Measuring the Impact of Memory Replay in Training Pacman Agents using Reinforcement Learning

Fabian Fallas-Moya, Jeremiah Duncan, Tabitha Samuel and Amir Sadovnik
Electrical Engineering and Computer Science Department
The University of Tennessee
Knoxville, United States
{ffallasm, jduncan51, tsamuel, asadovnik}@vols.utk.edu

***Abstract*—Reinforcement Learning has been widely applied to play classic games where the agents learn the rules by playing the game by themselves. Recent works in general Reinforcement Learning use many improvements such as memory replay to boost the results and training time but we have not found research that focuses on the impact of memory replay in agents that play simple classic video games. In this research, we present an analysis of the impact of three different techniques of memory replay in the performance of a Deep Q-Learning model using different levels of difficulty of the Pacman video game. Also, we propose a multi-channel image - a novel way to create input tensors for training the model - inspired by one-hot encoding, and we show in the experiment section that the performance is improved by using this idea. We find that our model is able to learn faster than previous work and is even able to learn how to consistently win on the mediumClassic board after only 3,000 training episodes, previously thought to take much longer.**

***Index Terms*—Q-Learning, memory replay, deep learning, reinforcement learning**

I. Introduction

One of the most important milestones in Reinforcement Learning (RL) was achieved when Mnih et al. [1] could implement the Deep Q-Learning idea in order to automatically play Atari games. After this, tons of research has been developed using the ideas of Deep Q-Learning and this led RL to be a hot topic along with Deep Learning models [2]–[4]. Video games - and especially classic videos games - offer a whole new opportunity to test RL models as a preliminary stage before moving to more complex problems [5]–[12]. Video games offer controlled environments at a very cheap-easy-low-cost to first try RL models. Then the algorithms can be applied to a wide variety of real-world problems. Video games offer a great opportunity of having well-defined environments with a great diversity of complexity.

On the one hand, the main advantage of RL is that it only needs feedback from the environment to learn. On the other hand, the training time has been always the main disadvantage of RL algorithms. To alleviate the training time, memory replay has been proposed [13] as a solution to have a faster convergence of the model. Unfortunately, we have not found a formal comparison of the different techniques of memory replay and video games environments. So, in this research,

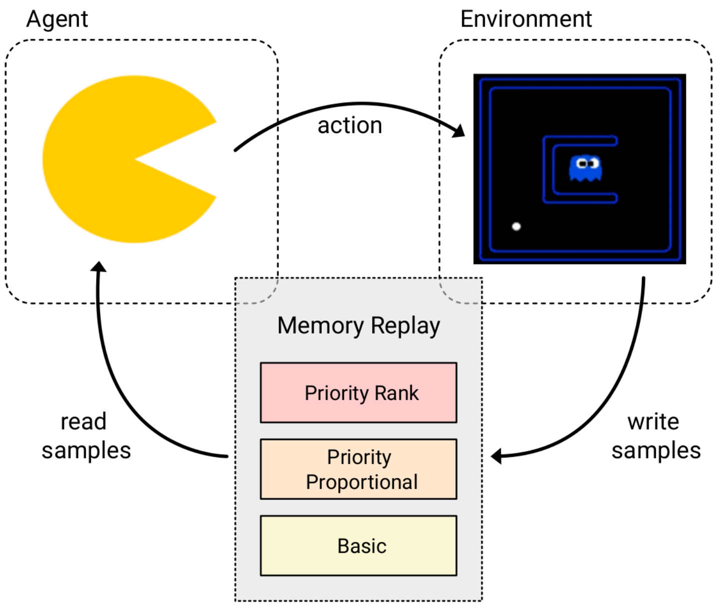
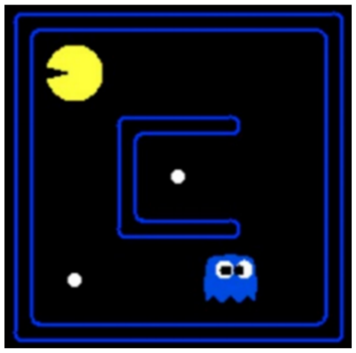


Fig. 1: The main idea of our approach. We have the memory replay as the intermediate buffer from where the agent takes samples for training.

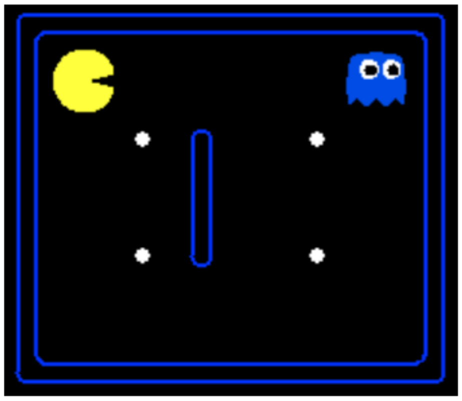
we compared different memory replay techniques and measure their performance using Deep Q-Learning. Figure 1 shows the idea of our experiments where memory replay works as a buffer where the interaction between the agent and the environment generates a lot of future states and the training works by sampling from this buffer. We used three different settings for the memory replay as defined by Liu and Zou [14]. Also, we explored the impact of two different input data, one input is a regular RGB image and the other has n channels where every channel represents one component of the environment.

