D \mathcal{L}(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{data(\boldsymbol{x})}} \log D(\boldsymbol{x}) + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}(\boldsymbol{z})}} \log(1 - D(G(\boldsymbol{z}))). \qquad (2.1)$$

In theory, as long as both models have sufficient capacity and the dataset is infinite, this game reaches a Nash equilibrium (a global optimum) when $p_{model} = p_{data}$, and $D(\boldsymbol{x}) = \frac{1}{2}$ for all $\boldsymbol{x}$ [12]. Equation 2.1 can be shown to minimise the Jensen-Shannon divergence between p_{model} and p_{data} [10]. Much of the literature proposing different architectures or loss functions for GANs is based around the idea of minimising some divergence or metric between these two probability distributions [65]. In practice, GAN training is implemented with the following generalised algorithm:

Algorithm 1: GAN training algorithm with the original mimimax training objective [12]. Many recent works have proposed different loss functions and hyperparameters, but few change this basic framework.

```
for training steps $t$ = 1 to $T$ do
    Sample a mini-batch $\{\boldsymbol{z}^{(1)}, \dots, \boldsymbol{z}^{(m)}\}$ from the noise distribution
      $p_{\boldsymbol{z}}(\boldsymbol{z}),\ \boldsymbol{z}^{(i)} \in \mathbb{R}^n$.
    Sample a mini-batch $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ from the real data distribution
      $p_{data}(\boldsymbol{x})$.
    for $k$ = 1 to $n_{dis}$ do
        Update the discriminator's parameters $\theta_d$ with the gradient:
            $$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} -\big[\log D(\boldsymbol{x}^{(i)}) + \log(1 - D(G(\boldsymbol{z}^{(i)})))\big]. \qquad (2.2)$$
    end for
    Update the generator's parameters $\theta_g$ with the gradient:
        $$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} - \log D(G(\boldsymbol{z}^{(i)})). \qquad (2.3)$$
end for
```

The hyperparameter n_{dis} is the number of times the discriminator is updated for each generator update. By default, this hyperparameter is 1 which often results in more unstable training. Several works [3], [13], [41] increase this hyperparameter, which has the effect of training the discriminator to convergence between each generator update, increasing the stability of training.

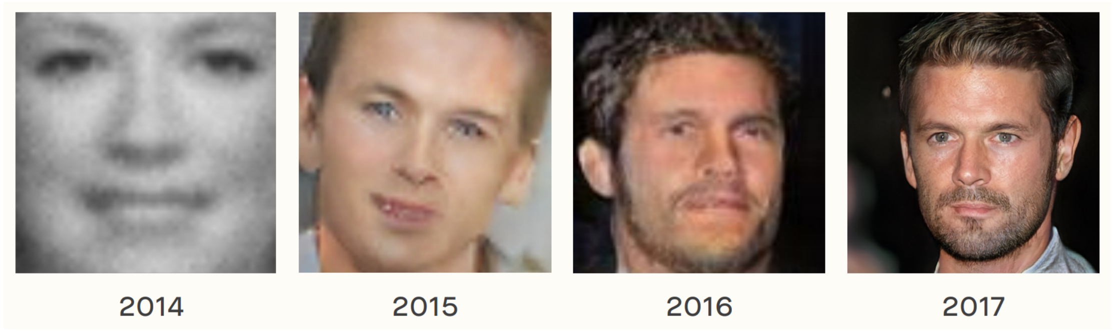

Figure 2.2: Improvements in GAN face generation from 2014 to 2017. From left to right, the images are from the papers [12], [48], [34], and [21]. Figure reproduced from [4].

2.2.1 Problems with GAN Training

As described above, training a GAN to convergence requires finding a Nash equilibrium to a non-convex game with high dimensional parameters. Using simultaneous gradient descent to try and find an equilibrium for this game can result in a failure to converge since in many situations changing θ_d to reduce D's loss can increase G's loss, and vice-versa [52]. This is one of the primary causes of the phenomenon known as mode collapse, which occurs when the generator fails to learn the full distribution of the training dataset, causing it to map different values of $\boldsymbol{z}$ to the same or very similar generated samples. As a result, many different GAN variants [65] have been proposed to stabilise training, reduce mode collapse and generate higher quality samples.

The primary design choices of GAN variants can be divided into 3 distinct categories: architecture, loss function, and regularisation/normalisation techniques. GAN architectures come in two varieties: those with a fully-convolutional G and D [48], and those with ResNet-based G and D, containing skip connections [15]. Normalisation techniques, such as Batch Norm [20] or Spectral Norm [41], aim to stabilise the discriminator during training by dividing the activations or weights by some factor. Regularisation in GANs usually refers to any kind of penalisation of the discriminator - one common way of doing this is by constraining the gradient norm to make sure that D is Lipschitz continuous [13]. A function $f : \mathbb{R}^n \to \mathbb{R}^m$ is defined as Lipschitz continuous if $\exists L \geq 0, L \in \mathbb{R}$ such that

$$\forall x, y \in \mathbb{R}^n, ||f(x) - f(y)||_p \leq L||x - y||_p. \tag{2.4}$$

The smallest possible L for which this is true is called the Lipschitz constant. In other words, changing the input of f by a certain amount cannot change its output by more than L times that amount. By default, $p = 2$, but these can be any metric (including non-p-norms). Several GAN papers [2], [13] have shown that forcing D to be Lipschitz continuous improves training stability and acts as a form of regularisation.

Aside from mode collapse, the other major issue in GAN research is that GANs suffer from a lack of a good evaluation metric [28]. For image generation GANs the two most commonly used metrics to evaluate the quality of generated images are Fréchet Inception Distance [17] and Inception Score [52], which both use the hidden layers and output of an InceptionV3 CNN [59] trained for classification on ImageNet to generate a perceptual metric that scores the quality of generated images. However, this means these metrics are of dubious value when we are evaluating a GAN that does not generate images of one of the 1000 ImageNet classes. Without a good metric, determining the best GAN architecture or loss function becomes more challenging.

The GAN framework is very general, and a wide variety of improvements and alternative architectures have been proposed within this framework, some producing extremely realistic results [22] (Figure 2.2). Despite the difficulty of training and evaluation, GANs have proven to be the most popular and effective class of deep learning generative model, in contrast to Variational Autoencoders [25] and Generative Flows [6].

2.2.2 Conditional GANs

So far we have only discussed unconditional GANs - when trained on a multi-class dataset such as CIFAR10 [26], these models output a generated image with a random class and can even interpolate between classes. In contrast, [40] introduces the idea of a conditional GAN. This model allows for generating images conditioned on some vector $\boldsymbol{y}$, usually a one-hot encoded class label. To condition on $\boldsymbol{y}$, this vector is simply combined with $\boldsymbol{z}$ into a joint hidden layer in both G and D.

A later work introduced a better performing method of incorporating the conditional information - the projection discriminator [42]. This involves an embedding layer that takes as input the class label. The output of this layer is multiplied with the output of the final convolutional layer in D, and the result is added to the output of the final (fully-connected) layer. This method tends to produce images that are more diverse and with better quality than the original conditioning technique described above.

The projection discriminator is combined with conditional batch normalisation [63] in the generator, to condition G on the class label. This is the same as regular batch normalisation, except that instead of learning the parameters γ and β, they are generated from an embedding layer (with the one-hot encoded class label as input to the embedding). This provides a convenient way to add conditional information to a network - even a pre-trained network - with only a small modification.

2.2.3 Spectral Normalisation

In [23], Keskar et al. use the fact that shallower local minimums of loss functions generalise better than sharper ones [19] to show that the eigenvalues of the Hessian matrix of the loss function defined with the training data predict the generalisation ability of a neural network. Further work [66] shows that regularising the spectral norm of the weight matrices can improve generalisation in deep neural networks.

Formally, the spectral norm of a real matrix W is defined as

$$\|W\|_2 = \sqrt{\lambda_{max}(W^\top W)}. \tag{2.5}$$

This is the square root of the largest eigenvalue of $W^\top W$, or equivalently the largest singular value of W. In spectral normalisation, each weight matrix W in a neural network is divided by its spectral norm on each iteration to force $\|W\|_2 = 1$, which ultimately constrains the Lipschitz constant of the network.

This concept has been applied to regularise the discriminator of a GAN [41], with great success - the Spectral Normalisation (SNGAN) model was the first to generate images from all 1000 ImageNet (ILSVRC2012) [51] classes with an unconditional GAN, showing that this setup is capable of high image diversity. A GAN with a ResNet-based architecture and spectral normalisation is often considered a strong baseline for further GAN experiments [65], and provides a good trade-off between performance and training time. For this reason, we used a SNGAN model with a conditional projection discriminator as the basis for our GAN experiments, described in Section 4.

2.3 Dataset Distillation

The dataset distillation algorithm [64] extends the concept of dataset reduction further - instead of selecting the most *important* data points, the algorithm compresses a large dataset into a small number of *synthetic* samples. This paper carries out all its experiments on image data. The basic technique aims to create a small set of distilled data by finding an input batch $\widetilde{\boldsymbol{x}}$ and a learning rate $\widetilde{\eta}$ that will train the entire network to high accuracy in a single gradient descent step.

The algorithm to obtain $\widetilde{\boldsymbol{x}}$ and $\widetilde{\eta}$ starts with the synthetic data points $\widetilde{\boldsymbol{x}}$ as random noise and performs a gradient descent step on the initial network weights θ_0 using the synthetic data. Then the loss function is recalculated using the updated weights θ_1 with real training data $\boldsymbol{x}$, and $\widetilde{\boldsymbol{x}}$ is updated using gradient descent on this loss. These steps are repeated many times, iterating over the training data $\boldsymbol{x}$. In particular, the following objective is minimised:

$$\widetilde{\boldsymbol{x}}^*, \widetilde{\eta}^* = \underset{\widetilde{\boldsymbol{x}}, \widetilde{\eta}}{\arg\min}\, \mathcal{L}(\boldsymbol{x}, \theta_1) = \underset{\widetilde{\boldsymbol{x}}, \widetilde{\eta}}{\arg\min}\, \mathcal{L}(\boldsymbol{x}, \theta_0 - \widetilde{\eta} \nabla_{\theta_0} \mathcal{L}(\widetilde{\boldsymbol{x}}, \theta_0)), \tag{2.6}$$

where θ_1 is calculated via gradient descent using $\widetilde{\boldsymbol{x}}$ and θ_0 (the initial weights). The only assumption made is that the loss function $\mathcal{L}$ is twice-differentiable - then any gradient-based optimisation method can be used to calculate $\widetilde{\boldsymbol{x}}$ and $\widetilde{\eta}$.

