university of groningen / faculty of science and engineering

Dreaming of Reinforcement Learning: GAN-enhanced Experience-Replay

Bachelor's Project Thesis

Alexandru Turcu, s3972445, a.turcu.1@student.rug.nl,
Supervisor: Dr. Matthia Sabatelli

Abstract: The last decade has seen an extraordinary rise in popularity of Deep Reinforcement Learning (DRL) techniques, due to their capacity of attaining super-human results in most types of control tasks. However, they all suffer from the substantial drawback of long training times. Past studies have tried to minimize training time by improving the trajectories of the Experience-Replay (ER) memory buffer. This thesis explores the possibility of improving the convergence time of the Deep Q-Learning algorithm by initializing its ER memory buffer with GAN-generated trajectories. The Deep Q-Network (DQN) agents are tested on the Atari 2600 game of Pong, in which the states to be generated are represented by frames of the game. Results show that the DQN agent corresponding to the initialized memory buffer does not exhibit a significantly faster convergence time to the same reward as the non-initialized agent.

1 Introduction

Reinforcement Learning (RL) is a branch of Machine Learning which concerns itself with learning from experience, rather than by correction from labeled or unlabeled data. Learning from experience translates to learning a value function that the agent can use to navigate through the state-action space of the task. There are two types of value functions: the state value function $V(s)$, which approximates the value of the agent being in state s and the state-action value function $Q(s, a)$, which indicates the value of the agent taking action a while in state s (Sutton & Barto, 2018). In model-free RL, these functions are approximated with algorithms like Q-Learning (Watkins & Dayan, 1992) and SARSA (Rummery & Niranjan, 1994). The main disadvantage with these type of algorithms is that they work in a tabular manner and most tasks that are worth researching are characterized by large state-action spaces that cannot effectively be solved by these methods alone. In order to overcome this drawback, artificial neural networks are used in their capacity as universal function approximators and feature extractors. In Deep Reinforcement Learning (DRL), classic RL algorithms are combined with non-linear function approximators, resulting "deep" algorithms in which the value function is learnt by gradient descent (LeCun et al., 2015). Even though this approach solves the state-action space problem, it also raises other issues, such as instability in training. Deep Q-Learning (Minh et al., 2015) requires multiple additions in order to be able to learn, including an Experience-Replay (ER) memory buffer (**?**), which must be filled with experiences (or trajectories). Not all trajectories are equally useful to the learning process and the good ones are encountered rarely in the beginning. This contributes to one of the most noticeable problems of the DRL algorithms, which is a very slow training time. This project attempts to come up with a method that can improve training time by modifying the ER technique. The core idea is that in order to learn faster, the agent must encounter good experiences more often. Therefore, a solution to the scarcity of the good experiences could be to generate them by means of Generative Adversarial Networks (GANs) (Goodfellow et al., 2014), instead of waiting for them to happen naturally. The focus of this thesis is to determine if initializing the ER memory buffer with GAN-generated trajectories will lead to faster learning in the Deep-Q-Network (DQN) algorithm.

1.1 Reinforcement Learning

The underlying mathematical framework for most RL problems is the Markov Decision Process (MDP) (Bellman, 1957), which can be formally defined as $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathfrak{R} \rangle$. At timestep t, the agent exists in state $s_t \in \mathcal{S}$ and is able to perform action $a_t \in \mathcal{A}$. By performing an action, the agent transitions to state s_{t+1}, according to the transition function $\mathcal{P}(s_{t+1}|s_t, a_t)$. By transitioning from a state to another, the agent receives a reward signal r_t as defined by the reward function $\mathfrak{R}(s_t, a_t, s_{t+1})$. The agent acts based on policy $\pi : \mathcal{S} \to \mathcal{A}$, which denotes the probability of taking action a while in state s:

$$\pi(a|s) = \Pr\{a_t = a|s_t = s\}, \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A} \tag{1.1}$$

The optimal policy π^* is learnt by maximizing the state-value function V or state-action value function Q, which are defined as follows:

$$V^\pi(s, a) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \middle| s_t = s, \pi\right] \tag{1.2}$$

$$Q^\pi(s, a) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \middle| s_t = s, a_t = a, \pi\right] \tag{1.3}$$

