DISCRETE WASSERSTEIN GENERATIVE ADVERSARIAL NETWORKS (DWGAN)

Anonymous authors
Paper under double-blind review

ABSTRACT

Generating complex discrete distributions remains as one of the challenging problems in machine learning. Existing techniques for generating complex distributions with high degrees of freedom depend on standard generative models like Generative Adversarial Networks (GAN), Wasserstein GAN, and associated variations. Such models are based on an optimization involving the distance between two continuous distributions. We introduce a Discrete Wasserstein GAN (DWGAN) model which is based on a dual formulation of the Wasserstein distance between two discrete distributions. We derive a novel training algorithm and corresponding network architecture based on the formulation. Experimental results are provided for both synthetic discrete data, and real discretized data from MNIST handwritten digits.

1 INTRODUCTION

Generative Adversarial Networks (GAN) (Goodfellow et al., 2014) have gained significant attention in the field of machine learning. The goal of GAN models is to learn how to generate data based on a collection of training samples. The GAN provides a unique training procedure by treating the learning optimization as a two player game between a generator network and discriminator network. Since the learning process involves optimization over two different networks simultaneously, the GAN is hard to train, often times unstable (Salimans et al., 2016). Newly developed models such as the Wasserstein GAN (Arjovsky et al., 2017) aim to improve the training process by leveraging the Wasserstein distance in optimization, as opposed to the Kullback-Leibler or Jensen-Shannon divergences utilized by the original GAN.

A source of interest in generative models arises from natural language processing. In natural language applications, a generative model is necessary to learn complex distributions of text documents. Although both the GAN and Wasserstein GAN approximate a distance between two continuous distributions, and use a continuous sample distance, prior research efforts (Gulrajani et al., 2017; Subramanian et al., 2017; Press et al., 2017) have applied the models to discrete probability distributions advocating for a few modifications. However, using a continuous sample distance for the discrete case may lead to discrepancies. More precisely, as will be demonstrated via explicit examples, a small continuous distance does not necessarily imply a small discrete distance. This observation has potentially serious ramifications for generating accurate natural language text and sentences using GAN models.

To address the above issues, we propose a Discrete Wasserstein GAN (DWGAN) which is directly based on a dual formulation of the Wasserstein distance between two discrete distributions. A principal challenge is to enforce the dual constraints in the corresponding optimization. We derive a novel training algorithm and corresponding network architecture as one possible solution.

2 GENERATIVE ADVERSARIAL NETWORKS (GANS)

Generative Adversarial Networks (GANs) (Goodfellow et al., 2014) model a sample generating distribution by viewing the problem as a two-player game between a generator and a discriminator which is an adversary. The generator takes an input from a random distribution $p(z)$ over a latent variable z, and maps it to the space of data $\mathbf{x}$. The discriminator takes inputs from real data and

Table 1: Example of a mismatch between continuous distance and discrete distance.

TRAINING SAMPLE	GENERATOR'S SOFTMAX OUTPUT	GENERATOR'S SAMPLE (ARGMAX, ONE-HOT)	DISCRETE DISTANCE	CONTINUOUS DISTANCE
$\begin{bmatrix}0 & 0 & 1\\ 0 & 1 & 0\end{bmatrix}$	$\begin{bmatrix}0.3 & 0.3 & 0.4\\ 0.3 & 0.4 & 0.3\end{bmatrix}$	$\begin{bmatrix}0 & 0 & 1\\ 0 & 1 & 0\end{bmatrix}$	0	1.04
$\begin{bmatrix}0 & 0 & 1\\ 0 & 1 & 0\end{bmatrix}$	$\begin{bmatrix}0.1 & 0.1 & 0.8\\ 0.5 & 0.3 & 0.2\end{bmatrix}$	$\begin{bmatrix}0 & 0 & 1\\ 1 & 0 & 0\end{bmatrix}$	1	0.92

Table 2: Example of a large gap between discrete and continuous distances for a discrete sample with 9 classes.

SAMPLES	DISCRETE DISTANCE	CONTINUOUS DISTANCE
Training sample: $\begin{bmatrix}0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0\end{bmatrix}$		
Generator's softmax output: $\begin{bmatrix}0.1 & 0.1 & 0.2 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1\\ 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.2 & 0.1 & 0.1\end{bmatrix}$	0	1.2
Generator's sample (argmax, onehot): $\begin{bmatrix}0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0\end{bmatrix}$		

samples from the generator, and attempts to distinguish between the real and generated samples. Formally, the GAN plays the following two player minimax game:

$$\min_G \max_D \; \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}\left[\log D(\mathbf{x})\right] + \mathbb{E}_{\mathbf{z}\sim p_z(\mathbf{z})}\left[\log(1 - D(G(\mathbf{z})))\right], \tag{1}$$

where D is the discriminator network and G is the generator network. In theory, the GAN approximates the Jensen-Shannon divergence (JSD) between the generated and real data distribution.

Arjovsky et al. (2017) showed that several divergence metrics including the JSD do not always provide usable gradients. Therefore, optimization based on JSD minimization, as incorporated in the GAN, will not converge in certain cases. To overcome the problem, Arjovsky et al. (2017) proposed the Wasserstein GAN which is an approximation to the dual problem of the Wasserstein distance. The authors showed that the Wasserstein distance provides sufficient gradients almost everywhere, and is more robust for training purposes. The dual problem of the Wasserstein distance involves an optimization over all 1-Lipschitz functions (Villani, 2008). The Wasserstein GAN approximates the dual problem by clipping all network weights to ensure that the network represents a k-Lipschitz function for some value of k. A recent variant of the Wasserstein GAN (Gulrajani et al., 2017) enforced the k-Lipschitz property by adding a gradient penalty to the optimization.

Although the formulation of the Wasserstein GAN approximates the Wasserstein distance between two continuous distributions, using a continuous sample distance $\|x - y\|$, existing research efforts (Gulrajani et al., 2017; Subramanian et al., 2017; Press et al., 2017) have directly used it to model discrete probability distributions by adding the following modifications. Each component of the input vectors of training data is encoded in a one-hot representation. A softmax nonlinearity is applied in the last layer of the output of the generator to produce a probability that corresponds with the one-hot representation of the training data. During training, the output of the softmax layers becomes the input to the critic network without any rounding step. To generate a new sample, an argmax operation over each generator's softmax output vectors is applied to produce a valid discrete sample.

The usage of continuous sample distance in the standard Wasserstein GAN for discrete problems as described above creates some discrepancies in the model. These discrepancies are illustrated in Table 1 and Table 2. In Table 1, we have two different outputs from the generator's softmax with the same real sample reference. Although the first softmax output produces the same value as the real sample when it is rounded using argmax (hence has discrete distance 0 to the real sample), it has a larger continuous distance compared to the second softmax output which produces one mistake when rounded (has discrete distance 1 to the real sample). In the discrete case, with a large number

Table 3: Example of the sample distance $d(\mathbf{x}_i, \mathbf{x}_j)$ when $\mathbf{x}_i$, and $\mathbf{x}_j$ consists of two variables where each can takes value from $\{1, 2, 3\}$.

	"11"	**"12"**	**"13"**	**"21"**	**"22"**	**"23"**	**"31"**	**"32"**	**"33"**
"11"	0	1	1	1	2	2	1	2	2
"12"	1	0	1	2	1	2	2	1	2
"13"	1	1	0	2	2	1	2	2	1
"21"	1	2	2	0	1	1	1	2	2
"22"	2	1	2	1	0	1	2	1	2
"23"	2	2	1	1	1	0	2	2	1
"31"	1	2	2	1	2	2	0	1	1
"32"	2	1	2	2	1	2	1	0	1
"33"	2	2	1	2	2	1	1	1	0

of classes, as shown in Table 2, even though the generator output produces a discrete sample with the same value as the real sample when rounded, there still exists a very large continuous distance. This difference between continuous and discrete distance becomes greater for a larger number of discrete classes.

