Learning to Simulate Dynamic Environments using Generative Adversarial Networks (Inspired by GameGAN)

BHASKAR MISHRA, University of Florida, USA

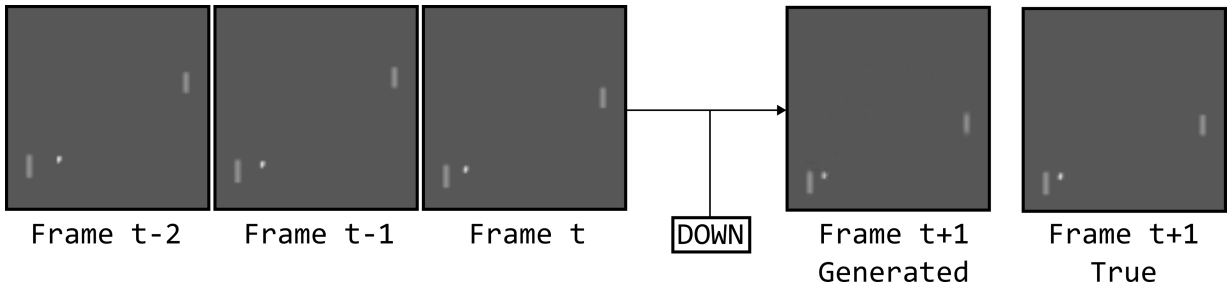

Fig. 1. Frame prediction example

When training AI agents for solving problems pertaining to real world environments, it is often useful to utilize simulations of those environments in order to accrue training data. Sometimes, however, these simulations can contain complex relationships and behaviors that are difficult to formalize and program through standard procedural programming. In this project, we use a Generative Adversarial Network to learn to simulate the rules and intricacies of a dynamic environment using an agent's interactions with the environment for training data. Specifically, we aim to simulate the game of Pong, training a model that takes frames from an arbitrary state in the game along with an action, and produces an image corresponding to the frame produced in the game after the given action is taken. This project is a heavily simplified version of the approach used to simulate VizDoom and Pacman in GameGAN.

Additional Key Words and Phrases: neural networks, generative adversarial networks, deep Q-learning

1 INTRODUCTION

When training AI agents for solving problems pertaining to real world environments, it is often useful to utilize simulations of those environments in order to accrue training data. For example, in one recent Deep Reinforcement Learning paper, simulations are used to train a robotic arm to do various tasks, and the resulting agent is used to perform those same tasks in the real world using a real robotic arm [2]. Sometimes, however, these simulations can contain complex relationships and behaviors that are too difficult to formalize and program through standard procedural programming. In this project, we aim to exhibit an example of how a Generative Adversarial Network (GAN) [3] can be trained to simulate the rules and intricacies of a dynamic environment using only an agent's interactions with the environment as training data. Specifically, we tackle the dynamic environment of the game of Pong. We aim to train a GAN that uses the three last seen frames in a Pong game, along with an action (either up, down, or stay), in order to produce a fourth frame which approximates the resulting frame in Pong if the given action were taken. We use some techniques from Deep Reinforcement Learning to create an agent which fully explores our environment, though given knowledge of an environment, one could program an agent which explores the majority of the environment space.

Author's address: Bhaskar Mishra, bmishra1@ufl.edu, University of Florida, Florida, USA.

2 RELATED WORKS

The work most directly related to this project is GameGAN [7]. GameGAN also aims to simulate dynamic environments, and in the work, they use much more complex GAN architectures than the one that will be used in this project in order to simulate the game of Pacman and the VizDoom environment [6]. In addition to simulating the environments, GameGAN also trains agents in those simulated environments using Deep Q-Learning [9], showing how such an approach can be used to train agents for the original environments, and also shows how their network can be used to create visually modified versions of the original environments. This project aims to implement a heavily simplified version of GameGAN's approach in the context of the simple environment of Pong.

Another work that uses a similar approach uses a concept called World Models [4]. They use a generative recurrent neural network in order to learn a compressed model of the real environment, and then use the simulated environment to train Reinforcement Learning based agents. The primary difference between this "World Models" approach and the approach used in GameGAN is the architecture used and GameGAN's ability to visually modify its environments [7].

