



Pázmány Péter Catholic University  
Faculty of Information Technology and Bionics



University of Oxford  
Torr Vision Group

# Triple Generative Multi-Adversarial Networks

Botos Csaba

2018

Submitted to Conference of Scientific Students' Associations

Supervisors: PhD. Adnane Boukhayma, PhD. András Horváth,  
Prof. Philip H. S. Torr

## Abstract

Standard adversarial training involves two agents, namely a generator and a discriminator, playing a mini-max game. However, even if the players converge to an equilibrium, the generator may only recover a part of the target data distribution, in a situation commonly referred to as *mode collapse*. In this work, we present Triple-GMAN, a new approach to deal with mode collapse in generative adversarial learning. First, we use multiple discriminators each enforcing the generator to cover an additional part of the target distribution. To ensure these parts don't overlap and collapse into the same mode, we add a classifier as a third agent in the game that decides for each sample which discriminator the generator is trained against. Through experiments on toy examples and real images, we show the merits of our method and its superiority with respect to competing methods.

**Keywords:** *unsupervised learning, game theory, generative adversarial network, mode collapse*

# Contents

|                                                      |           |
|------------------------------------------------------|-----------|
| <b>Abstract</b>                                      | <b>II</b> |
| <b>1 Introduction</b>                                | <b>2</b>  |
| 1 Motivation . . . . .                               | 2         |
| 2 Outline . . . . .                                  | 4         |
| <b>2 Related work</b>                                | <b>6</b>  |
| 1 Generative models . . . . .                        | 6         |
| 2 Generative Adversarial Networks . . . . .          | 10        |
| Convergence stability issues of GANs . . . . .       | 10        |
| Similar approaches . . . . .                         | 11        |
| <b>3 Method</b>                                      | <b>13</b> |
| 1 Preliminaries . . . . .                            | 13        |
| 2 Our approach . . . . .                             | 13        |
| <b>4 Experiments</b>                                 | <b>16</b> |
| 1 Synthetic low dimensional distributions . . . . .  | 16        |
| Observations . . . . .                               | 16        |
| Implementation details . . . . .                     | 18        |
| 2 Image generation . . . . .                         | 19        |
| Stacked-MNIST . . . . .                              | 19        |
| Qualitative results on CIFAR-10 and celebA . . . . . | 19        |
| <b>5 Summary</b>                                     | <b>24</b> |
| 1 Overview . . . . .                                 | 24        |
| 2 Discussion . . . . .                               | 24        |
| 3 Future work . . . . .                              | 24        |
| <b>Acknowledgement</b>                               | <b>25</b> |
| <b>Bibliography</b>                                  | <b>28</b> |
| <b>Appendices</b>                                    | <b>30</b> |
| <b>A Neural Network Basics</b>                       | <b>31</b> |
| 1 Single Layered networks . . . . .                  | 31        |
| Defining the Perceptron . . . . .                    | 31        |
| Basics of Learning . . . . .                         | 32        |
| 2 Inference . . . . .                                | 33        |
| Fully Connected layer . . . . .                      | 33        |

---

|   |                                    |    |
|---|------------------------------------|----|
|   | Activation layer . . . . .         | 33 |
| 3 | Measuring efficiency . . . . .     | 34 |
|   | Loss . . . . .                     | 34 |
|   | Supervised learning . . . . .      | 34 |
| 4 | Adjusting parameters . . . . .     | 35 |
|   | Gradient Descent . . . . .         | 35 |
|   | Training Policies . . . . .        | 35 |
|   | Networks in practice . . . . .     | 36 |
| 5 | Differentiation . . . . .          | 37 |
|   | Numeric differentiation . . . . .  | 37 |
|   | Complexity . . . . .               | 37 |
|   | Analytic differentiation . . . . . | 38 |
|   | Fully Connected Layer . . . . .    | 40 |
|   | Backpropagation . . . . .          | 42 |
|   | Activation Layer . . . . .         | 43 |
| 6 | Additional . . . . .               | 43 |
|   | Pooling layers . . . . .           | 43 |
|   | Dropout . . . . .                  | 43 |
|   | Output . . . . .                   | 44 |

To my heart, Mesi

# 1. Introduction

*The means of obtaining as much variety as possible, but with the greatest possible order... is the means of obtaining as much perfection as possible.*

— G. W. Leibniz

## 1.1 Motivation

**Era of supervised learning** Recently supervised learning outperformed previous approaches that relied on complex sets of delicate domain specific rules, handcrafted algorithms and on limited amount of data. Nowadays professional expertise is getting replaced by big data and general learning algorithms that are capable of utilizing the available computational power. On a restricted training set they perform moderately, usually worse than their predecessors, mainly because of over-fitting and simply memorizing the data rather than extracting knowledge that can be extrapolated from onto other areas of interest.

**Scalability** is the key for these approaches: while performance of expert designed algorithms saturates rather fast, given increasing number of training data, today's methods are capable of harnessing the potential of millions of samples, learning from pure data, pushing results beyond state of the art (or even surpassing human level). Just to mention a few classic tasks in which supervised deep learning has been successful: image classification, object detection and segmentation, depth estimation and 3D reconstruction, feature tracking, face and age recognition, natural language processing, translation, scene understanding, scene description, imitation and inverse reinforcement learning, etc.

**Problem of limited supervision** Supervision means that there is always an oracle with which answers our parametrized model is optimized. Scalable models are described by a single vague formulae: the more, the better – as we add more and more training data, labeled by the oracle, a scalable model is able to improve its generalization capabilities: to provide similar results to unseen samples the oracle would give. However reaching performance beyond human level takes increasing order of magnitudes of training data to improve slightly. Also the metric we use to measure the performance becomes questionable: are we aiming specialists' level of precision? Are we trying to fit a model on the most plausible answers using multiple annotations for the same sample? In some cases labels can be obtained efficiently (e.g. user preferences on a site), however in general, collecting and cleaning a training dataset involves **expensive human labour**. The trend of recent directions is rather contradictory. The tasks we ought to solve with machine learning should relieve us from the burdens of dull, unchallenging, repetitive tasks, while annotating training samples in our precious and limited time actually opposes this goal. *Our algorithms are proven to be scalable, but are we?*

**The next step** What if we were rather focusing on understanding how the presence of an unlabeled large dataset could be used to our advantage. It costs almost nothing to monitor and record data-flows and let the machines reveal the hidden pattern of what we are no longer able to handle due to the *curse of dimensionality*. Let the algorithms build a model that is able to represent the results. A **generative model**, is able to extrapolate from visible, *real* data points and show expectations for further observations. Generative models can be incentivized to cluster the space of *synthetic* data by providing them latent *labels*, which we call afterwards conditional sampling. If we have a functioning model, we can use its own labeled data points for training supervised algorithms in the inner loop. The success of the resulting ensemble mainly depends on how well the input space is covered (diversity) and the how much samples resemble to real ones (quality).

**Challenges** The first non-trivial task we face is finding a way to quantize or measure somehow these two key properties of generative models: **diversity** and sample **quality**. In case of supervised learning approaches the evaluation of a model at a given time step, in a given state is rather straightforward, since there is a way to derive equivalents of precision and recall in almost every situation (e.g. involving a disjunct set of previously unseen samples). Even if we lack notion to measure performance of the algorithm at a given iteration, we can still track its progress by monitoring the loss function over the course of training.

On the contrary, it is still an open question of recent research whether there are any generic norm that could evaluate a single approach alone or a model agnostic metric that could be used to compare two algorithms. In many application the definition of sample quality is ill-posed, since *realistic* can refer to multiple aspects of a given data point. For example imagine that the task is super resolution of low light photos, and our algorithm captures a unique distribution of stars when enlarging the image but completely ignores reflections on nearby objects. Application-wise the above algorithm could worth a fortune, or just a weakly performing solution. On the other hand, diversity could be approached from more principled ways, less depending on the applications' criteria. However diversity in itself is not enough, i.e. taking random samples from a uniform distribution quite covers the objective of diversity, still far away from producing realistic samples with high probability. In this work we will analyze our results using quantitative methods, but to provide a more insightful summary we will also include visual inspection and qualitative evaluation of the samples drawn from the modeled distribution.

**Our contributions** We propose an algorithm, the Triple Generative Multi-Adversarial Networks, providing a lower bound on diversity of synthetic data (for visual explanation see Figure 1.1), while harnessing the capabilities of Generative Adversarial Networks for high quality sample generation. First, we use multiple discriminators each enforcing the generator to cover an additional part of the target distribution. To ensure these parts don't overlap and collapse into the same mode, we add a classifier as a third agent in the game that decides for each sample which discriminator the generator is trained against. Through experiments on toy examples and real images, we show the merits of our method and its superiority with respect to competing methods.

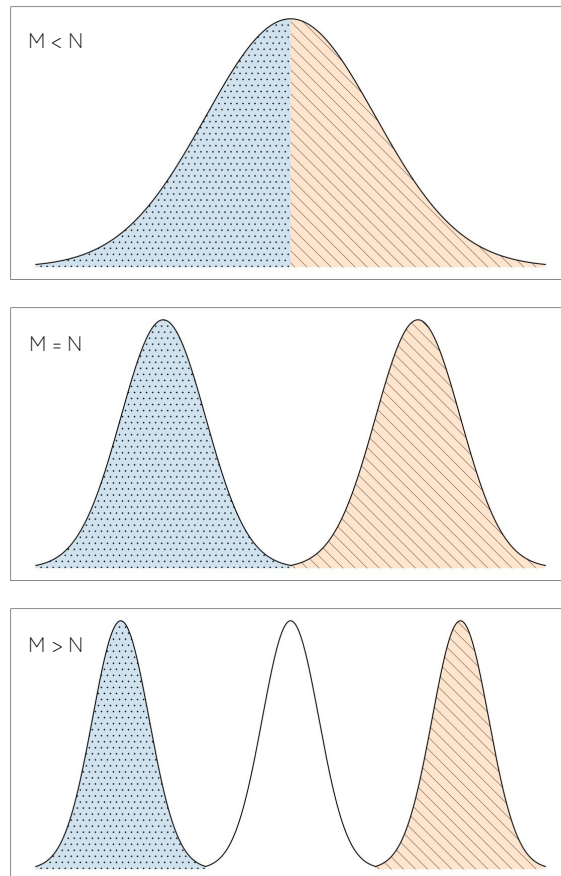


Figure 1.1: Illustration of the expected behaviour of our generator (Triple-GMAN) using two discriminators ( $N = 2$ ), in case of a  $M = 1$  Uni-modal (top),  $M = 2$  bi-modal (middle) and  $M = 3$  tri-modal (bottom) target distribution. At convergence, even in the worst case where both discriminators lead to mode collapse, the classifier ensures their respective associated generated modes (in orange and blue) do not overlap, therefore providing a lower bound on overall mode coverage.

## 1.2 Outline

Our main goal is to provide a concise manuscript of our work, well illustrated and elaborate on each aspect of the implied mechanism.

**Related works** At the time of writing there are 3 classic schools of generative modeling: 1) *auto-encoders* 2) *auto-regressive models* 3) *adversarial networks*. Even though every branch has its own domain of challenges we consider comparing the core ideas on a high level and show high resolution samples from each model to motivate our choice. Next, we are going to discuss the core concepts of standard adversarial training, and the inherent stability issues along with recent advances tackling these problems.

**Method** By noticing that using a single discriminator often leads to the generator covering only a part of the data, we bring more discriminators to the game so that each incentivises the generator to cover an additional part of the real data. To ensure that these various target data parts do not collapse into the same mode, we introduce a third player in the game, a classifier that decides which discriminator should the generator be trained against for each inputted real or fake sample. We find that this strategy, illustrated in Figure 3.1, yields better coverage of the real data at convergence and concordantly improves

the stability of the training.

**Experiments** We showcase our method on demonstrative toy problems and show that it outperforms competing methods vis-a-vis solving for mode collapse. We also show our method’s ability to generate good quality and diversified samples covering various modes on datasets of real images.

**Discussion** During our experiments we inspected how the *Multi-Adversarial* game could be extended for arbitrary number of adversaries while keeping the computational capacity in mind. We briefly discuss our findings with strong conjectures that could motive future work.

**Neural Network tutorial** We also kept in mind those who are new to the field of generative models and parametric function approximations and provided a short Neural Network tutorial in the Appendix A to introduce the frequently used terms, the basic steps and our notation.

## 2. Related work

### 2.1 Generative models

Formally the objective of generative models is to recover a probability distribution  $p_g$  from a fixed set of real data samples  $x^1, x^2, \dots, x^n$  that represents the true data distribution  $p_r(x)$ . Notice that in real life tasks usually each sample is represented by a real valued multidimensional vector  $x \in [0, 1]^d$ , for example a 128x128 coloured images can be represented in  $[0, 1]^{49152}$  space. That means  $p_r(x)$  is a 49152 dimensional joint probability distribution which renders sampling numerically intractable even if  $p_r$  was provided. Therefore different algorithms have emerged to provide solution for this problem.

**Common property** Even though the major classes of generative models approach the issue from completely different ways, one thing is common between them: they use significantly less parameters to represent the distribution than the complexity required to memorize every real sample  $x$ . It induces learning a lossy-compression of data which, roughly speaking, will be the main objective for generative models. In order to succeed in recovering the true data distribution  $p_r$  (and to minimize their respective objective function), the models must learn which part of the data is irrelevant, and which patterns could be reused when drawing samples from the model distribution  $p_g$ . In the era of Deep Learning generative models are implemented by neural networks (for which we provide a general overview in Appendix A), are a special type of deterministic parametric functions by default. The parameter set of a generative model is referred as  $\theta_g$  and in order to use them as non-deterministic functions during training we often condition them on a random variable  $z \sim p_{\text{random}}$ . We illustrated the workflow in Figure 2.1 that covers the general goal and mechanism of generative models. Numerous variants exist that approach the same problem from different aspects, however the three main categories are the following: *Auto-encoders*, *Auto-regressive models*, *Generative Adversarial Networks*.

**Auto-encoders** One approach is to implement two neural networks, explicitly trained to learn a compression. We start by designing the first network, the Encoder to project the input sample  $x$  to a latent space that has smaller dimension than the input space. Using this latent representation  $z$  the second network, the Decoder is tasked to learn the inverse of the Encoder, i.e. a mapping from the latent space to the input space, outputting  $\hat{x}$ . We train the Encoder and the Decoder jointly to minimize the predefined distance between  $d(x, \hat{x})$ . E.g. for image generation this distance is the photo-metric loss:  $L(x, \hat{x}) = |x - \hat{x}|$ . For increasing the robustness and the diversity of the model [27] introduce Variational Auto-encoders (VAEs), which redesigns the Encoder to output mean ( $\mu$ ) and standard deviance ( $\sigma$ ) values which will be used to parametrize a Normal distribution  $\mathcal{N}(\mu, \sigma)$ . Latent variable  $z$  is sampled from  $\mathcal{N}(\mu, \sigma)$  and fed to the Decoder as in the original formulation. The latent space in VAEs encode rich and diverse features that can be visualized via interpolations between different points  $z_1$  and  $z_2$ . This approach lets us train VAEs in the framework of probabilistic graphical models.  $\beta$ -VAEs [19] incentivizes

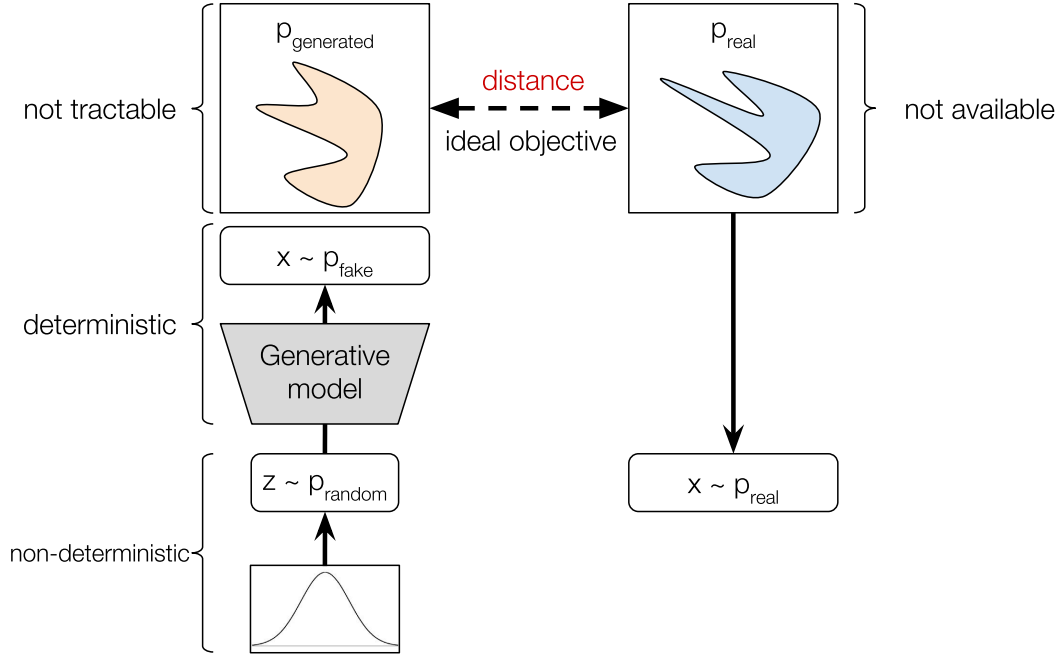


Figure 2.1: General overview of non-deterministic generative algorithms. Our overall objective is to reduce the distance between the real distribution  $p_r$  and the generated distribution  $p_g$ . The first issue is that the choice of distance function determines the success of the training. Secondly, in real life applications  $p_r$  is rarely provided, rather represented by a list of real samples. Furthermore, given a model joint probability distribution of multiple dimensions is numerically difficult to handle and computationally expensive to draw samples from. An efficient way to implement the model distribution is to use a neural network to learn a mapping from a lower dimensional, random distribution (here a 1D distribution) into the space of the target distribution (here a 2D distribution). To treat neural networks as non-deterministic parametric function approximations, during training a random variable  $z$  is fed to the network.  $z$  is sampled from a previously fixed latent distribution (here represented with a Gaussian).

the Encoder to disentangle latent space features by penalizing each dimension of  $\mathcal{N}(\mu, \sigma)$  that differs from the standard Gaussian distribution  $\mathcal{N}(0, 1)$ . We illustrate training Auto-encoders in Figure 2.2 and their limitations in image generation tasks 2.3.

