Generative Adversarial Structured Networks

Ben London[*]
Amazon
blondon@amazon.com

Alexander G. Schwing[*]
University of Illinois at Urbana-Champaign
aschwing@illinois.edu

Abstract

We propose a technique that combines generative adversarial networks with probabilistic graphical models to explicitly model dependencies in structured distributions. *Generative adversarial structured networks* (GASNs) produce samples by passing random inputs through a neural network to construct the potentials of a graphical model; maximum a-posteriori inference in this graphical model then yields a sample. To train a GASN, one must differentiate a bi-level optimization, which is non-trivial. We present a solution based on "smoothing" the generator, and propose two methods for obtaining the smoothed gradient. We show preliminary experimental results, demonstrating that training GASNs is feasible.

1 Introduction

The *generative adversarial* learning paradigm has significantly advanced the field of unsupervised learning. The adversarial framework pits a generator against a discriminator in a non-cooperative two-player game: the generator's goal is to generate artificial samples that are convincing enough to be mistaken for real samples; the discriminator's goal is to distinguish between real and fake samples. The players alternate moves, updating their respective models along the way, until either no progress is made or they reach a stalemate. The key condition that enables generative adversarial learning is that the outputs of each player are differentiable with respect to their parameters; thus, given a global learning objective, which measures the state of the game, the errors made by each player can be back-propagated to the relevant parameters. Even though, in practice, the global learning objective might need to be adjusted to improve the learning process.

When first proposed [4], generative adversarial learning was applied using multi-layer perceptrons for the generator and discriminator—both of which are differentiable functions of their parameters. These models were dubbed *generative adversarial networks* (GANs), which have become synonymous with generative adversarial learning. The technique was later extended to various other "deep" architectures, such as convolutional neural networks [12], conditional models [11, 3], variational autoencoders [1], moment-matching networks [9], and many others.

All of the above generate samples by passing noise through the input of a feed-forward neural network. One limitation of this approach is that the individual outputs of the generator (*e.g.*, words in a sentence, or pixels in an image) are not sampled in a way that captures dependencies between them. For example, if the samples are images, neighboring pixels should be more likely to have similar values. To address this issue, we propose *generative adversarial structured networks* (GASNs). GASNs account for structured dependencies by enhancing the generator with a probabilistic graphical model, thus enabling joint inference over the outputs. By inferring the outputs jointly, a structured generator can produce more realistic samples. However, as we discuss shortly, this added expressivity introduces a fundamental challenge: to obtain the gradient of the system, one must differentiate of a bi-level optimization.

[*]Equal contribution

Workshop on Adversarial Training, NIPS 2016, Barcelona, Spain.

Differentiating bi-level optimizations is a well-studied problem [see, e.g., 5], however, until now, it has not received attention in the context of generative adversarial learning, or in machine learning in general. One of our key contributions is to identify and analyze the issue. We then propose a workaround based on "smoothing" the generator, as well as two algorithms for efficiently computing the gradient of the smoothed generator.

We conclude with an application of the proposed approach on a small-scale, synthetic problem. These preliminary results demonstrate that the approach works, and encourages further experimentation with real datasets.

2 Preliminaries

In the following we briefly review the training process of GANs. Suppose we are interested in modeling the distribution over a set of random variables, $\mathbf{x} = (x_1, \ldots, x_N)$, distributed according to some unknown distribution, P. Our goal is to learn a *generator*, denoted $G_\theta(\mathbf{z})$, which depends on some adjustable parameters, θ, and a source of randomization, $\mathbf{z}$, drawn from a "simple" distribution, such as a multivariate Gaussian. To learn the parameters of the generator we introduce a *discriminator*, $D_\mathbf{w}(\mathbf{x})$, depending on parameters $\mathbf{w}$. The discriminator attempts to distinguish between *real* data points and *artificial* samples obtained, *e.g.*, via the generator. The output of the discriminator $D_\mathbf{w}(\mathbf{x})$ is the probability of its input being a real sample.

To train the discriminator, we want to maximize the likelihood of real samples and minimize the likelihood of artificial samples from the generator. Stated differently, we aim to maximize $D_\mathbf{w}(\mathbf{x})$ and $1 - D_\mathbf{w}(G_\theta(\mathbf{z}))$; or, alternately, minimize the negative logarithm of these quantities. We also want the generator to produce realistic data points; hence, the generator aims to reduce the latter quantity. Thus, given a set of real samples, $(\mathbf{x}_i)_{i=1}^m$, and a set of random noise samples, $(\mathbf{z}_i)_{i=1}^m$ (which could be fixed or regenerated at each step of learning), we can state the overall minimax objective as

$$\max_\theta \min_\mathbf{w} \; \frac{1}{m} \sum_{i=1}^m - \log D_\mathbf{w}(\mathbf{x}_i) - \log\left(1 - D_\mathbf{w}(G_\theta(\mathbf{z}_i))\right). \tag{1}$$

Provided the generator and discriminator are differentiable functions of their respective parameters, θ and $\mathbf{w}$, we can minimize Equation 1 using first-order optimization, *e.g.*, (stochastic) gradient descent.