Finally, this work also shares many similarities with the research area of video prediction. Video prediction typically involves using previous frames from a video to predict future frames [10], just as this project uses previous frames from a game of Pong to predict future frames. The only difference is that our approach takes into account conditional information - the action being taken - when predicting future frames.

3 OVERVIEW

This project is split into three phases. We begin by training an agent in the original environment of Pong using a Reinforcement

Learning (RL) approach known as Deep Q-Learning [9]. The Pong environment is emulated using the implementation in the Arcade Learning Environment (ALE) [1]. Once we are able to create a well-performing agent, we run several simulations in the environment with the agent playing Pong with high entropy, and save the frames from the simulated games as data. Of the games simulated, 66.7% of the games are set up such that at each state, there is a 20% chance of a random action being taken instead of the agent's intended action, and in the other 33.3% of games, there is a 50% chance of a random action being taken. The next phase is designing and testing the Generator and Discriminator architectures for our GAN. We use the generated data to test the ability of our Generator network to produce high quality images and the ability of our Discriminator to differentiate between the images produced by our Generator and real images. Finally, in the last phase, we tuned the hyper-parameters of our GAN, and trained a final model that could produce future frames with good image quality.

4 DEEP Q-LEARNING

We will begin by discussing the process by which we trained our RL Agent.

4.1 Theory

Deep Q-Learning is based on an older RL technique known as Q-Learning [11]. In Q-Learning, an agent acts so as to maximize future reward. Q-Learning makes an assumption that future reward is discounted by a factor γ at each time step, and hence future rewards R_t at a particular time step t is defined by $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r'_t$ where r_t represents the reward achieved at time step t [9]. Let S denote the set of all possible states in a game and let A denote the set of all possible actions. Let V for valid denote the set of pairs (s, a) with $s \in S$ and $a \in A$ such that a is a valid action at state s. Then Q-learning seeks to find a function $Q^* : V \rightarrow \mathbb{R}$ such that $Q^*(s, a) = \max_\pi \mathbb{E}\left[R_t | s_t = s, a_t = a, \pi\right]$, where π is a policy determining which actions are taken at each state.

The key concept of Q-learning is that this Q^* function obeys what is called *Bellman's Equation*. Given an arbitrary $(s, a) \in V$, Bellman's Equation states that

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s', a') | s, a\right]$$

where r is the reward from state s, the object s' is the state resulting from action a being taken at state s, and $\mathcal{E}$ is the environment, determining the distribution of what s' can be. Based on the Bellman's Equation, techniques like Q-learning use an iterative approach towards finding Q^*, making an initial guess Q_1 of the function's definition, and then getting progressively better definitions by setting $Q_{i+1} = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q_i(s', a') | s, a\right]$ [11].

How Deep Q-Learning differs from standard Q-Learning is that the former uses Deep Neural Networks in order to model the Q function. The Deep Neural Network, called a Deep Q-Network (DQN), takes states as inputs and for each possible action at each state, it outputs an approximation of the output of the Q^* function for the state, action pair. The DQN uses a similar version of the iterative approach defined above in order to produce targets for the network and train the network towards the Q^* function. The exact algorithm is as follows [9]:

Algorithm 1 Deep Q-Learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize DQN Q with random weights
for episode=1, M **do**
  Initialize state s_1 and preprocessed state $\phi_1 = \phi(s_1)$
  for t=1, T **do**
    With probability ϵ, select a random action a_t
    Otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t and observe reward r_t and state s_{t+1}
    Preprocess $\phi_{t+1} = \phi(s_{t+1},)$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1}) \in \mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{if } \phi_{j+1} \text{ terminal} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

In the algorithm, the function ϕ maps states to data in a form desired for the particular neural network architecture being used. Replay memory is a dataset that stores experiences encountered by the agent. Data from the Replay Memory is used to train the DQN.

4.2 Implementation

In order to implement our own Deep Q-Learning agent for the game of Pong, we use the "PongNoFrameSkip-V4" environment from the Arcade Learning Environment [1] along with a few wrappers[1] used in the code for the original Deep Q-Learning work [9].

For our DQN, we collect image data such that each state contains the image data at that frame, as well as the image data from the two previous frames. By default, all the images in the Pong environment are 210×160 RGB images. We preprocess these images by converting the images to grayscale, cropping them such that only the relevant portion of the game is visible, then scaling the image down to an 84×84 image, and then finally normalizing the values to the range $[0, 1]$. This preprocessing operation is visualized in Figure 2.

