Scene Prediction in Reinforcement Learning using Generative Adversarial Networks

Carter Blum

An Experiment

1 Abstract

Reinforcement Learning is a promising area of research that is capable of solving many problems in difficult domains. However, one of its largest problems that it currently has is its lack of sample efficiency. Agents can take millions of episodes to train fully, which is not feasible in many scenarios. One solution to this is to also learn a model, which can create an internal simulation of the environment. This can help reduce the number of samples needed, and can event reduce the computational requirements, but it has several weaknesses. Crucially, current models take a large amount of time to make predictions that are far in the future, which can severely limit the model's capabilities. What's more, the state of the art model quickly becomes inaccurate for long-range predictions. This paper proposes using a conditional generative adversarial network (GAN) to model these dependencies in cosntant time. To make computation more feasible, the GAN operates in a latent space that is tied to the agent. We show that this formulation is equivalently usable by the agent, and achieves competitive results on common Atari environments. A novel metric is suggested to compare this model with the state of the art, as this model's predictions are in a latent space, while the state-of-the-art makes predictions on the input space. Both models are evaluated using this metric, and significantly outperform baseline measures while reaching similar levels of accuracy by the end of training.

2 Introduction

Reinforcement Learning is an exciting area of machine learning and AI that is capable of solving a wide variety of tasks in different domains. By using rewards to teach an agent to act in an environment, a model can be trained end-to-end with data, giving state-of-the-art results with minimal hardcoding by software engineers. Recently, reinforcement learning has created superhuman agents for games that were previously intractable, including Dota 2 and the famous game of Go. However, these achievements were largely the result of incredible amounts of computing power. Modern reinforcement learning methods are difficult to apply to real-world tasks because of the massive amounts of data and computational resources that are needed to train them. Current methods are not very sample-efficient and difficult to train, leading to the machine-learning mantra: "If you don't have to use reinforcement learning, don't".

There are many methods that attempt to make reinforcement learning methods more sample-efficient. Deep models are one method for improving the performance of agents by simulating data from past experiences. However, simulating long-range dependencies with these models is computationally expensive and prone to error.

Generative Adversarial Networks (GANs) present a new opportunity for modeling, as they are able to generate samples from a distribution learned from data. Conditional GANs allow for GANs to model conditional distributions. GANs represent a massive leap forward in generative models, and may have application in generating possible future states from a current state.

This project shows a novel method for using GANs to make high quality predictions about future states in a method that takes constant time, regardless of how far in the future the prediction is. By predicting a latent space representation of the state space, the GAN is made much more lightweight and easy to train, without losing any interoperability with the agent.

3 Related Work

3.1 Reinforcement Learning

Reinforcement learning is a field in machine learning and artificial intelligence, concerned with teaching an agent to complete a task via a reward function, which the agent seeks to maximize. The problem differs from supervised learning, because the agent's actions affect the rewards it gets and the information it obtains. For instance, consider an agent attempting to make money. If the agent can play a violin well, building a network of relationships with famous orchestral members may be highly advantageous. However, if the agent has no musical skills, networking with musicians becomes much profitable. In this scenario, the agent's properties had an effect on the value of the action. In contrast, the correct answer in supervised learning is static - a class in classification problems or a value in regression problems [27] [23].

Until recently, reinforcement learning algorithms have struggled to solve problems in non-trivial domains. However, when Mnih et al. showed it was possible to use raw pixels to train an agent to play Atari games at super-human levels, the field was reinvigorated [19]. The method used deep learning to approximate the optimal policy [20]. Deep learning has recently seen a resurgence due to large increases in available processing power and data, as well as the use of GPUs to parallelize and speed up the relevant computations [16] . Companies and institutions like Google and OpenAI have made headlines by using similar techniques to create human-level bots for incredibly complex games, such as Go [25], Starcraft II, and Dota 2 [21]. However, these projects require incredible amounts of computational resources, which are not available to most researchers and are not practical for testing. As a result, most techniques use a set of common baseline environments to obtain empirical results.

3.1.1 Common Baselines

There are a variety of environment classes that are commonly reported as baselines in state-of-the-art papers. One of the most popular environments is OpenAI's Gym. Gym offers easy-to-use interfaces with many Atari games, as well as other environments. These games are relatively light to compute but difficult to learn, making them ideal benchmarks [18] [1].

CoMoCo Lab presents another useful way to test agents. Designed as physics engine by the University of Washington, CoMoCo Lab is often used for simulating tasks such as learning to walk or climb. Reinforcement learning has had a strong history of success in these environments, which can be incredibly varied. Perhaps the most recent addition is Deepmind's Lab, which consists of 30 levels for training and testing agents. Despite CoMoCo and Deepmind Labs presenting interesting challenges, this project will focus on solving Atari games. See the experiments section for further details.

