STROKENET: A NEURAL PAINTING ENVIRONMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

We've seen tremendous success of image generating models these years. Generating an image through a neural network is like "dreaming", which is fundamentally different from how humans create artwork using brushes. To imitate human drawing, interactions between the agent and the environment is required to allow trials from the agent. However, the environment is usually non-differentiable, leading to slow convergence and massive computation. In this paper we try to address the discrete nature of software environment with an intermediate, differentiable simulation, which can be interpreted as a neural perception of the surroundings of the upper agent. We present **StrokeNet**, a novel model where the agent is trained upon a well-crafted neural approximation of the painting environment. With this approach, our agent was able to learn to write characters such as MNIST digits very quickly in an unsupervised manner. Our primary contribution is the neural simulation of real-world environment. Furthermore, the agent trained with our approach can be directly transferred to real world with learned skills. To the best of our knowledge, StrokeNet is the first model to apply differentiable simulation to real-world learning problems and standard datasets. [1]

1 INTRODUCTION

To learn drawing or writing, a person first observes (encodes) the target image visually and uses a pen or a brush to scribble (decode), trying to reconstruct the original image. For an experienced painter, he or she foresees the consequences before taking any move, and could thus choose the optimal action.

Stroke-based image generation is fairly different from traditional image generation problems due to the intermediate rendering program. By traditional image generation methods, pixel values are directly calculated which allows effective optimization using back-propagation since such approaches are usually differentiable and capture the distribution of the image. While for stroke-based approaches, rather than learning to generate the image, it is more of learning to manipulate the painting program.

We define "drawing" by giving a formal definition of "stroke". In our context, a "stroke" consists of color, brush radius, and a sequence of tuples containing the coordinate and pressure of each point along the trajectory. We will later describe this in detail in Section 3.

An intuitive yet potentially effective way to tackle the problem is to first learn this mapping from "stroke data" to the resulted image with a neural network, which is analogous to learning painting experience. We can generate as many samples as possible to train the neural version of the environment since the data is cheap.

By doing so, we obtained a differentiable approximator of our painting software, which we called a "**generator**". We then tested the generator by training a vanilla CNN as an **agent** that encodes the image into "stroke" data as an input for the environment. Our proposed architecture, **StrokeNet**, basically comprises the two components, a **generator** and an **agent**, where the agent and the generator could be viewed as an encoder and a decoder respectively. Encoder-decoder architecture is adopted throughout the design of StrokeNet, as inside the generator, the input data is first encoded to spatial features using fully-connected layers and then decoded to the image using deconvolutional layers similar to the ones used in DCGAN (Radford et al., 2015).

[1] Code for the experiment at: `https://github.com/vexilligera/strokenet`.

Finally, the agent was trained to write MNIST (LeCun & Cortes, 2010) and Omniglot (Lake et al., 2015) characters based on the generator. Judging from the result described in Section 4.4, our agent converged rapidly and steadily to produce strokes that capture the trait of both datasets. Especially on MNIST, it took no more than two epochs for the agent to master writing digits.

2 Related Work

Generative models such as VAEs(Kingma & Welling, 2013; Sohn et al., 2015) and GANs(Goodfellow et al., 2014; Mirza & Osindero, 2014; Radford et al., 2015; Arjovsky et al., 2017) have achieved huge success in image generation in recent years. These models generate images directly to pixel-level and thus could be trained through back-propagation effectively.

To mimic human drawing, attempts have been made to model the process of how humans learn to draw (Lake et al., 2015). Recent work generally falls into two categories: RNN-based approaches and reinforcement learning.

For RNN-based approaches such as SketchRNN (Ha & Eck, 2017) and handwriting generation with RNN by Graves (Graves, 2013), they both rely on sequential datasets. Thus for images without the underlying stroke data, those models cannot be applied.

Another popular solution is to adopt reinforcement learning such as "artist agent"(Xie et al., 2012) and SPIRAL (Ganin et al., 2018). These methods train an agent that interact with the painting environment. For reinforcement learning tasks with large, continuous action space like this, the training process can be computationally costly.