The functions represent the value or the expected cumulative discounted reward that the agent receives when reaching state s or taking action a while in state s. The term $\gamma \in (0, 1)$ is the discount factor, which, depending on its value, assigns more importance to either short or long term rewards. The state-value function V can only be used to learn the optimal policy if the transition probabilities of the Markov chain are known. As mentioned in section 2.4, this is not the case for this project. Therefore, the Q function will be chosen as the target to be maximized. The Q function that acts on the optimal policy is then defined as:

$$Q^*(s, a) = \max_\pi Q^\pi(s, a) \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A} \tag{1.4}$$

The algorithm which this study is aiming to improve is the DQN, which combines the Q-Learning algorithm with a neural network to approximate the value of the Q function from high-dimensional inputs. The states observed by the agent are in image shape, so the neural network that is appropriate to work with visual data is a convolutional neural network (CNN) (Lecun et al., 1998). A more detailed explanation of the CNN and its components can be found in section 1.3.

In the original Q-Learning algorithm, the Q function is learned in the following tabular manner:

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha[r_t + \gamma \max_{a \in A} Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{1.5}$$

Training a neural network means constantly adjusting the weights of its connections, which are encompassed under the term θ. So, using a CNN to approximate the state-action value function translates to learning the weights of the neural network by minimizing the loss of θ. This is done by performing a gradient descent on the following loss function:

$$L(\theta) = E_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)}\Big[(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-) - Q(s_t, a_t; \theta))^2\Big] \tag{1.6}$$

The DQN algorithm requires two additional components to achieve stable training: the Target-Network (TN) and the Experience-Replay memory buffer.

In classic Q-Learning exactly one state-action value is updated at each timestep. In DQN, the value of executing an action in a state is calculated with regard to executing an action in another state. Both of these terms are continuously updated, so it can be said that, in DQN, an approximation is updated with another approximation. This leads to the stability issue of catastrophic interference (McCloskey & Cohen, 1989), in which the model "forgets" old information in favour of new information. Old information might be important to the model, so blindly deleting it can cause rapid downfalls in performance. The purpose of a TN is to prevent this by being a copy of the main network, which is used to update the weights in the main network. The TN

is meant to bring stability to DQN, so its parameters do not get updated in the same way that the original network's parameters do. Its weights have fixed values for a period of time, during which they are used as a reference point or Temporal Difference (TD) (Sutton, 1988) target for the DQN. By using the TN parameterized by θ^-, the Q function is approximated by TD-Learning, which uses TD-targets that are calculated by the same Q function, according to the following equation:

$$y_t^{DQN} = r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-) \tag{1.7}$$

After a fixed period of time has elapsed, the weights of the TN are synchronized with those of the main network and the process restarts.

In sequential decision tasks the states and actions respect the Markov chain property, so trying to extract relevant information from them as they happen is not very useful, given the high correlation they have to each other. For example, state s_{t+1} only exists because state s_t happened right before it, so the action a_{t+1} can end up depending on action a_t. In order to minimize this correlation effect, ER is used. In ER, a memory buffer is a queue that contains the experiences the agent has while navigating the task space. The experience tuple is defined as $e_t = \langle s_t, a_t, r_t, s_{t+1} \rangle$. So, when the network has to adjust its weights, the ER memory buffer is uniformly sampled and the tuple is used in the calculation of the loss function in equation 1.6. The network is exposed to experiences that happened at random timesteps, thus reducing harmful correlation. Since the memory buffer is implemented as a queue, this technique also has the advantage that an experience can be sampled multiple times. An experience is only removed from the queue in time, after multiple other trajectories have been added. The classic ER method samples the trajectories uniformly, however this is not always optimal. The agent stores many trajectories in its memory buffer and not all of them are equally important for learning the task. In Prioritized Experience Replay (Schaul et al., 2015), the experiences are given a priority depending on the magnitude of their TD-error. A higher priority means that the experience will be sampled more often, since it is considered to be more informative for the learning process.

1.2 Unsupervised Learning

Although this project is mainly focused on RL, the improvement that is made to the ER memory buffer is rooted in Unsupervised Learning (UL). In UL, algorithms make use of a dataset of unlabeled datapoints. Compared to RL, where the agent learns based on the reward signal from the environment, UL models learn by guidance from an objective function. UL algorithms are usually concerned with finding patterns between observed distributions rather than predicting labels (as in Supervised Learning) or learning a policy.

