Deep Q-Learning and Memory Replay in training Pacman agents

ECE-517 Reinforcement Learning
Prof. Dr. Amir Sadovnik

Jerry Duncan
EECS Department
The University of Tennessee
Knoxville, TN, USA
jduncan51@utk.edu

Fabian Fallas-Moya
EECS Department
The University of Tennessee
Knoxville, TN, USA
ffallasm@vols.utk.edu

Tabitha Samuel
EECS Department
The University of Tennessee
Knoxville, TN, USA
tsamuel@utk.edu

***Abstract*—In this work we implement Deep Q-Learning methods to solve the classic arcade game of Pacman. We compare two different models and the speed at which they learn to play the game on different size boards. We investigate the effect of using three different variants of experience replay and two different loss functions for a total of 36 independent experiments. 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*—Deep Q-Learning, Priority Replay**

I. Introduction

An important milestone in Reinforcement Learning (RL) research was a paper that showcased Deep Q-Learning models that could learn to play Atari games automatically [1]. Since then, many researchers have been interested in this field because video games offer controlled environments that many researchers can use to quickly iterate on their new RL algorithms. Then the algorithms they create can be applied to a wide variety of real world problems. So video games offer a great opportunity of having well defined environments with a great diversity of complexity.

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) [2]. We modified their environment to allow us to add our own agents, models, and replay memory. Figure 1 shows an example of one of the Pacman boards their environment provides.

II. Previous Work

Relevant work related to Pacman was done by Mnih et al. [3] from DeepMind[1], where they present the first deep learning model as a function approximation. However, the experiments

Fig. 1: A screenshot of the smallGrid environment.

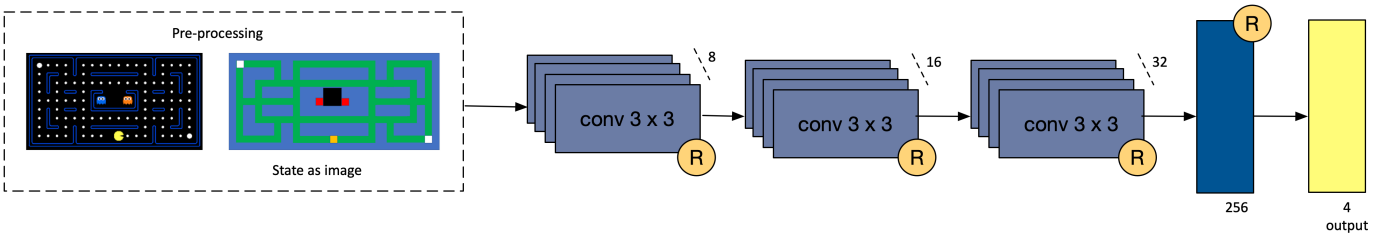
are applied to only few games and the ideas presented in the paper are no longer the standard for new RL research. For example, the authors use a semi-gradient method with a single neural network.

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).

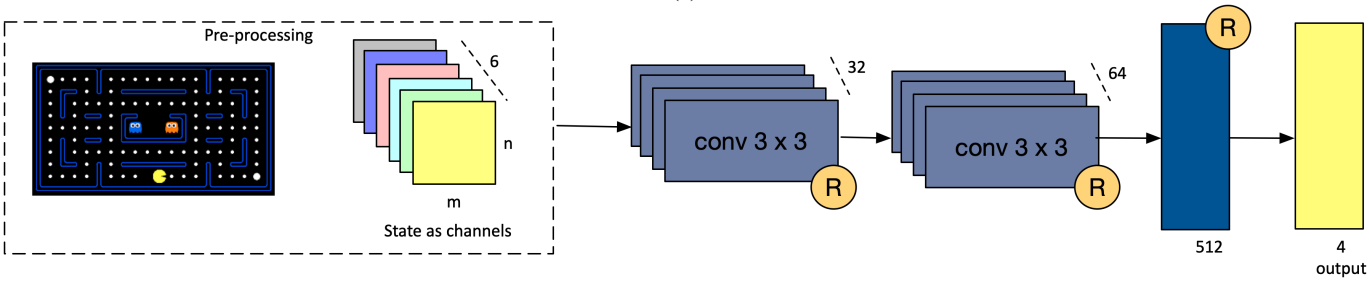
It is important to indicate that this project is based on a popular template that is freely available and used as a project for a RL class [2]. We want to make note that there are many available repositories on the internet with various solutions to this problem. We chose to use PyTorch for two main 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

[1]https://deepmind.com

[2]https://github.com/advaypakhale/Berkeley-AI-Pacman-Projects
[3]https://github.com/jasonwu0731/AI-Pacman



(a)



(b)

Fig. 2: 2a is the model proposed by Gnanasekaran et al. [4] that preprocesses the state into an image and 2b is our model that converts the state into a 6 dimensional tensor

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. [5] 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 any option to have a small environment as the one shown in Figure 1.

