Generative Adversarial Networks

Daniel Yukimura

May 2017

Contents

Chapter 1

Introduction

Generative adversarial networks (GANs) were introduced in the groundbreaking work of Goodfellow et al. [18]. They essentially break down into a new strategy for training neural networks to solve unsupervised learning tasks. This adversarial training strategy follows a game-type procedure where two machines play against each other and use the result of the rounds as feedback for learning. This innovative idea has been used to create powerful generative models and to drive lots of impressive applications on recent years.

While most of the recent fame of deep learning can be credited to striking results on supervised tasks [29, 28, 21], deep generative models seem to have gathered a lot of attention from the research community in the last few years following the proposal of GANs. Some of the field's pioneers even have been selling the concept of adversarial training as the most innovative idea in deep learning of the past decade [9].

The term "generative model" in machine learning refers to any model that learns to estimate distributions from a collection of samples. GANs, for instance, achieve this by learning how to sample from the estimated distribution, instead of the usual density estimation, They seem to perform pretty well even for complex data like images, sound, videos, etc. Now, simply being able to generate new images while we already have plenty of them might seem quite silly. However, the value of generative models extend way beyond just creating more data as we hope to show in this work.

Generative models, and in particular GANs, are already highly successful on semi-supervised learning problems [49, 47], in which the labels for a part or even most of the training set are missing. Traditional deep learning algorithms typically rely on a huge amount of labeled examples to be able to generalize well. Generative modeling on the other hand have the capacity to learn the distribution of the unlabeled data and then can be used to improve generalization for classification.

Besides studying GANs in their traditional form, as standard generative models, we'll give special attention to the problem of creating conditional generative models. We will show how we can modify our GANs or use their capacity so that we are able to generate samples from conditional distributions. This provides a different level of control that makes the GAN model even more powerful. Tasks like generating realistic images from a text description or from a sketch, and controlled speech synthesis are

now feasible thanks to recent work on this direction [45, 60, 26, 51]. Applications of these techniques to art creation, computer graphics, computational biology and many others are recently gathering attention from a broader public to this subject.

Another application of generative models is to improve reinforcement learning algorithms. On this subject, generative models have recently led to interesting results on learning conditional distributions for planning, imitation learning, inverse reinforcement learning, and even drawing a relation between GANs and actor-critic models [13, 12, 11, 22, 40]. Mainly, the whole idea that generative models should be able to internalize concepts of our physical and digital world is a possible explanation for why both the fields of reinforcement learning and artificial intelligence have recently shown a lot of interest on the subject.

While previous generative models were promising, they were very hard to train due to approximation of intractable probabilistic operations. The adversarial training strategy proposed by the GANs framework has the key advantage that it can be easily trained using traditional forward and back-propagation routines used for training neural networks. Not only the strategy proved to be powerful, but also very easy to use, allowing several groups to work on different modifications of the model, resulting on the large number of applications we see today.

We start in chapter 2 by giving an statistical overview of classical machine learning concepts that we think are necessary to understand subsequent chapters. Those concepts are essentially supervised and unsupervised learning, where the later one is done in the context of generative modeling which fits best our interests.

Next, in chapter 3, we give a brief introduction to deep learning. We define and give examples for the main structures, neural networks, and explain how our training procedure works. We'll also briefly discuss how this subject, where GANs are included, have become so big and relevant lately.

On chapter 4 we finally formalize GANs as generative networks and study the associated minimax game. We also try to give the intuition behind the model. We present computational experiments and discuss some of the advantages and disadvantages observed.

In the last part, chapter 5, we discuss two distinct ways of conditioning GANs. The two methods have very different flavours. The first one trains a network that directly samples from a conditional distribution but doesn't work to well for complicated conditions, while the later allows different conditioning settings but has to perform an expensive search for some conditioned sample using an already trained GAN in the full distribution. We also discuss experiments on both methods and evaluate these disparities.

The main interest in this work is not only to present these models, but to analyse and observe them in practice so we can get a better intuition, and perhaps move towards really understanding how these models are able to learn such complex distributions only by observing samples from it. Much like in the bigger subject of deep learning we still don't know much about how GANs and other neural networks learn, how they understand data and even how their structure affects learning. Besides, GANs present a whole new dynamic which probably presents different problems, which we seem to be even further away from fully understanding. Answering some of these questions, even partially, would perhaps allow us to create better models, develop new ways of

controlling/conditioning our distributions, and of course, could also broaden our range of applications.

Chapter 2

Introduction to Machine Learning

2.1 Intuition for Supervised Learning

The most canonical class of problems in machine learning is known as **predictive** or **supervised learning**. It typically consists of learning the underlying predictive relationship between the inputs and outputs coming from a given data collection. Some examples of supervised learning problems are:

- Recognize the numbers in a handwritten ZIP code from digitized images ([30]).
- Predicting house/property prices from its features ([41]).
- Online advertisement from processing users info and clicking patterns ([36]).
- Transcribing the information contained in a raw audio signal (speech recognition) ([19], [42]).
- Translating a sentence from a English to Portuguese (machine translation, [56]).
- Identifying the position of other cars from images and radar information for autonomous driving ([5]).

Supervised algorithms receive labeled examples during training. This means the dataset used for training contains examples with features and values associated to each. The training dataset consists on a list of pairs $\{(X_i, Y_i)\}_{i=1}^n$, where the X's are the examples features and live in a **instance** or **input space** $\mathcal{X}$ and the Y's are the associated outcomes living in an **outcome space** $\mathcal{Y}$. Its also common to distinguish learning algorithms by the task at hand and the data structure. We typically distinguish between **classification** and **regression**; where the first assumes that the outcome space $\mathcal{Y}$ is a finite categorical set, and the later usually assumes an euclidean space $\mathbb{R}^d$. Other tasks like transcription and when the considered outcome space has a more complex structure

or interdependencies don't fall exactly in either of these classes, but are still supervised tasks.

Let's suppose our dataset $\mathcal{D} = \{(X_i, Y_i)\}_{i=1}^n$ satisfies a condition $Y_i = f(X_i)$ $\forall i \in [n] = \{1, 2, \ldots, n\}$, for some unknown function $f : \mathcal{X} \to \mathcal{Y}$. Then our main goal consists on finding a function $\hat{f} \in \mathcal{Y}^{\mathcal{X}}$, depending on the dataset, that well approximates f. More precisely, in this context, one can define a **learning algorithm** as a function

$$\hat{f} : \mathcal{X} \times \bigcup_{n=1}^{\infty} (\mathcal{X} \times \mathcal{Y})^n \to \mathcal{Y} \tag{2.1}$$

which given a dataset $\mathcal{D} \in (\mathcal{X} \times \mathcal{Y})^n$ it returns a predictor $\hat{f} = \hat{f}(\cdot, \mathcal{D})$. Its important to emphasize that such estimator should be a good approximation for unseen input instances. When this is possible we say our learning algorithm is able to **generalize** the concept of f from the dataset $\mathcal{D}$.

A common mistake when designing a learning algorithm is focusing too much on fitting the data. In doing so, one can be mislead into a prediction model highly sensitive to noise, which is present in essentially any real application. We refer by **overfitting** when an algorithm sacrifices performance on unseen data in exchange of better fitting on the training data.

At the end of the day, choosing a good learning algorithm can be a challenge, since it will usually depend on the overall context defined by the task at hand. This lack of universality for the choice of the best learning algorithm can be made formal through the *No Free Lunch* theorem [55], which essentially states that there is no algorithm that works well in any situation. Making an assumption about the problem typically means to restrict our search to a **hypothesis class** $\mathcal{F} \subseteq \mathcal{Y}^{\mathcal{X}}$. And from this class we can ask ourselves how well the best learning algorithm does in the worst case scenario.

Example 2.1.1. (Linear Regression)

In linear regression we assume $\mathcal{X} = \mathbb{R}^d$, $\mathcal{Y} = \mathbb{R}$ and a hypothesis class consisting only of linear models

$$\mathcal{F} = \left\{ \hat{f} \in \mathcal{Y}^{\mathcal{X}} : \exists \theta \in \mathbb{R}^{d+1} \text{ s.t. } \hat{f}(x) = \hat{f}_\theta(x) = \theta_0 + \sum_{i=1}^{d} \theta_i x_i, \forall x \in \mathcal{X} \right\}$$

where for each $\hat{f}$ we have a vector of parameters θ associated.

Parameters are real values that control the behavior of a model. If there are finitely many parameters, we say that our model (or hypothesis class) is parametric. In our case θ_0 stands for the model bias and the weights θ_i can be seen as how much influence each feature x_i has on the final prediction. Given a dataset $\mathcal{D} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ with $x_i = (x_{i1}, x_{i2}, \ldots, x_{id})$ we are interested in finding a parameter vector $\hat{\theta}$ that best approximates the relation of y and x by a linear model.

2.2 Supervised Learning

For a more formal discussion on the subject we will assume a probabilistic setting. Consider $Z = (X, Y)$ as a random pair in $\mathcal{Z} = \mathcal{X} \times \mathcal{Y}$ distributed according to an unknown distribution P. This distribution describes the frequency of encountering a particular example in practice. Now, by observing a sequence of i.i.d. random pairs $\{Z_i = (X_i, Y_i)\}_{i=1}^n$ sampled according to P we are interested in constructing a predictor $\hat{f} : \mathcal{X} \times \bigcup_{n=1}^{\infty} (\mathcal{X} \times \mathcal{Y})^n \to \mathcal{Y}$ that makes good predictions for Y from X on samples that are likely to appear in practice.

One way of measuring the performance of a learning algorithm is to choose a **loss function** $\ell : \mathcal{F} \times \mathcal{Z} \to \mathbb{R}^+$ that somehow quantifies the loss of considering a given hypothesis $\hat{f} \in \mathcal{F}$ with respect to an arbitrary example $z = (x, y) \in \mathcal{Z}$.

We define the **risk** or **generalization error** associated to an hypothesis $\hat{f} \in \mathcal{F}$ by

$$R(\hat{f}) = \mathbb{E}_{Z \sim P}[\ell(\hat{f}, Z)]. \tag{2.2}$$

It's important to notice that when the given predictor $\hat{f} = \hat{f}(\cdot, \mathcal{D})$ comes from a learning algorithm and therefore is associated with a sample $\mathcal{D} \sim P^{\otimes n}$ the associated risk becomes a random variable depending on the sample $\mathcal{D}$.

Once we fix a loss function, we have a criterion for a good solution of the supervised problem. That is, we set as the predictor in our hypothesis class $\mathcal{F}$ the one that minimizes the risk value

$$f^* = \operatorname*{argmin}_{\hat{f} \in \mathcal{F}} R(\hat{f}) \tag{2.3}$$

A common choice for loss function in the regression setting ($\mathcal{Y} = \mathbb{R}$) is the squared error loss: $\ell : \mathcal{F} \times (\mathcal{X} \times \mathbb{R}) \to \mathbb{R}^+$, $\ell(\hat{f}, (x, y)) = (y - \hat{f}(x))^2$. In this case the associated risk is just

$$R(\hat{f}) = \mathbb{E}(Y - \hat{f}(X))^2. \tag{2.4}$$

The minimizer over $\mathcal{Y}^{\mathcal{X}}$ in this case is the conditional expectation

$$f^*(x) = \mathbb{E}[Y \mid X = x]. \tag{2.5}$$

To see this, observe that for $W = \mathbb{E}[Y \mid X] - \hat{f}(X)$ we have:

$$\mathbb{E}[W(Y - \mathbb{E}[Y \mid X])] = \mathbb{E}[WY - \mathbb{E}[WY \mid X]] = 0. \tag{2.6}$$

Then our risk satisfies

$$\begin{aligned} R(\hat{f}) &= \mathbb{E}(Y - \mathbb{E}[Y \mid X] + W)^2 \\ &= \mathbb{E}(Y - \mathbb{E}[Y \mid X])^2 + \mathbb{E}[W^2], \end{aligned} \tag{2.7}$$

which attains its minimum when $W = 0$.

Since the distribution P is unknown we can't actually compute the risk function in practice. This is where learning appears We can use the responses of our datasets to estimate the risk function and then use it as a guide to construct a predictor. We define the **empirical risk** as

$$R_n(\hat{f}) = \frac{1}{n}\sum_{i=1}^{n} \ell(\hat{f}, Z_i) \tag{2.8}$$

where the $Z_i = (X_i, Y_i)$ are our given examples.

Then assuming $R_n(\hat{f})$ is a good approximation for the theoretical risk we can obtain an estimator $\hat{f}_n : \mathcal{Z}^n \in \mathcal{F}$ for the supervised problem by choosing $\hat{f}$ that minimizes the empirical risk

$$\hat{f}_n = \operatorname*{argmin}_{\hat{f}\in\mathcal{F}} R_n(\hat{f}) \tag{2.9}$$

We call this strategy **empirical risk minimization** (**ERM**).

In learning theory the usual questions over this settings are:

- (*Prediction*) Does $\hat{f}_n$ predicts well the solution f^*, independent of what the distribution P is, i.e.

$$\forall \varepsilon > 0, \forall P \in \mathcal{P}, \mathbb{P}_P(R(\hat{f}_n) - R(f^*) \geq \varepsilon) \xrightarrow{n\to\infty} 0. \tag{2.10}$$

- (*Estimation*) Does $\hat{f}_n$ actually estimate f^*? i.e. $\hat{f} \xrightarrow{\mathbb{P}_P} f^*, \forall P \in \mathcal{P}$ in some sense.

- (*Inference*) How the difference $\hat{f}_n - f^*$ is distributed?

We introduce $\mathcal{P}$ as a set of probability distributions that may be restricted, due to prior information from the specific problem one is working in. The idea is that our learning procedure should work for any plausible unknown distribution. While those are important questions to have in mind we don't present answers here, but we refer the reader to [52, 53] for results on this topics.

Example 2.2.1. (Linear Regression - Least Squares)
Now, we apply *least squares* to obtain an estimator $\hat{\theta}$ for the linear regression problem, described previously, by minimizing the residual sum of squares

$$\begin{aligned} RSS(\theta) &= \textstyle\sum_{i=1}^{n}(y_i - \hat{f}_\theta(x_i))^2 \\ &= \textstyle\sum_{i=1}^{n}(y_i - \theta_0 - \sum_{j=1}^{d}\theta_j x_{ij})^2. \end{aligned} \tag{2.11}$$

If we consider $\mathbf{X}$ as the $n \times (d+1)$ matrix with each row an input vector plus a 1 in the bias coordinate, and similarly $\mathbf{y}$ as the n-vector with the associated outputs. Then we can rewrite 2.11 as

$$RSS(\theta) = (\mathbf{y} - \mathbf{X}\theta)^T(\mathbf{y} - \mathbf{X}\theta). \tag{2.12}$$

We can minimize this quadratic equation by differentiating it with respect to θ, obtaining

$$\frac{\partial}{\partial\theta}RSS(\theta) = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\theta) \tag{2.13}$$

$$\frac{\partial^2}{\partial^2\theta}RSS(\theta) = -2\mathbf{X}^T\mathbf{X}. \tag{2.14}$$

When $\mathbf{X}^T\mathbf{X}$ is positive definite (it is sufficient that $\mathbf{X}$ have full column rank) we obtain the unique solution

$$\hat{\theta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}. \tag{2.15}$$

Remark. *The choice of least squares for linear regression is somehow justified by the Gauss-Markov theorem that states that the least squares estimator is the unbiased linear estimator with the smallest variance (see [14] 3.2.2).*

2.2.1 Maximum Likelihood

It's somewhat naive to expect that the considered features $X \in \mathcal{X}$ can completely describe the responses $Y \in \mathcal{Y}$ in practice. Therefore we might want to extend our goal of finding a deterministic relationship $Y = f(X)$ by trying to directly approximate the conditional distribution of Y given X.