The core of this study is generating fake states for the agent to learn from, so a generative model must be used. In this context, a generator is a statistical model that has the objective of capturing the joint probability distribution $P(X, Y)$, where X is the observed variable and Y is the target variable. However, in the case of UL, there is no target variable, so the generator's role becomes capturing the probability distribution $P(X)$. Since there are no target labels, the generative model requires an additional mechanism to improve its performance. In GAN, this mechanism comes in the form of another model, namely a discriminative one. A discriminator is, like the generator, a statistical model, but its objective is to capture the conditional probability $P(Y|X)$, meaning that it outputs the probability that X comes from the distribution Y.

In GAN, the generative model (G) and the discriminative model (D) are implemented as MLPs, that continuously compete against each other. The purpose of the generator is to fabricate data similar to the one from the training set, while the scope of the discriminator is to estimate whether the data came from the generator or the original dataset. Formally, their relation can be described as finding a Nash equilibrium to the following min-max problem:

$$\begin{aligned} \min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \\ + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))] \end{aligned} \tag{1.8}$$

G is the generator network, so $G(x)$ represents the data generated by the network from the noise input x. Analogously, D is the discriminator, so $D(x)$ is the probability that x is a datapoint from the original dataset. There are two distributions being sampled for input, namely p_{data} which reflects

the distribution of the real data and p_z, which represents the input noise variables.

For the generator to learn a distribution p_g, that is similar to the original distribution p_{data}, it must start from some point. That starting point is represented by the input noise p_z, which the generator tries to transform into $p_g = p_{data}$. The noise distribution starts out as a Gaussian random distribution. This also has the advantage that the resulting distribution will be varied, so the generator will output different results.

The objective of GAN is to obtain a good generator, but to do that the discriminator must also improve. This means that a balance must be obtained between the performance of the adversarial models. If one of the models becomes better than its counterpart, it may happen that the opponent is not able to ever catch-up and the results are underwhelming. Thus, gradient descent must be used in the update functions of the discriminator and the generator:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} [\log D(x^{(i)}) + \log(1 - D(G(z^{(i)})))] \quad (1.9)$$

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log(1 - D(G(z^{(i)}))) \quad (1.10)$$

In both equations m is the minibatch sample size. In equation 1.8, the generator cannot affect the term $\log D(x)$, so that is the reason it is not included. The motive for using logarithms becomes apparent when splitting the loss function in equation 1.8 into the update functions of the discriminator and the generator. By using logarithms, differentiating the functions becomes feasible.

The Deep Convolutional Generative Adversarial Network (DCGAN) (Radford et al., 2015) works in a similar manner, the only difference being the architecture of the networks, which use convolutional layers instead of fully connected ones. Since the project uses image data, the DCGAN architecture was chosen to generate the artificial states.

1.3 Convolutional Neural Network

CNNs are multilayer perceptrons (MLPs) (Rosenblatt, 1957) that use convolutional filters as their hidden layers. These filters perform the operation of convolution, which means that a kernel or a filter slides along the input image and performs multiplications on the pixels that it slides over. After multiple convolutions, the input image becomes an abstract feature map that contains the most important features of the original input. Throughout the whole process, the image is downsampled, so the output's shape is different than the input's. Even though details of the input image are lost during this process, the output contains the features that are needed for learning. Conversely, a transpose convolution takes a feature map as input and performs the operation of upsampling, such that the output is a more detailed image. Convolution and transposed convolution are often used together, because working with downsampled images is less computationally expensive, and rebuilding them from a low resolution is done by upsampling.

Dropout is another type of layer that makes using CNNs possible. The dropout layer aims to reduce the overfitting problem that is constantly present in neural networks. By applying a dropout layer, random nodes are ignored during each training pass with a probability that is specified as a parameter. By temporarily removing nodes from the network, the activations of hidden layers have the side effect of becoming sparse (Nitish et al., 2014).

An important aspect in the achitecture of a neural network is the activation function used for each layer. In this project, 3 activation functions are used: rectified linear unit (ReLU) (Fukushima, 1975), leaky rectified linear unit (Leaky ReLU) and sigmoid. Activation functions dictate the activation of a node by transforming the sum of weighted input that originates from the node. The ReLU function is defined as:

$$f(x) = \max(0, x) \quad (1.11)$$

ReLU is vulnerable to the "dying ReLU" problem, in which the weights could update in such a way that some nodes become inactive throughout the entirety of the training process. Leaky ReLU attempts to solve this issue by outputting a small positive value when $x < 0$. The value of the output when $x < 0$ is influenced by the fixed parameter α.