3 Discrete Wasserstein Generative Adversarial Networks (DWGAN)

Motivated to correct modeling discrepancies as described in Section 2, which occur due to the mismatched use of the standard Wasserstein GAN in discrete problems, we propose a new GAN architecture that is directly based on the Wasserstein distance between two discrete distributions.

3.1 Wasserstein distance for discrete distribution

Let a vector $\mathbf{x} = (\mathbf{x}(1), \mathbf{x}(2), \ldots)$ be a discrete multivariate random variable where each component $\mathbf{x}(i)$ can take discrete values from $\{1, 2, 3, \ldots, k\}$. Let P_r and P_s be two probability distributions over the set of values for $\mathbf{x}$. The Wasserstein distance between two probability distributions P_r and P_s is defined as:

$$\mathbf{W}(P_r, P_s) = \min_{\gamma \in \Pi(P_r, P_s)} \mathbb{E}_{(\mathbf{x}, \mathbf{x}') \sim \gamma} \left[d(\mathbf{x}, \mathbf{x}') \right] = \min_{\gamma \in \Pi(P_r, P_s)} \sum_i \sum_j \gamma(\mathbf{x}_i, \mathbf{x}_j) d(\mathbf{x}_i, \mathbf{x}_j). \quad (2)$$

The notation $\Pi(P_r, P_s)$ denotes the set of all joint probability distributions $\gamma(\mathbf{x}, \mathbf{x}')$ whose marginals are P_r and P_s respectively, and $d(\mathbf{x}_i, \mathbf{x}_j)$ denotes the elementary distance between two samples $\mathbf{x}_i$ and $\mathbf{x}_j$. We are particularly interested with the sample distance that is defined as the hamming distance (the sum of zero-one distance of each component), i.e:

$$d(\mathbf{x}_i, \mathbf{x}_j) = \sum_k \mathbb{I}(\mathbf{x}_i(k) \neq \mathbf{x}_j(k)). \quad (3)$$

Table 3 shows an example of the sample distance metric.

Visible in the formulation above, computing the Wasserstein distance between two discrete probability distributions is a Linear Program (LP) problem for which the runtime is polynomial with respect to the size of problem. However, for generating real-world discrete distributions, the size of problem grows exponentially. For example, if the number of variables in vector $\mathbf{x}$ is 100, and each variable can take values in the set $\{1, 2, \ldots, 10\}$ so that $k = 10$, the size of the LP problem is $O(10^{100})$ reflecting the number of configurations for $\mathbf{x}$. The resulting LP is intractable to solve.

We follow a similar approach as in Arjovsky et al. (2017) by considering the dual formulation of Wasserstein distance. Kantorovich duality (Evans, 1997; Villani, 2008) tells us that the dual linear program of the Wasserstein distance can computed as:

$$\max_f \mathbb{E}_{\mathbf{x} \sim P_r} \left[f(\mathbf{x}) \right] - \mathbb{E}_{\mathbf{x} \sim P_s} \left[f(\mathbf{x}) \right] \quad (4)$$

$$\text{subject to: } f(\mathbf{x}_i) - f(\mathbf{x}_j) \leq d(\mathbf{x}_i . \mathbf{x}_j), \quad (5)$$

The function f maps a sample to a real value. Note that unlike for the continuous Wasserstein distance, in which the maximization is over all 1-Lipschitz functions without additional constraints, the maximization above is over all functions that satisfy the inequality constraints in Eq. 5.

3.2 DWGAN ARCHITECTURE AND LEARNING ALGORITHM

The dual formulation of the Wasserstein distance is still intractable since the maximization is over all functions that satisfy the inequality constraints. We aim to approximate the dual Wasserstein distance formulation by replacing f with a family of parameterized functions f_w that satisfy the inequality constraints. The parameterized functions f_w are modeled using a neural network. Unfortunately, it is difficult to construct a neural network architecture to model f_w while also explicitly satisfying the inequality constraints involving the discrete sample distance defined in Eq. 3.

To overcome the problem of approximating f with neural networks, we note that the maximization in the dual formulation is equivalent to the following optimization:

$$\max_{h} \mathbb{E}_{(\mathbf{x},\mathbf{x}')\sim(P_r,P_s)} \left[h(\mathbf{x},\mathbf{x}')\right] \tag{6}$$

$$\text{subject to: } h(\mathbf{x}_i,\mathbf{x}_j) \leq d(\mathbf{x}_i,\mathbf{x}_j), \tag{7}$$

where $h(\mathbf{x},\mathbf{x}') = f(\mathbf{x}) - f(\mathbf{x}')$. Instead of approximating $f(\mathbf{x})$, we aim to design a neural network architecture that approximates $h(\mathbf{x},\mathbf{x}')$ and satisfies the inequality constraints in Eq. 5. The key idea is that this new optimization is equivalent to the original dual formulation of the Wasserstein distance (explained in the sequel), even though the optimal form for h is not explicitly specified.

Our selected architecture for the generator network employs the same softmax nonlinearity trick for the standard Wasserstein GAN described in Section 2. The generator network is a parameterized function g_θ that maps random noise z to a sample in one-hot representation. The last layer of the generator network utilizes softmax nonlinearity to produce a probability which corresponds with the one-hot representation of the real sample. Our key modeling difference lies in the critic network. The critic network takes two inputs, one from the real samples, and one from the output of the generator. The architecture of the critic network is visualized in Figure 1.

Let $\mathbf{y} \in [0,1]^{m\times k}$ be the one-hot representation of $\mathbf{x}$ where m is the number of variables and k is the number of classes for each variable. The critic network takes two inputs: $\mathbf{y}$ from the real training data, and $\mathbf{y}'$ from the output of the generator network. Let us define ρ_w as a parameterized function that takes input $(\mathbf{y},\mathbf{y}') \in [0,1]^{2\times m\times k}$ and produces an output vector $\mathbf{v} \in [-1,1]^m$. From the generator output $\mathbf{y}'$, we compute the rounded sample $\tilde{\mathbf{x}}'$. Let $\mathbf{u} \in \{0,1\}^m$ be a vector that contains the element-wise zero one distance between a real training sample $\mathbf{x}$ and rounded sample $\tilde{\mathbf{x}}'$ from the generator, i.e. $\mathbf{u}(i) = \mathbb{I}(\mathbf{x}(i) \neq \tilde{\mathbf{x}}'(i))$. We define our approximation to the function h as a parameterized function h_w that is defined as $h_w = \mathbf{u}^T\mathbf{v} = \mathbf{u}^T\rho_w(\mathbf{y},\mathbf{y}')$. The "filter" vector $\mathbf{u}$ ensures that the output of h_w always satisfies the inequality constraints $h_w(\texttt{onehot}(\mathbf{x}_i),\texttt{onehot}(\mathbf{x}_j)) \leq d(\mathbf{x}_i,\mathbf{x}_j)$ as stated in Eq. 5. An illustration of this neural network architecture and construction is provided in Figure 1.

