Learning to Paint using Reinforcement Learning

Abhinav Kumar
160018

Deepankur Kansal
180226

Soumyadeep Datta
20104313

Sugam Srivastava
170728

1 Project Description and Motivation

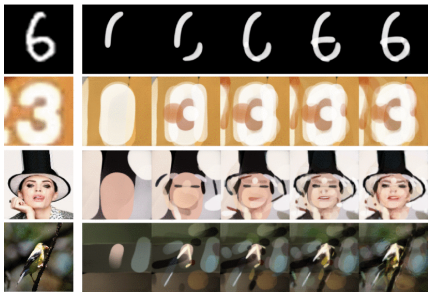


Figure 1: Target Image and painting progression

Generative art using machine learning algorithms is an emerging area of interest. Using deep reinforcement learning and such other state-of-the-art approaches, an AI agent can learn to "think" creatively and produce output of surprisingly good quality.

For our project, we choose the problem of painting using deep reinforcement learning. Given a target image, we want to have an AI agent that can effectively "paint" and produce a good replica of the original image. We base our project on the work done by Huang *et. al.* in [1]. The task of painting comprises the following sub-tasks:

- Describing each brush stroke in a continuous parameter space (location, colour, transparency etc.) and modelling the canvas on which the strokes are to be executed.
- Decomposing target images into an ordered sequence of strokes.
- End-to-end training without human expertise and stroke tracking.

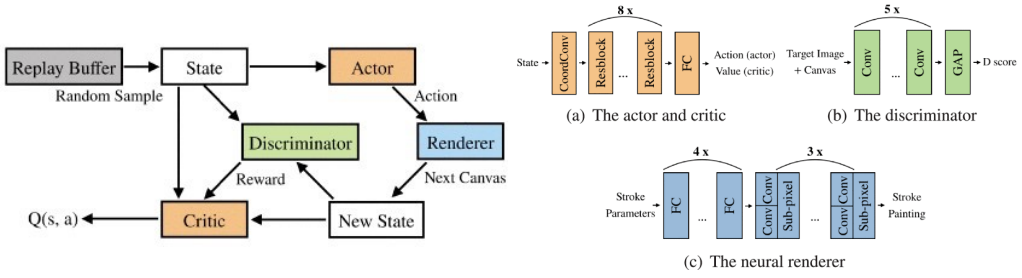


Figure 2: Overall Network Architecture

The overall network architecture to solve the problem statement, as described in [1], is shown in Fig. 2. Each of the above described sub-tasks is executed by a different part of the overall architecture and is introduced briefly as follows:

- Stroke-based renderer: It involves stroke design and implementation of a differentiable neural renderer that accurately models the environment (canvas).
- Model-based DDPG agent: It uses the stroke representation and environment model of the stroke-based renderer. It employs the Deep Deterministic Policy Gradient (DDPG) algorithm to decompose target images into an ordered sequence of strokes (actions).
- GAN discriminator: The discriminator employs a loss function that involves using a Generative Adversarial Network (GAN). Using this loss function, the reward for each stroke is computed and fed as feedback to the DDPG agent.

1.1 Renderer

1.1.1 Brief Overview

The renderer contributes to the painting quality by allowing training model based DRL agent in an end-to-end fashion which boosts the performance of the agent. It helps the agents to observe the environment which helps in better training of the agents. The is the interface that helps us in implementing different types of strokes, e.g. Bezier curve, triangle, and circle, that might be needed for the project. The renderer is able to execute basic features like inputting images, observing frames and implementing strokes on a window from which our agents learn to determine the position and color of each stroke and make long-term plans to decompose texture rich images into strokes. It is an integral part of the project which helps both the user and the agents to better understand the canvas on which the agents act upon.

1.1.2 Model

The neural renderer is a neural network consisting of several fully connected layers and convolution layers. Subpixel upsampling is used to increase the resolution of strokes in the network, which is a fast running operation and can eliminate the checkerboard effect. We can see the full model in Figure 2(c).