Consider a parametric setting

$$\mathcal{P}_\Theta = \{P_\theta \in \mathcal{P} : \theta \in \Theta\}, \tag{2.16}$$

where Θ is the set of parameters, usually a subset of $\mathbb{R}^d$. We also assume that there are a σ-finite measure μ on $\mathcal{Z}$ such that $\forall\theta \in \Theta\ P_\theta \ll \mu$ and therefore it assumes a density $p_\theta = \frac{\partial P_\theta}{\partial\mu}$. In this setting and with a sample $Z_1^n \sim P^{\otimes n}$ we define the **likelihood function** as

$$\mathcal{L}_n(\theta) := p_\theta(Z_1^n) = \prod_{i=1}^n p_\theta(Z_i), \tag{2.17}$$

and the **maximum likelihood estimate** (**MLE**) by

$$\hat{\theta}_n := \operatorname*{argmin}_{\theta\in\Theta} \mathcal{L}_n(\theta). \tag{2.18}$$

Essentially, the maximum likelihood principle expects to find a parametric distribution that maximizes the probability of encountering the given sample. We can also see this problem as **ERM** over the parameters set Θ instead of $\mathcal{F}$ by considering the loss function

$$\ell(\theta, Z) = -\ln p_\theta(Z). \tag{2.19}$$

Then, minimizing $\mathcal{L}_n(\theta)$ is the same as minimizing the empirical risk

$$R_n(\theta) = \frac{1}{n}\sum_{i=1}^n \ell(\theta, Z_i) = \frac{1}{n}\sum_{i=1}^n -\ln p_\theta(Z_i). \tag{2.20}$$

Assume that the true distribution P is also absolutely continuous with respect to μ with density p. From the **ERM** analogy we observe that the associated theoretical risk

$$R(\theta) = \mathbb{E}_P[\ell(\theta, Z)] = \mathbb{E}_P[p_\theta(Z)] \tag{2.21}$$

can be rewritten from a simple computation as

$$\begin{aligned} R(\theta) &= \mathbb{E}_P\left[\frac{p_\theta(Z)}{p(Z)}\right] + \mathbb{E}_P[-\ln p(Z)] \\ &= KL(P\|P_\theta) + H(Z), \end{aligned} \tag{2.22}$$

where the first term in the sum is the **Kullback-Leibler (KL) divergent** (or relative entropy) and the second term is the **entropy** of P. Therefore, minimizing the risk function in this case is equivalent to minimize the KL-divergence between P and P_θ over $\theta \in \Theta$.

The usual maximum likelihood setting as presented so far will be more important in the next section when we discuss generative modeling. For now we restate the framework into the supervised case

$$\hat{\theta}_n := \operatorname*{argmin}_{\theta\in\Theta} \frac{1}{n}\sum_{i=1}^{n} -\ln p_\theta(Y_i \mid X_i), \tag{2.23}$$

replacing the previous objective of approximating the distribution $p_\theta(z)$ by the conditional distribution $p_\theta(y \mid x)$ of the output given the input. These setting preserve similar properties, for example the relation

$$\mathbb{E}_P[-\ln p_\theta(Y \mid X)] = \mathbb{E}_X[KL(P(\cdot|X)\|P_\theta(\cdot|X))] + H(Y \mid X) \tag{2.24}$$

where now $H(Y \mid X)$ is the **conditional entropy** of Y given X.

Example 2.2.2. (Logistic Regression)
In logistic regression we are interested in doing classification by modeling the posterior distributions of m classes as linear functions of the features, at the same time that we ensure that it actually models a probability distribution.

We set $\mathcal{X} = \mathbb{R}^d$ as our input space, and $\mathcal{Y} = [m]$ as our output space, with $\Theta = R^{(d+1)\times m}$ we define the parametric posterior by

$$p_\theta(y = i \mid x) = \frac{\exp(\theta_i^T x)}{Z(\theta)}, \forall i \in [m] \tag{2.25}$$

where $Z(\theta)$ is the partition function

$$Z(\theta) = \sum_{j=1}^{m} \exp(\theta_j^T x) \tag{2.26}$$

and we consider x augmented with an extra coordinate for the bias parameter, i.e, we have $\theta_i^T x = \theta_{i0} + \sum_{k=1}^{d} \theta_{ik} x_k$.

To fit the parameters of the logistic model we use the maximum likelihood principle. For a sample of size N given by $S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$ we minimize the associated negative log-likelihood function

$$\begin{aligned} L(\theta) &= -\sum_{i=1}^{N} \ln p_\theta(y_i \mid x_i) \\ &= -\sum_{i=1}^{N} \theta_{y_i}^T x_i + N \ln Z(\theta). \end{aligned} \tag{2.27}$$

The solution of the logistic regression from the maximum likelihood approach has several interesting properties. So before we look into how to solve the optimization problem we'll look at some interesting consequences. First lets look at the gradient of $L(\theta)$, for $c \in [m]$ and $j \in [d]$

$$\begin{aligned} -\frac{\partial L(\theta)}{\partial \theta_{cj}} &= \frac{\partial}{\partial \theta_{cj}} \left(\sum_{i=1}^{N} \ln p_\theta(y_i \mid x_i) \right) \\ &= \sum_{\substack{i=1 \\ y_i = c}}^{N} \frac{1}{p_\theta(y_i \mid x_i)} \frac{\partial}{\partial \theta_{cj}} p_\theta(y_i \mid x_i) \\ &+ \sum_{\substack{i=1 \\ y_i \neq c}}^{N} \frac{1}{p_\theta(y_i \mid x_i)} \frac{\partial}{\partial \theta_{cj}} p_\theta(y_i \mid x_i) \\ &= \sum_{\substack{i=1 \\ y_i = c}}^{N} x_{ij}(1 - p_\theta(c \mid x_i)) - \sum_{\substack{i=1 \\ y_i \neq c}}^{N} x_{ij} p_\theta(c \mid x_i) \\ &= \sum_{\substack{i=1 \\ y_i = c}}^{N} x_{ij} - \sum_{i=1}^{N} x_{ij} p_\theta(c \mid x_i). \end{aligned} \tag{2.28}$$

When we are in a critical point we should have this quantity equal to zero, this implies the following condition for each $c \in [m]$ and $j \in [d]$

$$\sum_{i=1}^{N} x_{ij} p_\theta(c \mid x_i) = \sum_{i=1}^{N} \mathbb{1}\{y_i = c\} x_{ij}. \tag{2.29}$$

Therefore the solution of the maximum likelihood estimation must satisfy the conditions in 2.29, which we call **balance equations**. These conditions are special since the logistic regression classifier can be derived by maximizing the entropy of the posterior distributions constrained on the balance equations. One can solve the constrained optimization problem and check that it returns exactly our proposed distribution. This serves as one way of justifying our choice for classifier, since a distribution with maximum entropy is seen in information theory as the one that encodes the most information, so it should be a good candidate over our balance restrictions.

2.2.2 Gradient descent methods

To solve logistic regression in practice one can try several optimization strategies. Nevertheless we'll be restricting ourselves to gradient descent methods. This is the case be-

cause when working with more complex loss functions and with parameter spaces with higher dimensions, which will be common when working with deep learning methods, gradient descent is usually the most practical choice.

It's crucial to remind that using optimization algorithms for training a learning model differs from a pure traditional optimization task. In most cases the performance measure its defined over a test set of examples that are not experienced during training. This is a way of determining if our model is generalizing for instances outside of the data set. Therefore simply optimizing a loss function over a training set is not enough evidence of a successful learning procedure.

Another important difference is that often finding the global minimum when minimizing over the training set is usually not a good sign, since for models with high capacity this could certainly mean overfitting, i.e. fitting exactly the data without capturing structure for generalization. Therefore, what we would normally be interested in is on finding a good local minimum of the empirical risk that express good generalization compared with the true risk measure. Where here we mean generalization by when the empirical minimum is actually close to the global minimum of the true risk measure.

The classical **gradient descent** (GD) consists on the iteration

$$\theta_{t+1} = \theta_t - \varepsilon \nabla_\theta L(\theta_t) \tag{2.30}$$

where we update the parameters on the opposite direction of the gradient with a **learning rate** $\varepsilon > 0$. Essentially GD uses local information to find a direction where the function most decreases its values. Additionally, two common modifications are to add adaptative learning rates $\{\varepsilon_t\}_t$, i.e. having the size of the steps also varying through time, and to precondition the gradient vector by multiplying it by a matrix H.

We also introduce what we mean by **stochastic gradient descent** (SGD). On each iteration $t \in \mathbb{R}$ we choose uniformly at random an S-set $\mathcal{S} \subseteq [N]$ of indices and form the following gradient over what we call a "minibatch"

$$\hat{L}_S(\theta) = \frac{1}{S} \sum_{i \in \mathcal{S}} \ell(\theta, Z_i),\ \hat{g}_S(\theta) = \nabla_\theta \hat{L}_S(\theta) \tag{2.31}$$

we then perform the iteration with this stochastic gradient

$$\theta_{t+1} = \theta_t - \varepsilon \hat{g}_S(\theta_t). \tag{2.32}$$

Stochastic gradient descent, was proposed as a way of saving computational power when working with large data sets. But it turns out that the noise resultant of the minibatch approach has some unexpected advantages on achieving generalization over the traditional batch approach (full gradient) [6, 16]. The reasons for why this is the case is a current research problem and has attracted attention due to the success of SGD on training deep learning models.

We'll now present a simple experiment using logistic regression and discuss some practical aspects of doing machine learning. Our examples for this experiment are the notMNIST dataset [7], consisting on 10 classes of glyphs, from A to J, extracted from publicly available fonts. This dataset contains a collection of 19k hand-cleaned images

as test set and 500k uncleaned images as train set (see examples bellow). Different from the more classic MNIST dataset, this one is closer to what one would expect in a real application, with more challenging and raw examples, making it a more interesting demonstration of the presented methods.

Figure 2.1: Examples of letter A on notMNIST database

Before fitting any model is important that we observe if the given database is nicely balanced. In our case, since we have each letter organized separately, it is important to randomize their order [3]. When applying a method like stochastic gradient descent in an ordered set, the model can converge to a bad local minima at the beginning of training, For example if it is presented at first mainly with examples of only one class, the model can become stuck. Another precaution, which is not always feasible to take, is to avoid repetitions of examples or intersections between the train and test sets, this should guarantee faster learning rates and accuracy.

In this experiment we use the logistic regression model provided by the machine learning Python library Scikit-learn [39] set to fit the model by stochastic gradient descent on a maximum likelihood setting. We measure performance on the test set using a simple average agreement metric

$$\text{score}(\mathcal{D}_{test}) = \frac{1}{|\mathcal{D}_{test}|} \sum_{(x,y)\in\mathcal{D}_{test}} \mathbb{1}_{\hat{y}=y}. \tag{2.33}$$

The choice of performance metric is often a delicate issue. When designing machine learning applications. In practice, it is not always easy to measure the consequences of a bad prediction. As some decisions might cost time, money, human resources, etc. Nevertheless, our choice of performance metric will usually be of a simpler kind, since our applications are purely illustrative, and there is not much difference of importance on our test set.

Now, by observing the metric values over the number of samples observed during training, we can observe how the model evolves during training. We use a test set of

10k examples on the following results.

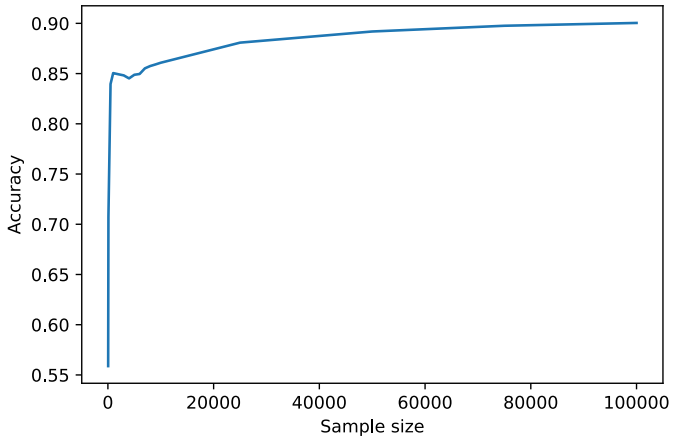

Figure 2.2: Test error performance for classification on the notMNIST dataset using SGD.

We start with 50 samples and go until 100k images, at the end the final score reaches about 90.0%. But we can see from the plot that we pass above 85% before 10k examples and from there on the curve increases very slowly. To measure misclassifications for our trained model we plot the associated confusion matrix, which shows the proportion of examples of one class that were predicted as another class.

	A	B	C	D	E	F	G	H	I	J
A	899	7	3	6	2	8	11	32	14	18
B	10	877	5	29	15	11	19	12	14	8
C	1	2	930	5	19	13	16	3	11	0
D	7	18	5	923	3	11	6	8	8	11
E	5	12	34	4	880	16	16	5	16	12
F	4	4	5	6	11	925	8	3	14	20
G	6	10	33	10	11	18	882	6	16	8
H	14	11	5	3	12	15	9	902	20	9
I	13	4	6	10	20	9	12	10	867	49
J	9	4	4	12	3	13	8	6	22	919

Figure 2.3: Confusion matrix for notMNIST classification test set.

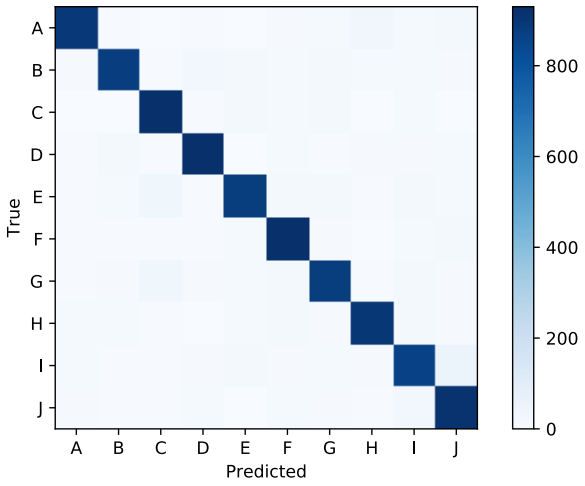

Figure 2.4: Color scale visualization of the confusion matrix.

These are two common ways of evaluating the performance of a machine learning classification problem. Later we'll comment on other possible strategies for this task and compare these results with the ones of other possibly more complex models.

2.3 Generative Modeling

Suppose we are interested in generating samples from a distribution that we don't actually know, but that we have access to a collection of samples from it, this is essentially what generative modeling tries to accomplish. Generative modeling (GM) allows us to dive into distributions that one wouldn't be able to design in practice, involving obscure properties and high complexity. Some real-life applications of generative modeling are

- Synthesizing convincing and photorealistic images by looking at natural images or artwork ([45], [38]).
- Super-resolution, i.e. generating high resolution versions of low resolution images ([33]).
- Image-to-Image translation tasks, like colorization, segmentation, style-transfer, stereo, etc ([26]).
- Modeling time-series for simulations and planning on reinforcement learning methods ([24]).