Gnanasekaran et al. [6] 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.

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. [7], proportional and rank based.

III. Domain Problem

The problem we are working on is training 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 an MDP in multiple ways. There are six primary components that constitute the state of the game whose location needs to be kept track of.

- Pacman: Where Pacman is currently at on the board.
- Ghost: Where the ghosts are currently at on the board.
- Capsules: The locations of the "large" dots. When Pacman eats a capsule, all of the ghosts become scared and edible.
- Food: The location of the "small" dots.
- Walls: The static positions of each wall on the board.
- 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) State as an 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 [4].

b) State by channels: 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.

As for the action space, like [4] and [2], we allow the cardinal directions and stopping to be valid actions. We use the rewards supplied by [2] 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).

[4]https://gym.openai.com

IV. Reinforcement Learning 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, a target and policy network, to perform Q-Learning using CNNs. We created two Deep Q-Learning networks with different goals in mind, to compare how well each is able to solve our problem. Figure 2 shows the two architectures we used in our experiments. The first one shown in Figure 2a was suggested by Gnanasekaran et al. [4] with three convolutional layers (8, 16, and 32 filters respectively) and a fully connected layer with 256 neurons. The second shown in Figure 2b, 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 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.

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 [7] 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. So, 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. [7] proposes 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 there are two hyper-parameters related to priority replay, α and β. As explained by Schaul et al. [7], α help 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:

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

TABLE I: The possible values for each of the four options we wanted to test the effect of. In total, there are 36 experiments (combinations, $3\times3\times2\times2$) that we tested.

Grid Size	Model	Experience Replay	Loss Function
smallGrid	Stanford	Basic Replay	Mean Squared Error
mediumGrid	Ours	Proportional Priority	Huber Loss
mediumClassic		Rank-Based Priority	

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 (smallGrid, mediumGrid, and mediumClassic, shown in Appendix A). In Table I, we summarize the options for each of these parameters. In total there are 36 experiments that we tested, one for each possible combination. We tested all three grid sizes provided by the UC Berkeley environment. We tested the Stanford model as defined in Gnanasekaran et al. [4], and our model (see figure 2). To strictly follow the approach of Gnanasekaran et al. [4], 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. [7]. And finally, we tried the two loss functions we previously described, MSE and Huber.

V. Code Design

We used an object oriented design. We wrote everything outside of the *pacman* folder except *play.py*, which was supplied by UC Berkeley but heavily modified by us. *agents.py* contains all code related agents playing the game.

Each Agent subclasses *DQNAgent*, which provides all methods needed to play and train the models.

- *getAction*: 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 *getAction*
- *final*: Called when an episode is finished to perform stat related calculations and set the weights of the target network to the ones of the policy network
- *registerInitialState*: On the very first run, it calls *build* to setup the initial models
- *train_step*: Performs a step of training. That entails: asking the replay for a batch of data, calculating the expected state action values, and performing one backpropagation step.

- *build*: Implemented by subclass
- *get_state_tensor*: Implemented by subclass

We have two agents that implement certain methods of *DQNAgent*. The first is the one used with the Stanford model, *ImageDQNAgent*.

- *build*: Creates a target and policy network using the Stanford model
- *get_state_tensor*: Converts the state given by the UC Berkeley environment to an image like in [4]

The second is our model, *ChannelDQNAgent*.

- *build*: Creates a target and policy network using our model
- *get_state_tensor*: Converts the state given by the UC Berkeley environment to a 6 channel tensor described in Section III

We set up a script, *experiments.py*, to run all of our experiments.

models.py contains the PyTorch code for the Stanford and Our model.

Lastly we implement our replays in an object oriented manner as well. Each replay subclasses *Replay*, which must provide a *sample* method for getting a batch of timesteps, a *push* method for saving new data, and *weight_losses* to allow priority replays to weight the losses.