To mitigate this situation, we simulate the environment in a differentiable manner much alike the idea in World Models (Ha & Schmidhuber, 2018) and earlier work done by J. Schmidhuber (Schmidhuber, 1990; 2018), where an agent learns from a neural network simulated, environment. In our scenario however, we face real-world learning problem and there are more subtleties in the design and training issues of the generator which we'll discuss in Section 3 and 4.

Differentiable rendering is an extensively researched topic in computer graphics. It's used to solve inverse rendering problems, in other words, given an image, how can we figure out the rendering process to recover the scene. Some differentiable renderers explicity model the relationship between the parameters and observations (Loper & Black, 2014), others use neural network to approximate the result (Nguyen-Phuoc et al., 2018) since neural nets are famouse for their ability to approximate all sorts of functions. While little has been done on simulating 2D rendering process adopted in digital painting software, we used a generator neural network to meet our needs.

3 StrokeNet Architecture and Environment

3.1 Stroke

We define a single stroke as follows,

$$s = \{(c, r), (x_1, y_1, p_1), \cdots, (x_n, y_n, p_n)\},\ n = 16, \tag{1}$$

where c stands for color, r stands for brush radius, and tuple (x_i, y_i, p_i) stands for an anchor point on the stroke, consisting of x, y coordinate and pressure p, and n is the maximum number of points in a single stroke, in this case, 16. These values are normalized such that

$$c, r, p_i \in [0, 1],\ x_i, y_i \in [-1, 1]. \tag{2}$$

Notice that in our neural network setup, c only represents a black and white color value instead of a RGB(A) value. We use single channel color because we can yield color strokes by filtering the output image with the desired color. We encode color information into stroke data because certain kinds of brush stroke effects may need the value that we are likely to model in future work. Most importantly, these filtering and stacking operations are differentiable since they are essentially basic arithmetics, which is a nice property.

Another thing is that all values are normalized to $[0, 1]$ except x_i, y_i because our painting program was developed using a OpenGL style coordinate, which is $[-1, 1]$.

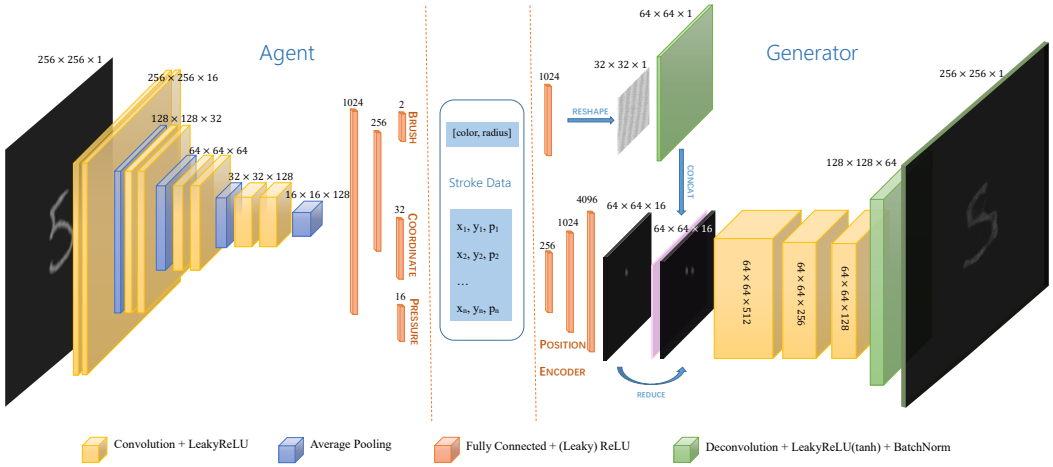


Figure 1: StrokeNet architecture. The generator part of the model outputs 256×256 images. The position encoder encodes input coordinate into 64×64 spatial feature for each point. The agent decodes different information about the stroke using three parallel FC-decoders.