Differently from the supervised problems we have seen so far, generative modeling poses new challenges. In general, unsupervised learning consists of analyzing unlabelled data with the purpose of extracting some underlying structure. By contrast generative modeling is just one flavor of this class of problems. Generative modeling

its essentially a density estimation problem. More than that, it allow us to access implicit knowledge about the distribution, like drawing samples, creating representations or expressing values associated with the phenomenon at hand.

Before going to more complex and specific models, we'll present the concept of latent variables, which will extend naturally in the subsequent models

2.3.1 Latent Variable Models

Definition 2.3.1. A **latent variable model** (LVM) p is a probability distribution, usually parametric, over two sets of random variables X and H. The first represent the variables observed at learning time, in a dataset $\mathcal{D}$, and also the ones we'll try to estimate the density

$$p_\theta(x) = \mathbb{E}_H[p_\theta(x \mid H)]. \tag{2.34}$$

The variable H represent the ones we refer as **latent variables**. They capture underlying concepts or possible correlations of the observed data. These variables are not visible but are assumed to be present.

Considering these hypothetical latent variables makes learning harder, but give us the capacity of revealing hidden concepts, relations or representations about the problem.

Example 2.3.1. (Mixture models) Let H be a categorical variable, $H \in [m]$, $H \sim \text{Cat}(\pi)$, i.e.

$$\mathbb{P}(H = k) = \pi_k, \forall k \in [m] \tag{2.35}$$

Then the observed variable of the LVM has density

$$\mathbb{P}(X = x) = \sum_{k=1}^{m} \pi_k \mathbb{P}(X = x \mid H = k) = \sum_{k=1}^{m} \pi_k p_k(x). \tag{2.36}$$

In essence a mixture model works by drawing X from $p_H(\cdot)$, where H has being chosen from the discrete distribution π.

A common example of a mixture model is a Gaussian mixture, in this case each distribution p_k is Gaussian

$$p_k(x) = \mathcal{N}(x \mid \mu_k, \Sigma_k) \tag{2.37}$$

Mixture models are applied in different ways in machine learning. Essentially what is assumed is that the given data comes from distinct distributions and one wants to recognize where those are coming from. This can be seen as a missing data problem, as classification or as a data clustering task. Usually, this models are trained by maximum likelihood estimation, but the presence of latent variables requires new tools. The usual method is expectation maximization or similar strategies.

The following framework will be our key idea when building generative models. Consider the latent variables as coming from a known distribution $H \sim p_H$ over a

space $\mathcal{H}$, which we'll call the latent space. Then a **generator function** $g : \mathcal{H} \to \mathcal{X}$ generates (or samples) X given H

$$X = g(H). \tag{2.38}$$

Here we interpret H as the randomness of the process. We are interested in learning a good generator that maps the distribution p_H into the unknown data distribution. We can see this through a special case.

Suppose we want to sample a random variable $X \in \mathbb{R}$ with cumulative distribution function (cdf) $F(x) = \mathbb{P}(X \leq x)$. If such function is invertible we can build a generator by first taking $H \sim U(0, 1)$ and $g = F^{-1}$. Then, this implies $g(H) \sim X$ since their cdf coincide. This method for sampling is known as *inverse transform sampling* and is a direct way (without ML) of solving the problem when we have access to F and we are able to compute its inverse. In a more general case when we only have samples from the desired distribution we must try to build a generator by learning from this dataset.

A simple example of a generator function is a **linear factor model** (LFM), characterized by two elements: a linear relation with the latent variables

$$X = g(H) = WH + b \tag{2.39}$$

where W and b are parameters of g, and a factorial latent distribution

$$p_H(h) = \prod_i p(h_i). \tag{2.40}$$

For example, in order to generate samples from a Gaussian distribution $X \sim \mathcal{N}(\mu, \Sigma)$ one can set the latent distribution p_H as a standard Gaussian $\mathcal{N}(0, I)$, and take the model parameters as $b = \mu$ and W as the Cholesky decomposition of the covariance matrix Σ. This is a very simple example of a LFM, but the whole setup can be applied for more interesting methods. Some well known examples are principal component analysis (PCA) and sparse coding. Typically these methods also apply to specific training routines that might contain regularization procedures that can better constrain the model characteristics.

In a more general context we consider a class of differentiable generators, usually a parametric family $\mathcal{G} = \{g_\theta \in \mathcal{H}^{\mathcal{X}}, \theta \in \Theta\}$. Moreover if a generator $g \in \mathcal{G}$ also has a differentiable inverse, then a change of variables give us the relation

$$p_H(h) = p_{X_g}(g(h)) \left| \det \left(\frac{\partial g}{\partial h} \right) \right| \tag{2.41}$$

where $X_g = g(H)$, which by the above formula has an implicit distribution

$$p_g(x) = p_{X_g}(x) = \frac{p_H(g^{-1}(x))}{\left| \det \left(\frac{\partial g}{\partial h} \right) \right|} \tag{2.42}$$

In those cases, where the class of generators we are interested in is simple enough so that the above explicit form can actually be computed, we can consider doing maximum

likelihood estimation to fit the parameters with respect to a given sample set, as we have already seen on the previous section. That usually will not be the case, i.e. most generator models with high expressivity capacity tend to be very hard or to expensive to compute. Therefore we will distantiate ourselves from the more traditional risk minimization approach and look for indirect means of learning the generator g_θ.

Chapter 3

Neural Networks

3.1 Feedforward Neural Networks

The study of (artificial) neural networks in machine learning began with the purpose of emulating how the human brain processes information [29] by building a collection of interconnected computational units that roughly model the behavior of a neuron. Nowadays it is known that most modern architectures of neural networks bear little resemblance to a real biological system of neurons. Nevertheless, these models remain popular as parametric models for various purposes.

We define our main component, the **neuron**, as a parametric function

$$\phi : \mathbb{R}^k \to \mathbb{R} \tag{3.1}$$

given by $\phi(x) = \sigma(w^T x + b)$, the composition of a linear operation and an activation function $\sigma : \mathbb{R} \to \mathbb{R}$, usually non-linear,

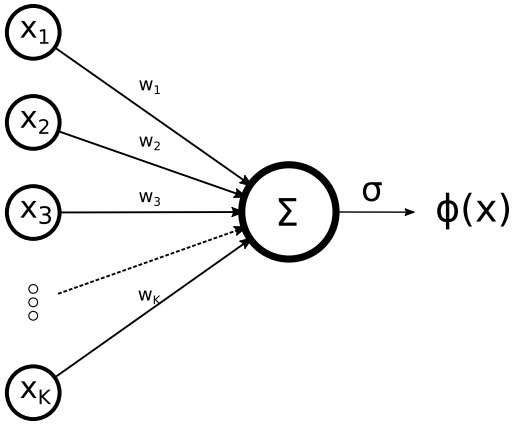

Figure 3.1: A structural representation of an artificial neuron.

When the activation function is the step function $\sigma(x) = \mathbb{1}_{\{x \geq 0\}}$ we have the classical Rosenblatt's perceptron, the first artificial model for a neuron [46]. As a binary classificator this model learns a separating hyperplane in $\mathbb{R}^k$ for the data points.

Next, we want to compose this neurons to build richer models that can learn all sorts of functions. Before we state our main structure, the feedforward network, we give a general definition for this concept

Definition 3.1.1. A neural network is a directed graph $\mathcal{N} = (V, E)$, where each vertex $v \in V$ is labelled with a neuron

$$\phi_v : \mathbb{R}^k \to \mathbb{R} \tag{3.2}$$

with $k = indeg(v)$ In addition, $\mathcal{N}$ contains at least one vertex with indegree zero (an input node) and one with outdegree zero (an output node).

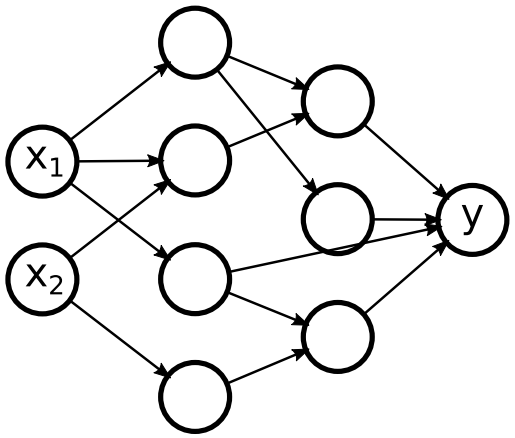

Figure 3.2: A neural network drawn as a graph.

The above graph can also be seen as a weighted graph where the weights come from the linear operation each neuron perform.

When the graph $\mathcal{N}$ is finite and acyclic we call it a **feedforward neural network** (FNN), but more specifically we use this name to refer to the typical FNN architecture which has a forward full connection between layers, i.e. $\mathcal{N} = (V, E)$ can be written as

$$V = H_0 \cup H_1 \cup \cdots \cup H_L, \text{ and} \tag{3.3}$$

$$E = \{(u, v), u \in H_{i-1}, v \in H_i, \text{for some } i \in [L]\} \tag{3.4}$$

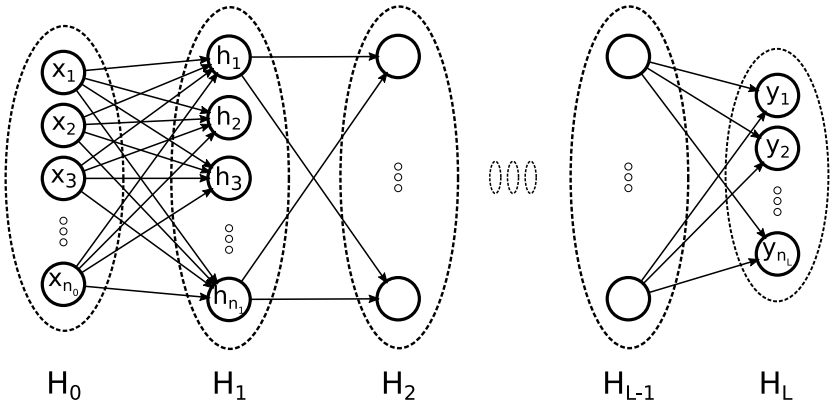

Figure 3.3: The feedforward neural network.

Many important models use different architectures, like residual networks ([20]) that make connections between distant layers and recurrent neural networks ([29] chapter 10) which allow cycles in the graph, but for our purposes this classical structure will be enough.

We can deduce the parametric function given by the above FNN by composing the neurons layer by layer, if $h^{(0)} = x$ is our input vector then for $i \in [L]$ we set

$$h^{(i)} = \sigma^{(i)}(W^{(i)T} h^{(i-1)} + b^{(i)}) \tag{3.5}$$

where $h^{(i)}$ corresponds to the vector of values labelling layer H_i, the lines of $W^{(i)}$ and $b^{(i)}$ are the parameters associated with the linear operation in each neuron in H_i and here $\sigma^{(i)}$ is applying the activation function coordinate by coordinate (it's assumed that all neurons in a given layer use the same activation). Finally, the induced parametric model is given by the final output $f(x, \theta) = h^{(L)}$, where θ encompasses all parameters in the network.

The universal approximation theorem for FNN's proved in [23] states that given enough neurons in at least one hidden layer (layer with positive in and out degrees) with a sigmoidal activation function (continuous and increasing in $[0, 1]$) the FNN can well approximate any Borel measurable function. Recent results ([10]) also give evidence of benefits of using more layers when building FNNs.

3.2 Deep Neural Networks

Deep neural networks (DNN) are networks with many layers, were for a long time overshadowed by other ML models because they were considered hard to train and less effective. In recent years a combination of greater computational power, larger datasets, and new regularization methods has made deep networks feasible. Their results surpass those of other methods in many learning problems. The term **deep learning** will, as it's usually done, refer to any learning algorithm that uses DNNs in its core.

One important tool when learning with a DNN is **backpropagation**. This is an efficient procedure for computing gradients of functions computed by neural networks.

Consider a cost function $c : \mathbb{R}^{|H_L|} \to \mathbb{R}$ and let's compute the gradient of $C(x, \theta) = c(f(x, \theta))$ where f is given as in 3.5.

If θ_ℓ are the parameters of layer $\ell \in [L]$ we can use chain rule to obtain

$$\frac{\partial C}{\partial \theta_\ell} = \sum_{j=1}^{|H_\ell|} \frac{\partial C}{\partial h_j^{(\ell)}} \frac{\partial h_j^{(\ell)}}{\partial \theta_\ell} =: \frac{\partial C}{\partial h^{(\ell)}} \cdot \frac{\partial h^{(\ell)}}{\partial \theta_\ell}. \tag{3.6}$$

the gradient $\dfrac{\partial h^{(\ell)}}{\partial \theta_\ell}$ can be directly computed from the definition while the values of $\delta^\ell = \dfrac{\partial C}{\partial h^{(\ell)}}$ can be obtain through a recursion on the oppose direction of the network:

- $\delta^L = \dfrac{\partial C}{\partial h^{(L)}}$ is the gradient of the cost function c.
- For $\ell \in [L-1]$ we can compute

$$\delta^\ell = \frac{\partial C}{\partial h^{(\ell+1)}} \cdot \frac{\partial h^{(\ell+1)}}{\partial h^{(\ell)}} = \delta^{\ell+1} \cdot \frac{\partial h^{(\ell+1)}}{\partial h^{(\ell)}} \tag{3.7}$$

  where the values of $\dfrac{\partial h^{(\ell+1)}}{\partial h^{(\ell)}}$ can also be computed directly.

Finally the backpropagation algorithm consists on computing the values of δ^ℓ using the above recursion from L to 1 and consecutively we are able to get the gradient values $\dfrac{\partial C}{\partial \theta_\ell}$.

The way backpropagation on neural networks is performed looks exactly like a feedback signal running on the opposite flow of the directed graph. Computing the loss function gradient by this procedure allow us to learn by stochastic gradient descent, which can be seen as updating the parameters by using the feedback of when prediction is performed by the network on training data.

Example 3.2.1. Let us apply DNNs to address the same problem as in example 2.2.2. For a classification problem on $[m]$ classes we can as in the logistic regression model the posterior distribution by

$$p_\theta(\cdot \mid x) = \text{softmax}(f(x, \theta)) \tag{3.8}$$

where the softmax operator is given by

$$\text{softmax}(h)_i := \frac{e^{h_i}}{\sum_{j=1}^{m} e^{h_j}}. \tag{3.9}$$

Then, we can use SGD to minimize a loss function given by the negative log-likelihood

$$L(\theta) = -\sum_{i=1}^{N} \log p_\theta(y_i \mid x_i) \tag{3.10}$$

where $\{(x_i, y_i)\}_{i=1}^{N}$ is our set of training examples.

Now, we return to the notMNIST dataset from our first experiment, we train a FNN by maximum likelihood with again the help of stochastic gradient descent computed by backpropagation. For such a task we used the TensorFlow library [1], we set a FNN with a hidden layer of 2048 neurons and a out layer that maps to 10 outputs neurons that are feeded to the sofmax layer as described above. The activation functions used after the hidden layer are rectified linear units (ReLU), $\sigma(x) = \max(0, x)$, which are perhaps the most frequently used activation function nowadays.