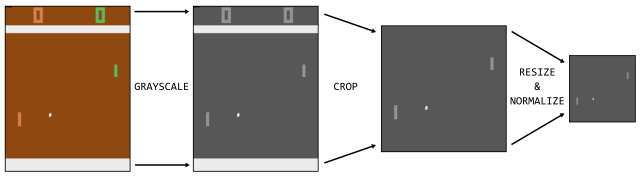

Fig. 2. Frame Preprocessing

Our network then passes the preprocessed frames through three convolution layers, each using ReLU activation functions, and two fully-connected linear layers, with the first using a ReLU activation

[1]The exact wrappers we use are NoopResetEnv, MaxAndSkipEnv with a skip of 4, EpisodicLifeEnv, and FireResetEnv.

function and the second using no activation function. The output consists of 3 units representing the Q value for the actions stay, up, and down, respectively. For the complete details of the architecture, see Figure 3.

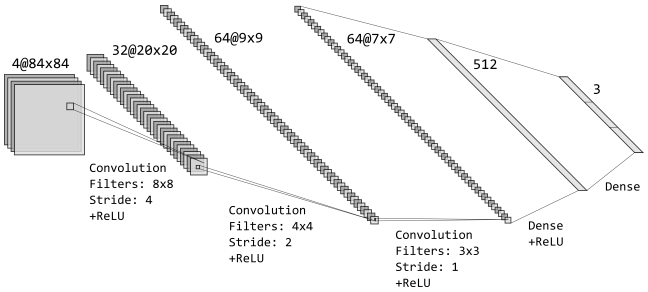

Fig. 3. DQN Architecture - Diagram created using Alex Lenail's NN-SVG

Finally, for the implementation of the Deep Q-Learning algorithm, we use a modified approach which uses a second Target DQN that is used exclusively for determining targets. This Target DQN is updated to equal the first DQN that is being actively trained every 1000 minibatches. This is done so that the first DQN can train towards a temporarily fixed target, rather than a moving one.

4.3 Training

During the actual training process, much time was spent tuning hyper-parameters and modifying the preprocessing steps in order to produce an agent that converged. Initially, our preprocessing step did not contain the normalization step. At this stage, multiple days were spent trying to tune hyper-parameters or use more advanced algorithms, but everything resulted in the agent remaining at the minimum total reward per episode of −21. It was only when we normalized the images to have all values between 0 and 1 that the model was able to converge to a policy with the 50 episode moving average for total reward being approximately 17.9 as can be seen in Figure 4.

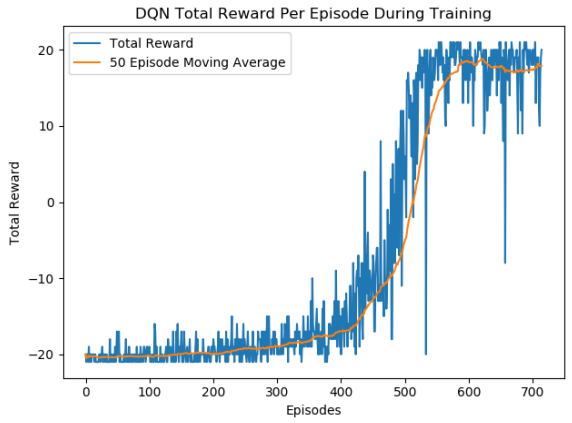

Fig. 4. Total reward achieved by Deep Q-Learning Agent each episode during training

In training, the value for ϵ was annealed from 1.00 to 0.02 over the course of one million frames. As a result, the rewards above underestimate the true performance of the agent. When tested in a pong environment with ϵ set to 0, our agent achieved a total reward of 21, the maximum reward for a single episode, in each of 100 episodes.

5 DATA GENERATION

With a functioning agent to explore the environment with, we will now discuss the data generation process. Instead of using the best Deep Q-Learning agent from the previous section, we used the agent from Epoch 500. This was because the best agent had a strategy that was too uniform, and as a result, wasn't optimal for exploring the environment. Using the agent from Epoch 500, we simulated several episodes, with 66.7% of the episodes using $\epsilon = 0.2$, and the remaining 33.3% of episodes using $\epsilon = 0.5$. In order to avoid dealing with frames from periods between when the ball leaves the screen and when a new ball appears, we end episodes as soon as a reward is given (when the ball leaves the screen, either on the player's side or the opponent's side). Each of the frames from the simulated episodes, along with the actions taken to get to those frames, are stored in a set of files, with each file containing all the frames from a total of 50 episodes. We generated 60 files, amounting to 3000 episodes of data.