In this work, we use one of the most popular games of all time: Pacman. We use a popular Pacman environment implementation supplied by The University of California, Berkeley (UC Berkeley) [15]. We modified their environment to allow

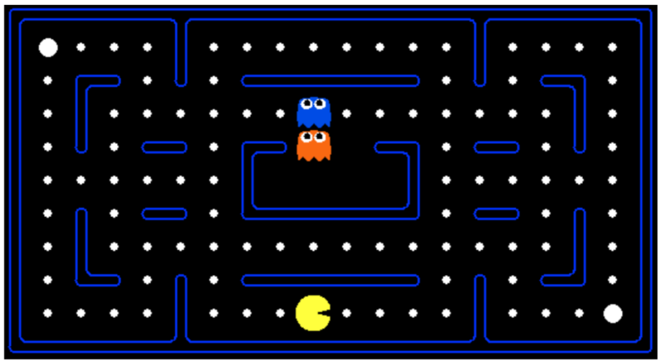
2021 XLVII Latin American Computing Conference (CLEI) | 978-1-6654-9503-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/CLEI53233.2021.9640031



(a) Small grid.



(b) Medidum grid.



(c) Large grid also known as medium grid classic.

Fig. 2: Example of images from the simulator we used for this research.

us to add our own agents, models, and replay memory. Figure 2 shows an example of the different environment size we used. Also, it is important to mention that we also experiment using two different input data, first we implemented experiments using regular RGB images and using the component-per-channel approach as shown in Figure 3.

In our work, we added a few key improvements that we did not come across in any previous work. For example, all of the previous work used basic memory replay as suggested by Mnih et al. [1] but to the best of our knowledge, no one has implemented prioritized replay when using Pacman with either of the open-source available environments (Gym and UC Berkeley). We implemented prioritized replay memory using both variants presented by Schaul et al. [13], proportional and rank based.

A. *Previous Work*

Relevant work related to Pacman was done by Mnih et al. [16] from DeepMind[1], where they present the first deep learning model as a function approximation. However, the experiments are applied to only a few games and the ideas presented in the paper are no longer the standard for new RL research. For example, the authors use a semi-gradient method with a single neural network.

Another paper from DeepMind [1] was an important work to create a robust solution where they used memory replay and two networks in order to tackle the semi-gradient issue. They showed how their ideas are better at generalizing across a broad variety of Atari games. This work focuses on implementing the ideas from their paper such as memory replay and a Deep Q-Learning Network (DQN).

It is important to indicate that this project is based on a popular template that is freely available and used as a project for an RL class [15]. We want to make note that there are many available repositories on the internet with various solutions to this problem. We chose to use PyTorch for two main reasons. The first is that many of those solutions are created using TensorFlow[2,3] and we do not want to be seen as replicating already established work. The second is because PyTorch is much more transparent in what each operation does and we find it is easier to understand from a glance as would a reader studying the implementation.

Domínguez-Estévez et al. [17] implemented a solution using DQN but they used the game “Ms. Pacman vs. Ghosts” from Gym[4] which is similar to the UC Berkeley environment but one provided by Gym does not offer an option to have a small environment like the one shown in Figure 2.

Gnanasekaran et al. [18] show interesting results using memory replay, DQN, and Double DQN. We use some of their ideas in our work, such as using the same model across different sized boards and implementing their model to test against ours.

B. *Input methods*

The main problem we are dealing with is related to train an agent to play Pacman on different board sizes. The rules of Pacman are simple: eat all of the food and capsules while dodging the ghosts. The goal is then to do this as quickly as possible.

The state-space of Pacman can be formulated as a Markov Decision Process (MDP) in multiple ways. There are six

[1]https://deepmind.com
[2]https://github.com/advaypakhale/Berkeley-AI-Pacman-Projects
[3]https://github.com/jasonwu0731/AI-Pacman
[4]https://gym.openai.com

