GANS for Sequences of Discrete Elements with the Gumbel-softmax Distribution

Matt Kusner
Alan Turing Institute
University of Warwick

José Miguel Hernández-Lobato
University of Cambridge

Abstract

Generative Adversarial Networks (GAN) have limitations when the goal is to generate sequences of discrete elements. The reason for this is that samples from a distribution on discrete objects such as the multinomial are not differentiable with respect to the distribution parameters. This problem can be avoided by using the Gumbel-softmax distribution, which is a continuous approximation to a multinomial distribution parameterized in terms of the softmax function. In this work, we evaluate the performance of GANs based on recurrent neural networks with Gumbel-softmax output distributions in the task of generating sequences of discrete elements.

1 Introduction

Generative adversarial networks (GANs) are methods for generating synthetic data with similar statistical properties as the real one [5]. In the GAN methodology a discriminative neural network D is trained to distinguish whether a given data instance is synthetic or real, while a generative network G is jointly trained to confuse D by generating high quality data. This approach has been very sucessful in computer vision tasks for generating samples of natural images [2, 3, 11].

GANs work by propagating gradients back from the discriminator D through the generated samples to the generator G. This is perfectly feasible when the generated data is continuous such as in the examples with images mentioned above. However, a lot of data exists in the form of squences of discrete items. For example, text sentences [1], molecules encoded in the SMILE language [4], etc. In these cases, the discrete data is not differentiable and the backpropagated gradients are always zero.

Discrete data, encoded using a one-hot representation, can be sampled from a multinomial distribution with probabilities given by the output of a softmax function. The resulting sampling process is not differentiable. However, we can obtain a differentiable approximation by sampling from the Gumbel-softmax distribution [8]. This distribution has been previously used to train variatoinal autoencoders with discrete latent variables [8]. Here, we propose to use it to train GANs on sequences of discrete tokens and we evaluate its performance in this setting.

An alternative approach to train GANs on discrete sequences is described in [14]. This method models the generation of the discrete sequence as a stochastic policy in reinforcement learning and bypasses the generator differentiation problem by directly performing gradient policy update.

2 Gumbel-softmax distribution

The softmax function can be used to parameterized a multinomial distribution on a one-hot-encoding d-dimensional vector $\mathbf{y}$ in terms of a continuous d-dimensional vector $\mathbf{h}$. Let $\mathbf{p}$ be a d-dimensional vector of probabilities specifying the multinomial distribution on $\mathbf{y}$ with $p_i = p(y_i = 1), i = 1, \ldots, d$.

Then

$$\mathbf{p} = \text{softmax}(\mathbf{h}) \tag{1}$$

where softmax$(\cdot)$ returns here a d-dimensional vector with the output of the softmax function:

$$[\text{softmax}(\mathbf{h})]_i = \frac{\exp(\mathbf{h}_i)}{\sum_{j=1}^{K} \exp(\mathbf{h}_j)}\,, \quad \text{for} \quad i = 1, \ldots, d\,. \tag{2}$$

It can be shown that sampling $\mathbf{y}$ according to the previous multinomial distribution with probability vector given by (1) is the same as sampling $\mathbf{y}$ according to

$$\mathbf{y} = \text{one\_hot}(\arg\max_i (h_i + g_i))\,, \tag{3}$$

where the g_i are independent and follow a Gumbel distribution with zero location and unit scale.

The sample generated in (3) has gradient zero with respect to $\mathbf{h}$ because the one_hot$(\arg\max(\cdot))$ operator is not differentiable. We propose to approximate this operator with a differentiable function based on the soft-max transformation [8]. In particular, we approximate $\mathbf{y}$ with

$$\mathbf{y} = \text{softmax}(1/\tau(\mathbf{h} + \mathbf{g}))\,, \tag{4}$$

where τ is an inverse temperature parameter. When $\tau \to 0$, the samples generated by (4) have the same distribution as those generated by (3) and when $\tau \to \infty$, the samples are always the uniform probability vector. For positive and finite values of τ the samples generated by (4) are smooth and differentiable with respect to $\mathbf{h}$.

The probability distribution for (4), which is parameterized by τ and $\mathbf{h}$, is called the Gumbel-softmax distribution [8]. A GAN on discrete data can then be trained by using (4), starting with some relatively large τ and then anealing it to zero during training.

3 A recurrent neural network for discrete sequences

In this section we describe how to construct a generative adversarial network (GAN) that is able to generate text from random noise samples. We also give a simple algorithm to train our model, inspired by recent work in adversarial modeling.