3 Generative Adversarial Structured Network

The above setup is identical to the original GAN formulation [4]. In traditional GANs, the generator is typically defined as the output of a feed-forward neural network. The random signal is passed through the network to produce a sample from the estimated distribution. Though a deep neural network can capture complex interactions between its inputs, it cannot explicitly model structured interactions between its outputs. To actually model interactions and produce more realistic samples, we propose the following structured extension of the generator.

Instead of using a neural network to directly yield the elements of the output sample independently, we enforce structure more explicitly by leveraging collective inference. For instance, it is often the case that neighboring pixels in an image have similar values. This type of reasoning is often encoded by a scoring function that encourages adjacent pixel variables to have the same value. By jointly inferring the all pixels to maximize the scoring function, we obtain an assignment that is jointly optimal.

We will assume that each variable, $x_i : i \in \{1, \ldots, N\}$, can take one state among a valid set of labels $\mathcal{L} = \{1, \ldots, |\mathcal{L}|\}$, *i.e.*, $x_i \in \mathcal{L}\ \forall i$. Given a set of parameters, θ, and a randomization, $\mathbf{z}$—*e.g.*, multivariate normally distributed—we denote the score of a configuration $\mathbf{x}$ via the function $F(\theta, \mathbf{z}, \mathbf{x})$. The generator outputs the configuration with the highest score, *i.e.*,

$$G_\theta(\mathbf{z}) = \arg\max_\mathbf{x} F(\theta, \mathbf{z}, \mathbf{x}). \tag{2}$$

Further, we let the indicator $G_{i,\theta}(x_i|\mathbf{z}) = \delta(G_\theta(\mathbf{z})_i = x_i)$ denote whether the ith variable of the configuration $G_\theta(\mathbf{z})$ is equal to the state x_i. Hereby the function δ equals one if its argument is true and zero otherwise.

By combining our generator given in Equation 2 with the adversarial program sketched in Equation 1 we obtain a program for what we refer to as a *generative adversarial structured network*:

$$\max_{\theta} \min_{\mathbf{w}} \; \ell(\theta, \mathbf{w}, \mathbf{z}) \triangleq \frac{1}{m} \sum_{i=1}^{m} -\log D_{\mathbf{w}}(\mathbf{x}_i) - \log\left(1 - D_{\mathbf{w}}(G_\theta(\mathbf{z}_i))\right), \qquad \text{s.t.} \quad G_\theta(\mathbf{z}) = \arg\max_{\mathbf{x}} F(\theta, \mathbf{z}, \mathbf{x}). \tag{3}$$

It is a bi-level optimization with the lower-level optimization corresponding to a generally NP-hard combinatorial program. In the following, we first review how to obtain a sample $\mathbf{x}$ from the generator before discussing optimization of the bi-level objective given in Equation 3.

3.1 Inference

Finding the highest scoring configuration is equivalent to *maximum a-posteriori* (MAP) inference in a probabilistic graphical model. For a large number of variables N, computing the score for all possible $\mathbf{x}$ is intractable, due to its combinatorial complexity; but for most applications, the scoring function decomposes additively into a sum of *local* scoring functions, f_r, each depending only on a subset of variables, $\mathbf{x}_r$; *i.e.*,

$$F(\theta, \mathbf{z}, \mathbf{x}) \triangleq \sum_{r \in \mathcal{R}} f_r(\theta, \mathbf{z}, \mathbf{x}_r). \tag{4}$$

The subset of variables is summarized in the index set r, which we refer to as a *region*. Moreover, $\mathbf{x}_r$ denotes the restriction of the assignment $\mathbf{x}$ to the variables indexed by r, *i.e.*, $\mathbf{x}_r = (x_i)_{i \in r}$. We use $\mathcal{R}$ to denote the set of all regions in the given application. Often, $\mathcal{R}$ is determined *a priori* by context or domain expertise. For example, if each instance is a sentence, its structure could be a chain of words, wherein $\mathcal{R}$ contains regions for each word and each pair of adjacent words.

