APML Final Project

Alon Netser I.D. 31160253-6
Rhea Chowers I.D. 20415064-3

February 2020

1 State Representation

As mentioned before, the objective of our agent is a variant of learning which action to take given a state. In the case of Q-Learning the objective is assigning the value of future expected reward to each possible action given a state (and then of course choosing the action which maximizes the reward). Therefore, we need to determine how to represent a state.
Naively, the state fed to the agent could be the entire game board. Additionally, the location and direction of the agent could also be added to the state. We thought this is not a good choice due to two main reasons:

1. In order to maintain reasonable response rates of the agent, we couldn't use a deep neural network with too many parameters. Since the number of parameters is correlated to the input size, and since the size of the board could vary between 20x20 and 150x150, we couldn't feed the entire board as a state in the sake of performance time.

2. Secondly, not all the information on the board is relevant. Far away objects return sparser rewards.

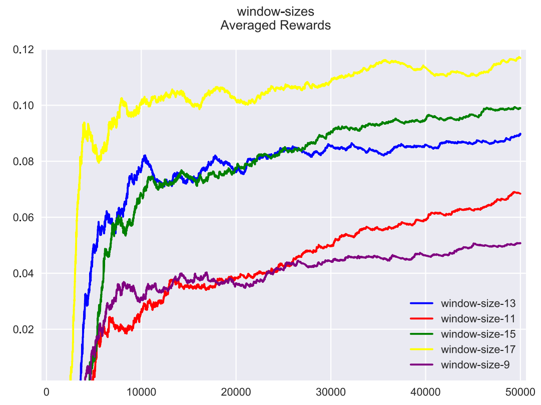
Therefore we decided to represent a state using a fixed-size crop of the board, where the window is taken around the snake's head.

In order to make our representation take the direction of the head into account (which is crucial for the learning to succeed) we rotated the window so that the direction of the snake's head is always up. This means that if the snake's head's direction is west, we rotate the window clock-wise 90 degrees, if it's east we rotate counter-clock-wise, if it's south we rotate 180 degrees, and if it's north we do not rotate.

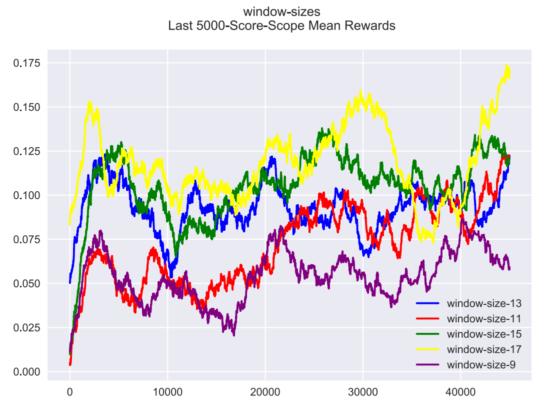
Another issue we had to consider was the representation of objects on the board which we have no prior knowledge about. In order to handle this we decided to divide the window into 11 boolean windows, each representing the location of an object in the window. For example, object 4 had the 4'th boolean window in the window stack. That way the agent could learn the correlation between a window and rewards it get.

Figure 1 shows the affect of using different window-sizes in our agent, showing that the performance is increased with the window-size. The final value we chose for the window-size is 17, since we had to take run-time considerations into account. Note that the maximal possible value is 19, because the minimal board-size is 20×20, as stated in the project instructions, and we want the window-size to be an odd number to enable putting the head in the middle. Therefore the choice of 17 is almost the largest window we can take.

Note that we had another idea which was to represent the board cells that are reachable in n steps (n can be whatever we want) and stack them to a vector (containing the same boolean one-hot vectors describing what's in the cell, as the 11 channels in the window above). However, we didn't choose this idea because it will not allow us to use convolutional neural-networks. We could have mask out the cells in the window that are not reachable by at most n steps (e.g. cells "behind" the snake will take more steps to reach), but we didn't find it logical to mask out information that is already being represented (since we use a square window). The representation we chose worked great, both with the linear learner and with the DQN (and its variants).



(a) Different window-sizes averaged rewards.



(b) Different window-sizes last score-scope (5000) rewards.