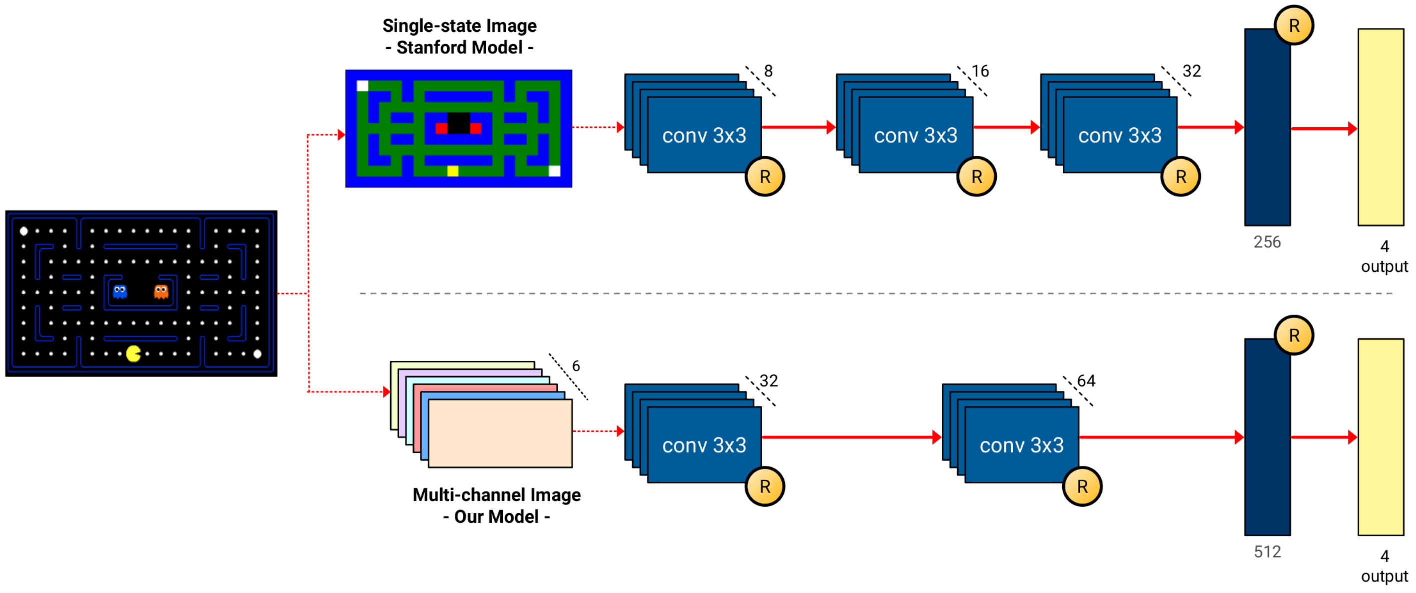


Fig. 3: Two different architectures we used. On top uses the traditional RGB single image (we call it *Single-state Image*) with an architecture proposed in one state-of-the-art papers [19]. The bottom model is our proposed model which uses *n*-channels to represent the image as a tensor (every channel has a component of the environment).

primary components that constitute the state of the game whose location needs to be kept track of.

1) Pacman: Where Pacman is currently at on the board.
2) Ghost: Where the ghosts are currently at on the board.
3) Capsules: The locations of the "large" dots. When Pacman eats a capsule, all of the ghosts become scared and edible.
4) Food: The location of the "small" dots.
5) Walls: The static positions of each wall on the board.
6) Scared Ghosts: The location of ghosts that are scared and edible.

From these, we create two ways to concatenate the state so that the model can learn from it.

a) Single-state image: The first is by converting the Pacman board and state into an image ($n \times m \times 3$ tensor). We assign colors to each of the six different parts of the state, as done by Gnanasekaran et al. [19]. This approach can be seen in Figure 3 on top.

b) Multi-channel image: The second is to create an $n \times m \times 6$ tensor where each channel corresponds to one of the different components of the state we care about. For instance, the first channel only contains the position of Pacman, while the second contains the location of walls, and so on. This approach can be seen in Figure 3 at the bottom. We came out with this idea that is closer to the one-hot implementation.

As for the action space, like [19] and [15], we allow the cardinal directions and stopping to be valid actions. We use the rewards supplied by our simulator (adapted from [15]) as the rewards sent to our agent. To expand, the rewards were set with inspiration from the classic maze escape problem where the default is to give -1 point at all time steps, -500 if eaten by a ghost (losing), +10 for eating food, +200 for eating a scared ghost, and +500 for eating the last food capsule (winning).

II. Methods

Since our problem is fairly complicated, we think that using hand-crafted features and a large state, action array would be time-consuming and unfruitful. Instead, we opted to explore Deep Q-Learning and its potential to solve complicated problems. Deep Q-Learning - introduced by [1] - suggests using two networks, target and policy networks, to perform Q-Learning using CNNs.