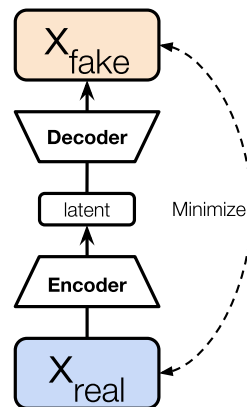


Figure 2.2: A schematic workflow of training autoencoders. The Encoder projects the input sample  $x_{real}$  (blue) to a latent space that has smaller dimension than the input space. The second network, the Decoder learns the inverse of the Encoder, i.e. a mapping from the latent space to the input space, outputting  $x_{fake}$  (orange). We train the Encoder and the Decoder jointly to minimize the predefined distance between  $d(x, \hat{x})$ .

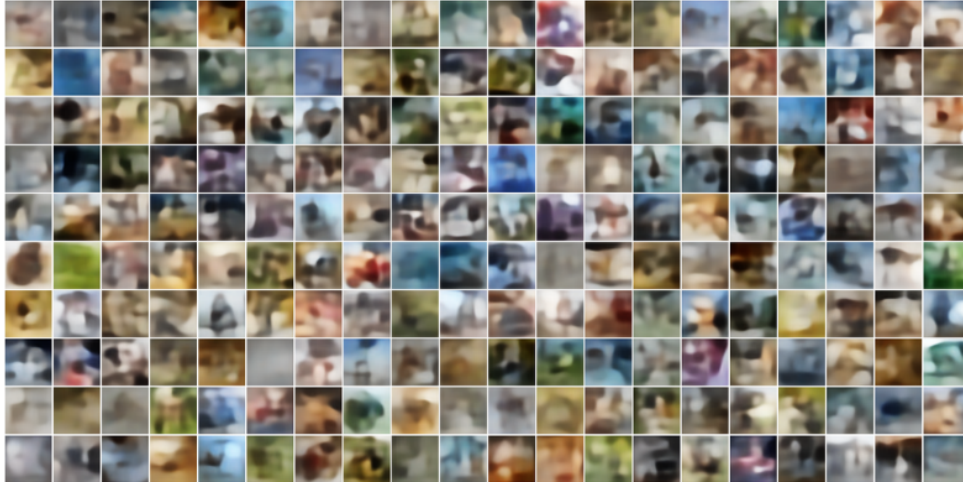


Figure 2.3: Even though Variational Auto-Encoders [27] (VAEs) are not producing mode collapse, the quality of the produced samples improve logarithmically over time (Source: <https://blog.openai.com/generative-models/>). Usually it takes several application specific modification to the standard algorithm to produce fine detailed textures.

**Auto-regressive models** Another approach to tackle high dimensional probability density estimation is using auto-regressive (AR) models, which decompose the complexity of the task by the following formulation:

$$p(\mathbf{x}) = \prod_{i=2}^T p(x_i | x_1, \dots, x_{i-1}) p(x_1)$$

, where we denote the dimension of the real data points with  $T$ . Given this decomposition AR model only needs to predict the value of the  $t^{th}$  dimension at step  $t$  using the previous values. Formally this is equivalent to recovering the conditional probability  $\prod_{i=2}^t p(x_i | x_1, \dots, x_{i-1}) p(x_1)$ . In the task of image generation AR models are well known for their ability to learn local, high frequency patterns, however on a larger scale (e.g. colored images of 64x64) the approach is sensitive to the choice of the serialization. In PixelCNN[37] multiple serialization techniques (row major, column major, diagonal, bi-directional, etc.) are investigated. Scalability is another critic issue of AR models, since generating a sample requires the model to first generate the sequence, element-by-element and we can only de-serialize the image when every pixel is evaluated. We illustrate the workflow of training AR models in Figure 2.4 and show the limits in image generation tasks in 2.5.

**Measuring efficiency** In low dimensional tasks we can make a histogram with arbitrary small bin sizes to approximate both  $p_r$  and  $p_g$ . This is useful, because using numerical approximations we are able to evaluate the following classic distances to measure the performance of an individual algorithm, using the definitions of elementary distances from [2].

Let  $\mathcal{X}$  be a compact metric set and  $\Sigma$  the set of all Borel subsets of  $\mathcal{X}$ . Let  $\text{Prob}(\mathcal{X})$  denote the space of probability measures defined on  $\mathcal{X}$ . [2] defines the distances between two the distributions  $\mathbb{P}_r, \mathbb{P}_g \in \text{Prob}(\mathcal{X})$  as the following:

- The *Total Variation* (TV) distance

$$\delta(\mathbb{P}_r, \mathbb{P}_g) = \sup_{A \in \Sigma} |\mathbb{P}_r(A) - \mathbb{P}_g(A)|.$$

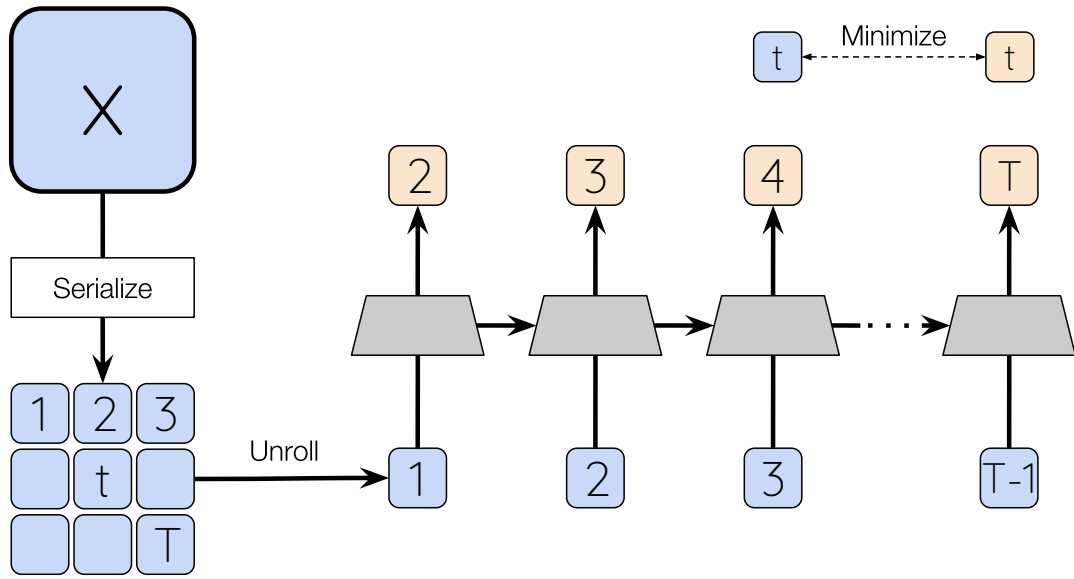


Figure 2.4: A schematic workflow of training auto-regressive (AR) models. The multidimensional sample is serialized into a sequence. We assume that the AR model (grey block) has an inner state which is initialized randomly or with constant values in the first step, and is continuously altered by each step. At each step  $t$  the current element (blue numbered box) of the sequence is fed into the AR model along with the state of the AR from the previous step, next the current state will be updated as a function of the former two input, finally at each time step the next element of the sequence (orange numbered box) will be sampled from the joint probability distribution  $\prod_{i=2}^t p(x_i|x_1, \dots, x_{i-1})p(x_1)$ . This property lets the autoregressive model preserve information from different regions of the sequency

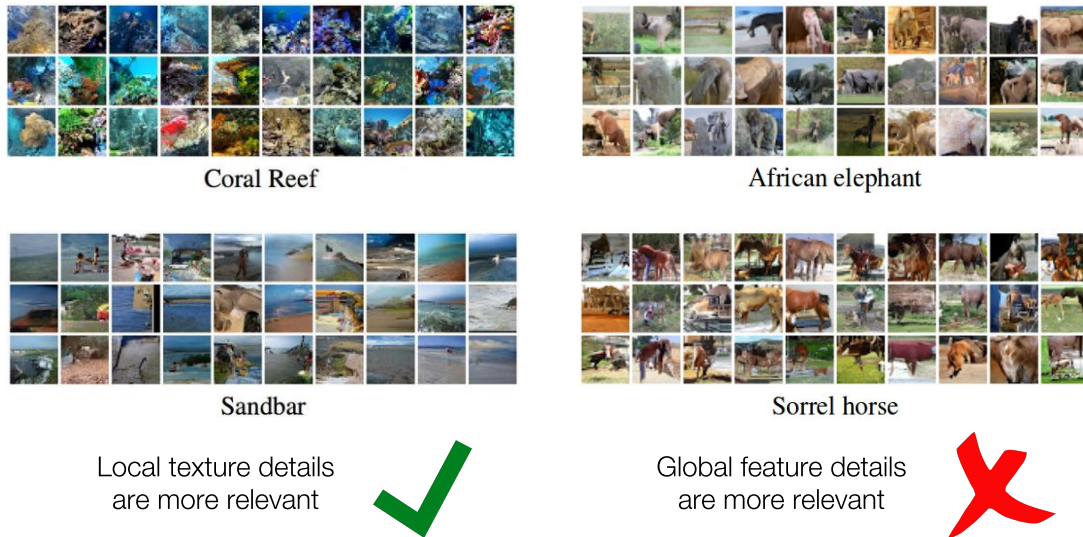


Figure 2.5: Auto-regressive models like PixelCNN [37] are able to generate high quality local visual features, which can be convincing in cases (left column) where repeated patterns occurs in the true data distribution as well, e.g. Coral Reefs and Sandbars (left-top and left-bottom respectively). On the other hand generating samples where global features (e.g. geometric shapes) span the entire sequence of the serialized sample (right column) seem to be more difficult for auto-regressive models.

- The *Kullback-Leibler* (KL) divergence

$$KL(\mathbb{P}_r || \mathbb{P}_g) = \int \log \left( \frac{P_r(x)}{P_g(x)} \right) P_r(x) d\mu(x),$$

where  $P$  is a density with respect to  $\mu$ , used in the definition  $\mathbb{P} = \int_A P(x) d\mu$ . As mentioned in [15, 2] KL is famously asymmetric and we should pay attention to the order in which we evaluate the divergence.

- The *Jensen-Shannon* (JS) divergence

$$JS(\mathbb{P}_r, \mathbb{P}_g) = KL(\mathbb{P}_r, \mathbb{P}_m) + KL(\mathbb{P}_g, \mathbb{P}_m),$$

where  $\mathbb{P}_m = (\mathbb{P}_r + \mathbb{P}_g)/2$ .

- The *Earth-Mover* (EM) distance

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

where  $\Pi(\mathbb{P}_r, \mathbb{P}_g)$  denotes the set of every joint distributions  $\gamma(x, y)$  whose marginals are respectively  $\mathbb{P}_r$  and  $\mathbb{P}_g$ .

- The *Chi-square* ( $\chi^2$ ) distance

$$\chi^2(\mathbb{P}_r, \mathbb{P}_g) = \sum_i \frac{(x_i - y_i)^2}{x_i + y_i}$$

where  $x_i, y_i$  are the frequency values of the corresponding histogram at the  $i^{th}$  bin. We have to note, that this distance is extremely sensitive to the binning size.

Considering findings in [13, 1, 2] throughout our experiments we will use  $KL$  divergence and  $\chi^2$  distance to validate our results.

## 2.2 Generative Adversarial Networks

Generative Adversarial Networks [16] (GANs) consist in training a deep generative model through a minimax game involving a competing generator and discriminator from a game theory perspective. The discriminator is tasked with learning the difference between real and fake samples, whereas the generator strives to maximize the mistakes of the discriminator. The information flow is illustrated in Figure 2.6. At convergence, the generator approximates the real data distribution and allows sampling from it. The generated samples, traditionally images, are shown to be sharper than models trained using pixel-wise losses. Consequently, GANs have demonstrated impressive results in various domains such as image generation [17], video generation [46], super-resolution [29], semi-supervised learning [9] and domain adaptation [50].

### Convergence stability issues of GANs

GANs are trained with the objective of reaching a Nash-equilibrium [32], which refers to the state where neither the discriminator nor the generator can further enhance their utilities unilaterally. Besides this training being prone to instability, reaching the equilibrium can still lead to mode collapse [14, 1, 6, 7, 41], that is when the generator only covers the real data distribution partially and hence produces a

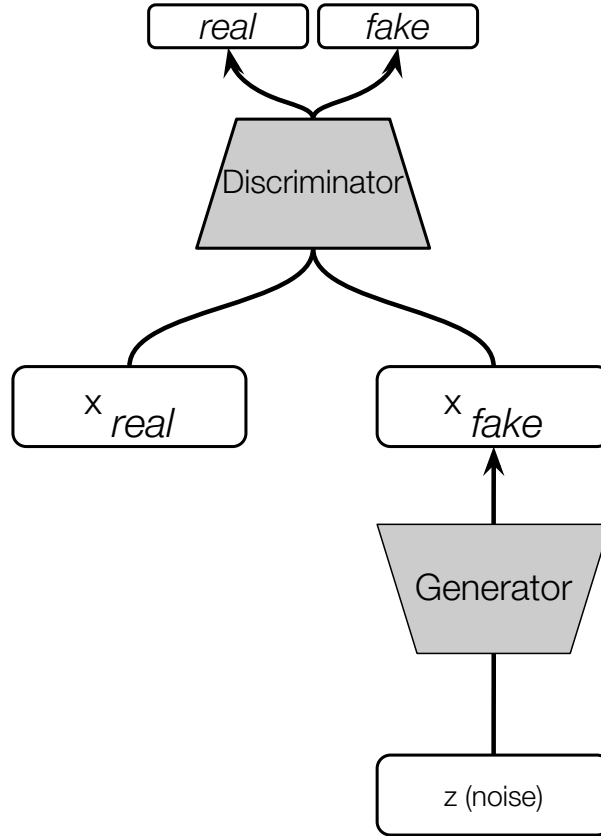


Figure 2.6: Information flow graph of standard Generative Adversarial Networks (GANs). First we sample  $z$  from the latent noise distribution. The generator takes  $z$  as input and outputs a synthetic sample  $x_{fake}$ . The discriminator takes either samples from the true data distribution  $x_{real}$  or synthetic samples  $x_{fake}$  and predicts the source of the sample. The discriminator is trained to tell apart real samples from fake ones, while the generator is trained to maximize the loss of the discriminator.

limited diversity of samples. To address the latter problem, the literature explores two main approaches: Improving GAN learning to reach a better optimum [1, 34, 41, 2, 17, 5], or explicitly urging GANs to produce various modes by design [7, 13, 11, 6, 31]. We hereby follow the latter strategy and propose a new way of dealing with GAN mode collapse.