This optimisation objective is similar to the Model-Agnostic Meta-Learning (MAML) [9] objective, but where MAML searches for a *weight initialisation* such that a small number of gradient descent steps will produce good accuracy, the distillation algorithm looks for a *synthetic input* such that a small number of gradient descent steps will produce good accuracy. In other words, distillation tries to find a small number of synthetic input data such that the gradient with respect to the weights is similar to the gradient computed on the full original dataset.

The above procedure creates distilled data $\widetilde{\boldsymbol{x}}$ that is tied to the weights θ_1 - this means that to get the correct performance these weights must be used at the start of training (the *fixed weight algorithm*). This is not very useful in practice, so to create input data which generalises to different models the authors repeatedly re-initialise the network weights during the distillation algorithm (the *random weight algorithm*).

Algorithm 2: The dataset distillation [64] algorithm. This version of the algorithm is for a random weight initialisation and M gradient descent steps with distilled data. Note that T is split into several epochs over the training dataset - the default is 400 and the default batch size n is 1024.

```
Input: $p(\theta_0)$: distribution of initial weights; $M$: the number of distilled data
Input: $\alpha$: step size; $n$: batch size; $T$: the number of optimisation iterations; $\tilde{\eta}_0$:
       initial value for all $\tilde{\eta}_i$
Initialise $\tilde{\boldsymbol{x}} = \{\tilde{x}_i\}_{i=1}^{M}$ randomly, $\{\tilde{\eta}_i\}_{i=1}^{M} \leftarrow \tilde{\eta}_0$
for training steps $t = 1$ to $T$ do
    Get a mini-batch of real training data $\boldsymbol{x}_t = \{x_{t,j}\}_{j=1}^{n}$
    Randomly initialise the weights $\theta_0 \sim p(\theta_0)$
    for $i = 1$ to $M$ do
        Update weights for step $i$ with distilled data $\tilde{\boldsymbol{x}}_i$: $\theta_i = \theta_0 - \tilde{\eta}_i \nabla_{\theta_0} \mathcal{L}(\tilde{\boldsymbol{x}}_i, \theta_0)$
        Update distilled data for step $i$: $\tilde{\boldsymbol{x}}_i \leftarrow \tilde{\boldsymbol{x}}_i - \alpha \nabla_{\tilde{\boldsymbol{x}}_i} \mathcal{L}(\boldsymbol{x}_t, \theta_i)$
        Update learning rate for step $i$: $\tilde{\eta}_i \leftarrow \tilde{\eta}_i - \alpha \nabla_{\tilde{\eta}_i} \mathcal{L}(\boldsymbol{x}_t, \theta_i)$
    end for
end for
Output: Distilled data $\tilde{\boldsymbol{x}} = \{\tilde{x}_i\}_{i=1}^{M}$ and optimised learning rates $\tilde{\eta} = \{\tilde{\eta}_i\}_{i=1}^{M}$
```

Algorithm 2 describes the distillation algorithm in detail for the case where we want to generate M distilled data points each with M corresponding learning rate values, which will train the network in M steps of gradient descent. Note that the M data points may be repeated several times as multiple epochs, which can improve the results [64]. The trained learning rates $\tilde{\eta}_i$ are not tied across epochs, so there are $M \times$ *distilled epochs* unique learning rates, but only M unique distilled data points.

The authors of [64] evaluate this technique on small to medium size image classification datasets, such as MNIST [32] and CIFAR-10 [26], using basic convolutional neural network architectures such as LeNet [30] or AlexNet [27]. The results show that these networks can be trained with synthetic data to high accuracy, although it is still below that of the networks trained on the original dataset.

The highest accuracies were achieved when the weights were fixed during optimisation - the fixed networks achieve a test accuracy of 96% on MNIST after training on just 10 images (compared to 99% when trained on the full dataset) and 54% on CIFAR-10 (compared to 80%). The random networks achieve an accuracy of 79.5% on MNIST and 36.8% on CIFAR-10. When trained with random weights, the distilled images also clearly show a distorted version of the labels for each class.

The paper also shows algebraically that for a simple linear regression example, M, the number of distilled data samples must be at least D (the dimensionality of the dataset) to reach the global minimum in a single gradient descent step. If this lower bound holds for more complex problems it must imply quite large values of M for some datasets, particularly image datasets. Therefore, the authors show that in practice, this lower bound is not in force when using multiple gradient descent steps. The paper shows that using AlexNet weights pre-trained for classification on the ImageNet dataset, dataset distillation can learn a small number of synthetic examples for the PASCAL-VOC classification dataset that give good test accuracy - almost on par with a full transfer learning approach [64].

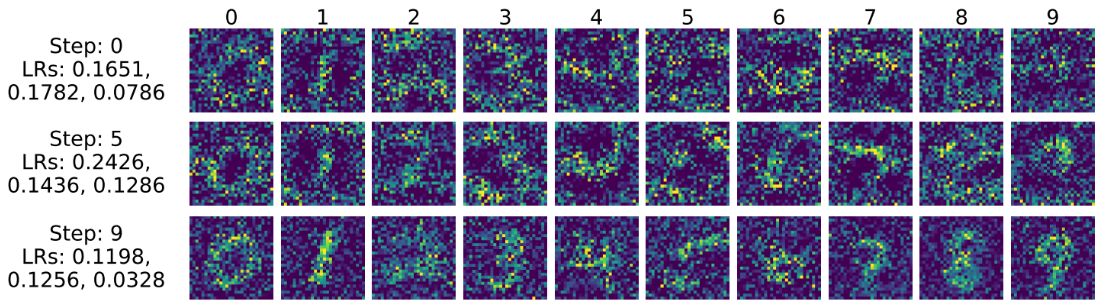
0
1
2
3
4
5
6
7
8
9
Step: 0
LRs: 0.1651, 0.1782, 0.0786
Step: 5
LRs: 0.2426, 0.1436, 0.1286
Step: 9
LRs: 0.1198, 0.1256, 0.0328

Figure 2.3: Distilled images trained from MNIST with random weight initialisation. These images are designed to learn MNIST in 10 gradient descent steps. Figure reproduced from [64].

Finally, the paper shows that the dataset distillation algorithm can be used to construct adversarial examples - examples that corrupt or greatly decrease the accuracy of a model when trained upon. Synthetic data constructed like this does not depend on access to the model that is being attacked, only on knowledge of the dataset used by the model. In a single gradient descent step, adversarial synthetic data can cause the attacked model to classify incorrectly a large percentage of examples from a particular class.

2.3.1 Soft-Label Dataset Distillation and Text Dataset Distillation

A followup paper [57] improves the basic technique of dataset distillation by allowing for class labels to take any real value (rather than being fixed), and modifying the algorithm to learn synthetic labels $\widetilde{\boldsymbol{y}}$ as well as synthetic data $\widetilde{\boldsymbol{x}}$. This results in a roughly 3% accuracy increase with the same datasets and hyperparameters.

The authors of [57] also extend the technique to text data by converting the dataset to pre-trained GloVe embeddings [45]. Once in the form of an embedding, text data can be distilled in the same way as image data. However, the resulting synthetic data samples are also in the form of embeddings, so they cannot be trained on directly - the real data is instead passed through the synthetic embedding layer. The paper evaluates this technique on several well-known NLP datasets including the IMDB movie reviews dataset [36] and the Text Retrieval Conference question classification dataset [62] and achieves test accuracies of 70 to 90% of that produced by a non-distilled approach, depending on whether fixed or random initialisation is used.

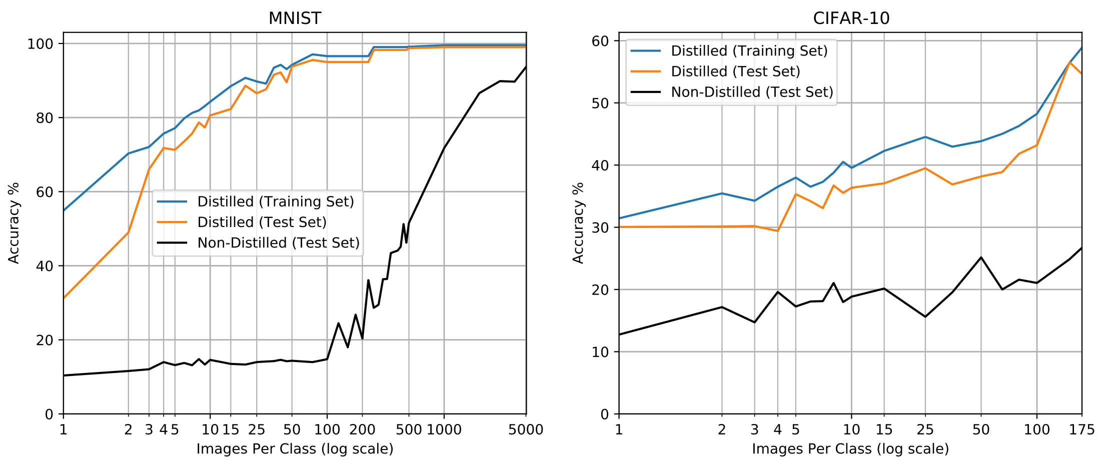

Figure 3.1: Accuracy scores on MNIST & CIFAR-10 with distilled data using random initialisations, plotted against the number of images per class in the training set. The black lines show the performance with random subsets of the original dataset. In all cases, training was done for only 3 epochs.

3 The Distillation Algorithm, Investigated

In this chapter, we look more deeply at the distillation algorithm [64] and carry out several experiments to understand its properties more deeply, including an attempt to train the algorithm with mixed proportions of fixed vs random weights. We also attempt to apply distillation to Neural Architecture Search - a class of machine learning algorithms that are designed to automatically find the optimal neural network architecture for solving a particular problem. These experiments fulfil project objective 1 listed in the Introduction.