A. Q-Learning models

We created two Deep Q-Learning networks with different goals in mind, to compare how well each is able to solve our problem. Figure 3 shows the two architectures we used in our experiments. The first one shown on top was suggested by Gnanasekaran et al. [19] with three convolutional layers (8, 16, and 32 filters respectively) and a fully connected layer with 256 neurons. The second model, shown at the bottom, was created by us with two convolutional layers (32 and 64 filters, respectively), and a fully connected layer with 512 neurons. We iterated towards this model by wanting to create a simpler model in terms of feature extraction (hence two layers instead of three), and with more parameters to be able to learn the correct actions for more states. In both models, each layer is followed by a ReLU activation layer.

B. Memory replay

Because Deep Q-Learning relies on replaying past experiences to learn, we wanted to see if different types of replay functions could aid in solving the problem more quickly. Our base, a basic replay function, stores new experiences in a large deque that gets rid of old experiences once it starts to overfill, and when we perform training steps, it randomly samples 32 previous time-steps. There are two more methods of picking which previous time-steps to sample suggested by [13] called Prioritized Experience Replay, and here we need to associate every experience with additional information (priority). The idea is to use the error of every sample and use that as the priority. In simple terms, the higher the error the higher the priority. First we take a sample batch from the memory replay and then we update the priority of this batch by using the error we got. Schaul et al. [13] propose two ways of getting priorities as follows:

- Proportional: $p_i = |\delta_i| + \epsilon$, here the ϵ is a small value just to ensure that no sample will have zero probability, that way all the samples will have a chance to be picked.
- Rank-based: $p_i = 1/rank(i)$ sorts the priorities according to $|\delta|$ to get the rank.

where δ is the error on that particular sample. It is also important to indicate that α and β are two hyper-parameters related to priority replay. As explained by Schaul et al. [13], α helps us to determine the level of prioritization. So, we have that

$$P(i) = \frac{p_i^\alpha}{\sum_k p_i^\alpha} \tag{1}$$

The β hyper-parameter is a weighting value that indicates how much we want to update the weights of the model. So, incorporating β we have the formula:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i))}\right)^\beta \tag{2}$$

β controls how much prioritization to apply to the current batch.

The suggested loss function for most reinforcement problems, to our knowledge, is Huber Loss. However, while we were initially creating these models and trying to get them to work, we noticed that sometimes using Mean Squared Error Loss would let the model converge more quickly. So for the purposes of our experiments, we run trials with both losses. Overall, we have described a large number of parameters to tweak. One more parameter that we seek to analyze is the different board sizes supplied by our Pacman environment (small grid, medium grid, and large grid, shown in Figure 2).

Table I summarizes the options for each of these parameters. We experiment with 36 combinations. We tested all three grid sizes provided by the UC Berkeley environment. We tested the Stanford model as defined in Gnanasekaran et al. [19], and our model (see Figure 3). To strictly follow the approach of Gnanasekaran et al. [19], we used images representing the states during training with their model while using a one-hot encoded binary structure for training with our model. Regarding the memory replay, we test three approaches: basic memory replay as explained by Mnih et al. [1], and proportional and rank-based prioritized memory replay as explained by Schaul et al. [13]. And finally, we tried the two losses we previously described, Mean Squared Error (MS)E and Huber.

Grid Size	Model	Experience Replay	Loss
Small Grid	Stanford	Basic	MSE
Medium Grid	Ours	Proportional	Huber
Large Grid (medium classic)		Rank-based	

TABLE I: Different factors we analyzed in this research. We tested three different scenarios, two models, two losses, and the experience replay memory which turns out to be the most important factor we wanted to test.

C. Agents

We perform Deep Q-Learning as describe before. Some of the most relevant features of our agents are as follows.

- *Actions*: Selects an action randomly or from the model, depending on the current ϵ value. Stores state-action data to the selected replay.
- *epsilon*: Determines the current ϵ value for the ϵ-greedy policy in
- *train_action*: Performs a step of training. That entails: asking the replay for a batch of data, calculating the expected state action values, and performing one back-propagation step.

We have two agents that were implemented for experiments. The first is the one used with the Stanford model [19], called *ImageDQNAgent*.

- It creates a target and policy network using the Stanford model.
- Also, it converts the state given by the UC Berkeley environment (the simulator) to an RGB image like in [19].

The second agent is the one we proposed, called *ChannelDQNAgent*.

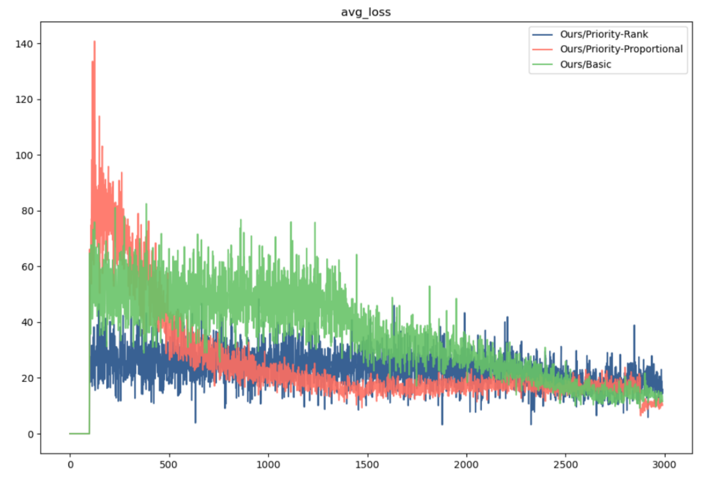
- It creates a target and policy network using our model
- Also, it converts the state given by the UC Berkeley environment to a 6 channel tensor.