By noticing that using a single discriminator often leads to the generator covering only a part of the data, we bring more discriminators to the game so that each incentivises the generator to cover an additional part of the real data. To ensure that these various target data parts do not collapse into the same mode, we introduce a third player in the game, a classifier that decides which discriminator should the generator be trained against for each inputted real or fake sample. We find that this strategy, illustrated in Figure 3.1, yields better coverage of the real data at convergence and concordantly improves the stability of the training.

We showcase our method on demonstrative toy problems and show that it outperforms competing methods vis-a-vis solving for mode collapse. We also show our method’s ability to generate good quality and diversified samples covering various modes on datasets of real images.

## Similar approaches

There is a rich literature on improving training stability and increasing sample diversity for GANs. We only focus in the following on a selection of works that relate closely to ours. [3] introduces theoreti-

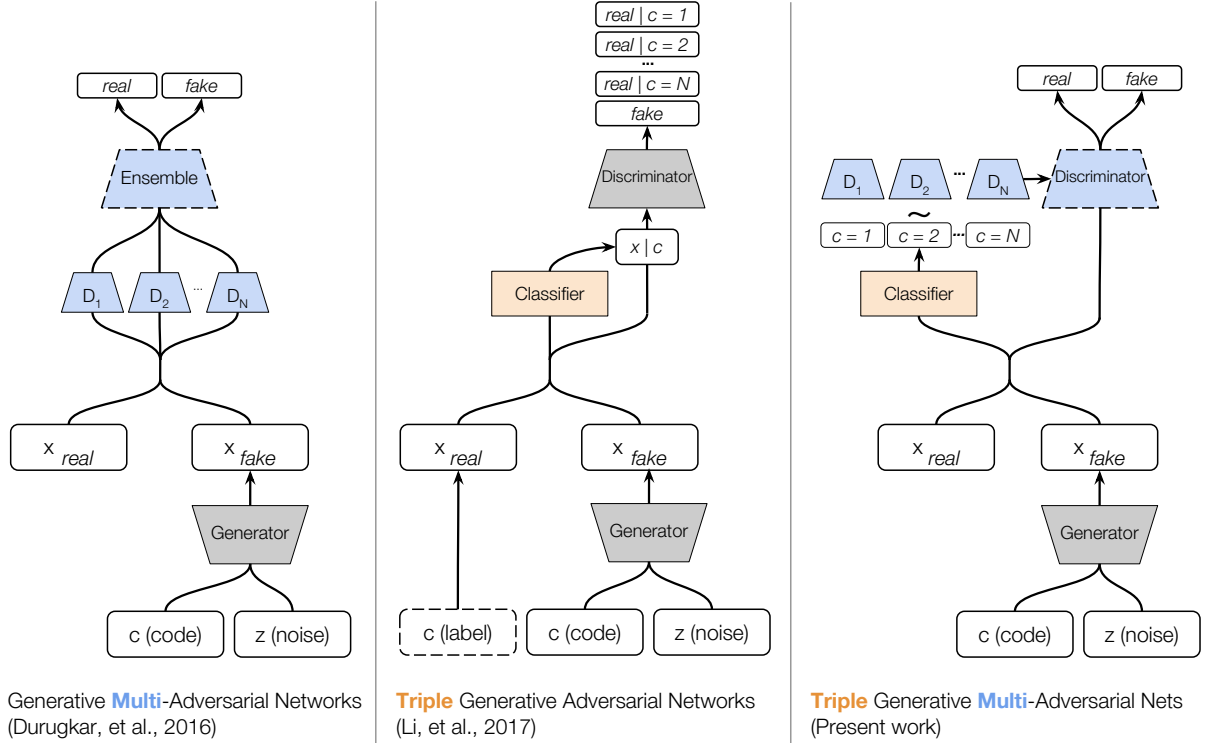


Figure 2.7: Our method is using multiple discriminators, similarly as in Generative Multi-Adversarial Networks [11], however instead of training the generator by a mixture of discriminators, we train  $D$ s separately on samples of corresponding classes. The classification task is carried out by a third, classifier network  $Q$ .  $Q$  is trained similarly to the classifier in Triple Generative Adversarial Networks [30], with the difference that our method is not relying on ground truth labels. In [30] they allow missing labels, therefore their method is *semi*-supervised, while our method can be trained both supervised and unsupervised. For further information on Triple-GMAN see Section 3, for detailed visual description see Figure 3.1.

cal formulation stating the importance of multiple generators and discriminators in order to completely model the data distribution. Similar to this work, GMAN [11] proposes to use multiple discriminators. They explore 3 settings where the generator can either be trained against the best discriminator, the averaged discriminators, or the weighted averaged discriminators. We improve on this strategy by adding a classifier as a third component to the minimax game, with the task of predicting the optimal discriminator for each input sample in training, and we compare our results to GMAN [11] in the experiments Section (4). Like our proposed method, Triple-GAN [30] incorporates a classifier in the adversarial training. This work focuses more on disentangling the variation modes of the data: The generator and the classifier characterize the conditional distribution of the labelled images, while the discriminator focuses on identifying fake image-label pairs.

Several works propose to use multiple generators [4, 12, 31] and MADGAN [13] improves them by explicitly enforcing the generators to produce diverse modes. This is achieved by requiring the discriminator to identify the generator that generated the fake samples along recognizing fake samples from reals. We compare to this method as-well in our experiments Section (4).

## 3. Method

### 3.1 Preliminaries

**Generative Adversarial Networks** Generative adversarial networks refer to a two-player zero-sum game, where the players are implemented by neural networks and are optimized in opposition to each other. Different tasks are assigned to the players: Firstly, the generator  $G$  takes an input noise  $z \sim p(z)$  and returns a sample  $x_{fake} = G(z)$  that yields a model distribution *fake*. The discriminator  $D$  takes both real samples  $x_{real}$  from the training set that represents the *real* distribution and generated samples from *fake*, and outputs a conditional probability distribution  $P(S | x)$  over the source of the sample. In practice  $D$  is a binary classifier that ideally outputs 1 if the sample is real and 0 if the sample is fake. Formally  $D$  is maximizing the log-likelihood of the true source:

$$L_S = E[\log P(S = 1 | x_{real})] + E[\log P(S = 0 | x_{fake})] \quad (3.1)$$

**Conditional generation** We can condition the modeled distribution by reformulating  $G$  to take a code vector  $c$  as an additional input to produce fake sample  $x_c = G(z, c)$ , as it is done in e.g. in InfoGAN[7] variants. Defining a conditional probability distribution  $P(C | x)$  we obtain a second objective function, the log-likelihood of the code vector.

$$L_C = E[\log P(C = c | x_c)] \quad (3.2)$$

We note that in other frameworks  $L_S$  and  $L_C$  are optimized concurrently by either  $G$  or  $D$  or both. Furthermore, in supervised [36] and semi-supervised [8] settings, and in addition to  $c$ , the ground truth labels  $y$  could be also provided along with  $x_{real}$ , thus extending the parameters of  $G$  (i.e.  $G(z, y, c)$ ).

### 3.2 Our approach

Our framework consists of three main components: A conditional generator  $G$ , a classifier  $Q$  and a set of independent discriminators  $\{D_i\}$ . We use categorical code vectors  $c \in \{1, 2, \dots, N\}$ , where  $N = |\{D_i\}|$ . As illustrated in Figure 3.1, the flow of the graph starts with generating a fake sample  $x = G(z, c)$  and feeding it to the classifier to get the categorical probability distribution:

$$P(C | x) = Q(x).$$

---

**Algorithm 1** Triple-GMAN training algorithm. One may draw new samples before each update to improve optimization stability, for brevity we excluded these steps.

---

- 1: **for** number of training iterations **do**
- 2:   Sample minibatch of  $m$  noise samples  $\{z^{(1)}, \dots, z^{(m)}\}$  from noise prior  $p_g(z)$ .
- 3:   Sample minibatch of  $m$  code samples  $\{c^{(1)}, \dots, c^{(m)}\}$  from code prior  $p_g(c)$ .
- 4:   Sample minibatch of  $m$  examples  $\{x^{(1)}, \dots, x^{(m)}\}$  from data generating distribution  $p_{\text{data}}(x)$ .
- 5:   Update  $Q$  by ascending its stochastic gradient:

$$\nabla_{\theta_q} \frac{1}{m} \sum_{i=1}^m -c \log Q \left( G \left( z^{(i)}, c^{(i)} \right) \right)$$

- 6:   Decide for every input which of the  $N$  discriminator to use by sampling from likelihood  $Q$ .

$$\sigma(i) \sim Q \left( x^{(i)} \right) \quad \hat{\sigma}(i) \sim Q \left( G \left( z^{(i)}, c^{(i)} \right) \right).$$

- 7:   Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[ \log D_{\sigma(i)} \left( x^{(i)} \right) + \log \left( 1 - D_{\hat{\sigma}(i)} \left( G \left( z^{(i)}, c^{(i)} \right) \right) \right) \right].$$

- 8:   Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log \left( 1 - D_{\hat{\sigma}(i)} \left( G \left( z^{(i)}, c^{(i)} \right) \right) \right).$$


---

By sampling  $\hat{c}$  from  $Q(x)$ , i.e.  $\hat{c} \sim Q(x)$ , we define here  $D := D_{\hat{c}}$  as the discriminator that will receive the sample  $x$ . We are now able to evaluate  $L_S$  by using the defined binary distribution  $D(x)$ , and the conditional objective function  $L_C$  by using cross-entropy. Next, we train  $G$  by minimizing  $L_D$ ,  $D$  by maximizing  $L_D$ , and  $Q$  by minimizing  $L_C$ . For the implementation of the training steps see Algorithm 1.

We note that even though we decide which discriminator to use by sampling from  $Q(x)$ , the resulting computation graph is completely differentiable with respect to  $x$ , therefore the gradients for the parameters of both the generator  $\theta_g$  and discriminator  $\theta_d$  can be obtained similarly to the standard setting:

$$\begin{aligned} \nabla_{\theta_g} L_S &= \nabla_{\theta_g} \log \left( 1 - D \left( G(z, c) \right) \right) \\ \nabla_{\theta_d} L_S &= \nabla_{\theta_d} \left[ \log \left( D(x_{\text{real}}) \right) + \log \left( 1 - D \left( G(z, c) \right) \right) \right]. \end{aligned}$$

Once  $D$  is selected using  $Q(x)$ , training requires no further modifications to the standard GAN optimization algorithm. Furthermore, different variants that aims to improve GAN learning to reach a better optimum [1, 33], like changing the objective function to approximate Wasserstein distance [2], stabilization using gradient penalties [41, 17, 5, 32] is completely compatible with our design.

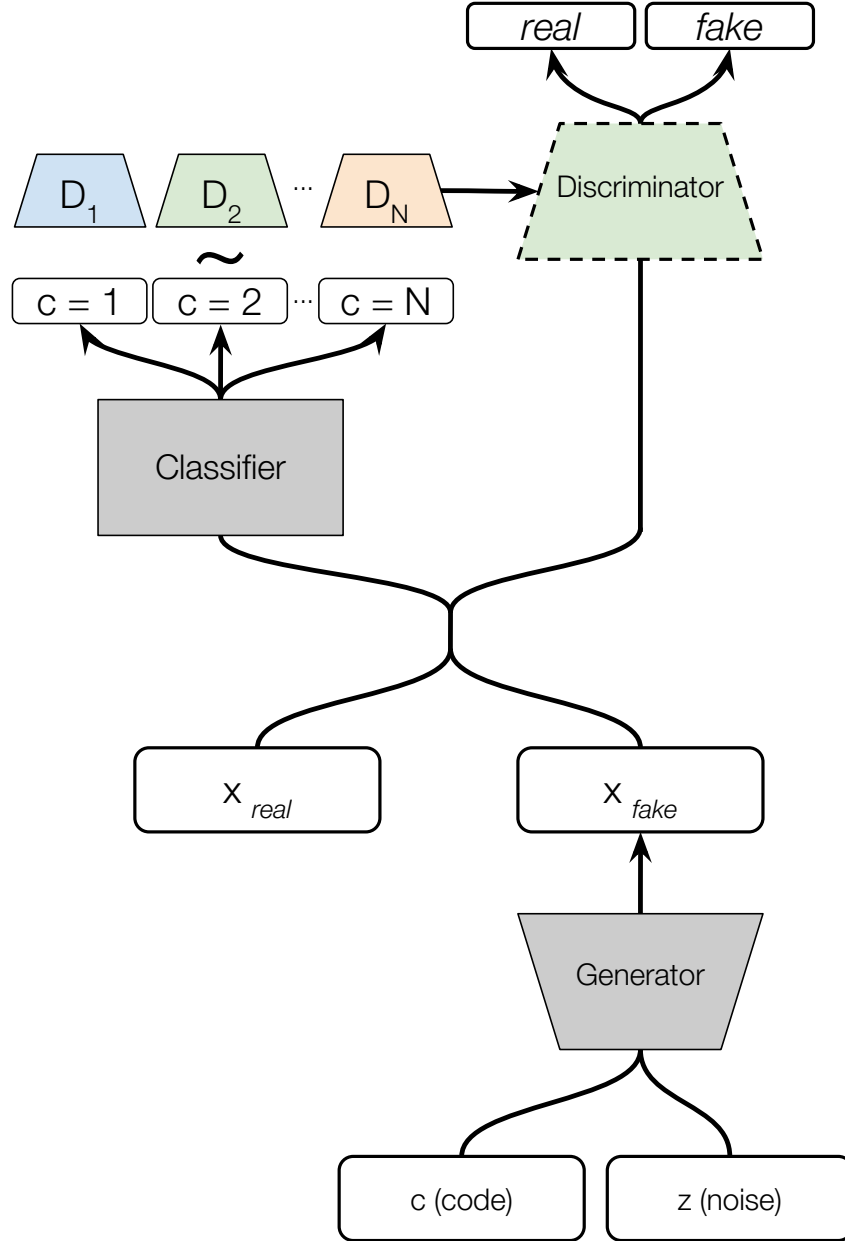


Figure 3.1: Triple-GMAN, our proposed framework. Here  $\sim$  denotes a sampling operation from the categorical probability distribution  $Q(x)$ . Using the resulting category index  $\hat{c} \sim Q(x)$  we select the corresponding discriminator  $D_{i=c}$  and connect it to the computation graph. From the perspective of  $x_{real}$  and  $x_{fake}$  the respective computation graph is differentiable, and can be trained identically to the standard GAN [15] algorithm.

## 4. Experiments

We demonstrate our performance on a diverse set of tasks with increasing complexity involving probability density estimation and image generation. To illustrate the functioning of our method, we set up two low-dimensional experiments (Section 1) using Gaussian Mixture Models as the target probability density function. For the 1D Gaussian Mixture case, we compare our robustness against other approaches by reproducing the experiment setting detailed in [13] and we outperform all competing methods. For the 2D Gaussian Mixture case, we compare our robustness against GMAN [11] and show that we outperform this latter.

For both the 1D and 2D case, we show how our training dynamics change according to the number of discriminators and show how our approach, unlike GMAN [11], is able to utilize the capacity of multiple discriminators. We finally demonstrate our results on commonly investigated datasets: MNIST, CIFAR10 and CelebA in Section 2.

### 4.1 Synthetic low dimensional distributions

In our approach the role of the  $Q$  network is to partition the real and synthetic datapoints into different clusters, and each discriminator is consequently only trained on a separate cluster. In order to fully understand how this helps the training, we experimented with two mixture of Gaussian variants: a 1D GMM with 5 modes, as used in [13], and a classic 2D circular GMM with 8 modes on the unit circle.