6 GENERATOR AND DISCRIMINATOR

In this section, we will discuss the process through which we developed the architectures for our Generator and Discriminator models and how we tested our architecture designs.

6.1 Generator

The general architecture of our Generator Neural Network follows the layout shown in Figure 5, taking the last 3 frames, along with an action, as input, and outputting a new frame.

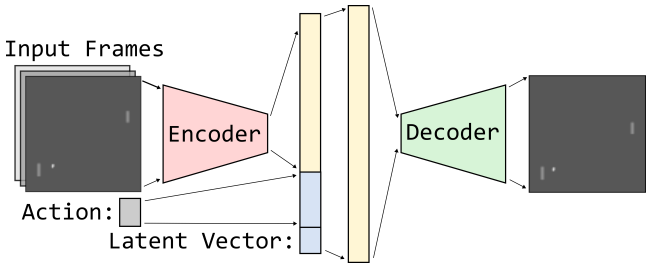

Fig. 5. Generator architecture layout

Initially, we tried to use a very simple encoder and decoder, using three convolution layers and three transposed convolution layers, each with a maximum of 16 3×3 filters. In order to test our architecture, we attempted to use the data produced by our agent to train our generator model using a standard supervised learning approach. We used the data generated by our agent to create input frames, input actions, and targets, and sought to minimize the Mean Squared Error between the produced images and the target images. Though this original architecture did converge, ending at an average loss of

approximately 0.0003, it was unable to model important features of the images. An sample output of the model can be seen in Figure 6.

Fig. 6. Simple generator model sample output

The model was able to produce bands of brighter pixels in the region where both paddles would typically be, but it wasn't able to precisely determine the location of the paddle, and didn't produce a ball at all. This suggested that the model wasn't complex enough to express the relationships we wanted it to. To resolve this, we changed the model architecture, increasing both the number of filters and the number of convolution / transposed convolution layers.

From here on out, we will use `Conv(a, b, c, d)` to denote a 2D convolution layer with `a` filters, `b` filter size, `c` stride, and `d` padding. We will use `BatchNorm2D(a)` to denote a 2D batch normalization layer with `a` features, and `BatchNorm1D(a)` to denote a 1D batch normalization layer with `a` features. We will use `Dense(a)` to denote a fully connected layer producing `a` outputs. We will use `ConvT(a, b, c, d, e)` to denote a 2D tranposed convolution layer with with `a` output channels, `b` filter size, `c` stride, `d` padding, and `e` output padding. We will use `LeakyReLU(a)` to denote a leaky ReLU activation function with negative slope `a` and `Reshape(s)` to denote a transformation that reshapes input vectors into shape `s`.

For our final generator architecture, we defined our Encoder and Decoder architectures as follows:

Encoder Architecture
`Conv(32, 3, 2, 1)`
`LeakyReLU(0.2)`
`Conv(32, 3, 1, 0)`
`LeakyReLU(0.2)`
`Conv(32, 3, 2, 0)`
`LeakyReLU(0.2)`
`Conv(32, 3, 1, 0)`
`LeakyReLU(0.2)`
`Conv(32, 3, 2, 0)`
`LeakyReLU(0.2)`
`Reshape(2048)`

Decoder Architecture
`Reshape((256, 7, 7))`
`ConvT(256, 3, 1, 0, 0)`
`LeakyReLU(0.2)`
`ConvT(128, 3, 2, 0, 0)`
`LeakyReLU(0.2)`
`ConvT(64, 4, 2, 0, 0)`
`LeakyReLU(0.2)`
`ConvT(32, 4, 2, 0, 0)`
`LeakyReLU(0.2)`
`ConvT(1, 3, 1, 0, 0)`
`LeakyReLU(0.2)`

These architectures were heavily based on the Dynamics Engine and Simple Rendering Engine architectures used for Pacman in GameGAN [7]. The complete generator architecture then becomes the Encoder followed by a `Dense(256)` layer, at which point the action vector (a three dimensional one-hot vector representing the action) and an 8-dimensional random latent vector are concatenated to the output, followed by a `Dense(12544)` layer and the Decoder.