Figure 1: As seen clearly in these graphs, the averaged and last-score-scope rewards increase as the window-size increases.

2 Agents

2.1 Linear Q-Learning

As suggested, we started by making the Linear Q-Learning agent work well (as much as this quite weak representation enables). We chose to implement this agent using NumPy alone, and not by using a Keras model (consisting of a single "Dense" layer). The reason behind this choice was to better understand the overall flow, and the fact that the gradients of the Linear Q-function are pretty simple.
Denote the dimension of the state as $d = (\text{window-size})^2 \cdot (\#\text{values})$. We approximated the Q-function $Q(s, a)$ using a linear-function $Q(s, a) \approx Q(s, a; \theta)$ where the parameters θ include a weight-matrix $W \in \mathbb{R}^{3\times d}$ and a bias-vector $b \in \mathbb{R}^3$ (the 3 corresponds to the number of actions). Given a state s we can compute all relevant state-action values using a single forward-pass (similar to the ways it's done in the DQN model), so we get that $Q(s, a; \theta) = (W \cdot \phi(s) + b)_a$ (we assume that the actions a are integers between 0 and 2, and $\phi(s)$ is our state-representation).

The gradient-update defined by an experience (s_t, a, r, s_{t+1}) with respect to the weight matrix is a matrix with the same shape as W (obviously) which is all zeros except the a-th row which equals $\phi(s)$. The gradient with respect to the bias-vector is simply 1. In total, the gradient-step equals $W \leftarrow W - \eta \cdot \delta(s_t, a, r, s_{t+1}; \theta) \cdot \phi(s)$, and $b \leftarrow b - \eta \cdot \delta(s_t, a, r, s_{t+1}; \theta)$, where $\delta(s_t, a, r, s_{t+1}; \theta)$ is the temporal-difference-error which is defined as $\delta(s_t, a_t, r_t, s_{t+1}; \theta) = Q(s_t, a_t; \theta) - (r_t + \gamma \cdot \max_{a'} Q(s_{t+1}, a'; \theta))$ (following slide 48 in Shai's presentation).

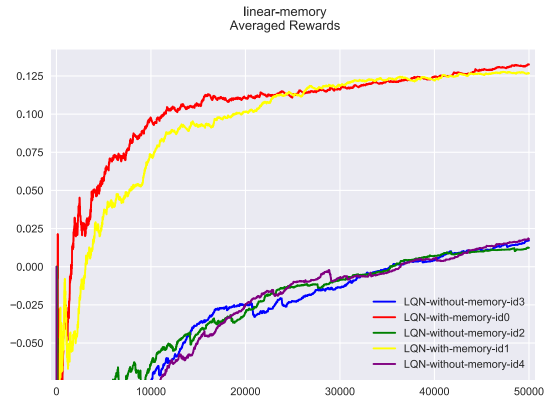
2.1.1 Linear-Q-Learning with memory

Preparing for the DQN, we tried to implement a Replay-Memory in the Linear-Q-Learning. Then, we sample mini-batches of experiences instead of just the last one. This improved the performance of our agent greatly.

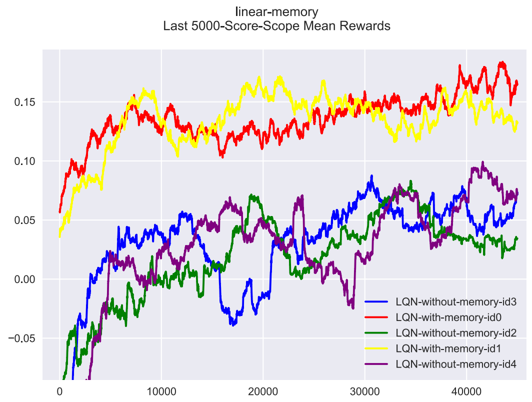
Figure 2 shows the affect of the memory on the linear q-learning agent.

2.2 Deep-Q-Network

We followed the work done in [1], approximating the Q-function using a deep convolutional neural-network. We used the same architecture, but we changed the width of the network to match the required running-time of our environment. Therefore, the network consists of 2 convolutional layers followed by a ReLU activation, with kernel-size 3 and increasing number of channels 16, 32. The first convolution layer is with stride 2 to down-sample the input and increase efficiency. Then there is one fully-connected layer containing 64 channels (followed by a ReLU activation), and finally the fully-connected layer containing 3 channels, one for each action.