#### Observations

**Comparing against other GAN variants** In Table 4.1, we show that our method outperforms other GAN architectures on 1D task by a large margin in terms of Chi-square distance and KL-Divergence. We believe that the success is due to the  $Q$  network’s capability to learn to partition the underlying distribution easily. We also show in Table 4.3, that our method achieves better performance in terms of both KL-Divergence and Chi-square than GMAN [11]. It was shown in [48], that using a softmax classifier, trained with cross-entropy often results in radial clusters by nature. Therefore it could be argued that our success on the 2D task is merely achieved by a fortunate choice of the  $Q$  network’s loss function and the architecture rather than the actual implied mechanism. However, it is straightforward to see that our success in the 1D task contradicts this claim. In this task, the modes of the input space lie on the same line and do not form a radial pattern, still the  $Q$  network is able to assign different discriminators to each separate clusters of the true data distribution, to distribute them in the most efficient way.

**Benchmarking multiple discriminators** We study how game dynamics change with regards to the number of discriminators used by both GMAN [11] and our method. The clustering mechanism is illustrated on 1D task in the bottom row of Figure 4.1. We see in this experiment that classes of

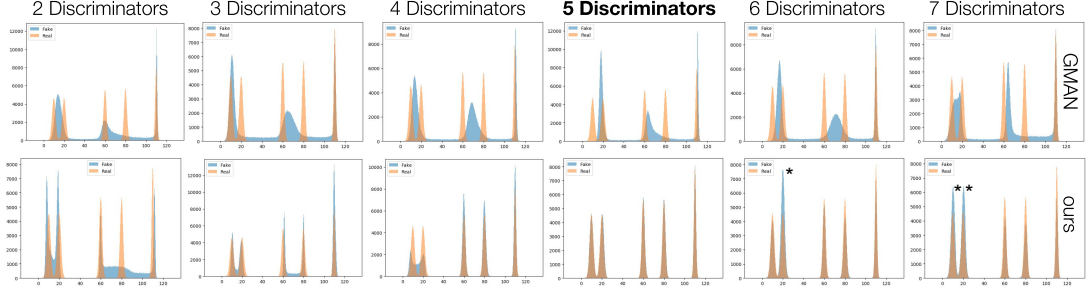


Figure 4.1: To study the behaviour of multi-discriminator settings with different number  $N$  of discriminators we trained GMAN [11] and our approach on a 1D data set with 5 modes. While both GMAN’s (top) and our (bottom) results improved by adding discriminators until  $N >$  the number of modes, we show how our lower bound on mode collapse takes effect: the  $Q$  network learns how to classify samples and reduced the complexity of the target distribution. To justify this we point at the case when  $N = 5$ : the model perform the best since each class has to capture only a single mode. In the case of  $N = 6$  and  $N = 7$  our algorithm (marked with \* and \*\*) decreased in performance only due to oversampling the two given classes.

the generated samples are first attracted towards larger clusters of the real data. By adding more discriminators, the quality of the reconstructed modes is refined. The refinement process starts first with the easiest separation, between the  $2^{nd}$  and the  $3^{rd}$  peaks, after that the  $4^{th}$  and  $5^{th}$  modes are distinguished, and so on. We see in Table 4.2 that increasing the number of discriminators improves the performance of both GMAN and our model up to a certain point where  $N$  (number of discriminators) matches the number of modes in the data. After this optimal point, increasing  $N$  yields a decreasing performance, because captured modes are oversampled. On Figure 4.1 we have marked examples of oversampling in the last two columns with \* symbols. It is interesting to note that when the same experiment was carried in 16 for the MADGAN, which is using multiple generators, their performance peaked at  $N_{Generators} = 4$  unlike GMAN and ours which logically both peaked at  $N_{Discriminator} = 5$ . This shows a difficulty of tuning the hyper-parameter  $N_{Generators}$  in [13] for different applications.

**Clustering the input space** In 2D, for both variants, GMAN[11] and Triple-GMAN we experiment with  $M = N = 8$  (where  $N$  is the number of discriminators,  $M$  is the number of modes) for quantitative results, listed in Table 4.3 and  $M = N = 3$  setting for qualitative results, illustrated in Figure 4.2. In all of the runs, our method was able to capture, and classify all modes of the true distribution correctly, while GMAN[11] failed on  $N = M = 3$  setting for every run, however performed well for  $N = M = 8$ . In Figure 4.2 we show a circular 2D GMM with 3 modes on the unit circle using Results show that the  $Q$  network is indirectly able (without optimizing  $G$  with  $\nabla_{\theta_g} L_C$ ) to control the conditioned samples  $G(z, c)$  by routing them to different discriminators. This figure illustrates that we utilize the capabilities of multiple  $D$ s indeed as we intended: different discriminators will begin to specialize on different modes therefore providing different gradients for the corresponding synthetic class as well. Although being trained with the generated code vectors only, and not sharing any weights with either of the adversarials, our  $Q$  network achieves fine partitioning of the original distribution. Different from other unsupervised or semi-supervised approaches, in our method the parameters of the generator  $\theta_g$  are *not* trained to minimize the classifier  $Q$ ’s loss  $L_C$ . We suggest that our approach succeeds because each discriminator is fed different samples from the beginning.  $Q$  is initialized to assign each real sample to every discriminator with equal probability, but given that the generator samples different points for every code vector  $Q$  quickly learns the different modes that the samples from  $G_i D_i$  pair are attracted towards (where  $G_i$  is referring to the conditional distribution modeled by  $G(z, c)$ ,  $c = i$  and  $D_c, c = i$ ). Given that the updated  $Q$  is already providing different subsets of the input space to different discriminators, they will provide

different gradients for each corresponding code vector. Therefore  $G$  learns to separate the conditional distributions  $p_c$  from each other.

## Implementation details

First, we reproduced the 1D setting in [13] with 5 modes at [10, 20, 60, 80, 110] and standard deviations [3, 3, 2, 2, 1] respectively and we compare to the numbers reported in that paper. Second, we compared our method directly to GMAN [11] qualitatively in Figure 4.2 and quantitatively in Table 4.3 using a circular 2D GMM distribution with 3 and 8 modes respectively, around the unit circle. To illustrate the advantage of our approach over GMAN [11] we plotted the gradient field to visualize the benefit of using multiple discriminators. The gradient field of this setup can be seen in Figure 4.2. In each case, to get quantitative results, we estimated the probability density distribution using a histogram with 1400 bins of the *real* and *fake* probability distribution and computed the Chi-square and KL-divergence between the two histograms.

**Comparing against other GAN variants** When comparing against other GAN Variants, we run the 1D experiments using a fixed set of 200,000 samples from the target distribution and sample 65,536 elements from each model. Since our method is directly designed to separate different modes to provide a lower bound for mode collapse, we outperform all the other methods as show in Table 4.1. In our case, we sample the code vectors for the generator from a categorical distribution with uniform probability. For the best results, we use 5 discriminators in both GMAN [11] and our setting, run each training 3 times and select the best score throughout the trainings.

**Benchmarking multiple discriminators** In 1D for better non-parametric probability density estimation, we increased the size of the training set from 65,536 to 1,000,000 samples as it was done in [13]. For more reliable results on the implied mechanism of both approaches, we run the training 20 times for each algorithm with number of discriminators  $N = 2, ..8$ , totaling 320 training. As in the previous experiment, we chose the best results from each run.

**Clustering the input space** In 2D, for both variants we experiment with  $M = N = 8$  (where  $N$  is the number of discriminators,  $M$  is the number of Gaussians we used in the mixture) for quantitative results, listed in Table 4.3 and  $M = N = 3$  setting for qualitative results, illustrated in Figure 4.2. For each experiment we use a fixed set of 1,000,000 samples and take 5 run per each algorithm, then report the best run. We took effort to make sure that the comparison was fair, and used the same set of parameters as it was done in the 1D experiments.

**Error analysis** From Figure 4.2 we perceive that GMAN fails to utilize multiple discriminators to their full potential. We argue that even in the best performing case, where the number of discriminators is equal to the number of modes  $N = M$ , the success is due to a rather a single or few discriminators rather than the effective ensemble of them.

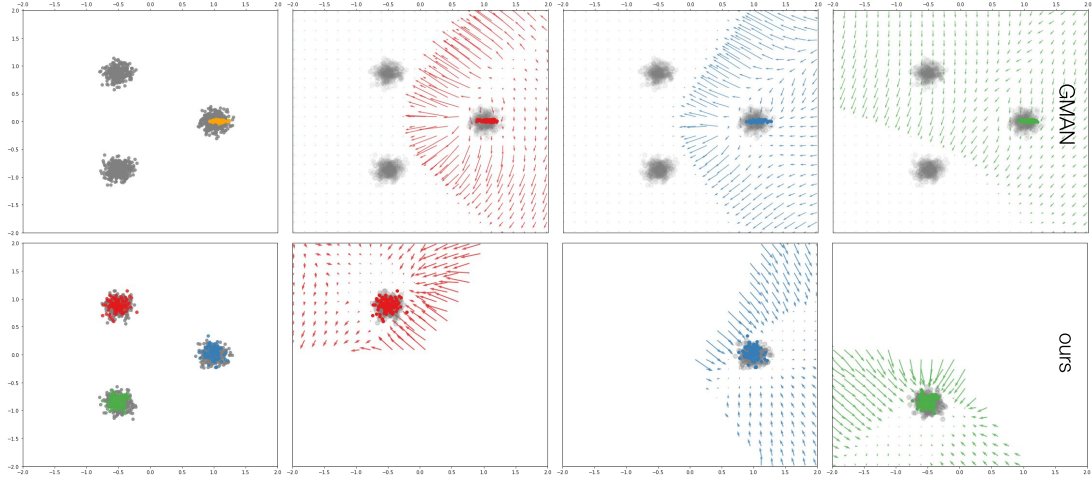


Figure 4.2: A circular 2D GMM with 3 modes on the unit circle with standard deviations of 0.01 and training GMANS and our method with  $N = 3$  setting. In the first column we show the generated distribution for each method in colours (orange for GMAN, a single color, since GMAN uses no conditioning on their samples, and red, green and blue for each  $c = 1, 2, 3$  respectively. ) In column 2, 3, 4 we plotted the gradient field for each three discriminator in either settings. In our setting those samples which are not assigned to a given discriminator gets zero gradient by definition. We see how  $Q$  network indirectly pushes different modes generated by  $G$  apart, more importantly the gradient field has the highest magnitude in the direction which separates the modes, while the same phenomenon is not present in the settings of GMAN.

## 4.2 Image generation

After the thorough investigation on low dimensional tasks, we validate our method on real image generation tasks as well.

### Stacked-MNIST

We have also investigated how well our approach can reconstruct the real distribution of the data using the Stacked-MNIST dataset [43]. This dataset contains three channel color images, containing a randomly selected sample from the MNIST dataset on each channel. This results in ten possible nodes on each channel so the number of all the possible modes in the dataset is  $10^3$ . It was shown in [13] that various architectures recovered only a small portion of these modes. A qualitative image depicting the recovered modes using the traditional DCGAN architecture and our approach can be seen on Fig. 4.3. We have also measured the Kullback-Leibler divergence between the real distribution and the two generated distributions. KL was 5.26 between the real distribution and the distribution generated by DCGAN and 0.13 using our architecture with ten discriminators.

### Qualitative results on CIFAR-10 and celebA

To show capabilities of our method, we trained the multi discriminator setting on a lower and a higher complexity image generation task, CIFAR-10 and CelebA respectively. For comparison we show best results from GMAN [11] on both tasks, according to their report in Figure 4.6.

**CIFAR-10** While learning the distribution of 32x32 colored images may sound an easy task, the main challenge is to learn geometric structures from low resolution and reproduce them in various colors,

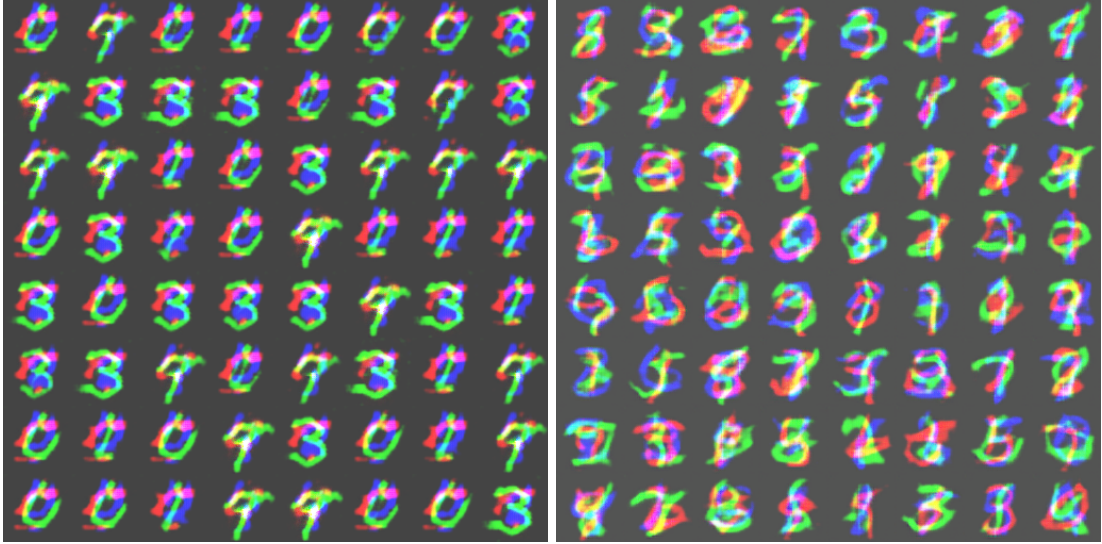


Figure 4.3: Sample images from stacked mnist. The samples on the left image were generated using the traditional architecture of DCGAN (Implementation details can be found at: <https://github.com/carpedm20/DCGAN-tensorflow>), the samples on the right image were generated using 10 discriminators with the Triple-GMAN.

backgrounds, angles etc. We illustrate in Figure 4.4 that our method is capable of capturing class invariant features such as the hull and masts of ships in different angles depicted in various weather conditions; horses with different hair colours, in various environments, either galloping, eating or standing still; cars from different angles, featuring well recognizable parts like wheels and windshields, and carroseries of diverse shapes.

**CelebA** We show how our method performs on larger scale of images such as 128x128 by training a residual network for 100k iterations on the celebA dataset. In this dataset various modes exist like lighting, pose, gender, hair style, clothing, facial expressions which is challenging to capture for generative models. On Figure 4.5 we demonstrate that our approach is capable of recovering the mentioned visual features.

**Implementation Details** For both the generator and discriminator we use ResNet-architectures [18], with 18 layers each in the CIFAR-10 experiments, and 26 layers each in the CelebA experiments. As it was done in [32] we multiply the output of the ResNet blocks with 0.1, use 256-dimensional unit Gaussian distribution. For categorical conditional image generation we use an embedding network that projects category indices to 256 dimensional label vector, normalized to the unit sphere. In the case of conditional image generation the  $Q$  network is trained on a different code vector, that is not constrained to learn the original class labels. We embed the code vector similarly to the ground truth labels in this setting. We use Leaky-RELU nonlinearities everywhere, without BatchNorm.

Following the considerations in [32] for optimizing parameters of  $Q$ ,  $D$ ,  $G$  we use the RMSProp with  $\alpha = 0.99$ ,  $\epsilon = 10^{-8}$ , and initial learning rate of  $10^{-3}$ . We use a batch size of 64, and train the algorithm for 700,000 and 400,000 iterations for CIFAR-10 and CelebA tasks respectively. Similarly to works that provided state of the art results on image generation tasks [26, 32] for visualizing the generator’s progress we use an exponential moving average of the parameters of  $G$  with decay 0.999.

