GAN Training for Discrete Generators

Viktor Toth
Department of Computer Science
University of Cambridge
vt289@cam.ac.uk

Abstract

Generative Adversarial Networks (GANs) have been highly successful in numerous continuous generation tasks. However, training them often requires the differentiation of the generator outcome, making it a less straightforward solution for problems in discrete spaces, where the output is usually obtained by sampling. In this report, I present a novel way of performing GAN training for discrete tasks using the generation likelihood; a quantity already implicitly calculated by most recurrent neural models. To illustrate the new approach, I provide an implementation of a character-level name generator.

1 Introduction

In recent years, Generative Adversarial Networks have gained prominence in several machine learning areas, most notably in computer vision. State-of-the-art models are capable of generating hyper-realistic human faces [1], illustrative pictures for short descriptions [2] or 3D object reconstructions from 2D images [3]. This prompted an increased interest in applying GANs to discrete domains such as natural language processing problems.

Generating data in discrete spaces requires sampling, which is a non-differentiable operation. However, the standard training process for GANs relies on the output of the generator being differentiable with respect to its parameters. Therefore, applications in discrete problems – such as text generation – has proven to be more challenging.

Over the years, several techniques have been explored as a solution: using reinforcement learning [4], replacing the Softmax operation with the differentiable Gumbel-Softmax approximation [5] or employing autoencoders for transforming the discrete problem space into a continuous one [6]. This report will present a novel approach that does not require additional model alterations.

First, I will introduce the mathematical background of training GANs and the derivation of the commonly used optimisation problem. Then, we will explore how to transform the loss function by applying importance sampling – similarly to the "reparametrisation trick" used in variatonal autoencoders [7] – to derive an equivalent loss metric more suited for training on discrete problems. Finally, I will illustrate this new approach with generating English first names using recursive neural networks (RNNs) for the generator and discriminator models.

2 Overview of Generative Adversarial Nets

In generative tasks, the goal is to learn a probability distribution of some underlying random variable(s) based on a dataset of empirical samples. For example, when generating human faces, we are interested in learning the joint probability distribution of individual pixel values. Once such a model is trained, we can generate synthetic data by sampling it using a randomly generated seed as input.

GANs were first described by Goodfellow et. al. [8] in 2014. They described two competing models: a generator and a discriminator. Informally, the aim of the discriminator is to differentiate true

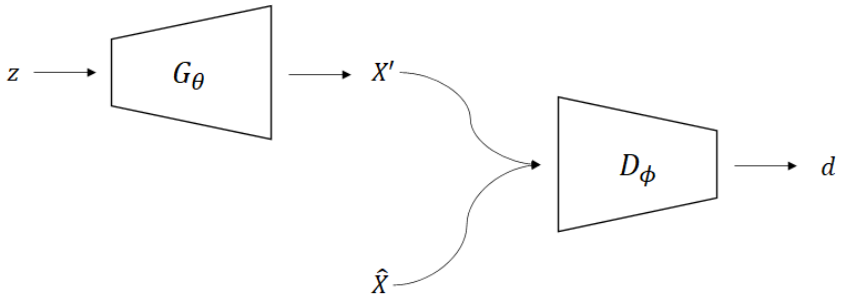


Figure 1: A schematic illustration of GANs with generator G_θ and discriminator D_ϕ (θ and ϕ are the model parameters). The random input to the generator is denoted by z, X' is the random variable representing the output of the generator, and $\hat{X}$ is the random variable drawn from the empirical distribution. The two are passed to the discriminator which outputs the decision d.

samples from generated fake samples, while the generator tries to "fool" the discriminator. An illustration of the GAN paradigm is shown in Figure 1.

The job of the discriminator can be construed as a supervised learning problem: the real ($\hat{x}_i$) and synthesised (x'_i) data is its input while the binary label ($\hat{y}_i$ and y'_i respectively) represents its desired output. Conventionally, the label $\hat{y}_i = 1$ is associated with true samples, while $y'_i = 0$ is for synthetic ones. Furthermore, the predicted label can be modelled as independently drawn from $Y_i \sim \text{Bin}(1, D_\phi(x_i))$.