$$f(x) = 1(x < 0)(\alpha x) + 1(x \geq 0)(x) \quad (1.12)$$

The sigmoid function is used in neural net-

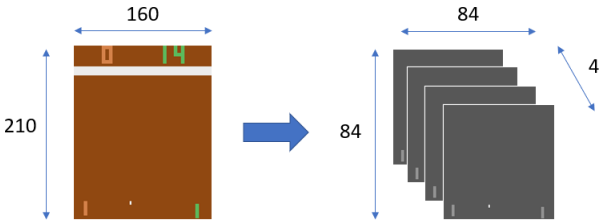

Figure 2.1: The original state (left side) and the preprocessed state (right side) with their dimensions

works for binary classification. It's domain spans $(-\infty, \infty)$, while its range is $(0, 1)$. Even though it can theoretically output any value in $(0, 1)$, small values (< 5) return a value close to 0, while larger values (> 5) return a value close to 1. This makes it appropriate to learn linearly separable problems, such as the output of the discriminator model in GAN. It's formal definition is:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1.13}$$

2 System Description

2.1 Preprocessing

The OpenAI Gym Atari emulator (Brockman et al., 2016) outputs 210×160 RGB images, which means that using these images as raw input from the algorithm would be too computationally expensive. The agent does not need this level of detail to learn, so, in order to improve the performance of the algorithm, these images need to be preprocessed. Since colors are not important for this task, the image is first converted to gray-scale and down-sampled to a size of 110×84. The upper part of the frame contains a display of the score in current game. This feature is not needed for learning, so there is no need to keep it. Also, the DQN architecture used expects square inputs, so the image is cropped to the appropriate size of 84×84. There is no way to figure out which way the ball is going by observing the frames one by one. As a solution, 4 consecutive frames are stacked together, so in the end, the states have the shape $84 \times 84 \times 4$. The transformation is illustrated in figure 2.1.

2.2 Model Architecture

Most of the data used in this project is in image form, so the architectures of the models use 2D convolutional layers. The DQN architecture is inspired by the one used in the DeepMind paper (Mnih et al., 2013). Its architecture is as follows: The input for the neural network is an image of the shape $84 \times 84 \times 4$. The first layer is a convolutional one with 32 8×8 filters with stride 4. The next convolutional layer involves 64 filters with a size of 4×4 and stride 2. The last convolutional layer contains 64 3×3 filters with stride 1. The last hidden layer is fully-connected with 512 units. All hidden layers use ReLU activation. Finally, the output layer is fully-connected, with 6 outputs, and it features a linear activation, representing the Q values of each action.

As for the Deep Convolutional Generative Adversarial Networks, the generator and discriminator networks are built in a symmetric fashion. The discriminator takes a $64 \times 64 \times 4$ shape as input. Afterwards, 64 4×4 convolutional filters with stride 2 with a leaky rectified linear activation (Leaky ReLU) are applied. The Leaky ReLU alpha is 0.2 for all layers. The next two layers are convolutional with 128 4×4 filters and stride 2, again followed by Leaky ReLU activation. The fourth layer flattens the results, followed by a dropout layer with a rate of 0.2. The last layer is densely connected with sigmoid activation since it outputs the probability that the input image comes from the original dataset.

The generator takes as input a latent space vector of size 128, which it projects by densely connecting it to $8 \times 8 \times 128$ nodes and then reshapes to the same dimensions. Next, the convolutional layers of the discriminator are replaced with transpose convolutional ones with Leaky ReLU activation. The transpose convolutional layers use a kernel size of 4 and a stride of 2. The number of filters for each layer are 128, 256 and 512, in this order. The last layer is a 5×5 convolutional with sigmoid activation and 4 output channels, one for each frame that is stacked in a single image.

2.3 Hyperparameter Choice

The exploration strategy used in DQN is ϵ-greedy with a decaying ϵ which ranges from 1 to 0.02. The ϵ

reaches its minimum in 260800 frames of gameplay, or approximately 60 games. The optimizer used to train the DQN is Adam (Kingma & Ba, 2014), with a learning rate of 0.0001. The last parameter of the DQN algorithm is the value of future reward γ. The value chosen for this is 0.99, which indicates that the algorithm should focus on future rewards more than on the immediate ones. This is appropriate in the game of Pong, because the positive rewards are attached to terminal states only. The positive reward is not obtained when the agent's paddle deflects the ball, but when the goal is scored. So, the action that triggers the positive reward is taken a couple of timesteps before the positive reward is actually registered.