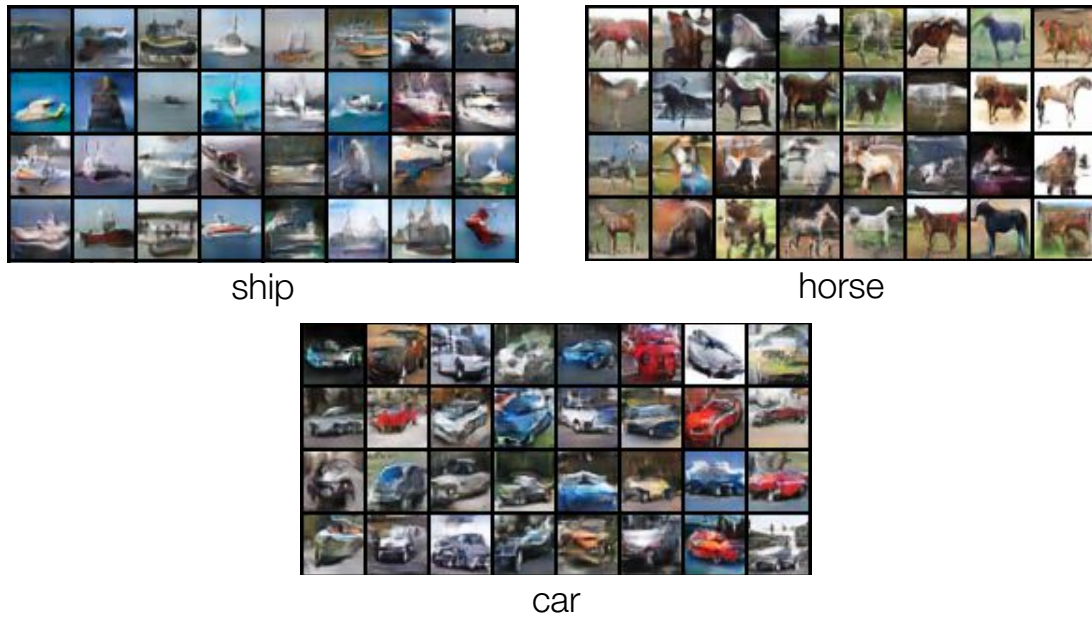


Figure 4.4: 32x32 CIFAR-10 samples drawn from an  $N = 5$  training of 700k iterations where ground truth labels were provided. We call the attention to the high geometrical precision on the depicted classes which causes other variants to mode collapse.

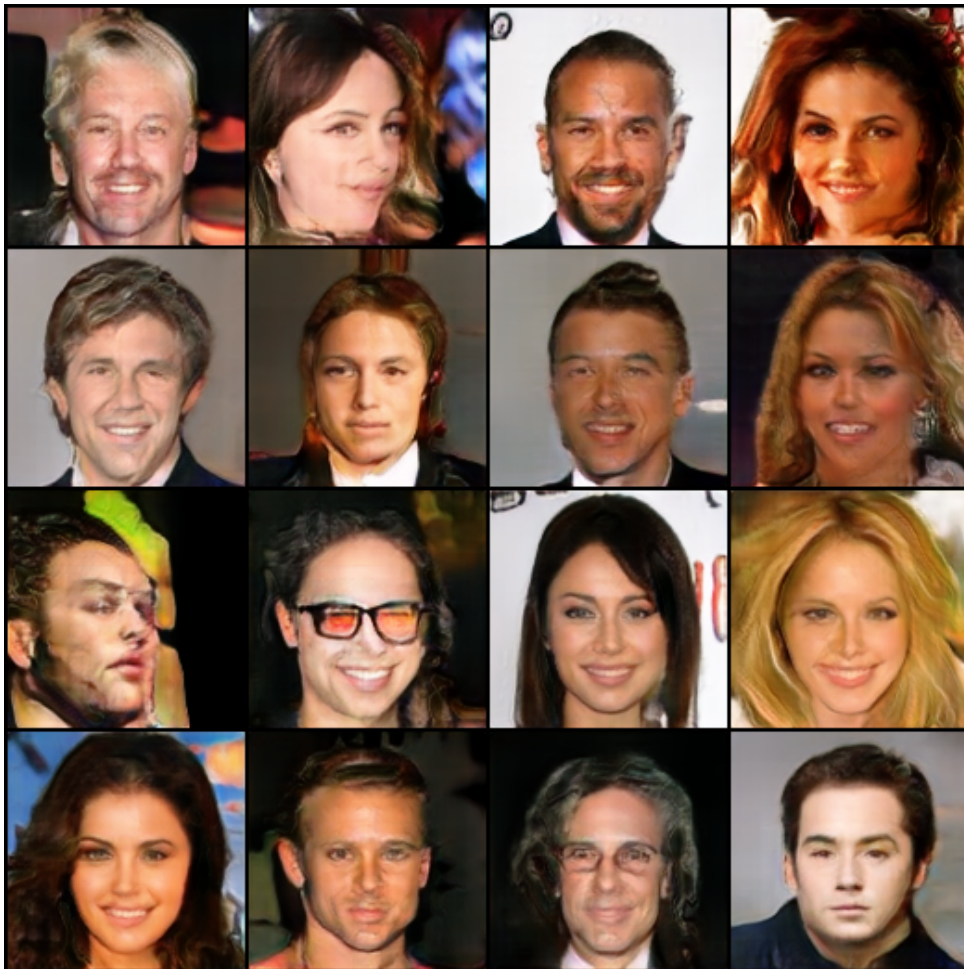


Figure 4.5: Random samples from our network trained at scale of 128x128 images on unaligned CelebA set for 100k iterations. As it can be seen, some faces appear in diverse poses with different background, and rarely occurring accessories such as orange sunglasses are learned by the model.

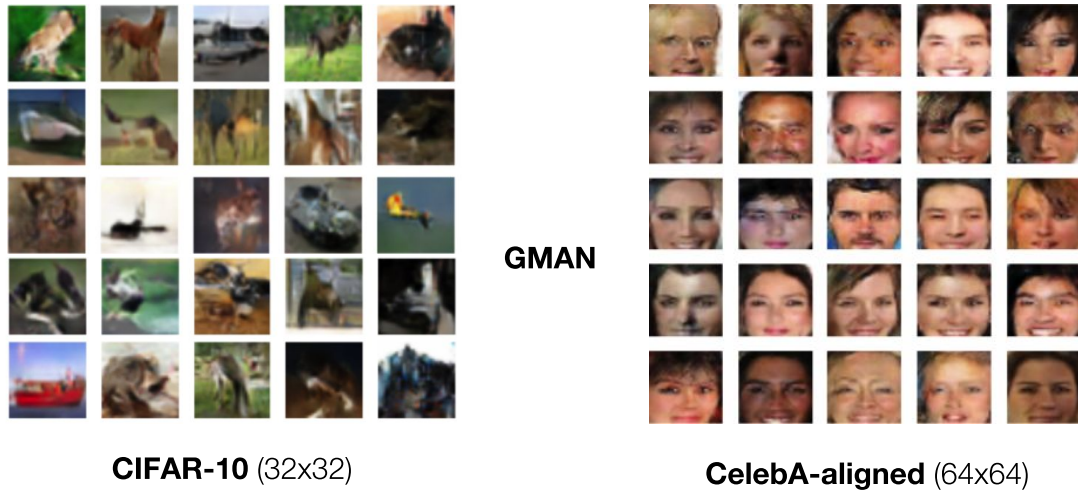


Figure 4.6: Random samples presented in [11] for image generation tasks such as CIFAR-10 and CelebA. Best results were achieved with GMAN-0 variant with  $N = 5$  discriminators. For CelebA they cropped the images to exclude background.

| GAN Variants   | Chi-square( $\times 10^5$ ) | KL-Div      |
|----------------|-----------------------------|-------------|
| DCGAN          | 0.90                        | 0.322       |
| WGAN           | 1.32                        | 0.614       |
| BEGAN          | 1.06                        | 0.944       |
| GoGAN          | 2.52                        | 0.652       |
| Unrolled GAN   | 3.98                        | 1.321       |
| Mode-Reg DCGAN | 1.02                        | 0.927       |
| InfoGAN        | 0.83                        | 0.210       |
| MA-GAN         | 1.39                        | 0.526       |
| MAD-GAN        | 0.24                        | 0.145       |
| GMAN           | 1.44                        | 0.69        |
| <b>ours</b>    | <b>0.03</b>                 | <b>0.02</b> |

Table 4.1: 1D Gaussian Mixture Model experiment using best results from 3 runs for GAN variants that aims to solve mode collapse. Available values were taken from [13].

| $N$ | Chi-square( $\times 10^7$ ) |                                 | KL-Div          |                                 |
|-----|-----------------------------|---------------------------------|-----------------|---------------------------------|
|     | GMAN                        | <b>ours</b>                     | GMAN            | <b>ours</b>                     |
| 2   | 5.00 $\pm$ 6.80             | <b>1.89<math>\pm</math>0.92</b> | 1.74 $\pm$ 0.63 | <b>0.81<math>\pm</math>0.27</b> |
| 3   | 2.96 $\pm$ 2.88             | <b>1.10<math>\pm</math>2.43</b> | 1.50 $\pm$ 0.57 | <b>0.55<math>\pm</math>0.36</b> |
| 4   | 3.41 $\pm$ 2.73             | <b>0.74<math>\pm</math>0.98</b> | 1.48 $\pm$ 0.42 | <b>0.50<math>\pm</math>0.41</b> |
| 5   | 4.62 $\pm$ 3.92             | <b>0.27<math>\pm</math>0.54</b> | 1.55 $\pm$ 0.30 | <b>0.25<math>\pm</math>0.26</b> |
| 6   | 3.94 $\pm$ 3.22             | <b>0.41<math>\pm</math>0.50</b> | 1.56 $\pm$ 0.22 | <b>0.35<math>\pm</math>0.20</b> |
| 7   | 2.84 $\pm$ 1.51             | <b>0.42<math>\pm</math>0.43</b> | 1.45 $\pm$ 0.38 | <b>0.36<math>\pm</math>0.21</b> |
| 8   | 2.80 $\pm$ 1.55             | <b>0.93<math>\pm</math>1.14</b> | 1.36 $\pm$ 0.43 | <b>0.56<math>\pm</math>0.31</b> |

Table 4.2: 1D Gaussian Mixture Model experiment using best results from 20 runs with different numbers of discriminators (4.1).

| GAN Variants | Chi-square( $\times 10^6$ ) | KL-Div       |
|--------------|-----------------------------|--------------|
| Standard GAN | 3.883                       | 2.860        |
| GMAN         | 1.253                       | 0.636        |
| <b>ours</b>  | <b>0.449</b>                | <b>0.246</b> |

Table 4.3: 2D Gaussian Mixture Model experiment with  $M = N = 8$  (where  $N$  is the number of discriminators,  $M$  is the number of Gaussians we used in the mixture) For each experiment we use a fixed set of 1,000,000 samples and take 5 run per each algorithm, then report the best run. We took effort to make sure that the comparison was fair, and used the same set of parameters as it was done in the 1D experiments.

## 5. Summary

### 5.1 Overview

**Method** We have introduced a new approach to train Generative Adversarial Networks [15], and claimed that our algorithm to use multiple discriminators is more efficient than the original GMAN [11], since each agent learns a subspace of the true data distribution and is able to provide more useful gradients for the discriminator.

**Experiments** In our experiments we have proven our claims with empirical results on both low-dimensional (1D and 2D GMM) tasks and on large scale image generation (CelebA and CIFAR-10). On the Gaussian Mixture Model experiments we illustrated how Triple-GMAN mechanism takes effect. On image generation tasks we have shown qualitative advantages of our approach.

### 5.2 Discussion

Triple-GMAN was proven experimentally to utilize the capacity of multiple discriminator by partitioning the target distributions and reducing the complexity for each discriminator to learn. We showed that increasing the number of discriminator improves the performance of the network when the training has converged until the point where the number of discriminators exceeds the number of underlying modes. On the other hand, we saw that in real life problems the number of modes is higher than what we can practically outnumber with the discriminators, therefore we are able to improve on the lower bound of mode collapses as long as we are able to provide enough compute. Our main concern is that the weights that models only a limited distribution could be reused, or more generally learned.

### 5.3 Future work

We saw that it is not necessary for a generator to have equal capacity adversary or adversaries to converge, meaning that the standard GAN training procedure could be replaced with multiple (even weaker) discriminators specialized only for attracting the conditional model distribution of the generator on a single mode.

# Acknowledgement

We would like to acknowledge Viveka Kulhari, Arnab Ghosh for their helpful discussions. Viveka shared his evaluation code for the 1D experiments with us. We implemented our experiments in PyTorch [38]. This work was supported by the EPSRC, ERC grant ERC-2012-AdG, EPSRC grant See-bibyte EP/M013774/1 and EPSRC/MURI grant EP/N019474/1. We would also like to acknowledge the Royal Academy of Engineering and FiveAI. Botos Csaba is supported by the joint scholarship of Pázmány Péter Catholic University, University of Oxford, the Erasmus+ Mobility Program and the New National Excellence Program.

# Bibliography

- [1] Martin Arjovsky and Léon Bottou. “Towards principled methods for training generative adversarial networks”. In: *ICLR*. 2017.
- [2] Martin Arjovsky, Soumith Chintala, and Léon Bottou. “Wasserstein gan”. In: *arXiv preprint arXiv:1701.07875* (2017).
- [3] Sanjeev Arora et al. “Generalization and Equilibrium in Generative Adversarial Nets (GANs)”. In: *Proceedings of the 34th International Conference on Machine Learning*. 2017.
- [4] Sanjeev Arora et al. “Generalization and equilibrium in generative adversarial nets (gans)”. In: *ICML*. 2017.
- [5] David Berthelot, Thomas Schumm, and Luke Metz. “BEGAN: boundary equilibrium generative adversarial networks”. In: *arXiv preprint arXiv:1703.10717* (2017).
- [6] Tong Che et al. “Mode regularized generative adversarial networks”. In: *ICLR*. 2017.
- [7] Xi Chen et al. “Infogan: Interpretable representation learning by information maximizing generative adversarial nets”. In: *NIPS*. 2016.
- [8] LI Chongxuan et al. “Triple generative adversarial nets”. In: *Advances in Neural Information Processing Systems*. 2017, pp. 4088–4098.
- [9] Jeff Donahue, Philipp Krähenbühl, and Trevor Darrell. “Adversarial feature learning”. In: *ICLR*. 2017.
- [10] John Duchi, Elad Hazan, and Yoram Singer. “Adaptive subgradient methods for online learning and stochastic optimization”. In: *Journal of Machine Learning Research* 12:Jul (2011), pp. 2121–2159.
- [11] Ishan Durugkar, Ian Gemp, and Sridhar Mahadevan. “Generative multi-adversarial networks”. In: *ICLR*. 2017.
- [12] Arnab Ghosh, Viveka Kulharia, and Vinay Namboodiri. “Message passing multi-agent gans”. In: *arXiv preprint arXiv:1612.01294* (2016).
- [13] Arnab Ghosh et al. “Multi-agent diverse generative adversarial networks”. In: *arXiv preprint arXiv:1704.02906* 1.4 (2017).
- [14] Ian Goodfellow. “NIPS 2016 tutorial: Generative adversarial networks”. In: *arXiv preprint arXiv:1701.00160* (2016).
- [15] Ian Goodfellow et al. “Generative adversarial nets”. In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.
- [16] Ian Goodfellow et al. “Generative adversarial nets”. In: *NIPS*. 2014.
- [17] Ishaan Gulrajani et al. “Improved training of wasserstein gans”. In: *NIPS*. 2017.
- [18] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 770–778.