Our first experiment was done to obtain more insight into the distillation algorithm than is available in the paper [64]. We ran the algorithm with varying numbers of images per class and compared with accuracies obtained by training for the same number of epochs on subsets of the original dataset. In total this involved 130 separate trained networks, with the distillation training times ranging from 1 hour to 5 days - the results are shown in Figure 3.1. We can see that the distilled data trains the network far more effectively than the original data at low numbers of images per class. On MNIST, the distilled data even does better than 3 epochs of the full original dataset at 250 images per class and above, but this is likely because MNIST is a very easy dataset to solve since the same effect is not seen with CIFAR-10.

To carry out the experiments in this chapter, we modified the freely available code for dataset distillation by the paper authors, hosted on Github and written in PyTorch. At the start of the project, this repository was downloaded and run on the MNIST and CIFAR-10 datasets, which produced results almost identical to those in the paper [64] - verifying that the paper's results are reproducible.

3.1 Neural Architecture Search

3.1.1 Background

Neural Architecture Search (NAS) is an umbrella term for several different types of algorithms that are designed to find the optimal neural network architecture for solving a problem, often using reinforcement learning-based approaches [8]. In some cases, NAS algorithms can find networks that outperform the best human-designed architectures for a particular task.

Traditional Neural Architecture Search (NAS) [67] uses a controller in the form of a recurrent neural network, which samples an architecture from the search space (the child model), trains it to convergence, then evaluates the child model on a task. The performance of the child model is used as a signal to optimise the controller network towards finding better architectures.

This procedure is naturally very time consuming, since the algorithm trains all child models to convergence before throwing away the trained weights of all but one candidate architecture. As a result many NAS algorithms, such as NAS-Net [68], take days of training with dozens of GPUs to find good architectures.

The main idea of Efficient Neural Architecture Search (ENAS) [46] is that it forces all child models to share their weights. In other words, when two child models have been given the same sub-network they share the weights of that sub-network - which means that once these weights have been trained to convergence they do not need to be trained in all future child models that have that sub-network. This is un-intuitive because naively, we might think that networks need different weights to optimise different architectures. The motivation behind this idea comes from transfer learning and multi-task learning, which established that parameters learned for one task can generalise well to another task [35], [49] - although in ENAS it is the model, not the task which changes.

However, ENAS shows that this idea works well, achieving test error on CIFAR-10 and the Penn Treebank dataset [37] comparable or superior to that of the original NAS algorithm. Most importantly, the computation time in GPU-hours decreased by more than 1000x times compared to NAS, allowing for architecture search to take less than 16 hours on a single GPU.

3.1.2 Experiments

Despite the relatively fast training times of ENAS [46], this algorithm can still take large amounts of time for sizeable datasets. Therefore we aim to speed up ENAS by investigating ways of applying dataset distillation to Neural Architecture Search, with the goal of detecting performance differences in the discovered architectures when the search is done with distilled data, compared with normal data.

The first experiments evaluated distillation with ENAS, by altering the learning rate (which is shared by all child models in the architecture search space) to the synthetic learning rates produced by the distillation algorithm. Code in PyTorch that implements ENAS was downloaded from Github and modified to achieve this. After obtaining an optimal network with ENAS, this network was trained on the original datasets (both MNIST and CIFAR-10 were used).

For MNIST, the accuracy of the ENAS-discovered network trained on the MNIST

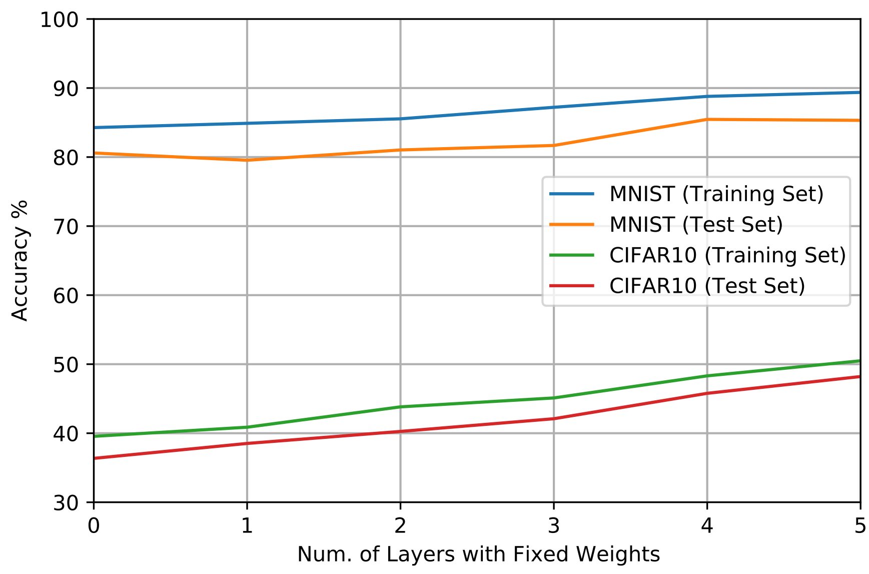

Figure 3.2: Accuracy of dataset distillation on the MNIST & CIFAR-10 datasets, trained with different numbers of layers with fixed weights.

dataset was **88.6%**, approximately 10% lower than the accuracy you would expect from a standard network. Although MNIST is a very easy dataset for a model to solve, this is explained because ENAS produced a surprisingly small network - only a few layers. For CIFAR-10, the ENAS-discovered network had an accuracy of **52.1%**, roughly 28% lower than the accuracy of the original network trained on the full CIFAR-10 dataset.

The reason for these somewhat poor results appears to be that the synthetic learning rate associated with each distilled data point turns out to be very architecture-specific, since different architectures induce different error surfaces. Note that in theory, two different networks that happen to learn at very similar rates - networks that have similar gradient magnitudes - will be able to use the same distilled learning rate schedule to get similar results.

Another problem is that ENAS by default sets the learning rate of the child networks being optimised to around 20 for stability reasons - this is several orders of magnitude higher than the synthetic learning rates in the distillation algorithm. These results mean that the original idea of using the distillation algorithm to speed up NAS will not work without significant alteration of the algorithm to make synthetic data architecture-independent, which in turn means discarding the idea of synthetic learning rates. Therefore after discovering these issues, we decided to abandon experiments with architecture search and move onto an investigation of different ways to initialise the weights in the distillation algorithm.

3.2 Weight Initialisation Experiments

The question these initialisation experiments attempted to answer was: if randomly initialised weights in the distillation algorithm produce lower accuracy, what happens if we partially train with fixed weights and partially with random? To test this, we used Algorithm 2, but only re-initialised the weights on line 4 for some of the layers of the network - the remaining layers kept the same θ_0 for each iteration. The layers with fixed weights were kept fixed during test time - only the other layers were reinitialised. The results are shown in Figure 3.2 - note that both network architectures used (LeNet and AlexNet) have 5 layers.

The results show that accuracy increases in proportion with the number of layers kept fixed throughout the optimisation process. However, using a higher number of fixed weight layers only gives accuracy increases while these layers are kept in their final configuration when training on the distilled data. Using an arbitrary model initialisation will decrease accuracy down to the level of 0 fixed weight layers. This means there is no 'free' accuracy increase for all models by keeping some layers fixed - and that no attempt to tweak the optimisation process using fixed layers will generalise well. After these experiments, we move on to looking at ideas for an entirely new dataset distillation algorithm in the next chapter.

4 A GAN Approach to Distillation

This chapter fulfils the second project objective described in the Introduction: to investigate whether generative adversarial networks can be used to generate distilled data that can train a network to a given accuracy faster than the original dataset. Since GANs are the most popular class of generative image model, they seem a natural candidate for this problem. The primary difficulty is that GANs are designed to produce images which are similar to the original dataset, so generating images which are dissimilar in one crucial way (the speed at which they train a network) is a challenging task.

The objective can be split into two sub-objectives, given the two possible ways of generating images provided by the GAN framework:

1. To attempt to modify the GAN loss function in such a way that distilled images are generated.
2. To attempt to use gradient descent in the latent space of a trained GAN to find values for $\boldsymbol{z}$ that produce distilled images.

We will investigate both of these approaches in this chapter. In particular, we will focus on ideas utilising a network separate from the GAN to evaluate a loss function. For example, we might want to minimise the difference between gradients of real and generated images with respect to this network - or to maximise the gradient magnitude of generated images when passed through this network in the hope that this will encourage the generated images to act as distilled data.

4.1 Methodology

We chose to train the GAN models on CIFAR10 [26] since it is a good baseline dataset with 10 classes, and does not take too much time to train compared with larger datasets. If a given distillation technique proves to work well on CIFAR10, further work would be necessary to ensure it generalises to more complex datasets. To evaluate how well our GAN generated data does at speeding up training, we compare the CIFAR10 test set accuracy of a network trained on the generated data with two baselines: the test accuracy when training on CIFAR10 itself, and the test accuracy when training on data generated from an unmodified GAN.

This benchmark allows us to compare two points: the final accuracy on the CIFAR10 test set, and the speed at which the generated data causes the CIFAR10 test accuracy to rise. If the generated data is indeed more efficient at training a network to classify CIFAR10, then we would expect the network to reach the same test accuracy in fewer epochs. In other words, at some number of epochs n, we would like to see a higher test accuracy when the network is trained on generated data than when it is trained on CIFAR10.

The network used for this benchmark is a standard residual network with 21 layers - the full architectural details are in Appendix C. The code is based on a public Github repository containing benchmark implementations of many networks for CIFAR10 [47], and is written in the PyTorch deep learning framework [44], with the PyTorch Lightning library as a high-level wrapper. PyTorch Lightning provides a modular interface for PyTorch that makes it easier to distribute training across multiple GPUs and log progress.

To maintain consistency, all training runs were done for 100 epochs, with a batch size of 256 and a learning rate of 0.01. However, this learning rate was cycled throughout training from an initial value of 0.0004 up to 0.01 and back down to 0.000001. This is called super-convergence [56] and has been shown empirically to train many popular image classification networks to high test accuracy in fewer iterations. All training runs are logged using the `wandb` (Weights and Biases) Python package and website, which allows for the real-time monitoring of training progress in the cloud, along with the ability to compare multiple runs and carry out hyperparameter searches.