To obtain a sample from the generator corresponds to solving the combinatorial optimization in Equation 2, *i.e.*, MAP inference. Taking advantage of the assumed decomposability of the scoring function, F, as given in Equation 4, maximization of F w.r.t. the output assignment, $\mathbf{x}$, is equivalent to the following integer linear program by introducing indicator variables, $b_r(\mathbf{x}_r) \in \{0, 1\}$, $\forall r, \mathbf{x}_r$:

$$\arg\max_{b} \sum_{r, \mathbf{x}_r} b_r(\mathbf{x}_r) f_r(\theta, \mathbf{z}, \mathbf{x}_r) \quad \text{s.t.} \quad \left\{ \begin{array}{ll} b_r(\mathbf{x}_r) \in \{0,1\} & \forall r, \mathbf{x}_r \\ \sum_{\mathbf{x}_r} b_r(\mathbf{x}_r) = 1 & \forall r \\ \sum_{\mathbf{x}_p \setminus \mathbf{x}_r} b_p(\mathbf{x}_p) = b_r(\mathbf{x}_r) & \forall r, \mathbf{x}_r, p \in P(r) \end{array} \right. . \tag{5}$$

Hereby, the set of regions $p \in P(r) = \{p : r \subseteq p \in \mathcal{R}\}$ refers to the set of parents of region r. Strictly speaking, not all super-sets of region r are required, but we omit the details for simplicity of exposition. With b^* as the maximizer of Equation 5, we take $G_{i,\theta}(x_i|\mathbf{z}) = b_i^*(x_i)$ to decode the optimization. A large variety of techniques have been proposed to address this integer linear program or its relaxation (see [16] for a nice review), and any of the proposed techniques can be applied in our case.

3.2 Learning

Learning the parameters, θ and $\mathbf{w}$, involves solving the optimization in Equation 3. Traditionally, this is accomplished by stochastic first-order optimization, which requires the gradient of the loss with respect to the parameters. Using the chain rule, the gradients are given by:

$$\underbrace{\frac{\partial \ell(\theta, \mathbf{w}, \mathbf{z})}{\partial \mathbf{w}}}_{|\mathbf{w}|\times 1} = \underbrace{C\mathbf{w}}_{|\mathbf{w}|\times 1} + \underbrace{\frac{\partial D_{\mathbf{w}}(G_\theta(\mathbf{z}))}{\partial \mathbf{w}}}_{|\mathbf{w}|\times 1} \cdot \underbrace{\frac{\partial \ell(\theta, \mathbf{w}, \mathbf{z})}{\partial D_{\mathbf{w}}(G_\theta(\mathbf{z}))}}_{1\times 1} \tag{6}$$

$$\underbrace{\frac{\partial \ell(\theta, \mathbf{w}, \mathbf{z})}{\partial \theta}}_{|\theta|\times 1} = \underbrace{\frac{\partial F(\theta, \mathbf{z}, \cdot)}{\partial \theta}}_{|\theta|\times L} \cdot \underbrace{\frac{\partial G_\theta(\mathbf{z})}{\partial F(\theta, \mathbf{z}, \cdot)}}_{L\times |G|} \cdot \underbrace{\frac{\partial D_{\mathbf{w}}(G_\theta(\mathbf{z}))}{\partial G_\theta(\mathbf{z})}}_{|G|\times 1} \cdot \underbrace{\frac{\partial \ell(\theta, \mathbf{w}, \mathbf{z})}{\partial D_{\mathbf{w}}(G_\theta(\mathbf{z}))}}_{1\times 1} \tag{7}$$

The dimensions of each term are indicated underneath, in blue. The number of region scores is denoted by $L \triangleq \sum_{r, \mathbf{x}_r} 1$. Note that the number of generator outputs, $|G|$, does not equal L, since G outputs a one-hot encoding of the unary regions only, whereas $\mathcal{R}$ typically contains higher-order regions. Since there are N variables x_i, $i \in \{1, \ldots, N\}$, each arising from a discrete state space $\mathcal{L}$, the generator outputs has cardinality $|G| = N\,|\mathcal{L}|$.

Assuming the loss is smooth, and that the discriminator is a differentiable function of $\mathbf{w}$ (such as a feed-forward neural network), obtaining $\frac{\partial \ell(\theta,\mathbf{w},\mathbf{z})}{\partial \mathbf{w}}$ is trivial. Similarly, if the scoring function is differentiable, obtaining $\frac{\partial F(\theta,\mathbf{z},\cdot)}{\partial \theta}$ and $\frac{\partial D_{\mathbf{w}}(G_\theta(\mathbf{z}))}{\partial G_\theta(\mathbf{z})} \cdot \frac{\partial \ell(\theta,\mathbf{w},\mathbf{z})}{\partial D_{\mathbf{w}}(G_\theta(\mathbf{z}))}$ is trivial. Due to the bi-convex objective, the real challenge is in computing $\frac{\partial G_\theta(\mathbf{z})}{\partial F(\theta,\mathbf{z},\cdot)}$, which we discuss in the sequel.