- [19] Irina Higgins et al. “beta-vae: Learning basic visual concepts with a constrained variational framework”. In: *International Conference on Learning Representations*. 2017.
- [20] Geoffrey E Hinton et al. “Improving neural networks by preventing co-adaptation of feature detectors”. In: *arXiv preprint arXiv:1207.0580* (2012).
- [21] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. “Multilayer feedforward networks are universal approximators”. In: *Neural networks* 2.5 (1989), pp. 359–366.
- [22] Moos Huetting. Accessed: December 14, 2018. URL: <http://math.stackexchange.com/questions/945871>.
- [23] Andrej Karpathy. “Connecting Images and Natural Language”. PhD thesis. Stanford University, 2016.
- [24] Andrej Karpathy. “Andrej Karpathy blog Hacker’s guide to Neural Networks”. Accessed: December 14, 2018. URL: <http://karpathy.github.io/neuralnets/>.
- [25] Andrej Karpathy, Fei-Fei Li, and Justin Johnson. “CS231n Convolutional Neural Networks for Visual Recognition.” Accessed: December 14, 2018. 2016. URL: <http://cs231n.github.io/>.
- [26] Tero Karras et al. “Progressive growing of gans for improved quality, stability, and variation”. In: *ICLR*. 2018.
- [27] Diederik P. Kingma and Max Welling. “Auto-Encoding Variational Bayes.” In: *CoRR* abs/1312.6114 (2013). URL: <http://dblp.uni-trier.de/db/journals/corr/corr1312.html#KingmaW13>.
- [28] Diederik Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).
- [29] Christian Ledig et al. “Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network.” In: *CVPR*. 2017.
- [30] Chongxuan Li et al. “Triple generative adversarial nets”. In: *NIPS*. 2017.
- [31] Ming-Yu Liu and Onel Tuzel. “Coupled generative adversarial networks”. In: *NIPS*. 2016.
- [32] Lars Mescheder. “On the convergence properties of GAN training”. In: *arXiv preprint arXiv:1801.04406* (2018).
- [33] Lars Mescheder, Sebastian Nowozin, and Andreas Geiger. “The numerics of gans”. In: *Advances in Neural Information Processing Systems*. 2017, pp. 1825–1835.
- [34] Luke Metz et al. “Unrolled generative adversarial networks”. In: *ICLR*. 2017.
- [35] Michael Nielsen. “Neural Networks and Deep Learning”. Accessed: December 14, 2018. 2016. URL: <http://neuralnetworksanddeeplearning.com/>.
- [36] Augustus Odena, Christopher Olah, and Jonathon Shlens. “Conditional Image Synthesis with Auxiliary Classifier GANs”. In: *ICML*. 2017.
- [37] Aäron van den Oord et al. “Conditional Image Generation with PixelCNN Decoders”. In: *CoRR* abs/1606.05328 (2016).
- [38] Adam Paszke et al. “Automatic differentiation in PyTorch”. In: *NIPS-W*. 2017.
- [39] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering, 1994.
- [40] David Saad. *On-line learning in neural networks*. Vol. 17. Cambridge University Press, 2009.
- [41] Tim Salimans et al. “Improved techniques for training gans”. In: *NIPS*. 2016.
- [42] Jonathan Richard Shewchuk. *An introduction to the conjugate gradient method without the agonizing pain*. 1994.

- [43] Akash Srivastava et al. “Veegan: Reducing mode collapse in gans using implicit variational learning”. In: *Advances in Neural Information Processing Systems*. 2017, pp. 3308–3318.
- [44] Ilya Sutskever et al. “On the importance of initialization and momentum in deep learning.” In: *ICML (3)* 28 (2013), pp. 1139–1147.
- [45] Tijmen Tieleman and Geoffrey Hinton. “Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude”. In: *COURSERA: Neural Networks for Machine Learning* 4.2 (2012).
- [46] Carl Vondrick, Hamed Pirsiavash, and Antonio Torralba. “Generating videos with scene dynamics”. In: *NIPS*. 2016.
- [47] Li Wan et al. “Regularization of neural networks using dropconnect”. In: *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*. 2013, pp. 1058–1066.
- [48] Feng Wang et al. “NormFace:  $L_2$  Hypersphere Embedding for Face Verification”. In: *CoRR* abs/1704.06369 (2017). arXiv: 1704.06369. URL: <http://arxiv.org/abs/1704.06369>.
- [49] Paul John Werbos. *The roots of backpropagation: from ordered derivatives to neural networks and political forecasting*. Vol. 1. John Wiley & Sons, 1994.
- [50] Jun-Yan Zhu et al. “Unpaired image-to-image translation using cycle-consistent adversarial networks”. In: (2017).



# Appendices

# A. Neural Network Basics

This small introduction intends to cover the very basic steps towards experimenting with deep neural networks. The main significance of commonly used machine learning frameworks is that they can be decomposed to core building blocks that involves elementary operations between single nodes, or neurons. These operations are straightforward to implement, and easy to parallelize if the algorithms used are vectorized. The tone of the following sections follow a more informal manner that aims to help the reader get more familiar with the topic.

## A.1 Single Layered networks

For example if a small picture is given to a person, he or she guesses about it. If that person is told that there are four classes, like *Car*, *Plane*, *Cat*, *Kid*, he can tell how likely it is that the given picture falls in that class, so he can give a so-called *confidence parameter*. Patterns which are associated with cats may be associated with kids too, but is unlikely to be associated with planes. Deciding whether a pattern improves the confidence on each class or not, yields a **sign** for the given pattern, and the measure how strongly it influences the likelihood is called the **weight** of the pattern. In *neural networks* there are small nodes based on the biological model of neurons, that is responsible for recognizing and weighting the patterns. These nodes are called **perceptrons**.

### Defining the Perceptron

Let  $x$  be preprocessed information, the perceptron  $\mathbf{P}$  decides which parts of it are important in recognizing a cat: having a corresponding weight with large magnitude, and which of them are irrelevant: having a weight with magnitude close to zero. Therefore a *perceptron* has a weight  $w_i$  for each input  $x_i$  and its transient state can be now formally written:

$$\mathbf{P}_{transient} = \sum_i x_i w_i$$

Which can be also rewritten as the inner product of  $\mathbf{x}$  and  $\mathbf{w}$

$$\mathbf{P}_{transient} = \langle \mathbf{x}, \mathbf{w} \rangle = \mathbf{x}^T \mathbf{w}$$

In the following examples perceptrons will be arranged in a special way, to form a **Feed Forward** network, meaning that the information flow will occur in a direct way, without feedback - see figure A.1. However wiring many perceptrons together results in a numerically unstable system, caused by the lack of any restriction on the magnitude of the weights. Regularization addresses exactly this problem, implicitly

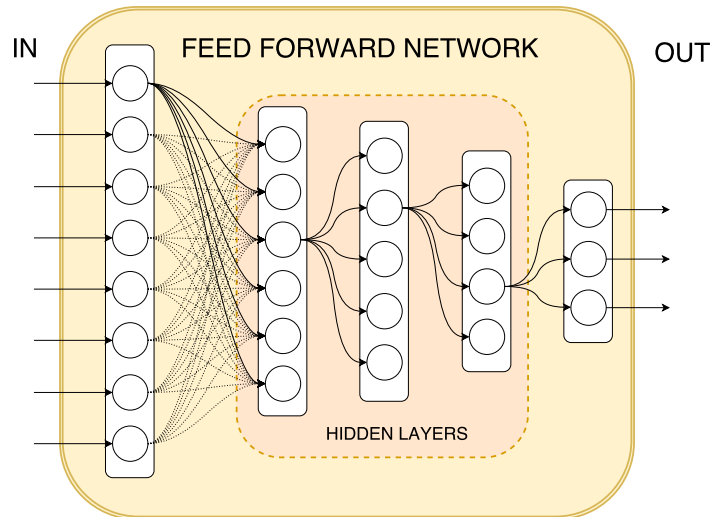


Figure A.1: A small feed forward network with three hidden layer composed of Fully Connected layers.

making the network less likely to simply memorize pictures, or only respond to samples from the training set – the problem of overfitting.

## Basics of Learning

After describing the model, the first question is:

— *what are the correct values for  $\mathbf{w}_{\mathbf{P}}$  for each  $\mathbf{P}$  in network  $\mathcal{F}$ ?*

There are a lot of intuitive explanations how this problem should be approached and solved. For an exhaustive study of the topic, visit Michael Nielsen’s website [35]. Anyhow, in general this is still the most studied question in machine learning. Luckily, if  $\mathcal{F}$ ’s performance can be measured, thanks to its feed-forward structure a *numerical suggestion* can be defined as well, which tells how to change  $\mathbf{w}_{\mathbf{P}}$  to improve efficiency of  $\mathcal{F}$ . In practice an  $\mathcal{L}$  Loss function is defined, and the objective of the training is to reduce it. Without digging deep in math we can find an intuitive situation with the same results.

Take a perceptron  $\mathbf{P}$  with input  $\mathbf{x}$  of objects on a given picture of a cat. Suppose that this perceptron *fires* (has high output value) when wheels are on the picture. It should remain silent when there are no round objects listed in  $\mathbf{x}$ , still it turns on, affecting the output of  $\mathcal{F}$ , producing high error. Since we know the original label of the picture, we can tell which assumptions were wrong, and which were good - which to decrease, which to increase if the next time  $\mathcal{F}$  is given the same input. This information is distributed between the previous perceptrons which caused the current one to fail, by simply multiplying the error with the weight corresponding to the previous node. Also with the information of how much the output of the actual node influenced the error, and with its input values, the significance of misleading  $\mathbf{x}_i$  can be reduced, and important features’ weights can be increased, which is actually finding a better  $\mathbf{w}_{\mathbf{P}}$ . For further intuitive examples see section 5, or visit Andrej Karpathy’s guide [24].

**SGD.** The example led to the practical application of the so called *Stochastic Gradient Descent*. Originally every samples in the training set should be introduced to the system before updating its weights and that would be the *Batch Gradient Descent*, but in the hope that the samples fall in a subset of the whole space of all possible inputs, the algorithm uses mini-batches for the learning process. When the

number of samples in the batches reduces to 1, the training is called on-line training. And if it happens on  $\mathcal{F}$  containing a single perceptron it is called *Rosenblatt perceptron learning algorithm*.

## A.2 Inference

Let us suppose that for a particular task an  $\mathcal{F}$  is given, first what we have to understand is how input data  $x \in \mathbb{X}$  (interchangeably  $x_1$ ) is inferred. First, assume that the data can be expressed as multi-dimensional matrix, like RGB pictures, audio recordings, gene maps. Some networks preserve the spatial information i.e. feature extraction performed on images, while other instances operate on the whole input data i.e. processing audio samples in frequency domain. Either way the output  $y$  (or  $y_L$ ) can be obtained by feeding  $x$  to the network:  $y = \mathcal{F}(x)$ . The core concept is that we can compose such an  $\mathcal{F}$  function by applying multiple projections to  $x$ . Practically that means sending input through the first layer, the second, and all the way through to the last layer  $L^{th}$ , which output would be the value of  $\mathcal{F}(x)$ , the response of the network. Therefore in terms of evaluating  $\mathcal{F}(x)$  layer by layer, actually translates to a single function call, which can be unfolded to a sequence of embedded projections:

$$\begin{aligned}\mathcal{F}(x) &= F_L(x_L) = F_L(y_{(L-1)}) \\ F_L(y_{(L-1)}) &= F_L(F_{(L-1)}(x_{(L-1)})) = F_L(F_{(L-1)}(\cdots F_1(x)))\end{aligned}$$

Using the function composition operator  $\circ$ , rewritten in the classical notation:

$$\mathcal{F}(x) = F_L \circ F_{(L-1)} \circ \cdots \circ F_1(x) \quad (\text{A.1})$$

The A.1 equation is the most fundamental idea behind feed-forward neural networks, namely the *inference* or *forward-propagation*. As mentioned above, every layer  $l$  is represented by an  $F_l$ . The most basic layers are the *Fully Connected* and *Activation* layers.

### Fully Connected layer

These layers carry out the heavy-lifting of inference by performing linear projection and translation transformations. The operations are following the rules of basic linear algebra, where the input  $x_{FC} \in \mathbb{R}^N$  and the output  $y_{FC} \in \mathbb{R}^M$  are specified as real valued vectors. The parameters of the layer  $\phi_{FC} = (W, b)$  are the corresponding weights and biases of each perceptron node in the layer forming a *weight matrix*  $W \in \mathbb{R}^{M \times N}$  and a *bias vector*  $b \in \mathbb{R}^M$  respectively. Therefore evaluating the output of the Fully Connected layer is defined by the following:

$$\begin{aligned}y_i &= \left( \sum_j W_{i,j} x_j \right) + b_i \\ y &= W \cdot x_j + b \\ [M] &= [M \times N] \cdot [N] + [M]\end{aligned} \quad (\text{A.2})$$

### Activation layer

Nodes in activation layers are introducing non-linearity to the network, by applying the same non-linear activation function to the corresponding output of the previous layer, performing element-wise operation.

Let  $F$  be an activation layer with activation function  $f$  after a fully connected layer with 3 neurons:

$$F(x) = \begin{pmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \end{pmatrix}$$

These functions are essential for the network, since they increase the numerical stability: they *squeeze* or *mitigate* the input preventing the network from *saturation* or *explosion* (numerical of course). Conventionally the following functions are applied most often as activation function:

$$\text{RectifiedLinearUnit(ReLU)} := \max\{0, x\} \quad (\text{A.3})$$

$$\text{HyperbolicTangent(TanH)} := \tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \quad (\text{A.4})$$

$$\text{SoftPlus(SP)} := \ln(1 + e^x) \quad (\text{A.5})$$

$$\text{Logistic(Log)} := \frac{1}{1 + e^{-x}} \quad (\text{A.6})$$

The only constraint on these functions that they have to keep the dimension of the input, namely  $F_{act} : \mathbb{R}^N \mapsto \mathbb{R}^N$ . *Note:* These functions do not have any variable parameters, therefore activation layers cannot be trained.

### A.3 Measuring efficiency

If the function  $\mathcal{F}$  mentioned above is given, and satisfies our needs, then we are done. However this is usually not the case, and finding the optimal  $\mathcal{F}^*$  is the main challenge targeted by many branches of Machine Learning. Despite it was proven, that standard multilayer feed-forward networks are capable of approximating any measurable function to any desired degree of accuracy [21], if the goal function  $\mathcal{F}^*$  is unknown, or too abstract to be *measurable* (i.e. telling how funny a picture is), we cannot utilize the universal approximator.

### Loss

By reformulating the objective, we can define a Loss function  $\mathcal{L} : \mathcal{F}(\mathbb{X}) \mapsto \mathbb{R}$ , which maps our candidate  $\mathcal{F}$  network to a scalar field, that represents the general correctness of  $\mathcal{F}$  over the space of possible inputs  $\mathbb{X}$  – the lower its value the better  $\mathcal{F}$  is performing. By doing so we may apply Machine Learning algorithms that would *minimize* the Loss, therefore  $\mathcal{F}$  would converge towards  $\mathcal{F}^*$  implicitly.

*Note:* In practical implementations  $\mathcal{F}$  is not evaluated over the whole space of possible inputs, instead in the hope that a small subset of both *training* and *validating* samples called a mini-batch will approximate  $\mathcal{L}(\mathcal{F})$  as well. Useful practices for reducing computing complexity and improving stability, and rate of convergence will be covered later.

### Supervised learning

In cases where the parameters of  $\mathcal{F}^*$  are not known, but we know how it would map the input space  $\mathbb{X} \mapsto \mathbb{Y}$ , e.g. which character appears on the input image  $x$ , then we can define a set of previously *labeled* pairs of input - solution sample which could be used later on for training, and evaluating performance of the network.

## A.4 Adjusting parameters

Generally speaking, applying Machine Learning algorithms boils down to the process of iteratively altering parameters  $\phi$  of  $\mathcal{F}$  to optimize the Loss. Once  $\mathcal{L}(\mathcal{F})$  is obtained, we can evaluate how changing the parameters would influence it – evaluate the gradient  $\nabla_{\phi}\mathcal{F}$  of the parameters with regards to  $\mathcal{L}$ .

### Gradient Descent

Updating  $\phi$  by descending on the gradient slope with a small step size  $\epsilon$  will decrease  $\mathcal{L}$ . Derivation from the general form of Gradient Descent depends on the architecture of the network, but there is a main concept for doing so, called Backpropagation described by Werbos *et al.* [49]. The exact method which can be applied for Fully Connected networks will be explained in detail in section 5.

### Training Policies

Though theoretically after finite steps of iterations, with Gradient Descent  $\mathcal{F}$  may approximate any Borel-measurable function, in practice Deep Neural Networks can fall into local-minimum of the parameter-space. I.e. the network tends to learn the very basic features of the input space, if it is not forced to generalize. The problem with generalization is that the training set has finite samples and extending it requires human-supervision. Therefore training networks with many layers requires some advanced techniques for training. There are multiple way to improve the *convergence rate* and the *stability* of the network.