To measure the effectiveness of the discriminator, we can use the (normalised) log-likelihood of outputting the correct labels:

$$\begin{aligned} L(D_\theta) &= \frac{1}{N} \log \Pr(\hat{y_1}, ..., \hat{y_N}, y'_1, ..., y'_N) \\ &= \frac{1}{N} \sum_{i=1}^{N} \log \Pr(\hat{y}_i) + \frac{1}{N} \sum_{i=1}^{N} \log \Pr(y'_i) \\ &= \frac{1}{N} \sum_{i=1}^{N} \log \Pr(D_\phi(\hat{x}_i)) + \frac{1}{N} \sum_{i=1}^{N} \log \Pr(D_\phi(x'_i)) \end{aligned}$$

Taking the limit of the above approximation, we can derive the usual optimisation problem for the discriminator:

$$\lim_{N\to\infty} L(D_\theta) = \underbrace{\mathbb{E}_{x\sim \Pr_{\hat{X}}} \log D_\phi(x)}_{\text{Real data}} + \underbrace{\mathbb{E}_{x\sim \Pr_{X'}} \log(1 - D_\phi(x))}_{\text{Synthesised data}} \tag{1}$$

It is important to point out that the expectations for the synthesised data is over the generator output distribution. Hence, it is commonplace to write Equation 1 in the following format:

$$\lim_{N\to\infty} L(D_\theta) = \mathbb{E}_{x\sim \Pr_{\hat{X}}} \log D_\phi(x) + \mathbb{E}_{z\sim \Pr_Z} \log(1 - D_\phi(G_\theta(z)))$$

The goal of the discriminator is to maximise the log-likelihood, while the generator aims at minimising it. Therefore, we can describe the combined optimisation problem of GANs as a two-player min-max game:

$$\min_\theta \max_\phi \; \mathbb{E}_{x\sim \Pr_{\hat{X}}} \log D_\phi(x) + \mathbb{E}_{z\sim \Pr_Z} \log(1 - D_\phi(G_\theta(z))) \tag{2}$$

It can be shown that the global optimum of Equation 2 is achieved when the generated distribution (X') is identical to the empirical distribution ($\hat{X}$) [8].

In practice, training GANs often consists of alternately updating ϕ and θ using stochastic gradient descent. When updating the discriminator, we fix the generator model and use the optimisation problem described Equation 1. This step only requires sampling the synthesised data distribution, and does not depend on the generator being differentiable. However, when we update the generator,

we have the following optimisation problem (we can drop the real data portion of Equation 1, since it does not depend on the generator parameter θ):

$$\min_{\theta} \ \mathbb{E}_{z\sim \mathrm{Pr}_Z} \log(1 - D_\phi(G_\theta(z))) \tag{3}$$

Performing stochastic gradient on this function requires the end-to-end differentiation of $D_\phi(G_\theta(z))$, which is not possible if the generator is non-differentiable.

3 Rewriting the Optimisation Problem

The main problem with applying GANs to discrete spaces is that the generator model usually contains sampling and is therefore non-differentiable. As presented in Section 2, training the generator is traditionally done by performing stochastic descent on the optimisation problem described in Equation 3. However, this requires the end-to-end differentiation of the GAN pipeline. In this section, we will apply importance sampling to the optimisation problem and will derive an alternative optimisation problem that does not require the differentiation of the generator output.

3.1 Importance Sampling

The standard way of performing stochastic descent on Equation 3 is to approximate the expectation using Monte Carlo sampling. That is, we sample random values $z_1, ..., z_n$ from the generator input distribution Z and calculate the average of the expectation function:

$$\mathbb{E}_{x\sim \mathrm{Pr}_{X'}} \log(1 - D_\phi(x)) = \mathbb{E}_{z\sim \mathrm{Pr}_Z} \log(1 - D_\phi(G_\theta(z)))$$

$$= \int_Z \log(1 - D_\phi(G_\theta(z)))\mathrm{Pr}(z)\ \mathrm{d}z \approx \frac{1}{N}\sum_{i=1}^{N} \log(1 - D_\phi(G_\theta(z_i)))$$

Importance sampling allows us to estimate the expectation using samples generated from different distributions. In this case, we can replace the distribution X' with a more suitable one. The goal is to avoid using the output of the generator in the formula and hence circumvent the need for differentiating it. Therefore, the new sampling distribution should not depend on the generator. Moreover, importance sampling provides better approximations if the sampled distribution is close to the original one. Hence, using the empirical distribution ($\hat{X}$) as the sampling distribution appears to be a natural choice.