3.2.1 Differentiating the Generator

Differentiating the generator—that is, computing the term $\frac{\partial G_\theta(\mathbf{z})}{\partial F(\theta,\mathbf{z},\mathbf{x})}$ in Equation 2—involves taking the gradient of the maximizing argument of a discrete program (MAP inference). Unfortunately, this gradient is undefined. To see why, and to find a valid gradient, we consider a "smoothed" variant of the generator task:

$$p^*_{\theta,\epsilon}(\mathbf{x}\|\mathbf{z}) = \arg\max_{p_{\theta,\epsilon}\in\Delta} \sum_{\mathbf{x}} p_{\theta,\epsilon}(\mathbf{x}\|\mathbf{z})\, F(\theta,\mathbf{z},\mathbf{x}) + \epsilon H(p_{\theta,\epsilon}). \quad (8)$$

Hereby, H denotes the Shannon entropy and Δ refers to the probability simplex. Given the maximizing distribution, we can define *marginal* distributions over region r being in state $\mathbf{x}_r$ as

$$p^*_{r,\theta,\epsilon}(\mathbf{x}_r\|\mathbf{z}) = \sum_{\hat{\mathbf{x}}:\hat{\mathbf{x}}_r=\mathbf{x}_r} p^*_{\theta,\epsilon}(\hat{\mathbf{x}}\|\mathbf{z}). \quad (9)$$

We then define the smoothed generator output as the univariate marginal probabilities,

$$\tilde{G}_{i,\theta,\epsilon}(x_i\|\mathbf{z}) = p^*_{i,\theta,\epsilon}(x_i\|\mathbf{z}), \quad \text{and let} \quad \tilde{G}_\theta(\mathbf{z}) = \left(\tilde{G}_{i,\theta,\epsilon}(x_i|\mathbf{z})\right)_{i,\,x_i} \quad (10)$$

denote the concatenation of all univariate marginals.

The following Lemma characterizes the optimal distribution.

Lemma 1 ([16])**.** *The maximizing distribution for the program given in Equation 8 is given by*

$$p^*_{\theta,\epsilon}(\mathbf{x}\|\mathbf{z}) = \frac{1}{Z_\epsilon(\theta,\mathbf{z})}\, \exp\left(\epsilon^{-1}F(\theta,\mathbf{z},\mathbf{x})\right), \quad (11)$$

where $Z_\epsilon(\theta,\mathbf{z})$ denotes a normalization constant, commonly referred to as the partition function.

It is immediately obvious that the *temperature parameter*, $\epsilon \geq 0$, controls the uniformity of the distribution. As $\epsilon \to 0$, the distribution becomes peaked around the MAP assignment. Thus, taking $\epsilon = 0$ retrieves the original generator, $G_{i,\theta}(x_i\|\mathbf{z})$.

We now combine the results of Lemma 1 with our smoothed generator (Equation 10) and the decomposability assumption (Equation 4) to arrive at the following claim.

Claim 1. *Assume that the generator returns the univariate marginals, as defined in Equation 10, and that the scoring function, $F(\theta,\mathbf{z},\mathbf{x})$, decomposes as in Equation 4. We then obtain the following derivative:*

$$\frac{\partial \tilde{G}_{i,\theta,\epsilon}(x_i|\mathbf{z})}{\partial f_r(\theta,\mathbf{z},\hat{\mathbf{x}}_r)} = \frac{1}{\epsilon}\left(p^*_{i,r,\theta,\epsilon}(x_i,\hat{\mathbf{x}}_r\|\mathbf{z}) - p^*_{i,\theta,\epsilon}(x_i\|\mathbf{z})\, p^*_{r,\theta,\epsilon}(\hat{\mathbf{x}}_r\|\mathbf{z})\right), \quad (12)$$

where $p^*_{i,r,\theta,\epsilon}(x_i,\hat{\mathbf{x}}_r\|\mathbf{z})$ denotes the marginal joint probability of variable i in state x_i and region r being in state $\hat{\mathbf{x}}_r$.