The TN shares the architecture of the DQN, its only specific parameter being the number of training steps after which its weights are synchronized with those of the DQN's. This value must be set with respect to the training frequency of the main network, which is 1. The target update frequency is set to 1000, 3 magnitudes higher.

The ER has two important parameters: batch size and memory buffer size. The batch size is set to the fixed value of 32, since the architecture of the neural network only allows for an invariable input length. Due to computational resources, the memory buffer size is set to 10000. Since this is considered a relatively small size, sampling only begins when the memory buffer is full. A larger memory buffer size would further help reduce unwanted correlation, however it would also reduce training time, which is more valuable in this case.

Both the discriminator and the generator of the DCGAN are trained with the help of the Adam optimizer, with a learning rate of 0.0001.

2.4 Task Description

The task to be learnt is the Atari 2600 Pong, from the OpenAI Gym environment. In this game, the agent controls the right paddle and the computer controls the left one. The objective is to deflect the ball from the agent's goal and score on the computer's side. Each goal is valued to one point, with the game ending when one of the players reaches 21 points. After a goal is scored, the player which lost the point has to start the next game by serving the ball. In terms of MDP, a state is a frame of the game, which is a 210×160 RGB image (before preprocessing). Even though the paddle's position can only be modified vertically, 6 different actions exist:

- NOOP: the paddle does not do anything
- FIRE: this is used to start a new game after the previous has ended
- RIGHT: the paddle moves up
- LEFT: the paddle moves down
- RIGHTFIRE: serve the ball upwards
- LEFTFIRE: serve the ball downwards

The transition probability function $\mathcal{P}$ is not known, so the agent must learn to solve the task according to the state-value function Q. As for the reward signal, one point is rewarded for a goal on the opponent's side and one is subtracted for receiving a goal. If no one receives a goal, the reward is equal to 0. This means that the role of the agent will not only be to win the whole match but to do so with a high difference in score as well. For example, winning a match 21 to 20 will only award one point, which is not a good performance on the agent's part. As a result, the agent will learn to not only score goals, but also to defend them. As there is no time limit on Pong, the agent does not have incentive to finish the match quickly.

The specific task of Pong was not chosen arbitrarily. The main advantage that makes it a good choice for this project is that states that hold a positive reward are also terminal states. This makes it possible to artificially generate ER trajectories, as explained in section 2.5.

2.5 GAN-Replay

The process of GAN-Replay begins with the generation of positive reward states. The database used by the DCGAN is built during the training of the DQN agent with traditional ER. Whenever the agent encounters a positive reward state while interacting with the environment, it stores it in a list, which is saved every 10000 timesteps.

After the DCGAN model is trained on the generated database, the generator model is used to create artificial positive reward states, which are used, in turn, to fill the ER memory buffer at the start of

the DQN training sequence. Although the DCGAN only generates states, this is enough to fill the memory buffer with simulated trajectories. The components of the imagined experiences are as follows:

- s_t: DCGAN generated state
- a_t: random action from the action space
- r_t: the value 1
- d_t: the value "True"
- s_{t+1}: empty

The action is set to random because of the specific mechanics of the task. In a positive reward state, the ball is scored on the opponent's side and since the paddle controlled by the agent does not touch the ball at that time, the action it takes is not important. The next state is empty because there is no state after a goal is scored. All of the generated states are positive reward states, which, as specified in section 2.4, are terminal. The state-action value of a terminal state can be set to 0, so when the DQN encounters a positive reward state, the loss function that it had to originally minimize (equation 1.6) becomes:

$$L(\theta) = E_{\langle s_t, a_t, r_t, s_{t+1}\rangle \sim U(D)}\left[(r_t - Q(s_t, a_t; \theta))^2\right] \tag{2.1}$$

The term $\gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-)$ can be removed, thus also removing the need to artificially generate s_{t+1}.

For the sake of simplicity, the pseudocode for the DQN algorithm combined with GAN-Replay can be seen in algorithm 2.1.