(a) Linear agents with and without memory - averaged rewards.



(b) Linear agents with and without memory - last score-scope (5000) rewards.

Figure 2: As seen clearly in these graphs, the performance of the linear agents dramatically improve when using a replay-memory and not updating only on the last experience. Note that all agents were trained using exactly the same hyper-parameters, the only difference was whether a memory was used or not.

Following the work done in [1], we used a Replay-Memory holding the last 1000 experiences and each learning-step is done by sampling uniformly a mini-batch of 32 experiences. We also used two networks in the model - one is the Q-network and the other is the target-network. The target-network use the same architecture as the Q-network, and its weights are copied from the Q-network every 100 iterations.

Note that we have tried a lot of different architectures, but due to the quite harsh time limits on the school's cluster, we had to use a quite small architecture. Using larger models give slightly better results, but we reduced the width and depth to almost did not decreases the performance.

2.2.1 Deep Reinforcement Learning with Double Q-learning

Following the work done in [2], we implemented Double Q-learning algorithm.

Standard Q-Learning is known to overestimate action values under certain conditions, because the labels for an experience (s, a, r, s') is defined as $y^{DQN} = r + \gamma \cdot \max_{a'} Q(s', a'; \theta^{-})$ and it uses the same function (the target-network) both for choosing the action which maximizes the expected future reward, and for evaluation what this future reward is. The solution offered in [2] is to choose the action which maximizes the future reward according to the Q-network, and evaluate the future reward using the target-network. Thus, the label is changed from y^{DQN} to $y^{DDQN} = r + \gamma \cdot Q(s', arg\max_{a'} Q(s', a'; \theta); \theta^{-})$.

This change to the learning-algorithm reached a new state-of-the-art on the Atari domain, and it improves the stability of the convergence as well as the quality of the final learned policy.

2.2.2 Prioritized Experience Replay

Initially we followed the work done in [1], by sampling from the Experience-Memory uniformly at random during training, which is the naive approach to use the memory and seems not optimal. As the authors of [1] stated as "future work", sampling from the memory can be done more efficiently.

To address this issue we first tried to change each experience probability according to its reward. The first experiment was to sample less (or not at all) from experiences with negative reward. This caused severe degradation in the performance, which seems reasonable - we need to learn from mistakes, not just from successes. The second experiment was to sample less (or not at all) from experiences with zero reward. This seemed more reasonable, because most of the time the snake is moving and not getting any reward, and we would like to sample less from these experiences. This also caused degradation in the performance (although it performed better than the previous method).

It make sense that the above two methods for sampling "more efficiently" from the memory did not work. Indeed, there is more to an experience than its reward! For example, the snake might not get any reward in a particular state, but there might be many good fruits ahead (in the window, so the snake "sees" them). It also might be the case where the snake is going to a dead-end (full of obstacles), or to a zone without any potential reward, and these states are also important for learning.

Therefore, we followed the work done in [3] and implemented a Prioritized-Experience-Replay. Briefly, this means that each experience $e_i = (s_i, a_i, r_i, s'_i)$ has a priority $p_i > 0$ which is defined as the temporal-difference error δ_i plus a small positive ϵ. Every time an experience is sampled from the memory its priority is updated $p_i \leftarrow \delta_i + \epsilon$, to match the current TD-error of the model (the δ_i change during training, because the weights are changing, and the epsilon is there to ensure the experiences will always have a positive sampling probability, even where their TD-error will be close to zero).

The Prioritized-Experience-Replay has additional hyper-parameters $\alpha, \beta \in [0,1]$, which we set according to [3] to be $\alpha = 0.6, \beta = 0.4$. The probability of an experience is defined as $\Pr[i] = \frac{p_i^\alpha}{\sum_j p_j^\alpha}$ so α controls how much prioritization is done (note that $\alpha = 0$ means no prioritization at all - the probabilities will be uniform).