3.2 Generator

The outline of the StrokeNet architecture is shown in Figure 2. The generator part first takes s as input, and the FC-encoder transforms the vector into spatial features. The generator's encoder comprises two parts, both are simple fully-connected layers. One is the **position encoder** which encodes (x_i, y_i, p_i) into 64×64 feature maps, the other, **brush encoder** encodes the color and radius of the brush, which outpus a single 64×64 feature. Then the encoder concatenate them and pass it to the (de)convolution layers. For the position encoder, it's first trained separately. Once the position encoder produces reliable enough signals, we freeze its parameters and train the rest of the network for a while. Finally, we unlock all the parameters and each part of the generator is trained jointly as a whole. We'll discuss training in details in Section 4.

To preserve the sequential and pressure information of each point (x_i, y_i, p_i), the **position encoder** first maps (x_i, y_i) to the corresponding position onto a 64×64 matrix by putting a bright dot on that point. The bright dot is modeled by a 2D Gaussian function with its peak scaled to 1 and $\sigma_1 = 1, \sigma_2 = 1, \rho = 0$ for separate training, which simplifies to:

$$g_i(x, y) = exp[-\frac{1}{2}((x - x_i)^2 + (y - y_i)^2)], \tag{3}$$

for $i = 1, 2, \cdots, n$ where the value is calculated for each point (x, y) on the 64×64 map. Denote this mapping from (x_i, y_i) to $\mathbb{R}^{64\times64}$ as pos:

$$m_i = p_i \cdot pos(x_i, y_i),\ m_i \in \mathbb{R}^{64\times64}. \tag{4}$$

By multiplying the corresponding pressure p_i, we now have n position features.

However, if we directly feed these features into the generating part of the network, the generator fails to converge partly because the single bright dot on each channel could be too sparse for the deconvolution layers to learn the features. Instead, we take every two neighbouring feature maps and add them together (this step is represented by "reduce" in Figure 2.),

$$f_i = m_i + m_{i+1},\ i = 1, 2, \cdots, n - 1. \tag{5}$$

Now, each feature map f_i represents a segment of the stroke. By learning to connect and curve the $n - 1$ "segments", we are able to reconstruct the stroke. By appending the encoded color and radius data we now have the feature with shape $64 \times 64 \times n$. We then feed the features into three convolutional layers with batch-normalization (Ioffe & Szegedy, 2015) activated by LeakyReLU (Xu et al., 2015) without pooling, then follows one deconvolution layer with batch-norm and LeakyReLU and the other with the same configuration except activated by tanh.

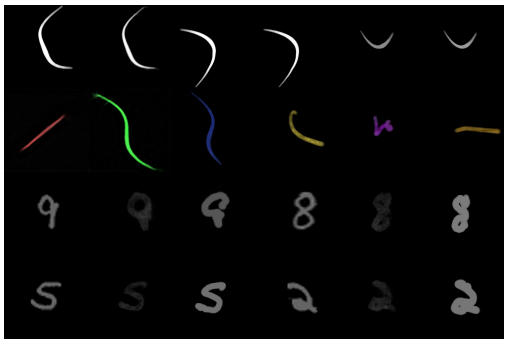


Figure 2: A trained StrokeNet generates images that resemble the output of painting software. The first row depicts results generated by our model (left) and by the software (right) given the same input. The second row shows the model could produce strokes with color and texture using simple arithmetic operations. The third and fourth row shows the model's ability to draw MNIST digits (left) on both its own generative model (middle) and real-world painting software (right).

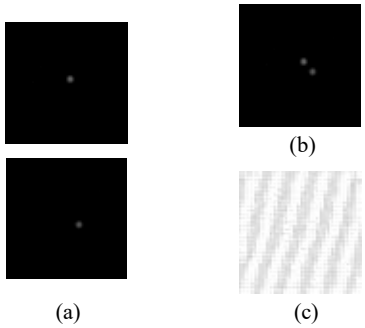


Figure 3: (a) A trained position encoder maps a (x_i, y_i, p_i) tuple to spatial feature. (b) Each pair of neighbouring features are added together to eliminate sparsity and preserve sequential information. (c) A trained brush encoder encodes color and radius information into a spatial feature which is later concatenated to the end of position features.

