Using DQN Agents to play Battleship

Niklas Z.
University of California, Los Angeles
niklasz@g.ucla.edu

Abstract

In this project, we developed an AI agent to play the game Battleship as the attacker. The agent is implemented using a greedy epsilon algorithm, with a Deep Q Network (DQN) as a policy. Using this architecture, the agent manages to outperform random and naive agents, but falls short when compared with state-of-the-art deterministic agents.

1. Introduction

1.1. Battleship

Battleship is a 2-player pen-and-paper game, said to have been invented in the 1930s. Each player has a 10×10 grid on which they place ships, hidden from the other player's viewer. The winner is the one who finds and sinks their opponent's ships the fastest. The game is played as follows:

1. Players 1 and 2 place their 5 ships of lengths 2,3,3,4 and 5 (width 1) on their respective grids. Ships may be placed vertically or horizontally, but may not intersect and have to fit onto the grid.
2. Player 1 fires a missile onto a coordinate (a, b) (e.g $(4, 1)$) of Player 2's grid. Player 2 announces whether this misses, hits or sinks a ship.
3. Player 2 performs the same actions vice-versa with Player 1 in step 2.
4. Steps 2. and 3. are repeated until either Player 1 or Player 2 has had all of their ships sunk.

An example of the game state is given in figure 1.1.

1.2. Reinforcement Learning

From a reinforcement learning perspective, we can see Battleship as a 2 agent game: an attacker (who sinks ships) and a defender (who places them at the start). We will consider only the attacking agent in this project. The environment obeys the *Markov Property* and is given by the 10×10 grid, where each cell can be in one of 4 states: `unknown`, `hit`, `missed` or `sunk`. During each step of an episode (game) an attacker can fire at any of the 100 cells, even if it is not in an `unknown` state (this will be considered an illegal action). This means that there are $4^{100} \approx 10^{60}$ possible game states (although not all of them are legal, a more reasonable lower bound would be $2^{100} \approx 10^{30}$), where an agent can take 100 actions per state. The reward for each action given cell state is listed in Table 1.2. From it, we can tell that a perfect game will yield a *return* (total reward) of 185. That said, a simpler metric to compare agents is usually the number of steps needed to complete the episode, which can range from 17 to 100 (where the episode automatically ends past 100).

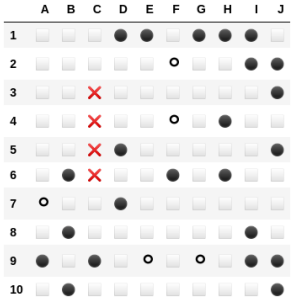


Figure 1: Grid visible to attacking player

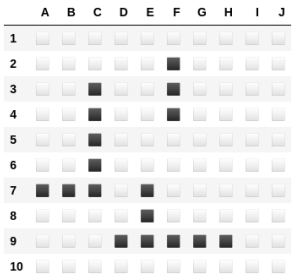


Figure 2: Grid visible to defending player

Action a	**Reward** r
Missing a shot	-1
Hitting a ship	+5
Repeated fire at a cell of known state	-10
Winning the game	100

Table 1: Battleship environment rewards for each action

1.3. Deep Q Network

Q-learning uses a state-action value function $Q^{\pi}(s, a)$ which measures the expected return of taking some action a in state s and then following policy π thereafter. In order to obtain an optimal policy $Q^{*}(s, a)$ we iteratively update $Q_{i+1}(s, a) \leftarrow \mathbb{E}[r + \gamma \max_{a'} Q_i(s', a')|s, a]$ which is the

expectation of the reward r and some γ-discounted future return of choosing the policy-optimal action a' (*Bellman Equation*). This can be shown to become the optimal policy $Q^*(s, a)$ as $i \to \infty$ [8].