Before training, the images were normalised to zero mean and unit standard deviation. Each image is padded with zeros to 40×40 (up from 32×32), then randomly cropped to 32×32 before being horizontally flipped with probability 0.5. These augmentations are standard and have been shown empirically to improve test accuracy on CIFAR10. We also used the `Pillow-SIMD` and `libjpeg-turbo` image processing Python libraries to parallelise image decoding and resizing on the CPU. We observed a roughly 5% decrease in training times and a similar increase in GPU utilisation when adding this performance optimisation to the code.

To minimise any other sources of variability in the results, the random seed for both Python and CUDA was set to 0 before every run. In addition, the cuDNN deterministic flag was set to True - this tells the cuDNN library of CUDA neural network primitive operations to use fully deterministic matrix operations only (some deterministic operations slow down training slightly, which is why they are not used by default). We verified that with this setup, training the network twice on the same dataset produced *exactly identical* gradients, loss values, and accuracies at each training step.

In addition to evaluating how well the generated images train a network for image classification, we also evaluate whether the images are perceptually realistic. Realism is not necessarily required in distilled images - indeed, we might expect highly distilled images which train a network quickly to not be particularly similar to the original images. However, measuring this can show how much we lose in realism when trying to make distilled images. The standard way of quantitatively measuring this is to use the Fréchet Inception Distance (FID) [17], which is calculated as follows:

$$\text{FID} = \|\mu_r - \mu_g\|^2 + \text{Tr}(\Sigma_r + \Sigma_g - 2(\Sigma_r \Sigma_g)^{\frac{1}{2}}), \tag{4.1}$$

where $X_r \sim \mathcal{N}(\mu_r, \Sigma_r)$ and $X_g \sim \mathcal{N}(\mu_g, \Sigma_g)$ are the activations of the `pool3` layer of an InceptionV3 pre-trained network [59] for real and generated images respectively. The Inception network has been trained on the ImageNet dataset [51], so its layer statistics capture some of the high-level structure of images. Lower FID scores are better and imply that the generated images are more similar to the real ones in quality and diversity. To calculate the FID, we based our code on a PyTorch implementation [54] which is designed to exactly replicate the original FID code written in Tensorflow.

The GAN models for our experiments were implemented using Mimicry [33], a small library built on PyTorch that contains common functions and standardised, easily reproducible implementations for many popular GAN models, including the

Spectral Normalisation GAN (SNGAN). This library allowed for easy modification of the GAN loss function and architecture - since the library is built with object-oriented design in a very hierarchical way, modifying a GAN is as simple as subclassing the original and overriding the `train_step` function.

The baseline GAN model we used was a conditional GAN (with a projection discriminator) using spectral normalisation. The basic network architectures are those used in the SNGAN model [41] - both G and D are convolutional residual networks. The generator uses conditional batch normalisation and the discriminator applies spectral normalisation to every layer. Full architectural details are in Appendix C, and Figure B.1 contains an example of the images generated by this GAN.

The baseline loss function is a hinge loss, inspired by Support Vector Machines:

$$\mathcal{L}_D = \max\left(0, 1 - D\left(\boldsymbol{x}, \boldsymbol{y}\right)\right) + \max\left(0, 1 + D\left(G\left(\boldsymbol{z}\right), \boldsymbol{y}\right)\right), \tag{4.2}$$

$$\mathcal{L}_G = -D\left(G\left(\boldsymbol{z}\right), \boldsymbol{y}\right), \tag{4.3}$$

where $\boldsymbol{x}$ are the real images, $\boldsymbol{y}$ are the class labels that condition the GAN, and $\boldsymbol{z} \in \mathbb{R}^{128}$ is the Gaussian random vector defining the latent space. All experiments were run with hyperparameters from [41]: $n_{dis} = 5$ for more stable training, batch size $= 64$, a learning rate of 0.0002 linearly decayed to 0 throughout training, and the Adam optimiser's [24] hyperparameters were set to $\beta_1, \beta_2 = 0, 0.9$. Although training GANs for CIFAR10 is typically done for 100,000 iterations, we trained for 50,000 iterations to save time. The random seed was again set to 0 for all runs.

4.2 Distillation Loss Function Experiments

Two main ways of modifying the GAN loss function were investigated - our hypothesis was that either of these ideas could produce distilled images:

1. Minimising the difference between the gradient of a real batch of images passed through a freshly initialised residual network and the gradient of a fake (GAN generated) batch of images passed through the same network. The idea behind this is to try and get the GAN to produce images which have similar gradients to the original images, since generating a small batch of images which have the same gradients as a larger batch of images from the original dataset seems like a promising direction for distillation.

2. Maximising the magnitude of the gradient of a fake batch of images passed through a freshly initialised or pre-trained residual network. This idea is designed to get the GAN to produce images with a larger gradient than those of the original dataset so that the residual network can be trained faster.

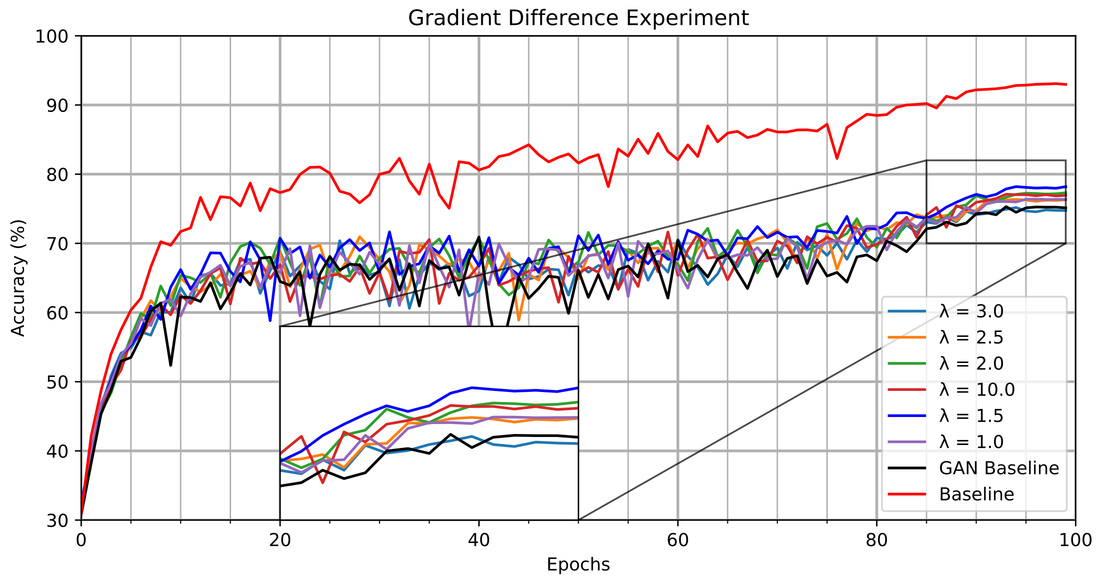

Figure 4.1: Test set accuracy on CIFAR10 using GAN generated data with the loss in Equation 4.4, compared with the original data (in red). The GAN Baseline results are from generated data with no modifications to the loss function.

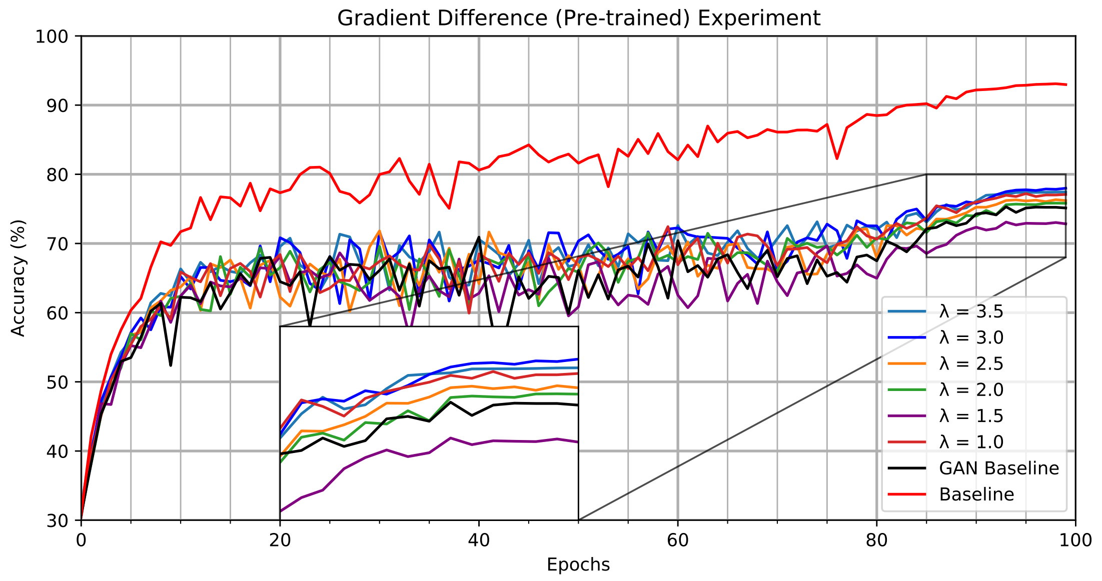

Figure 4.2: Test set accuracy on CIFAR10 using GAN generated data with the loss in Equation 4.4, using a pre-trained R.

Mathematically, the first idea can be expressed as

$$\begin{aligned}\mathcal{L}_D = \max\left(0, 1 - D\left(\boldsymbol{x}, \boldsymbol{y}\right)\right) + \max\left(0, 1 + D\left(G\left(\boldsymbol{z}\right), \boldsymbol{y}\right)\right) \\ + \lambda \|\nabla_{\boldsymbol{x}} R(\boldsymbol{x}, \boldsymbol{y}) - \nabla_{G(\boldsymbol{z}, \boldsymbol{y})} R(G(\boldsymbol{z}, \boldsymbol{y}))\|_2, \quad (4.4)\end{aligned}$$

where R is the same residual network used to evaluate the success of the distilled images (described above), except that the network is re-initialised on every training iteration to ensure that the GAN generalises across all gradients in the error surface.