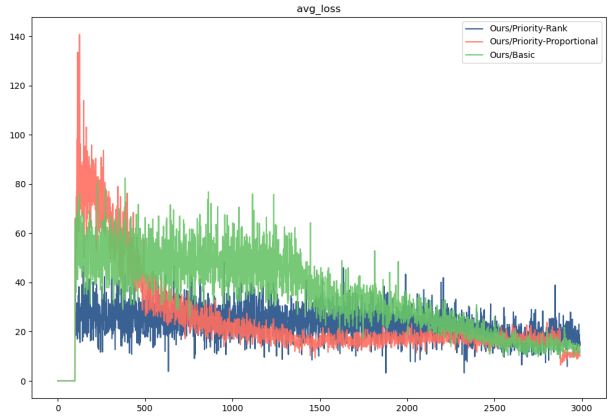
VI. Results Analysis

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), and the results can be seen in figure 4.

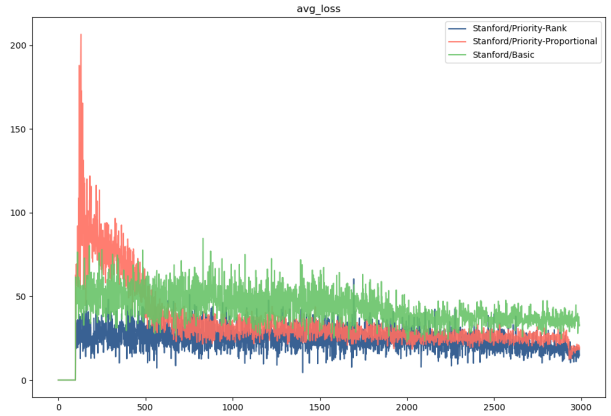
a) Hyperparameters: There are a number of hyperparameters that can be tweaked across runs that we needed to determine how to set. Our goal was to have results at least as good as the ones reported by Gnanasekaran et al. [4] so we first started with the settings recommended in the paper. Unfortunately, using 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 went on worked well. We used these hyperparameter settings across all of our models 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.

A. Loss Analysis and Training

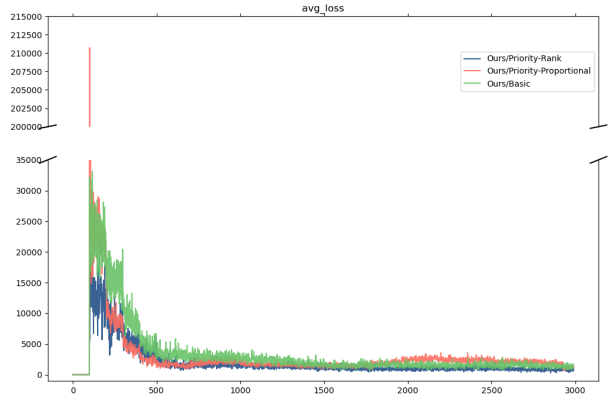
First, we want to discuss the behavior of the training related to our experiments. Figure 3 shows the results of the loss during training of the smallGrid by comparing the behavior of



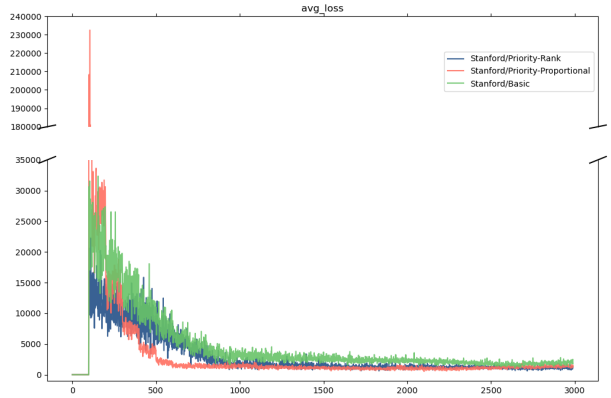
(a) Huber loss and our model on the smallGrid layout.



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

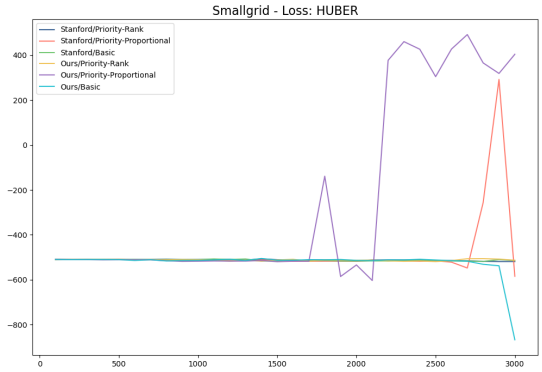


(c) MSE loss and our model on the smallGrid layout.