We again tested this architecture by training the model using a standard supervised learning approach. Initially, we used a fixed learning rate of 0.001. This led to the model's converging to a very good average loss of 0.000059 but then suddenly diverging to 132271614. This is likely due to the learning rate being too large for that point, causing the model to experience explosive gradients. We resolved this by training the model such that the learning rate was updated to 0.0001 once the average loss went below 0.00012. With this, our model converged to an average loss of 0.00004, a 32% improvement over the minimum average loss achieved by the fixed learning rate approach.

Some sample outputs of the generator compared to the ground truths can be seen in Figure 7.

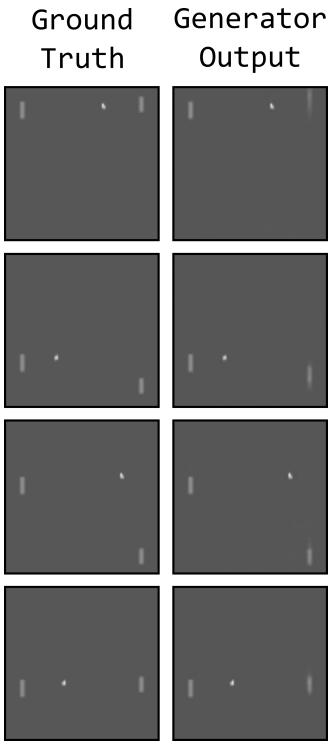

Fig. 7. Sample output from generator

The samples above show why a standard supervised learning approach fails in this context and why something similar to a GAN would be needed for high quality outputs. Since our Pong environment is not completely deterministic, when players move UP or DOWN, there may be a few possible positions that the right paddle can end up in. Since our standard supervised learning approach uses Mean Squared Error, it is incentivized to brighten pixels in all possible regions the paddle can be in, brightening pixels that are more likely to contain the paddle more intensely than those that are less likely. This ensures that the **expected** Mean Squared Error is minimized. This can be observed in the samples, as we see the

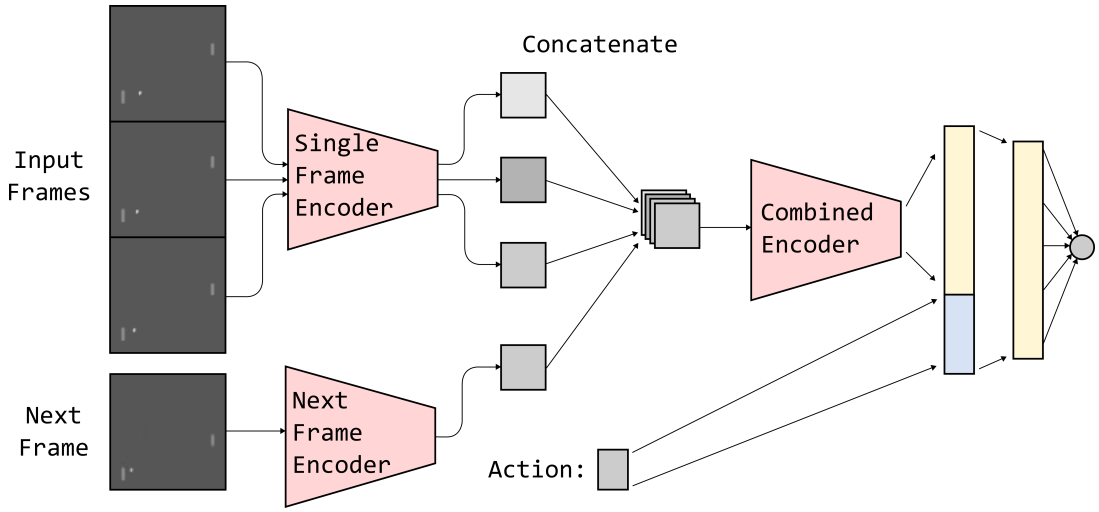

Fig. 8. Discriminator architecture layout

right paddles seem to have trails of less bright pixels surrounding the bright pixels in the center.