1.1.3 Stroke-Based Renderer

The stroke-based renderer has the following design components:

- Stroke design: Each stroke is drawn using a Quadratic Bézier curve (QBC). The QBC is a quadratic interpolation between three control points, $P_0 : (x_0, y_0)$, $P_1 : (x_1, y_1)$ and $P_2 : (x_2, y_2)$, and has the following equation

  $$B(t) = (1-t)^2 P_0 + 2(1-t)tP_1 + t^2 P_2, \quad 0 \leq t \leq 1.$$

  Therefore, each stroke or action, a_t, is represented by the following tuple

  $$a_t = (x_0, y_0, x_1, y_1, x_2, y_2, r_0, t_0, r_1, t_1, R, G, B)_t,$$

  where (R, G, B) controls colour, (r_0, t_0) and (r_1, t_1) control thickness of two endpoints.
- Neural renderer: It is a neural network which accepts as input the stroke parameters or the action, a_t. It implements a model for the environment (canvas), which given the current state s_t and the action a_t, is able to execute the action and obtain as output the rendered image, or the next state, $s_{t+1} \in \mathcal{S}$. It, therefore, implements a model-based, differentiable transition dynamic $s_{t+1} = \text{trans}(s_t, a_t)$.

1.2 Deep Deterministic Policy Gradients (DDPG) agent

1.2.1 Background on Reinforcement Learning and Actor-Critic methods

The Reinforcement Learning (RL) paradigm involves training an agent (player) to achieve a certain objective in a given setting (environment) with a sequence of actions, defined by a Markov Decision

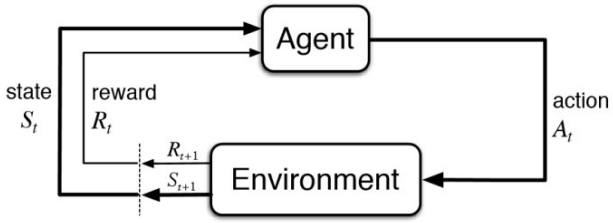


Figure 3: RL paradigm

Process (MDP). At each timestep, using the inputs of the current state of the environment, the agent chooses an action to perform. The environment accordingly gives a feedback on the action, known as the reward, whose value informs the agent of the "effectiveness" of the action.

The Deep Deterministic Policy Gradients (DDPG) algorithm [2] utilizes actor-critic networks and deterministic policy gradients to effectively execute deep reinforcement learning in a continuous action space. The action is modelled using a policy function, $\pi : \mathcal{S} \rightarrow \mathcal{A}$, which maps the state space, $\mathcal{S}$, to the action space, $\mathcal{A}$. This function is learnt by the actor neural network. The Q-value function, $Q : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, maps every state-action pair to a real-valued evaluation metric which measures its effectiveness. At any timestep t, we define the state-action tuple as (s_t, a_t). Our goal is to maximize the discounted expected reward, $R_t = \sum_{i=t}^{T} \gamma^{(i-t)} r(s_i, a_i)$, where $r(s, a)$ denotes the reward for the given state-action pair and the discount factor is $\gamma \in [0, 1]$. The Bellman equation is used to approximate the discounted expected reward along with the data that is taken off-policy from a buffer and thus the Q-value update is done as follows:

$$Q\left(s_t, a_t\right) = r\left(s_t, a_t\right) + \gamma Q\left(s_{t+1}, \pi\left(s_{t+1}\right)\right), \tag{1}$$