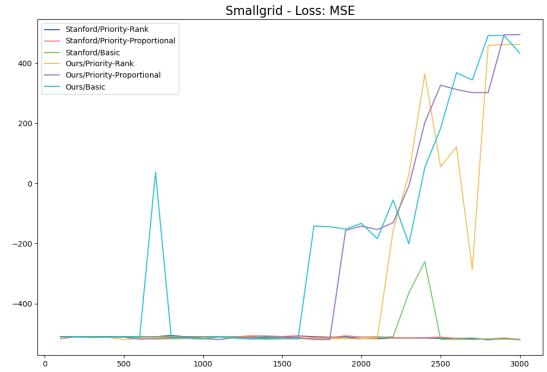


(d) MSE loss and the Stanford model on the smallGrid layout.

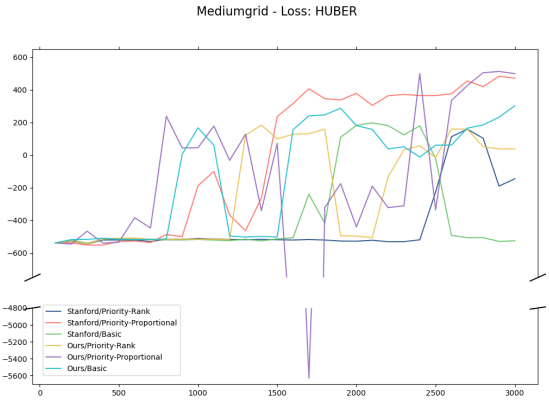
Fig. 3: Loss plots during training with the three different approach of replay: basic, proportional, and rank-based.



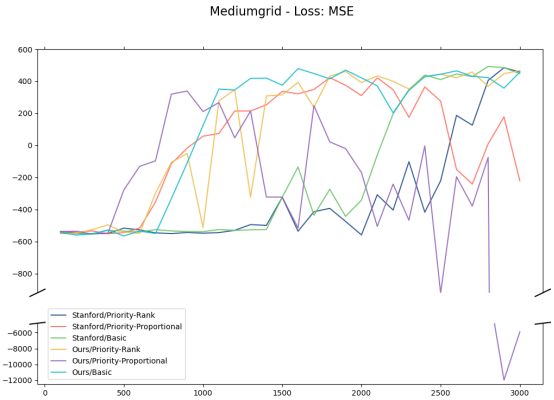
(a) smallGrid scores using Huber loss.



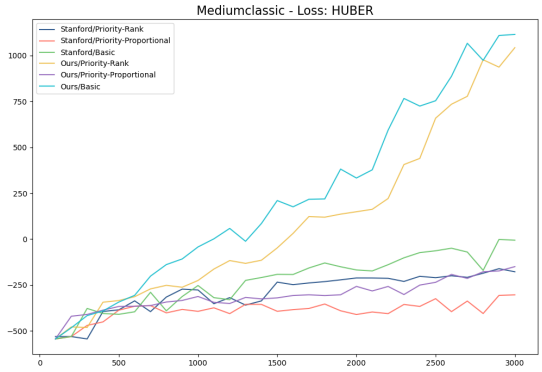
(b) smallGrid scores using MSE loss.



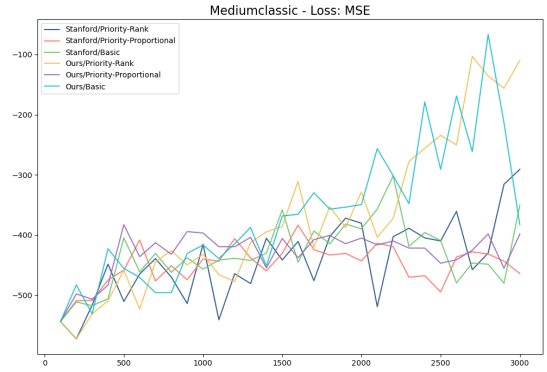
(c) mediumGrid scores using Huber loss.



(d) mediumGrid scores using MSE loss.



(e) mediumClassic grid scores using Huber loss.



(f) mediumClassic grid scores using MSE loss.