**Batch processing.** There are two radical methods of updating the parameters in the network.

Theoretically if we wanted to create a perfect network, we would infer all possible samples from  $\mathbb{X}$  and the Loss function  $\mathcal{L}$  would evaluate the network on every solution (in a single iteration, without the network being updated), then we would take the mean of the Loss to evaluate the gradient, e.g.

$$\mathcal{L}^* = \frac{1}{N} \sum_{x \in \mathbb{X}} \mathcal{L}(\mathcal{F}(x))$$

One update on the whole training set is called a **batch**. This way, if the capacity of  $\mathcal{F}$  is large enough, it is possible to train the network to be able to solve every problem. In a special case when we want to simulate logical circuits with real values instead of Boolean **true** or **false**, then we can train a network to act like a logical processing unit - since we are able to map  $\forall x \in \mathbb{X} \mapsto y \in \mathbb{Y}$  by external evaluation.

The other case is more practical, when the train data is not available at once, but in sequence. The network is usually updated in *testing* time and that is why it is called **on-line** training. The topic is described exhaustively in [40]. In practice the network is trained previously before on-line training (even if a very few sample are available), because the responses are not just used for evaluating the Loss function, but taken into account as valid predictions. Normally there is no opportunity to label freshly acquired samples, and this is the main reason that this technique is used mainly in *unsupervised learning* tasks. However when supervision is available, then the network can be trained in the following manner: predictions which were correct are put into the training set, and the network weights are updated immediately to encourage the same response for similar samples. A frequent application of on-line training is called *Q-learn*, for case studies see [39].

Both methods are powerful for specific tasks, but in practice the solution is in between: **mini-batch** training. Inferring multiple samples at the same time with the same network not just yields a more stable convergence, but is an excellent opportunity to parallelize the process: Extending by one extra dimension to each stage of the inference (inputs, transient activations, outputs) can be done easily, also almost every library can take the advantage to perform optimized matrix operations on tensors (at least *NumPy* can). On the second hand, processing only *some* instead of all of the input samples at once is computationally a better choice. In real applications it is worth considering  $2^k$  mini-batch sizes because of the requirement of memory allocation [25].

**Advanced First order derivatives.** Simple heuristics applied to first order derivatives, such as:

**Momentum** [44]  $v_{(n+1)} = \gamma v_n + \nabla_{\phi_n} \mathcal{L}$  and  $\phi_{(n+1)} = \phi_n - \epsilon v_n$

**Adagrad** [10]  $r_{(n+1)} = r_n + (\nabla_{\phi_n} \mathcal{L})^2$  and  $\phi_{(n+1)} = \phi_n - \frac{\epsilon}{\gamma + \sqrt{r_n}} (\nabla_{\phi_n} \mathcal{L})^2$  (element-wise)

**RMSProp** [45]  $r_{(n+1)} = (1 - \epsilon)r_n + \epsilon (\nabla_{\phi_n} \mathcal{L})^2$

**Adam** [28] Combination of RMSProp and momentum.

A more sophisticated, but expensive method is using second order derivative, for instance Conjugate gradient [42].

**Decreasing the learning rate  $\epsilon$ .** For large networks it is inevitable to use decreasing learning rate, otherwise the convergence would take too much time (small  $\epsilon$ ) or would not even occur (large  $\epsilon$ ).

**Cross-Validation.** Special  $\hat{\mathcal{L}} = (\mathcal{L}_{train}, \mathcal{L}_{validation})$  functions approximate the efficiency of the network not just by evaluating them on the currently or previously trained samples, but on samples which are from a totally **disjunct set**  $\mathbb{X}_{validation}$ . The motivation behind it is to see how well would the network perform on samples which it has not faced before. By doing so we can evaluate how well the network *generalized* the information from samples of the training set. In order to keep  $\mathbb{X}_{train} \cap \mathbb{X}_{validation} = \emptyset$  we may not use  $\mathcal{L}_{validation}$  for parameter updating. However in order to **prevent overfitting** we can monitor e.g. generalization error  $E_g = \frac{\hat{\mathcal{L}}_1}{\hat{\mathcal{L}}_2}$  during training and use **Early Stopping** when  $E_g > \delta$ , where  $\delta$  is a parameter of how much we tolerate overfitting.

## Networks in practice

Though networks can vary in shape and function, the representation of the succeeding layers, evaluated by embedded functions described in A.1 is a common feature. Each of these layers serve as nodes of the computational graph of the network. The type of the layer determines whether it can be updated or not: for *Fully Connected* layers  $\nabla_{\phi} F_{FC}$  is well defined, explained in the following example.

**Toy example.** Assume  $\mathcal{L}$  is given, and we have a fully connected network with 2 hidden layers, i.e.  $L = 3$ , which maps a  $\mathcal{F}$  dimensional input vector  $x$  to a  $M$  dimensional output vector  $y$ , namely  $\mathcal{F} : \mathbb{R}^N \mapsto \mathbb{R}^M$ . The constraint on the parameters are:

$W^1$  of the first layer must have  $N$  columns

$W^3$  and the bias  $b^3$  of the last layer must have  $M$  rows, dimensions respectively.

Number of rows of  $W^l$  must match  $\dim(b^l)$  3 dimensional

Number of columns of  $W^l$  must match  $\dim(b^{(l-1)})$

Then the evaluation unfolded would look like the following:

$$y = \mathcal{F}(x) = F_3 \circ F_2 \circ F_1(x) = W^3(W^2(W^1(x) + b^1) + b^2) + b^3$$

For further usage and simplicity, we would like to fix these numbers. Let  $\mathcal{F}$  be a network with the following *shape*:  $[5, 4, 3]$  meaning that in each layer there are 5, 4, 3 neurons respectively,  $M = 3$  and all nodes are connected to the previous layer. The width of the input layer is not yet defined, let it be  $N = 10$ . *Note*: It is usually distracting and redundant to explicitly write the width of the outermost layers when testing different networks, because the input and the output layers must have fixed dimension for the same task, while the width of the hidden layers are varied. Define an  $L_2$  Loss function on the toy example. For one sample-label pair  $(x, y^*)$  the  $L_2$  Loss is:

$$\mathcal{L}_2(\mathcal{F}) = \frac{1}{2} \sum_i (y_i^* - \mathcal{F}(x)_i)^2 = \frac{1}{2} \sum_i (y_i^* - y_i)^2 \quad (\text{A.7})$$

$L_2$  is a universal Loss function, which is used in cases where the label space  $\mathbb{Y}$  is continuous (e.g. floorspace, consumption and height of a house, based on the price:  $\mathbb{R}^1 \mapsto \mathbb{R}^3$ ).

## A.5 Differentiation

In the above evaluating of  $\nabla_\phi \mathcal{F}$  can be done in two ways, namely by numerical approximation, or by analytical derivation, in the following we will discuss both.

### Numeric differentiation

Evaluating the numerical gradient (or difference) is an elementary, yet powerful operation, in which we would *perturb*, or modify one parameter  $\phi$  of our system  $\mathcal{F}$  at once. That is done by first adding  $\phi^+$  and after subtracting  $\phi^-$  a little amount  $d\phi$  from the original  $\phi$  and evaluate  $\mathcal{L}^\pm = \mathcal{L}(\mathcal{F}_{\phi^\pm})$ , namely the *Loss* of the system in the modified state, yielding the numerical gradient in the following equation:

$$\frac{d\mathcal{L}(\mathcal{F})}{d\phi} = \frac{\mathcal{L}^+ - \mathcal{L}^-}{2d\phi} = \frac{\mathcal{L}(\mathcal{F}_{\phi^+}) - \mathcal{L}(\mathcal{F}_{\phi^-})}{2d\phi} \quad (\text{A.8})$$

**Summary.** In a nutshell value of  $\frac{d\mathcal{L}(\mathcal{F})}{d\phi}$  tells how changing the parameter  $\phi$  by  $d\phi$  would change the performance of the network. If it is positive then updating  $\mathcal{F}$  by adding  $d\phi$  to  $\phi$  would result in higher Loss value, which is the opposite of our goal, so we just subtract it, if it is negative, then trivially we should add  $d\phi$  to  $\phi$  since it is making some good progress.

### Complexity

Though letting the computer do the hard work seems to be a good idea, it is worth considering that the simple method above will be applied to every  $\phi$  of  $\mathcal{L}$ . It means that for the network in the toy example, we need to evaluate  $\mathcal{L}(\mathcal{F})$  two times for each parameter in the weight matrices and the bias vectors of

the network, totaling in

$$\#(\phi) = 2 \times \sum_{i=1}^3 N_{(i-1)} \cdot N_i + N_i = 2 \times (10 \cdot 5 + 5 \cdot 4 + 4 \cdot 3 + 3) = 188$$

Even if  $\mathcal{F}$  is approximated by using  $k$ -sized mini-batches for evaluation it is still a computationally very expensive function, because the inference would result in the following number of operations of addition and multiplication:

$$\#(\text{operations}) = k \times \sum_{i=1}^3 2 \cdot N_{(i-1)} \cdot N_i + N_i = k \times 176$$

Therefore approximated with  $k = 10$  mini-batches would a single parameter update of a very tiny network would require total operations of:

$$\#(\text{total}) = \#(\phi) \times k \times \#(\text{operations}) = 188 \cdot 10 \cdot 176 = 330880 \quad (\text{A.9})$$

Because both  $\#(\phi)$  and  $\#(\text{operations})$  has complexity of  $\mathcal{O}(N^2)$ , one update will yield complexity of

$$\#(\text{total}) = \mathcal{O}(N^4) \quad (\text{A.10})$$

We can see that even for a shallow and relatively small network (industrial Awe networks has billions of parameters, and uses much larger batches) described in the toy example the method is really costly. That encourages us to derive our differentials on paper first, and use *numerical gradient approximation* for checking our solution. Using *Gradient Check* is essential when implementing new architectures, because it is a very efficient tool for debugging in comparison with updating. The method in a few words is about setting an error rate  $\epsilon$ , and decrease  $d\phi$  until the numeric solution does not match the analytic solution with  $1 - \epsilon$  significance. If the analytic solution is incorrect the cycle will not terminate.

## Analytic differentiation

Deriving the update by hand requires basic knowledge in calculus extended to multivariate cases, though since the operations are elementary, in general we must understand only three basic definitions to do so. Some formality before starting: we have three independent variables  $x, y, z$  and functions  $f, g, h$ . The result of operations performed on variables, i.e.  $x + 2y$  can be represented by a function  $f = x + 2y$ . If the value depends on a variable then it can be written explicitly, passing the variable as *the argument* of the function  $f(x, y) = x + 2y$ . For the sake of simplicity assume that the variables are not general objects from an abstract space, they are only real values:  $x, y, z \in \mathbb{R}$ . However the following description could be extended for the above-mentioned variables as well. For any one-dimensional function  $f(x) : \mathbb{R} \mapsto \mathbb{R}$  we say that the value represented by the function depends on the variable by the extent of its derivative. The derivative (or differential) of the function can be seen as an ideal case of A.8 where the perturbation would approach zero, namely:

$$\frac{\partial f}{\partial x} = \lim_{dx \rightarrow 0} \frac{df(x)}{dx} = \lim_{dx \rightarrow 0} \frac{f(x + dx) - f(x - dx)}{2dx} \quad (\text{A.11})$$

**Multiplication rule.** Consider a value  $x \cdot y \cdot z$  represented by  $f(x, y, z)$ .  $f$  is now depending on three variables, we can define the measure of this dependency on one variable by the formal equation:

$$\begin{aligned} \frac{\partial f}{\partial x} &= \lim_{dx \rightarrow 0} \frac{f(x + dx, y, z) - f(x - dx, y, z)}{2dx} = \lim_{dx \rightarrow 0} \frac{((x + dx) \cdot y \cdot z) - ((x - dx) \cdot y \cdot z)}{2dx} \\ &= \lim_{dx \rightarrow 0} \frac{(x + dx) - (x - dx)}{2dx} yz = \lim_{dx \rightarrow 0} \frac{2dx}{2dx} yz = yz \end{aligned}$$

We can apply the same method for each variable, the result will be the elements of the *gradient*  $\nabla f$

$$\frac{\partial f}{\partial x} = yz \quad \frac{\partial f}{\partial y} = xz \quad \frac{\partial f}{\partial z} = xy \quad (\text{A.12})$$

The important thing to understand that in a computational graph, a multiplicative node, which takes  $N$  arbitrary parameters (or arguments), will have a *partial derivative* for each variable its output is depending on. In general if these derivatives are represented as a vector, then it is called the gradient  $\nabla f$  of  $f$ . Also the value of the derivative will be the product of all variables except the one of which we are computing the influence of on the output.

**Addition rule.** Consider a value  $x \cdot y + x \cdot z$  represented by  $g(x, y, z)$ . The change of  $g$  with respect to  $x$  is defined with the following shortened equation:

$$\frac{\partial g}{\partial x} = \lim_{dx \rightarrow 0} \frac{((x + dx)y + (x + dx)z) - ((x - dx)y + (x - dx)z)}{2dx} = y + z \quad (\text{A.13})$$

Notice that – in the terms of computational graphs – if a node contributes to other different operations (namely  $x \cdot y$  and  $x \cdot z$ ), then the derivative of each occurrence in *later* values will be summed up.

**Chain rule.** Let  $f(x) = 2x + 3$  and  $g(f) = 5f$ . Suppose that we would like to know the derivative of  $g$  with respect to  $x$ . At first we cannot do so, but there are two options: in the hope that substituting the value represented by  $f$  into  $g$  would not make the equation too complex we can unroll the references and rewrite  $g(x) = 5 \cdot (2x + 3)$ , or we could use the chain rule:

$$\frac{\partial g}{\partial x} = \lim_{dx \rightarrow 0} \frac{g(f(x + dx)) - g(f(x - dx))}{2dx}$$

Assume that  $f(x + dx) - f(x - dx) \neq 0$ .

$$\lim_{dx \rightarrow 0} \frac{g(f(x + dx)) - g(f(x - dx))}{f(x + dx) - f(x - dx)} \cdot \frac{f(x + dx) - f(x - dx)}{2dx}$$

$$\frac{\partial g}{\partial x} = \frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial x} \quad (\text{A.14})$$

Which is a formula of the products of partial derivative of  $g$ , that treats  $f$  like a variable, and  $f$  explicitly operating on variable  $x$ . The derivative of  $g$  with respect to variable  $x$  is the product of the *local derivative* of  $g$  is  $\frac{\partial g}{\partial f} = 5$  and  $\frac{\partial f}{\partial x} = 2$  which equals  $\frac{\partial g(x)}{\partial x} = \frac{\partial(5 \cdot (2x+3))}{\partial x} = 10$ , the function strictly depending on  $x$ . The important message is that we can interchangeably use function values and variables with a constraint that at a point, in an arbitrary depth there must be a real variable. *Note:* the statement above stands for computations with *Acyclic Graph*, meaning that there should not be any feedbacks or loops – no definitions like  $f(g), g(f)$  or  $f(f)$ . We will see that these rules play a very fundamental role in training networks.

**Vector Notation.** Before drilling deep into mathematical equations, a small reminder: the following vector and matrix formulation is just a special annotation, using the rules above, which helps to make clear both the definition, and the computations done by the network when it is implemented. The vectors with partially derivatives inside are just representing *real values*, arranged in a fancy way. Every vector and matrix defined in forward propagation, has its corresponding derivative w.r.t. the Loss. More trivially, if any value is depending on a list of variables  $f(x_1, x_2 \dots x_n) = f(\mathbf{x})$  (a vector) then there is a list of *partial derivatives* w.r.t. to  $f$  – forming the gradient  $\nabla f = \left( \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2} \dots \frac{\partial f}{\partial x_N} \right)$ . *Notation:* When the vector notation is emphasized the variable name is conventionally written in bold font  $\mathbf{x}$ , or is underlined  $\underline{x}$ . Because later the indexing would become too crowded, we only use the indexed notation when it is necessary, otherwise using  $x$ .

The next step is formulating the dependency of multiple functions on multiple variables. As seen above, a multivariate function has its gradient vector – in the same fashion as the list of variables were organized into vectors  $\mathbf{x}$ , values composed of them can also form a vector  $F = (f_1, f_2 \dots f_M)$ , composing a multivalued function depending on the same variables. Doing so yields a first-order derivative matrix, composed of gradient vectors  $J = (\nabla f_1, \nabla f_2 \dots \nabla f_M)^T$ , called the *Jacobian of F*. The Jacobian has as many rows as output values  $F$  has, and the same number of columns of the variables that  $f_i$  is a function of.

$$J(F) = \begin{pmatrix} \nabla f_1 \\ \vdots \\ \nabla f_M \end{pmatrix} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_M}{\partial x_1} & \dots & \frac{\partial f_M}{\partial x_N} \end{pmatrix} \quad (\text{A.15})$$

The matrix and vector operations (such as addition and inner product) that can be performed on the derivative *arrays* are identical defined in the inference section. That is important because product of derivatives introduced by the chain rule, can be applied as well for multidimensional array of derivatives too.