Proof. The claim follows by combining the density function in Equation 11 with the decomposition given in Equation 4 and the definition of the smoothed generator, given in Equation 10. A careful calcluation retrieves the claimed result. ■

The quantity on the righthand side of Equation 12 can be recognized as the *covariance* between variable i and region r under the distribution $p^*_{\theta,\epsilon}$. Indeed, the relationship between the covariance and the second derivative of the log-partition function is well known. Moreover, it is immediately clear that the derivative of the original, non-smoothed generator is undefined, since it is the zero-limit of the smoothed generator; i.e.,

$$\frac{\partial G_{i,\theta}(x_i|\mathbf{z})}{\partial f_r(\theta,\mathbf{z},\hat{\mathbf{x}}_r)} = \lim_{\epsilon\to 0} \frac{\partial \tilde{G}_{i,\theta,\epsilon}(x_i\|\mathbf{z})}{\partial f_r(\theta,\mathbf{z},\hat{\mathbf{x}}_r)}.$$

To circumvent this issue, we can simply use the smoothed generator. Computing the marginals output by the smoothed generator is #P-hard in the general case, but it is tractable for certain models (e.g., chains and trees), and can be efficiently approximated using standard inference techniques, such as convex belief propagation [e.g., 17, 7, 10, 14, 15]. Even so, computing the smoothed derivative is non-trivial, due to the need for marginal probabilities of high-order and non-adjacent regions—i.e., the first term in Equation 12. In the following, we discuss a procedure to approximate this quantity.

3.2.2 A General Method for Computing the Generator Derivative

To compute the gradient, we need to compute $p^*_{i,r,\theta,\epsilon}(x_i, \hat{\mathbf{x}}_r \| \mathbf{z})$, the marginal joint distribution of variable i being in state $\hat{x}_i$, and region r being in state $\hat{\mathbf{x}}_r$. Observe that the joint distribution factorizes as

$$p^*_{i,r,\theta,\epsilon}(x_i, \hat{\mathbf{x}}_r \| \mathbf{z}) = p^*_{i,\theta,\epsilon}(x_i \| \mathbf{z})\, p^*_{r,\theta,\epsilon}(\hat{\mathbf{x}}_r \| x_i, \mathbf{z}), \tag{13}$$

which follows from Bayes' rule. The righthand side is the product of two marginals: $p^*_{i,\theta,\epsilon}(x_i \| \mathbf{z})$, the marginal probability of variable i in state x_i, which comes from marginal inference; and $p^*_{r,\theta,\epsilon}(\hat{\mathbf{x}}_r \| x_i, \mathbf{z})$, the marginal of region r in state $\mathbf{x}_r$, given i is in state x_i. This latter term can be computed by conditioning on x_i and re-running marginal inference. Since inference can produce the conditional marginals for all $r \in \mathcal{R}$ and $\mathbf{x}_r$ simultaneously, we only need to run inference once for each i and x_i, resulting in $|G|$ calls to inference.

Conditioning on variable i effectively projects the local scoring functions of its neighboring regions onto the subspace in which i is in state x_i. Thus, to compute the conditional marginals, we simply update the local scoring functions of any region adjacent to the variable being fixed, then run marginal inference. Recall that, in the general case, this task is still #P-hard, but we can use approximate inference.

3.2.3 A Dynamic Programming Algorithm for Chains

For the special case when the graph is a chain, we can use dynamic programming to compute $p^*_{i,r,\theta,\epsilon}(x_i, \hat{\mathbf{x}}_r \| \mathbf{z})$ exactly, using only one call to marginal inference. In a chain, the regions of the graph are the variables and adjacent pairs. We index the pairwise terms by their constituent indices. To simplify notation, we temporarily omit the dependence on ϵ, θ and $\mathbf{z}$, since, for this stage, they are fixed.

We start by assuming that marginal inference has been run once, resulting in $p^*_i(x_i)$ for all $x_i : i = 1, \ldots, N$, and $p^*_{i,i+1}(x_i, x_{i+1})$ for all $x_i, x_{i+1} : i = 1, \ldots, N-1$. Consider any two variables separated by 2 steps, i and $i+2$. To compute the gradient, we need to compute the marginal joint probability $p^*_{i,i+2}(x_i, x_{i+2})$. We also need to compute the marginals for the triad $(i, i+1, i+2)$, which could correspond to a variable i and pair $(i+1, i+2)$, or a pair $(i, i+1)$ and variable $i+2$. Unfortunately, neither $(i, i+2)$ nor $(i, i+1, i+2)$ are regions in a chain graph, so we do not already have this marginal. However, observe that

$$p^*_{i,i+2}(x_i, x_{i+2}) = \sum_{x_{i+1}} p^*_{i,i+1,i+2}(x_i, x_{i+1}, x_{i+2}) \tag{14}$$

$$= \sum_{x_{i+1}} \frac{p^*_{i,i+1}(x_i, x_{i+1})\, p^*_{i+1,i+2}(x_{i+1}, x_{i+2})}{p^*_{i+1}(x_{i+1})}. \tag{15}$$