We observe that the activation function we are using is not actually differentiable, which goes against the hypothesis for backpropagation. Nonetheless, this can be treated by taking subgradients when necessary. For a real function $f : U \subset \mathbb{R}^n \to \mathbb{R}$, a vector u is called a **subgradient** at a point $w_0 \in U$ if for any $w \in U$ one has

$$f(w) - f(w_0) \geq \dot{u(w - w_0)}. \tag{3.11}$$

In a sugradient descent method one then chooses one of this directions to update the parameters. In our case the only point where ReLU is not differentiable is at zero, which makes the process looks just like gradient descent most of the time.

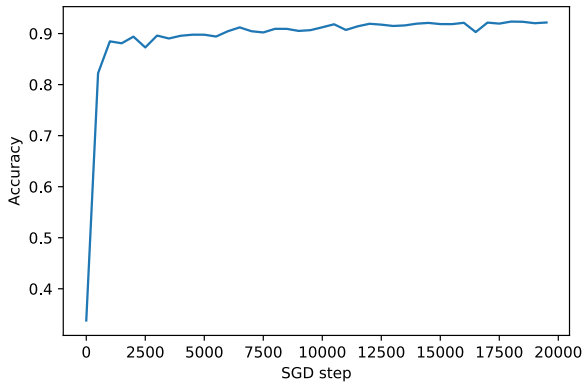

Figure 3.4: Test set accuracy over SGD steps of a simple FNN with one hidden layer on the notMNIST dataset.

The above figure shows our accuracy results (over the test set) on the notMNIST dataset during training, i.e. over the number of steps of SGD, using a minibatch of size 100. We obtained after 20k steps an accuracy of 92%, which is just an slight improvement over the previous experiment. Here we choose to not extend our analysis since we'll repeat it in the next section using a FNN with a more suitable structure.

It's important to emphasize that even when a DNN have the capacity to represent our desired function, a lot can still go wrong during training if proper measures aren't taken. Since the loss functions of deep models are almost always non-convex, gradient-based methods do not give much guarantee of convergence, which can be sometimes misleading. Another fact is that DNNs are highly prompt to overfitting due to high

capacity of the network. Also in typical architectures the dimension of the space of parameters is sometimes larger than the size of training set, which means the network is even capable of memorizing the labels of the whole training data set [59].

3.3 Convolutional Neural Networks

In computer vision and image processing, the use of **feature descriptors** is the classic approach when analyzing visual information. A feature descriptor is any procedure that can extract information (features) from data which is relevant for solving a given task. Typically one could apply a collection of known sophisticated filters and image operations and get in return the position of corners, edges, blobs, etc. But for more specialized applications we might be interested in a combination of descriptors that can identify some specific information. For example, in a face recognition task we could want a descriptor that can find pairs of eyes in a photo, or the aspect of someons face. This specialized descriptors are very hard to build if we are hand-engineering them ourselves. Machine learning is a natural tool for this type of problems, we use data to learn parametric descriptors that can extract the most relevant features for our task. **Convolutional neural networks** (**CNNs**) can be seen as a specialized FNN that are able to learn specialized filters. Using a shared-parameter scheme we turn our typical layer into a sequence of convolutional operations that translates into paramatrized filters that can be learned from data as usual.

Despite the fact that CNNs are a direct extension of image processing filters, their creation is actually biologically-inspired. The early work of Hubel and Wiesel on the visual cortex of cats [25] showed that the cells in this region had a special tiling structure. Such arrangement is able to partition the visual field in smaller sub-regions. Giving this cells the capacity of acting as local filters that exploit the spatially correlation on natural images. This study inspired the work of Fukushima on the first proposed model for CNNs called the Neocognitron [15]. Latter on, similar models appeared, where the LeNet-5 [31] stands out by being the first one really trained from data.

When working with very high dimensional data, like images, supervised learning tends to pose a bigger challenge, and the only way we can expect to learn related functions is by assuming the presence of strong regularity properties. This may mean to expect that the high dimensional manifold where our data lives can actually be represented on a reduced dimension version, allowing learning from fewer examples. Deep convolutional neural networks (DCNN) have shown remarkable results on problems of that nature ([29], [28], [20]). Intuitively DCNNs are able to progressively contract and linearize the input space without hurting the separation properties of the estimator function. They do it by learning local symmetries on the input space allowing attention independent of operations like translation and scaling.

The usual way of filtering an image or a discrete signal is to convolute such object with a kernel. The discrete convolution of h and K defined in $\mathbb{Z}^d$ can be defined as

$$(h * K)(u) = \sum_{v \in \mathbb{Z}^d} h(v)K(u - v), \forall u \in \mathbb{Z}^d. \tag{3.12}$$

For restricted regions, like images for example, when h and K are not defined in the

whole $\mathbb{Z}^d$, we need to extend the regions somehow. This extension is what we refer as padding. In most cases padding is done by adding zeros in all missing entries, but different strategies could also be useful (continuous padding, reflected padding).

Given a 3D array $h_0 \in \mathbb{R}^{n_0 \times m_0 \times \ell_0}$ the typical convolutional layer consists on a sequence of kernels $\{K_1^{(1)}, K_2^{(1)}, \ldots, K_{\ell_1}^{(1)}\}$ each of dimensions $q \times q \times \ell_0$, for an odd q. The output of this layer is a $n_0 \times m_0 \times \ell_1$ array h_1 given by convoluting h_0 with each kernel

$$h_1(i,j,\ell) = \sum_{r=1}^{n_0} \sum_{s=1}^{m_0} \sum_{t=1}^{\ell_0} h_0(r,s,t) K_\ell^{(1)}(i-r, j-s, t), \tag{3.13}$$

where here the indices on our kernels are centered at 0, i.e. $K(0,0,\ell)$ is the $\left(\left\lfloor \frac{q}{2} \right\rfloor, \left\lfloor \frac{q}{2} \right\rfloor, \ell\right)$ entry in the formula above.

For data structures with different dimensional setting, convolutional layers can also be derived much like in the 3D case. Each layer has the purpose of decomposing the input signal in more refined versions, that can accentuate different aspects. In some way, it strips down the input of unnecessary information and return only what is truly necessary for the task at hand.

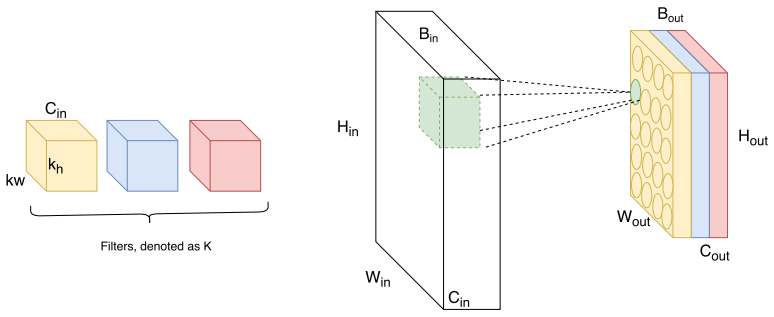

Figure 3.5: Illustration of a convolutional layer ([35]).

Convolutional layers are commonly followed by sampling operations. This procedures help reducing the input dimension, discarding what is considered unnecessary information. More precisely, they can be usually be seen as a way of changing the discretization sampling rate, when seeing data as a signal. In CNNs these are performed mainly in two ways: adding a stride to the convolutional operations or adding a pooling operation.

Stride controls how the filters shift through the input volume. When the stride is 1 we have the usual convolution as defined above. For a stride bigger then one the size of the input shrinks, since the kernel takes fewer steps then the length of the dimension it shifts on. To compute the coordinate (i,j) in a 2D array we look to the neighbourhood of the input centered at $(1+(i-1)S, 1+(j-1)S)$. This means we have already taken i steps of size S and j of size S in our convolutional operation (The stride in each dimension can be taken differently). If the array had dimensions $n \times n$ and the

kernel $q \times q$, with a padding of size P, the output dimension is $n_1 \times n_1$ where

$$n_1 = \frac{n - q + 2P}{S} + 1. \tag{3.14}$$

Pooling is a down-sampling operation (usually nonlinear). We partition the input region in perhaps overlapping rectangles and apply a scalar function that for each sub-region. Two common operations on CNNs are max pooling and average pooling. Where as the names suggest, they return the maximum or the average, respectively, of all values in the a region. Strides can also be seen as a pooling operations following a convolutional layer by taking the center of each region for some sparser partition.

A convolutional layer can also be seen as regular FNN layer where the neurons share parameters, in fact the weights from H_0 to any neuron in H_1 is the same for all neurons, fixed the last dimension, but coming from different sets in H_0, i.e. they share this parameters that are actually the ones coming from the given set of kernels.

A deep convolutional neural network combines both convolutional and fully connected layers and typically also includes pooling operations (up and down sampling) and regularization procedures like batch normalization and dropout ([29] chapter 7).

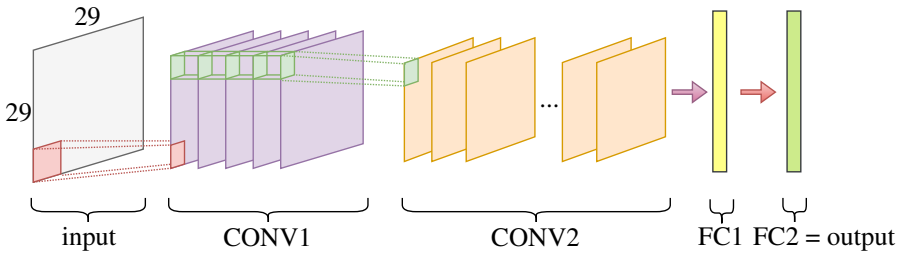

Figure 3.6: Illustration of a CNN architecture ([48]).

Example 3.3.1. Now we return to the notMNIST classification problem from examples 2.2.2 and 3.2.1, this time we trained a convolutional neural network to solve this task. Our architecture consist of two convolutional layers with 16 kernels 5×5 each with maxpooling and one hidden fully connected layer with 1024 neurons, finishing with a linear layer that passes through the usual softmax structure.

Again we used the Tensorflow library [1] for building and training our convolutional network. This time, we also count with the Tensorboard tool to visualize our accuracy and evolution of the network parameters. The following plot shows the accuracy of the network on the test set over the training steps (steps of SGD). We show next the values of the loss function (cross entropy) during training.

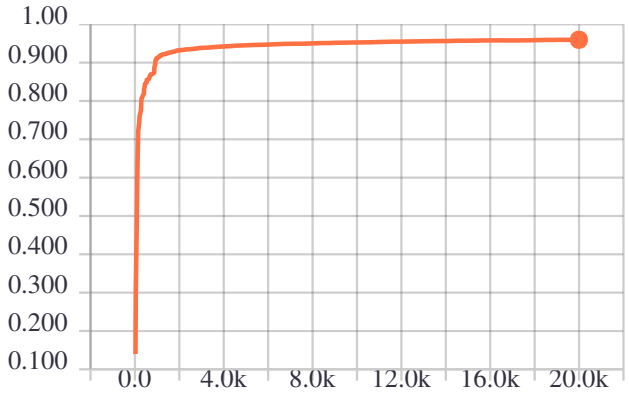

Figure 3.7: Accuracy on test data over 20k training steps.

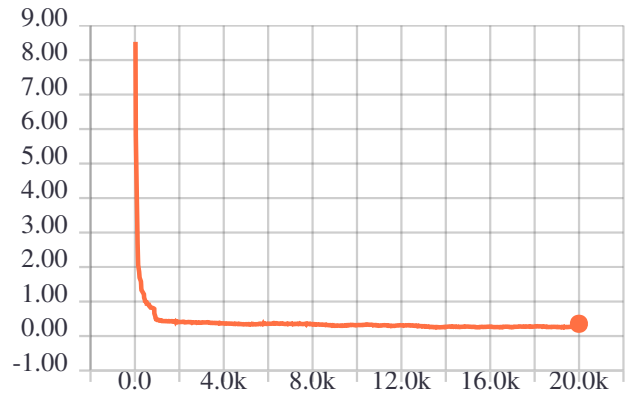

Figure 3.8: Values of loss function on the training set over 20k training steps.

We observe on the results above figures that our results reached an accuracy on the test set of 95%. Better results can be obtained (up to 98%) if we optimize our hyperparameters and perhaps adding one or two layers. Here we do not focus on how to do such tasks, since they are usually done in a exhaustive and empirical manner which is out of our scope in this example. We obtained better results with CNNs than with the regular FNNs; this is somehow expected (experimentally), when working with visual data as we had observed before.

Chapter 4

Generative Adversarial Networks

The breakthrough idea of generative adversarial networks (GANs), proposed in [18], gives a novel strategy for training neural networks for generative modeling: an adversarial strategy where a pair of networks are trained by competing with each other. The competition allows feedback without the need of annotated data. If successful, it is capable of modeling implicitly the data distribution. This emerging technique is very promising because of its results on high dimensional distributions, and for having a less expensive training routine if compared with previous unsupervised learning methods like variational autoencoders, Boltzmann machines and others [17].

In our studies we chose to verify the extensions of such technique and to apply in several experiments that we think express well its capabilities. We begin with a theoretical vision for this idea and we slowly build to some interesting applications and also some more heuristic concepts that arise with this topic.

4.1 Adversarial Training

Suppose we want to synthesize an image, i.e. sample from a distribution of images. This could be the distribution where a collection of images is supposed to come from or maybe a distribution that reproduces well the characteristics and structure of a given texture sample. Traditionally, this task is addressed either by matching on hand-engineered statistics (very common on texture synthesis) or by fitting parametric generators (distributions) [54, 43]. The latter is usually done only on very simple models, since, as we have seen (section 1.2), fitting generators directly by MLE is not feasible for more general parametric models, like neural networks for example, because of the associated inverse problem.

In this section we introduce the idea of **adversarial training**, the core idea of GANs, where we see a way of doing estimation from a competition based objective.

In our text we'll conceptualize adversarial training as a setting where a collection of machines learn together by pursuing competing goals. This means our framework

should allow learning from the feedback of adversarial machines. As a concrete example, we can think of a game between a forger and an expert: the first tries to fool the latter, several times, and in the process the forger learns to produce more believable pieces, while the expert learns how to better spot them. This heuristic essentially explains our main strategy for generative modeling through adversarial training, as we'll explain later in more detail.

The main characteristic that sets apart adversarial training from the standard kind is that learning works by finding a Nash equilibrium in a game between two or more machines instead of the usual optimization problem we are used to in more traditional training routines. To each player i we associate an strategy θ_i and a cost function $c_i(\theta) = c_i(\theta_1, \theta_2, \dots, \theta_n)$ that also depends on all other strategies. Every player then wishes to minimize their cost while only being able to control its own strategy θ_i. A **Nash equilibrium** here is then a special collection of strategies θ such that for each $i \in [n]$

$$c_i(\theta) \leq c_i(\tilde{\theta}_i; \theta_{-i}), \tag{4.1}$$

is satisfied for all possible choices of $\tilde{\theta}_i$ and where θ_{-i} denotes θ minus coordinate i. Essentially what this says is that once other players have fixed their strategies, it is not a good idea to change yours. Soon we'll see how looking for a equilibrium can be interesting to solve the generative problem we are interested.

The original strategy for generative modeling ([18]) consists of a two-player game where a generative model is pitted against a discriminative model. The discriminative one learns from data and must tell if a sample in fact comes from the data distribution or if it is a fake, i.e. if it came from the generator. Meanwhile the generative model learns to deceive the discriminator, and by doing so it must learn to approximate well the data distribution. The goal is to establish a game such that in the equilibrium state the generator's distribution and the true data distribution cannot be told apart by the discriminator. Training should therefore use data to improve both players strategies in the direction of such equilibrium.