As we can see from Figure 1, the critic network consists of two separate sub-networks. The first sub-network takes input from a batch of samples of the training data, while the second sub-network takes input from a batch of samples produced by the generator. Each sub-network has its own set of intermediate layers. The outputs of the first and second layers are concatenated and taken as an input to a fully connected layer which produces a tensor of size $n \times m$. The dimension n indicates the number of samples in a batch, and m is the number of variables. To produce a tensor $\mathbf{v}$ whose values range from -1 to 1, a `tanh` nonlinearity is applied. The "filter" tensor $\mathbf{u}$ is applied to $\mathbf{v}$ via an element-wise multiplication. The output of the critic network is calculated by taking the sum of the result of the element-wise multiplication of $\mathbf{u}$ and $\mathbf{v}$, yielding a vector of n elements containing the value of $h_w(\mathbf{y},\mathbf{y}')$ for each pair of real and generated samples.

We also included additional modifications based on theory to facilitate the training of networks. Note that since $h(\mathbf{x},\mathbf{x}') = f(\mathbf{x}) - f(\mathbf{x}')$, we can attempt to enforce this optimum condition known from theory. If we flip the inputs to h_w we will get the negative of the output; i.e. $h_w(\mathbf{y}',\mathbf{y}) = -h_w(\mathbf{y},\mathbf{y}')$. To model this fact, we randomly swapped the sample from the real training data and generator output so that some of the real data was fed to the first sub-network and some to the second sub-network. If a pair of samples was flipped, we multiplied the output of the network with -1. Another modification that we applied to the network was to introduce a scaling factor to the softmax function such that the output of the scaled softmax was closer to zero or one. Specifically, we applied the function: $\texttt{softmax}(\mathbf{x})(i) = \frac{\exp(k\cdot\mathbf{x}(i))}{\sum_j \exp(k\cdot\mathbf{x}(j))}$, for some constant $k \geq 1$. The training algorithm for our proposed architecture is described in Algorithm 1.

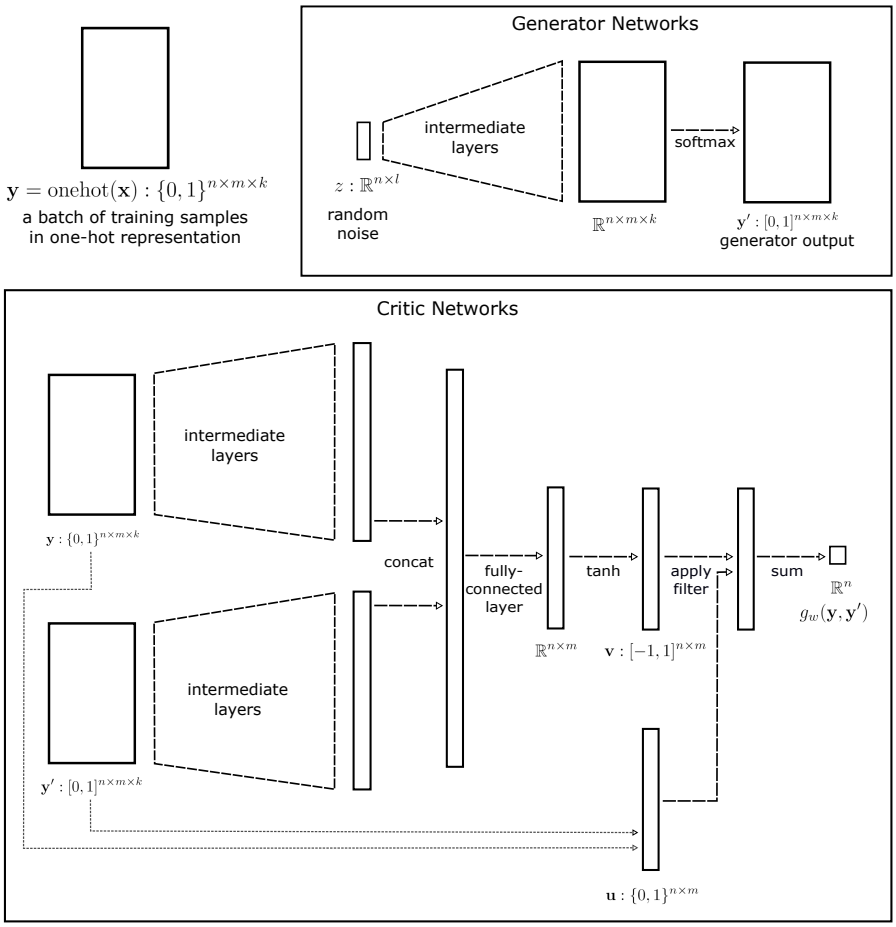


Figure 1: An example of the Discrete Wasserstein GAN architecture. The tensor dimensions indicate a batch of samples. In the architecture, n, m, k, l denote the number of samples in a batch, the number of variables, the number of classes, and the number of noise variables respectively.

Algorithm 1 Discrete Wasserstein GAN

Input: learning rate α, batch size n, the number of critic iteration per generator iteration n_{critic}
repeat
  for $t = 1, \ldots, n_{\text{critic}}$ **do**
    Sample a batch from real data $\{\mathbf{x}_i\}_{i=1}^n \sim P_r$
    Sample a batch of random noise $\{z_i\}_{i=1}^n \sim p(z)$
    $w \leftarrow w + \alpha \cdot \nabla_w [\frac{1}{n} \sum_{i=1}^n h_w(\mathbf{x}_i, g_\theta(z_i))]$
  end for
  Sample a batch from real data $\{\mathbf{x}_i\}_{i=1}^n \sim P_r$
  Sample a batch of random noise $\{z_i\}_{i=1}^n \sim p(z)$
  $\theta \leftarrow \theta - \alpha \cdot \nabla_\theta [\frac{1}{n} \sum_{i=1}^n h_w(\mathbf{x}_i, g_\theta(z_i))]$
until converge

4 Related works

In contrast with the continuous GANs where many models have been proposed to improve the performance of GAN training, only a few GAN formulations have been proposed for modeling discrete probability distributions. Gulrajani et al. (2017) use the standard continuous Wassersten GAN with adjustments described in Section 2. Similar techniques are used by Subramanian et al. (2017) to address several natural language generation tasks. Che et al. (2017) augment the original GAN

architecture with a maximum likelihood technique and combine the discriminator output with importance sampling from the maximum likelihood training. Hjelm et al. (2017) propose a Boundary-seeking GAN (BGAN) that trains the generator to produce samples that lie in the decision boundary of the discriminator. BGAN can be applied for discrete cases provided that the generator outputs a parametric conditional distribution. Other GAN models (Yu et al., 2016; Li et al., 2017) exploit the REINFORCE policy gradient algorithm (Williams, 1992) to overcome the difficulty of back-propagation in the discrete setting. Kim et al. (2017) combine adversarial training with Variational Autoencoders (Kingma & Welling, 2014) to model discrete probability distributions.

5 EXPERIMENTS

5.1 SYNTHETIC EXPERIMENTS WITH OBJECTIVE EVALUATION

Evaluating the performance of generative models objectively and effectively is hard, since it is difficult to automatically tell whether a generated sample is a valid sample from the real distribution. Previous research advocates user studies with human graders, especially in image generation tasks, or proxy measures like perplexity and corpus-level BLEU in natural language generation. However, such techniques are far from ideal to objectively evaluate the performance of GAN models.