3.1.2 Models

Across all environments, reinforcement learning has run in to several severe problems that inhibit it from succeeding in real-world environments. Agents take extraordinarily large amounts of data and computing power to train - typically on the order of millions of timesteps or higher. In these simulated environments, obtaining large amounts of data is typically not a problem, but it is often difficult to create environments for learning in practice.

One historical way to attempt to address this problem is through the use of models. With a model, the agent learns to internally simulate the environment as it interacts with the outside world [7]. With the rise of deep learning, this method has recently been updated to use a neural network to model the environment [14]. Kaiser et al.'s work uses a modified convolutional neural network (CNN) to model it's environment by predicting the next frame from the current frame. A CNN is used to extract high-information features from recent input frames. These features, along with information about the agent's move at that state, are then fed in to deconvolutional layers. The output is interpreted as the change between the current frame and the next, and is added to the input frame. The network is trained to minimize the clipped mean squared error between the pixels of the predicted next frame and the true next frame. The model can then be used to model longer time dependencies by predicting further observations based on its own predictions. For instance, assume that the model predicts the lightning will light up the sky in 1 second. To obtain a prediction for 2 seconds in the future, one can treat the predictions so far as fact and ask the model 'what will

happen 1 second after the sky lights up with lightning' (to which the model predicts the sound of thunder). To obtain a prediction for 3 seconds in the future, you simply ask the model 'what will happen 1 second after we hear thunder?'. In this way, you can continue using the model's outputs as its inputs to model dependencies further and further in to the future.

Unfortunately, to predict t seconds in the future, the model needs to make t predictions, which can be costly. For instance, a human might hear the thunder and naturally assume that it will soon begin raining if it is not already, without having to think about when they will hear thunder. Similarly, because the model always follows a particular realization of the possible chains of events, it may predict make increasingly unlikely predictions as it predicts further in to the future.

Another method for making reinforcement learning more sample-efficient is known as hindsight-experience replay. After each failure, the formulates a new reward function that would its actions would have gained rewards for, and uses this to improve its future iterations. For example, if an agent were learning to shoot a hockey puck at a goal and accidentally shot the puck three feet to the left of the goal, it can learn from what would have happened if the goal were three feet to the left (score!). It then uses this information to try to better adjust its next shot at the real goal. This method has been shown to be effective at speeding up learning in sparse reward spaces, such as the above example, where the agent receives a reward if and only if it makes a goal. However, it is not well-studied in other environments and requires a method for adapting its goal function after failures, which can be difficult to implement. As a result, it is not a general-purpose substitute for a model.

3.2 Generative Adversarial Networks

Neural Networks are a common method for modeling large amounts of data with complex dependencies. These models are effectively stacked layers that each map their input to a latent space, that hopefully represents the important information for the model. The most basic type of layer is a dense layer, which is simply a linear projection followed by a simple non-linearity, which is applied pointwise. Some popular non-linearities are the sigmoid, tanh and (leaky) ReLU functions. The parameters of the linear projection are usually fitted by a variant of stochastic gradient descent.

3.2.1 Convolutional Neural Networks

When handling image data, a convolutional neural network is usually used to reduce the number of parameters that need to be fitted. A convolutional kernel is applied in a sliding-window fashion to the data to extract meaningful features that exploit the spatial organization of the data. For example, a low-level kernel may act as an edge detector, and will output a 2D array showing where vertical edges are found in the picture. By having a stride greater than 1, this process can also reduce the height and width of the image. This process can easily be inverted, resulting in deconvolutional layers that can be used to map from lower-dimensional spaces to higher-dimensional spaces (e.g. increase the number of pixels in an image) [9].

3.3 Generative Models

One common application of neural networks is approximate distributional sampling via generative models. Given a dataset that is assumed to be generated by sampling from some prior distribution, neural network architectures can be devised to approximate samples from the prior distributions given the observed data. One classical way of achieving this was through the usage of variational autoencoders(VAEs) [15]. Variational autoencoders are neural networks that attempt to reconstruct their own inputs, but have heavily reduced dimensionality in some layers (see figure 1). This forces the network to attempt to represent the observed data in a latent space of heavily reduced dimensionality. Samples are drawn from the latent space (represented in the middle), and fed through the second half of the VAE to obtain samples from the target distribution. This project uses some ideas from a VAE, using layers trained for one purpose in a related way.