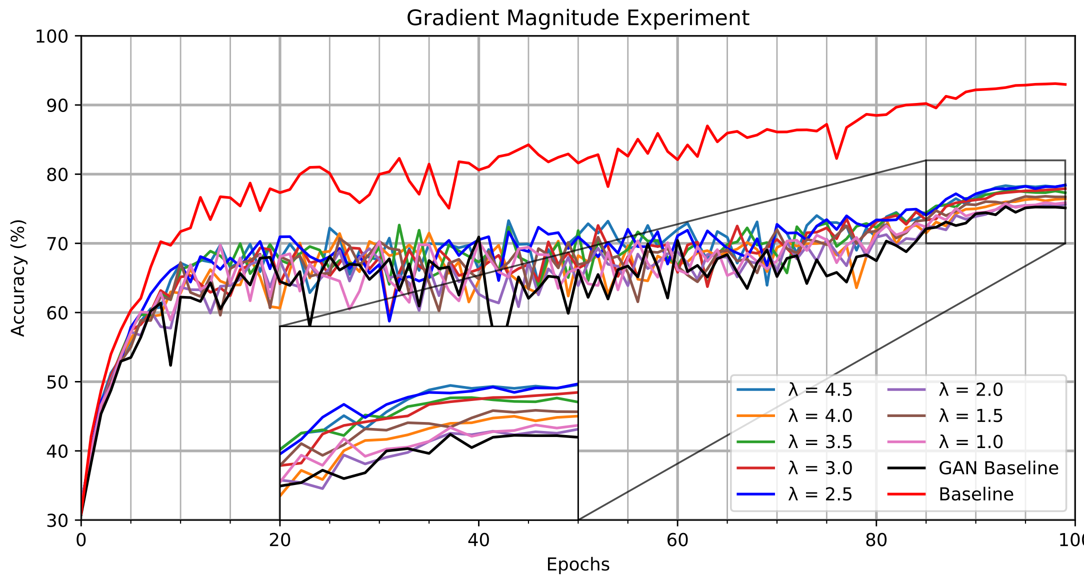

Figure 4.3: Test set accuracy on CIFAR10 using GAN generated data with the loss in Equation 4.5, compared with the original data (in red). The GAN Baseline results are from generated data with no modifications to the loss function.

During experiments we trained numerous GANs with the above loss function, carrying out a hyperparameter search that tested the following values for the gradient loss hyperparameter: $\lambda \in \{1.0, 1.5, 2.0, 2.5, 3.0, 10.0\}$. This sweep was done to try and detect whether increasing the relative importance of the gradient term in the loss function led to GAN generated data which trained the evaluation network more effectively.

The results are shown in Figure 4.1 and clearly show that although adding the gradient difference loss term increases the final test accuracy, none of the runs achieve an accuracy with GAN generated data that is above the baseline accuracy at any point, indicating that this distilled data fails to effectively train the network faster. However, we can see that $\lambda = 1.5$ produces the highest final accuracy out of all the runs with generated data, and increasing λ further causes the accuracy to decline. Therefore further hyperparameter tuning is unlikely to yield any useful results.

We can also investigate whether using a pre-trained R makes any difference, although it no longer makes sense to re-initialise R on each iteration in this test. For this test we used $\lambda \in \{1.0, 1.5, 2.0, 2.5, 3.0, 3.5\}$. The results are shown in Figure 4.2 and show that the run with $\lambda = 3.0$ produces the highest final accuracy, but no runs show accuracy above baseline at any point.

We also attempted to add the gradient difference loss term to both G and D. This resulted in no significant improvement but did produce a better final accuracy than most experiments with the loss on just the discriminator (see Table 4.1). Finally, we tried training the GAN with only the λ term in Equation 4.4 in the discriminator loss. Unsurprisingly this GAN failed to converge at all, since a paired G and D loss function is critical for ensuring the GAN does not collapse. This experiment produced an FID score of over 400, and a corresponding CIFAR10 accuracy little better than random.

The second idea listed above can be described mathematically as

$$\mathcal{L}_D = \max\left(0, 1 - D\left(\boldsymbol{x}, \boldsymbol{y}\right)\right) + \max\left(0, 1 + D\left(G\left(\boldsymbol{z}\right), \boldsymbol{y}\right)\right) \\ - \lambda \|\nabla_{G(\boldsymbol{z},\boldsymbol{y})} R(G(\boldsymbol{z}, \boldsymbol{y}))\|_2. \quad (4.5)$$

Note that this gradient magnitude term can also be added to the generator. Figure 4.3 shows the results of training 8 GAN models with the above loss function. This time we tested $\lambda \in \{1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5\}$. The results clearly show a similar result to the first experiment - the runs with $\lambda = 2.5$ and $\lambda = 4.5$ show the highest final test accuracy, but again none of the runs with generated data show accuracy above baseline at any point in training. A single run was also done with the gradient term in Equation 4.5 added to the generator as well as the discriminator - this gave results that were more or less identical to those in Figure 4.3.

In total these experiments took more than 30 separate training runs, taking between 10-16 hours each depending on the loss function. All experiments were run on NVIDIA V100 or Titan RTX GPUs, hosted by the cloud GPU provider vast.ai.

Loss Type	λ	Final Accuracy (%)	FID
Baseline	-	**92.97**	-
GAN Baseline	0	75.13	14.291
Gradient Difference (Equation 4.4)	1.0	76.36	14.276
Gradient Difference	1.5	**78.19**	13.849
Gradient Difference	2.0	77.31	13.722
Gradient Difference	2.5	76.3	13.898
Gradient Difference	3.0	74.74	13.289
Gradient Difference	10.0	76.93	14.646
Gradient Difference (pre-trained R)	1.0	77.09	12.88
Gradient Difference (pre-trained R)	1.5	72.84	15.153
Gradient Difference (pre-trained R)	2.0	75.81	14.248
Gradient Difference (pre-trained R)	2.5	76.2	14.36
Gradient Difference (pre-trained R)	3.0	**77.98**	13.635
Gradient Difference (pre-trained R)	3.5	77.44	13.369
Gradient Magnitude (Equation 4.5)	1.0	75.87	14.588
Gradient Magnitude	1.5	76.71	14.215
Gradient Magnitude	2.0	75.63	13.973
Gradient Magnitude	2.5	**78.44**	12.965
Gradient Magnitude	3.0	77.91	13.31
Gradient Magnitude	3.5	77.31	13.805
Gradient Magnitude	4.0	76.44	13.659
Gradient Magnitude	4.5	78.38	14.24
Equation 4.4 λ term on both D and G	1.0	77.74	13.555
Equation 4.4, λ term only	1.0	10.91	401.5

Table 4.1: Accuracy and FID scores across all GAN loss function experiments.

Loss Type	α	β	N	Final Accuracy (%)	FID
Baseline	-	-	-	**92.97**	-
GAN Baseline	-	-	-	75.13	14.291
Gradient Difference	1	0	10,000	12.51	185.57
Gradient Magnitude	0	1	10,000	12.29	198.87
Combined	1	1	10,000	11.25	199.03
Combined	1	0.1	1,000	23.54	80.522

Table 4.2: Accuracy and FID scores across all GAN latent space experiments.

4.3 Latent Space Experiments

There is one alternative way of generating images using a GAN - exploring the latent space of an already trained model to find images. Several works have attempted this by starting from a random initial latent vector $\boldsymbol{z}$ and using gradient descent to optimise it towards a vector that produces the desired image when passed through the generator [1], [39]. We carried out experiments attempting to find distilled images in the latent space of a GAN - we used the same base model as in the experiments described in the previous section, except we trained for 100,000 steps instead of 50,000 so that higher quality images would be generated.

The optimisation procedure followed was to randomly generate a noise vector $\boldsymbol{z}$ for a single class. Along with the corresponding class label $\boldsymbol{y}$, this was passed through the generator (note that the only conditional aspect of the generator is the conditional batch normalisation) and a loss function $\mathcal{L}_{latent}$. Then the gradients were back-propagated all the way to the latent vector $\boldsymbol{z}$ to optimise it in a direction that will minimise the loss. For almost all experiments this was repeated $N = 10,000$ times for each class, based on the empirical observation that the loss curves seemed to flatten out around 8,000 iterations.

Using similar ideas to the experiments in the previous section, we attempted several different loss functions based on different values for α and β in the following generalised function:

$$\mathcal{L}_{latent} = \alpha\|\nabla_{\boldsymbol{x}} R(\boldsymbol{x},\boldsymbol{y}) - \nabla_{G(\boldsymbol{z},\boldsymbol{y})} R(G(\boldsymbol{z},\boldsymbol{y}))\|_2 + \beta\|\nabla_{G(\boldsymbol{z},\boldsymbol{y})} R(G(\boldsymbol{z},\boldsymbol{y}))\|_2. \quad (4.6)$$

The results are shown in Table 4.2, and show that all experiments fail to generate any usable data - in fact they fail to produce any accuracy on the CIFAR10 test set significantly above random! This may be caused by the gradient descent in latent space moving into areas where $\boldsymbol{z}$ has high magnitude, creating an incoherent image since the generator was only trained on values of $\boldsymbol{z}$ close to the origin. The final experiment used $N = 1,000$ gradient descent steps and achieved slightly higher accuracy and a lower FID score, lending more support to the hypothesis that the problem is an unconstrained gradient descent procedure. More detailed potential explanations for these results will be provided in Chapter 6.

5 Project Management

This chapter contains a detailed description of the management of this project - how it was planned, and how the objectives have changed and evolved over the course of the project. The project began as an attempt to analyse the distillation algorithm [64] and apply it to Neural Architecture Search and language models.

When the research proposal was written in February 2020, the timeline planned for 3 phases of the project. Phase 1 would carry out experiments with distillation on Neural Architecture Search algorithms until the beginning of the summer term on 20th April. Phase 2 was planned to last until mid-July, containing experiments with distillation on Transformers and the work on the interim report. Finally, Phase 3, containing final experiments and writing up, was planned to start in the first week of August.

A Transformer [60] is a type of language model that has become extremely popular in the NLP field for a wide variety of tasks, due to their ability to excel at transfer learning. However, they are usually pre-trained in a very computationally expensive way using huge amounts of unlabelled data before being fine-tuned for a specific task. The original second research goal for this project was to investigate ways of training Transformers using distilled data, with the aim of speeding up pre-training of these models. The focus on Neural Architecture Search and Transformers at the beginning of the project was motivated solely by the fact that these are two of the most computationally intensive models to train - therefore these models would benefit the most if they could be trained on distilled data.

