University of Zurich

Reinforcement Learning

Semester Project

Mountain Car with Q-Learning

Atari Breakout with DQN

Fall Semester 2020

Team name: Happy Campers

Nick Vecci, 18-747-782

Kevin Hardegger, 12-758-785

Anej Zver, 19-769-645

Dec 13th, 2020

1. Summary

We used reinforcement learning to tackle two problems - the Mountain Car and Atari Breakout. For the mountain car problem, we employed Q-learning and trained seven models with different hyperparameters to 10'000 episodes. Our best model achieved an average score of -130 (the score for a single episode is either -200 or 0) over 500 episodes. For Breakout, we followed the approach taken by DeepMind and employed a Deep-Q Network with four hidden convolutional layers and one output layer as a function approximator, training seven different configurations. Our best model achieved a maximum score of 271, with the average and mean scores of 65.83 and 23, respectively.

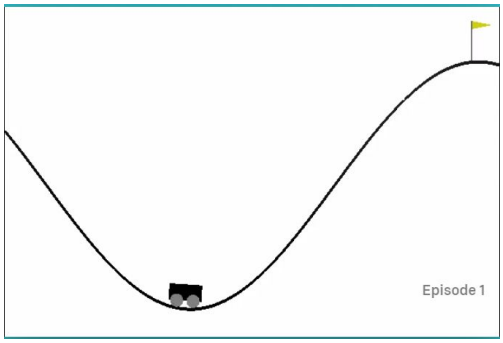
2. Introduction to the problems

For our project, we decided to focus on two problems of varying difficulties. Our first problem is the Mountain Car Problem, for which we use simple Q-Learning.

The mountain car is a one dimensional track situated between two mountains. The goal is to drive the car to the top of the hill on the right hand side and reach the flag. However, the car's engine is not strong enough to scale the mountain in a single instance. Therefore, players need to drive back and forth across the two mountains to gain momentum.

Our second problem is the famous 1970s Atari game Breakout. We follow DeepMind's 2013 paper [4] and implement our own Deep Q-Network (DQN).

In Breakout, a layer of bricks lines the top third of the screen and the goal is to destroy them all. A ball moves straight around the screen, bouncing off the top and two sides of the screen [2].



Figures 1 and 2: Mountain Car and Atari Breakout

3. Mountain Car with Q-Learning

3.1. Set up

Our code is based on the Q-learning tutorial by sentdex at pythonprogramming.com [8]. We adapted the code to improve on the initial parameters that were used in the tutorial.

3.1.1. Environment

We used OpenAI Gym's Atari Mountain Car environment (MountainCar-v0).

3.1.2. State-space

We reduced the state space from a continuous space to a discrete 20x20x3 (position, velocity, action) observation space, as recommended [8].

3.1.3. *Action space*

The space is discrete and consists of 3 actions:

1. Push car left
2. Push car right
3. Do nothing

3.1.4. *Rewards*

If the car doesn't reach the flag, the score for that episode is -200. The score for reaching the flag is 0.

3.1.5. *Q-learning algorithm*

Based on sentdex's tutorial, we built a Q-learning algorithm with the following starting parameters:
Learning rate = .1
Discount = .95
Epsilon = .5

We ran alternate specifications of the Q-learning algorithm to identify the best version. We ran every specification for 10,000 episodes.

3.1.6. *Random agent vs greedy player*

To help with the comparison, we ran every specification with a random agent and a greedy agent. We defined the random agent as playing purely randomly (where epsilon is the probability of choosing a uniformly random action), whereas we defined the greedy agent as the agent maximizing the q-table with probability 1-epsilon. The average score is compared for each specification, and is calculated as the average score every 500 episodes.

3.2. Sensitivities and parameters

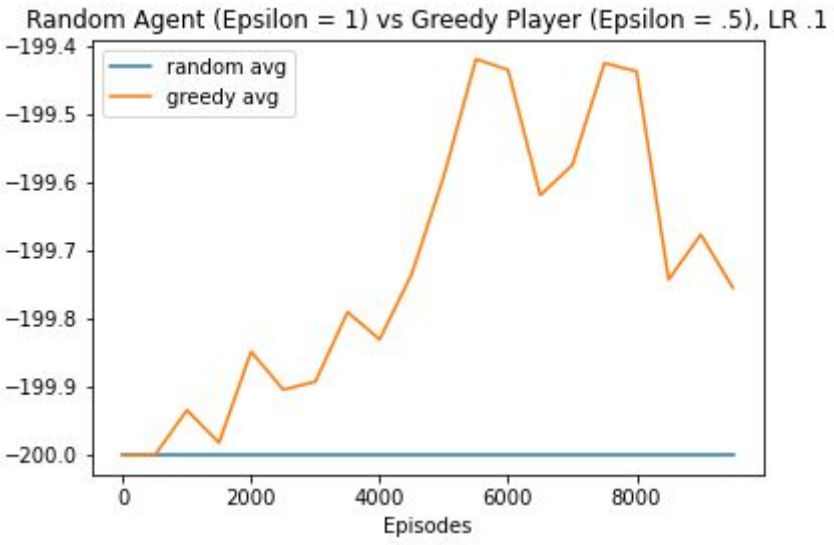
We ran a total of 7 configurations.