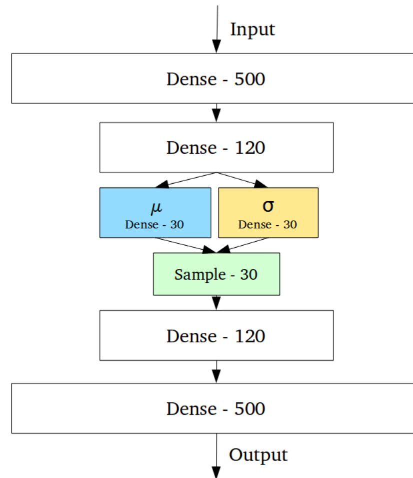

Figure 1: Diagram of layout of a generic VAE

While these methods provide reasonable results, more recently, generative adversarial networks (GANs) have been used to achieve similar goals with great effect [10][22]. In the set-up of GANs, a generator is fed noise and its output is interpreted as a draw from the target distribution. It maps the low-dimensional noise to high-dimensional spaces such as images by using layers that have more output dimensions than input dimensions (and, in the specific case of images, deconvolutional layers are used) [22]. A separate network, called the discriminator, takes in a datapoint x, generated either from the (true) training set, or the (fake) outputs of the generator. The discriminator is trained to predict the probability that the datapoint came from the training set versus from the generator. For instance, if the goal is to generate pictures of animals, the discriminator would be trained to recognize 'is this a real picture of an animal?'. The discriminator's loss is then the Kullback-Leibler Divergence between the true label distribution and the target distribution, while the generator's loss is the negative of the discriminator's loss (see figure 2 for reference). In this way, the discriminator and the generator play a game against each other, with the generator trying to learn to fool the discriminator, and the discriminator trying to get better at not being fooled. While the theoretical underpinnings for this method are not well-understood, it has achieved many excellent results.

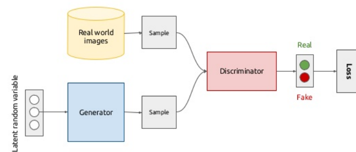

Figure 2: Basic Diagram of Generative Adversarial Network Organization

3.3.1 Conditional GAN

The method can be extended to generate samples from conditional distributions as well. In the conditional formulation, when the generator is given noise from which to generate a sample, it is also given additional information representing the information to be conditioned upon [17] . For example, the generator might receive 10 dimensions of Gaussian noise, as well as 2 dimensions, indicating whether the generated picture should be of a cat or a dog. The discriminator is also fed the additional information at decision time. So, instead of asking the discriminator 'is this a picture of an animal?', the algorithm would sometimes be asked 'is this a picture of a cat?' and other times, it would be asked 'is this a picture of a dog?'. This formulation has had success in generating distributions on command, with applications generating pictures

of faces[2] and in style transfer of artwork [13]. These methods allow for modeling random function draws from a wide domain of distributions, making GANs an interesting technique for modeling reinforcement learning environments.

4 Methods

This project relies on training a model to encode information about and act within the environment. One popular method for training an agent to maximize its reward in a reinforcement learning environment is Deep Q-learning.

4.1 Deep Q-Learning

We assume that the environment consists of a real-valued, finite reward function R that is a function of the current state s and the action that the agent takes, a. Similarly, we assign a transition probability $P(s'|s,a)$ that represents the probability we are in state s' immediately following executing action a in state s. Let π be an agent such that $\pi(a|s)$ represents the probability of the agent choosing action a in state s. If we intend to maximize our total reward, then we can define the value of a state to be

$$V(s) := \sum_{a \in A(s)} \pi(a|s) \mathbb{E}_{r_t \sim R(s,a), s' \sim P(s'|s,a)} \Big[r_t + \gamma V(s') \Big]$$

Where $0 < \gamma < 1$ is a term included to ensure that the value function always converges. This represents the sum of all of the rewards we can expect to receive after landing in state s, with rewards t timesteps in the future discounted by a factor γ^t. If we decide on an action a, we can similarly define the utility of that action using a Q-value:

$$Q(s,a) := \mathbb{E}_{r_t \sim R(s,a), s' \sim P(s'|s,a)} \Big[r_t + \gamma V(s') \Big]$$

If an agent were to behave optimally to obtain the maximum rewards, it would always choose the action with the highest Q-value with probability 1 [27][23]. As a result, learning the Q function is sufficient to learning the optimal policy in a given environment. However, storing these values in a table takes $|S| \times |\mathcal{A}|$ entries, which is intractable in most environments. Instead, a neural network is often used to approximate the Q function. After each observation, the network's loss is proportional to

$$r_t + \gamma \max_{a' \in \mathcal{A}(s)} \Big[Q(s_{t+1}, a') \Big] - Q(s,a) \tag{1}$$

, which is the error between the predicted and observed reward. Because of the presence of γ, we can show that this function is a contraction, and will always converge to a fixed point.

Notably, this process is sped-up using the bootstrapped predictions for $Q(s_{t+1}, a')$. However, training a network's losses based on its own predictions can be unstable, so we typically use a target network, denoted Q^-, to approximate Q-values for subsequent states. This substantially improves the stability and speed of the convergence to the correct values. In practice, the target network is typically an older version of the main Q-network, but it isn't updated as frequently [20].