To properly set the generative game we associate to each player a function as their strategy. The generator is given by a function $G : \mathcal{Z} \to \mathcal{X}$, which will work as a mapping between distributions in the same way we have seen before for latent models in section 2.3. We call by p_{data} our goal distribution defined in $\mathcal{X}$, p_Z the latent distribution in $\mathcal{Z}$ and p_G as the distribution induced by the map G. The discriminator is a function $D : \mathcal{X} \to [0, 1]$ associating a probability to each point in $\mathcal{X}$. Later we'll consider both functions as parametric, and the game will then be defined over this parameters, but for the present purposes of looking into a Nash equilibrium we need to consider both functions in their most general form.

The role of the discriminator D in our game is to tell the probability of a given sample coming from the true distribution p_data instead of the fake one p_G. Therefore, in this direction the cost function chosen for the player D is given by the negative log-likelihood loss (see 2.2.1) for such binary classification objective,

$$c_D(G, D) = -\frac{1}{2}\left(\mathbb{E}[\log D(X)] + \mathbb{E}[\log\left(1 - D(G(Z))\right)]\right), \tag{4.2}$$

where $X \sim p_{data}$ and $Z \sim p_Z$. To clarify the link between the classification problem

we just told and the above cost function we can take

$$\tilde{X} = YX + (1-Y)X_G, \tag{4.3}$$

for auxiliar $Y \sim Ber(1/2)$ indepentent of both X and X_G. Then the cost function comes from the negative log-likelihood estimation for the conditional distribution $\mathbb{P}(Y = y|x)$, where Y labels x as 1 if it comes from p_{data} and 0 if it comes from p_G, in fact a simple computation gives

$$\begin{aligned} c_D(G, D) &= \mathbb{E}\left[-\log\left(\mathbb{P}(Y|\tilde{X})\right)\right] \\ &= \mathbb{E}\left[-\log\left(D(X)^Y(1 - D(X_G))^{1-Y}\right)\right]. \end{aligned} \tag{4.4}$$

Now, for the simplest case where we have a **zero-sum game**, i.e. where the cost functions of all players always sum to zero, the cost function for the generator is consequently given by

$$c_G(G, D) = -c_D(G, D). \tag{4.5}$$

In this case we have both costs tied up and the game can be summarized into a value function

$$v(G, D) = \mathbb{E}[\log D(X)] + \mathbb{E}[\log(1 - D(X_G))]. \tag{4.6}$$

For zero-sum games the Nash equilibrium coincides with the solution for a minimax problem of the value function above. For this reason we also call this type of game a minimax game. This equivalence due to von Neumann's minimax theorem (see [27]) and even the existence of the Nash equilibrium doesn't come for free without some hypothesis about the space where G and D live in, this will be easier to guarantee later when we move to the parametric setting.

From the last comment we finally set our generator of interest as the solution of the following minimax problem

$$G^* = \operatorname*{argmin}_G \max_D v(G, D). \tag{4.7}$$

Now, we still haven't argued why an equilibrium for this game should induce our desired distribution p_{data}. We claim that in this broad setting it does, and therefore an approximation for G^* could also induce a good approximated distribution. The claim will be a consequence of the following two propositions.

Proposition 1. *For G fixed, the optimal discriminator, i.e. the one maximizing $v(G, D)$ is given by*

$$D^*_G(x) = \frac{p_{data}(x)}{p_{data}(x) + p_G(x)}. \tag{4.8}$$

Proof. Observe that the value function can be written as

$$v(G, D) = \int_{\mathcal{X}} p_{data}(x)\log(D(x)) + p_G(x)\log(1 - D(x))dx. \tag{4.9}$$

For fixed p and q the function $y \to p\log(y) + q\log(1-y)$ achieves its maximum in $[0,1]$ at $y = \frac{p}{p+q}$. Therefore, the value function is maximized by the proposed function, which is unique since $v(G,D)$ is concave on D (Considering that they only need to be defined on $Supp(p_{data}) \cup Supp(p_G)$). □

Proposition 2. *The solution for the minimax problem in 4.7 is achieved if and only if* $p_{G^*} = p_{data}$.

Proof. We'll rewrite our value function using statistical metrics. For that we recall some definitions.

For density functions p, q defined in the same space Ω

$$KL(p \parallel q) = \int_{\Omega} \log \frac{p(\omega)}{q(\omega)} p(\omega) d\omega \tag{4.10}$$

is the **Kullback-Leibler divergent**, or relative entropy, between p and q.

Setting the mixture $M = \frac{p+q}{2}$ we can also define the **Jensen-Shannon divergence**

$$JS(p \parallel q) = \frac{1}{2}\left[KL(p \parallel M) + KL(q \parallel M)\right] \tag{4.11}$$

which is essentially a symmetrized version of the KL-divergence that induces a metric (the square root of JS forms a metric).

Now, the operation $v(G, D_G^*)$ can be seen as

$$\begin{aligned} v(G, D_G^*) &= \mathbb{E}\left[\log \frac{p_{data}(X)}{p_{data}(X) + p_G(X)}\right] + \mathbb{E}\left[\log \frac{p_G(X_G)}{p_{data}(X_G) + p_G(X_G)}\right] \\ &= -\log 4 + \mathbb{E}\left[\log \frac{2p_{data}(X)}{p_{data}(X) + p_G(X)}\right] + \mathbb{E}\left[\log \frac{2p_G(X_G)}{p_{data}(X_G) + p_G(X_G)}\right] \\ &= -\log 4 + KL\left(p_{data} \,\middle\|\, \frac{p_{data} + p_G}{2}\right) + KL\left(p_G \,\middle\|\, \frac{p_{data} + p_G}{2}\right) \\ &= -\log 4 + JS(p_{data} \parallel p_G). \end{aligned} \tag{4.12}$$

This means the problem of finding the generator of minimum value is equivalent to finding the p_G that minimizes $JS(p_{data} \parallel p_G)$. Nevertheless, JS is nonnegative and is zero only when $p_{data} = p_G$ (see [50]), so the proposition follows. □

Remark. *From the last proposition we saw that the out minimization of the minimax game considering an optimal discriminator was equivalent to minimizing the the Jensen-Shannon divergent between the induced distribution* p_G *and* p_{data}*. This was the case due to our choice of cost functions. Under different choices similar scenarios can be found for other metrics.*

Another choice of cost function c_D *for the discriminator player that has become popular recently is based on the Wasserstein metric. The paper where the idea originated [2] also comment on other possible metric choices, and argues the superiority of the Wasserstein over the others. In theory using this metric should improve training stability and in practice this has been shown to be case.*

We showed that the minimax game 4.7 is solved for G^* mapping p_Z into p_{data}, while D^* becomes irrelevant since it is just the constant function equals to $1/2$, rather than a good classifier for the dataset as one could also naturally wrongly assume. Also, such equilibrium is obtained without restricting the spaces where G and D belong and therefore some sort of regularity would be necessary to argue that a good approximation of this model is actually useful in practice. Nevertheless, since we'll be most of the time working with a continuous non-convex game, guarantees of this kind are hard to get.

Remark. *In our adversarial training (AT) description we have only addressed the case of games with two players. While applications of AT for a large number of players is unknown for us, a few new models that follow the same spirit as the described AT with more then two players have become popular recently [61, 4]. We briefly describre one of these ideas, the one from [61], which envolves four players.*

Suppose we want to build a map between two distributions which might come from two distinct datasets. Let's say $X \sim p_X$ and $Y \sim p_Y$, respectively defined in $\mathcal{X}$ and $\mathcal{Y}$, represent random elements of this data distributions. Then, our main goal is to approximate an operator $T : \mathcal{X} \to \mathcal{Y}$ such that $Y = T(X)$.

To form the adversarial game for this translation we also define $R : \mathcal{Y} \to \mathcal{X}$ to model the inverse operation T^{-1}, and two descriptors $D_X : \mathcal{X} \to [0, 1]$ and $D_Y : \mathcal{Y} \to [0, 1]$ serving a similar role as our usual descriptor.

This functions (T, R, D_X, D_Y) are our four players and their associate game can be summarized from the following costs

- *T must fool D_Y*

$$\mathcal{L}(T, D_Y, X, Y) = \mathbb{E}\left[\log D_Y(Y)\right] + \mathbb{E}\left[\log\left(1 - D_Y(T(X))\right)\right]. \tag{4.13}$$

- *R must fool D_X*

$$\mathcal{L}(R, D_X, Y, X) = \mathbb{E}\left[\log D_X(X)\right] + \mathbb{E}\left[\log\left(1 - D_X(R(Y))\right)\right]. \tag{4.14}$$

- *Cycle consistency between T and R*

$$\mathcal{L}_{cyc}(T, R) = \mathbb{E}\left[\|R(T(X)) - X\|\right] + \mathbb{E}\left[\|T(R(Y)) - Y\|\right] \tag{4.15}$$

The problem we are interested is then essentially again descriptors versus generators, in our case translators

$$T^*, R^* = \operatorname*{argmin}_{T,R} \max_{D_X, D_Y} v(T, R, D_X, D_Y), \tag{4.16}$$

where the value function v is given by

$$v(T, R, D_X, D_Y) = \mathcal{L}(T, D_Y, X, Y) + \mathcal{L}(R, D_X, Y, X) + \mathcal{L}_{cyc}(T, R) \tag{4.17}$$

*Such procedure was first proposed in [61] and was called **Image-to-Image translation** since it was supposed to be applied on image datasets. Generalizations of this ideas and the original one can be done to obtain other kinds of applications, usually with the purpose of building maps between distributions, latent or coming from datasets.*

So far we have seen that the adversarial strategy promises a solution for the generative problem. Next we start developing GANs, which will represent a practical way of solving the adversarial problem by using neural networks.

4.2 Generative Adversarial Networks

First, let's rewrite the already presented adversarial problem in the neural networks setting. Consider two differentiable feedforward neural networks, $G(\cdot) = G(\cdot, \theta^G)$ and $D(\cdot) = D(\cdot, \theta^D)$ modeling respectively the generator and discriminator from our original explanation, where θ^G and θ^D denote the parameters from both models. The same minimax game setting we stated before will be assumed here with the difference that the game strategies are now taken by the parameters $(\theta_G, \theta_D) \in \Theta^G \times \Theta^D$ of our networks. The value function from our 2-player minimax game can then be restated as

$$v(\theta^G, \theta^D) = \mathbb{E}\left[\log D(X)\right] + \mathbb{E}[\log\left(1 - D(G(Z))\right)]. \quad (4.18)$$

And the minimax problem is also restated in function of the parameters

$$\theta^G_* = \underset{\theta^G}{\operatorname{argmin}} \max_{\theta^D} v(\theta^G, \theta^D). \quad (4.19)$$

Now that we have moved to a parametric setting the idea is to take advantage of our networks differentiability to use gradient-based methods to improve the strategies of each of our players in the direction of the desired equilibrium. Therefore, the standard procedure for training GANs utilizes the traditional backpropagation routine alternating between optimizing the discriminator D and the G. The training routine, as originally proposed, is summarized in Algorithm 1.

Intuitively, training works by playing the game and improving the stragies from the obtained results. Each player makes their moves (forward propagation) either generating or evaluating samples. Then feedback is given to each depending on their performance (backpropagation). After many rounds, we expect the generator to have learned a good strategy to fool the discriminator, which should mean being able to produce samples that appear to be coming from the data distribution.

It's important to have in mind that the order we train each network will matter here. Trying to completely train D all at once before minimizing the generator's cost is highly prone to overfitting. Instead, one must try to maintain D close to the optimal solution by iterating a few steps on gradient descent and later walk one step on the generator's optimization procedure. The idea is that we can keep D close to optimal if in each turn θ^G moves slow enough.

The present strategy for training GANs has a heuristic motivation. It presents the same theoretical challenges that deep learning is known to have, plus new ones surrounding the stability of this alternate optimization solution. Next we present our first experiment, where we test the GAN model on a known distribution.

Algorithm 1 Stochastic gradient descent training of GANs.

for number of training iterations **do**
  for k steps **do**
    Sample minibatch $\{z^{(1)}, \dots, z^{(m)}\}$ from p_Z.
    Sample minibatch $\{x^{(1)}, \dots, x^{(m)}\}$ from the data set.
    Update the discriminator network D by the stochastic gradient:

$$\nabla_{\theta^D} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(x^{(i)}\right) + \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)\right]$$

  end for
  Sample minibatch $\{z^{(1)}, \dots, z^{(m)}\}$ from p_Z.
  Update the generator network G by the stochastic gradient:

$$\nabla_{\theta^G} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)$$

end for

Example 4.2.1. (Gaussian mixtures)
Let's apply the adversarial training strategy to train a GAN on samples of a known Gaussian mixture. In this way, we expect to be able to quantify and expose the capacity of this method. For this purpose we consider $X \sim p_{\pi,\mu,\Sigma}$ a random variable on $\mathbb{R}^d$ distributed as a mixture of Gaussians (Recall 2.3.1), our goal distribution, which we want to approximate only by sampling from it. We define for $k \in \mathbb{N}$ and given $\pi = \{\pi_i\}_{i=1}^k \in [0,1]^k$, $\mu = \{\mu_i\}_{i=1}^k \in (\mathbb{R}^d)^k$ and $\Sigma = \{\Sigma_i\}_{i=1}^k \in (\mathbb{R}^{d\times d})^k$ the associated Gaussian mixture distribution

$$p_{\pi,\mu,\Sigma}(x) = \sum_{i=1}^{k} \pi_i \mathcal{N}(x|\mu_i, \Sigma_i), \tag{4.20}$$

where π need to satisfy $\sum_{i=1}^k \pi_i = 1$.

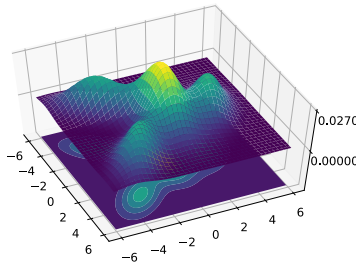

Figure 4.1: Surface plot.

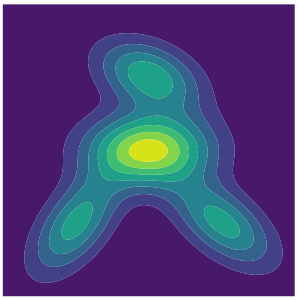

Figure 4.2: Contour plot.

We'll work with a latent variable $Z \sim \text{Unif}[0,1]^m$, for some fixed positive integer m. We then follow by approximating a neural network $G(\cdot) = G(\cdot, \theta^G)$ from the latent space $\mathcal{Z} = [0,1]^m$ to $\mathcal{X} = \mathbb{R}^d$ with the help of a second neural network $D(\cdot) = D(\cdot, \theta^D)$ as the discriminator following algorithm 1. For this example we have trained standard fully connected neural networks using as activation function the rectified linear unit ($\sigma(x) = \max(0, x)$) in all hidden layers, and a standard sigmoid function at the end of the discriminator net. All our tests have been done using a combination of Python and the Tensorflow library ([1]). That will also be the case in all subsequent cases.