A GAN resolves the above problem by changing the loss function to an Adversarial loss function. In this case, the generator aims to maximize the discriminator's measure of the "realness" of the produced images, given the input images and the input action. Since light colored pixel trails easily make a produced image stand out from real frames, the GAN will not produce such pixel trails. The GAN will produce images that look as real as possible, which is exactly what we are looking for in this context.

6.2 Discriminator

The general architecture of our Discriminator follows the layout shown in Figure 8. The Single Frame Encoder and the Next Frame Encoder have identical architectures defined as follows:

Single Frame / Next Frame Encoder
`Conv(16, 5, 2, 0)`
`BatchNorm2D(16)`
`LeakyReLU(0.2)`
`Conv(32, 5, 2, 0)`
`BatchNorm2D(32)`
`LeakyReLU(0.2)`
`Conv(32, 3, 2, 0)`
`BatchNorm2D(32)`
`LeakyReLU(0.2)`
`Conv(32, 3, 2, 0)`
`BatchNorm2D(32)`
`LeakyReLU(0.2)`

Here the Single Frame Encoder is an encoder for a single frame, so all three input frames are passed individually through the encoder. The Combined Encoder has a very simple architecture using a single convolution layer and a single batch normalization layer defined as follows:

Combined Encoder
`Conv(32, 3, 1, 0)`
`BatchNorm2D(32)`
`LeakyReLU(0.2)`
`Reshape(32)`

The action vector is passed through a `Dense(16)` layer, followed by a `LeakyReLU(0.2)` activation function, before being concatenated to the output of the Combined Encoder, and after concatenation, the result is passed through a `Dense(32)` layer, followed by a `BatchNorm1D(32)` layer and a `LeakyReLU(0.2)` activation function. Finally, the output of the dense layer is passed through `Dense(1)` layer to produce the output.

In order to test our discriminator architecture, we used the generator trained in the previous section in order to produce fake images, and then used the real images, input images, and input actions provided by our data set in order to train our discriminator in a standard supervised learning manner. The discriminator easily converged to a low average loss of 0.005 within just 20 epochs, and seemed to be continuing to decrease after that. This suggested that our discriminator was sufficiently complex for our GAN.

7 GENERATIVE ADVERSARIAL NETWORK

With architectures in place for the Generator and Discriminator, we can begin discussing our GAN implementation.

7.1 Theory

Generative Adversarial Networks or GANs are an approach to solving generative modeling problems using two competing neural networks, a generator G and a discriminator D [3]. The role of the generator is to generate samples from the distribution that is being modeled, and the role of the discriminator is to determine whether a sample is from the original distribution or the generator's distribution.

A typical formulation of the loss functions for networks G and D is as follows:

$$J^{(D)} = -\frac{1}{2}\mathbb{E}_{x\sim p_{data}} \log D(x) - \frac{1}{2}\mathbb{E}_z \log(1 - D(G(z)))$$
$$J^{(G)} = -\frac{1}{2}\mathbb{E}_z \log D(G(z)).$$

Here p_{data} is the distribution of the dataset taken from the true distribution, and z is a latent vector, which is an arbitrary random vector of some pre-selected size, that is used as the input for the Generator. The latent vector effectively represents a random seed that is used to determine which element of the modeled distribution to sample.

The intuition behind the above loss function is that $J^{(D)}$ is minimized when D maps all samples from the true distribution to a value of 1, representing "real" samples, and maps all samples generated by G to a value of 0, representing "fake" sampes. On the other hand, $J^{(G)}$ is minimized when all the samples generated by G are mapped to 1 by D.

In order to then train the GAN, one can go back and forth between training the discriminator and training the generator. When training the discriminator, minibatches can be sampled with all real samples and all fake samples, and the discriminator's weights can be updated using backpropagation and $J^{(D)}$. When training the generator, mini-batches can be sampled with all fake samples, and then the generator's weights can be similarly updated using backpropagation and $J^{(G)}$.

7.2 Implementation

For our context, we don't seek to simply model a distribution, but aim to model a distribution under some conditions - we want to sample frames from the game Pong, given the previous three frames and an action. The type of GAN that is typically used to solve this problem is some variant of a Conditional GAN. Conditional GAN's are identical to regular GAN's except that they take the conditional data as additional inputs to their networks G and D [8]. Since our problem is very similar to that of Image to Image Translation, we use a very similar training approach for our GAN as the one used in the Pix2Pix model [5]. Specifically, we train a regular Conditional GAN, and modify the $J^{(G)}$ loss to also include an L1 Loss between the produced frame and the real frame, with the L1 Loss being weighted ten times as heavily as the ordinary GAN generator loss.