III. Results Analysis

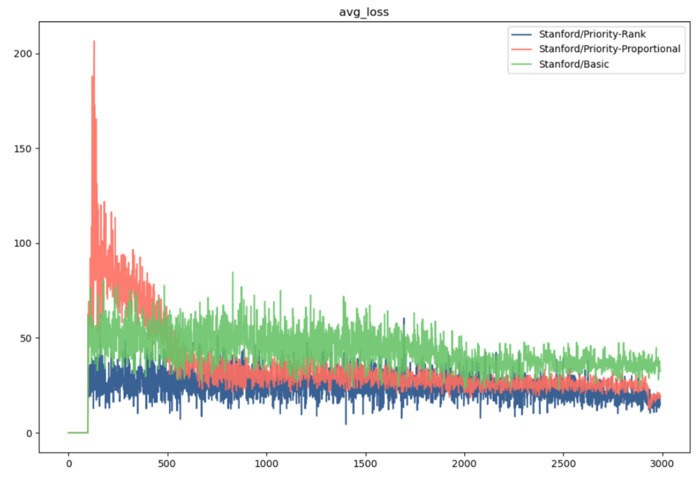
As indicated in Table I, we ran 36 experiments where we combined the grid size (the size of the Pacman maze), the model (Stanford or our model), the three approaches for the memory replay (basic, prioritized-proportional, and prioritized-rank-based), and the loss function (Huber and MSE). Figure 4 shows the behavior during training, whereas Figure 5 shows the testing results.

A. Hyperparameters

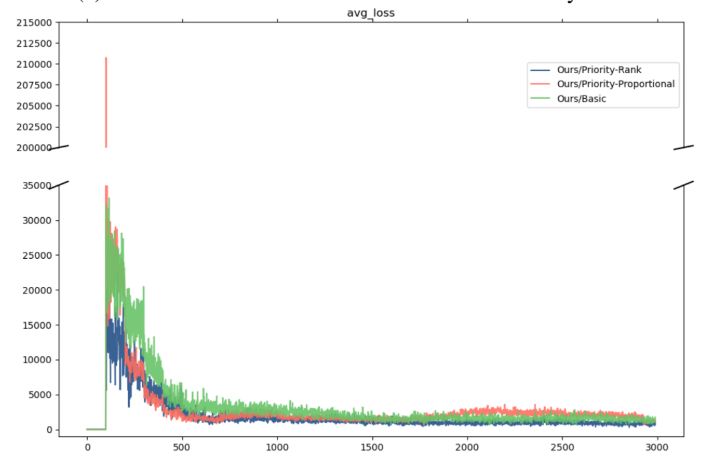
There are a number of hyperparameters that can be tweaked across runs that we needed to determine how to set. An important thing is to replicate results from previous work, in this case, our benchmark was related to the results reported by Gnanasekaran et al. [19], consequently, we first started with the settings recommended in this paper. Unfortunately, using



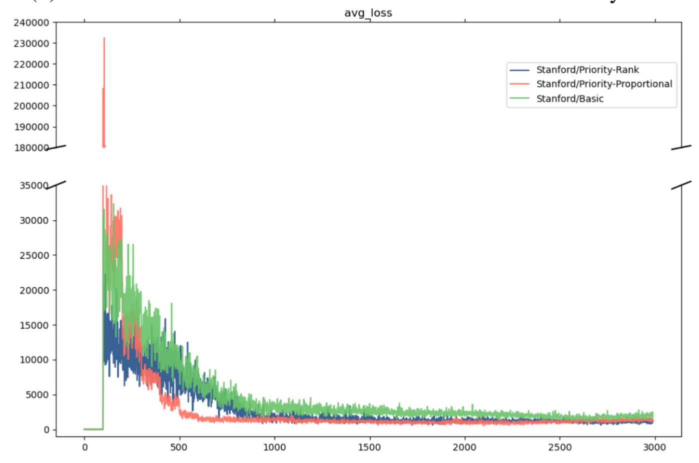
(a) Huber loss and our model on the smallGrid layout.



(b) Huber loss and the Stanford model on the smallGrid layout.



(c) MSE loss and our model on the smallGrid layout.



(d) MSE loss and the Stanford model on the smallGrid layout.