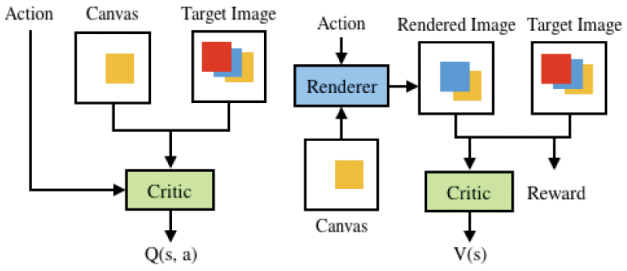
The critic network tries to minimize the following mean-squared loss

$$\mathcal{L} = \left(Q_\phi\left(s_t, a_t\right) - \left(r\left(s_t, a_t\right) + \gamma Q_{\phi'}\left(s_{t+1}, \pi_{\theta'}\left(s_{t+1}\right)\right)\right)\right)^2, \tag{2}$$

where ϕ and ϕ' denote the parameters of the critic network and target critic network, respectively, and θ and θ' denote the parameters of the actor network and target actor network, respectively. The target networks are introduced to make the training more stable and their parameters are updated using Polyak averaging as $\phi' \leftarrow \rho_c \phi' + (1 - \rho_c)\phi, 0 \leq \rho_c \leq 1$ and $\theta' \leftarrow \rho_a \phi' + (1 - \rho_a)\phi, 0 \leq \rho_a \leq 1$. The actor network, on the other hand, learns the optimal policy which maximizes the Q-value at each time step as follows

$$\theta = \arg\max_{\theta} Q_\phi(s_t, \pi_\theta(s_t)) \tag{3}$$

1.2.2 Background on Model-Based DDPG



(a) Original DDPG (b) Model-based DDPG

Figure 4: Original [2] and Model-Based DDPG

For our problem, the model for painting on a canvas, as defined by the neural renderer, becomes our environment. We define the state tuple to contain the current state of the canvas, the current

time-step and the target image $s_t = (C_t, t, I)$. A stroke is our action (a_t) given this state (s_t) and is modeled using a policy π as $a_t = \pi(s_t)$. We also evaluate a return/reward value which tells us how good our action was given the state, $r(s_t, a_t) = L_t - L_{t-1}$ where L_t is the metric for loss between the canvas C_t and the target image I. The loss metric is computed by the GAN-based discriminator, which is described in detail in the following section.

It is difficult for the critic network to model the complex environment of the canvas implicitly and hence compute an appropriate Q-value of the state-action tuple. Thus, following [1], we use a model-based DDPG algorithm. Here, the neural renderer explicitly models the action (stroke) in the environment (canvas) to execute the transition and obtain the next state (new image) as $s_{t+1} = \text{trans}(s_t, a_t)$. This new image is fed to GAN discriminator to obtain the reward. A value function, $V : \mathcal{S} \to \mathbb{R}$, is used in place of Q-value, to map the state directly to the evaluation metric of the critic. Using a discount factor of $\gamma \in (0, 1)$, the new expected reward is written as

$$V(s_{t+1}) = r(s_t, a_t) + \gamma V(s_t),$$

1.3 Generative Adversarial Network (GAN)-based discriminator

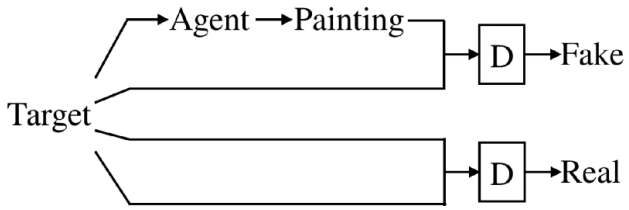


Figure 5: Discriminator Training

GANs are used a popular loss function in image restoration as it measures the distance in distribution between generated and target images. The Wasserstein GAN-GP [3] is employed in the reference paper [1]. It maximizes the Wasserstein-I or Earth-mover distance, defined as

$$\max_D \mathbb{E}_{y\sim\mu}[D(y)] - \mathbb{E}_{x\sim\nu}[D(x)], \tag{4}$$

where D denotes the discriminator function, ν and μ denote the probability distributions of fake and real samples, respectively. The difference of D-scores between s_t and s_{t+1}, obtained from (4), is fed as the reward, $r(s_t, a_t)$, to the DDPG training module.