Algorithm 2.1 DQN with GAN-Replay

Require: Experience Replay Queue of D size N
Require: $N = 10000$
Require: Q network with parameters θ
Require: Target Q network with parameters θ^-
Require: total_games = 0
Require: total_frames = 0
Require: total_reward empty Queue of size 100
Require: update_freq = 1000
initialize D with simulated trajectories
while True **do**
  set s_t as initial state
  total_games += 1
  while s_t is not terminal **do**
    total_frames += 1
    with probability ϵ select a random action a_t
    otherwise select $a_t = \max_{a \in \mathcal{A}} Q^*(s_t, a; \theta)$
    $\epsilon -= 0.02$
    execute a_t in emulator and get r_t, d_t and s_{t+1}
    store $\langle s_t, a_t, r_t, d_{t+1}, s_{t+1}\rangle$ in D
    $s_t := s_{t+1}$
    sample a random minibatch B = $\{\langle s_t, a_t, r_t, d_{t+1}, s_{t+1}\}\rangle$ of size 32 from D
    for i = 1 to 32 **do**
      if d_t^i is True **then**
        $y_t^i := r_t^i$
      else
        $y_t^i := r_t^i + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}^i, a; \theta^-)$
      end if
    end for
    $\theta := \underset{\theta}{argmin} \sum_{i=1}^{32} (y_t^i - Q(s_t^i, a_t^i; \theta))^2$
    if total_frames % update_freq = 0 **then**
      $\theta^- = \theta$
    end if
  end while
  mean_reward = mean of last 100 games
  if mean_reward > 19 or total_games > 300 **then**
    break
  end if
end while

3 Results

Both the discriminator and the generator are trained on the successful state database for 10000 epochs. Unfortunately, there are not many methods to measure the quality of the images generated by GAN. From a qualitative point of view, a selection of images that were randomly picked from the generated dataset, alongside images from the original dataset can be observed in figure 3.1. A more objective, quantitative, method does exist in

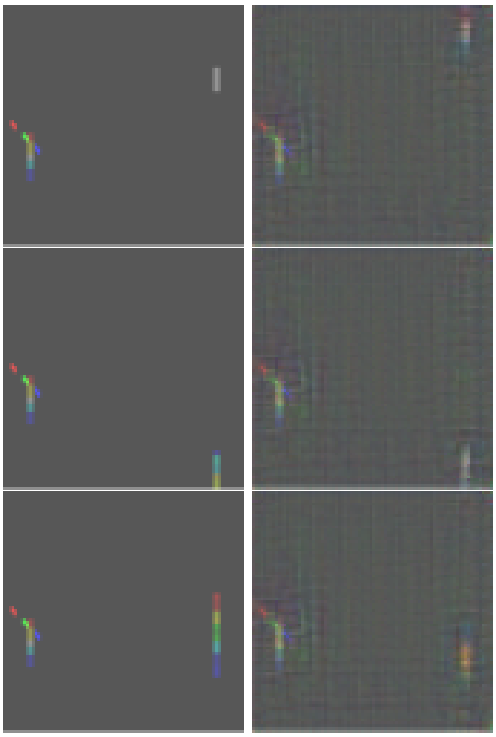

Figure 3.1: Hand-picked images from the original dataset (left column) and randomly-picked generated images (right column). Each overlapping frame is colored as to showcase the movement of the ball and paddles. The images used in the algorithm are completely gray-scale.

the form of the Frechet Inception Distance (FID) (Heusel et al., 2017). The FID uses the Inception v3 network (Szegedy et al., 2015) to compare the distributions of the generated images and the original ones. Upon calculating the FID between 1000 generated images and the original images used to generate them, the obtained score is 320.027.

In the context of ER, the quality of the GAN-generated images is not the only decisive feature. What is also crucial is the diversity of the images. In order to measure how diverse the real and the generated images are, the following qualitative method was devised. The mean image of the dataset is calculated and pixels that have a standard deviation higher than a fixed threshold are colored in bright red. The resulting images are displayed in Figure 3.2.

The performance of the DQN agent with both traditional ER and GAN-Replay is evaluated by

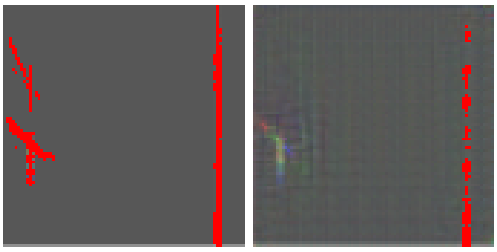

Figure 3.2: Average image of the original dataset (left side) and average image of generated dataset (right side). The pixels colored red diverge from the mean by 10 standard deviations.

Table 3.1: Scores averaged over last 100 games for each method

Method	Average score
DQN with ER	18.84 ± 3.084
DQN with GAN-Replay	18.54 ± 2.635
Human player	9.50
Random policy	-20.80

keeping the same hyperparameters for all runs of the experiment.