Fig. 4: Loss plots during training with the three different approach of replay: basic, proportional, and rank-based.

their settings with the models we had did not produce results that were optimal. Instead of 1500-2500 episodes needed for a 100% win rate, we had to increase all of our models to 3,000 training episodes.

The learning rate they suggested, 2.5×10^{-4} worked for that many training episodes. We also found that linearly decreasing ϵ from 1 to 0.1 as training progressed worked well. We used these hyperparameter settings across all of our models and for all of the grids. While we might be able to get better results using different hyperparameters in certain scenarios and parameter combinations, we wanted to give a fair comparison to all models across all of our parameter combinations so we kept them static across runs.

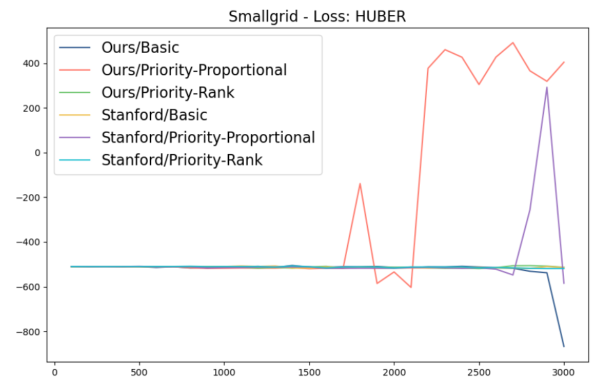
B. Loss Analysis and Training

First, we want to discuss the behavior of the training related to our experiments. Figure 4 shows the results of the loss during training of the smallGrid by comparing the behavior of the three approaches we used for memory replay. Regarding the Huber loss, Figure 4a shows that rank-based replay exhibits a slow convergence rate while basic and proportional replay has a more linearly decreasing behavior. In the intermediate episodes, proportional seems to converge faster and in the last episodes (2800-3000) proportional improves with a sudden fall to beat basic replay. Figure 4b shows the case where basic replay has the worst behavior and rank-based has the best performance. An important aspect is to realize the size of the scale (the y-axis) between Figures 4a and 4b, where our model (Figure 4a) has better results (smaller losses) during training.

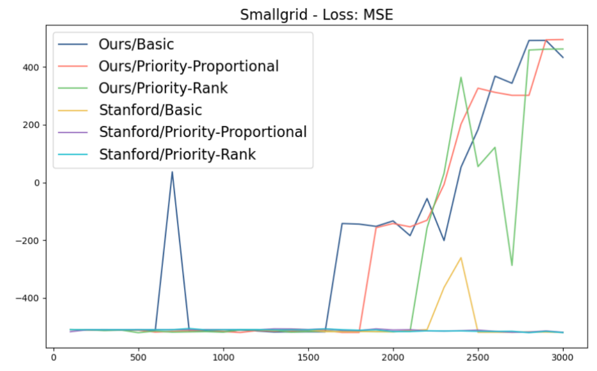
Figures 4c and 4d show the behavior using MSE as the loss function. As seen, the three approaches of memory replay have similar behavior, for example, it is hard to distinguish a real “winner”in the last episode (3000), since the results are overlapping - as in the case of Figure 4c, or pretty close to each other - as in the case of Figure 4d. However, by looking at the small differences we can say that rank-based has the best performance since it shows a linearly decaying value in the average loss.

C. Testing results

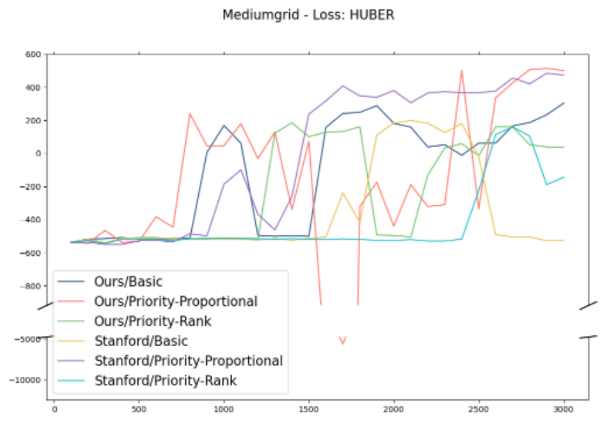
Figure 5 shows the testing results for all three grids using our model, the Stanford model, and the three versions of



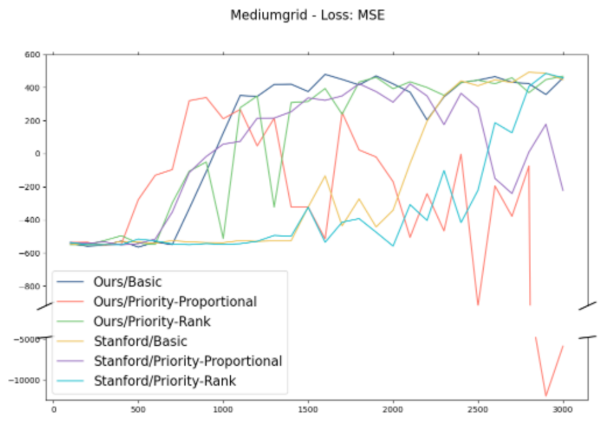
(a) smallGrid scores using Huber loss.