2 Literature Review

The goal of this project is to teach machines to paint like humans painters using deep reinforcement learning. We base our work on the paper by Huang *et al* [1]. Earlier attempts at solving the same problem include the works [4–6]. SPIRAL [4] is an adversarially trained DRL agent that learn structures in images, but fails to recover the details of human portraits. StrokeNet [5] combines differential and recurrent neural networks to train agents to paint but fails to generalize the color images. Doodle-SDQ [6] trains to emulate human doodling with Q-learning.

The project can broadly divided into three parts:

• Making a Neural Renderer

We create a basic neural renderer following the implementation in [7]. The renderer would be able to execute basic features like inputting images, observing frames and implementing strokes on a window. The earlier works were made using

• Applying deep reinforcement learning (DRL) models to learn how to paint

Following the current paper [1], we approach the painting task with a model-based version of a popular actor-critic based DRL algorithm known as deep deterministic policy gradients (DDPG) [2], which is specifically designed for continuous action spaces, as shall be required for our problem when the actions are continuous Bëzier curves. We further explore two state-of-the-art improvements to the DDPG algorithm, namely, prioritized experience replay (PER) [8], which improves on

experience replay in DDPG by introducing a weighted sampling technique, and adding parameter-space noise [9], which aids in better exploration of the action space and quicker generalization.

• Using Generative Adversarial Networks (GANs) as discriminators.

We are using GANs as discriminators to produce the appropriate loss function by computing distance between the generated image by the DRL agent and the target image. The paper currently uses WGAN-GP [3], which we attempt to improve upon by employing more recent developments like FisherGAN [10], CramerGAN [11], DRAGAN [12], SNGAN [13] and SAGAN [14] to reduce the training time and produce more realistic paintings.

3 Novelty of our work

3.1 Neural Renderer and Strokes

3.1.1 Renderer model enhancements

The differential neural renderer helps the agents to observe the environment and render the strokes made by the agents on the canvas. We increased the overall efficiency of the neural renderer by optimizing the overall architecture of the model. This was achieved by redesigning and optimizing various components of the previous model. We experimented with different forms of optimization methods and activation functions to maximise the performance of the renderer. We also added additional CNN layers to learn more composable features. This helped in minimizing the L1 and L2 losses and achieving greater performance.

3.1.2 Feature for conversion to Pixel Art

This was a new feature added by us to the baseline model. It helps the actors to paint pixel art using the current DDPG model. Two methods can be employed to achieve this task, pre-processing and post-processing. The pre-processing involved implementing custom strokes instead of the Quadratic Bezier strokes that were previously being used by the model.

3.2 Modifications to DDPG

Most, if not all, off-policy based algorithms make use of a *replay buffer* to store experiences that the agent sees when exploring the environment. To break the correlation between continuous data from the environment, random uniform sampling is used to sample a minibatch from the buffer and use it for training the agent. This is known as *experience replay*.

We implement the Prioritized Experience Replay (PER) technique [8]. PER assigns weights to the samples based on their importance. Instead of uniform sampling, now experiences are sampled from the replay buffer in an importance-weighted manner. The idea is to pick those samples first which have a higher error magnitude, $|\delta_i|$, where the error is defined as

$$\delta_i = r(s_t, a_t) + \gamma V_{\theta'}(s_{t+1}) - V_\theta(s_t),$$

where θ and θ' are the parameters of the critic and target critic networks, respectively. The priorities, p_i, are defined *proportionally* as $p_i = |\delta_i| + \epsilon$. The sampling probabilities, $P(i)$, are calculated as

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha},$$

where the exponent α is typically 0.7. We now define the weights assigned for sampling experiences from the buffer as

$$w_i = \left(\frac{1}{N}\frac{1}{P(i)}\right)^\beta,$$

where β is an exponent that is typically initialized as 0.4 or 0.6 and is gradually annealed upwards to 1 as the training progresses. We give more weight to a large deviation from the learnt model as the training progresses since it is increasingly informative.