All of the righthand terms refer to regions in the graph, so their respective marginals are known. Thus, we can compute $p^*_{i,i+1,i+2}(x_i, x_{i+1}, x_{i+2})$, then sum over x_{i+1} to obtain $p^*_{i,i+2}(x_i, x_{i+2})$. Given $p^*_{i,i+2}(x_i, x_{i+2})$, we can use the same identity to compute $p^*_{i,i+2,i+3}(x_i, x_{i+2}, x_{i+3})$ and $p^*_{i,i+3}(x_i, x_{i+3})$:

$$p^*_{i,i+3}(x_i, x_{i+3}) = \sum_{x_{i+2}} p^*_{i,i+2,i+3}(x_i, x_{i+2}, x_{i+3}) \tag{16}$$

$$= \sum_{x_{i+2}} \frac{p^*_{i,i+2}(x_i, x_{i+2})\, p^*_{i+2,i+3}(x_{i+2}, x_{i+3})}{p^*_{i+2}(x_{i+2})}. \tag{17}$$

Algorithm 1 Computes the generator derivate for a chain-structured model.

1: **procedure** CHAINGRADIENT
2: **input:** Unary marginals, $p_i^*(x_i)$, and adjacent pairwise marginals, $p_{i,i+1}^*(x_i, x_{i+1})$.
3: **for** $i = 1, \ldots, N-1$ **do**
4: $\forall x_i,\ \nu_{i,i}(x_i, x_i) = p_i^*(x_i)$
5: $\forall x_i, x_{i+1},\ \nu_{i,i+1}(x_i, x_{i+1}) = p_{i,i+1}^*(x_i, x_{i+1})$
6: $\forall x_t, x_{i+1},\ \nu_{i,i,i+1}(x_i, x_i, x_{i+1}) = p_{i,i+1}^*(x_i, x_{i+1})$
7: **for** $j = i+1, \ldots, N-1$ **do**
8: $\forall x_i, x_j, x_{j+1},\ \nu_{i,j,j+1}(x_i, x_j, x_{j+1}) = \frac{\nu_{i,j}(x_i,x_j)\, p_{j,j+1}^*(x_j,x_{j+1})}{p_j^*(x_j)}$
9: $\forall x_i, x_{j+1},\ \nu_{i,j+1}(x_i, x_{j+1}) = \sum_{x_j} \nu_{i,j,j+1}(x_i, x_j, x_{j+1})$
10: **end for**
11: $\forall x_i, x_{i-1},\ \nu_{i-1,i}(x_{i-1}, x_i) = p_{i-1,i}^*(x_{i-1}, x_i)$
12: $\forall x_i, x_{i-1},\ \nu_{i,i-1,i}(x_i, x_{i-1}, x_i) = p_{i-1,i}^*(x_{i-1}, x_i)$
13: **for** $j = i-1, \ldots, 2$ **do**
14: $\forall x_i, x_{j-1}, x_j,\ \nu_{i,j-1,j}(x_i, x_{j-1}, x_j) = \frac{\nu_{i,j}(x_i,x_j)\, p_{j-1,j}^*(x_{j-1},x_j)}{p_j^*(x_j)}$
15: $\forall x_i, x_{j-1},\ \nu_{i,j-1}(x_i, x_{j-1}) = \sum_{x_j} \nu_{i,j-1,j}(x_i, x_{j-1}, x_j)$
16: **end for**
17: **end for**
18: **for** $i = 1, \ldots, N$ **do**
19: **for** $j = 1, \ldots, N$ **do**
20: $\forall x_i, x_j,\ \frac{\partial G_{i,\theta}(x_i \| \mathbf{z})}{\partial f_j(\theta, \mathbf{z}, x_j)} = \frac{1}{\epsilon}\left(\nu_{i,j}(x_i, x_j) - p_i^*(x_i) p_j^*(x_j)\right)$
21: **if** $j \leq N$ **then**
22: $\forall x_i, x_j, x_{j+1}, \frac{\partial G_{i,\theta}(x_i \| \mathbf{z})}{\partial f_{j,j+1}(\theta, \mathbf{z}, \mathbf{x}_{j,j+1})} = \frac{1}{\epsilon}\left(\nu_{i,j,j+1}(x_i, x_j, x_{j+1}) - p_i^*(x_i) p_{j,j+1}^*(x_j, x_{j+1})\right)$
23: **end if**
24: **end for**
25: **end for**
26: **end procedure**