3.3 Agent

The agent is just a plain VGG (Simonyan & Zisserman, 2014)-like CNN that encodes the target image into its underlying stroke representation s. The convolutional layers For the last layer it uses three different FC-decoders with different activations to decode position (tanh), pressure (sigmoid) and brush data (sigmoid). The agent network used in our experiment is shown in Figure 2. Here we used average-pooling instead of max-pooling to improve gradient flow.

3.4 Environment

We first built a painting software using JavaScript and WebGL. We later tailored this web application for our experiment. [2]

The spline we used to fit the sampled points from user input is centripetal Catmull-Rom spline (Catmull & Rom, 1974; Barry & Goldman, 1988).

Let $P_i = [x_i, y_i]^T$ denote the coordinate of a sampled point. For a curve defined by points P_0, P_1, P_2, P_3, the spline can be produced by: ,

$$C = \frac{t_2 - t}{t_2 - t_1} B_1 + \frac{t - t_1}{t_2 - t_1} B_2, \tag{6}$$

where

$$\begin{aligned} B_1 &= \frac{t_2 - t}{t_2 - t_0} A_1 + \frac{t - t_0}{t_2 - t_0} A_2, \quad B_2 = \frac{t_3 - t}{t_3 - t_1} A_2 + \frac{t - t_1}{t_3 - t_1} A_3, \\ A_1 &= \frac{t_1 - t}{t_1 - t_0} P_0 + \frac{t - t_0}{t_1 - t_0} P_1, \quad A_2 = \frac{t_2 - t}{t_2 - t_1} P_1 + \frac{t - t_1}{t_2 - t_1} P_2, \\ A_3 &= \frac{t_3 - t}{t_3 - t_2} P_2 + \frac{t - t_2}{t_3 - t_2} P_3, \quad t_{i+1} = \|P_{i+1} - Pi\|_2^{\alpha} + t_i. \end{aligned} \tag{7}$$

with $\alpha = 0.5$, $t_0 = 0$ and $i = 0, 1, 2, 3$. By interpolating t from t_1 to t_2 linearly, we generate the curve between P_1 and P_2. The pressure values between neighbouring points are interpolated linearly.

We then draw circle centers at each interpolated point. For each pixel inside the circle, its color depends on both the pressure and the brush color, as well as the original color at that position

[2] Code for the web application available at: `https://github.com/vexilligera/drawwebapp`.

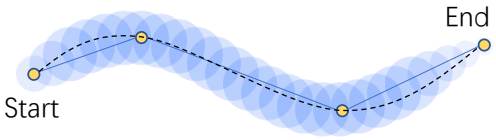


Figure 4: Illustration of how a stroke is rendered. Our engine first fit the control points with Catmull-Rom spline, then for each interpolated point it draws a circle to fill the stroke path. Size and color of each circle depend on the pressure of the corresponding point.

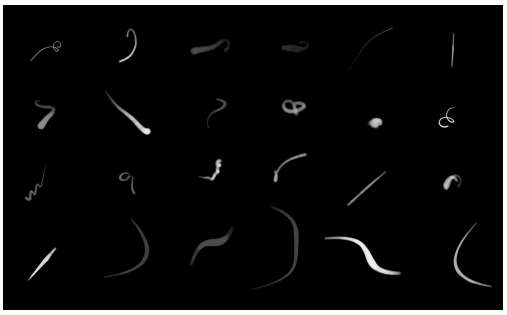


Figure 5: Images from our dataset. The first three rows demonstrate the samples generated from three-body motion. The fourth row is obtained from simple polynomials.

by alpha channel mixing. Our software supports multiple image layers, so that the stroke is first rendered to an intermediate layer and blended to the target depending on the blending mode and the brush parameters. The implementation detail is beyond the scope of this paper.