Drawing samples $\hat{x_1}, ..., \hat{x_N}$ from the empirical distribution $\hat{X}$, we derive the following approximation for the optimisation problem of the generator:

$$\mathbb{E}_{x\sim \mathrm{Pr}_{X'}} \log(1 - D_\phi(x)) \approx \frac{1}{N}\sum_{i=1}^{N} \log(1 - D_\phi(\hat{x}_i))\frac{Pr_{X'}(\hat{x}_i)}{Pr_{\hat{X}}(\hat{x}_i)} \tag{4}$$

In many discrete generation tasks, the dataset contains unique values and we sample them using a uniform distribution. In this case, $Pr_{\hat{X}}(\hat{x}_i) = 1/|S|$, where $|S|$ is the dataset size. Applying it to Equation 4 results in the following optimisation problem for training the generator:

$$\min_{\theta} \ \frac{1}{N}\sum_{i=1}^{N} \Big[\underbrace{\log(1 - D_\phi(\hat{x}_i))}_{\text{Discriminator prediction}} \ \cdot \ \underbrace{Pr_{X'}(\hat{x}_i)}_{\text{Generation likelihood}} \Big] \tag{5}$$

We have now eliminated the need to differentiate the output of the generator model, the main obstacle for applying GANs to discrete problem spaces. However, we have introduced a new term: the generation likelihood. In the following section, we will discuss how to adapt common recurrent neural to compute its value in a differentiable way.

3.2 Calculating the Generator Likelihood

Performing stochastic gradient descent on the newly acquired optimisation problem requires computing and differentiating the generation likelihood – that is the probability of our model generating

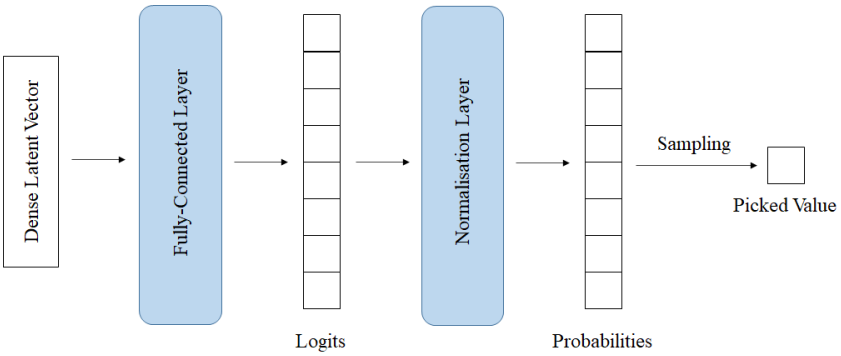


Figure 2: Illustration of a common sampling process for discrete neural models.

the given output. Luckily, most modern model architectures are already based on calculating output probabilities in a continuous space. Below, we will explore how we can make such probabilities explicit and perform back-propagation on them.

When generating discrete data, it is commonplace to first calculate the output of the model as a dense latent vector. It is then transformed into a sparse, one-hot style vector representation of the "scores" the model assigns to each output (often called "logits"). Since there is no restriction on the values logits can take, they are not immediately suited for sampling. Instead, we perform a normalisation transform (such as softmax), which ensures that the results form a valid discrete distribution (i.e. they are non-negative and sum to 1). Finally, we can use a sampling operation to pick the output value[1]. When generating discrete sequences, the output value is then fed as input into the next layer. Figure 2 illustrates the described sequence of transformations and intermediate states.

This setup lends itself naturally to calculating the likelihood of a specific output: we omit the sampling process and simply use the calculated probability for the desired sequence. This is equivalent to computing the dot-product of the probabilities vector with the one-hot encoding of the respective token in the sequence. Furthermore, we can use the desired sequence as a constant input for the next layer. Since we only performed continuous operations, the probabilities vector (or its dot-product) is differentiable with respect to the model parameters, making gradient descent on the generator's optimisation problem (Equation 5) possible.