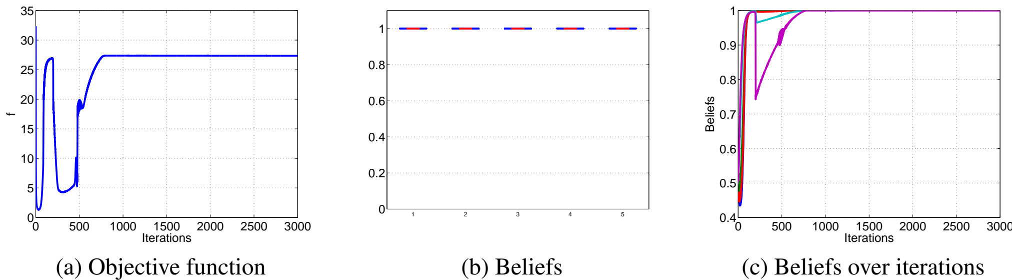
We can therefore compute $p_{i,i+k-1,i+k}^*(x_i, x_{i+k-1}, x_{i+k})$ and $p_{i,i+k}^*(x_i, x_{i+k})$, for any $k > 1$, via dynamic programming. For $i = 1, \ldots, N-1$, we compute $p_{i,j,j+1}^*(x_i, x_j, x_{j+1})$ and $p_{i,j+1}^*(x_i, x_{j+1})$ in a forward pass over $j = i+1, \ldots, N-1$; then, a backward pass fills in $p_{i,j-1,j}^*(x_i, x_{j-1}, x_j)$ and $p_{i,j-1}^*(x_i, x_{j-1})$ for $j = i-1, \ldots, 2$. This algorithm is described in Algorithm 1, which computes the full gradient. All in all, if each variable takes one of $|\mathcal{L}|$ discrete values, and the chain has length N, then the overall time complexity is $\mathrm{O}(N^2 |\mathcal{L}|^3)$.

4 A Simple Example

To demonstrate the applicability of our proposed method, we consider a simple synthetic example. We construct a dataset of binary strings of length 5, wherein each character in the string is always 1. We represent a string by its one-hot encoding.

The generator's inputs are a pair of uniformly random numbers, $\mathbf{z} \sim \mathcal{U}(0,1)^2$. We define the generator as a multilayer perceptron with one hidden layer of width 10 and $\tanh$ activations. The output layer produces the scoring functions, f_r, for each region-state pair in the graph; namely, $(i, x_i)_{i=1}^5$ and $((i, i+1), (x_i, x_j))_{i=1}^4$. The sum of scoring functions yields Equation 4. We use the smoothed generator described in Section 3.2.1, with $\epsilon = 0.1$, which outputs the unary marginal probabilities (Equation 10). The generator's parameters, θ, are the weights of the neural network. Given the constructed distribution of "real" samples, we aim to achieve a θ such that an artificially generated sample, $\tilde{G}_\theta(\mathbf{z})$, always equals $(1,1,1,1,1)$, irrespective of the uniformly drawn inputs.

The task of the discriminator, $D_\mathbf{w}(\mathbf{x})$, is to differentiate between real samples and artificially generated samples. We model the discriminator as another multilayer perceptron, with one hidden layer of width 10 and $\tanh$ activations. The output layer consists of a single neuron with a logistic activation function. The discriminator's parameters, $\mathbf{w}$, are again just the weights of the neural network.



(a) Objective function (b) Beliefs (c) Beliefs over iterations

Figure 1: (a) Convergence of the regularized learning objective (Equation 3); (b) averaged state 1 belief of the generated samples for each of the 5 variables; (c) averaged state 1 belief over the number of iterations.

Combining the generator and discriminator, we obtain the adversarial learning objective in Equation 3. To prevent the discriminator from overfitting, we add a regularization term, $\frac{1}{2}\|\mathbf{w}\|_2^2$, to this objective. We optimize the learning objective using the stochastic gradient method: at each iteration, we draw $m = 10$ new noise samples, $(\mathbf{z}_1, \ldots, \mathbf{z}_m)$, and generate artificial samples, $(G_\theta(\mathbf{z}_1), \ldots, G_\theta(\mathbf{z}_m))$; we then differentiate the discriminator and update its parameters, $\mathbf{w}$, via gradient *descent*; finally, we differentiate the generator (using Algorithm 1) and update its parameters, θ, with gradient *ascent*.

Figure 1 illustrates the results of our synthetic experiment. Figure 1a plots the learning objective as a function of iterations. After less than one thousand iterations, the learning objective converges to an equilibrium between the adversaries. Figure 1b provides a box-plot for the beliefs of the generated samples. Since the provided data samples all had the first states belief equal to one and the second states belief equal to zero, we expect the same behavior from the generated samples. The horizontal axis indicates the variable index and the vertical axis indicates the marginal probability of state the first state. We find that the learned generator's outputs exactly match the data distribution. In Figure 1c we show the mean of the first states belief for the generated samples over the number of iterations. We observe that the computed mean gradually converges to one for all beliefs, a reassuring behavior that our proposed approach is able to pick up the provided signal, despite the more complex computation of the derivative.