Another issue introduced by the non-uniform distribution among experiences is that the gradient estimator used in the SGD is biased. This is fixed by using "Importance-Sampling" (or "sample-weight" in keras). Each experience in the mini-batch has a weight which is defined as $w_i = (\frac{1}{N} \cdot \frac{1}{\Pr[i]})^\beta$ that fully compensates for the non-uniform probabilities $\Pr[i]$ if $\beta = 1$. For stability reasons, the weights are normalized by $\frac{1}{\max_i w_i}$ so that they only scale the update downwards. Followed the ideas in [3] the hyper-parameter β is annealed linearly from his initial value (0.4) in the beginning of the training to 1 in the end (as the unbiased nature of the updates is most important near convergence at the end of training).

2.2.3 Dueling Network Architectures for Deep Reinforcement Learning

Following the work done in [4], we implemented the Dueling Network architecture.

In many games, given certain states some actions taken do not have an effect on the future reward. For example, in an empty board, it doesn't to which direction the snake turns and therefore the empty board state does not contain any valuable information on the required action. In order to handle this issue, the Dueling Network architecture takes a series of convolutional layers and splits their output into two streams - one of them learning the value function and the other the advantage function. Their outputs are used to calculate the network output. This is demonstrated in the following figure:

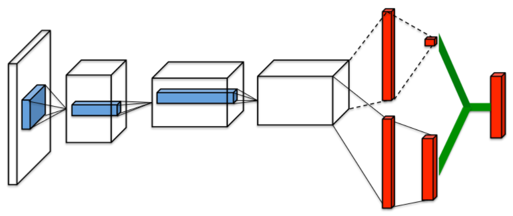


Figure 3: Dueling Network Architecture - after a series of convolutional layers the network splits into two streams, one for the value function which produces a scalar and one for the advantage function which produces an ouput with the dimension of the action space. These are combines to calculate the Q function.

Recall:

$$\begin{aligned} A(s,a) = Q(s,a) - V(s) \Leftrightarrow Q(s,a) = V(s) + A(s,a) \\ \Rightarrow Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\alpha) + A(s,a;\theta,\beta) \end{aligned}$$

The Dueling network learns the weight given to each of the value and advantage functions given a certain state. In other words it learns if it should prioritize the advantage function, or choosing a certain action, or if it should prioritize the value function, or the future reward disregarding the immediate action.

In practice, an adjusted formula for $Q(s, a)$ was used in the network to address issues of identifiability (recovering V and A from Q):

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \alpha) + (A(s, a; \theta, \beta) - \overline{A}(s, a; \theta, \beta))$$

Where $\overline{A}(s, a; \theta, \beta)$ is the mean of the advantage function over all actions. It is clear that in this formula, if the state has no information on the action the agent should choose, we would want the advantage to be equal for all actions, and therefore get $A(s, a; \theta, \beta) - \overline{A}(s, a; \theta, \beta) = 0$, prioritizing the value function instead. This also works better in practice.

3 Additional Experiments

3.1 Reward killing other snakes

Our motivation in this experiment was to encourage our snake to kill other snakes (for example going around them and "locking them up", forcing them to crash into his tail).

In every experience where there is a switch from the previous state to the next one, we took a window around the snake's head in both states (at a fixed location, e.g. its previous state location). Then we checked the differences between these two windows, trying to find a relatively long chain of 1's. A relatively long chain of 1's can happen only if a snake died, since other differences (such as fruits appearing randomly on the board, snakes moving, etc) can only cause single cells to be changed, but not a long chain of cells.

The challenge was to make this work fast enough to now slow us down, since this happens in every 'act' function (where we store the experience in the memory). Taking the window and calculating the differences was done in a vectorized way using NumPy. Then, we found the 1's in a vectorized way using np.where. From this point we treated the boolean matrix as a graph, where the 1's represents the vertices, and two vertices are connected if they are left/right/bottom/top of each other. We removed isolated vertices using boolean vectorized 'and' operation with the np.roll of the window to each of the 4 directions. Then, we are left with not a large number of 1's and we perform DFS to find whether a connected component of size greater than 3 is found.