In the next section, we will explore an example of the presented new approach to training Generative Adversarial Networks in discrete problem spaces. We will use a recursive neural network for both the generator and discriminator, showing in detail how to incorporate the generation likelihoods into the training pipeline.

4 Character-Level Name Generation

To illustrate the above described training method, I have implemented a character-level name generator for English first names in the popular PyTorch deep-learning framework [9]. This task was chosen for its relative simplicity and the availability of suitable training datasets.

4.1 Discriminator Model

The goal of the discriminator model is to assign a scalar label y to the inputted word. For this, I have used a recursive neural network built upon LSTM layers [10]. Each character in the inputted word is transformed into a one-hot encoded vector and then used as an input to the respective LSTM layer. To allow for batching, the words need to be padded to equal length. This can be achieved by introducing an extra character to the alphabet ("$").

[1] Sometimes models pick the largest probability value and achieve randomness by seeding the initial input of the network and not the sampling process. However, calculating the generation likelihood for these models requires Mont Carlo estimation and is generally non-differentiable.

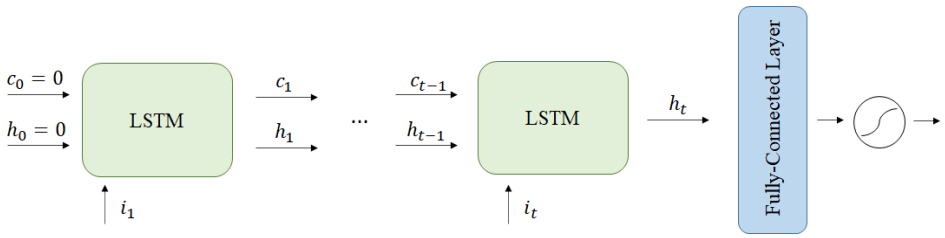


Figure 3: Overview of the discriminator architecture. The initial channel and hidden states are set to 0. The individual characters of the word in one-hot encoding $i_1, ..., i_t$ is used as the input of the LSTM layers, while the final hidden state h_t is used in a fully-connected layer to predict the probability of the inputted word being real.

Finally, the hidden state of the of the last LSTM is fed through a fully-connected layer to produce the assigned scalar value. In order to interpret it as the parameter to the binary distribution model described in Section 2, we normalise it using the sigmoid function. Figure 3 shows an overview of the discriminator model.

4.2 Generator Model

The generator model is built similarly to the discriminator: we use a string of LSTM layers to probabilistically generate the individual characters. The output of each layer – the hidden state – is used to produce the probabilities associated with each character (see Figure 2).

The model has two modes of operation: generating sequences and calculating the generation likelihood. When generating, the sampled character is inputted to the next LSTM layer in a one-hot encoded format. On the other hand, when calculating the likelihood, we are interested "generating" the selected sequence, and hence use its characters as input to the model. The input of the first layer is a special "start-of-word" character (".") in order for the model to be capable of learning the distribution of starting characters. An overview of the generator model is shown in Figure 4.

While the generator model doesn't have an explicit input, we write $G_\theta(z)$ as the deterministic function generating a new word with z as a seed for the sampling process.

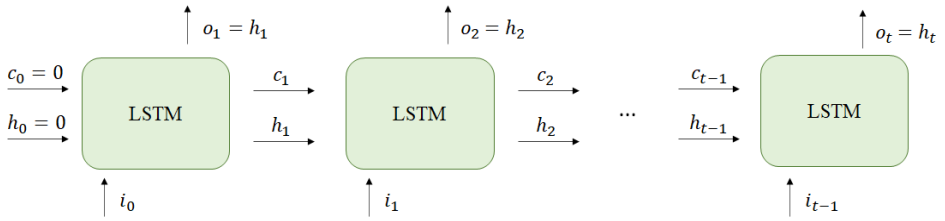


Figure 4: Overview of the generator model. The initial channel and hidden states are set to 0. The characters are generated using the output of the LSTM layers ($o_1 = h_1, ..., o_t = h_t$). Assigned probabilities are calculated using the pipeline described in Section 3.2. The first input (i_0) is the one-hot encoding of the special character ".", while following inputs are either the sampled (generated) characters or the ones in the word used for likelihood calculation.