A common implementation of Q is a lookup table containing the value of every possible state-action pair. However, considering earlier estimates, it is not really viable to keep a table of 10^{32} entries. DQN uses a neural network with parameters θ to approximate this $Q(s, a; \theta) \approx Q^*(s, a)$ [6]. This means our Q-net will take in some state s and output scores $\{q_1 ... q_{100}\}$ for each possible action $\{a_1, ..a_{100}\}$. We train the Q-net by taking the mean-squared error at each step i:

$$L(\theta_i) = \mathbb{E}[(y_i - Q(s, a; \theta_i))^2]$$
$$y_i = r + \gamma \max_{a'} Q(s', a'; \theta_i^-)$$

This is also referred to as the *Temporal Difference Error*. Note here, that we keep an older copy of parameters θ_i^- (aka *target network*) which is updated with the weights from θ_i every k training steps. The target network improves training stability [7]. Additionally we will apply soft updates [5] using a learning hyper-parameter $\tau \in (0, 1]$, again to improve stability.

$$\theta_i^- = (1 - \tau)\theta_i^- + \tau\theta_i$$

It's easy to see that subsequent state-action pairs s, a and s', a' in an episode are highly correlated events. To mitigate this, we use a FIFO replay queue [6] to keep track of the D most recent training steps. Each entry in the queue contains a transition $\{r, s, a, s'\}$. We uniformly randomly sample some m transitions from the queue and use them for mini-batch GD training on the Q-net. Aside from training, we follow an ε-greedy policy. For a complete overview, see Algorithm 1.

Algorithm 1 DQN Algorithm

Initialise replay queue R of size D.
Populate R with U transitions from a random agent.
Initialise Q function with random weights θ.
Initialise target $\hat{Q}$ function with $\theta^- = \theta$.
Initialise training step counter $i = 0$
while $i < N$ **do**
  Initialise epoch step counter $t = 0$.
  Initialise starting state $s^{(t)} = s^{(0)}$
  while $s^{(t)}$ is not terminal **do**

$$a^{(t)} = \begin{cases} \text{random action} & \text{w.p } \varepsilon \\ \text{argmax}_a Q(s^{(t)}, a; \theta) & \text{otherwise} \end{cases}$$

    Execute $a^{(t)}$ and observe reward $r^{(t)}$ and $s^{(t+1)}$
    Append $(r^{(t)}, s^{(t)}, a^{(t)}, s^{(t+1)})$ to R.
    Sample m random transitions from R.
    Perform a gradient descent on $L(\theta_i)$ w.r θ_i.
    $t \leftarrow t + 1$, $i \leftarrow i + 1$
    Every k steps let $\theta_i^- = (1 - \tau)\theta_i^- + \tau\theta_i$.
  end while
end while

2. Results

We measured the performance of 4 agents and compared them with 1 agent from an external work[3]. All of them are summarised in Table 2. Initial training of the DQN algorithm showed that the agent struggled significantly to complete episodes on its own, even when playing randomly. After 800,000 training steps it managed to raise its average return from -944 to -300 (the worst possible return is -1000), where it first completed an episode in 98 steps. It's likely the agent would have improved further, but unfortunately, such a training process takes about half a day and tuning hyper-parameters with such a time scale is simply not viable.

Instead, we opted to introduce an *invalid action mask*, which prevents the agent from executing actions on cells in state `hit`, `missed` or `sunk`. This measure has a significant effect on any agent (compare the random agents) and guarantees that every episode will complete in at most 100 steps without a cut-off point. With the mask, the DQN achieves an average return of 118.3, taking 76.4 steps on average. Comparing this to the deterministic algorithm, which takes 45 steps on average, there is still a lot of room for improvement. Some additional research also suggests that an optimal deterministic algorithm should take $30.8 < s < 55.6$ steps [2], meaning there are likely even better algorithms that widen the gap.

3. Discussion

Here we will discuss some of the more interesting observations and decisions made during the implementation of the DQN algorithm and training.