4.1.1 Double Q Network

However, this formulation can still cause issues. If we rewrite equation 1 with the target network as $r_t + \gamma Q^- \Big(s_{t+1}, argmax_{a' \in \mathcal{A}(s_{t+1})} \Big[Q^-(s_{t+1}, a') \Big] \Big) - Q(s,a)$, we can see that Q^- is being used twice, selecting the maximum among its own predictions.

This can lead to the overestimation of the value of certain states, which can substantially slow learning. Because of this phenomenon, some actions may not be explored sufficiently and the Q-values will have slower convergence to their fixed points. The simple fix is to separate the approximate used to choose the action a' and the one used to evaluate a'. Luckily, we already have 2 different estimates: Q and Q^-. As a result, we

make a minor edit to the equation to improve stability. Our loss function is then :

$$\left(r_t + \gamma Q^- \left(s_{t+1}, argmax_{a' \in \mathcal{A}(s_{t+1})} \Big[Q(s_{t+1}, a') \Big] \right) - Q(s,a) \right)^2 \tag{2}$$

This formulation, using two Q-networks, is known as a double deep q-network (DDQN) [29].

4.1.2 (Prioritized) Experience Replay

Even with the DDQN formulation to help stabilize training, deep q-networks may still not reach stable estimates if naively trained on data as it is observed. Let w_t represent the weights of the network at epoch t and $(\Delta w)_t$ represent the gradients of the network at that epoch. In a network with a constant weight decay $\beta \in (0,1)$ and learning rate $\alpha \in (0,1)$, the new weights will be $w_{t+1} = \alpha(\Delta w)_t + \beta w_t$. We can then define w_{t+1} as a weighted sum of previous updates [28] :

$$w_{t+1} = \sum_{j=0}^{t} \alpha \beta^{t-j} (\Delta_w)_j \tag{3}$$

As a result, the network can only learn a the average of a set of values if each batch of values is selected from the same distribution. Otherwise, it will be biased towards fitting the most recent observations[19]. However, observations in reinforcement learning are strongly autocorrelated. The observation at time t is typically not very different from the observation 1 nanosecond later, so the agent may overfit its current observations and 'forget' what it has seen before [24].

The solution to this problem is to store recent observations at memory, and update the network by randomly selecting observations from memory, 'replaying' those experiences. This helps keep the distribution of observations more stable, and allows the network to learn low-probability events and better handle environments with longer episode times. To improve sample-efficiency, we prioritize replaying experiences where our Q-value estimates were most incorrect. Intuitively, humans do something similar. You never lie awake in bed thinking about how well you did at walking today, because it's something that you understand well. Instead, your brain focuses on the moments in the day that were most surprising, as there is potentially more to learn from them. While this method is a near-strict upgrade on performance, it can consume a lot of memory, because large amounts of previous frames need to be stored. As a compromise, many algorithms (including the implementation used in this project), only store the most recent couple-thousand of frames, discarding old memories entirely.

4.1.3 ϵ-greedy exploration

Because the environment is inherently stochastic, the agent must collect many samples at each state to get an accurate estimate of its value. Unfortunately, it is computationally infeasible to explore ever state equally in large environments. As a result, an exploration policy is needed. One option is an ϵ-greedy approach, where it always takes the action with the highest Q-value, except that it takes a random action with probability ϵ.

There are many other exploration schemes, such as Thompson Sampling [5], UCB-methods [4] and exploration bonuses [25]. However, to keep the implementation and interactions with the prediction model simple, this project uses a simple ϵ-greedy approach. ϵ is steadily decreased as the network is trained, allowing it to approach a more optimal solution (because taking random actions is not optimal). However, the floor for ϵ is 0.01, so that the algorithm never stops exploring and does not get stuck.

4.1.4 Dueling Network

Another method for improving the performance of a Deep-Q Network is through the use of a dueling network. Because in many states, estimating the value of the state is more important than accurately modeling the Q-values, the dueling network architecture splits the two functionalities [30]. The network bifurcates in to two heads, one predicting the value of the state and the other predicting the advantage of each action. The advantage $A(s,a)$ of the action is the advantage the agent gains by taking action a in

state s over the expected value of the state. The Q-values can then be reconstructed from the value and the advantages.

$$Q(s,a) = V(s) + A(s,a) - \frac{1}{|\mathcal{A}(s)|} \sum_{a' \in \mathcal{A}(s)} A(s,a')$$

While this increases the computational requirements of running a single pass of the network, it does substantially decrease the training time.