In total there have been two revisions to the project timeline so far - each has made the project timeline more refined and detailed. For the project presentation on 21st April, the timeline was reviewed and altered slightly - Phase 1 was extended to include the experiments described above, on different ways of initialising the distillation algorithm. The target of completing experiments on NAS by 20th April was met. At the time the interim report deadline on the module page was 24th June, so Phase 1 was planned to last until this date. Phase 2, consisting of experiments with distillation on Transformers, was scheduled to last from the end of Phase 1 until August 1st, before writing up would begin. This plan intentionally left a considerable buffer of time during August, so that if the experiments took longer than expected then writing up the dissertation would not be rushed.

After the examination period ended on 4th June, the timeline was again reassessed in light of the progress made so far. At this time the final interim report deadline was 17th July, so the decision was made to extend Phase 1 slightly until 31st June - to allow more time for experiments before the interim report.

As a result of extensive reading done over the course of Phase 1, it was decided that going forward Phase 2 would focus on ideas for a new dataset distillation algorithm since this research direction seemed more promising and ambitious than the ideas for distillation with Transformers, which would simply extend the work of [57] to another type of neural language model. After considering several possibilities, we decided to investigate generative algorithms that use the activations of a pre-trained convolutional network - these ideas are described fully below. The final project timeline is shown as a Gantt chart in Figure 5.1. For comparison, the initial project timeline from February is shown in the Appendix in Figure A.1.

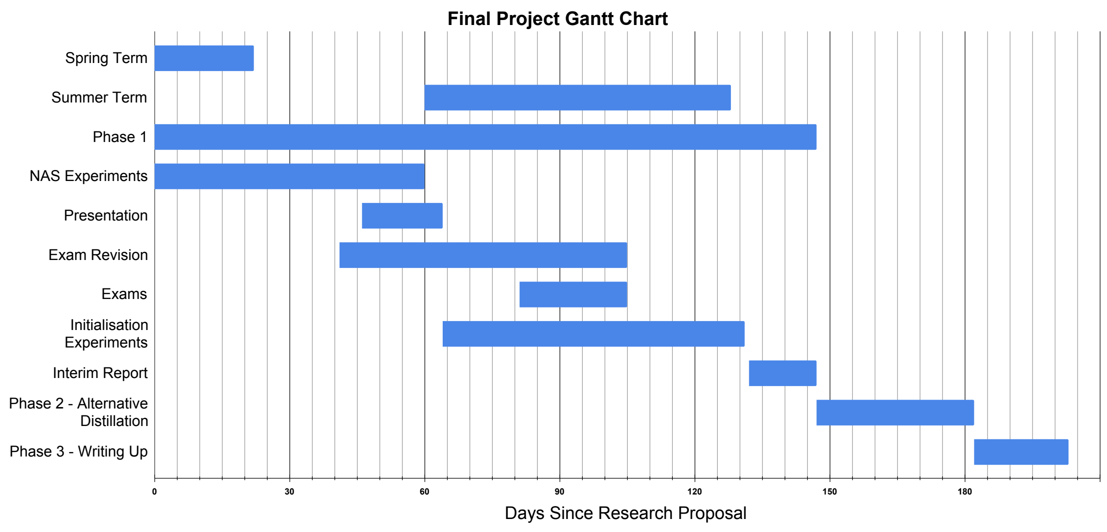

Figure 5.1: Final project plan visualised as a Gantt Chart.

5.1 Phase 2

We can divide ideas for dataset distillation into three categories, based on the information they use to construct the distilled dataset:

1. Algorithms which only rely on the data itself, and use no other information. An example would be images constructed by averaging over several images or eigenimages created by finding the eigenvectors of the covariance matrix of an image dataset.
2. Algorithms that rely on the data plus gradient information about the error surface of a machine learning model given the dataset. The original dataset distillation algorithm [64] falls into this category.
3. Algorithms that use the data, gradients, and information from the activations or weights of a neural network already trained on the data. Note that the trained weights of a network implicitly contain some gradient information about the error surface of the model, when compared with randomly initialised weights.

After initially deciding to focus on generative algorithms that use the activations of a pre-trained convolutional network (a 3rd category idea in the list above), we expanded the Phase 2 ideas slightly during the course of working on them - the final set of experiments for this project investigated ideas in the 2nd and 3rd categories. In particular, we looked at whether it is possible to train a Generative Adversarial Network (see Section 2.2) to generate images with gradients of higher magnitude than the average image in the original dataset. These generated images could then use the gradients to reach a higher test set accuracy in fewer training steps - this would effectively be distilled data.

This investigation lasted from July 17th until August 20th, in parallel with writing up the dissertation chapters covering the first half of the project. After this date, experiments were wrapped up and the final dissertation was completed before the deadline on September 10th.

6 Discussion and Conclusion

Although the main work of the project - the experiments in Chapter 4 - failed to produce any useable distilled data, there are still several key insights we can gain from the results. The primary takeaway is that based on the results in Table 4.1, modifying the GAN loss function to try and generate distilled data seems unlikely to work well. The most obvious possible explanation for this is that the GAN is trying to generate images which are as similar as possible to the training dataset. Adding extra terms to the loss function to try and generate distilled images (which are inherently dissimilar to the training set in at least one way) creates a problem of two competing training objectives, with the model unable to optimise well for either objective.

Removing the normal GAN loss function entirely and attempting to train with a function to produce distilled images only (as in the last experiment in Table 4.1) prevents the GAN from converging at all. This is likely because the GAN framework model relies on the interplay between closely related G and D loss functions that optimise the same underlying objective and bring the model closer to a Nash equilibrium.

For the latent space experiments in Section 4.3, it is more difficult to work out what went wrong. What is clear is that the latent space of our GAN contains many regions where $G(\boldsymbol{z})$ does not return a coherent image - perhaps when $\boldsymbol{z}$ has a high magnitude. Since $\boldsymbol{z}$ is drawn from a Gaussian distribution ($\boldsymbol{z} \in \mathbb{R}^{128} \sim \mathcal{N}(0, I)$), most of the values of $\boldsymbol{z}$ that are sampled for training or normal image generation will be relatively close to the origin. Therefore because the gradient descent in the latent space does not constrain itself to remain near the origin, it may quickly stray into areas where the generator has no idea what image to output. This could potentially be solved in further work by constraining the gradient descent in the latent space.

6.1 Future Work

The primary immediate direction for further work is in the experiments exploring the GAN latent space. As mentioned above, a key problem is that a vector $\boldsymbol{z}$ simply being in the latent space does not guarantee that $G(\boldsymbol{z})$ will output a coherent image. One way to solve this is to add l_2 regularisation to the latent space optimisation procedure, to force $\boldsymbol{z}$ to remain near the origin since the GAN was trained with a Gaussian prior. However, since most of the mass of a high-dimensional Gaussian distribution is near the surface of a sphere of radius $\sqrt{d}$ [39], [61], where d is the dimensionality of the latent space, we do not necessarily want to force $\boldsymbol{z}$ to remain near the origin. Motivated by this, Menon et al. [39] present a technique for exploring the latent space of a GAN trained with a Gaussian prior by performing projected gradient descent on a sphere of radius $\sqrt{d}$, centred at the origin. This is a promising direction for further experiments into generating distilled data with a GAN.

There are also more speculative ideas for distillation algorithms. It is important to bear in mind that the most common proposed application of distilled data is to use it to speed up hyperparameter searches or automated Neural Architecture Search; it seems inevitable that on a dataset of any complexity the distilled data will not reach the same final training accuracy as the original. Therefore, we want distilled data which mimics the behaviour of the original data in training despite giving lower accuracy - if changing a hyperparameter and training on distilled data improves the accuracy then this hyperparameter change should also give a higher accuracy on the original dataset.

Using this reasoning, one of the most promising ideas for distillation might be to create a smaller batch of data which gives almost identical gradients during training as the original. For example, we can imagine an algorithm which takes in a single batch of 64 images and optimises some loss to create a generated batch of 32 images with very similar gradients and training behaviour. This could be done for each batch in the original training set and repeatedly applied to reduce the size of the generated batches further.

6.2 Conclusion

Existing work on dataset distillation has revealed a promising but highly under-explored idea, with potential for a variety of applications. In this project, we first attempted to explore the application of dataset distillation as a way of accelerating the training of architecture search algorithms. However, using distilled data for this application gave poor results and showed that the distillation algorithm is inherently too architecture-dependent to use well with architecture search. We then looked at initialisation techniques for the distillation algorithm, to understand more about its behaviour. These two investigations, described in Chapter 3, constituted the first part of the project - objective 1 in the Introduction.

Since generative adversarial networks (GANs) are the most popular class of generative image model, they seem like a good candidate for the problem of generating distilled data. Therefore, in Chapter 4 we looked at both of the two possible ways of generating images using GANs: modifying the loss function to train a GAN to *generate* distilled images, and attempting to use gradient descent in the latent space of an already trained GAN to *find* distilled images.

We evaluated these ideas by training the GAN on the CIFAR10 dataset to obtain the distilled data, then using this data to train a separate image classification network, which was evaluated on the CIFAR10 test set. If the generated data is indeed more efficient at training a network to classify CIFAR10, then we would expect the network to reach the same test accuracy in fewer epochs - at some number of epochs n, we would like to see a higher test accuracy when the network is trained on generated data than when it is trained on CIFAR10.

The results showed that although changing the GAN loss function can increase the final test accuracy above that obtained with data from a normal GAN, none of the experiments achieved accuracy with GAN generated data that is above the baseline accuracy at any point. This indicates that the GAN generated data fails to effectively train the network faster. The results from the latent space experiments also failed to produce useable data but showed that the optimisation procedure quickly strayed into areas of the latent space which produced incoherent generated images. Therefore, constraining the gradient descent in the latent space is a promising direction for further work that may yield distilled data.

References

[1] R. Abdal, Y. Qin, and P. Wonka, "Image2StyleGAN: How to Embed Images Into the StyleGAN Latent Space?" In *Proceedings of the 2019 IEEE International Conference on Computer Vision (ICCV)*, 2019.

[2] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein Generative Adversarial Networks," in *International Conference on Machine Learning (ICML)*, 2017.