An example

Consider the problem of learning to generate simple one-variable arithmetic sequences that can be described by the following context-grammar:

$$S \to x \,||\, S + S \,||\, S - S \,||\, S * S \,||\, S/S$$

where $||$ divides possible productions of the grammar. The above grammar generates sequences of characters such as $x + x - x/x$ and $x - x * x * x * x$.

Our generative model is based on a Long Short Term Memory (LSTM) recurrent neural network [6], shown in the top of Figure 1. The LSTM is trained to predict a hidden-state vector $\mathbf{h}$ at every time-step (i.e., for every character). The softmax operator is then applied to $\mathbf{h}$ as in equations (2) and (1), whic gives a distribution over all possible generated characters (i.e., $x, +, -, /, *$). After training, the network generates data by sampling from the softmax distribution at each time-step.

One way to train the LSTM model to predict future characters is by matching the softmax distribution to a one-hot encoding of the input data via maximum likelihood estimation (MLE). In this work, we are interested in constructing a generative model for discrete sequences, which we will accomplish by sampling through the LSTM, as shown in the bottom of Figure 1. Our generative model takes as input a sample-pair which effectively replace the initial cell and hidden states. From this sample our generator constructs a sequence by successively feeding its predictions as input to the following LSTM unit. Our primary contribution is designing a method to train this generator to generate real-looking discrete sequences.

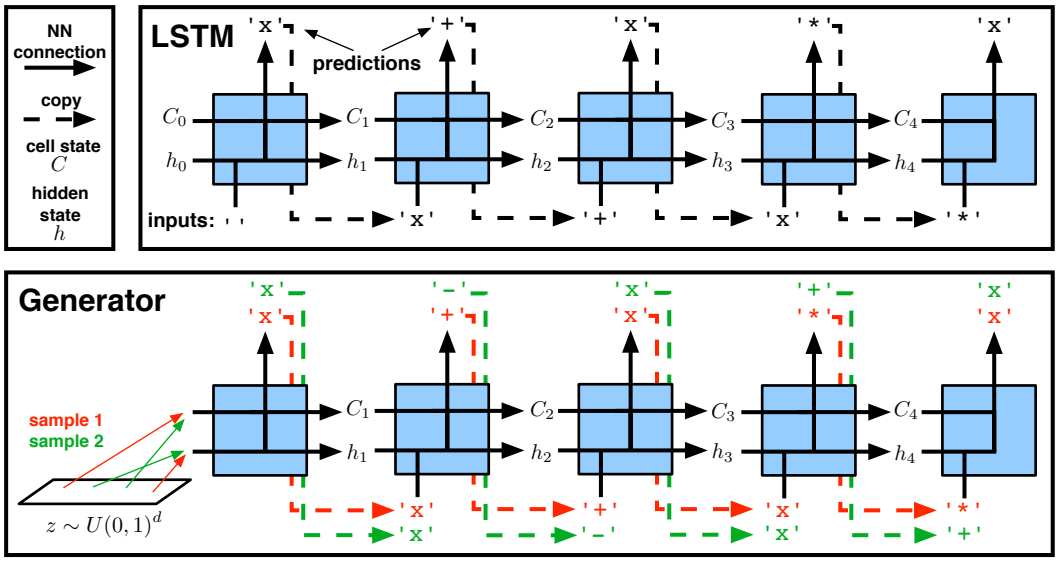

Figure 1: Models to generate simple one-variable arithmetic sequences. (*Top*): The classic LSTM model during the prediction phase. Each LSTM unit (shown as a blue box) makes a prediction based on the input it as seen in the past. This prediction is then used as input to the next unit, which makes its own prediction, and so on. (*Bottom*): Our generative model for discrete sequences. At the beginning we draw a pair of samples which are fed into the network in place of the initial cell state C_0 and hidden state h_0. Our trained network takes these samples and uses them to generate an initial character, this generated character is fed to the next cell in the LSTM as input, and so on.