5 Related Work

A significant amount of research has been devoted to understanding and improving the training of generative adversarial networks (GANs). The adversarial regime was originally proposed by [4]. Since then, a series of improvements have been proposed. For example, Salimans et al. [13] proposed feature matching, minibatch discrimination, historical averaging of generator parameters, label smoothing, and virtual batch normalization to to improve numerical stability and avoid vanishing gradients. Li et al. [9] introduced *generative moment matching networks*, an approach similar to feature matching that combines the statistical maximum mean discrepancy test [6] with deep networks.

Various architectural variants have been proposed. *Deep convolutional GANs* [12] use convolutional neural networks for the generator, which has been effective in modeling image distributions. Similarly, *recurrent adversarial networks* [8] use recurrent neural networks in the generator to model dependencies between image pixels. The *Laplacian pyramid* concept [3] generates samples in a coarse to fine manner. GANs that maximize the mutual information between the latent variables and the observations [2] were demonstrated to learn disentangled representations. *Energy-based GANs* [18] increase the expressivity of the discriminator by using an energy-based model. While these approaches are related to GASNs, none of them truly optimize the joint assignment of all generator outputs simultaneously, leveraging structural dependencies.

6 Conclusion

In this paper, we proposed a structured extension to the generative adversarial framework, for training generators that jointly infer multiple interdependent outputs. Importantly, we identified the key technical challenge in training these models—namely, differentiating a bi-level optimization, caused by the generator's global optimization. Moreover, we proposed a method to overcome this issue, by introducing a smoothed generator. We also proposed two tractable algorithms for computing the gradient of a smoothed generator. Finally, we conducted a synthetic experiment that validates the efficacy of our approach. These preliminary results encourage us to apply our method to real datasets, which we plan to do in future work.

References

[1] A. Boesen, L. Larsen, S. Sønderby, H. Larochelle, and O. Winther. Autoencoding beyond pixels using a learned similarity metric. In *International Conference on Machine Learning*, 2016.

[2] X. Chen, Y. Duan, R. Houthooft, J. Schulman, I. Sutskever, and P. Abbeel. InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets. In *https://arxiv.org/pdf/1606.03657v1.pdf*, 2016.

[3] E. Denton, S. Chintala, A. Szlam, and R. Fergus. Deep generative image models using a Laplacian pyramid of adversarial networks. In *Neural Information Processing Systems*, 2015.

[4] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Neural Information Processing Systems*, 2014.

[5] S. Gould, B. Fernando, A. Cherian, P. Anderson, R. S. Cruz, and E. Guo. On Differentiating Parameterized Argmin and Argmax Problems with Application to Bi-level Optimization. http://arxiv.org/pdf/1607.05447v2.pdf, 2016.

[6] A. Gretton, K. M. Borgwardt, M. J. Rasch, B. Schölkopf, and A. Smola. A Kernel Two-Sample Test. *JMLR*, 2012.

[7] T. Hazan and A. Shashua. Convergent message-passing algorithms for inference over general graphs with convex free energies. In *Uncertainty in Artificial Intelligence*, 2008.

[8] D. J. Im, C. D. Kim, H. Jiang, and R. Memisevic. Generating images with recurrent adversarial networks. In *https://arxiv.org/abs/1602.05110*, 2016.

[9] Y. Li, K. Swersky, and R. Zemel. Generative moment matching networks. *CoRR*, abs/1502.02761, 2015.

[10] O. Meshi, A. Jaimovich, A. Globerson, and N. Friedman. Convexifying the Bethe free energy. In *Uncertainty in Artificial Intelligence*, 2009.

[11] M. Mirza and S. Osindero. Conditional generative adversarial nets. *CoRR*, abs/1411.1784, 2014.

[12] A. Radford, L. Metz, and S. Chintala. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. In *https://arxiv.org/abs/1511.06434*, 2015.

[13] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen. Improved Techniques for Training GANs. In *https://arxiv.org/abs/1606.03498*, 2016.

[14] A. G. Schwing, T. Hazan, M. Pollefeys, and R. Urtasun. Distributed Message Passing for Large Scale Graphical Models. In *Proc. CVPR*, 2011.

[15] A. G. Schwing, T. Hazan, M. Pollefeys, and R. Urtasun. Globally Convergent Dual MAP LP Relaxation Solvers using Fenchel-Young Margins. In *Proc. NIPS*, 2012.

[16] M. Wainwright and M. Jordan. *Graphical Models, Exponential Families, and Variational Inference*. Now Publishers Inc., 2008.

[17] Y. Weiss, C. Yanover, and T. Meltzer. MAP estimation, linear programming and belief propagation with convex free energies. In *Uncertainty in Artificial Intelligence*, 2007.

[18] J. Zhao, M. Mathieu, and Y. LeCun. Energy-Based Generative Adversarial Network. In *https://arxiv.org/pdf/1609.03126v2.pdf*, 2016.