3.3 Improvements in Adversial discrimination

The loss function generated by the discriminator in WGAN-GP [3] is used to determines how good the mapping policy of the agent being trained. This function is trained and used as a metric to decide upon the selection of the action of the actor. Therefore, using an improved GAN will lead to better results and training in less iterations. Staring with WGAN-GP, we experimented and trained using the following GANs in chronological order.

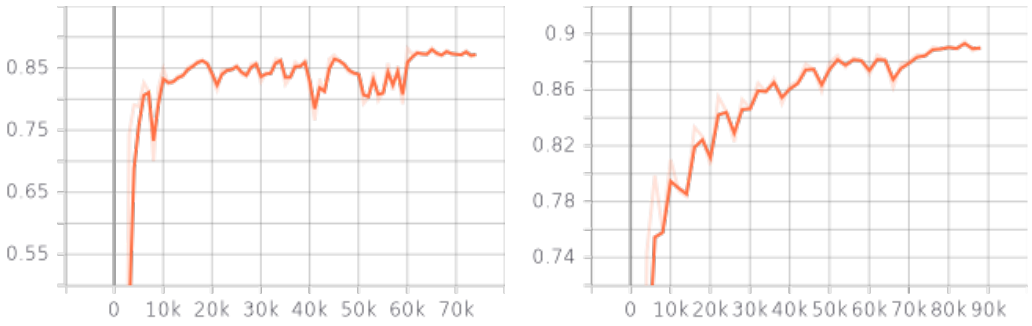
- DRAGAN [12] (slightly worse performance) uses a discriminator loss where local equilibrium can be avoided with a gradient penalty scheme near which sharp gradients of the discriminator function are exhibited around some real data points. (No change in architecture of the discriminators)
- FisherGAN [10] (slightly better performance) incorporates unconstrained capacity and augmented Lagrangian to withstand weight clipping. (No change in architecture of the discriminators)
- CramerGAN [11] (slightly better performance) adds the property of unbiased sample gradients to the WGAN-GP model and uses Cramer distance using KL divergence. (No change in architecture of the discriminators)
- SNGAN [13] (better performance) uses Spectral Normalization in discriminator network to stabilize the training and improve efficiency. (Change in architecture of the discriminator as well as Loss function used)
- SAGAN [14]‘(best performance till experimentation)uses Self-Attention modules in the kernels of discriminators are used and is effective in modelling long range dependencies. (Change in architecture of the discriminator as well as Loss function used)

The performance is tested on training loss for a fixed number of iterations (5k) as a reward metric to the critic to check for the fastest learning.

4 Experimental Setup and Results

We implement the proposed network architecture using PyTorch [15], a popular open source machine learning library and execute the codes on Google Colaboratory using GPU mode.
Our modified code and a notebook for running it can be found on the following links: GitHub repository and Google Colab notebook.

4.1 DDPG with PER



(a) DDPG with PER (b) DDPG without PER

Figure 6: Validation results

The original paper uses the *celebA* data-set with over 2 hundred thousand images of size 128x128 pixels and they run it for 2 million iterations which needs huge amounts of RAM and GPU memory. We used the limited resources on Google Colab and were able to test our improvements only on 25 thousand images from the *celebA* data-set and for only 2 hundred iterations. For a fair comparison we also ran the original code for only 25 thousand images and 2 hundred thousand iterations.

(a) (b) (c) (d)

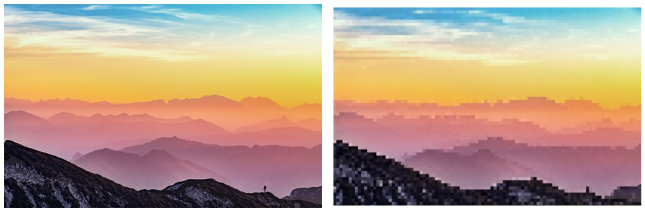
Figure 7: (a) Target Image. (b) Painting by original paper with 200k images and 2 mil iterations without PER. (c) Painting with 25k images and 200k iterations with PER. (d) Painting with 25k images and 200k iterations without PER

Validation Results: In figure 6 we have the validation results and we can see that the PER outperforms the vanilla DDPG by reaching close to the convergence value by only 10k time-step whereas the vanilla DDPG reaches there at around 40k time-step. At around 70k time-steps both are reaching towards similar values.