3.1. Encoding the Input

One of the first issues that needed to be solved, was finding a good representation of the input grid. Naturally a 10×10 grid becomes a matrix of the same size, but choosing suitable values to represent the cell states `unknown`, `hit`, `missed`, `sunk` is not a clear process. Our initial attempts used encodings such as $\{-1, 0, 1, 2\}$, $\{-1, 0, 1, 2\}$ and other variations, but none of them did particularly well against the same Q-net (a CNN). In the end, we opted for a binary encoding composed of a $10 \times 10 \times 4$ tensor. Here, each 10×10 slice represents the following:

1. Whether a missile has been fired at the cell (1) or not

(0).

2. Whether the cell is in a `missed` state (1) or not (0).
3. Whether the cell is in a `hit` state (1) or not (0).
4. Whether the cell is in a `sunk` state (1) or not (0).

Incidentally, swapping the 0 and 1 for the first slice would yield the aforementioned invalid action mask. This new encoding generally performed better with the Q-net. This could be because it separates relevant values into slices, which when convolved in a CNN layer, will be multiplied exclusively with different weights. These weights could then more easily adopt their own case-specific roles. For example, the weights at depth 4 could act as inhibitors when they detect `sunk` ship cells while `hit` weights at depth 3 can be excitatory. This theory could be verified by gathering weight and activation statistics, but there was not enough time to do this.

3.2. Picking Model Parameters

We used a Convolutional Neural Network (CNN) to be the Q function of our agent (architecture shown in Figure 3). It is fairly small, comprising 89,188 parameters. This is likely a contributing factor to the agent's weaker performance. However, adding more filters to the convolution did not improve results and the small size of the input limits the convolutional layer depth (without adding lots of padding). In hindsight, the small size of the input might have made a conventional Fully Connected net a better choice, as there is still a lot of room when it comes to model complexity.

3.3. Measuring Performance during Training

The most interesting and also most vexing part of this project was training in deep reinforcement learning. What is previously learned from supervised learning is not really useful here: our training data is constantly changing and because we are only sampling tiny parts of the game's state-action distribution, said data is also incredibly biased. During training, this has the consequence that loss is not really indicative of how well a model is learning, as the agent could be going through the same states over and over, overfitting them.

The more reliable metric should therefore be an agent's performance such as the average return or step count over multiple episodes. However, even here, sample size is very important. As evaluating 1 episode may take up to 100 steps, it's tempting to draw averages from a low number of episodes such as 5, 10 or 20 in order to keep the cost of evaluation lower than training. We initially used 10 episodes for these averages and found that when we snapshotted promising policies $Q(s, a; \theta)$ with low training step averages (e.g 60 steps), these same policies would perform significantly worse (80 step averages) when evaluated against 100s or 1000s of episodes. In response we adjusted the evaluation episode count to 50, evaluated less frequently and found that this was usually within 1-2 steps of larger samples in the 1000s.

There are likely a lot more problems in training like this, which need to be identified and fine-tuned. They will have to wait for another time.

References

[1] Tf.keras.optimizers.adam tensorflow core v2.8.0. `https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam`.

[2] M. Audinot, F. Bonnet, and S. Viennot. Optimal strategies against a random opponent in battleship. *The 19th Game Programming Workshop 2014*, Jan 2014.

[3] N. Berry. `https://www.datagenetics.com/blog/december32011/`, Dec 2011.

[4] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In Y. W. Teh and D. M. Titterington, editors, *AISTATS*, volume 9 of *JMLR Proceedings*, pages 249–256. JMLR.org, 2010.

[5] T. Kobayashi and W. E. L. Ilboudo. t-soft update of target network for deep reinforcement learning. *CoRR*, abs/2008.10861, 2020.

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

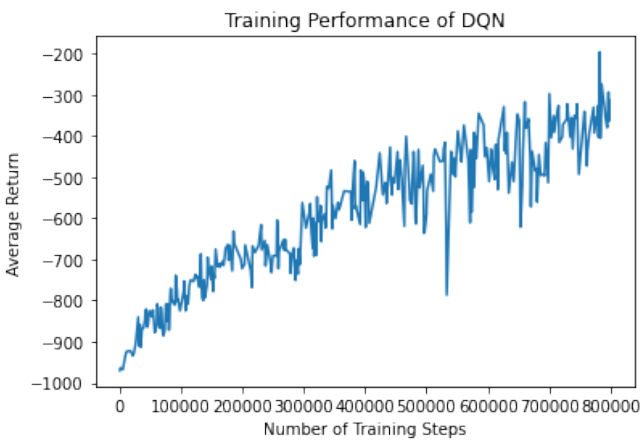
[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb. 2015.