As our first result we chose to train on a 2D mixture and graphically show the resemblance of the obtained distribution with the original one. We have set the latent dimension as $m = 10$, our networks both have two hidden layers each, with 64 on the first layer and 32 on the second on both cases. The training turns were taking using one turn improving the generator and one the discriminator with a learning rate of 0.0002 and using a minibatch of size 64. The figure below compares the histogram obtained for the generators distribution after 30k turns and the contour plot of the original distribution.

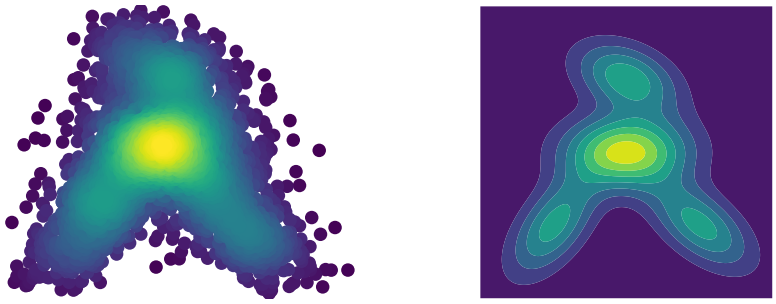

Figure 4.3: Side by side comparison between induced distribution after 30k p_G (left) and the true distribution p_X (right).

Using the histogram we can also obtain quantifiable evidence that our method approximates the desired distribution, we use a quadratic difference between the amount of samples inside each bin of the histogram with the expected value in the bins interior. Then we can plot the value of this metric over the training steps to visualize how the method converges to some close distribution.

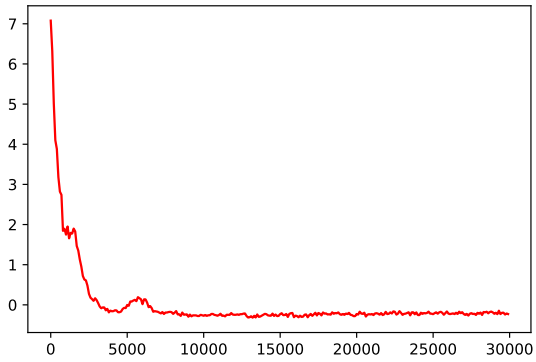

Figure 4.4: Values of the quadratic difference over 30k training steps (Using a $\log$ scale).

Next we also show some examples of histograms plotted during training, for a different visualization of the network evolution.

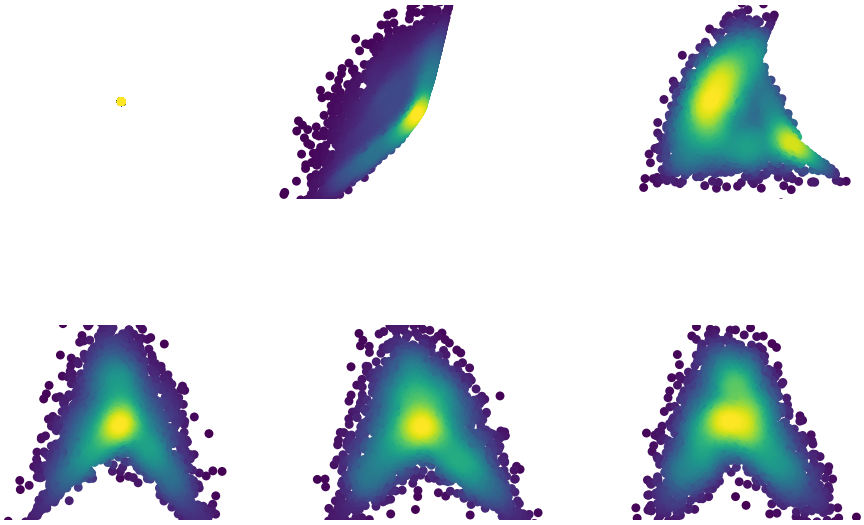

Figure 4.5: Histograms during training, respectively after 0, 1, 2.5, 5, 10 and 15 thousand steps

Looking at figure 4.2.1 we see that the distributions similarity doesn't change much after for example turn 10k, but still since as we can see the evolution isn't exactly decreasing over time we have to be careful with when to stop training. Observe that our example is totally theoretical and that in practice we don't have a way comparing with the true distribution, and therefore we can't say when to stop the process or which point during training is the best. Next we use again this metric to compare the results on this problem when varying some of the algorithm hyperparameters.

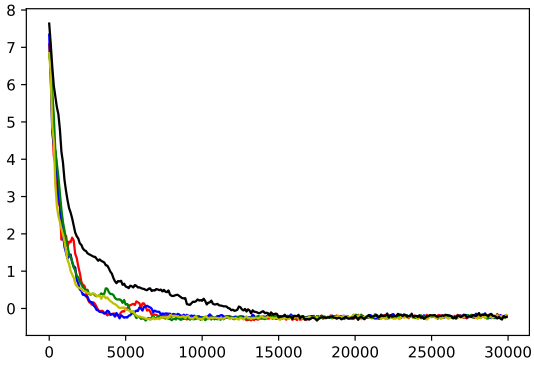

Figure 4.6: Values of the quadratic difference (on log scale) over 30k training steps changing the numbers of steps given by the discriminator in each turn (1 - red, 2 - blue, 5 - green, 10 - yellow, 100 - black).

We first changed the number of steps the discriminator optimizes in each turn (figure 4.6). The final result is essentially the same, i.e. in all test cases the method converged, perhaps taking longer then the others, which might point out that this parameter isn't decisive for that matter. Still we observe some differences on the curves, with 2 steps (blue curve) the curve seems to present the smallest values (the original paper [18] suggests this setting), but nevertheless the cases with 5 (green) and 10 (yellow) steps seem to hit the minimum faster with the later case presenting a more stable decreasing behavior. As we'll see in our next figure, the number of steps taking on the parameters of G is much more delicate (maintaining one step per turn for the discriminator).

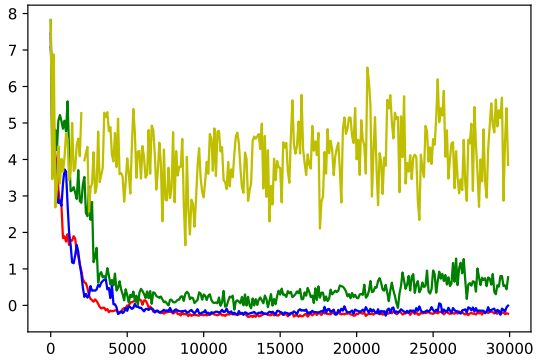

Figure 4.7: Values of the quadratic difference (on log scale) over 30k training steps changing the numbers of steps given by the generator in each turn (1 - red, 2 - blue, 5 - green, 10 - yellow).

Observing figure 4.7 we can see that by rising the number of steps taking by the generator we can fall into a dangerous zone, where the system is very unstable and doesn't converge at all. In the same spirit we decided to check with mixed settings taking some steps with G and some with D in each turn. This can be seen in the next figure.

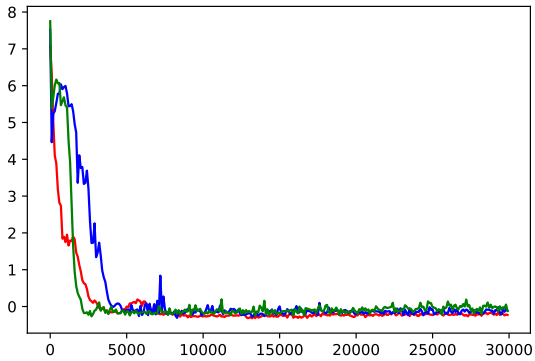

Figure 4.8: Values of the quadratic difference (on log scale) over 30k training steps with a mixed setting (1×1 - red, 2×2 - blue, 5×5 - green).

In these cases the method presents some bad behavior at the beginning of training (before 5k), but it seems to converge at the end only presenting larger variance on the result compared with the regular behavior.

The next two hyperparameters we have observed are the applied learning rate and the batch size, this time we used a smaller window of 10k iterations, which we think is enough for our purposes.

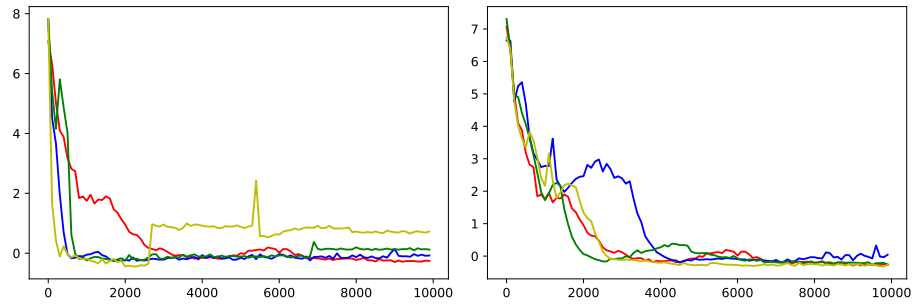

Figure 4.9: On the left comparison varying learning rate (0.0002 - red, 0.001 - blue, 0.005 - green, 0.01 - yellow), and varying batch size on the right (64 - red, 16 - blue, 128 - green, 512 - yellow).

From figure 4.9 we see that a high learning rate can compromise convergence of our method, while changing the batch size doesn't seem to affect as much, in fact in this window a larger batch seem to converge faster. In the case of the batch of size 16 graphically it looks like what we wanted but the quadratic error fluctuates a little over the other curves (this one has been observed in a larger window).

The next analysis we make is on the number of layers and neurons in each layer, this time we used a window of 20k turns, all other parameters besides the size of each

layer are exactly as in the first model.

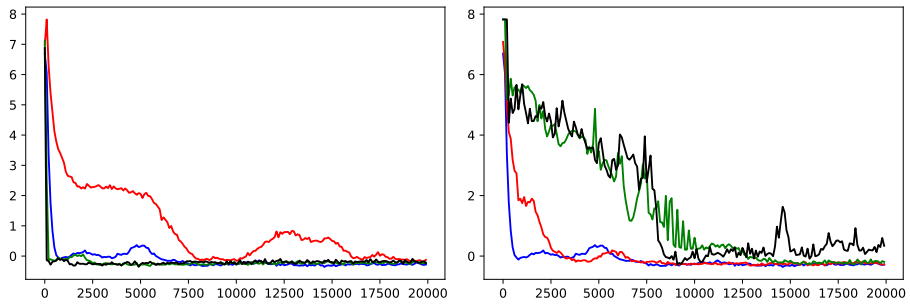

Figure 4.10: The left graph compares between different amounts of neurons in the only hidden layer of each network (16 - red, 64 - blue, 512 - green, 2048 - black), and the right one varies the number of layers (64 - blue, $64|32$ - red, $64|32|16$ - green, 8×64 - black).

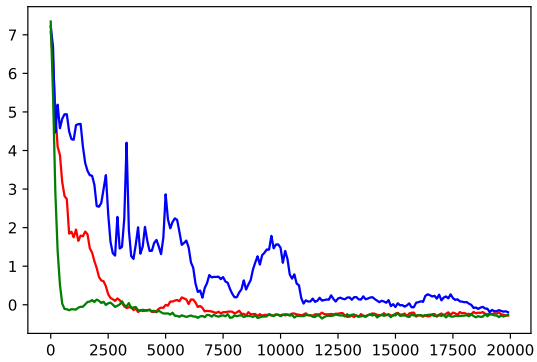

Figure 4.11: Networks with higher capacity on G or D ($G : 64|32 \times D : 64|32$ - red, $G : 64|32 \times D : 64$ - blue, $G : 64 \times D : 64|32$ - green).

Using one hidden layer in each network and changing the number of neurons on this layer (Figure 4.10 left) we observe that as the number of neurons grows the performance of the method also improves. On the other hand when we add more layers in the network (Figure 4.10 left) the effect goes in the other direction, the performance becomes worse when the network has more layers and it can even diverge as we have seen on the case with 8 layer (in black). While some of this results might change in function of the other hyperparameters the outcome is not truly surprising, increasing the capacity of a neural network that is already capable of approximating the goal function can actually push the model to overfit as we have discussed before. Now for the cases when we build G and D with distinct capacity (Figure 4.11) we observed that in

the case that D has one layer less than G it was much more difficult for the algorithm to find a good result, while the other way around have only improved performance. This last case was not explored to their whole capacity but while in most cases adding more layers to the discriminator (more tests were done) can improve performance in practice the standard setting is to build both networks with similar capacity.

Remark. *In practice the training strategy runs by simultaneously updating both θ^G and θ^D. One samples a minibatch from the dataset and one from the latent distribution p_Z, compute the respective gradients and update both networks simultaneously.*

Also, since in early stages of the training G still don't have any chance of fooling the discriminator, it is expected that the gradient provided by equation 4.18 is too week. Instead one chooses to maximize $\log D(G(Z))$ which is equivalent to the original problem of minimizing $\log(1 - D(G(Z)))$. This provide a much stronger gradient and should be able to keep the gradient from saturating.

The next example show GANs being applied in image datasets, in this case we assume that the data comes from a distribution and try to generate similar images by sampling from the modelled distribution.

Example 4.2.2. (Image generation)

In our image generation experiment we trained GANs on two famous datasets the MNIST database [32] and the Large-scale CelebFaces Attributes (CelebA) Dataset [34]. The first one consist on 55k images of handwritten digits (0 through 9) black and white and of dimension 28×28, the later contains around 202k images of celebrity faces colored and of dimension 218×178.

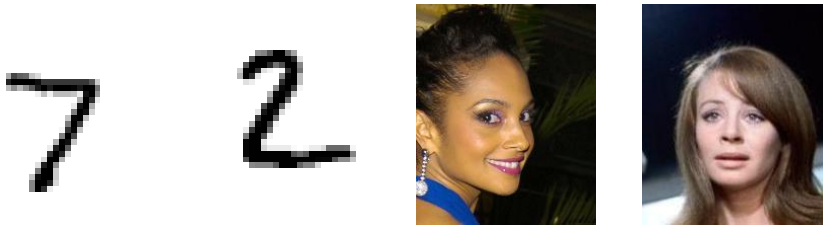

Figure 4.12: MNIST examples on the left and CelebA examples on the right.

As we have noticed before, using convolutional layers when working with images tend to produce better results then using only conventional fully-connected layers, and for GANs the tendency remains. The first results combining convolutional layers and GANs appear on [44], and we'll use this paper as our starting point. The called **deep convolutional generative adversarial networks (DCGANs)** adds convolutional layers both in the generator as in the discriminator, while in the generator the operation is transposed which means we train special upsampling procedures instead of what we are used to.

One way of seing transposed convolutions is by considering a sort of fractional stride. For example a $1/S$ stride could be applied by adding S zeros between each value

in the array and then convoluting the kernel. How to compute the output dimension is mantained, but this time we have S multiplying instead of dividing. The transposed convolution can be seen as a composition of signals, returning signals that combine details encoded in coarser signals. This fits well with our goal of generating images.

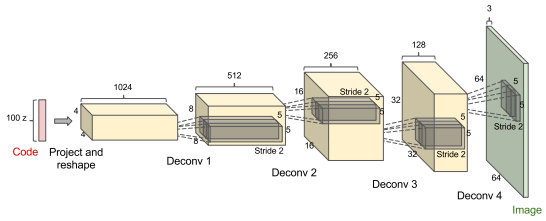

Figure 4.13: Example of a generators architecture on a DCGAN.