Configuration 1 Learning rate: 0.1 Epsilon: 0.5 (greedy), 1 (random) Discount rate: 0.95	**Configuration 2** Learning rate: 0.5 Epsilon: 0.5 (greedy), 1 (random) Discount rate: 0.95
Configuration 3 Learning rate: 0.1 Epsilon: 0.1 (greedy), 1(random) Discount rate: 0.95	**Configuration 4** Learning rate: 0.1 Epsilon: 0.01 (greedy), 1 (random) Discount rate: 0.95
Configuration 5 Learning rate: .01 Epsilon: 0.01 (greedy), 1 (random) Discount rate: 0.95	**Configuration 6** Learning rate: 0.1 Epsilon: 0.001 (greedy), 1 (random) Discount rate: 0.95
Configuration 7 Learning rate: 0.01 Epsilon: 0.01 (greedy), 1 (random) Added epsilon decay Discount rate: 0.95	

Table 1: Parameters used for each of the models.

3.3. Results and discussion

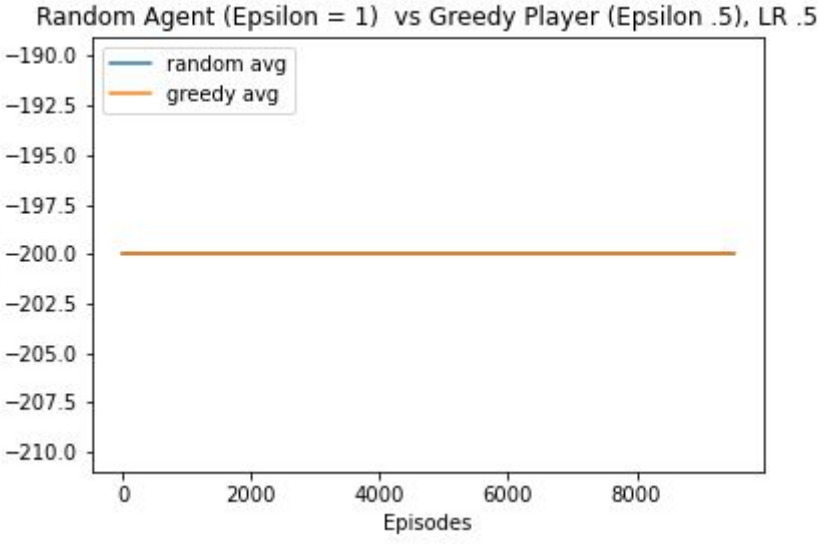
Figure 3



As we can see from figure 3, the agent playing randomly stays at -200, which means it was never able to reach the flag. The agent playing randomly with probability 0.5 and selecting an action maximizing the q-table with probability 0.5, steadily increased the average score for the first approximately 6,000 episodes. However, the actual score was only an improvement of approximately 0.6 of one point. The average score steadily decreased after that. Although this shows the greedy player was able to reach the flag more times than the random player, it is a poor result.

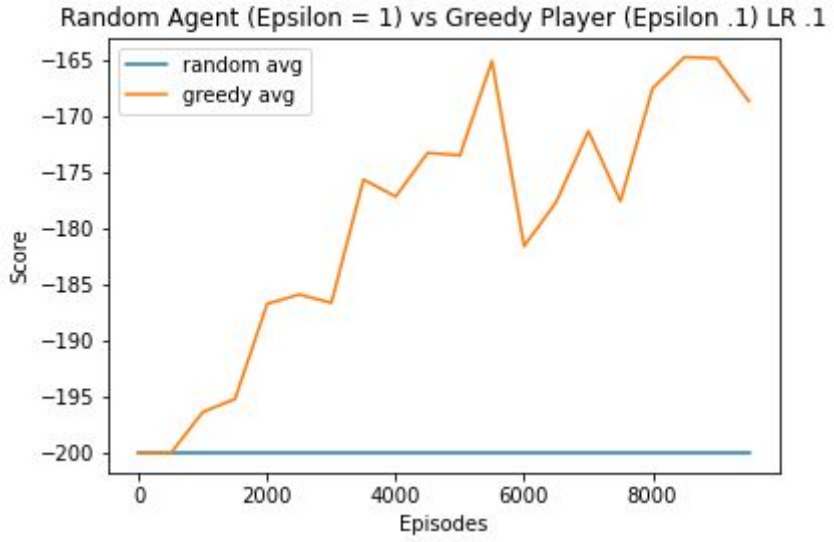
The Bellman equation tells us that increasing the learning rate means placing more weight on recently learned values when updating the q-value. Perhaps the algorithm is not updating its q-value fast enough. After all, we can see that the agent reached the flag within approximately 500 episodes, but was not able to score consistently afterwards. Therefore we tried increasing the learning rate to 0.5, as seen in figure 4.