To address the limitations above, we propose a synthetic experiment that captures the complexity of modeling discrete distributions, but still has a simple strategy to objectively evaluate performance. The synthetic experiment is based on a classic tic-tac-toe game. We generalize the classic 2 player tic-tac-toe game to include arbitrary k players and arbitrary m-by-m board sizes (rather than the default 3-by-3 board). The goal is to model the true generating distribution P_r which is the uniform distribution over **valid** configurations of the board when a generalized tic-tac-toe game has ended (e.g. the final game state). We generalized the concept of a valid board in 3-by-3 games, in which one player has a winning state and marks filling a full column, row, or diagonal. For the purpose of our experiment, we made a simplification to the valid rule, i.e. as long as the board has at least one full column, row and diagonal taken by at least one player, it is considered to be a valid configuration. Figure 2 shows examples of valid and non-valid board configurations.

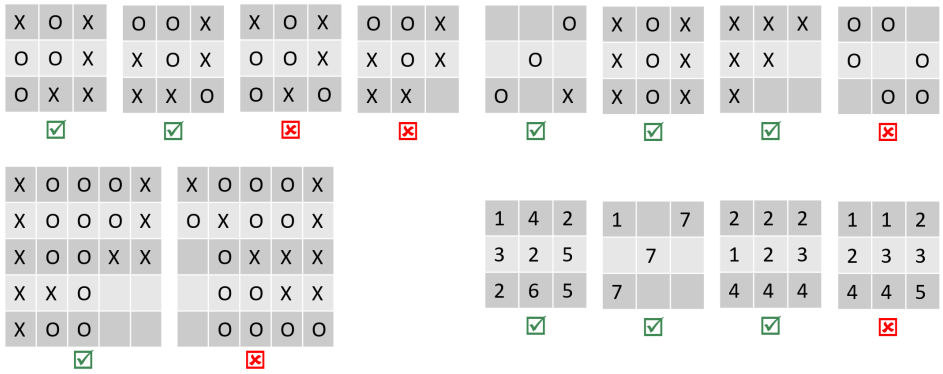


Figure 2: Examples of valid and non-valid board configurations of the generalized and simplified tic-tac-toe game with multiple players.

In our construction above, it is easy to check if a generated sample is a valid sample under the real distribution. Hence it is possible to validate objectively the performance of a generative model. Furthermore, it is also easy to sample from the real distribution to create synthetic training data. We uniformly sample random board configurations, accepting a sample if it is valid, and rejecting it if invalid. We construct several metrics to track the performance

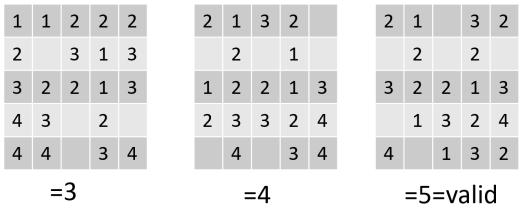
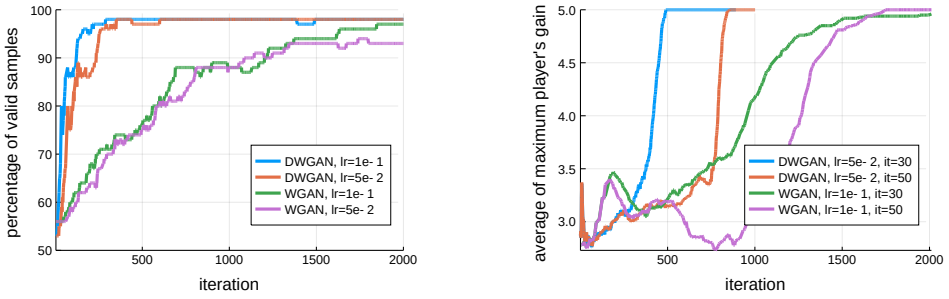
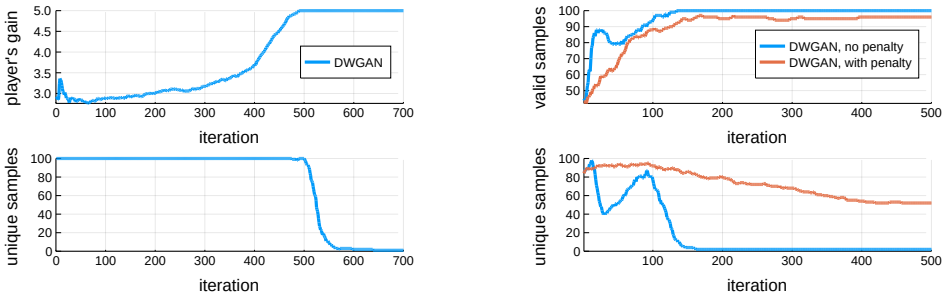


Figure 3: The example of maximum player's gain for three different board configurations.



(a) (b)

Figure 4: Comparison between Discrete-WGAN (DWGAN) and the standard WGAN: (a) percentage of valid samples for a 3-by-3 board with 2 players, (b) average if maximum player's gain for a 5-by-5 board with 8 players. The best results for both networks over several learning rates (`lr`) and the number of iterations (`it`) for each learning rate decay are presented.



(a) (b)

Figure 5: Examples of mode collapse in Discrete-WGAN (DWGAN) on: (a) 5-by-5 board with 8 players, (b) 3-by-3 board with 2 players and the effect of adding a norm penalty.

of the model. The first measure is the **percentage of valid samples** which characterizes the quality of the samples generated by the generator network. For a bigger board the percentage of valid samples does not tell much about the progress of learning since it takes a while to get to a valid sample. We construct another metric which is the **average of maximum player's gain**. The maximum player's gain for a board configuration is defined as the maximum number of cells taken by a player in a full column, row, or diagonal. Figure 3 shows the value of maximum player's gain for three different 5-by-5 board configurations. In the left board, player 2 and 4 have the maximum (3 cells); in the middle board player 2 takes 4 cells; and in the right board, player 2 achieves the maximum of 5 cells. Note that for k-by-k boards, if the average of maximum player's gain is equal to k, it means that all the samples are valid. Therefore, closer average of maximum player's gain to k indicates a better quality of samples. Besides those two metrics, we also track the **percentage of unique samples** and the **percentage of new samples**, i.e. samples that do not appear in the training data.

In the experiment, we compare our Discrete Wasserstein GAN model with the standard Wasserstein GAN model (with tricks described in Section 2) on 3-by-3 and 5-by-5 board with 2 players and 8 players. Note that the number of classes is equal to the number of players plus one since we need an additional class for encoding empty cells. We restrict the generator and critic networks in both models to have a single hidden layer within fully connected networks to ease training. As we can see from Figure 4, our DWGAN networks achieve good performance (in terms of the average of the percentage of valid samples and the maximum player's gain metrics) much faster than the standard WGAN with softmax and one-hot representation tricks. In both 3-by-3 boards with 2 players and 5-by-5 boards with 8 players our DWGAN networks only take less than a third of the iterations taken by the standard WGAN to achieve similar performance.

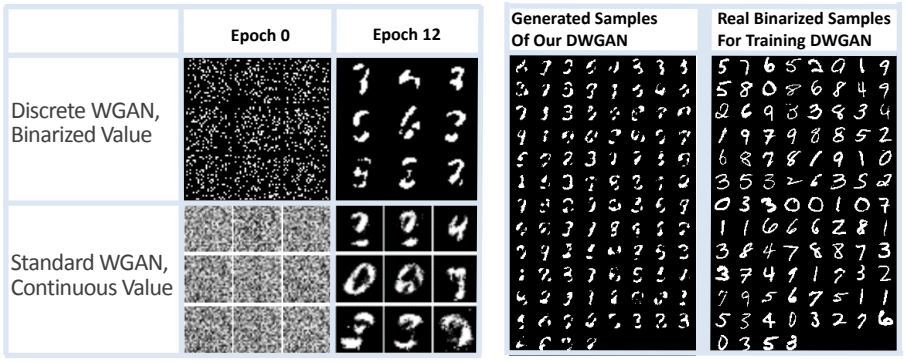


Figure 6: An example of Discrete WGAN with binary values vs. Standard WGAN with continuous values applied to generate MNIST handwritten digits. Both models feature 1 hidden layer for both the generator and critic within a fully-connected network. Modeling complex discrete distributions with GANs still requires future refinements in optimization, training, and stability.