4.3 Training

I have trained the models using a dataset containing 7,944 unique names (5,001 female and 2,943 male) collected by Mark Kantrowitz[2]. I have only used names whose length is at most 9 letters and do not contain any special characters (e.g. an apostrophe or space).

[2] `http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/nlp/corpora/names/`

The discriminator loss function was a Monte Carlo approximation of the negated GAN optimisation problem (Equation 2):

$$\ell(D_\phi, \hat{\mathbf{x}}, \mathbf{z}) = -\frac{1}{N}\left[\sum_{i=1}^{N} \log D_\phi(\hat{x}_i) + \sum_{i=1}^{N} \log(1 - D_\phi(G_\theta(z_i)))\right]$$

For the generator, we use the optimisation problem of Equation 4. However, applying it directly outputs negative results with extremely small absolute values, making gradient descent-based optimisers less effective. Hence, I have used the following loss function (which has identical optimums):

$$\ell(G_\theta, \hat{\mathbf{x}}) = -\log(-L(G_\theta, \hat{\mathbf{x}}))$$

where the function L is defined as:

$$L(G_\theta, \hat{\mathbf{x}}) = \frac{1}{N}\sum_{i=1}^{N}\Big[\log(1 - D_\phi(\hat{x}_i))Pr_{X'}(\hat{x}_i)\Big]$$

5 Results

In this section, we will investigate the effectiveness of the presented new discrete GAN training method on the character-level name generation problem. To perform the learning steps, I have used the Adam optimiser [11]. All presented results were obtained from models trained with 200 epochs and a batch size of 128 examples.

5.1 Convergence of Loss Functions

To examine the convergence of the defined loss metrics, I have performed 30 independent training sessions. For each, I have calculated the epoch loss as the average of the batch losses (see Section 4.3 for details). To better understand the variance in the loss function between separate runs, I have plotted their averages as well as their minimum and maximum values among the different training sessions (Figure 5). The loss values for the discriminator tended to be extremely small, hence I have used a log-scale plot for better readability.

We can observe that the loss functions show only a limited amount of variance between the different training sessions. This implies, that – at least for this relatively simple problem – the new training objectives provide a robust method for learning. We can also see, that after 200 epochs, the learning algorithm is still considerable improving on the loss metrics, indicating that the models have not yet been saturated.

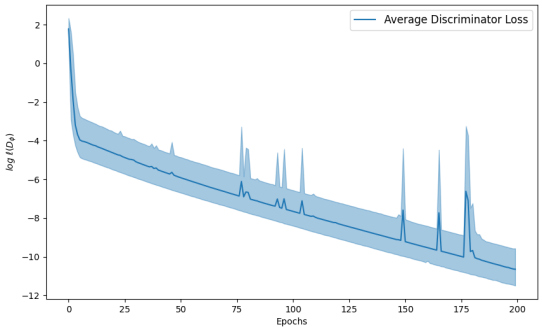
It is also important to note, that the discriminator seems to be extremely good at differentiating between generated and real names. This suggests that there is still considerable room for improvement for the generator model.

5.2 Generated Names

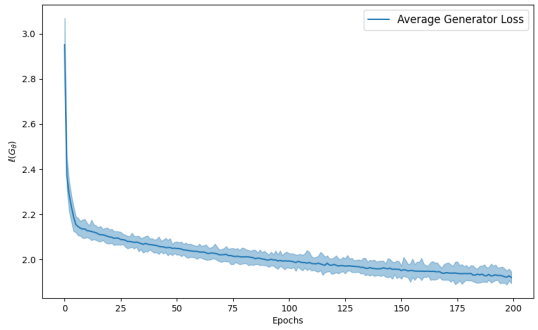
Below is a random sample of the names generated by a model trained for 200 epochs:

- vin
- eran
- ranna
- ola
- ki

One immediate observation is that generated names tend to be shorter than their real counterparts. This is further reinforced by looking at the length distribution of 200 generated and 200 randomly selected real names (Figure 6). This phenomenon can be explained by the relative ease of learning shorter sequences compared to longer ones. I expect the generated name length distribution to become more similar to the empirical one with more training epochs.