Figure 4



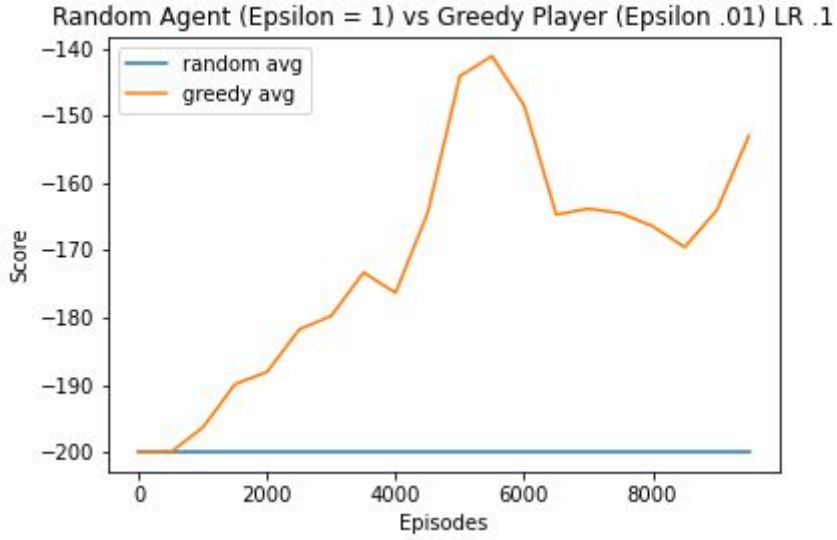
After increasing the learning rate to 0.5, we can see in figure 2 that both the random and greedy algorithms never reached the flag in 10,000 episodes. Holding epsilon constant and increasing the learning rate has decreased the models performance. Therefore we decided to return to the original learning rate and decrease the epsilon value. Decreasing the epsilon value will mean the algorithm will exploit rather than explore more often.

Figure 5



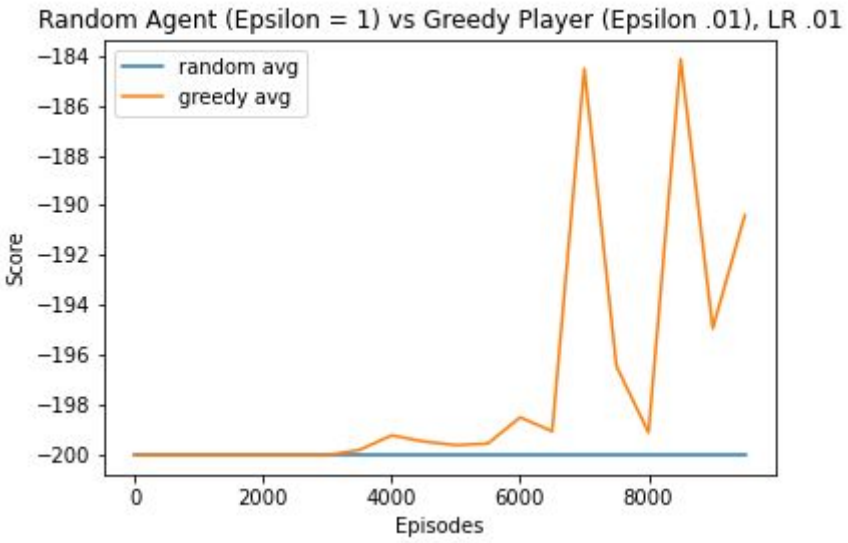
As we can see in figure 5, reducing the learning rate back to its original value (0.1) and decreasing the epsilon increased the model performance substantially. The algorithm quickly reaches the flag, and once it does average performance continues to improve, although there was a plateau and decline across the last approximately 1500 episodes.