Test Results: The images in figure 7 part (a) and part (b) are the target image and test result from the original paper's saved models. In figure 7 part (c) and part (d) are our test results on 25k images and 200k iterations with PER and without PER. In part (d) we have the original paper's code's test result. In part (c), we can really see the PER working. Its fairly visible that not all parts of the image are being given equal importance like those on the boundaries, we can also see the parts in the middle seem more close to the original image. Especially the block in second row and second column, it appears closest to the target image. Part (c) also seem to be showing color gamete closer to target image than part (d) which gives uniform importance to all parts of the image and has a green tint all over the test image.

We cannot comment on how PER would compete if run for 200k images and 2 million iterations as in the original paper due to limited resources.

4.2 Stroke-based Neural Renderer



(a) (b)

Figure 8: (a) Painting using the original strokes. (b) Painting using Pixel stroke

The final paintings drawn using the QBC stroke and the custom stroke for Pixel Art are shown in Fig. 8. We used CELEBA dataset in comparing with 25000 images along with 5000 iterations to check for convergence on final training loss as a metric on all variations.

Validation Results:- As seen in Fig. 9, the curves of both train and validation loss become much more smoother after enhancing the previous model.

4.3 GAN-based Discriminator

We used CELEBA dataset in comparing with 25 thousand images along with 5 thousand iterations to check for convergence on final training loss as a metric on all variations.

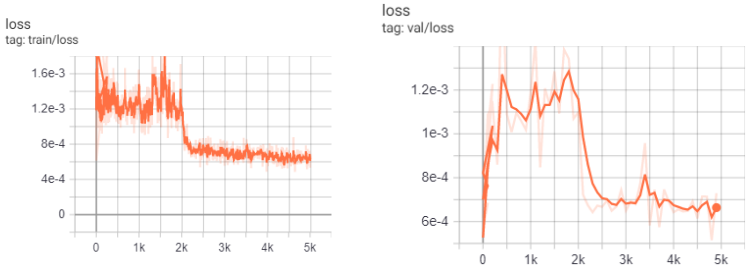


Figure 9: Final Renderer Train and Validation Loss vs Steps

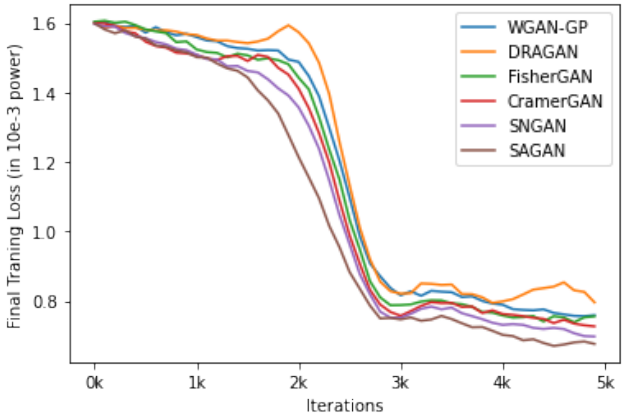


Figure 10: Comparison of different GANs using final training loss for a fixed number of iterations(5k) under the purview of fastest convergence.

Validations Results:- Here, we can see that SAGAN outperforms the WGAN-GP model in less than 3k time stamp as compared to 5k time stamp of WGAN-GP.

We have not considered time to train per iteration as a metric in our experiments but by the training time SAGAN takes considerable more time than SNGAN which in turn takes more time than any of the other variations. Also, WGAN-GP, DRAGAN, FisherGAN and CramerGAN take approximately similar time in the training step.

5 Learning Experience

We firstly list some of the issues faced in the course of the project.

