

# REINFORCEMENT LEARNING CONNECT FOUR

Hanan Benzion 026606608

Idan Azuri 302867833

January 25, 2018

## 1. PRACTICE RESULTS

**Board representation.** We chose to represent the board as a vector in dimension (batch\_size, 42). The optimal board representation should be smooth so it will be possible to use SGD to optimization for our objective function. Another parameter that we considered is the minimal divergence rate is also possible for all games, but unlike with smoothness is actually practical for a reasonable number of games. Essentially optimality can be achieved by ensuring that any state  $S'$  derivable from state  $S$  has the property that  $|S - S'| = 1$ . In that manner also (6,7) shape would be a good choice, because it has this property. So the consideration of choosing this shape was to simplify our deep neural network and to make an easier representation in our code

**The Algorithm.** The Q-Network is a method based on Q-Learning which is trying to predict the maximum expected cumulative reward for a given a pair(state  $s$ , action  $a$ ). Formally'

$$Q^*(s', a) = \underset{\pi}{MaxE} \left[ \sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi \right]$$

$\pi$  – the policy     $s'$  – next state     $a$  – action     $\gamma$  – uncertainty value

$Q^*$  satisfies the following Bellman equation uncertainty

$$Q^*(s', a) = \mathbb{E}_{s' \sim \varepsilon} [r + \gamma \max_{a'} Q^*(s', a') | s, a]$$

The intuition is, if the optimal state-action values for the next step  $Q^*(s', a)$  are known, then the optimal strategy is to taken the action that maximizes the expected value of  $r + \gamma Q^*(s', a')$ . Now we can not learn the entire space of the Q function,  $S \times A$  states and actions because it is too large  $O(2^{42})$ . So the Deep Q learning is an approximation for this Q function where the Forward pass is defined as follows,

$$(1.1) \quad L_i(\theta_i) = \mathbb{E}_{s, a \sim p()} \left[ (y_i - Q(s, a; \theta_i))^2 \right]$$

where

$$(1.2) \quad y_i = \mathbb{E}_{s^i \sim \varepsilon} \left[ r + \gamma \max_{a'} Q(s', a'; \theta_{i-1} | s, a) \right]$$

and the Backward pass is

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s_i, a \sim p(); s' \sim \varepsilon} [r + \gamma \max_{a'} Q^*(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \nabla_{\theta_i} Q(s, a; \theta_i)]$$

We chose the Deep Q-Network method, our mainly consideration to chose this method was the efficiency of this approach, because it is needed only one feedforward pass to compute the Q-values for all actions from a given current state. The efficiency is very important in this task due to the “online learning” constraint that we had to deal with. Moreover it is very intuitive so you can see the learning at a quit early stage of the training and it is an easier method to debug comparing to the other methods (i.e. policy gradients would be much more difficult to understand wether the model works well).

**Our model.**

*General description.* Our model is an online Q-Network where it plays many games against itself (or another opponent), it saves the values of a quartet of  $s', s, a, r$  (next\_state, state, action, reward) and it learn every end game. The learning stage can be treated as a regression problem where we are trying to find the optimal value of the  $maxQ^*(s', a')$  as described above (1.2). We chose to work with a Deep Neural Network (DNN) from the reason we can frame the problem of estimating  $Q(s, a)$  as a simple regression problem. Given an input vector consisting of  $s$  and  $a$  the neural net is supposed to predict the a value of  $Q(s, a)$  equal to  $target : r + \gamma * maxQ(s', a')$ . If we are good at predicting  $Q(s, a)$  for different states  $S$  and actions  $A$ , we have a good approximation of  $Q$ . Note that  $Q(s', a')$  is also a prediction of the neural network we are training.

---

**Algorithm 1** Pseudo of our the learning stage in our model

---

```

1: batch  $\leftarrow$  sample a random batch ▷ Sample a random batch from the ExperienceReplay DB
2: while left samples in batch do
3:   For each possible action  $a'(1,2,3,4,5,6,7)$ , predict expected future reward  $Q_t$  using DNN
4:   Choose the highest value of the actions predictions  $max_a Q(s', a')$  ▷ in our case we have 7 actions
5:   Calculate  $r + \gamma \times max_Q(s', a')$ . This is the target value for the DNN.
6:   Train the neural net using the loss function  $1/2(predicted_Q(s, a) - target)^2$ 
7: end while

```

---

*Model architecture.* We designed a DNN model like regression problem where the loss is defined (1.1). We chose a relatively small network in order to keep the number of parameters small yet, not too small because we do want to generalize the Q function well. The architecture is,

$LeakyRelu(Dense(Input, 128)) \implies LeakyRelu(Dense(128, 64)) \implies LeakyRelu(Dense(64, 64)) \implies TanH(Dense(64, 32)) \implies$   
 where input - (batch size, 42) ; actions - (7, )

*The importance of the move.* We noticed that most important sample is the sample from the last move in the game, so in order to increase our dataset quality we created a “Balanced experience replay” . The balanced database has two different memories, one for regular moves, and one for the last moves in the game. We used that property to create balanced batch which has the same amount of regular moves as the last moves. For example for batch\_size=16 we split it to 8-regular moves, 8-last moves. This trick showed an improvement on relatively small number of training iterations.

*The invert trick.* We thought about how to teach the model more last moves without playing the entire game, or maybe how can we teach the model to block the opponent when he is closed to win. We came up with an idea the invert the board hence switch between player 1 and player 2 pieces TODO\*\*\*\*

*Experience replay.* We sampled a quartet of  $s', s, a, r$  (next\_state, state, action, reward) from all the played moves in the previous games and store them, we also stored the last move of the game (before someone wins or losses), Then in the training we sampled a random shuffled batches from this dataset, (as a side note, it is very important to sample each batch where its samples are not consecutive in order to avoid wrong policy learning i.e. bad feedback loops).

**Exploration-Exploitation Trade-off.** One problem found in many areas of AI is the exploration problem. Should the agent always preform the best move available or should it take risks and “explore” by taking potentially suboptimal moves in the hope that they will actually lead to a better outcome (which can then be learnt from). A game that forces explorations is one where the inherent dynamics of the game force the player to explore, thus removing the choice from the agent.

In our model we used exploration rate of 10% which decays  $\times 2$  every 5000 rounds all the way to 0.1% and it stays there for the rest of the game. We didn’t have enough time to benchmark the different values of exploration-exploitation values but as a rule of thumb you do not want to eliminate the exploration in any stage of the game, yet you do not want to let the exploration a big portion of the moves, according to that we determined this ratio.

**A short description of the tests and results you got when you trained your policy.** TODO

**A detailing of other possible solutions that you tried and decided not to hand in and why (if were any).** TODO

**Any other things you would like to note about your implementation.** TODO