4.2 Prediction

As discussed earlier, predicting future states can be very useful for increasing sample efficiency and training speed. As seen in Kaiser et al.'s work, this has traditionally been done by predicting the full observable state. In the case of Atari games, this amounts to generating a new RGB image. However, much of the information in the image is superfluous or redundant and predicting it can be difficult. In a simple case, if you asked someone what would happen if you jumped in Mario, they would respond that you everything else would probably remain unchanged, but Mario would be launched in to the air. However, if you asked them to draw what that looks like, the drawing you would receive would likely be very different from the actual screenshot of what happens. One might naturally object that one's ability to draw frames of Mario is irrelevant to their ability to understand the mechanics of the game. The description that they initially responded could be viewed as the latent space representation of the most important information of the state.

4.2.1 Latent Space Representation

Because the output of each layer of a neural network relies only on the output of the previous layer, we can represent it as a composition of functions. For a given agent π, we can write $\pi = \pi_N \circ \pi_{N-1} \circ \ldots \circ \pi_1$. In fact, we can interpret the sequential application of any number of these layers as a mapping to and from latent spaces that represent the important information in condensed form. Formally, we can define $\pi_A = \pi_N \circ \pi_{N-1} \circ \ldots \circ \pi_{K+1}$, $\pi_L = \pi_K \circ \pi_{K-1} \circ \ldots \circ \pi_1$. Similarly, $\pi = \pi_A \circ \pi_L$. Assuming the inputs are in some space $\mathcal{I}$, then $\pi_L : \mathcal{I} \to \mathcal{L}$ maps the input space to an information-dense latent space $\mathcal{L}$, and $\pi_A : \mathcal{L} \to \mathcal{A}$ maps this latent space to Q-values for each of the actions. This latent space $\mathcal{L}$ has the benefit that it has all of the information that the agent will need for decision making, and, as the agent continues training, will become increasingly information-dense, with less noise. As a result, making predictions in this space, with reduced dimensionality, is just as useful to the agent as making predictions in the original space.

4.2.2 Distributional Perspective

In a stochastic environment, making a prediction about a future state is equivalent to making generating samples from a random function. Consider a state s'. In predicting the future, we should predict less likely futures less often and relatively likely futures are something that the agent should suggest. The probability of state s' occurring k frames from now, given that we are at state s is given by $T(s'|s,k)$. We define it as:

$$T(s'|s,k) = \sum_{s^*} T(s^*|s,k-1) \sum_{a \in \mathcal{A}(s^*)} \pi(a|s) P(s'|s^*,a)$$

Where $T(s'|s,0)$ is 1 if $s' = s$ and 0 otherwise. As a result, we have a function for the probability distribution of states k timesteps from now, and making predictions is equivalent to drawing samples from this target distribution. Because we can define $P(s'|s,a)$ even when the state is not fully observable, this formulation works equivalently for a partially-observable markov decision process (POMDP).

4.2.3 GAN Predictions

Because we have reframed our problem as sampling from a conditional distribution, we can use a GAN to predict future spaces. If we condition on our current state s and the desired time delta k, we can use a conditional GAN to approximate function draws from $T(s'|s,k)$. We simply provide s and k as inputs to the discriminator and the generator, and we can train on examples generated by agent trajectories. One item that

is particularly of note, is that this formulation works equally well even if the underlying process is partially observable. As a result, even if the representation of s provided does not include all relevant information, this model will still attempt to make the best possible prediction with the information available. Therefore, even if the latent space does not successfully encode all relevant information, the information encoded can still be used to make predictions about the future state space.

5 Experimental Design

5.1 Experiment Design

5.1.1 Environments

The model was tested in OpenAI's gym environment, using some popular Atari environments - MsPacman, Breakout and Asteroids [6]. MsPacman is a game similar to the original Pacman, in which the agent attempts to collect as many dots as possible while running away from ghosts. Breakout, also known as Brickbreaker, is a game where the agent tries to bounce a ball in to static bricks on screen in a similar manner to pong. Asteroids is a game in which the agent moves a small spaceship and attempts to avoid incoming asteroids in order to stay alive. Experiments were conducted in each of these environments, using the same design for each game for consistency.

5.2 Agent Design

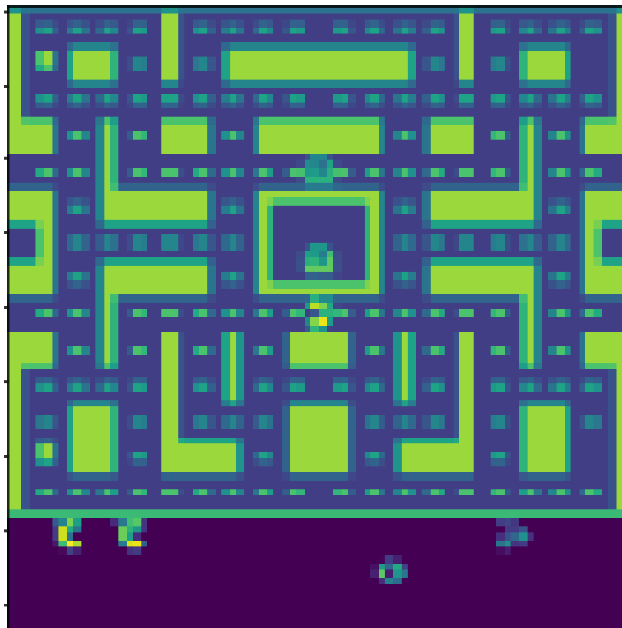

Figure 3: Sample picture of state from MsPacman after preprocessing. Shown with colors for increased contrast.