The architecture trained on both datasets is essentially the same with only a few differences. The training procedure is also the same as we have used in the previous example. Our latent distribution is uniform with dimension 100. The results shown next for the MNIST case we used a network G with one fully connected layer of size 256 and three convolutional layers of dimensions $128, 64, 32$ respectively, and a network D with convolutional layers of dimensions $32, 64, 128$ respectively and a fully connected at the end. We have run a total of 20 epochs with a 0.001 learning rate, where one epoch means enough batches to see the whole dataset even if we are picking images at random.

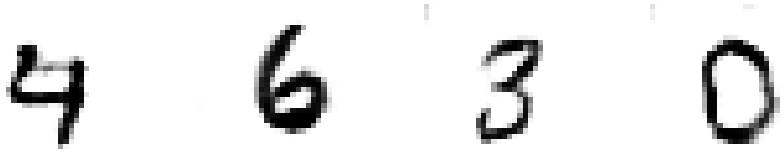

Figure 4.14: Generated samples from a GAN trained on the MNIST dataset after 20 epochs.

For MNIST some good results are observed very soon in training, but it takes some time until the model becomes stable, perhaps with better hyperparameters the model can stabilize sooner, but that wasn't our concern at the moment.

When training on the CelebA dataset we have chosen to crop and resize the images so they have dimensions 64×64. We trained a GAN using the same architecture as before for about 20 epochs. Below we present a few of the final results.

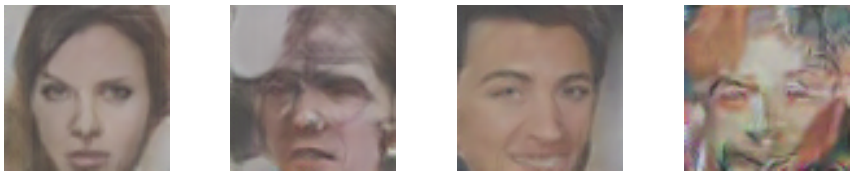

Figure 4.15: Generated samples from a GAN trained on the CelebA dataset after 20 epochs.

On the generated samples above we tried to show both good and bad results, but recall that we are using the same architecture as before to express much more detail, assuming that the space of photos of human faces is more complex than handwritten digits. Using neural networks with higher capacity is definitely recommended when generating natural images as those, but looking for the best hyperparameters for such a task in our chosen architecture is not our main goal. To observe better how the GAN model samples progress we also show the "evolution" of a sample during training.

Figure 4.16: Generated samples from a GAN trained on the CelebA dataset after 20 epochs.

Chapter 5

GANs for conditional distributions

We have seen so far how powerful GANs can be for learning distributions and specially how well they perform even on high dimensional and complex datasets, like images for instance (see example 4.2.2). Nevertheless, our control over its generative capacity is still very limited. We would like to improve the so far presented methods so that we gain a certain level of control over the generated samples. The way we hope to do this is by learning to sample from conditional distributions. This means we would be able to use extra information to condition the generative procedure and then impose some control over characteristics of our generated samples. Such ability would grant us a whole new range of criative applications, like

- Generating natural images from text descriptions [45, 57], or from a sketch drawing [26].
- Image reconstroction, or Inpainting [58, 8].
- Super-resolution [33].

In this chapter will see how to modify or apply GANs for conditional generative modeling. As we'll see, conditioning is not a straightforward operation and it surely depends on what kind of condition or the level of control we want to impose. We discuss two distinct approaches for conditional generative modeling through GANs.

5.1 Conditional Generative Adversarial Networks (CGAN)

Conditional generative adversarial networks (CGANs) were proposed in [37] and are the most commonly known approach for conditioning GANs. This model extends the traditional GAN architecture by feeding extra information directly into the generator and discriminator networks. This extra information is usually given through another variable, i.e. the conditioned event we consider is derived from a related random variable $E = [Y = y]$, $y \in \mathcal{Y}$, where y could represent a class label, an encoded text

sentence, a matrix of pixels, or any other information related to the input. Therefore this conditioned idea is more suited for labelled data.

For presenting this extention consider $(X, Y) \sim p_{data}$ and $Z \sim p_Z$ some latent random variable, and our new pair (G, D) of networks

$$G : \mathcal{Z} \times \mathcal{Y} \times \Theta_G \to \mathcal{X} \tag{5.1}$$

$$D : \mathcal{X} \times \mathcal{Y} \times \Theta_D \to [0, 1] \tag{5.2}$$

that have almost the same appearance as the original ones, but also taking the label information $[Y = y]$ as inputs.

The minimax game is defined using a value function analagous to the original one, but this time having variable Y as a parameter

$$\begin{aligned} v(G, D) = \mathbb{E}_{(X,Y)\sim p_{data}} \left[\log D(X \mid Y)\right] + \\ + \mathbb{E}_{\substack{Y\sim p_Y \\ Z\sim p_Z}} \left[\log\left(1 - D(G(Z \mid Y) \mid Y)\right)\right]. \end{aligned} \tag{5.3}$$

In this context we are interested that for each fixed $y \in \mathcal{Y}$ the random variable $G(Z \mid y)$ as a latent model approximates the data distribution conditional in Y. The reason for this choice of value function is the same as the one presented in the previous chapter, the only difference being that now X and Y are correlated random variables. Solving the minimax game for v

$$G^* = \operatorname*{argmin}_G \max_D v(G, D) \tag{5.4}$$

is now equivalent to find $G(\cdot \mid \cdot)$ minimizing the averaged Jensen-Shannon divergent between the conditional data distribution and the model's distribution,

$$\mathbb{E}_{Y\sim p_Y} \left[JS(\mathbb{P}(X \in \cdot \mid Y) \parallel \mathbb{P}(G(Z \mid Y) \in \cdot))\right]. \tag{5.5}$$

The choice of cost function is again motivated by the negative log-likelihood of identifying samples coming from X or $G(Z \mid Y)$. The implicit dependence between the variables (X, Y) forces the equilibrium to induce not only $G(Z \mid Y)$ as an approximation for X, but the conditioned models $G(Z \mid y)$ to approximate the desired conditional distributions $\mathbb{P}(X \in \cdot \mid Y = y)$.

Conditioning GANs in general should be harder for training, since besides learning the distribution it must capture the dependence between features and labels in the dataset. Nevertheless, this will not be the case always, since as we'll see in one of our examples labels can also help the network knowing what to generate. Next, we present two examples that use CGAN, one in a controlled setting of Gaussian mixtures and another applied into a image dataset.

Example 5.1.1. (Conditioned Gaussian mixtures)

We return to the Gaussian mixture experiment (example 4.2.1) and apply the previous idea to train conditional models. We first trained a GAN conditioned on the mixture classes, i.e. we consider each Gaussian in the mixture as a different class. We trained on the same artificial example we have used previously, which is the mixture of four Gaussians with parameters:

$$\Sigma = \left\{ \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 2 & -\frac{1}{2} \\ -\frac{1}{2} & 1 \end{bmatrix}, \begin{bmatrix} \frac{3}{2} & 1 \\ 1 & 2 \end{bmatrix}, \begin{bmatrix} 2 & -1 \\ -1 & \frac{3}{2} \end{bmatrix} \right\}, \tag{5.6}$$

$$\mu = \left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 3 \end{pmatrix}, \begin{pmatrix} -3 \\ -3 \end{pmatrix}, \begin{pmatrix} 3 \\ -3 \end{pmatrix} \right\}, \tag{5.7}$$

$$\pi = \{0.4, 0.2, 0.2, 0.2\}. \tag{5.8}$$

We trained a GAN with one hidden layer of size 1024 for each G and D and set the latent dimension this time as 100, as describe before the class is also fed in both networks (see 5.3) and trained as usual.

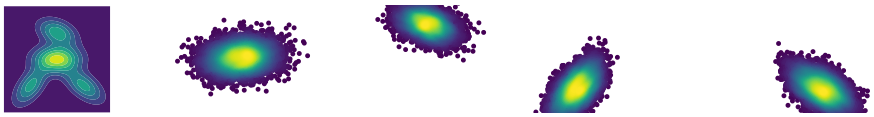

Figure 5.1: Contour plot of the original mixture compared with the learned four classes.

As before, we have used a quadratic difference on the distribution histogram to try to quantify the progress of our model, for our context we have separately computed this values for each of the four classes. We observe this values on a window of 50k training steps.

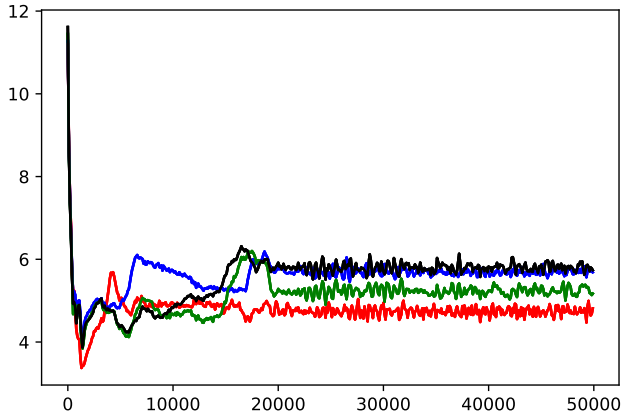

Figure 5.2: Performance (on log scale) during training for 50k steps where each line correspond to one class, (class 1 - red, class 2 - black, class 3 - green, class 4 - blue).

Instantly we can observe a different behavior of the evolution of the model when compared with our previous results on the case where we didn't condition the model. The model takes a lot more time to stabilize, around 20k steps. It seems to reach lower values in the first moments of training which is a strange behavior that repeats itself often when varying hyperparameters. We observed that the model is even more sensitive to the hyperparameters than the original one, which made it harder to work with. Nevertheless, we observe that the model stabilizes in similar regimes as the ones observed in our first experiment. We can also observe that in this case the first class (red) stabilizes sooner and in a better range then the others. This may be linked with the fact that it is the one with higher chance of appearing in the samples. At the same time, class 3 (green) seems to have some advantage over the remaining two that are together equally likely to appear. This last remark applies to different hyperparameter regimes, but unfortunately we don't have a good explanation for why that is the case.

Example 5.1.2. (Image generation conditioned)
We return to example 4.2.2. This time we condition the GAN model on discrete classes coming from the labels paired with each image. This can give us greater control on generating images with specific properties.

On the MNIST case we are interested in conditioning on the class digit label (0-9). We have used the same architecture as in the original experiment. Below we show a few results after 30 epochs.

Figure 5.3: Samples generated conditioning on each class in order.

An interesting remark is that while in the previous case of gaussian mixtures separating classes seem to be harder for the model, in this case it seems that knowing the class made it easier to the model to learn from it. That is, we observe that the final results are better and we need less steps to reach the same quality as in the original experiment. The fact that different classes associate to really distinct shapes seems to help the model. This wasn't the case with gaussian mixtures, where any sample in $\mathbb{R}^d$ could be associated with any class with a reasonable probability.

5.1.1 Conditioning from optimal latent representation

Another form of providing conditional samples is to look over the latent space of a pre-trained generator for vectors $z^* \in \mathcal{Z}$ corresponding to good conditional generated samples, i.e. where $G(z^*)$ satisfies the given condition or that minimizes some equivalent criteria. So, instead of learning the conditional maps directly we use a GAN (G, D) trained as usual (4.18) to provide a generator and discriminator for $X \sim p_{data}$. To find the latent representation that matches a condition $\mathcal{C}$ we must define a loss function that quantifies if $G(z)$ is close to satisfy $\mathcal{C}$ and at the same time guarantees resemblance with the training data. The second point is typically measured by evaluating the confidence of the trained discriminator. We define the perceptual loss by

$$L_{perceptual}(z) = \log\left(1 - D(G(z))\right). \tag{5.9}$$

We also need a way of quantifying the agreement with our condition. This can be done in different ways. For example one will need auxiliary classifiers if conditioning on a class label, and in general some kind of encoding that can map $G(z)$ to a metric space where we can measure proximity with the given condition. This loss function we denote as the contextual loss $L_{contextual} : \mathcal{Z} \to \mathbb{R}^+$. Our main loss function is then a balance between both losses

$$\mathcal{L}(z) = L_{contextual}(z) + \lambda L_{perceptual}(z) \tag{5.10}$$

where λ is a hyperparameter. Finally, our optimal latent representation is found by minimizing such loss function

$$z^* = \underset{z \in \mathcal{Z}}{\operatorname{argmin}}\, \mathcal{L}(z). \tag{5.11}$$

This representation can be found by updating z backpropagating over the networks internal paramaeters to such loss function.

Example 5.1.3. Inpainting:

In this next application we want to infer the damaged part of an image using a GAN model. This means the non-damaged part will work as our condition and the generated image should be a good sample to "inpaint" the conditioned part. The idea is as in [58]; to use a mask operator M to compute a loss contextual function that indicates how the section of the generated image matches with the conditional pixel set we want to condition the sample

$$L_{contextual}(z) = \|M \odot G(z) - M \odot y\|_1 \tag{5.12}$$

where y is the original image we are conditioning the model, which we may only have partial information $M \odot y$.

Then, our full loss objective is given by

$$\mathcal{L}(z) = \|M \odot G(z) - M \odot y\|_1 + \lambda \log\left(1 - D(G(z))\right) \tag{5.13}$$

where for the network parameters fixed we minimize over the latent space $z \in \mathcal{Z}$.

For our experiments we used the already trained GAN model showed in example 4.2.2 and applied the above strategy using a binary mask of the size of the images 64×64 with the center square of 32×32 removed.

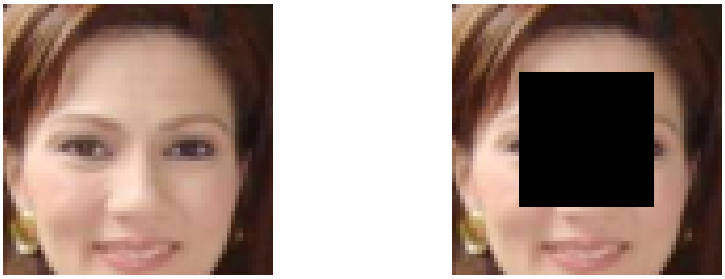

Figure 5.4: Original image on the left and after applying the mask on the right.

For all the following results we used $\lambda = 0.003$ as recommended on the original work [58]. We runned 5k steps with the TensorFlow's [1] Adam optimizer for 0.005 as learning rate. Differently from all the other experiments we have presented using neural networks, the gradient step is done taking the input z as parameter while the parameters of both networks remain frozen. This means we don't retrain the network on data, we only search for a good sample on the latent space.

A step that we have chosen to do, that we think gave better results, was to optimize batches of latent samples, i.e. we start with a batch os 64 samples which in the end of the optimization routine give us 64 new image samples, with those we chose as our output the one with smaller value on the loss function. While the original work [58] doesn't seem to need such procedure, we observed that in our case sometimes samples diverged depending on where the start point was taken. By using batches, we were able to guarantee decent results on average.

Figure 5.5: Impaint generated image from condition given in 5.4.

The results from the method clearly depend on the quality of the image generator we have in hand. Therefore a few artifacts that we already observed in the original experiment where we trained our network appear here. We can see that in our typical examples, the features seem to match with the outside neighbourhood, but the colors of the image seem a little off.