Generative adversarial modeling

Given a set of n data points $\{\mathbf{x}_1, \mathbf{x}_2 \ldots, \mathbf{x}_n\}$ independently and identically drawn from a d-dimensional distribution $p(\mathbf{x})$ (in our case each $\mathbf{x}$ is a one-hot encoding of a character), the goal of generative modeling is to learn a distribution $q(\mathbf{x})$ that accuratley approximates $p(\mathbf{x})$. The framework of generative adversarial modeling has been shown to yield models $q(\mathbf{x})$ that generate amazingly realistic data points. The adversarial training idea is straight-forward. First, we are going to learn a so-called *generator* G that transforms samples from a simple, known distribution (e.g., a uniform or Gaussian distribution) into samples that approximate those drawn from $p(\mathbf{x})$. Specifically, we define $q(\mathbf{x}) := G(\mathbf{z})$, where $\mathbf{z} \sim U(0,1)^d$ (let $U(0,1)^d$ be the d-dimensional uniform distribution on the interval $[0,1]$). Second, to learn G we will introduce a classifier we call the *discriminator* D. The discriminator takes as input any real d-dimensional vector (this could be a generated input $G(\mathbf{z})$ or a real one $\mathbf{x}$) and predicts the probability that the input is actually drawn from the real distribution $p(\mathbf{x})$. It will be trained to take samples $G(\mathbf{z})$ and real inputs $\mathbf{x}$ and accurately distinguish them. At the same time, the generator G is trained so that it can fool the discriminator D into thinking that a fake point it generated is real with high probability. Initially, the discriminator will be able to easily tell the fake points from the real ones and the generator is poor. However, as training progresses the generator uses this signal from the discriminator to determine how to generate more realistic samples. Eventually, the generator will generate samples so real that the discriminator will have a random chance of guessing if a generated point is real.

Using the Gumbel-softmax distribution

In our case G and D are both LSTMs with parameters Θ and Φ, respectfully. Our aim is to learn G and D by sampling inputs $\mathbf{x}$ and generated points $\mathbf{z}$, and minimizing differentiable loss functions for G and D to update Θ and Φ. Unfortunately, sampling generated points $\mathbf{z}$ from the softmax distribution given by the LSTM, eq. (1), is not differentiable with respect to the hidden states $\mathbf{h}$ (and thus Θ). However, the Gumbel-softmax distribution, eq. (4) is. Equipped with this trick we can take any differentiable loss function and optimize Θ and Φ using gradient-based techniques. We

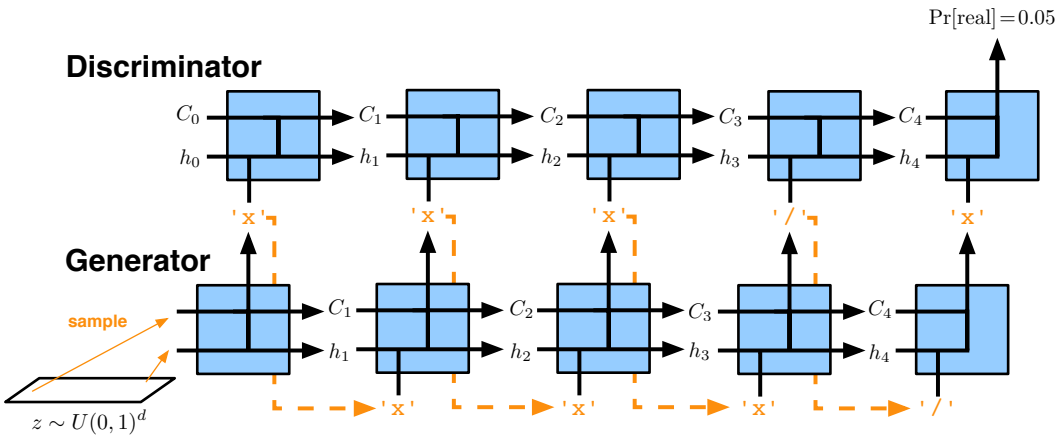

Figure 2: The adversarial training procedure. Our generative model first generates a full-length sequence. This sequence is fed to the discriminator (also a LSTM), which predicts the probability of it being a real sequence. Additionally (not shown), the discriminator is fed real discrete sequence data, which again it predicts the probability of it being real. The weights the networks are modified to make the discriminator better at recognizing real from fake data, and to make the generator better at fooling the discriminator.

describe our adversarial training procedure in Algorithm 1, inspired by recent work on GANs [12]. This algorithm can be shown in expectation to minimize the KL-divergence between $q(\mathbf{z}) = G(\mathbf{z})$ and $p(\mathbf{x})$.