Additionally, rather than start our generator from scratch, we initialize the generator with the pre-trained weights from the generator we trained earlier. For training, we begin by training the GAN with learning rates for both G and D set to 0.0001 for 200 epochs, then train the GAN with both learning rates set to 0.00001 for 260 epochs. After not noticing much difference in performance from decreasing the learning rate, we increased the learning rates again, setting the learning rate for G to 0.0001 and the learning rate for D to 0.0002, and changed the training approach such that the discriminator was updated four times for every time the generator was updated. The GAN was trained like this for another 325 epochs before we had our final model.

7.3 Results

Here are some samples produced by our final GAN model. It can

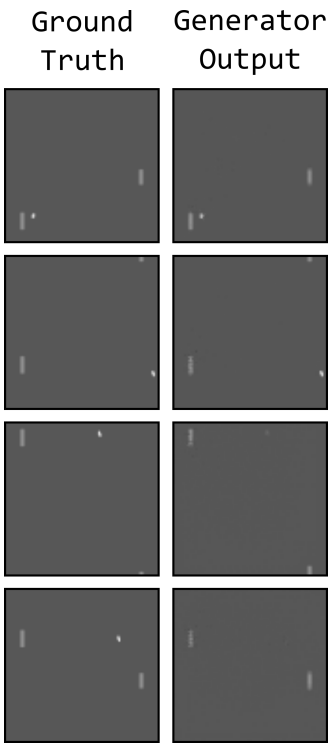

Fig. 9. Sample output from GAN

immediately be noticed that the GAN output has improved over the Generator's output with respect to the quality of the paddle's produced. There is no longer any white trail, and the paddle is always in a very precise location, typically very close to the true location. One area where the GAN seems to have regressed with respect to the Generator's output, however, is the ball. Though there are some examples where the GAN produces a ball with a bold white color, in most cases, the ball is colored with a light shade, as seen in the third image above, or not produced at all, as seen in the final image above.

We were unable to determine what the cause for this was. Intuitively, the lack of a ball would immediately give away to the discriminator that a produced frame was fake, so it doesn't seem like the generator should ever fail to produce a missing ball. Unfortunately, due to time constraints, we were unable to experiment with other GAN architectures. Still, in most cases the location of the ball is clearly correctly determined by the GAN (whether or not the ball is shaded boldy or lightly), and the paddles are also very precise with the location they're produced in. As a result, we believe we have at least somewhat exhibited an example of how a GAN may have the potential to simulate a dynamic environment like this one, or even a more complicated environment.

8 CONCLUSION

In this project, we successfully create a Deep Q-Learning agent for the game of pong, which we use to generate data in order to train our GAN. We design architectures for the generator and discriminator which we show to have sufficient expressability to solve the problem we're trying to solve. Finally, we train a GAN which is able to produce decent predictions of future frames. The location of paddles and the ball in the produced frames are mostly consistent with the rules and laws of the environment. Our GAN model does, however, seem to have a weakness for clearly producing the left paddle and producing the ball with sufficiently intense pixels. Despite this, we believe our GAN model is able to serve as an example of how a GAN can be used to predict future states in a simulation. In a real world context, an approach similar to ours could be used to simulate more complex real world environments and then be used in order to improve decision-making in those complex environments.

9 FUTURE WORK

A clear path for Future Work is improving the quality of our GAN. Though we weren't able to experiment with our final model due to time constraints, in the future, we can determine whether the weakness of the GAN lies in the hyper-parameters, the architecture, or the data distribution itself.

In addition to this path, it is also worth noting that we didn't construct a complete simulation in this project. Though we were able to make predictions about future frames given data from previous frames and a decision, in order for our GAN to truly simulate dynamic envionments, it needs to be able to take frames produced by itself as input as well. Training the GAN to be able to do that, as is done by GameGAN [7], is a worth pursuit with regards to Future Work.

Finally, trying to apply this approach to some real life context would also be very worthwhile, as it would provide a better understanding of the practicality and generalizability of this approach.

ACKNOWLEDGMENTS

Thanks to Dr. Toler-Franklin, whose class taught me several of the techniques used in this project.