4 Experiment

4.1 Dataset for Generator

For the generator, a **huge** dataset containing all different kinds of human scribbles and doodlings within a single stroke with varying pressure would be an ideal candidate for training, which does not exist. We need to generate a large amount of samples each of length n that captures both the randomness and the smoothness of human writing. Pure random walk results in wiggling strokes which is not desirable, while polynomials offer too little variation in shapes. Thus it is evident to incorporate **chaos**, most notably, the motion of three-body(Nielsen et al., 2001).

There's no closed-form solution to three-body problem, thus error accumulates in simulation using numerical methods, leading to unpredictable and chaotic results. We simulate three-body motion in space (z-component for pressure) with random initial conditions and sample the trajectories as strokes for our dataset. The simulation is done with a set of equations using Newton's universal law of gravitation:

$$\begin{cases} \vec{F_1} = \frac{Gm_1m_2}{\|P_2-P_1\|_2^3}(P_2-P_1) + \frac{Gm_1m_3}{\|P_3-P_1\|_2^3}(P_3-P_1) \\ \vec{F_2} = \frac{Gm_1m_2}{\|P_1-P_2\|_2^3}(P_1-P_2) + \frac{Gm_2m_3}{\|P_3-P_2\|_2^3}(P_3-P_2) \\ \vec{F_3} = \frac{Gm_1m_3}{\|P_1-P_3\|_2^3}(P_1-P_3) + \frac{Gm_2m_3}{\|P_2-P_3\|_2^3}(P_2-P_3) \end{cases}, \tag{8}$$

where $P_i (i = 1, 2, 3,\ P_i \in \mathbb{R}^3)$ denotes the position of the three objects respectively, $\vec{F_i}$ denotes the gravitational force exerted on the ith object. In our simulation we set mass $m_1 = m_2 = m_3 = 1$ and gravitational constant $G = 5 \times 10^{-5}$. We also always keep our camera (origin point) at the center of the triangle formed by the three objects to maintain relatively stable "footage".

Using this method we collected about 550K images since there's virtually no cost to generate samples. Along with data generated from simple polynomials, this dataset consists of approximately 590K images. Samples from the dataset are shown in Figure 5.

4.2 Datasets for Agent

To prove the effectivess of our neural environment, we trained an agent to perform drawing task. The agent is later transferred to real-world environment to validate its ability.

MNIST (LeCun & Cortes, 2010) contains 70,000 examples of handwritten digits, of which 10,000 samples are divided into a test set. Each example is a 28×28 grayscale image.

Omniglot (Lake et al., 2015) contains 1623 different handwritten characters from 50 different alphabets. It is a challenging task to learn to write samples from the dataset because the shape of the characters varies significantly. Also, the agent is only allowed to draw one stroke. Thus it's up to the agent how to depict the character as accurately as possible.

We resized all the input images to 256×256 with antialias and paddings. We also dimmed the images by a factor of 0.4 because this makes it closer to the output brightness of our generator. We also enhanced the brightness of the output images from the generator in Figure 6 and 7 for more readability.

4.3 Training Methodology

Intially we train the **position encoder** separately. As described in Section 3 we first guide it with function pos that maps a coordinate to a 64×64 matrix with l^2 distance to measure the loss. We train it until the loss is less than 3×10^{-5}.

Next we **freeze** the **position encoder** and train the other parts of the generator with l^2 loss to measure the performance. Using the three-body dataset, we train the network until the loss drops around 700.

We then train the **generator** as a whole with the 590K dataset and the loss converges at around 200. In practice, we generate 65,536 samples each time to train the generator for no more than 4 epochs. We then generate another 65,536 samples to evaluate its performance and for the next-round training. As the generator converges and no longer improves, we have accumulated about 550K images. We finally train the generator on the 550K three-body images along with the 40K polynomial images until convergence.