Algorithm 1: Generative Adversarial Network [12]

data: $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \sim p(\mathbf{x})$,
Generative LSTM network G_Θ
Discriminative LSTM network D_Φ
while loop until convergence **do**
  Sample mini-batch of inputs $B = \{\mathbf{x}_{B_1}, \ldots, \mathbf{x}_{B_m}\}$
  Sample noise $N = \{\mathbf{z}_{N_1}, \ldots, \mathbf{z}_{N_m}\}$
  Update discriminator $\Phi = \text{argmin}_\Phi -\frac{1}{m}\sum_{\mathbf{x}\in B} \log D_\Phi(\mathbf{x}) - \frac{1}{m}\sum_{\mathbf{z}\in N} \log(1 - D_\Phi(G_\Theta(\mathbf{z})))$
  Update generator $\Theta = \text{argmin}_\Theta -\frac{1}{m}\sum_{\mathbf{z}\in N} \log \frac{D_\Phi(G_\Theta(\mathbf{z}))}{1-D_\Phi(G_\Theta(\mathbf{z}))}$
end while

Figure 2 shows a schematic of the adversarial training procedure for discrete sequences.

4 Experiments

We now show the power of our adversarial modeling framework for generating discrete sequences. To illustrate this we consider modeling the context-free grammar introduced in Section 3. We generate 5000 samples with a maximum length of 12 characters from the context-free grammar (CFG) for our training set. We pad all sequences with less than 12 characters with spaces.

Optimization details

We train both the discriminator and generator using ADAM [9] with a fixed learning rate of 0.001 and a mini-batch size of $m = 200$. Inspired by the work of [12] who use input noise to stabilize GAN training, for every input $\mathbf{x}$ we form a vector $\mathbf{h}$ such that its softmax (instead of being one-hot) places a probability of approximately 0.9 on the correct character and a probability of $(1 - 0.9)/(d - 1)$ on the remaining $d-1$ characters. We then apply the Gumbel-softmax trick to generate a vector $\mathbf{y}$

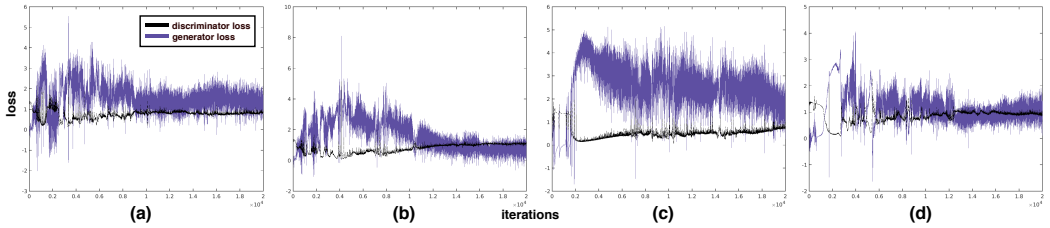

Figure 3: The generative and discriminative losses throughout training. Ideally the loss of the discriminator should increase while the generator should decrease as the generator becomes better at mimicking the real data. **(a)** The default network with Gumbel-softmax temperature annealing. **(b)** The same setting as (a) but increasing the size of the generated samples to $1,000$. **(c)** Only varying the input vector temperature. **(d)** Only introducing random noise into the hidden state and not the cell state.