We observe that our DWGAN networks have a mode collapse problem that occurs after achieving top performances. Figure 5a shows that the DWGAN can achieve the average of maximum player's gain close to 5 for a 5-by-5 board in 500 iterations while maintaining the percentage of unique samples close to 100%. After it produces those diverse samples, the network model begins to suffer from a mode collapse and the percentage of unique samples decrease to less than 10% after iteration 550. Based on our analysis, this behavior is caused by the fact that the network optimizes the function difference $\frac{1}{n}\sum_{i=1}^{n} h_w(\mathbf{x}_i, g_\theta(z_i)) = \frac{1}{n}\sum_{i=1}^{n}[f(\mathbf{x}_i) - f(g_\theta(z_i))]$, which tends to cause an advantage if the values of $g_\theta(z_i)$ are not diverse. To overcome this issue, we add a norm penalty to the critic network optimization, i.e:

$$\frac{1}{n}\sum_{i=1}^{n} h_w(\mathbf{x}_i, g_\theta(z_i)) + \lambda\sqrt{\sum_{i=1}^{n} h_w(\mathbf{x}_i, g_\theta(z_i))^2}, \tag{8}$$

where λ is the penalty constant. Figure 5b shows the effect of the norm penalty to the performance of DWGAN and its sample diversity. We observe that the DWGAN network with a norm penalty can achieve 96% valid samples while maintaining the diversity in samples it generates (around 50% unique samples).

5.2 Experiments with Real Data

To model more complex discrete distributions, we used MNIST digits discretized to binary values (LeCun et al., 1998) as the training data with the goal to generate new digits with our proposed Discrete Wasserstein GAN. As a baseline, we trained a standard Wasserstein GAN on the continuous digit dataset. Similar to our synthetic experiments, we restricted the generator and critic networks to have only a single hidden layer within fully connected networks. Figure 6 shows that our model produces a similar quality of discretized digit images compared to the continuous value digits produced by the standard Wasserstein GAN trained on continuous-valued data. We further generated 100 samples from our DWGAN model, prior to mode collapse, illustrating the diversity of samples.

6 Conclusion

We proposed the Discrete Wasserstein GAN (DWGAN) which approximates the Wasserstein distance between two discrete distributions. We derived a novel training algorithm and corresponding network architecture for a dual formulation to the problem, and presented promising experimental results. Our future work focuses on exploring techniques to improve the stability of the training process, and applying our model to other datasets such as for natural language processing.

REFERENCES

Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In *International Conference on Machine Learning*, pp. 214–223, 2017.

Tong Che, Yanran Li, Ruixiang Zhang, R Devon Hjelm, Wenjie Li, Yangqiu Song, and Yoshua Bengio. Maximum-likelihood augmented discrete generative adversarial networks. *arXiv preprint arXiv:1702.07983*, 2017.

Lawrence C Evans. Partial differential equations and monge-kantorovich mass transfer. *Current developments in mathematics*, 1997(1):65–126, 1997.

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*, pp. 2672–2680, 2014.

Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. Improved training of wasserstein gans. *arXiv preprint arXiv:1704.00028*, 2017.

R Devon Hjelm, Athul Paul Jacob, Tong Che, Kyunghyun Cho, and Yoshua Bengio. Boundary-seeking generative adversarial networks. *arXiv preprint arXiv:1702.08431*, 2017.

Yoon Kim, Kelly Zhang, Alexander M Rush, Yann LeCun, et al. Adversarially regularized autoencoders for generating discrete structures. *arXiv preprint arXiv:1706.04223*, 2017.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *Proceeding of International Conference on Learning Representations 2014*, 2014.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Jiwei Li, Will Monroe, Tianlin Shi, Alan Ritter, and Dan Jurafsky. Adversarial learning for neural dialogue generation. *arXiv preprint arXiv:1701.06547*, 2017.

Ofir Press, Amir Bar, Ben Bogin, Jonathan Berant, and Lior Wolf. Language generation with recurrent generative adversarial networks without pre-training. *arXiv preprint arXiv:1706.01399*, 2017.

Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. In *Advances in Neural Information Processing Systems*, pp. 2234–2242, 2016.

Sandeep Subramanian, Sai Rajeswar, Francis Dutil, Christopher Pal, and Aaron Courville. Adversarial generation of natural language. *ACL 2017*, pp. 241, 2017.

Cédric Villani. *Optimal transport: old and new*, volume 338. Springer Science & Business Media, 2008.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu Seqgan. Sequence generative adversarial nets with policy gradient. arxiv preprint. *arXiv preprint arXiv:1609.05473*, 2(3):5, 2016.

SUPPLEMENTARY MATERIALS

A DISCRETE WASSERSTEIN DISTANCE

A.1 STANDARD LP AND DUAL LP CONVEX OPTIMIZATION PROBLEMS

A linear program (LP) is a convex optimization problem in which the objective and constraint functions are linear. Consider a vector variable $\mathbf{x} \in \mathbb{R}^n$, matrix $A \in \mathbb{R}^{n \times m}$, and vectors $\mathbf{c} \in \mathbb{R}^n$, and $b \in \mathbb{R}^m$. An LP is given in the following standard form,

$$\begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ \text{subject to} \quad & A\mathbf{x} = \mathbf{b}, \\ & \mathbf{x} \succeq 0. \end{aligned}$$

The Lagrange dual function is given by,

$$g(\lambda, \nu) \triangleq \begin{cases} -\mathbf{b}^T \nu, & A^T \nu - \lambda + \mathbf{c} = 0; \\ 0, & \text{otherwise.} \end{cases}$$

The Lagrange dual problem is to maximize $g(\lambda, \nu)$ subject to $\lambda \succeq 0$. Equivalently, the dual problem may be written as an LP in inequality form with vector variable $\nu \in \mathbb{R}^m$,

$$\begin{aligned} \max \quad & -\mathbf{b}^T \nu \\ \text{subject to} \quad & A^T \nu + \mathbf{c} \succeq 0. \end{aligned}$$

The dual of the above problem is equivalent to the original LP in standard form. Due to the weaker form of Slater's condition, strong duality holds for any LP in standard or inequality form provided that the primal problem is feasible. Similarly, strong duality holds for LPs if the dual is feasible.