## Fully Connected Layer

Return to the toy example and begin with the last layer, with 3 nodes. If a sample  $x$  is inferred  $\mathcal{F}(x) = y$ , then the response of the network will be a vector of  $\dim(y) = 3$ . If we took the  $L_2$  distance between the response and the goal  $y^*$ , then it would tell how far we are from the ideal, by a single scalar value. Since we want to minimize it, we have to adjust the parameters of the network, namely descend on the gradient slope. To get the small extent of the update we have to evaluate  $\frac{\partial \mathcal{L}}{\partial \phi}$ . Intuitively in the case we would like to correct the weights of a decision, it would require two things:

The original situation (the input of the  $l^{th}$  layer  $x_l$ ), which the decision was made in.

The error on the decision – the derivative  $\delta^l$  of the Loss with regards to the decision.

Since  $x_l$  is obtained via inference, what we have to calculate is  $\delta^l$  for the  $l^{th}$  layer in order to acquire the parameter gradient. The first step is to evaluate the  $\delta_L$ , or the *error* of the last layer's response  $y^3$ , namely  $\delta^3 = \nabla_{y^3} \mathcal{L}$ . Begin with the first element:

$$\delta_1^3 = \frac{\partial \mathcal{L}}{\partial y_1} = \frac{\partial}{\partial y_1} \frac{1}{2} ((y_1^* - y_1)^2 + (y_2^* - y_2)^2 + (y_3^* - y_3)^2) = y_1 - y_1^*$$

Expanding it to the whole array:

$$\delta^3 = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial y_1} \\ \frac{\partial \mathcal{L}}{\partial y_2} \\ \frac{\partial \mathcal{L}}{\partial y_3} \end{pmatrix} = \begin{pmatrix} \frac{\partial}{\partial y_1} \frac{1}{2} \sum_i (y_i^* - y_i)^2 \\ \frac{\partial}{\partial y_2} \frac{1}{2} \sum_i (y_i^* - y_i)^2 \\ \frac{\partial}{\partial y_3} \frac{1}{2} \sum_i (y_i^* - y_i)^2 \end{pmatrix} = \begin{pmatrix} y_1 - y_1^* \\ y_2 - y_2^* \\ y_3 - y_3^* \end{pmatrix}$$

Consider the following notation:  $\delta_i^3 = (y_i^* - y_i)$ , called parametric vector notation. Writing arrays in this way, saves a lot of space. However, when this notation gets jammed with indexes, it is useful to write down explicitly the whole array for clarification.

**The last layer** Now we have exact values of  $\delta^3$  and  $x^3$ , so we can calculate how should the weights in  $W^3$  be changed in order to get a better network. Applying the differentiation rules for each weight (forming a matrix) of the layer will result in a derivative for each weight (also forming a matrix). Taking the first perceptron of the layer, it has a weight  $W_1^3 = (W_{1,1}, W_{1,2}, W_{1,3}, W_{1,4})$  for each output of the previous layer.

Suppose that this neuron had to tell how rounded is the object on an image sample, and the  $i^{th}$  neuron of the  $2^{nd}$  layer fires when it recognizes sharp edges. Of course it would be bad if our neuron had a large weight on  $x_i^3$ . If this node performs poorly because of  $x_i^3$ , then it would contribute a lot to the Loss function while inferring sharp objects, with its output  $y_1$  being far away from  $y_1^*$ , resulting in a positive  $\delta_1^3$ . In case of  $x_i^3 = 0$ , the weight  $W_{1,i}$  has nothing to do with the error  $\delta_1^3$  of the neuron. Notice that the error of each weight (w.r.t  $\mathcal{L}$ ) should be proportional to the error and the input as well:  $\frac{\partial \mathcal{L}}{\partial w_{1,i}} = x_i^3 \cdot \delta_1^3$ . However it can be also derived in terms of the differentiation rules:

$$\frac{\partial \mathcal{L}}{\partial W_{1,i}} = \frac{\partial \mathcal{L}}{\partial y_1^3} \cdot \frac{\partial y_1^3}{\partial W_{1,i}} = \delta_1^3 \cdot x_i^3$$

**Considering the parameter update.** assume that a picture of an origami sculpture (a very edgy one) was inferred and the  $i^{th}$  node of the second layer worked correctly. Both  $x_i^3$  and  $\delta_1^3$  is positive, however the weight should be decreased: that is why we will take the negative of the derivative for updating  $w_{1,i}$ . Substituting  $i = 1, 2, 3, 4$  into  $\frac{\partial \mathcal{L}}{\partial w_{1,i}}$  yields a gradient of  $\mathcal{L}$  with regards to the weights of the first neuron of the last layer. Doing so for each neurons in the layer would result in 3 gradient vectors  $\nabla_{W_1} \mathcal{L}$ ,  $\nabla_{W_2} \mathcal{L}$  and  $\nabla_{W_3} \mathcal{L}$  which is practically stacked to make a matrix which can be later added element-wisely to the weight matrix  $W$ .

$$\nabla_W \mathcal{L} = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial W_{1,1}} & \frac{\partial \mathcal{L}}{\partial W_{1,2}} & \frac{\partial \mathcal{L}}{\partial W_{1,3}} & \frac{\partial \mathcal{L}}{\partial W_{1,4}} \\ \frac{\partial \mathcal{L}}{\partial W_{2,1}} & \frac{\partial \mathcal{L}}{\partial W_{2,2}} & \frac{\partial \mathcal{L}}{\partial W_{2,3}} & \frac{\partial \mathcal{L}}{\partial W_{2,4}} \\ \frac{\partial \mathcal{L}}{\partial W_{3,1}} & \frac{\partial \mathcal{L}}{\partial W_{3,2}} & \frac{\partial \mathcal{L}}{\partial W_{3,3}} & \frac{\partial \mathcal{L}}{\partial W_{3,4}} \end{pmatrix} = \begin{pmatrix} x_1 \delta_1 & x_2 \delta_1 & x_3 \delta_1 & x_4 \delta_1 \\ x_1 \delta_2 & x_2 \delta_2 & x_3 \delta_2 & x_4 \delta_2 \\ x_1 \delta_3 & x_2 \delta_3 & x_3 \delta_3 & x_4 \delta_3 \end{pmatrix}$$

The last part of the equation can be also expressed as:

$$\nabla_W \mathcal{L} = \left( \frac{\partial \mathcal{L}}{\partial W_{i,j}} \right) = (x_j \delta_i) = x \wedge \delta \quad (\text{A.16})$$

Where  $\wedge$  denotes the *outer product* operator. The gradient of the bias is simply  $\nabla_{b^L} \mathcal{L} = \delta^L$ .

**The  $(L-1)^{th}$  layer.** Gradient descent can be applied on networks with more than one layer, however continuing the example of the image descriptor network requires a bit more abstraction. In the previous explanation we assumed that the  $i^{th}$  neuron of the second layer worked properly. In general cases this assumption is incorrect, if the whole network is initialized at once. If we think about that also the mentioned neuron in the *last hidden layer* should be trained, then we can apply the same method with a 4 dimensional  $\delta^{(L-1)}$  and with a 5 dimensional  $x^{(L-1)}$  input.

*Note:* Capital  $L$  is representing the number of layers. In the toy example  $L = 3$ .

Acquiring the  $\delta^{(L-1)}$  is where the chain rule (A.14) steps into the scene. While in the example before we could find an intuitive workaround, in this case it would be quite strained, since  $\mathcal{L}$  does not depend directly on  $y_{(L-1)}$ . Utilizing the chain rule:

$$\delta^{(L-1)} = \frac{\partial \mathcal{L}}{\partial y^{(L-1)}} = \frac{\partial \mathcal{L}}{\partial y^L} \cdot \frac{\partial y^L}{\partial y^{(L-1)}} = \delta^L \cdot J(F_L) \quad (\text{A.17})$$

Where  $J(F_L)$  denotes the *Jacobian* (A.15) of the function representing the projection of the last layer  $F_L : \mathbb{R}^4 \mapsto \mathbb{R}^3$ . It is a map of how each input variable affects the output of the layer. In general: the Jacobian numerically can be represented as a  $3 \times 4$  matrix, analytically as a *local derivative* of function  $F$ . In case of *Fully Connected* layers the Jacobian is simply the weight matrix  $F(F_l) = W_l$  (the bias drops out here).

For derivative of scalar values ( $\mathbb{R}$ ) the *product* operator is well defined, and it can be expanded the same way to derivatives of multidimensional values as regular *inner product* of the values they are composed of. *Note:* In order to stay consistent with the dimensions of the computation, we have to switch sides of the matrix multiplication defined in (A.2):

$$\begin{aligned} \delta_j^{(L-1)} &= \sum_i \delta_i^L W_{i,j}^L \\ \delta^{(L-1)} &= \delta^L \cdot W^L \\ [4] &= [3] \cdot [3 \times 4] \end{aligned} \quad (\text{A.18})$$

## Backpropagation

The backpropagation algorithm, first described by Werbos *et. al* [49]

**The  $l^{th}$  layer.** The contribution to the Loss of the general  $l^{th}$  layer  $\delta^l$  can be retrieved by unfolding  $\frac{\partial \mathcal{L}}{\partial y^l}$  applying the chain rule, namely the backpropagation:

$$\begin{aligned} \delta^l &= \frac{\partial \mathcal{L}}{\partial y^l} = \frac{\partial \mathcal{L}}{\partial y^L} \cdot \frac{\partial y^L}{\partial y^{(L-1)}} \cdots \frac{\partial y^{(l+1)}}{\partial y^l} \\ \delta^l &= \delta^L \cdot J(F_L) \cdot J(F_{(L-1)}) \cdots J(F_{(l+1)}) \\ [dim(l)] &= [dim(L)] \cdot [dim(L) \times dim(L-1)] \cdots [dim(l+1) \times dim(l)] \end{aligned} \quad (\text{A.19})$$

*Note:* the order of evaluating these derivatives is theoretically irrelevant, however computationally there is an opportunity to implement it in two ways [23]:

*forward-mode differentiation:* evaluating in the order of layers is efficient in cases where the output of the network is much larger than the input

*reverse-mode differentiation*: evaluating in reversed order for networks with fewer outputs than inputs.

As pointed out in the thesis, the former would require  $(L - l)$  times evaluating a *matrix-matrix* product and one *vector-matrix* operation, while the latter would require  $(L - l)$  times evaluating a *vector-matrix* product and one *matrix-matrix* at the end. Using forward-mode differentiation does not require keeping transient activations  $y^l$ , however it is computationally costly. Using backward-mode differentiation does not strain the CPU, but the memory. It is because if  $J(F_l)$  is not linear, then it requires the input  $x_l$  to evaluate the first order derivatives.

## Activation Layer

Though the activation layer operates on the input, it has no adjustable parameter. Since we have to evaluate  $\delta$  for layers behind activation layers, the error must pass through  $J(F_{activation})$  as well. Luckily activation layers applies a scalar function element-wise on the inputs, the Jacobian of the composite function  $F$  has a special attribute: it is **diagonal**, meaning that:

$$J(F) = \frac{\partial F_i}{\partial x_j} = \begin{cases} \frac{\partial f(x_i)}{\partial x_i} & i = j \\ 0 & i \neq j \end{cases}$$

We can exploit this function when implementing activation layers, by simply using element-wise product  $\frac{\partial f(x_i)}{\partial x_i}$  instead of a matrix multiplication. The mentioned activation functions (A.3, A.4, A.5, A.6), are differentiable *almost everywhere*, the corresponding derivatives:

$$(\text{ReLU})' := \text{Heaviside}(x) \tag{A.20}$$

$$(\text{TanH})' := 1 - \tanh(x)^2 \tag{A.21}$$

$$(\text{SP})' := \frac{1}{1 + e^{-x}} \tag{A.22}$$

$$(\text{Log})' := \text{Logistic}(x) \cdot (1 - \text{Logistic}(x)) \tag{A.23}$$

## A.6 Additional

### Pooling layers

Pooling layers are decreasing the parameter space with a fixed down-sampling projection which operates independently on tiles of input. In general pooling layers are moving a  $N$ -sized window, or a tile through the data taking **stride** steps. Data in each window is collapsed by an aggregation function, such as **mean()** or **max()** lowering the computational complexity of the next layers processes. Implementation of this layer can be a bottleneck of inference, since the operations carried out in each window may rely on the same input values if  $N > \text{stride}$  i.e. windows overlap. In our library a single **max-pool** layer is implemented with fixed parameters: **stride** =  $N = 2$ . Further information see [25] `~/convolutional-networks/#pool`.

### Dropout

Dropout layers target the problem of overfitting, being considered as a method of regularization. In theory they reduce complex co-adaptation via model averaging, by generating different networks (using the same parameters) in each forward pass [20]. Practically they drop activations of the previous layer

(with random *Bernoulli*( $p$ ) probability) when propagating information, forcing succeeding layers to not depend on an individual neuron of the layer before. Computationally it is very efficient to create different, so called **sub-networks**: both for forward and backward pass a single *Boolean array* mask is used. *Note*: Dropout layers are reducing the overall magnitude of the activation of the previous layer, considering that latent layers would be *saturated* when dropout layers were removed after training, the activations passed by this layer are multiplied by the constant  $\frac{1}{p}$ .

**DropConnect.** A similar concept, introduced by Li Wan *et al.* [47]: instead of dropping the activation of the previous layer, the weights of the layer are canceled out in the same fashion.

## Output

On the top of every network there should be a dedicated layer, which arranges the networks response. If a *(sample, label)* pair was inferred it also evaluates the result and output a *(response, loss)* pair. Therefore the output layer will determine how the network will be trained, and the response interpreted

**Softmax.** In case of classification, when the network has to decide which set was the input sampled from, then we can transform the response into a categorical probability distribution, i.e. how confident each neuron in the last layer in classifying the input as their 'own'. For that, we use **one-hot** notation, where  $y_i^* = [0, 0 \dots 1 \dots 0]$  (with only one non-zero element) represents the *ground-truth* label, the  $i^{th}$  class. The response will be turned into a discrete density function, e. g.  $= [0.1, 0.6, 0.2, 0.1]$  can be interpreted as the following: the input  $x$  is categorized as a sample from the second class with 60% confidence. Notice that  $|p|_1 = \sum_i y_i = 1$  constraint should be satisfied. To transform a real valued  $N$  dimensional vector  $y \in \mathbb{R}^N$  to the mentioned probability distribution  $p \in \Omega^N$  we apply the *Softmax* function:

$$p_i = \frac{\exp(y_i)}{|\exp(y)|_1} \quad (\text{A.24})$$

Conventionally the corresponding Loss function is the **log-loss** or **cross-entropy** function:

$$\mathcal{L}(y) = - \sum_i y_i^* \log p_i \quad (\text{A.25})$$

and the derivation of its gradient w.r.t. the original response of the last layer  $y$ , is the following (based on [22]):

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial y_i} &= - \sum_k y_k^* \frac{\partial \log p_k}{\partial y_i} = - \sum_k \frac{y_k^*}{p_k} \frac{\partial p_k}{\partial y_i} \\ &= -y_i^* (1 - p_i) - \sum_{j \neq i} \frac{y_j^*}{p_j} (p_i p_j) \\ &= -y_i^* - y_i^* p_i + \sum_{j \neq i} y_j^* p_i \\ &= -y_i^* + \sum_j y_j^* p_i \\ &= p_i \left( \sum_j y_j^* \right) - y_i^* \\ &= p_i - y_i^* \end{aligned} \quad (\text{A.26})$$

The trick of the derivation is that  $y^*$  has only one non-zero element which is 1, and the lemma:

$$\frac{\partial p_j}{\partial y_i} = \begin{cases} p_i(1 - p_i) & i = j \\ -p_i p_j & i \neq j \end{cases}$$