Figure 6



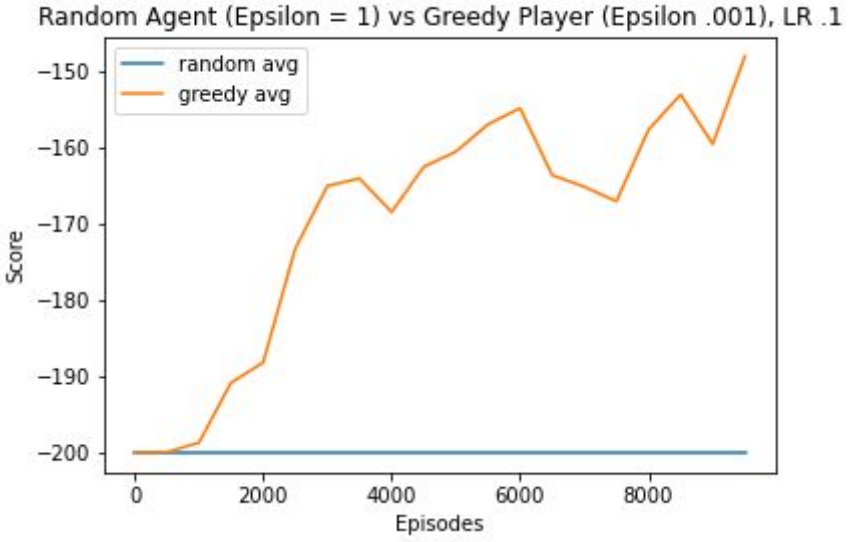
Given the success we had with decreasing the epsilon value for the third model, we decided to decrease it further again. Figure 6 shows that this model performs even better, consistently scoring higher than the model in figure 7. It seems to reach a peak of close to -140 after 6,000 episodes, declining sharply but then improving again.

Figure 7



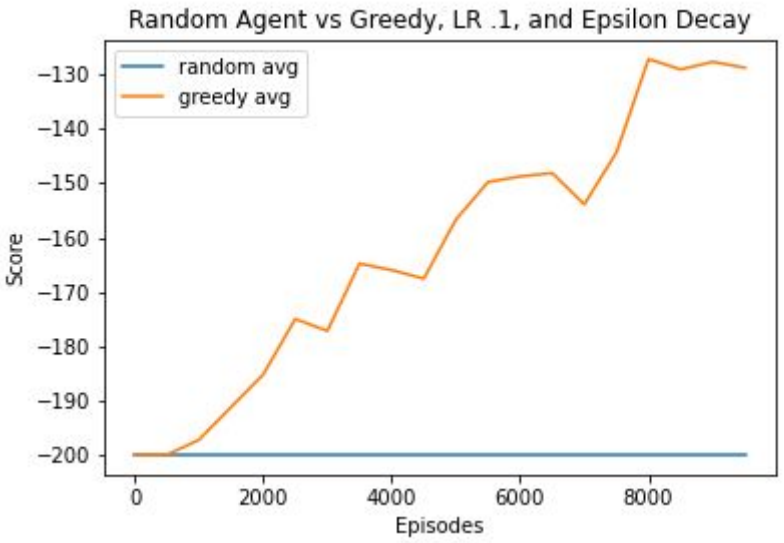
This time, instead of changing the epsilon value, we changed the learning rate. The model in figure 7 is the result of decreasing the learning rate further to 0.01. We can now see the average score changing very sharply after 6,000 episodes. We can see that once the agent has some success it scores very highly for a short period of time, but the reverse is also true. Perhaps the learning rate is now too low, and the agent is not updating its q-value fast enough so it does not correct itself fast enough when it is on a downward trajectory. A learning rate that is too low could explain the phenomena seen in figure 7.

Figure 8



For our next specification, we returned the learning rate back to 0.1, where it seemed to perform well previously, and we reduced the epsilon value in order to exploit more and explore less. Figure 8 plots the results. As we can see, this model seems to perform very well. The model does not seem to have sharp changes in average values like in figure 7 and the average score continues to increase throughout the episode range, albeit with a decline between episode 6,000 - 7,500.

Figure 9



So far we have had two models that perform similarly well. Both had learning rates of 0.1, which signals that this learning rate for this game gets the learning step size right. As the game only has three actions, a reasonably small discrete observation space, it is reasonable that a low learning rate, which uses small step sizes when updating q-values is optimal. This might be beneficial because there is not a large environment, action space or sophisticated reward structure to be learned. However, we have also shown that there seems to be a limit--a learning rate of 0 .001 was too low.

Given we think we have a decent understanding of a suitable learning rate, and two models with differing epsilon values that perform similarly well, we decided to add some sophistication into the epsilon parameter. Figure 7 shows the results of a learning rate of 0.1, an epsilon value of 0.01, and epsilon decay which decreases the exploration as the algorithm gets better at the game. We can see here that this model performs the best, consistently scoring higher than any of the other games and reaching an average of approximately -130.