In order to reduce the dimensionality of the input space and reduce computation time, some preprocessing was included (see above figure for example). The images of the state space were downsampled to 84x84 and converted to grayscale before being input to the network. The DDQN consisted of three convolutional layers with 5x5 kernels and 64 features, followed by 2 dense layers with 256 nodes each. From here, the DDQN splits into two heads, one to predict the value function and the other to predict advantage. Each head has two layers: an output layer and a hidden layer with 256 nodes. The other details of the network are as described in the methods section. The DDQN was trained with the loss function described in methods

using RMSProp, trained for a total of twenty million timesteps a piece. ReLU is the non-linearity for each layer.

5.2.1 Latent Space Representation

The first four layers of the network (plus preprocessing) are used to encode the state in to the latent space. This was chosen as a middle point, because it has significantly reduced dimensionality (256 dimensions versus the 7,000+ dimensions in the input), but it is still detecting relatively low-level details that are more likely to be informative about the environment than they are about what action to take. This encoding is never trained independently, and is simply taken from the the model after training. It is not trained while the generator and discriminator are training. These layers are equivalent to π_L described in methods. Because π_L is an intermediate layer in the agent (π), the final layer has ReLU as an activation function. The rest of the network is denoted as π_a.

5.3 Model Design

5.3.1 Generator

The generator consists of two dense layers, with 256 hidden nodes. It receives the latent space representation of the state space concatenated with noise as input. It outputs a prediction in the latent space, which has 256 dimensions. All nonlinearities used are ReLU. Because ReLU is applied in the final layer of π_L, it is applied in the output layer here as well. No activation functions are present in the final layer of any other networks.

5.3.2 Discriminator

Similarly, the discriminator takes in the initial state (s) and the future state (s') and predicts whether or not the future state is generated from a real distribution. Both inputs are first projected in to the latent space using π_L, so the two states together mean it has 512 dimensions as input. The network has 2 layers, with 256 hidden nodes and ReLU as an activation function on all layers except the output.

5.4 Existing Algorithms

5.4.1 Baseline

To check that the methods in this project are non-trivially predicting the future, a trivial baseline is included for comparison. The baseline is simply the identity function. As a result, it predicts that the future state is the same as the current state, forecasting that there is no change whatsoever. This baseline provides a simple sanity check and helps visualize how long it takes for the networks to achieve predictive capabilities.

5.4.2 Iterative Predictor

This project also implements as lightweight version of the state-of-the-art model as comparison. In Kaiser et al.'s work, they train an iterative model to predict the next frame based on the previous four frames and the agent's next action. Because the GAN model proposed in this paper has no information about previous states or the actions of the agent, this information is removed from this model as well. The model takes a preprocessed image as input and applies 2 convolutional layers to it, each with a 5 by 5 kernel and stride of 2. After a single dense layer with 441 nodes, 2 deconvolutional layers are applied to get an output of the same shape as the input. This output is interpreted as the change between the two states (so no change would be represented by a mask of all 0's). To obtain a prediction for the new state, it is added to the input. The loss function is as described in Kaiser et al., using the mean-squared error between the predicted pixels and the actual observation. To prevent the loss from growing to large, the loss is clipped at 10. To obtain predictions k timesteps in the future, the model makes k successive predictions based on its own outputs. For more details, see the description in 'Related Work'.

5.5 Testing

For each environment, the agent was trained for twenty million timesteps. After training, the parameters on all layers were frozen and the first four layers were taken as π_L. The network was then used to sample trajectories from the environment. From these trajectories, random pairs of frames were selected to form the training and test set, which each had 5,000 pairs. For simplicity, all pairs were separated by 10 timesteps. Future work will expand this to work with multiple time intervals. Both the GAN and the iterative predictor were trained for 5,000 epochs, with a batch size of 32 for each epoch. After each epoch, the models were evaluated on the test set, and the results are reported down below.