[3] A. Brock, J. Donahue, and K. Simonyan, "Large Scale GAN Training for High Fidelity Natural Image Synthesis," in *International Conference on Learning Representations (ICLR)*, 2019.

[4] M. Brundage, S. Avin, J. Clark, H. Toner, P. Eckersley, B. Garfinkel, A. Dafoe, P. Scharre, T. Zeitzoff, B. Filar, H. Anderson, H. Roff, G. C. Allen, J. Steinhardt, C. Flynn, S. Ó. hÉigeartaigh, S. Beard, H. Belfield, S. Farquhar, C. Lyle, R. Crootof, O. Evans, M. Page, J. Bryson, R. Yampolskiy, and D. Amodei, "The Malicious Use of Artificial Intelligence: Forecasting, Prevention, and Mitigation," 2018. arXiv: `1802.07228 [cs.AI]`.

[5] "CS231n Convolutional Neural Networks for Visual Recognition." [Online]. Available: `https://cs231n.github.io/convolutional-networks/` (visited on 09/06/2020).

[6] L. Dinh, D. Krueger, and Y. Bengio, "NICE: Non-linear Independent Components Estimation," in *International Conference on Learning Representations (ICLR) 2015, Workshop Track Proceedings*, 2015.

[7] H. Dong, W. Hsiao, L. Yang, and Y. Yang, "MuseGAN: Multi-track Sequential Generative Adversarial Networks for Symbolic Music Generation and Accompaniment," in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)*, 2018.

[8] T. Elsken, J. H. Metzen, and F. Hutter, "Neural Architecture Search: A Survey," *Journal of Machine Learning Research*, vol. 20, no. 55, pp. 1–21, 2019.

[9] C. Finn, P. Abbeel, and S. Levine, "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks," in *International Conference on Machine Learning (ICML)*, 2017.

[10] I. Goodfellow, "NIPS 2016 Tutorial: Generative Adversarial Networks," 2016. arXiv: `1701.00160 [cs.LG]`.

[11] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[12] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Nets," in *Advances in Neural Information Processing Systems 27*, 2014.

[13] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville, "Improved Training of Wasserstein GANs," in *Advances in Neural Information Processing Systems 30*, 2017.

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," in *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, 2015.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Identity Mappings in Deep Residual Networks," in *Proceedings of the 14th European Conference on Computer Vision (ECCV)*, 2016.

[17] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, "GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium," in *Advances in Neural Information Processing Systems 30*, 2017.

[18] S. Hochreiter, "The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, no. 02, pp. 107–116, 1998.

[19] S. Hochreiter and J. Schmidhuber, "Simplifying Neural Nets by Discovering Flat Minima," in *Advances in Neural Information Processing Systems 7*, 1994.

[20] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *International Conference on Machine Learning (ICML)*, 2015.

[21] T. Karras, T. Aila, S. Laine, and J. Lehtinen, "Progressive Growing of GANs for Improved Quality, Stability, and Variation," in *International Conference on Learning Representations (ICLR)*, 2018.

[22] T. Karras, S. Laine, M. Aittala, J. Hellsten, J. Lehtinen, and T. Aila, "Analyzing and Improving the Image Quality of StyleGAN," in *Proceedings of the 2020 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[23] N. Keskar, J. Nocedal, P. Tang, D. Mudigere, and M. Smelyanskiy, "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima," in *International Conference on Learning Representations (ICLR)*, 2019.

[24] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *International Conference on Learning Representations (ICLR)*, 2014.

[25] D. P. Kingma and M. Welling, "Auto-Encoding Variational Bayes," in *International Conference on Learning Representations (ICLR)*, 2014.

[26] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," University of Toronto, Tech. Rep., 2009.

[27] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25*, 2012.

[28] K. Kurach, M. Lučić, X. Zhai, M. Michalski, and S. Gelly, "A Large-Scale Study on Regularization and Normalization in GANs," in *International Conference on Machine Learning (ICML)*, 2019.

[29] A. Lapedriza, H. Pirsiavash, Z. Bylinskii, and A. Torralba, "Are all training examples equally valuable?" 2013. arXiv: `1311.6510 [cs.CV]`.

[30] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[31] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[32] Y. LeCun, C. Cortes, and C. Burges, "MNIST handwritten digit database," 1998. [Online]. Available: `http://yann.lecun.com/exdb/mnist`.

[33] K. S. Lee and C. Town, "Mimicry: Towards the Reproducibility of GAN Research," 2020. arXiv: `2005.02494 [cs.CV]`.

[34] M.-Y. Liu and O. Tuzel, "Coupled Generative Adversarial Networks," in *Advances in Neural Information Processing Systems 29*, 2016.

[35] M.-T. Luong, Q. V. Le, I. Sutskever, O. Vinyals, and L. Kaiser, "Multi-task Sequence to Sequence Learning," in *International Conference on Learning Representations (ICLR)*, 2016.

[36] A. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning Word Vectors for Sentiment Analysis," in *Proceedings of the 49th Annual Meeting*

of the Association for Computational Linguistics: Human Language Technologies, 2011.

[37] M. Marcus, G. Kim, M. A. Marcinkiewicz, R. MacIntyre, A. Bies, M. Ferguson, K. Katz, and B. Schasberger, "The Penn Treebank: Annotating Predicate Argument Structure," in *Proceedings of the Workshop on Human Language Technology*, Association for Computational Linguistics, 1994.

[38] C. de Masson d'Autume, S. Mohamed, M. Rosca, and J. Rae, "Training Language GANs from Scratch," in *Advances in Neural Information Processing Systems 32*, 2019.

[39] S. Menon, A. Damian, M. Hu, N. Ravi, and C. Rudin, "PULSE: Self-Supervised Photo Upsampling via Latent Space Exploration of Generative Models," in *Proceedings of the 2020 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[40] M. Mirza and S. Osindero, "Conditional Generative Adversarial Nets," 2014. arXiv: `1411.1784 [cs.LG]`.

[41] T. Miyato, T. Kataoka, M. Koyama, and Y. Yoshida, "Spectral Normalization for Generative Adversarial Networks," in *International Conference on Learning Representations (ICLR)*, 2018.

[42] T. Miyato and M. Koyama, "cGANs with Projection Discriminator," in *International Conference on Learning Representations (ICLR)*, 2018.

[43] "Overview of GAN Structure — Generative Adversarial Networks," Google Developers. [Online]. Available: `https://developers.google.com/machine-learning/gan/gan_structure` (visited on 08/19/2020).

[44] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, 2019.

[45] J. Pennington, R. Socher, and C. D. Manning, "GloVe: Global Vectors for Word Representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

[46] H. Pham, M. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient Neural Architecture Search via Parameters Sharing," in *International Conference on Machine Learning (ICML)*, 2018.

[47] H. Phan, "PyTorch_CIFAR10," `https://github.com/huyvnphan/PyTorch_CIFAR10`, 2020.

[48] A. Radford, L. Metz, and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," in *International Conference on Learning Representations (ICLR)*, 2016.

[49] A. S. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition," in *Conference on Computer Vision and Pattern Recognition DeepVision Workshop*, 2014.

[50] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized Evolution for Image Classifier Architecture Search," in *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-19)*, 2019.

[51] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[52] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, X. Chen, and X. Chen, “Improved Techniques for Training GANs,” in *Advances in Neural Information Processing Systems 29*, 2016, pp. 2234–2242.

[53] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, “How Does Batch Normalization Help Optimization?” In *Advances in Neural Information Processing Systems 31*, 2018.

[54] M. Seitzer, “PyTorch-FID,” `https://github.com/mseitzer/pytorch-fid`, 2020.

[55] O. Sener and S. Savarese, “Active Learning for Convolutional Neural Networks: A Core-Set Approach,” in *International Conference on Learning Representations (ICLR)*, 2018.

[56] L. N. Smith and N. Topin, “Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates,” in *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications*, 2019.

[57] I. Sucholutsky and M. Schonlau, “Soft-Label Dataset Distillation and Text Dataset Distillation,” 2019. arXiv: `1910.02551 [cs.LG]`.

[58] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, “Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning,” in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI-17)*, 2017.

[59] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the Inception Architecture for Computer Vision,” in *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[60] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is All you Need,” in *Advances in Neural Information Processing Systems 30*, 2017.

[61] R. Vershynin, “Random vectors in high dimensions,” in *High-Dimensional Probability: An Introduction with Applications in Data Science*, ser. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 2018, pp. 38–69.

[62] E. M. Voorhees, “The TREC-8 Question Answering Track Report,” in *Proceedings of TREC-8*, 1999.

[63] H. de Vries, F. Strub, J. Mary, H. Larochelle, O. Pietquin, and A. C. Courville, “Modulating early visual processing by language,” in *Advances in Neural Information Processing Systems 30*, 2017.

[64] T. Wang, J.-Y. Zhu, A. Torralba, and A. A. Efros, “Dataset Distillation,” 2018. arXiv: `1811.10959 [cs.LG]`.

[65] Z. Wang, Q. She, and T. E. Ward, “Generative Adversarial Networks in Computer Vision: A Survey and Taxonomy,” 2019. arXiv: `1906.01529 [cs.LG]`.

[66] Y. Yoshida and T. Miyato, “Spectral Norm Regularization for Improving the Generalizability of Deep Learning,” 2017. arXiv: `1705.10941 [stat.ML]`.

[67] B. Zoph and Q. V. Le, “Neural Architecture Search with Reinforcement Learning,” in *International Conference on Learning Representations (ICLR)*, 2017.

