REINFORCEMENT LEARNING
CONNECT FOUR

Hanan Benzion 026606608
Idan Azuri 302867833

January 26, 2018

1. Practical Results

Board representation. We chose to represent the board as a vector in dimension 1×84. The intuition behind that was to simplify the board separating the players hence, we divided the board for two matrices of 6×7 each board contains only one player pieces and finally we flattened it. The main reason we chose this board representation is because it should help the model to separate between each player's pieces on the board. 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.

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'

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

$\pi - the\, policy \quad s^{'} - next\, state \quad a - action \quad \gamma - uncertainty\, value$

$Q*$ satisfies the following Bellman equation uncertainty

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

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,

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

where

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

and the Backward pass is

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

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

```
batch ← sample a random batch                ▷ Sample a random batch from the ExperienceReplay DB
while left samples in batch do
    For each possible action a'(1,2,3,4,5,6,7),predict expected future reward Q_t using DNN
    Choose the highest value of the actions predictions max_a' Q_{t+1}(s', a')   ▷ in our case we have 7 actions
    Calculate r + γ × max_Q(s', a'). This is the target value for the DNN.
    Train the neural net using the loss function 1/2(predicted_Q(s, a) − target)^2
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 to 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 - $(batch\,size, 84)$; $actions - (7,)$

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 movs 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 so the model would learn how to block the opponent. For instance in case that we are player 1 and player 2 is going to win in the next move, if we will reward player 1 to win the inverted game it actually the same as training it to block his opponent. To do so on each sample we save in the ExperienceReplay in the "last move" data structure, we also save the inverted game. Moreover, in order to boost the model to block player 2 one move before win - we take the games we won, invert them and then we reward the model for the same action that it would play to win, just that in this case he rewarded for blocking.

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 20% which decays ×2 every 200 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 we trained your policy. We checked our agent performance against the three main policies he will compete against. The RandomAgent and MinmaxAgent with depth 1 and 2. We checked every time 1000 rounds. We saw that against the Random it was a very easy game and we won in the beginning 70-80% and our wining rate went up to 90%. Against the Minmax depth=1 the game started with a winning rate of 45-55% and our wining rate went up till the end of the 1000 rounds to 65% Against the Minmax depth=2 we started with a winning rate of 7% and got at our best to 45% wining rate. All performance checks after we trained the model against itself.

A detailing of other possible solutions that we tried. We tried many models with the same method of Q-Network (we didn't implemented more methods due to lack in time). We use the simple Q-Network model for benchmarking many variants that we tried. One interesting trial that we had - we implemented a model with two objective functions when the first loss was the same as the Q-Network and the second loss function was to minimize the probability vector of the actions for the predicted state for prev_state. Another trial was to use CNN instead DNN from the reason the model can use the convolutional layers to catch a connected four (filters with receptive field of four).

However all of these models did not show an improvement in the long run.

Additional notes on our model. We mainly dealt with improving our techniques to have a better learning for any king of policy (as described in Our model section), but we did not have time for hyperparamters tuning and implementing different policies which probably would produce a big in performances.