- Resource Constraints: The code of the original paper made use of large RAM and GPU memory, even the resources provided by Google Colab were less. Moreover, Google Colab's session restarts every 12 hours, so we trained only as much as possible within 12 hours. We reduced various parameters and made changes to the code to stop memory leaks. We changed batch-size and cleared cache memory at the end of every training iteration.
- The code had some outdated libraries and code snippets which had to be changed.
- DDPG implementation with Parameter Noise [9] did not learn anything as here a model-based DDPG with CNN layers is used whereas as original paper uses model-free DDPG.

In this online semester, synchronizing our schedules and working together was indeed a challenge and we do realize we could probably have made much more progress in an offline semester. Notwithstanding these problems, it has been a tremendous learning experience for all of us. Each of us had our own unique strengths and skills to contribute and it all came together beautifully.

6 Possible Future Work

We propose the following possible improvements to our work:

- A more stable version of DDPG is the Twin-Delayed DDPG (TD3) [16] where we two critic networks which are time-delayed which means that actor receives a delay in its update. This method solves the overestimation of the Q-values in critic network of a vanilla DDPG.
- Make changes to the discriminator architecture by using improvements to SAGAN which are heavy in computational constraint.
- We can explore the usage of these methods in countless other fields.
- Add some more types of strokes [17] to make different type of artworks

Acknowledgments

We thank our course instructor, Prof Piyush Rai, for trusting us with this project and giving us the freedom to explore a topic that is not being covered in the course. His support, by way of waiving course components, enabled us to focus on the project and complete our work in time.

References

[1] Zhewei Huang, Wen Heng, and Shuchang Zhou. Learning to paint with model-based deep reinforcement learning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[2] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[3] Ishaan Gulrajani, Faruk Ahmed, Mart Arjovsky, Vincent Dumoulin, and Aaron C. Courville. Improved training of wasserstein gans. *CoRR*, abs/1704.00028, 2017.

[4] Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, S. M. Ali Eslami, and Oriol Vinyals. Synthesizing programs for images using reinforced adversarial learning. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1666–1675, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[5] Ningyuan Zheng, Yifan Jiang, and Dingjiang Huang. Strokenet: A neural painting environment. In *International Conference on Learning Representations*, 2019.

[6] Tao Zhou, Chen Fang, Zhaowen Wang, Jimei Yang, Byungmoon Kim, Zhili Chen, Jonathan Brandt, and Demetri Terzopoulos. Learning to sketch with deep Q networks and demonstrated strokes. *CoRR*, abs/1810.05977, 2018.

[7] Reiichiro Nakano. Neural painters: A learned differentiable constraint for generating brushstroke paintings, 2019.

[8] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016.

[9] Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y. Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. Parameter space noise for exploration, 2018.

[10] Youssef Mroueh and Tom Sercu. Fisher GAN. *CoRR*, abs/1705.09675, 2017.

[11] Marc G. Bellemare, Ivo Danihelka, Will Dabney, Shakir Mohamed, Balaji Lakshminarayanan, Stephan Hoyer, and Rémi Munos. The cramer distance as a solution to biased wasserstein gradients. *CoRR*, abs/1705.10743, 2017.

[12] Ishaan Gulrajani, Faruk Ahmed, Martín Arjovsky, Vincent Dumoulin, and Aaron C. Courville. Improved training of wasserstein gans. *CoRR*, abs/1704.00028, 2017.

[13] Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral normalization for generative adversarial networks. *CoRR*, abs/1802.05957, 2018.

[14] Han Zhang, Ian Goodfellow, Dimitris Metaxas, and Augustus Odena. Self-attention generative adversarial networks, 2019.

[15] 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.

[16] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In Jennifer Dy and Andreas Krause, editors, *Proceedings of Machine Learning Research*, volume 80, pages 1587–1596, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[17] Zhengxia Zou, Tianyang Shi, Shuang Qiu, Yi Yuan, and Zhenwei Shi. Stylized neural painting. 2020.