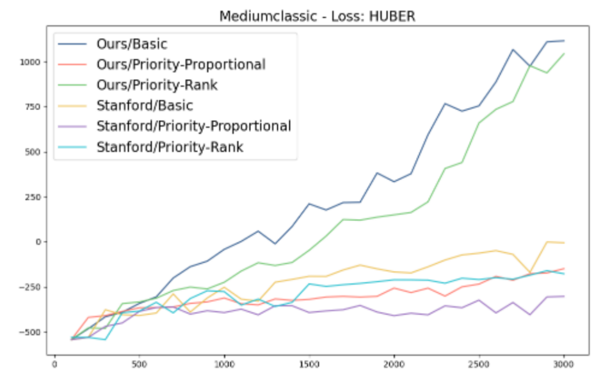
(b) smallGrid scores using MSE loss.



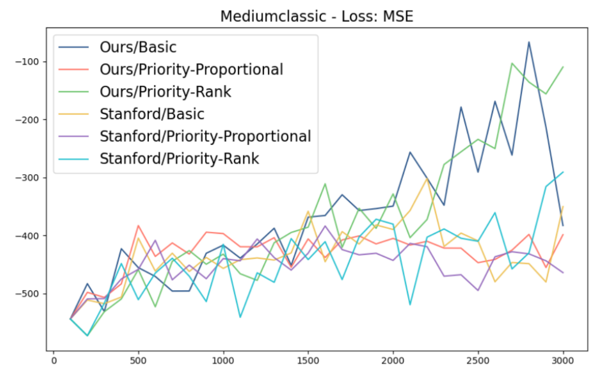
(c) mediumGrid scores using Huber loss.



(d) mediumGrid scores using MSE loss.



(e) mediumClassic grid scores using Huber loss.



(f) mediumClassic grid scores using MSE loss.

Fig. 5: Results of the scores during testing. In every case, we compared the three different memory replay approaches and the two models we used.

memory replay. Figure 5a and 5b show the results using the smallGrid. Interestingly, the smallGrid behaves better when using MSE during training, as seen in Figure 5b where 3 out of 6 experiments show high/positive scores in the last episodes. However, using the Huber loss, just one model has an acceptable result (Figure 5a). Also, the three successful experiments in Figure 5b were using our model, and it can be seen that our proposed model surpasses the Stanford model in the smallGrid experiments using all three different replay memory approaches.

Figure 5c shows that in the case of using Huber loss, the Stanford model and ours have the best results when using prioritized proportional memory replay. Our model also has good behavior when using the basic memory replay, and the rest of the combinations show erratic, unsuccessful behavior. In this case, it could be possible that more episodes would help to have more consistent and successful results. When using MSE (see Figure 5d) the behavior changes drastically where the Stanford model and our model show the worst performance when using proportional replay. Furthermore, 4 out of 6 experiments have good results when using MSE, and it seems that the combination here of MSE and basic and rank-based replay have the best results, and our model (with basic replay and rank-based) is more consistent in converging.

Finally, we have the mediumClassic grid experiments in Figures 5e and 5f. The first one, using Huber, shows a trend to increase their scoring results. However, only our model using rank-based and basic replay is able to converge and consistently win on the mediumClassic board. That is not the case of Figure 5f, where none of the models using MSE are able to converge. Of all the experiments we performed, this is the only one where no positive scores were achieved.

IV. Conclusions

From the previous section, we can say that MSE has better results in the smallGrid and mediumGrid with our initial configuration of 3000 episodes. By using the smallGrid we found that only the proportional replay memory showed good results regardless of the loss function used. In the case of the mediumGrid, Huber loss performs better even though MSE also has good results. However, when using the mediumClassic grid which is the most complicated scenario, Huber has the best results over MSE which does not achieve any positive scoring. Consequently, with simple scenarios, MSE is an acceptable option whereas Huber needs a lot of episodes to converge, but for generalization purposes, Huber seems to be the right choice in the Pacman game, and most real Pacman levels are more similar to the mediumClassic experiment than to the smallGrid and mediumGrid ones.

Regarding the memory replay we found that in some cases the basic replay is good enough in some cases, this can be explained by the fact that the Pacman is not a complicated game -in terms of the goals to achieve- so, having the basic replay memory is an acceptable choice. The real advantages of rank-based and proportional memory replay are in specific scenarios, for example, proportional seems to be better in simple scenarios (smallGrid and mediumGrid) while rank-based seems to work better in more complex ones (mediumClassic). In any case, our model along with basic memory replay always shows promising results and it is the most consistent of all the different experiments.