The DQN is run until the mean reward of the last 100 games is higher or equal to 19 or 300 games have been played. This mean reward bound is given because the purpose of this experiment is to observe if GAN-Replay leads to a faster convergence of the reward. This means that the most important aspect is the time of convergence and not the maximum reward.

The performance of the DQN algorithm in combination with the ER versions is evaluated on 2 criteria: average score and time of convergence. The scores averaged over the last 100 games can be seen in table 3.1, while the score is reported in graph form in figure 3.3. In DRL literature it is customary to add the score obtained by a human player and a random policy. Those are also included in table 3.1. To ensure consistency, all of the results were averaged over 5 experiment runs for each method.

4 Conclusion

The DQN agent learns by sampling the ER memory buffer, so the time it takes to improve might be dependent on the quality of the trajectories stored

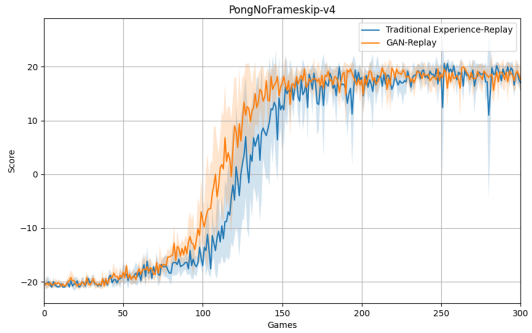

Figure 3.3: The scores obtained by DQN agents trained with traditional ER vs. GAN-Replay

in it. In an ideal setting, the memory buffer would contain diverse trajectories, consisting of high quality generated states, different actions and both positive and negative rewards.

4.1 GAN Image Quality

Upon visual inspection, the DCGAN generated images are blurry, but they resemble the original ones. The reported FID, however, is equal to 320.027, which is a large value for such a test. FID is a distance measure, so the ideal result would be 0. Applying the FID on the same dataset would produce a score of 0. This large value might be due to the fact that the used DCGAN architecture was initially designed to replicate RGB images with the shape $64 \times 64 \times 3$. It has been modified to accommodate gray-scale images with the shape $64 \times 64 \times 4$, which might be the case for the reduced performance.

4.2 GAN Image Diversity

The diversity of the images cannot be quantified, but it can be visually examined. The red pixels in the images from figure 3.2 signify that those pixels are different from the mean. This can be interpreted as a representation of the pixels that differ across the dataset. In both images, the majority of the background is black, because that part of the frame is never different across any of the images. Another common feature among the two pictures is that the space in which the agent-controlled paddle moves is red. This signifies that the datasets contain frames in which the agent's paddle is in many positions, although this is more visible in the left image. One of the crucial features is the coloring on the left side of the images. In the image from the original dataset, two distinct scoring patterns can be observed, represented by the opponent's paddle and a diagonal streak across it, signifying the movement of the ball. In contrast, these red pixels cannot be seen in the generated image. This means that most of the images in the generated dataset have the opponent's paddle and the ball fixed in the same position. This means that out of the two possible paddle and ball positions exhibited in the original dataset, only one is generated by the DCGAN, indicating that the architecture might be partially suffering from mode collapse.

4.3 Average DQN Score

According to Table 3.1, DQN achieves a higher average score with the traditional ER than with GAN-Replay by a small margin. This is, however, insignificant and could have been caused by the randomness element of the ϵ-greedy exploration strategy. Although they were not able to reach the maximum score of 21, both algorithms still out-perform most human players. In terms of average score obtained, traditional ER and GAN-Replay can be considered equal.

4.4 DQN Convergence Time

Even though they have the potential to reach impressive scores, perhaps one of the worst drawbacks of DRL methods is the time it takes for an agent to learn the given task. Both agents receive minimal rewards in the first 60 games, as that is approximately the time it takes ϵ to reach its minimum value. The bad performance after the exploration phase finishes is, in part, due to the fact that the agent does not know which states could reward him a positive score. As the agent acts mostly randomly in the beginning, such states are rarely reached. By initializing the ER memory buffer with those kind of trajectories, the agent not only skips the time it takes to actually collect those experiences, but it also forms a representation of what a successful state looks like, which is reflected in the DQN's weights. Thus, when the exploitation stage begins, the GAN-Replay agent benefits from the initializa-

tion and starts to achieve better rewards earlier than the non initialized one. As seen in figure 3.3, the score plot of the GAN-Replay begins its main climb at 70 games, approximately 30 games before the traditional ER. The GAN-Replay score stabilizes at 120, while traditional ER does so at 150, maintaining the same 30 game interval between them. These results show that the GAN-Replay converges to the same score apparently faster than the traditional ER, even though it does so by a small margin.