In short, after trying 7 specifications we believe our best model was specification 7, because it had a higher average score. It had a low learning rate of 0.1, which we believe makes sense given the nature of the game, and the small action and observation space. We found the best epsilon value was also quite low (0.01), which also makes sense because there is less to be gained from exploring in this environment, than a larger, more complex environment with a more sophisticated reward structure. Adding an epsilon greedy parameter also seemed to improve the score, which allows for more exploration at the beginning.

3.4. Possible improvements

First, we never altered the discount rate to see how that would change the results. In the future, we could run more configurations that include changing the discount rate and different levels of epsilon decay. Second, we could use an area under the curve measure to compare models, instead of relying on graphed average values. Finally, we could use a different benchmark. In the end, the random agent did not serve as a very instructive tool for comparison.

4. Atari Breakout with DQN Learning

4.1. Set up

Our code is based on DeepLizard's reinforcement learning tutorial[5]. We adapted it for Breakout, as it's used for the cart pole problem in the tutorial. The code can also be used to train models for other Atari games with minimal changes.

4.1.1. Environment

For the environment, we used OpenAI Gym's [6] Arcade Learning Environment [1], which uses the Stella Atari emulator [7]. The environment provides observations in the form of an array of shape (210, 160, 3), which represents an RGB image of the screen.

There are multiple versions of the environment, which differ in the number of frames for which each chosen action is performed and action repeat stochasticity. We use Breakout-NoFrameskip-v4 with no frameskip and no action repeat stochasticity.

The model does not have a conception of the environment and doesn't see the environment's internal state at any point.

4.1.2. State-space

We preprocess the observations from the environment using OpenAI Gym's wrappers. The raw images are cropped to 84x84 pixels and the last four frames are stacked. The frames are stacked to obtain all the necessary information: position, direction and velocity of the ball and paddle.

The resulting stack of four 84x84 frames represents our state space and serves as the input to the model.

4.1.3. Action space

The action space is discrete and consists of four actions:

NOOP = no movement
FIRE = start the game by launching the ball
RIGHT = move paddle to the right
LEFT = move paddle to the left

4.1.4. Rewards

The reward is the number of points obtained during one episode.

4.1.5. Deep-Q-Network

We replicate the convolutional neural network used in the DeepMind paper, consisting of:

- A convolutional layer with 4 in channels and 32 out channels
- A convolutional layer with 32 in channels and 64 out channels
- A convolutional layer with 64 in channels and 64 out channels
- A linear layer with 64*7*7 in features and 512 out features

- An output layer with 512 in features and 4 out features, which represent our action space

The rectified linear activation function is applied to all but the output layer.

Since we need to pass through the network two times to calculate the current and target Q-values, we create a copy of the original policy network and call it the target network. This network lags behind the policy network and is updated with the policy network's weights every n-steps. How often the target network is updated is one of the parameters.

4.1.6. Experience replay and the Epsilon-Greedy strategy

For training, we use experience replay, which draws minibatch samples of experiences from our replay memory. The replay memory consists of the last 100'000 of the agent's experiences $e_t = (s_t, a_t, r_t, s_{t+1})$.

The agent chooses the next action according to an Epsilon-Greedy strategy.

4.2. Sensitivities and parameters

We trained a total of 8 different models. The parameters used are in the table on the next page.

Configuration 0 (base) - 26'000 episodes batch_size = 256 gamma = 0.999 eps_start = 1 eps_end = 0.01 eps_decay = 0.001 target_update = 10 memory_size = 100000 lr = 0.00001	**Configuration 1 - 6'000 episodes** batch_size = 256 gamma = 0.999 eps_start = 1 eps_end = 0.01 eps_decay = 0.001 target_update = 10 memory_size = 100000 lr = 0.001
Configuration 2 - 10'500 episodes batch_size = 256 gamma = 0.999 eps_start = 1 eps_end = 0.01 eps_decay = 0.001 target_update = 10 memory_size = 100000 lr = 0.000001	**Configuration 3 - 12'000 episodes** batch_size = 128 gamma = 0.999 eps_start = 1 eps_end = 0.01 eps_decay = 0.001 target_update = 10 memory_size = 100000
Configuration 4 - 12'000 episodes batch_size = 64 gamma = 0.999 eps_start = 1 eps_end = 0.01 eps_decay = 0.001 target_update = 10 memory_size = 100000 lr = 0.00001	**Configuration 5 - 9'000 episodes** batch_size = 256 gamma = 0.998 eps_start = 1 eps_end = 0.01 eps_decay = 0.001 target_update = 10 memory_size = 100000 lr = 0.00001
Configuration 6 - 12'000 episodes batch_size = 256 gamma = 0.995 eps_start = 1 eps_end = 0.01 eps_decay = 0.001 target_update = 10 memory_size = 100000 lr = 0.00001	

Table 2: Parameters used for each of the models. Highlighted in yellow are the hyperparameters we changed from the base configuration.