Finally, we see throughout all the experiments that our proposed model performs better than the Stanford model. Our model has more filters and a larger dense layer, so, in this case, having more weights equates to better performance when using the Pacman video game.

These sets of experiments were performed using PacMan as a platform to show that memory replay in backpropagation can increase the efficiency of the model significantly. Instead of backpropagation every result of the feed-forward network, using memory replay the algorithm selectively picks a subset of results to backpropagate. This approach can be easily extended to other image-based reinforcement systems to increase efficiency.

V. Future Work

We realize that choosing the model is extremely important in this problem, so an interesting path for future experimentation would be to apply a parameter search to find the best architecture for the model.

Also, we found that using our novel idea of the multi-channel image - inspired by the one-hot encoding principle - we got much better results, having more challenging scenarios can give more confidence in order to generalize this finding. Definitely, this idea is worth to be used instead of the regular RGB image when the problem setting allows using this idea.

And finally, due to time constraints, we did not implement double Deep Q-Networks (DDQN). It will be interesting to observe if DDQN can achieve comparable or better results than what we have achieved in the same or fewer episodes given that we needed to use 3000 episodes on average.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb 2015.

[2] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[3] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "A brief survey of deep reinforcement learning," *IEEE Signal Processing Magazine, Special Issue on Deep Learning for Image Understanding*, vol. abs/1708.05866, 2017.

[4] P. Dayan, *Reinforcement Learning.* American Cancer Society, 2002.

[5] K. Shao, Z. Tang, Y. Zhu, N. Li, and D. Zhao, "A survey of deep reinforcement learning in video games," *ArXiv*, vol. abs/1912.10944, 2019.

[6] A. Tucker, A. Gleave, and S. J. Russell, "Inverse reinforcement learning for video games," *ArXiv*, vol. abs/1810.10593, 2018.

[7] C. Amato and G. Shani, "High-level reinforcement learning in strategy games," in *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1 - Volume 1*, AAMAS '10, (Richland, SC), p. 75–82, International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[8] M. E. Taylor, N. Carboni, A. Fachantidis, I. Vlahavas, and L. Torrey, "Reinforcement learning agents providing advice in complex video games," *Connection Science*, vol. 26, no. 1, pp. 45–63, 2014.

[9] M. Jaderberg, W. M. Czarnecki, I. Dunning, L. Marris, G. Lever, A. G. Castañeda, C. Beattie, N. C. Rabinowitz, A. S. Morcos, A. Ruderman, N. Sonnerat, T. Green, L. Deason, J. Z. Leibo, D. Silver, D. Hassabis, K. Kavukcuoglu, and T. Graepel, "Human-level performance in 3d multiplayer games with population-based reinforcement learning," *Science*, vol. 364, no. 6443, pp. 859–865, 2019.

[10] M. E. Taylor, N. Carboni, A. Fachantidis, I. Vlahavas, and L. Torrey, "Reinforcement learning agents providing advice in complex video games," *Connection Science*, vol. 26, no. 1, pp. 45–63, 2014.

[11] M. Jaderberg, W. M. Czarnecki, I. Dunning, L. Marris, G. Lever, A. G. Castañeda, C. Beattie, N. C. Rabinowitz, A. S. Morcos, A. Ruderman, N. Sonnerat, T. Green, L. Deason, J. Z. Leibo, D. Silver, D. Hassabis, K. Kavukcuoglu, and T. Graepel, "Human-level performance in 3d multiplayer games with population-based reinforcement learning," *Science*, vol. 364, no. 6443, pp. 859–865, 2019.

[12] K. Shao, D. Zhao, N. Li, and Y. Zhu, "Learning battles in vizdoom via deep reinforcement learning," in *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 1–4, 2018.

[13] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *International Conference on Learning Representations*, vol. abs/1912.10944, 2016.

[14] R. Liu and J. Zou, "The effects of memory replay in reinforcement learning," in *2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 478–485, 2018.

[15] "Uc berkeley cs188 intro to ai." http://ai.berkeley.edu.

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[17] F. Domínguez-Estévez, A. A. Sánchez-Ruiz-Granados, and P. P. Gómez-Martín, "Training pac-man bots using reinforcement learning and case-based reasoning," in *CoSECivi*, 2017.

[18] A. Gnanasekaran, J. Feliu-Faba, and J. An, "Reinforcement learning in pacman." http://cs229.stanford.edu/proj2017/final-reports/5241109.pdf. Accessed: 2020-11-30.

[19] J. An, J. Feliu, and A. Gnanasekaran, "Reinforcement learning for pacman," 2017.