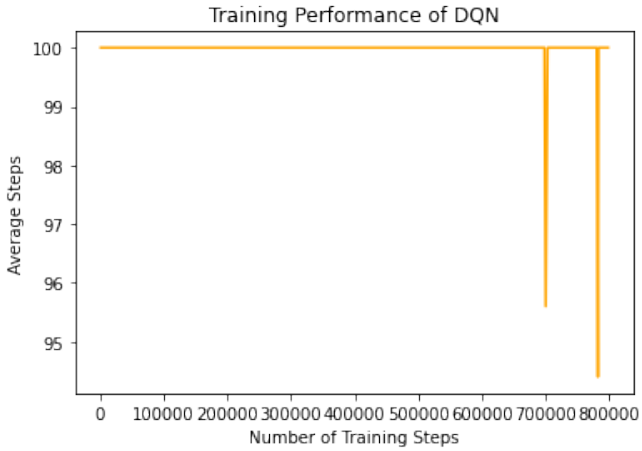
[8] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. The MIT Press, 2020.

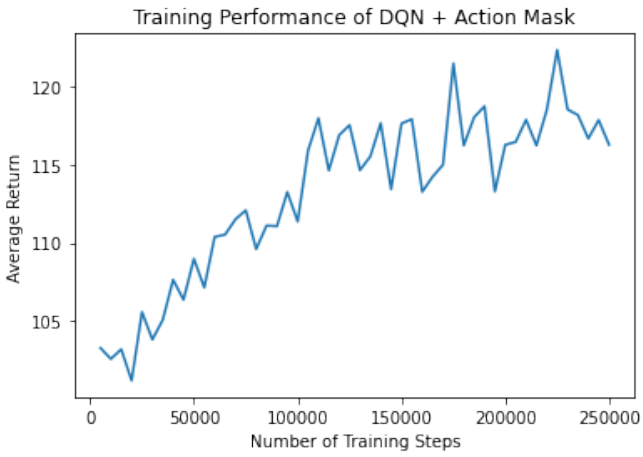
4. Result Tables and Charts

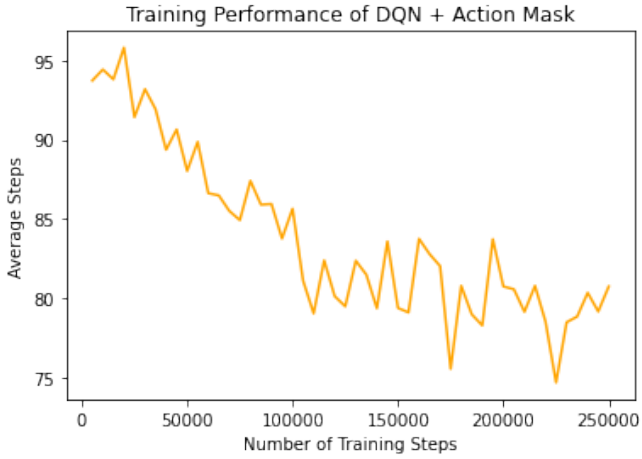
Algorithm Name	Min Steps	Max Steps	Mean Steps	Median Steps
Random Agent	100	100	100	100
Random Agent + Action Mask	55	100	95.3	97
DQN	98	100	99.3	100
DQN + Action Mask	**33**	**100**	**76.4**	**78**
Probability Density Algorithm	17	72	45	42

Table 2: These metrics were sampled over 10,000 games. An optimal play of Battleship would require 17 steps, whereas the worst possible play would take 100.