To train the **agent**, we freeze the **generator**. Denote the agent loss as l_{agent}, the generated image and ground-truth image as i_{gen} and i_{gt} respectively, the loss is defined as:

$$l_{agent} = \|i_{gen} - i_{gt}\|_2 + \frac{\lambda}{n-1} \sum_{k=1}^{n-1} \|P_k - P_{k+1}\|_2^2 , \tag{9}$$

where $P_k = [x_k, y_k, p_k]^T$ is the data describing the kth anchor point on the stroke. Here the summation term constrains the average distance between neighbouring points, where λ denotes the penalty strength. If we drop this term, the agent fails to learn the sequence of the points of the stroke because the generator itself is, after all, not robust to all cases of input, and the agent is very likely to "overfit" the generator.

4.4 Result

We trained our models by the procedures above. The generator is firstly trained to become stable for the agent to be trained on. After several hours of training on a single NVIDIA Tesla P40, we've obtained a usable approximation for our agent. As we can see from Figure 6, for large, smooth strokes, the generator could predict the shape and color precisely, while for delicate, intertwined ones (the leftmost sample on second row) the generated image is blurry.

From Figure 7 and 8, it is clear that for simple pictures like MNIST digits, the agent is able to learn the general structure of the digits and produce strokes that generate digit images similar to the input. However, the agent is not always stable. For more complicated characters such as those in Omniglot, the agent could only give a sketch of the character, due to the constraint of single stroke. This is also easily seen from the loss functions in Figure 9, where in (c) the loss of agent trained on MNIST converged rapidly to a relatively low value compared to the loss depicted in (d). The penalty strength λ is set lower for the Omniglot agent because the characters from the dataset are more complex, thus we would allow more freedom when the agent tries to generate the stroke.

The generator is able to transfer its ability to fit our three-body dataset to unseen strokes that generate MNIST and Omniglot images. The spatial information is mostly correct thanks to the reliable performance of the position encoder. However, the color of the strokes is generally darker than the ground-truth, so that we had to enhance the brightness to provide better readability of the figures. It seems that the generator neural network isn't confident about what it is being produced,

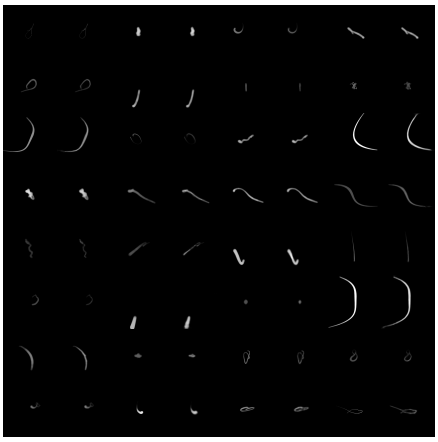


Figure 6: The generator tries to predict what the real environment would ouput given the same input stroke data. Software output (left), generator prediction (right).

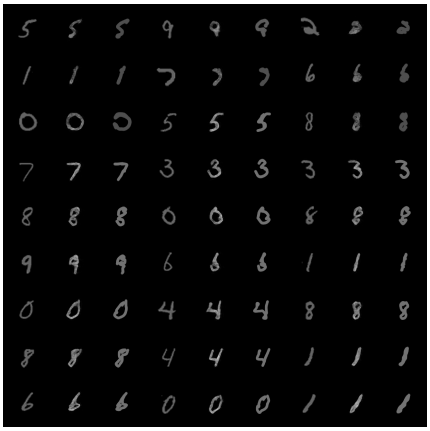


Figure 7: Agent trained on MNIST dataset can write digits. MNIST sample (left), generator output (middle), Draw WebApp reconstruction (right). Figure 8 is of the same layout.