In order to analyse if the difference between the two plots is significant enough so that this version of GAN-Replay can be considered an actual improvement, a statistical test is required. The statistical test that measures if the difference between two groups is significant is the t-test. If the data respected the statistical assumptions of independence of observations, homogeneity of variance and normality of data, then an independent t-test could have been used. However, the results are not a part of a normal distribution, so a nonparametric statistical test must be used. The nonparametric equivalent of the independent t-test is the Wilcoxon Rank-Sum test. Upon performing it on the results of the two ER methods, the results are as follows: the statistic value is equal to -1.2233 and the p-value is 0.2211. The p-value is larger than 0.05, which means that the null hypothesis of equal means cannot be rejected, meaning that the data could come from the same distribution. Thus, in the current version of this algorithm, there is no significant improvement in convergence time.

5 Future Research

In the GAN-Replay method, only positive reward states are generated, since states with negative or no reward are easily encountered by the agent in the exploration phase. Positive reward experiences contain states that are informative for the agent, since reaching those states would help it further toward its goal. However, negative or no reward states are also useful for the learning process, since they represent the majority of the states that the agent will ever encounter. The positive reward states can be seen as the goal of the agent, but in order to reach them, it must also learn to navigate no reward states and even avoid negative reward states. In the case of the game of Pong, the feature states that the agent could learn the most from might not actually be the positive reward ones. In a positive reward state, the moment the ball is scored is captured. However, the agent does not act to score the goal in that very state. It does so a couple of frames before. The ball ricochets from the agent's paddle and then scores after a few frames. The state in which the ball touches the paddle of the agent before scoring could be considered as the state that the agent could learn the most from. Even though they have a neutral reward attached to them, those are the states that led to a positive reward, so the GAN-Replay method could be updated by initializing the ER memory buffer with such states.

References

Bellman, R. (1957). A markovian decision process. *Journal of mathematics and mechanics*, *6*, 679-684.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

Fukushima, K. (1975). Cognitron: A self-organizing multilayered neural network. *Biological Cybernetics*, *20*(3), 121-136.

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... Bengio, Y. (2014). Generative adversarial networks. Retrieved from `https://arxiv.org/abs/1406.2661` doi: 10.48550/ARXIV.1406.2661

Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., & Hochreiter, S. (2017). Gans trained by a two time-scale update rule converge to a local nash equilibrium. Retrieved from `https://arxiv.org/abs/1706.08500` doi: 10.48550/ARXIV.1706.08500

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. Retrieved from `https://arxiv.org/abs/1412.6980` doi: 10.48550/ARXIV.1412.6980

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*, 436-444. doi: https://doi.org/10.1038/nature14539

Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278-2324. doi: https://doi.org/10.1109/5.726791

McCloskey, M., & Cohen, N. J. (1989). Catastrophic interference in connectionist networks: The sequential learning problem. In G. H. Bower (Ed.), (Vol. 24, p. 109-165). Academic Press. doi: https://doi.org/10.1016/S0079-7421(08)60536-8

Minh, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., ... Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*, 529-533. doi: https://doi.org/10.1038/nature14236

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. Retrieved from `https://arxiv.org/abs/1312.5602` doi: 10.48550/ARXIV.1312.5602

Nitish, S., Geoffrey, H., Alex, K., Ilya, S., & Ruslan, S. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, *15*(56), 1929-1958. Retrieved from `http://jmlr.org/papers/v15/srivastava14a.html`

Radford, A., Metz, L., & Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. Retrieved from `https://arxiv.org/abs/1511.06434` doi: 10.48550/ARXIV.1511.06434

Rosenblatt, F. (1957). The perceptron — a perceiving and recognizing automaton. *Technical Report 85-460-1*.

Rummery, G., & Niranjan, M. (1994). On-line q-learning using connectionist systems. *Technical Report CUED/F-INFENG/TR 166*.

Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized experience replay. Retrieved from `https://arxiv.org/abs/1511.05952` doi: 10.48550/ARXIV.1511.05952

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, *3*, 9-44.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT Press.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2015). Rethinking the inception architecture for computer vision. Retrieved from `https://arxiv.org/abs/1512.00567` doi: 10.48550/ARXIV.1512.00567

Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8(3-4)*, 279-292.