5. Architecture and Hyper-parameters

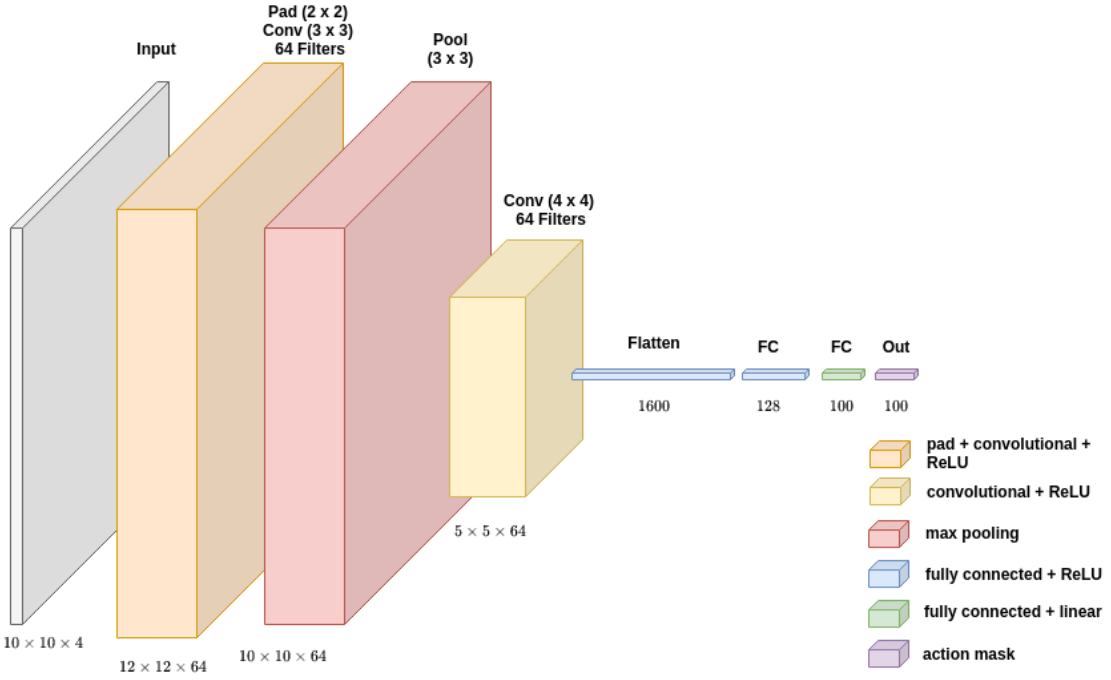


Figure 3: CNN architecture used for the DQN + Action Mask

Hyper-parameter	**Value**	**Notes**
Training Iterations N	250,000	Mostly bounded by machine performance, although this can probably be reduced with further optimisations.
Replay Buffer Length D	2,000	Other work[7] has suggested sizes of up 1,000,000, but in practice larger values have tended to slow down training.
Initially Collected Steps U	500	Used by the random agent to populate R for the first few training steps.
Future Reward Decay γ	0.9	Chosen via hyper-tuning.
Initial Exploration Rate ε Final Exploration Rate	1.0 0.01	This variable was set to linearly decay to the final rate upon reaching the Nth training iteration.
Initial Learning Rate α Decay Factor Decay Every	0.001 0.99 10,000	This variable was set to exponentially decay every number of steps by the factor. The learning rate controls how much the gradient updates the training network.
Training Batch Size m	64	Controls how many samples are taken from R per training step.
Soft Update Learning Rate τ	0.01	Affects how much the of the training network's weights are used to update the target network. As the effect compounds with that of α, they are both kept reasonably high.
Optimiser - Adam with: β_1 β_2 ϵ	 0.9 0.99 1e-7	These are default settings recommended by `tensorflow`[1].
L2 Regularisation α_2	0.01	Also the default from `tensorflow`[1].
Weight Initialisation Method	-	Normalised Xavier with no offset [4].

Table 3: Hyper-parameters set, that are not directly part of the neural network.