4.3. Results and discussion

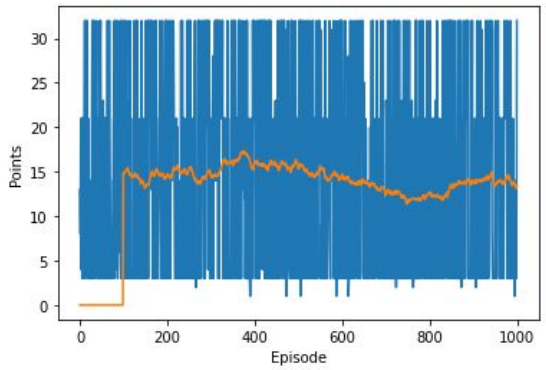
After training the models, we played one thousand episodes of Breakout with each of the trained models. The results are summarized in the below table.

Metric/Configuration	0	1	2	3	4	5	6
Maximum score	32	10	30	38	20	20	271
Minimum score	1	0	0	1	1	2	1
Median score	13.5	2	3	16	5	6	23
Average score	14.26	2.23	4.7	17.35	7.64	7.31	65.832

Table 3: Scores after playing 1'000 episodes.

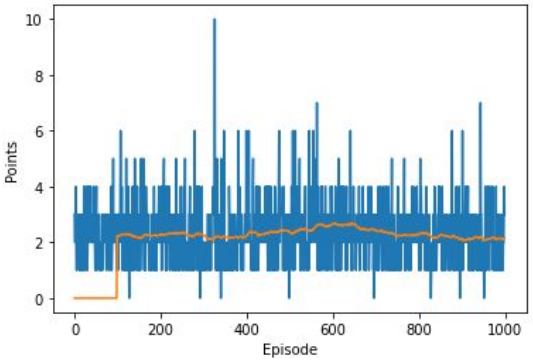
Figures 10 to 16 below show the scores and the 100 episode moving average plotted for 1’000 episodes played with each configuration.

Figure 10: configuration 0



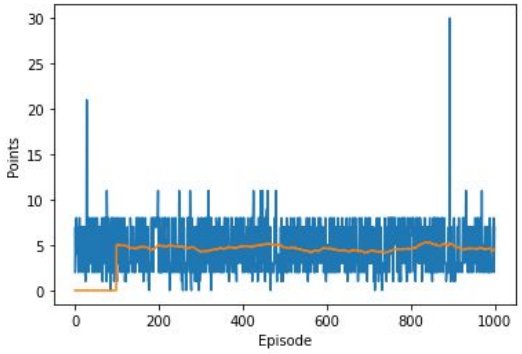

```
Episode 1000
 100 episode moving avg: 13.43
Max score:  32.0
Min score:  1.0
Median score:  13.5
Average score:  14.259
```

Figure 11: configuration 1



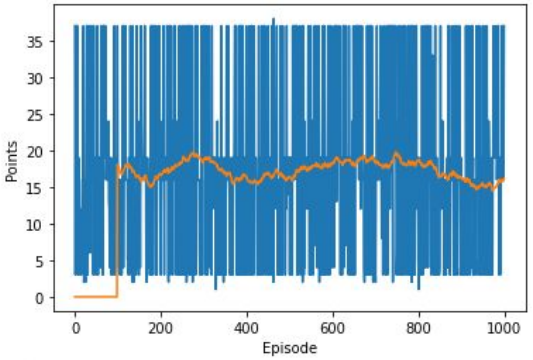

```
Episode 1000
 100 episode moving avg: 2.15
Max score:  10.0
Min score:  0.0
Median score:  2.0
Average score:  2.299
```

Figure 12: configuration 2



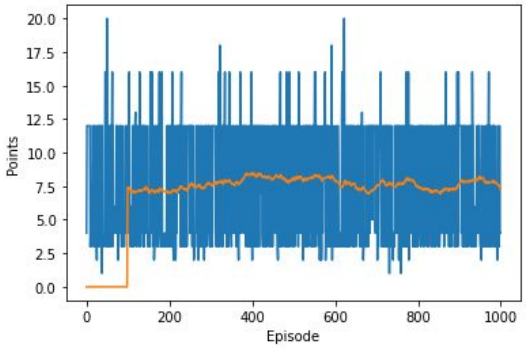

```
Episode 1000
 100 episode moving avg: 4.52
Max score:  30.0
Min score:  0.0
Median score:  3.0
Average score:  4.7
```