[68] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, “Learning Transferable Architectures for Scalable Image Recognition,” in *Proceedings of the 2018 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

A Gantt Charts

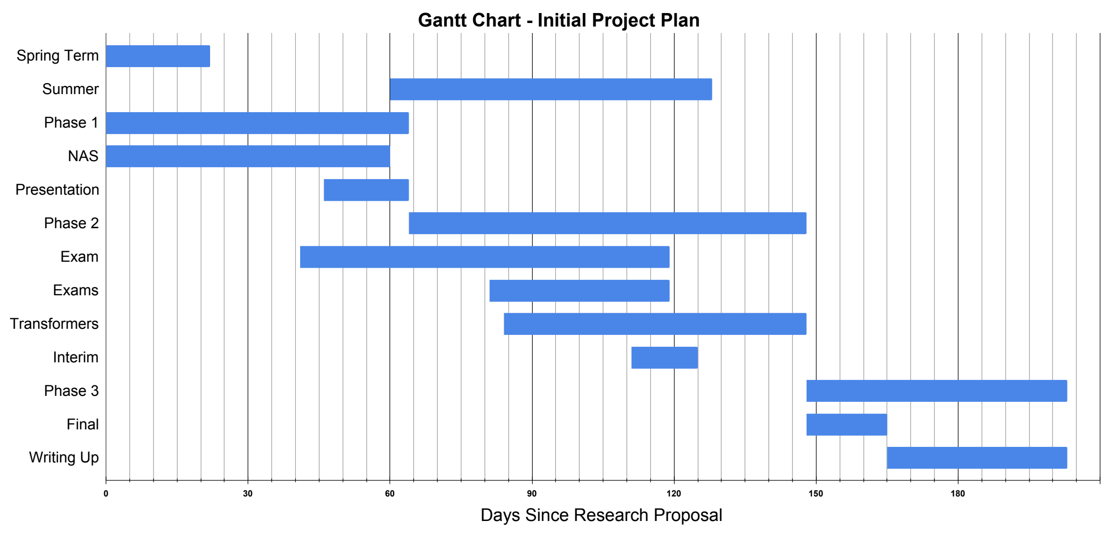

Figure A.1: Initial project plan at the time of the research proposal (20th February 2020), visualised as a Gantt Chart.

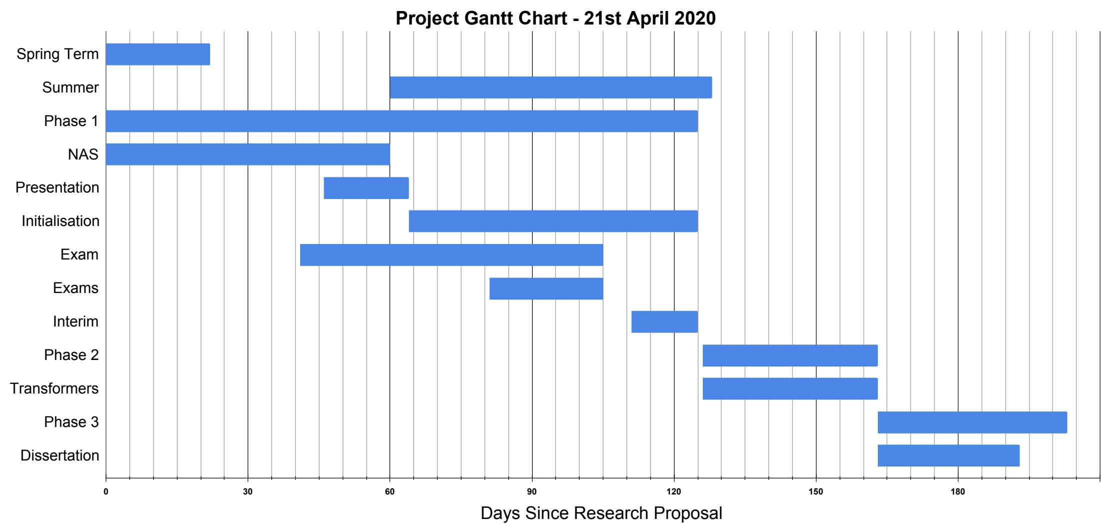

Figure A.2: Project plan at the time of the project presentation presentation (21st April 2020), visualised as a Gantt Chart.

B Additional Figures

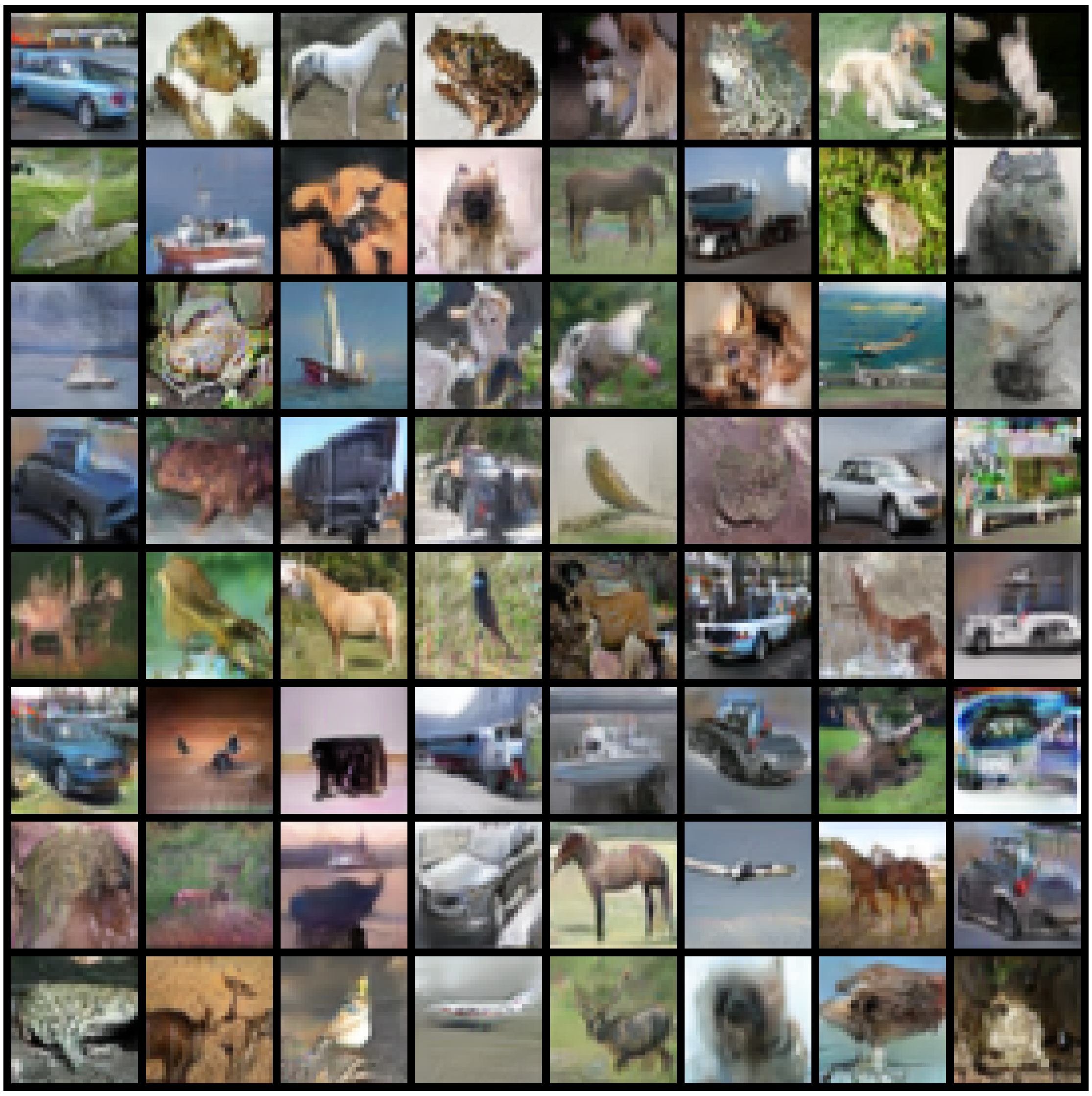

Figure B.1: Grid of 32×32 images generated by our baseline GAN model, trained for 100,000 iterations on CIFAR10.

C Network Architectures

3×3, 64, stride 1 3×3 max pooling, stride 2
3×3, 64, stride 1 3×3, 64, stride 1 Residual Connection
3×3, 64, stride 1 3×3, 64, stride 1 Residual Connection
3×3, 128, stride 2 3×3, 128, stride 1 Residual Connection
3×3, 128, stride 1 3×3, 128, stride 1 Residual Connection
3×3, 256, stride 2 3×3, 256, stride 1 Residual Connection
3×3, 256, stride 1 3×3, 256, stride 1 Residual Connection
3×3, 512, stride 2 3×3, 512, stride 1 Residual Connection
3×3, 512, stride 1 3×3, 512, stride 1 Residual Connection
Average pooling $512 \to 10$ fully connected layer

Table C.1: Network architecture of the residual network used for the experiments in Chapter 4. The syntax first describes the filter size of each conv layer, then the number of filters in the output volume, then the stride. Every conv layer is followed by batch normalisation and the ReLU activation function. The 3rd, 5th and 7th residual blocks also apply a 1×1, stride 2 conv layer *to the skip connection* only.

RGB image $x \in \mathbb{R}^{3\times32\times32}$
3×3, 128, stride 1 3×3, 128, stride 1 Average pooling, stride 2 Residual Connection
3×3, 128, stride 1 3×3, 128, stride 1 Average pooling, stride 2 Residual Connection
3×3, 128, stride 1 3×3, 128, stride 1 Residual Connection
3×3, 128, stride 1 3×3, 128, stride 1 Residual Connection
$\boldsymbol{h}_1$ = Global sum pooling $\boldsymbol{h}_2 = 128 \to 1$ fully connected layer Output $= \boldsymbol{h}_2 + (\text{Embed}(y) \times \boldsymbol{h}_1)$

Table C.2: Architecture of the GAN discriminator described in Chapter 4. A 1×1, 128, stride 1 convolutional layer is applied only to the skip connection of the residual blocks. Spectral normalisation is applied on every layer, and the activation function is ReLU. y denotes the class label.

$\boldsymbol{z} \in \mathbb{R}^{128} \sim \mathcal{N}(0, I)$ $128 \to 4096$ fully connected layer
3×3, 256, stride 1 3×3, 256, stride 1 Residual Connection
3×3, 256, stride 1 3×3, 256, stride 1 Residual Connection
3×3, 256, stride 1 3×3, 256, stride 1 Residual Connection
3×3, 3, stride 1 Tanh activation

Table C.3: Architecture of the GAN generator described in Chapter 4. A 1×1, 256, stride 1 convolutional layer is applied only to the skip connection of all the residual blocks. Conditional batch norm and the ReLU activation function are applied immediately before every conv layer.