REFERENCES

[1] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. 2013. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research* 47 (Jun 2013), 253–279. https://doi.org/10.1613/jair.3912

[2] Andrea Franceschetti, Elisa Tosello, Nicola Castaman, and Stefano Ghidoni. 2020. Robotic Arm Control and Task Training through Deep Reinforcement Learning. arXiv:2005.02632 [cs.RO]

[3] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Networks. arXiv:1406.2661 [stat.ML]

[4] David Ha and Jürgen Schmidhuber. 2018. Recurrent World Models Facilitate Policy Evolution. arXiv:1809.01999 [cs.LG]

[5] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. 2018. Image-to-Image Translation with Conditional Adversarial Networks. arXiv:1611.07004 [cs.CV]

[6] Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. 2016. ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning. arXiv:1605.02097 [cs.LG]

[7] Seung Wook Kim, Yuhao Zhou, Jonah Philion, Antonio Torralba, and Sanja Fidler. 2020. Learning to Simulate Dynamic Environments with GameGAN. arXiv:2005.12126 [cs.CV]

[8] Mehdi Mirza and Simon Osindero. 2014. Conditional Generative Adversarial Nets. arXiv:1411.1784 [cs.LG]

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. arXiv:1312.5602 [cs.LG]

[10] Sergiu Oprea, Pablo Martinez-Gonzalez, Alberto Garcia-Garcia, John Alejandro Castro-Vargas, Sergio Orts-Escolano, Jose Garcia-Rodriguez, and Antonis Argyros. 2020. A Review on Deep Learning Techniques for Video Prediction. arXiv:2004.05214 [cs.CV]

[11] Christoper JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine Learning* 8 (May 1992), 279–292. Issue 3.

A CODE RUNNING INSTRUCTIONS

The code for this project has several scripts, so I believe it is important to clearly explain exactly what each one does and how to use them. I'll do this in list format.

(1) `dqn_training.py`: This script is used to train the Deep Q-Learning agent for the game of Pong. This script will print the Total Reward at each episode it simulates, and save the current model of the DQN every 50 episodes. This file depends on the `dqn_network.py` file to provide it with the DQN architecture and `atari_wrappers.py` to provide it with the used wrappers.

(2) `dqn_test.py`: This script is used to see the best Deep Q-Learning agent trained during this project in action. This file depends on `dqn_network.py` and `atari_wrappers.py` for the same reason as above. It also depends on `dqn_best` in order to use the best agent.

(3) `data_generation`: This script is used to produce the data that is used to train the GAN. It depends on `dqn_network.py` and `atari_wrappers.py` for the same reasons as above, and depends on `dqn_exploratory` in order to have a trained agent to use. It will store data files in the Data directory. It requires the existence of the Data directory to function.

(4) `generator.py`: This file is used to train our Generator architecture using a supervised learning approach based on data produced by the previous script. It requires the 60 data files to be present in the Data directory. The code produces sample input, output, and target images, along with back ups of the generator model, every 5 epochs.

(5) `generator_test.py`: This file is used to test the trained Generator model, producing example images. In the images produced, the first three images represent input images, the fourth image represents the generated image, and the fifth image represents the target image. The action is not displayed. This file depends on the existing of the `Data-1.npy` file in the Testing directory, as well as the `pretrained_G.mdl` file. A key will need to be pressed to continuously see new images.

(6) `discriminator.py`: This file is used to train our Discriminator architecture using a supervised learning approach based on data that has been produced. It requires the existance of the `pretrained_G.mdl` file in order to have a trained generator to produce fake images with, and it requires the 60 data files in the Data directory. Similar to the generator file, this will save a back up of the model every five epochs.

(7) `game_gan.py`: This file is used to train the GAN model. It will frequently print the current average Generator loss and the current average Discriminator loss. This file requires the

`pretrained_G.mdl` file to start the generator off with pre-trained weights, and requires the 60 data files in the Data directory to use for training. Similar to the generator, this code will also produce sample images and back ups of the two models every 5 epochs.

(8) `gan_test.py`: This file uses the exact same code that is used by `generator_test.py` with the exception that the generator that is loaded is one that was trained using the GAN. This file requires the `gan_generator_best.mdl` file. The images produced are in the same format as `generator_test.py`, and key presses again allow new images to be seen.