Figure 13: configuration 3



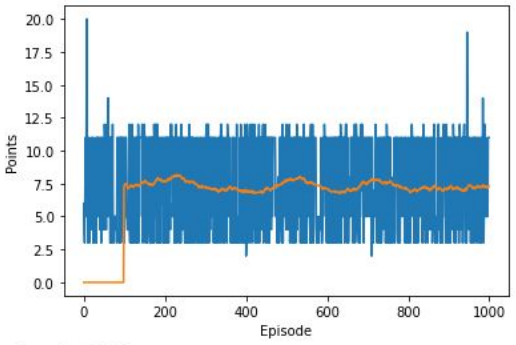

```
Episode 1000
 100 episode moving avg: 16.15
Max score:  38.0
Min score:  1.0
Median score:  16.0
Average score:  17.348
```

Figure 14: configuration 4



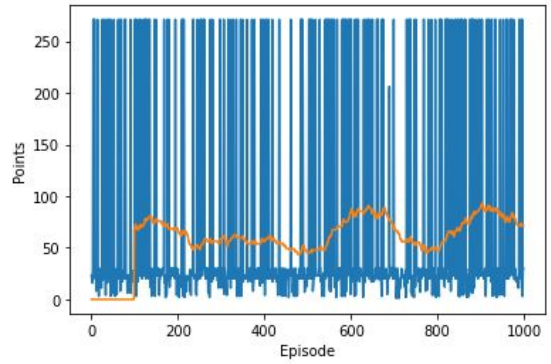

```
Episode 1000
 100 episode moving avg: 7.32
Max score:  20.0
Min score:  1.0
Median score:  5.0
Average score:  7.636
```

Figure 15: configuration 5




```
Episode 1000
 100 episode moving avg: 7.34
Max score:  20.0
Min score:  2.0
Median score:  6.0
Average score:  7.309
```

Figure 16: configuration 6




```
Episode 1000
 100 episode moving avg: 71.16
Max score:  271.0
Min score:  1.0
Median score:  23.0
Average score:  65.832
```

We compare how models change in their performance when tweaking single hyperparameters. In addition to our baseline model, we change three hyperparameters and test two different values, resulting in a total of seven model variations, which we then let play one thousand episodes. The final results of these episodes is then used for comparison.

We define model Configuration0 as our baseline model. Configuration0 has been trained for 26'000 episodes and achieved a max. score of 30, and never scored less than 1. Furthermore, the model reaches a median score of 13.5 while the average score sits at 14.269. This means that it obtains more points in the lower half bound of the point range.

4.3.1. Adjusting learning rate

We start by altering the learning rate. The learning rate specifies how much the weights are updated at each step, i.e. how quickly the model adapts to the problem. A high learning rate can cause the model to converge too quickly on a suboptimal solution. We adjust the learning rate for both models, Configuration1 and Configuration2 to 0.001 and 0.000001. We can clearly see that both variations perform much worse than the baseline model, in fact, their agents are the only ones that score zero points. Configuration1 presents the flattest performance. The agent scores values between zero and ten while reaching an average score of 2.23. Knowing this, we can read from figure 11 that therefore, most points have been scored between one and three. For Configuration2, figure 12 shows the agent achieves an average score of 4.7 and a median score of three which is slightly better than Configuration1. Moreover, the maximum score the agent reaches is 30 points. However, in contrast to Configuration0, it doesn't attain it on a regular basis.

4.3.2. Adjusting batch size

In comparison to the baseline model, Configuration3 and Configuration4 vary in their batch size. Both models possess smaller batch sizes with values of 128 for Configuration3 and 64 for Configuration4. Further, both models have been equally trained for a total of 12'000 episodes. The results, however, are counterintuitive. On the one hand, the performance of the agent for Configuration3 improves solidly as it realizes an average score of 17.348, while the median score of 16 indicates that the

points are slightly negatively skewed. In addition, the chart shows that the agent regularly reached the maximum score of 38. However, on the other hand, the agent of Configuration4 performs worse than the baseline model. Its maximum score is only 20 while the median score is 5 which, in fact, is less than half of the median score the baseline agent attains.

4.3.3. Adjusting the gamma

In the last two versions, we adjust gamma, which is the rate of discount of future rewards. We lower the gamma to values of 0.998 for Configuration5 and 0.995 for Configuration6. We see that Configuration5 does worse than the baseline model as it achieves more or less about half of the median and average score, however, it is the only model that never scores less than two points. Yet, Configuration6 is clearly the best performing model so far. By scoring a max of 271, reaching over 200 points frequently, and having a median and average score of 23 and 65.832 respectively, the agent outshines every other agent's results significantly.