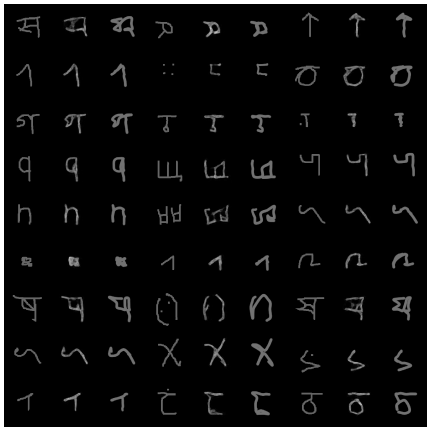


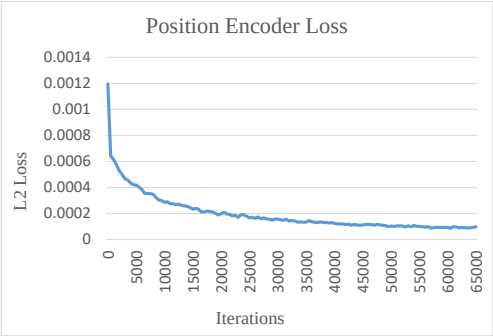
Figure 8: Agent trained on Omniglot dataset learns to "sketch" the characters. Since many characters consist of multiple strokes while the agent can only draw one, the agent tries to capture the contour of the character.

which makes sense because, after all, the distribution of our dataset still differs from real-world quite a lot. A mitigation of the situation is described in later discussion.

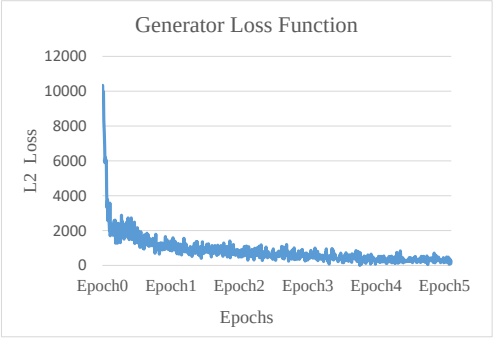
5 Discussion

For future work, there're several major improvements we want to make both to the network structure and to the algorithm.

One of the biggest limitations of our model is that we only solved the problem of single stroke rendering and learning. One easy solution is that we can wrap the whole network inside a recurrent structure. The simulation of rendering multiple strokes on one canvas is easy, we simply render all the strokes separately and add them together and clip out-of-range pixel values, which is exactly what the engine does in our painting software. On the other hand, our software does real-time rendering on user drawing a stroke, since the Catmull-Rom spline algorithm only needs four points to generate a segment of the curve. However, if we also do this recurrently, the time complexity of the generator is raised by an order of n. Of course one can use attention mechanism to cut unnecessary computation and focus only on the region where a new segment is being generated. In this experiment though, we made simplification to avoid real-time rendering because empirically

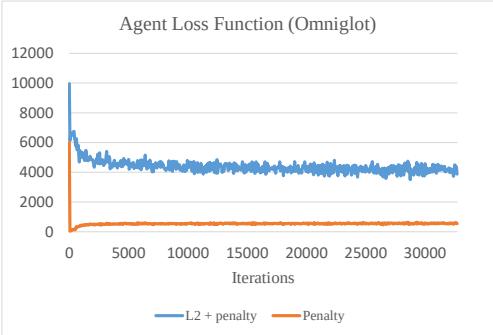


(a) Loss of encoder when trained separately at first.



(b) Loss of generator trained on our 590K dataset.

(c) Agent trained on MNIST, $\lambda = 1.5 \times 10^5$.



(d) Agent trained on Omniglot, $\lambda = 1.0 \times 10^5$.

Figure 9: Training loss of generator and agent. The agent loss equals to the l^2 distance between the generator output and agent input plus the penalty term constraining the average point distance within a stroke. For (c) and (d) the learning rate is set to 8×10^{-5}, batch size equals to 64.

humans tend to evaluate the effect of a stroke after the moving hand is lifted up from the canvas, when the stroke is drawn.

Apart from that, the generator and the agent were trained as two separate parts throughout the experiment. We can somehow train them as a whole in the following way: during the training of the agent, store all the intermediate stroke representation. After a period of training, sample images from the real environment with the stroke data just collected, and train the generator with the data. By doing so in an iterative manner, the network could be more aware of what its exactly doing.