This implementation lacks the information that this snake is not our snake itself, and indeed training using this implementation caused our snake to "commit suicides". We fixed this by removing from the differences matrix every cell that is equal to our id. We further improved this method by checking if the snake that died touched our snake, since we saw that sometimes another snake dies around our snake's head, but it's not because he crashed upon our snake, but because he crashed into something else.

This reward worked, in the sense that the trained snake killed more snakes. However, it did not contribute to the overall score, so we left this functionality off. We conjecture this is because eating good fruits is much more easier that killing other snakes, and we did not find the right weights between the two objectives.

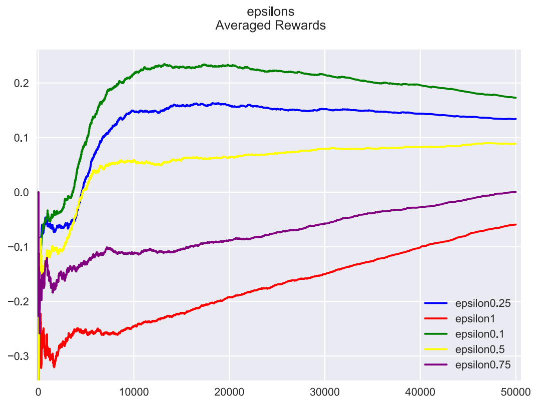
3.2 Exploration-Exploitation Trade-Of

We handled the exploration-exploitation trade-of by following an ϵ-greedy policy during act time, where ϵ decay linearly from a quite large value to 0 (in the beginning of the last score-scope iteration). We tried many different values for ϵ, and looked at the graph of the rewards. As expected, the larger the epsilon is the lower cumulative reward is gained, but the final policy in the last score-scope iterations is better. The final value for the initial ϵ was chosen to be 0.5.

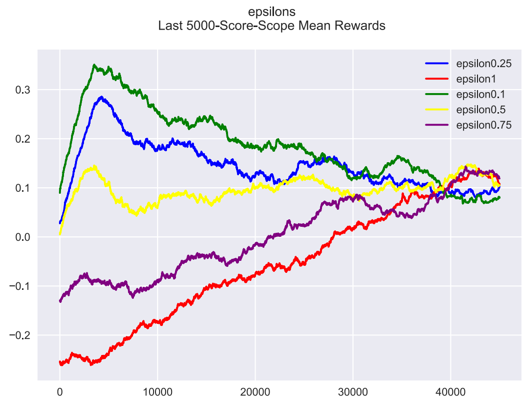
Figure 4 shows the affect of different initial epsilons on the reward achieved by the learned policy.

3.3 Decaying the learning-rate

Since the rewards graph are quite non-stable (in the end of the game they goes up and down), we wanted to lower the learning-rate as the learning progresses. In order to do so, we used Adam optimizer with a piece-wise constant learning-rate. The learning-rate begins with a value of 0.0005 and multiplied by half twice during the learning (equal length intervals). These hyper-parameters was chosen by using an informal grid-search and choosing the ones that reached the highest reward.



(a) Averaged reward - in iteration t the graph shows the averaged reward in the first t iterations.



(b) Last score-scope (5000) reward - in iteration t the graph shows the averaged reward in the last 5000 iterations.

Figure 4: As seen clearly in these graphs, the averaged reward for lower epsilons is better, because they accumulate more rewards in the beginning of the game (because they explore less). However, in the final iterations the exploration pays off, and we see that the larger the exploration is that larger the final reward.

3.4 Symmetric states

In order to enrich the amount of different states the agent sees, especially in early stages of the learning we took advantage of different symmetries in the game. For every triplet of states and action (s_t, a, s_{t+1}), we added at least one symmetric state to the replay memory. Since our windows where always aligned such that the agent was looking to the north, taking a left or right turn was the same as taking the opposite direction in the mirror of the same window. In a similar manner, taking the forward action was the same as taking left or right in a rotation of the same window. Even though this multiplies the replay memory, we didn't find it contributed to the learning process as can be seen in the following figure.