Fig. 4: Results of the scores during testing. In every case we compared the three different memory replay approaches and the two models we used.

the three approaches we used for memory replay. Regarding the Huber loss, Figure 3a shows that rank-based replay exhibits a slow convergence rate while basic and proportional replay have a more linearly decreasing behavior. In the intermediate episodes, proportional seems to converge faster and in the last episodes (2800-3000) proportional improves with an sudden fall to beat basic replay. Figure 3b 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 3a and 3b, where our model (Figure 3a) has better results (smaller losses) during training.

Figures 3c and 3d show the behavior using MSE as the loss function. As seen, the three approaches of memory replay have a 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 3c, or pretty close to each other - as in the case of Figure 3d. 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.

B. Testing results

Figure 4 shows the testing results for all three grids using our model, the Stanford model, and the three versions of memory replay. Figure 4a and 4b show the results using the smallGrid. Interestingly, the smallGrid behaves better when using MSE during training, as seen in Figure 4b 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 4a). Also, the three successful experiments in Figure 4b 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 4c 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 a 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 4d) the behavior changes drastically where the Stanford and ours 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 4e and 4f. 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 4f, where none of the models using MSE are able to converge. Of all the experiments we performed, this is the only ones where no positive scores were achieved.

VII. Conclusions

From the previous section we can say that MSE has better results in the smallGrid and mediumGrid with our initial setting 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 good enough and Huber is slow 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 between 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 a better performance in Pacman.

VIII. 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 the one-hot encoding we got better results, having more challenging scenarios can give more confidence in order to generalize this finding. 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 that what we have achieved in the same or fewer episodes.

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, no. 7540, pp. 529–533, Feb. 2015, ISSN: 1476-4687. DOI: 10.1038/nature14236. [Online]. Available: https://doi.org/10.1038/nature14236.

[2] *Uc berkeley cs188 intro to ai*, http://ai.berkeley.edu/project_overview.html, Accessed: 2020-11-30.

[3] 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. arXiv: 1312.5602. [Online]. Available: http://arxiv.org/abs/1312.5602.

[4] J. An, J. Feliu, and A. Gnanasekaran, *Reinforcement learning for pacman*, 2017.

[5] 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.

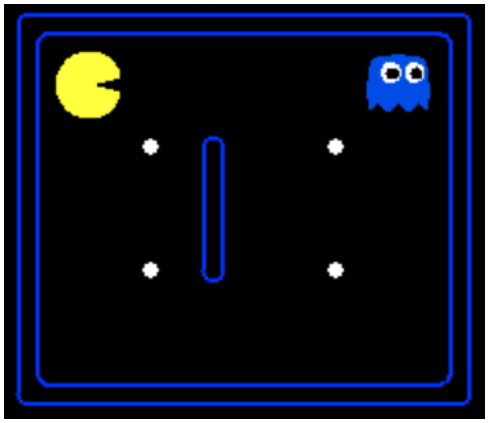
[6] A. Gnanasekaran, J. Feliu-Faba, and J. An, *Reinforcement learning in pacman*, http://cs229.stanford.edu/proj2017/final-reports/5241109.pdf, Accessed: 2020-11-30.

[7] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, *Prioritized experience replay*, 2016. arXiv: 1511.05952 [cs.LG].

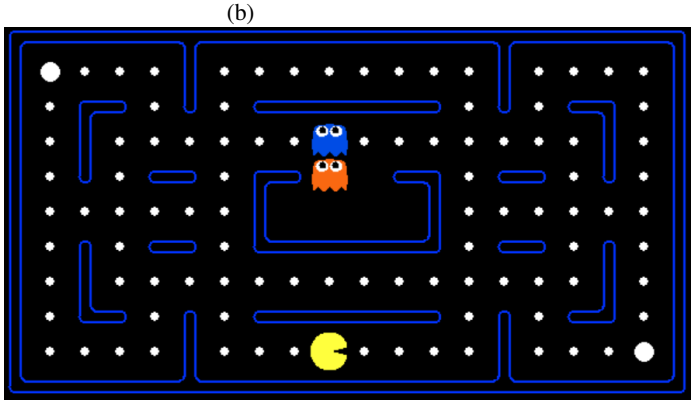
APPENDIX

In Figure 5 we showcase what the different grids look like so the reader understands the complexity of each.

(a)



(b)



(c)

Fig. 5: All of the grid sizes that we test. 5a is smallGrid, 5b is mediumGrid, and 5c is mediumClassic.