(a) Discriminator loss in log-scale



(b) Generator loss

Figure 5: The discriminator and generator epoch losses for 30 independent training sessions. The blue line represents the average losses over the training sessions, while the blue region represents the difference between the minimum and maximum losses. The discriminator losses are on a log-scale. For the definition of the loss functions see Section 4.3.

As an interesting experiment, I have forced the generator to produce sequences with length 10 by manually overwriting the logits associated with the end-of-sequence character ('$'). Below are a few samples:

- dagninanns
- tonannanin
- marinnanan
- eleanniann
- reynannnna

Apparently, the generator model associates longer names with a high frequency of 'a' and 'n' characters, which is probably true for the original distribution as well. However, the quality of these generations would likely benefit from increasing the number of epochs during training.

6 Conclusion and Future Work

Generative Adversarial Networks constitute as the state-of-the art method for generation tasks in continues spaces. Building on their success, applying GANs to discrete problems – such as text generation – has garnered significant interest. However, discrete generators are usually non-differentiable, providing a considerable hurdle for direct application.

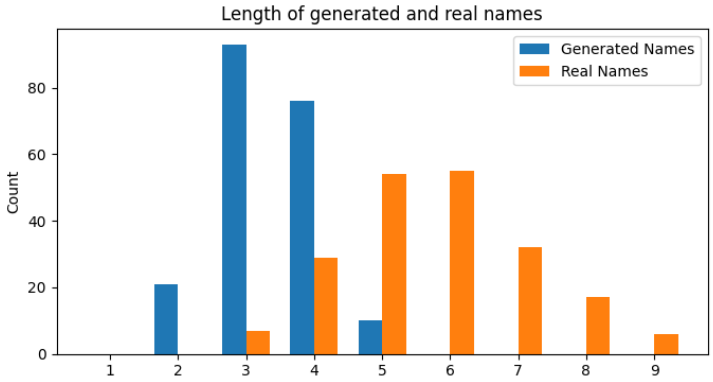


Figure 6: The distribution of the length of 200 generated and 200 randomly selected real names.

In this project, I have presented a novel method for training Generative Adversarial Networks for discrete problem spaces. We have seen how rewriting the optimisation problem using importance sampling can result in a differentiable loss metric containing the generation likelihood. This enables the use of GANs for text generation since most recursive neural architectures are already implicitly calculating the generation likelihoods.

Finally, I have illustrated this new approach with generating English first names character-by-character. The presented empirical data suggests that the derived loss functions provide a robust learning method; their convergence shows little variance for the initial parameters and random decisions. It would be interesting to continue experimentation by applying the method to different tasks of increasing complexity.

References

[1] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation. *CoRR*, abs/1710.10196, 2017.

[2] H. Zhang, T. Xu, H. Li, S. Zhang, X. Wang, X. Huang, and D. Metaxas. Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 5908–5916, 2017.

[3] Jiajun Wu, Chengkai Zhang, Tianfan Xue, William T. Freeman, and Joshua B. Tenenbaum. Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 82–90, Red Hook, NY, USA, 2016. Curran Associates Inc.

[4] Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. Seqgan: Sequence generative adversarial nets with policy gradient. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, page 2852–2858. AAAI Press, 2017.

[5] Matt J. Kusner and José Miguel Hernández-Lobato. GANS for sequences of discrete elements with the gumbel-softmax distribution. *CoRR*, abs/1611.04051, 2016.

[6] Md. Akmal Haidar and Mehdi Rezagholizadeh. Textkd-gan: Text generation using knowledgedistillation and generative adversarial networks. *CoRR*, abs/1905.01976, 2019.

[7] Diederik P. Kingma and M. Welling. Auto-encoding variational bayes. *CoRR*, abs/1312.6114, 2014.

[8] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27, pages 2672–2680. Curran Associates, Inc., 2014.

[9] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[10] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[11] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

A Accessing the Codebase

The implementation can be accessed on GitHub.

`https://github.com/tarnag57/particle_filter_gan`