5.5.1 Q-Value Reconstruction Error

Because the GAN outputs predictions in the latent space while the iterative model outputs predictions in the original image space, it is difficult to compare the quality of the results from the two models. To attempt to address this, this paper uses a novel metric to assess how useful the predictions are for decision making. For each prediction, we obtain Q-value estimates from the state output by the prediction method. We report the mean-squared error between the Q-value estimates given by the model on the prediction versus the real observation. Intuitively, this is a measure of how similar the agent thinks the two states are from a decision making standpoint. Two states that are functionally identical will have the same Q-values for each action. Let $G(\pi_L(i))$ be the (latent space) prediction given by the GAN after observation i and $P(i)$ be the prediction given by the iterative model. Let i_{t+10} be the ground truth future state.

We then define the GAN loss : $\sum_{a\in\mathcal{A}(s)}\left(\pi_A(G(\pi_L(i))) - \pi(i_{t+10})\right)^2$ And the iterative predictor loss : $\sum_{a\in\mathcal{A}(s)}\left(\pi(P(i)) - \pi(i_{t+10})\right)^2$ The results using this loss are examined in closer detail below.

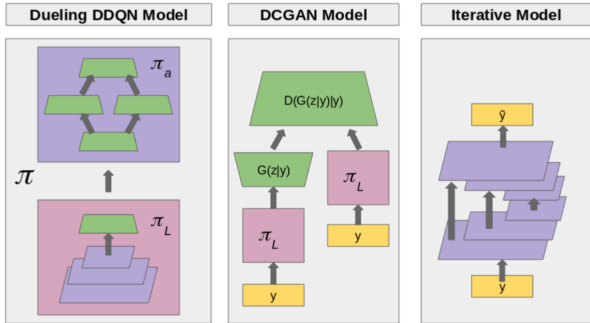

Figure 4: Diagram of models. DQN shown on the left, divided in to π_A and π_L. In the center is the GAN, with the preprocessing steps shown, courtesy of π_L. On the right is a simple diagram of a single pass of hte iterative model. The forward arrows indicate data flow in all models.

6 Results, Analysis

6.1 MsPacman

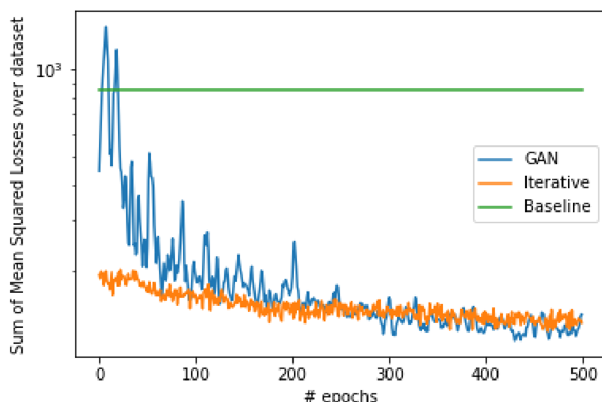

Figure 5: Plot of the mean-squared error for MsPacman

As seen in figure 5, the GAN takes significantly longer to train than the default predictor model. In fact, the iterative model appears to be reasonably well-trained after a single epoch, which is likely an artifact of its architecture. Because the architecture for the iterative includes a bypass layer, it is very easy for the model to learn an identity function and improve on that. For contrast, it takes the generative model a while to even learn the identity function.

Nonetheless, both models outperform the baseline by orders of magnitude, which confirms that both are valid models for this time span. However, one does have to note that it takes the generative model significantly longer to converge than the iterative model. Regardless, both models seem effective for the environment of MsPacman. This is a relatively easy environment to predict, as MsPacman tends to keep moving in the direction she is facing, making linear prediction possible.

6.2 Breakout

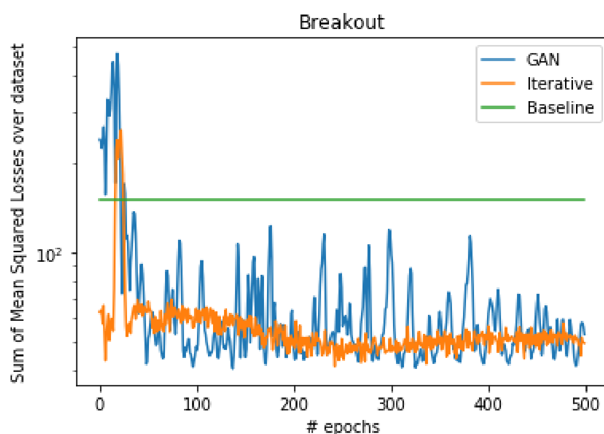

Figure 6: Plot of the mean-squared error for Breakout

Looking at 6, there are a couple interesting things to note. The first is that both models spike in error rate after about 20 epochs, which I do not have a strong explanation for. Additionally, it looks as if the iterative model begins overfitting after 250 epochs, as its test error begins rising after that. When

implemented side-by-side with a agent that is learning as the iterative model is, this wouldn't be a problem, as there would be effectively infinite data for the model to train off of.