Finally, for the evaluation metrics, the naive l^2 loss can be combined with discriminator loss given its popularity and mightiness to achieve adversarial learning. We did test WGAN-GP (Gulrajani et al., 2017) conditioned on the output of the position encoder to train the generator, which didn't work out well. Reason may be that the ground-truth output image of a given set of stroke data is precisely generated through algorithms, thus it would be challenging for the discriminator to tell whether one approximation is more accurate than another. As a result, we simply adopted l^2 loss to measure the distance between images produced by the generator network and the software.

6 Conclusion

In this paper we bring a proof-of-concept that an agent is able to learn from its neural simulation of an environment. Especially when the environment is definite given the action, or contains a huge action space that require massive computation and training if using traditional reinforcement learning methods that requires direct interactions between the agent and environment, our method provides a layer that could serve as the model's own perception of its surroundings. The agent trained with our method converges quickly and is able to adapt its skills to real world. Hopefully such approaches can be useful for more sophisticated inverse rendering problems such as style transfer by stroke as well as other reinforcement learning problems.

REFERENCES

Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 214–223, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

Phillip J. Barry and Ronald N. Goldman. A recursive evaluation algorithm for a class of catmull-rom splines. *SIGGRAPH Comput. Graph.*, 22(4):199–204, June 1988. ISSN 0097-8930. doi: 10.1145/378456.378511.

Edwin Catmull and Raphael Rom. A class of local interpolating splines. *Computer Aided Geometric Design*, pp. 317–326, 1974.

Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, S. M. Ali Eslami, and Oriol Vinyals. Synthesizing programs for images using reinforced adversarial learning. *CoRR*, abs/1804.01118, 2018.

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. D. Lawrence, and K. Q. Weinberger (eds.), *Advances in Neural Information Processing Systems 27*, pp. 2672–2680. Curran Associates, Inc., 2014.

Alex Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013.

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

David Ha and Douglas Eck. A neural representation of sketch drawings. *CoRR*, abs/1704.03477, 2017.

David Ha and Jürgen Schmidhuber. World models. *CoRR*, abs/1803.10122, 2018.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. *CoRR*, abs/1312.6114, 2013.

Brenden M. Lake, Ruslan Salakhutdinov, and Joshua B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 2015.

Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

Matthew M. Loper and Michael J. Black. OpenDR: An approximate differentiable renderer. In *Computer Vision – ECCV 2014*, volume 8695 of *Lecture Notes in Computer Science*, pp. 154–169. Springer International Publishing, September 2014.

Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *CoRR*, abs/1411.1784, 2014.

Thu Nguyen-Phuoc, Chuan Li, Stephen Balaban, and Yong-Liang Yang. Rendernet: A deep convolutional network for differentiable rendering from 3d shapes. *arXiv preprint arXiv*, abs/1806.06575, 2018.

E. Nielsen, D.V. Fedorov, A.S. Jensen, and E. Garrido. The three-body problem with short-range interactions. *Physics Reports*, 347(5):373 – 459, 2001. ISSN 0370-1573. doi: https://doi.org/10.1016/S0370-1573(00)00107-1.

Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *CoRR*, abs/1511.06434, 2015.

Jürgen Schmidhuber. Making the world differentiable: On using self-supervised fully recurrent neural networks for dynamic reinforcement learning and planning in non-stationary environments. Technical report, 1990.

Jürgen Schmidhuber. One big net for everything. *CoRR*, abs/1802.08864, 2018.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

Kihyuk Sohn, Honglak Lee, and Xinchen Yan. Learning structured output representation using deep conditional generative models. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 28*, pp. 3483–3491. Curran Associates, Inc., 2015.

Ning Xie, Hirotaka Hachiya, and Masashi Sugiyama. Artist agent: A reinforcement learning approach to automatic stroke generation in oriental ink painting. *CoRR*, abs/1206.4634, 2012.

Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *CoRR*, abs/1505.00853, 2015.