A.2 LP FOR WASSERSTEIN DISTANCE OF DISCRETE PROBABILITY DISTRIBUTIONS

Consider discrete probability distributions over a finite set $\mathcal{X}$ with cardinality $|\mathcal{X}|$. Assume an elementary sample distance $d_{\mathcal{X}}(x_1, x_2)$ for $x_1, x_2 \in \mathcal{X}$. The sample distance evaluates the semantic similarity between members of set $\mathcal{X}$. Define $P_r(x)$ and $P_s(x)$ for $x \in \mathcal{X}$ as two discrete probability distributions. In this case, we may define the exact discrete Wasserstein distance between P_r and P_s as a linear program as follows, with $D \in \mathbb{R}_+^{|\mathcal{X}| \times |\mathcal{X}|}$ whose matrix entries correspond to the sample distance $D_{x_1, x_2} = d_{\mathcal{X}}(x_1, x_2)$.

$$\begin{aligned} W(P_r, P_s) = & \inf_{T \in \mathbb{R}_+^{|\mathcal{X}| \times |\mathcal{X}|}} \langle T, D \rangle, \\ \text{subject to} \quad & T1 = P_r,\ T^T 1 = P_s. \end{aligned}$$

The dual LP is given as follows.

$$\begin{aligned} W_{dual}(P_r, P_s) = & \sup_{\nu \in \mathbb{R}^{|\mathcal{X}|},\ \mu \in \mathbb{R}^{|\mathcal{X}|}} \nu^T P_r + \mu^T P_s, \\ \text{subject to} \quad & \nu(i) + \mu(j) \leq D_{i,j}. \end{aligned}$$

At the optimum it is known that $\nu = -\mu$, and the dual LP is equivalent to the following optimization problem. Note that there still exist $|\mathcal{X}| \times |\mathcal{X}|$ constraints.

$$\begin{aligned} W^*_{dual}(P_r, P_s) = & \sup_{\nu \in \mathbb{R}^{|\mathcal{X}|}} \nu^T (P_r - P_s), \\ \text{subject to} \quad & \nu(i) - \nu(j) \leq D_{i,j}. \end{aligned}$$

Example 1. *The following example provides a closer look at the dual optimization problem. Consider a finite set $\mathcal{X} = \{1, 2, 3\}$. Let $P_s(x)$ be given by the discrete distribution $P_s(1) = 0.2$, $P_s(2) = 0.7$ and $P_s(3) = 0.1$. Similarly, let $P_r(x)$ be given by the discrete distribution $P_r(1) = 0.4$, $P_r(2) = 0.4$, $P_r(3) = 0.2$. Define the elementary sample distance $d_{\mathcal{X}}(x_1, x_2) = 1$ if $x_1 \neq x_2$ and*

$d_{\mathcal{X}}(x_1, x_2) = 0$ *if* $x_1 = x_2$. *Therefore, the sample distance matrix* D *for this discrete example is the following:*

$$D = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}.$$

The optimal value of the matrix T *provides the optimal transport of mass from* P_s *to* P_r*,*

$$T = \begin{bmatrix} 0.2 & 0.2 & 0.0 \\ 0.0 & 0.4 & 0.0 \\ 0.0 & 0.1 & 0.1 \end{bmatrix}.$$

The objective value of the primal and dual is equal to 0.3 *which is the total mass moved from* P_s *to* P_r*. In the solution to the dual problem,* $\nu = \begin{bmatrix} 0 & -1 & 0 \end{bmatrix}^T$ *and* $\mu = -\nu$*. In this example, it is seen that the optimal* $\nu = -\mu$*.*

B Source Code Implementation

B.1 Synthetic Experiments

For the synthetic experiments, we use **Julia v0.5.2** programming language with **Knet** deep learning framework. Below is the code containing functions needed to generate tic-tac-toe board.

```
# check if a grid configuration is valid
function valid(D::Matrix)
    k = size(D, 1)       # k = grid size
    np = Int(maximum(D))      # np = number of player
    for i = 1:np
        if k in sum(D .== i, 1)                          # vertical
            return true
        elseif k in sum(D .== i, 2)                      # horizontal
            return true
        elseif sum(diag(D .== i)) == k                   # diagonal
            return true
        elseif sum(diag(flipdim(D, 2) .== i)) == k       # flipped diagonal
            return true
        end
    end
    return false
end

# convert grid to one-hot tensor
function onehot(D::Matrix, np::Integer=Int(maximum(D)))
    k = size(D, 1)       # grid size
    nc = np + 1          # number of class (0 for empty, 1... for players)
    Dh = zeros(Float32,k,k,nc)
    for i = 0:np
        Dh[:,:,i+1] = Int.(D .== i)
    end
    return Dh
end

# convert one-hot tensor to grid
function revonehot(Dh)
    k = size(Dh, 1)
    np = Int(maximum(Dh))
    D = zeros(Float32,k,k)
    for i = 1:np
        D += i * Dh[:,:,i+1]
    end
    return D
end
```

```
# generate valid grid samples
function generate_samples(k::Integer, n::Integer, np::Integer)
    nc = np + 1
    X = zeros(Float32,k,k,nc,n)
    i = 0
    trial = 0
    tic()
    while i < n
        trial += 1
        r = rand(0:np, (k,k))
        if valid(r)
            i += 1
            X[:,:,:,i] = onehot(r, np)

            if i % 10 == 0
                println(i)
                toc()
                tic()
            end
        end
    end
    toc()
    return X
end

# vectorize sample
function vec_ttt(X)
    sz = size(X)
    return reshape(X, (sz[1] * sz[2] * sz[3], sz[end]))
end

# devectorize sample
function devec_ttt(X, np)
    sz = size(X)
    nc = np + 1
    k = Int(sqrt(sz[1] / nc))
    return reshape(X, (k, k, nc, sz[end]))
end

# convert softmax output to grid
function to_ttt(X)
    sz = size(X)
    T = zeros(Int, sz[1], sz[2], 1, sz[end])
    for i = 1:sz[1]
        for j = 1:sz[2]
            for k = 1:sz[end]
                T[i,j,1,k] = indmax(X[i,j,:,k]) - 1
            end
        end
    end
    return T
end

# convert vectorized softmax output to vectorized grid
function round_sample(X, np)
    D = devec_ttt(X, np)
    T = to_ttt(D)
    sz = size(T)
    nc = np + 1
    TX = zeros(Float32,sz[1],sz[2],nc,sz[end])
    for i = 1:sz[end]
        TX[:,:,:,i] = onehot(T[:,:,1,i], np)
    end
    return vec_ttt(TX)
end
```

```
# convert vectorized softmax output to vectorized grid for KnetArray
function round_sample_knet(X, np)
    D = devec_ttt(X, np)
    mD = maximum(D, 3)
    T = D .== mD
    return vec_ttt(T)
end

# print samples
function print_ttt(T, cols=20)
    sz = size(T)
    n = sz[end]
    it = 1
    while it <= n
        rg = it:min(it+cols-1,n)
        for i = 1:sz[1]
            for k in rg
                for j = 1:sz[2]
                    print(T[i,j,1,k], " ")
                end
                print("| ")
            end
            println()
        end
        println(("--"^sz[1])*"--")^(cols))
        it += cols
    end
end

# count # of valid samples
function count_valid(T)
    v = 0
    for i = 1:size(T)[end]
        if valid(T[:,:,1,i])
            v += 1
        end
    end
    return v
end

# count # of unique samples
function count_unique(T)
    n = size(T)[end]
    A = Vector{Matrix{Int64}}(n)
    for i = 1:n
        A[i] = T[:,:,1,i]
    end
    Aun = unique(A)
    return length(Aun)
end

# count # of new samples (does not appear in training set)
# TrArr is the training set
function count_new(T, TrArr)
    n = size(T)[end]
    A = Vector{Matrix{Int64}}(n)
    for i = 1:n
        A[i] = T[:,:,1,i]
    end
    Aun = unique(A)
    v = 0
    for i = 1:length(Aun)
        if Aun[i] in TrArr
        else
```

```
            v += 1
        end
    end
    return v
end

# return statistics of samples
function stats_num(T)
    v = 0
    vm = Vector{Int64}(size(T)[end])
    np = Int(maximum(T))
    for i = 1:size(T)[end]
        D = T[:,:,1,i]
        vi = 0
        for j = 1:np
            m = max(maximum(sum(D .== j, 1)), maximum(sum(D .== j, 2)),
                            sum(diag(D .== j)), sum(diag(flipdim(D, 2) .== j)))
            if m > v
                v = m
            end
            if m > vi
                vi = m
            end
        end
        vm[i] = vi
    end
    return Dict(:max => float(v), :mean => mean(vm))
end
```