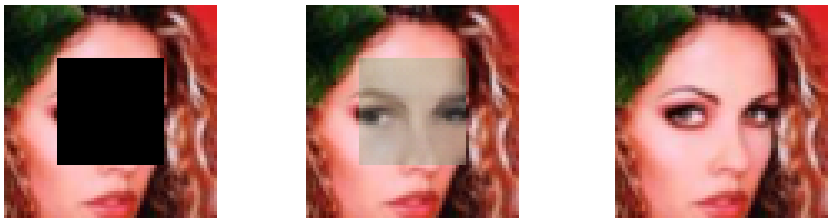

Figure 5.6: From left to right: Condition, generated image, original image.

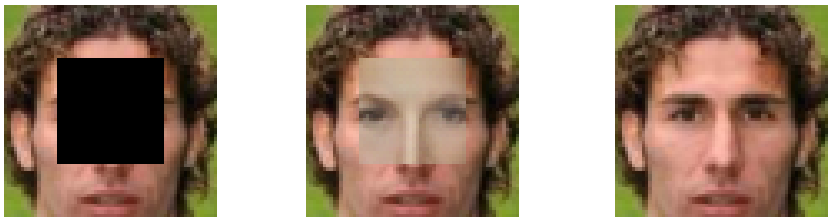

Figure 5.7: From left to right: Condition, generated image, original image.

One potential improvement that we haven't applied here is when computing the contextual loss to give higher importance to pixels closer to the mask boundary. This could be done using a distance map. Another thing that our result could benefit from is to instead of simply adding the generated result to reconstruct the condition $M \odot y + (M^{-}1) \odot G(z)$ we could use a smarter blending procedure, like a Poisson blending as it is suggested on the original work we have cited before.

We also show some bad samples which are important to think in what direction such methods must improve

Figure 5.8: Bad samples.

The first sample has the classic mistake of not realizing that glasses should be on the picture. The two other samples are also bad, but the last one is truly alarming since it got even the face's skin color wrong. One might think that in this case the reason was that the network diverged on all samples towards white faces, but to our surprise that wasn't the case and we show what we mean next.

Figure 5.9: First the original image, then three generated images trying to match the condition, being the first the one matched at the end of the search.

This bizarre phenomenon observed above can perhaps be interpreted in two ways. The first is that the network didn't learn at all how faces should look like since it couldn't even learn how to compose the skin color of someon's face. Another one is that since searching from samples in the latent space isn't really the same as conditioning the learned distribution, samples that should be too rare to appear end up being reached by the method. Most probably, both things are true and we couldn't learn the distribution properly nor we are able to condition by this approach. Nevertheless, the method is still promising and latent search has shown pretty good results even for situations where conditions are given by structured data as in impainting. Therefore, it would surely be worth investigating how to improve such processes in the future.

This method seems to allow us to condition on more complex information, as the previous example suggests. Differently from the CGAN approach, we are not actually learning to sample from the conditional distribution. Also, this second method requires an expensive optimization procedure for each generated sample, differently than the one before which generated samples in one feedforward pass.

Chapter 6

Conclusion

Generative adversarial networks have become a staple in generative modeling in the deep learning community in recent years due to the impressive results several groups have shown by using such class of models. In this work we tried to show a bit of that power through experiments and theoretical discussions, while also presenting the necessary framework so the reader can understand the concepts behind the technique. We have also move the discussion to conditional generative modeling where much of the interest on this method is current focused, since it provides a new level of control over the generative power.

The experiments presented on this work were made with the purpose of understanding how the methods works in practice. We were less worried about embelishing our results or taking much time tuning to the best hyperparameters. We refer the reader to several nice applications cited in the text which one can find impressive results using essentially the same methods we have used but with a lot more time and computational power applied into it.

Training GANs on known distributions as synthetic data, as in the case for Gaussian mixtures, gave us the possibility of measuring the performance of our methods both in the classic and conditional cases. While much more could be done in this sense, having a ground truth distribution allowed us to verify, or at least gave us some evidence, that the method is really capable of learning distributions. Such problem of measuring performance of generative models is a hard one and to our knowledge not very well understood. Nevertheless, to verify newly designed generative models in ground truth distributions is a rare common practice, and we think more should be done them just verifying the method on popular datasets. On the other hand, if we consider the fact that our true interest is to be able to learn high dimensional distributions, which is a big reason why GANs and deep learning are used, estimating whether the model learned appoximates a known distribution may become impractical for the high dimensional case. Therefore, observing the results on popular datasets might be the best we can do. The problem of measuring the performance of generative models, which wasn't at first one of our main interests for this work, seems to be an important one and we think it can lead us to very interesting future directions.

The main idea behind the original GAN architecture and perhaps the true innova-

tion we have observed in this work is adversarial training. Seeing learning as players improving their strategies in a game brings a whole new dynamic to the deep learning framework. Part of this new dynamics consists of finding equilibriums for games, which is different then the usual optimization approach, and therefore might need a different set of tools for better understanding the problem. The concept of generalization is still very misterious when learning is done by the usual means of deep learning. Adversarial training it's even less understood since the theory behind it is so new that perhaps the proper concepts haven't even being defined yet. We think these ideas, although really new, will stay in the field as important tools for a long time, and studying their dynamics and properties will be essential to reach their full potential.

We have seen how this new approach allows powerful generative models, but the same ideas could be applied in other contexts. If we extend the concept behind GANs we might be able to design games specifically for solving learning tasks. Based on the success of the GAN model we think a good work direction would be to apply and develop from theory of mechanism design to create new learning games, which could be usefull for a new range of learning applications.

Controlling and conditioning generative models have been in the center of interest in GANs, since it trully shows how powerful these networks can be. In our work we discuss this task through two different approaches, that are to the best of our knowledge the two most popular. We saw how these models are applied in different contexts and have their own advantages and disadvantages. Developing and perfecting this and new conditional methods are central to the problem we are interested in. We also think studying how these models can be properly conditioned extends and can provide better understanding of the learning process behind GANs. Therefore, we can say our main focus now and for future work has been on studying this conditional settings and the problems that may appear.

Training GANs efficiently and with scarce data is also an interesting problem here as it is in deep learning in general. When the applications we are interested in start becoming more and more complex, the tendency is usualy to require more and more data and computational power. How to solve this in the adversarial framework is perhaps a different problem than the usual one and would also be a very interesting research problem. This pratical problems may seem industrial concerns, but it really turns out to represent a big problem in the academic community, since such restrictions concentrate progress on the hand of groups that have more money and consequently better resources. Therefore, working on this kind of problems are a main concern of the field not only in the generative setting but also in general.

Most GAN applications are related to image generation and perhaps where most of the method's success have originated in this area. We can ask ourselves if the adversarial intuition from forger versus detective means that in fact the best we can hope is for a good forger and not to trully learn the nuances that our distribution represents. The question might be if the network trully learns to be creative in some sense or if it just fooling us by repeating the same patterns that it knows we are used to seeing on the images that we have feeded it with. We can't ourselves determine what a natural image distribution looks like and therefore evaluate such fact becomes an impossible task. Nevertheless, a way one could try to force GANs to express creativity is to use conditional models. If a network has to generate images conditioned on features that

are maybe unseen in any of the training examples, if it does generate, this could be a good sign of generalization. Since we can't measure if our method approximates de true distribution in the case of real datasets it is hard to fix what is good enough evidence that the network is generalizing the data. How we know if the network isn't just memorizing different pieces and patterns of the image dataset without undestanding what they form properly? Such discussion is a complicated one, but we think it has a lot of open problems that may be interesting as future work.

GANs are a really powerful thecnique that have shown impressive results and will probabibly keep doing it for quite a long time. As many other machine learning models they become extra special when combined with thecniques and knowledge of specialized applications. Using GANs for art composition is already becoming a reality and a lot more is expected on how GANs can extend our creative capabilities. We can expect that in the future the work of a computer graphics artist might be assisted by a trained GAN machine that can add textures and fine details to projects, making his work faster and perhaps better. The range of possible applications only becomes larger as the quality of the results keeps improving. Together with other known generative models being improved recently the applications on sound and music synthesis, 3D modeling, reinforcement learning are showing that the generative power is starting to reach and run beside the successes we have seen on supervised learning.

As the results become more and more realistic and sometimes impossible to distinguish from real data, the implications of GANs can lead to alarming consequences.

As the results become more and more realistic and sometimes impossible to distinguish from real data, the implications of GANs can lead to alarming consequences. Such tools could be used to frame people with synthetic photos or videos or even writing damaging fake news, and we might not be able to recognize when thats the case, or maybe we do only when is too late. Understanding how this methods are able to create such realistic samples could help us preventing such consequences. Even the jobs of some artistits, or musicians can also be affected by GANs, since the quality of their generated work is becoming more and more like our own. This touches the issue many are discussing on how the intelligent automation deep learning and the field in general are creating will affect our future as a society. Ethics and fairness are recently taking central position in machine learning and the related community, but we seem to be very far from solving these issues. While many people are worried on how the machines might gain superhuman intelligence, we should be more aware on how automatization is already affecting our surroundings. We think these problems are relevant not only to the machine learning community but to every one.

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein gan. *arXiv preprint arXiv:1701.07875*, 2017.

[3] Y. Bengio, J. Louradour, R. Collobert, and J. Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.

[4] D. Berthelot, T. Schumm, and L. Metz. Began: Boundary equilibrium generative adversarial networks. *arXiv preprint arXiv:1703.10717*, 2017.

[5] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[6] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[7] Y. Bulatov. notmnist. `http://yaroslavvb.com/upload/notMNIST/`.

[8] Z. Chen, S. Nie, T. Wu, and C. G. Healey. High resolution face completion with multiple controllable attributes via fully end-to-end progressive generative adversarial networks. *arXiv preprint arXiv:1801.07632*, 2018.

[9] Y. L. Cun. Unsupervised learning: The next frontier in ai. RI Seminar - Carnegie Mellon University in Pittsburgh, Pennsylvania, 2016.

[10] R. Eldan and O. Shamir. The power of depth for feedforward neural networks. In *Conference on Learning Theory*, pages 907–940, 2016.

[11] C. Finn, P. Christiano, P. Abbeel, and S. Levine. A connection between generative adversarial networks, inverse reinforcement learning, and energy-based models. *arXiv preprint arXiv:1611.03852*, 2016.

[12] C. Finn, I. Goodfellow, and S. Levine. Unsupervised learning for physical interaction through video prediction. In *Advances in Neural Information Processing Systems*, pages 64–72, 2016.

[13] C. Finn and S. Levine. Deep visual foresight for planning robot motion. *arXiv preprint arXiv:1610.00696*, 2016.

[14] J. Friedman, T. Hastie, and R. Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[15] K. Fukushima and S. Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer, 1982.

[16] R. Ge, F. Huang, C. Jin, and Y. Yuan. Escaping from saddle points—online stochastic gradient for tensor decomposition. In *Conference on Learning Theory*, pages 797–842, 2015.

[17] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[18] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[19] A. Graves and N. Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1764–1772, 2014.

[20] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[21] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[22] J. Ho and S. Ermon. Generative adversarial imitation learning. In *Advances in Neural Information Processing Systems*, pages 4565–4573, 2016.

[23] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[24] R. Houthooft, X. Chen, Y. Duan, J. Schulman, F. De Turck, and P. Abbeel. Vime: Variational information maximizing exploration. In *Advances in Neural Information Processing Systems*, pages 1109–1117, 2016.

[25] D. H. Hubel and T. N. Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195(1):215–243, 1968.

[26] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros. Image-to-image translation with conditional adversarial networks. *arXiv preprint arXiv:1611.07004*, 2016.

[27] A. Karlin and Y. Peres. *Game Theory, Alive:*. Miscellaneous Book Series. American Mathematical Society, 2017.

[28] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[29] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[30] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*, pages 396–404, 1990.

[31] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[32] Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010.

[33] C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, et al. Photo-realistic single image super-resolution using a generative adversarial network. *arXiv preprint arXiv:1609.04802*, 2016.

[34] Z. Liu, P. Luo, X. Wang, and X. Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

[35] W. Ma and J. Lu. An equivalence of fully connected layer and convolutional layer. *arXiv preprint arXiv:1712.01252*, 2017.

[36] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin, et al. Ad click prediction: a view from the trenches. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230. ACM, 2013.

[37] M. Mirza and S. Osindero. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*, 2014.

[38] A. v. d. Oord, N. Kalchbrenner, and K. Kavukcuoglu. Pixel recurrent neural networks. *arXiv preprint arXiv:1601.06759*, 2016.

[39] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[40] D. Pfau and O. Vinyals. Connecting generative adversarial networks and actor-critic methods. *arXiv preprint arXiv:1610.01945*, 2016.

[41] V. Plakandaras, R. Gupta, P. Gogas, and T. Papadimitriou. Forecasting the us real house price index. *Economic Modelling*, 45:259–267, 2015.

[42] R. Prabhavalkar, K. Rao, T. N. Sainath, B. Li, L. Johnson, and N. Jaitly. A comparison of sequence-to-sequence models for speech recognition. In *Proc. of Interspeech*, 2017.

[43] L. Raad, A. Davy, A. Desolneux, and J.-M. Morel. A survey of exemplar-based texture synthesis. *arXiv preprint arXiv:1707.07184*, 2017.

[44] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[45] S. Reed, Z. Akata, X. Yan, L. Logeswaran, B. Schiele, and H. Lee. Generative adversarial text to image synthesis. In *Proceedings of The 33rd International Conference on Machine Learning*, volume 3, 2016.

[46] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[47] T. Salimans, I. J. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen. Improved techniques for training gans. *CoRR*, abs/1606.03498, 2016.

[48] P. Y. Simard, D. Steinkraus, J. C. Platt, et al. Best practices for convolutional neural networks applied to visual document analysis. In *ICDAR*, volume 3, pages 958–962, 2003.

[49] J. T. Springenberg. Unsupervised and semi-supervised learning with categorical generative adversarial networks. *arXiv preprint arXiv:1511.06390*, 2015.

[50] A. B. Tsybakov. Introduction to nonparametric estimation. revised and extended from the 2004 french original. translated by vladimir zaiats, 2009.

[51] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. Wavenet: A generative model for raw audio. *CoRR abs/1609.03499*, 2016.

[52] A. W. Van der Vaart. *Asymptotic statistics*, volume 3. Cambridge university press, 1998.

[53] L. Wasserman. *All of statistics: a concise course in statistical inference*. Springer Science & Business Media, 2013.

[54] L.-Y. Wei, S. Lefebvre, V. Kwatra, and G. Turk. State of the art in example-based texture synthesis. In *Eurographics 2009, State of the Art Report, EG-STAR*, pages 93–117. Eurographics Association, 2009.

[55] D. H. Wolpert. The lack of a priori distinctions between learning algorithms. *Neural Comput.*, 8(7):1341–1390, Oct. 1996.

[56] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[57] T. Xu, P. Zhang, Q. Huang, H. Zhang, Z. Gan, X. Huang, and X. He. Attngan: Fine-grained text to image generation with attentional generative adversarial networks. *arXiv preprint arXiv:1711.10485*, 2017.

[58] R. Yeh, C. Chen, T. Y. Lim, M. Hasegawa-Johnson, and M. N. Do. Semantic image inpainting with perceptual and contextual losses. *arXiv preprint arXiv:1607.07539*, 2016.

[59] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.

[60] H. Zhang, T. Xu, H. Li, S. Zhang, X. Huang, X. Wang, and D. Metaxas. Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks. *arXiv preprint arXiv:1612.03242*, 2016.

[61] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. *arXiv preprint arXiv:1703.10593*, 2017.