4.3.4. Discussion

According to the results, we see that the configuration with a batch size of 128 performs better than higher or lower sizes. Since batch sizes are highly linked with the learning rate, we can draw the conclusion that this might be the optimal batch size for a learning rate of 0.00001. Bigger batch sizes should enable larger learning rates, as the gradient updates are of higher quality. This is because bigger batch sizes mean more confidence in the direction of the descent of the error surface. While on the other hand, the smaller a batch size the more it turns to a rather stochastic descent, as it follows smaller steps. Put differently, this means that a larger learning rate would turn out to be better for a larger batch size such as 256. However, when we increase the learning rate to 0.001 in Configuration3 in combination with a batch size of 256 the performance doesn't improve, on the contrary, it gets worse. The learning rate may be too large, causing the model to converge too rapidly to a suboptimal solution. Furthermore, Configuration1 with a 100 times smaller learning rate delivers better performance than Configuration2. This hints that the optimal learning rate for the batch size of 256 doesn't necessarily need to be larger than in the baseline model. Due to this divergent outcome, to find the optimal learning rate for certain batch sizes we would have to examine the relationship between the loss and the learning rate and batch sizes.

Nevertheless, the results suggest that tweaking the gamma value in the baseline model has a much higher positive effect on outcome. Gamma represents the discount value in the Bellman equation and is therefore important for calculating target q values. For illustration, as gamma approaches 1 the agent takes future rewards into account more strongly, in other words, the agent becomes more farsighted. This could lead to a policy where the agent might exploit mediocre but stable behaviour of realizing medium amounts of points for endless episodes. However, as Configuration6 possesses a gamma rate of 0.995 future rewards are discounted more heavily, leading to a more shortsighted approach.

Caveats

The results have to be taken with caution as there are several caveats to take into account. The main point is the difference in the number of episodes with which these models have been trained. Due to computation power and time the number of episodes vary for most models, thus no exact comparison can be made. On the other hand, different models can have different optimal training episodes. For example,

Configuration3 and Configuration4 did very well during the first 2000 episodes of training and then its performance began to drop. This means that the agents forget what they have learnt and start to follow poor quality policies. This is where one could consider early stopping as a possible method for mitigating such “forgetting risks". In short, stop training as soon as good consistent performance has been achieved. If this is implemented, results could be compared for agents in a potential top performance training state.

4.4. Possible improvements

To find better performing models, as a next step, experimenting with lower gamma values could be conducted. In addition, the approach in finding optimal learning rates and batch sizes by minimizing loss can complement these new models. Due to time constraints, we didn’t try different variations of all of the parameters. In the future, it would be sensible to experiment with epsilon decay and the final epsilon value, the memory size and the target update parameters and see how they affect the model’s performance. Lastly, depending on the configuration, early stopping could be implemented to prevent a model’s performance from decreasing.

5. Guide to the submitted files

Our submission file contains the following:

1. **Breakout_with_DQN_train**
   A Google Collab file with the code for training the models. The Google Collab files interact with Google Drive and save/load models directly from there. A folder called ‘models’ has to be created in GoogleDrive’s root folder.
2. **Breakout_with_DQN_play**
   A Google Collab file with the code for simulating the computer playing Breakout without updating the network
3. **Breakout_with_DQN_render**
   A Jupyter Notebook file with the code for rendering one episode of the game on the screen. The dependencies listed in the first cell of the Google Collab files should be installed on the local machine. The notebook loads the file directly from the models folder in the submission folder. Currently, it’s set up to run configuration 6.
4. **Models folder**
   A folder with the Breakout models for each configuration, from which all three python files load the models
5. **Mountain Car folder**
   A file ‘All mountain car’ - Jupyter Notebook with all 7 configurations.
   A file ‘best mountain car’ a Jupyter Notebook which contains the code for configuration 7 - the one we deem the best.
6. **Write-up PDF**

6. Bibliography

[1] Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence* Research, 47, 253-279.

[2] Breakout (video game), www.wikipedia.com, https://en.wikipedia.org/wiki/Breakout_(video_game, retrieved in December 2020

[3] Earle, R. (2020). Slides for the Reinforcement Learning seminar at the University of Zurich.

[4] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

[5] Reinforcement Learning - Goal Oriented Intelligence, www.deeplizard.com, https://deeplizard.com/learn/video/nyjbcRQ-uQ8, retrieved in November 2020

[6] OpenAI Gym, https://gym.openai.com/

[7] Stella: A Multi-Platform Atari 2600 VCS emulator, https://stella-emu.github.io/

[8] Q-Learning introduction and Q Table - Reinforcement Learning w/ Python Tutorial p.1, www.pythonprogramming.net, https://pythonprogramming.net/q-learning-reinforcement-learning-python-tutorial/, retrieved in November 2020