The following code construct Discrete Wasserstein GAN networks (generator and critic) and run the experiment on them.

```
### DISCRETE WASSERSTEIN GAN ###

using Knet
using PyCall
@pyimport tensorboard_logger as tl

### Network ###

# Generator
function netG(w, z, np, softmax_scaling)
    x = relu(w[1]*z .+ w[2])
    x = w[3]*x .+ w[4]
    M = devec_ttt(x, np)
    eM = exp(softmax_scaling * M)          # softmax (with scaling)
    Ms = eM ./ sum(eM, 3)
    x = vec_ttt(Ms)
    return x
end

# critic
function netC(w, real, fake, np, lambda)
    nc = np + 1
    k = Int(sqrt(size(real, 1) / nc))
    n = size(real)[end]
    # filter
    fake_rounded = round_sample_knet(fake, np)
    M = devec_ttt(real .* fake_rounded, np)
    Ms = 1f0 - sum(M, 3)
    Ms = max(Ms, 0f0)       # if there's problem in rounded fake
    filter = reshape(Ms, (k^2, n))
    # random flip real / fake
    idr = rand(0:1, n)'  # 1:keep, 0:flip
    idx = KnetArray(map(a -> convert(Float32, a), idr))
```

```
    top = real .* idx + fake .* (1-idx)
    bottom = real .* (1-idx) + fake .* idx
    # top
    top = relu(w[1]*top .+ w[2])
    # bottom
    bottom = relu(w[3]*bottom .+ w[4])
    # combine
    x = cat(1, top, bottom)
    x = w[5]*x .+ w[6]
    x = tanh(x)
    x = filter .* x
    x = sum(x, 1)
    # apply sign, because of flipping
    sgn = idx * 2f0 - 1                # sign, if flipped : -1
    x = sgn .* x
    penalty = sqrt(sum(x .* x))            # norm penalty (for diversity)
    return sum(x) / n  +  lambda * penalty
end

function netGC(wG, wC, z, real, np, softmax_scaling, lambda)
    fake = netG(wG, z, np, softmax_scaling)
    return netC(wC, real, fake, np, lambda)
end

function weightsG(k=3, nz=5, np=2; atype=KnetArray{Float32})
    nc = np + 1
    w = Array{Any}(4)
    nv_out = k^2*nc
    w[1] = xavier(4nv_out,nz)
    w[2] = zeros(4nv_out,1)
    w[3] = xavier(nv_out,4nv_out)
    w[4] = zeros(nv_out,1)
    return map(a->convert(atype,a), w)
end

function weightsC(k=3, np=2; atype=KnetArray{Float32})
    nc = np + 1
    w = Array{Any}(6)
    nv_in = k^2*nc
    nv_out = k^2
    # top
    w[1] = xavier(4nv_in,nv_in)
    w[2] = zeros(4nv_in,1)
    # bottom
    w[3] = xavier(4nv_in,nv_in)
    w[4] = zeros(4nv_in,1)
    # combined
    w[5] = xavier(nv_out,8nv_in)
    w[6] = zeros(nv_out,1)
    return map(a->convert(atype,a), w)
end

##### run experiment ######
# configs
k = 4                       # grid size : k-by-k
np = 20                     # # of players (# of class = np + 1)
n = 10 * k * np             # # of training samples
nz = 10                     # # of random noise for generator
n_gen = 2000                # max # of generation
softmax_scaling = 7         # softmax scaling ==> exp(scale * x) / sum(exp(scale * x))
c_iter = 5                  # # of iteration : critic
g_iter = 1                  # # of iteration : generator
lrG = 5e-2                  # learning rate : generator
lrC = 5e-2                  # learning rate : critic
lambda = 0.                 # norm penalty (for diversity)
```

```
decayC = 0.95
decayG = 0.95
decayitC = 50
decayitG = 50
srand(0)                    # random seed
gpu(0)                      # set gpu id

tl.configure("runs/dwgan-k3-np2", flush_secs=5)

# define gradient function
netC_grad = grad(netC)
netGC_grad = grad(netGC)

# real data
rv = KnetArray(vec_ttt(generate_samples(k, n, np)))
fixed_z = KnetArray(randn(Float32, nz, 100))
# convert to TrArr
TrT = to_ttt(devec_ttt(Array(rv), np))
TrArr = Vector{Matrix{Int64}}(n)
for i = 1:n
    TrArr[i] = TrT[:,:,1,i]
end

# init weights
wC = weightsC(k, np)
wG = weightsG(k, nz, np)

# fake
fv = netG(wG, fixed_z, np, softmax_scaling)
T = to_ttt(devec_ttt(Array(fv), np))
println("Iter: 0")
println("Fake pixel prob: max: $(maximum(fv)), min: $(minimum(fv))")
println("# valid : $(count_valid(T))/$(size(T)[end])  |  ",
        "# unique : $(count_unique(T))/$(size(T)[end])",
        "  |  # new : $(count_new(T, TrArr))/$(size(T)[end])  |  ",
        "stats num : $(stats_num(T))")

it = 0
itC = 0
itG = 0
for iter = 1:n_gen

    # train critic
    outputC = 0.
    for j = 1:c_iter
        # fake samples
        z = KnetArray(randn(Float32, nz, n))
        fv = netG(wG, z, np, softmax_scaling)
        # real + fake
        outputC = netC(wC, rv, fv, np, lambda)
        tl.log_value("output_C", outputC, itC)
        tl.log_value("output", outputC, it)
        gC = netC_grad(wC, rv, fv, np, lambda)
        tl.log_value("grad_C_mean", mean(map(x -> mean(Array(x)), gC)), itC)
        tl.log_value("grad_C_std", mean(map(x -> std(Array(x)), gC)), itC)
        for i in 1:length(wC)
            wC[i] += lrC * gC[i]
        end
        it += 1
        itC += 1
    end

    if itC % decayitC == 0
        lrC = decayC * lrC
    end
```

```
    # train generator
    for j = 1:g_iter
        z = KnetArray(randn(Float32, nz, n))
        outputG = netGC(wG, wC, z, rv, np, softmax_scaling, lambda)
        tl.log_value("output_G", outputG, itG)
        tl.log_value("output", outputG, it)
        gG = netGC_grad(wG, wC, z, rv, np, softmax_scaling, lambda)
        tl.log_value("grad_G_mean", mean(map(x -> mean(Array(x)), gG)), itG)
        tl.log_value("grad_G_std", mean(map(x -> std(Array(x)), gG)), itG)
        for i in 1:length(wG)
            wG[i] -= lrG * gG[i]
        end

        if j == g_iter
            fv = netG(wG, fixed_z, np, softmax_scaling)
            T = to_ttt(devec_ttt(Array(fv), np))

            tl.log_value("stats_valid", count_valid(T), iter)
            tl.log_value("stats_unique", count_unique(T), iter)
            tl.log_value("stats_new", count_new(T, TrArr), iter)
            tl.log_value("stats_mean", stats_num(T)[:mean], iter)
            tl.log_value("stats_max", stats_num(T)[:max], iter)

            if iter % 1 == 0
                println("Iter: $iter")
                println("Fake pixel prob: max: $(maximum(fv)), min: $(minimum(fv))")
                println("# valid : $(count_valid(T))/$(size(T)[end])  |  ",
                "# unique : $(count_unique(T))/$(size(T)[end])",
                "  |  # new : $(count_new(T, TrArr))/$(size(T)[end])  |  ",
                "stats num : $(stats_num(T))")
                println("output C : $outputC, output G : $outputG")
            end
        end

        it += 1
        itG += 1
    end

    if itG % decayitG == 0
        lrG = decayG * lrG
    end
end
```