3.5 Evaluating our agents

At first, we trained the agents and watched them playing, because we wanted to see it they learn anything. After we saw that the learning was successful, we had to come up with a more sophisticated way of comparing the performance of several agents (besides the final scores). Therefore, we dumped the rewards the agents received during training, and plotted graphs of the averaged rewards, as well as the last score-scope (which was 5,000) average rewards. This enabled us to see the speed of convergence, and the stability of the learning process. All of the figures in this paper we extracted using this method.

3.6 Network Parameters

Our original architecture consisted of 3 convolutional layers and two affine layers as can be described in section 2.2. In order to maintain the runtime dictated for the project, changes were to be made. We identified several parameters that could be changed in order to improve the agents response time. These were the batch size, window size, network depth, the size of each layer (number of filters) and adding pooling layers (or strided convolutions) in order to decrease the dimensions through the network.

We noticed was that the batch size deeply effected performance and runtime (see Figure 5). Down-sampling the input by using a strided convolution enabled us to increase the number of channels and it also performed great. We tried many different hyper-parameters defining the network's architecture, and we think we reached a nice spot. Of course, further hyper-parameters search will probably be helpful, be we had to stop after a certain number of days spent on this.

3.7 DQN Variants

As mentioned before, we implemented three papers written by the same authors as DQN (DeepMind). These were Double Q-learning algorithm, Prioritized Experience Replay and a Dueling Network. These improved the state-of-the-art on the Atari domain, but in our snake environment they did not help so much. We tried many sessions with different hyper-parameters, and we saw that these variants did not decrease the performance, and the rewards and more stable, so we decided to submit our final agent with these variant turned on.

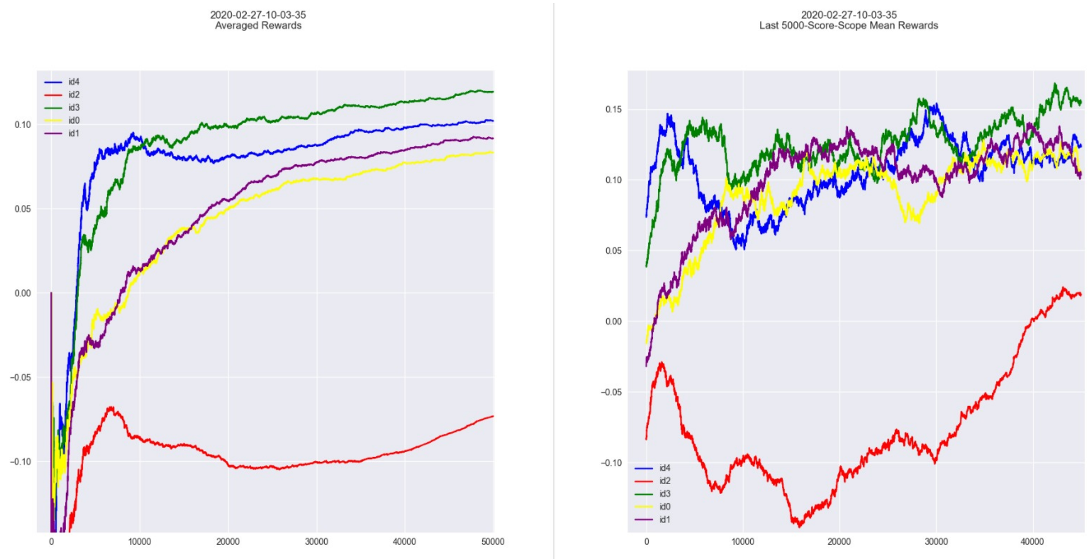


Figure 5: In red we can see an agent with a batch size of 16, while the rest are with batch size of 32.

References

[1] Mnih, Volodymyr, et al. ”Human-level control through deep reinforcement learning.” Nature 518.7540 (2015): 529-533.
[2] Van Hasselt, Hado, Arthur Guez, and David Silver. ”Deep reinforcement learning with double q-learning.” Thirtieth AAAI conference on artificial intelligence. 2016.
[3] Schaul, Tom, et al. ”Prioritized experience replay.” arXiv preprint arXiv:1511.05952 (2015).
[4] Wang, Ziyu, et al. ”Dueling network architectures for deep reinforcement learning.” arXiv preprint arXiv:1511.06581 (2015).