One concerning element is that the GAN's losses continues oscillating wildly. This likely indicates nothing more than that the learning rate is too high, but it is certainly worth looking in to. As an environment, Breakout is moderately difficult to predict from a single frame. It's impossible to infer which direction the ball is going, which significantly harms prediction. However, the Q-values are likely similar when it is going in either direction, as the correct move is to move towards the ball no matter what. The fact that the loss function used in this project doesn't distinguish between these scenarios is potentially a weakness of the system.

6.3 Asteroids

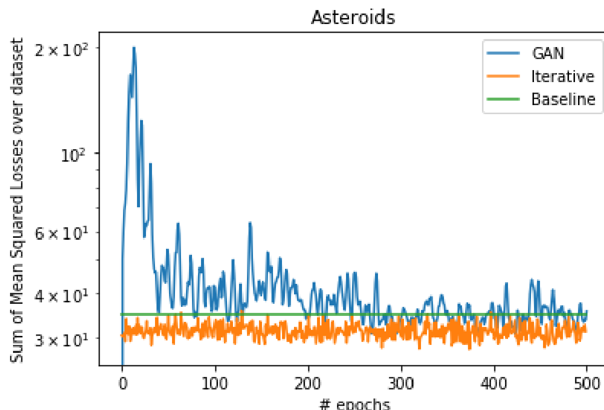

Figure 7: Plot of the mean-squared error for Asteroids

The results for Asteroids are difficult to interpret. Because the asteroids move vertically at a slow rate and the agent moves horizontally, there isn't much of a difference between the current state and 10 timesteps from now. If the agent is currently under an asteroid, it wants to move to either side. Otherwise, it likely has a very high value for its current state and all moves are roughly equal. As a result, the baseline performs very well on this dataset, and is surprisingly difficult to beat. Only after it's done training does the GAN beat it. Similarly, the iterative model trains extremely quickly by learning little more than the identity function.

6.4 Takeaways

In all of the examples shown, the generative model takes significantly longer to train than the iterative approach. To some extent, this is to be expected, as the iterative approach easily learns the identity function. Because 10 timesteps (approximately 0.25 seconds) is a short duration, the identity function is a pretty decent predictor.

However, the GAN model does show that it has the capacity to perform at the same level as the iterative model, which gives it credence as a model. It would be interesting to see whether or not the difference in training times matters when the models are trained alongside an agent in real time. It is possible that the GAN would learn fast enough to keep up, making the difference in training time negligible.

7 Conclusion

The GAN model proves to be a competitive approach to the existing state of the art prediction methods. Although it takes longer to train, it provides similar accuracy on the test metric, and significantly outperforms the trivial baseline (something that can unfortunately not always be taken for given in reinforcement learning [26]).

That said, there are numerous improvements that can be made to the current model. Optimally, the model would train at the same time as the agent. For simplicity, this was not done in this realization of the project. Additionally, the model needs to be tested on a larger scale with more complex environments and a variety of timespans. More metrics will likely be useful for further analyzing the performance of these methods.

Finally, there are a number of structural improvements that can be made. Wasserstein GANs typically provide better performance than vanilla GANs [3][11] and there are other improvements that can be made to improve the performance of DQNs ([12] [8]). This project presents an exciting and promising direction of research, but there is still much more work to be done in understanding its capabilities!

References

[1] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, pages 5048–5058, 2017.

[2] Grigory Antipov, Moez Baccouche, and Jean-Luc Dugelay. Face aging with conditional generative adversarial networks. In *2017 IEEE International Conference on Image Processing (ICIP)*, pages 2089–2093. IEEE, 2017.

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

[4] Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3(Nov):397–422, 2002.

[5] Olivier Chapelle and Lihong Li. An empirical evaluation of thompson sampling. In *Advances in neural information processing systems*, pages 2249–2257, 2011.

[6] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines, 2017.

[7] Kenji Doya, Kazuyuki Samejima, Ken-ichi Katagiri, and Mitsuo Kawato. Multiple model-based reinforcement learning. *Neural computation*, 14(6):1347–1369, 2002.

[8] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561*, 2018.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[10] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[11] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of wasserstein gans. In *Advances in Neural Information Processing Systems*, pages 5767–5777, 2017.

[12] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[13] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. In *European conference on computer vision*, pages 694–711. Springer, 2016.

[14] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, et al. Model-based reinforcement learning for atari. *arXiv preprint arXiv:1903.00374*, 2019.

[15] DP Kingma and M Welling. Auto-encoding variational bayes. 2014.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[17] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*, 2014.

[18] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[19] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[21] OpenAI. Openai five. https://blog.openai.com/openai-five/, 2018.

[22] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[23] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach.* Malaysia; Pearson Education Limited,, 2016.

[24] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[25] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[26] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

[27] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[28] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[29] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[30] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1995–2003, 2016.