B.2 Real Data Experiments

For the experiments with MNIST dataset, we use **Python** programming language with **PyTorch** deep learning framework.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
import torchvision.utils as vutils
from torch.autograd import Variable
import torch.nn.init as init
from os.path import isfile, isdir, join
import os
from tensorboard_logger import configure, log_value

# arguments
class Args:
    pass
```

```
args = Args()
args.lrD = 5e-4
args.lrG = 5e-4
args.batch_size = 100
args.cuda = True
args.epochs = 1000
args.device = 5
args.seed = 1
args.nz = 10
args.d_iter = 5
args.g_iter = 1
args.lamba = 1e-2     # constant for L2 penalty (diversity)
args.name = "mnist-experiment"

configure("runs/run-" + args.name, flush_secs=5)
torch.manual_seed(args.seed)
if args.cuda:
    torch.cuda.set_device(args.device)
    torch.cuda.manual_seed(args.seed)

data_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=True, download=True,
                   transform=transforms.Compose([
                       transforms.ToTensor(),
                   ])), batch_size=args.batch_size, shuffle=True)

class NetD(torch.nn.Module):
    def __init__(self, use_cuda=True):
        super(NetD, self).__init__()
        self.use_cuda = use_cuda
        # top
        self.t1 = torch.nn.Linear(28 * 28, 1024)
        # bottom
        self.b1 = torch.nn.Linear(28 * 28, 1024)
        # combined
        self.fc = torch.nn.Linear(2 * 1024, 28 * 28)

    def forward(self, xr, xf):
        # get filt
        filt = 1 - (xr * (xf >= 0.5).float()) - ((1-xr) * (xf < 0.5).float())
        # random swap
        idr = torch.multinomial(torch.Tensor([0.5,0.5]), xr.size(0), replacement=True)
        idrx = idr.float().unsqueeze(1).expand_as(xr)
        if self.use_cuda: idrx = idrx.cuda()
        idrx = Variable(idrx)
        xt = xr * idrx + xf * (1 - idrx)
        xb = xr * (1 - idrx) + xf * idrx
        # top : real
        xt = F.relu(self.t1(xt))
        # bottom : fake
        xb = F.relu(self.b1(xb))
        # combined
        x = torch.cat((xt, xb), 1)
        x = F.tanh(self.fc(x))
        # apply filter, aggregate
        x = filt * x
        x = x.mean(dim = 1).squeeze()
        # use sign, because of swapping
        sgn = idr * 2 - 1
        if self.use_cuda: sgn = sgn.cuda()
        sgn = Variable(sgn.float())
        x = sgn * x
        return x
```

```
netG = torch.nn.Sequential(
    torch.nn.Linear(args.nz, 1024),
    torch.nn.ReLU(),
    torch.nn.Linear(1024, 28 * 28),
    torch.nn.Sigmoid()
)

# networks
netD = NetD()

print(netG)
print(netD)

optimizerG = optim.RMSprop(netG.parameters(), lr=args.lrG)
optimizerD = optim.RMSprop(netD.parameters(), lr=args.lrD)

one = torch.FloatTensor([1])
mone = one * -1

if args.cuda:
    netD.cuda()
    netG.cuda()
    one, mone = one.cuda(), mone.cuda()

gen_iterations = 0
for epoch in range(args.epochs):
    data_iter = iter(data_loader)
    i = 0
    while i < len(data_loader):
        ############################
        # (1) Update D network
        ###########################
        for p in netD.parameters(): # reset requires_grad
            p.requires_grad = True # they are set to False below in netG update

        d_iter = args.d_iter
        j = 0
        while j < d_iter and i < len(data_loader):
            j += 1

            # load real data
            i += 1
            X, _ = data_iter.next()
            X = X.view(X.size(0), -1)
            X = (X >= 0.5).float()
            if args.cuda: X = X.cuda()
            real = Variable(X)

            # generate fake data
            noise = torch.randn(args.batch_size, args.nz)
            if args.cuda: noise = noise.cuda()
            noisev = Variable(noise, volatile = True) # totally freeze netG
            fake = Variable(netG(noisev).data)

            # compute gradient, take step
            netD.zero_grad()
            out = netD(real, fake)
            outputD = torch.mean(out) + args.lamba * out.norm()
            stdD = torch.std(out)
            outputD.backward(mone)
            optimizerD.step()

        ############################
        # (2) Update G network
        ###########################
```

```
        g_iter = args.g_iter
        j = 0
        while j < g_iter and i < len(data_loader):
            j += 1

            for p in netD.parameters():
                p.requires_grad = False # to avoid computation
            netG.zero_grad()

            # load real data
            i += 1
            X, _ = data_iter.next()
            X = X.view(X.size(0), -1)
            X = (X >= 0.5).float()
            if args.cuda: X = X.cuda()
            real = Variable(X)

            # update generator
            noise = torch.randn(args.batch_size, args.nz)
            if args.cuda: noise = noise.cuda()
            noisev = Variable(noise)
            fake = netG(noisev)
            out = netD(real, fake)
            outputG = torch.mean(out) + args.lamba * out.norm()
            stdG = torch.std(out)
            outputG.backward(one)
            optimizerG.step()

            gen_iterations += 1

            print('[%d/%d][%d/%d][%d] Loss_D: %f Loss_G: %f '
                % (epoch, args.epochs, i, len(data_loader), gen_iterations,
                outputD.data[0], outputG.data[0]))

            log_value('output_D', outputD.data[0], gen_iterations)
            log_value('output_G', outputG.data[0], gen_iterations)
            log_value('std_D', stdD.data[0], gen_iterations)
            log_value('std_G', stdG.data[0], gen_iterations)

            if gen_iterations % 100 == 0:
                if not isdir('images/{0}'.format(args.name)):
                    os.mkdir('images/{0}'.format(args.name))
                real = real.data[0:100,:]
                real = real.view(real.size(0), 1, 28, 28)
                vutils.save_image(real, 'images/{0}/real_samples.png'.format(
                        args.name, gen_iterations))

                noise = torch.randn(min(100, args.batch_size), args.nz)
                if args.cuda: noise = noise.cuda()
                fake = netG(Variable(noise, volatile=True))
                # fake = (fake.data >= 0.5).float()
                R = torch.rand(fake.size())
                fake = (fake.data.cpu() >= R).float()
                fake = fake.view(fake.size(0), 1, 28, 28)
                vutils.save_image(fake, 'images/{0}/fake_samples_{1}.png'.format(
                        args.name, gen_iterations))

    # do checkpointing
    if not isdir('checkpoint/{0}'.format(args.name)):
        os.mkdir('checkpoint/{0}'.format(args.name))
    torch.save(netG.state_dict(), 'checkpoint/{0}/netG_epoch_{1}.pth'.format(
            args.name, epoch))
    torch.save(netD.state_dict(), 'checkpoint/{0}/netD_epoch_{1}.pth'.format(
            args.name, epoch))
```