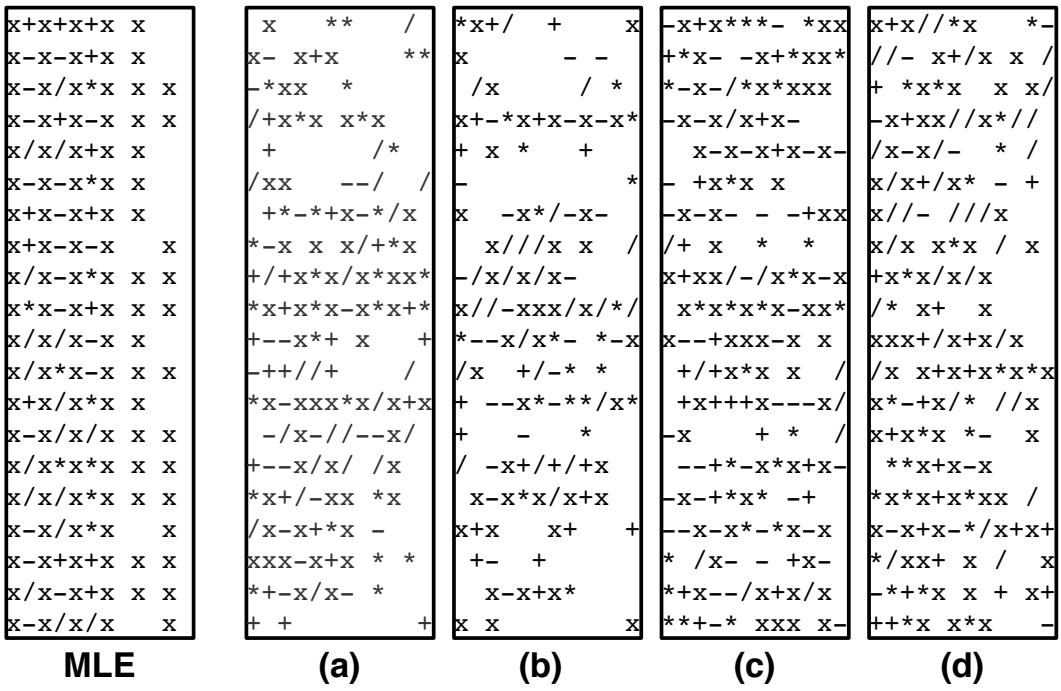

Figure 4: The generated text for MLE and GAN models. The plots **(a)**-**(d)** correspond to the models of Figure 3.

as in equation (4). We use this vector instead of $\mathbf{x}$ throughout training. We train the generator and discriminator for $20,000$ mini-batch iterations. During the training we linearly anneal the temperature of the Gumbel-softmax distribution, from $\tau = 5$ (i.e., a very flat distribution) to $\tau = 1$ (a more peaked distribution) for iterations 1 to $10,000$ and then kept at $\tau = 1$ until training ends.

Learning a CFG

Figure 3 **(a)** shows the generator and discriminator losses throughout training for this setting. We experimented with increasing the size of the generated samples to $1,000$, as this has been reported to improve GAN modeling [7], shown in Figure 3 **(b)**. We also experimented with just varying the temperature for the input vectors $\mathbf{y}$ and fixing the generator temperature to $\tau = 1$ (in Figure 3 **(c)**). Finally, we also tried just introducing random noise into the hidden state and allowing the network to learn an initial cell state C_0 (Figure 3 **(d)**).

Figure 4 shows the text generated by MLE and GAN models. Each row is a sample from either model, each consisting of 12 characters (we have included the blank space character as some training inputs are padded with spaces if less than 12 characters). While the MLE LSTM is not strictly a generative model in the sense of drawing a discrete sequence from a distribution, we include it for reference. We can see that our GAN models are learning to generate alternating sequences of x's, similar to the MLE result. Specifically, the 4th, 10th, and 17th rows of plot **(a)**, show samples that are very close to the training data, and many such examples exist for the remaining plots as well.

We believe that these results, as a proof of concept, show strong promise for training GANs to generate discrete sequence data. Further, we believe that incorporating recent advances in GANs such as training GANs using variational divergence minimization [10] or via density ratio estimation [13] could yield further improvements. We aim to experiment with these in future work.

References

[1] S. R. Bowman, L. Vilnis, O. Vinyals, A. M. Dai, R. Jozefowicz, and S. Bengio. Generating sentences from a continuous space. *In SIGNLL Conference on Computational Natural Language Learning, CONLL*, 2016.

[2] Emily L Denton, Soumith Chintala, Rob Fergus, et al. Deep generative image models using a Laplacian pyramid of adversarial networks. In *Advances in neural information processing systems*, pages 1486–1494, 2015.

[3] Alexey Dosovitskiy and Thomas Brox. Generating images with perceptual similarity metrics based on deep networks. *arXiv preprint arXiv:1602.02644*, 2016.

[4] Rafael Gómez-Bombarelli, David Duvenaud, José Miguel Hernández-Lobato, Jorge Aguilera-Iparraguirre, Timothy D Hirzel, Ryan P Adams, and Alán Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *arXiv preprint arXiv:1610.02415*, 2016.

[5] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, pages 2672–2680, 2014.

[6] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[7] Ferenc Huszár. How (not) to train your generative model: Scheduled sampling, likelihood, adversary? *arXiv preprint arXiv:1511.05101*, 2015.

[8] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

[9] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[10] Sebastian Nowozin, Botond Cseke, and Ryota Tomioka. f-gan: Training generative neural samplers using variational divergence minimization. *arXiv preprint arXiv:1606.00709*, 2016.

[11] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. In *ICLR*, 2016.

[12] Casper Kaae Sønderby, Jose Caballero, Lucas Theis, Wenzhe Shi, and Ferenc Huszár. Amortised map inference for image super-resolution. *arXiv preprint arXiv:1610.04490*, 2016.

[13] Masatoshi Uehara, Issei Sato, Masahiro Suzuki, Kotaro Nakayama, and Yutaka Matsuo. Generative adversarial nets from a density ratio estimation perspective. *arXiv preprint arXiv:1610.02920*, 2016.

[14] Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. Seqgan: Sequence generative adversarial nets with policy gradient. *arXiv preprint arXiv:1609.05473*, 2016.