Reinforcement Learning for playing "Connect Four"

Description

Using reinforcement learning allows artificial neural networks to learn playing games successfully. Connect Four is a simple game for two players in which the goal is to build a line of 4 adjacent blocks before the opponent.

Goal

Design and implement a Neural Network that learns to play “Connect Four” via Reinforcement Learning.

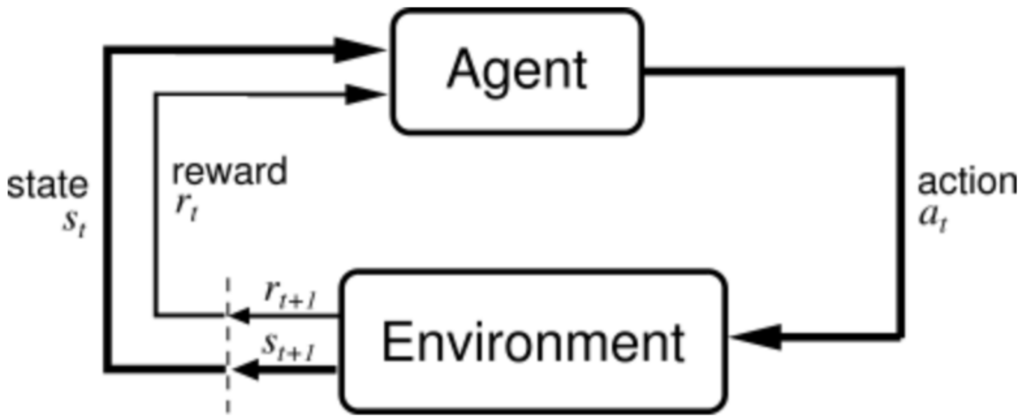
Reinforcement Learning (RL)

As we humans learn, it is by exploring / doing new things and getting rewards positive as negatives from our environment afterwards. So, we know how good our action was by the feedback of others. The same principle works for Reinforcement Learning – a computational approach of learning how we know it. RL algorithms can start from a blank slate, and under the right conditions, they achieve superhuman performance. These algorithms are one part of Machine Learning (ML), speaking of ML, the most common way to train models is to use labeled data which is called Supervised Learning. RL is statistically less efficient, but the advantage is, that it can start with nothing more than a state.

How does Reinforcement Learning work?

To introduce some terminologies:

1. Agent – learner and decision maker
2. Environment – where the agent learns and decide what actions to perform
3. Action – a set of actions which the agent can perform
4. State – the state of the agent in the environment
5. Reward – for each action selected by the agent the environment provides a reward. Usually a scalar value



source

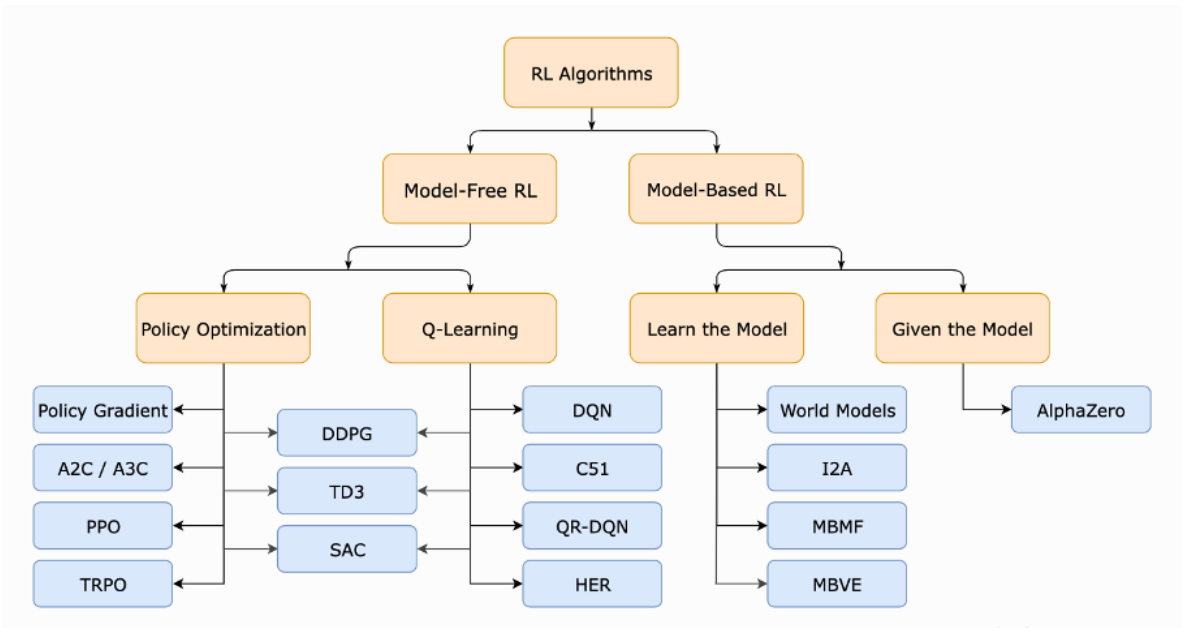
So environments are functions that transform an action taken in the current state into the next state and a reward; agents are functions that transform the new state and reward into the next action. Most of the time we don't know the function of the environment. It is a black box where we only see the inputs and outputs. Reinforcement Learning represents an agent’s attempt to approximate the environment’s function, such that we can send actions into the black-box environment that maximize the rewards.

In the feedback loop above, the subscripts denote the time steps t and t+1, each of which refer to different states: the state at moment t, and the state at moment t+1. Unlike other forms of Machine Learning, RL can only be thought about sequentially in terms of state-action pairs that occur one after the other.

As for our setting - the connect four environment - we can not reward every move, because until the end of the game it is unknown if the move was good or bad. So RL solves the difficult problem of correlating immediate actions with the delayed returns they produce. After a win we reward every move with a +20, after a loss with -20. With time and many simulations you sum (for every state) the rewards of moves up and end up (hopefully) with the best possible move. Sometimes it can be difficult to understand which actions lead to the best move. There is a wide variety of Reinforcement Learning algorithms that can achieve this goal.

Variety of Reinforcement Learning Algorithms

We decided to take a deeper look in the Model-free RL part. This means we have to learn directly out of the environment, which can achieve the same optimal behavior as model based RL algorithms.



source

We chose Deep Q Learning (DQN) out of interest, because of its simple structure and the possibility to work with Neural Networks. DQN is based on Q Learning, which stores for every game state the best possible move in a Q-Table. This table for our environment would have been around 4 quadrillion entries big – way too big for our computers.

Deep Q Learning

Q Learning

Q Learning is a value-based RL algorithm. The Q stands for quality - how useful a given action is in gaining some future reward. This means that the algorithm stores values in the Q-Table which have the maximum expected future reward for that given state and action. For updating the values of this Q-Table, you use the Q Learning formula, which takes just the state and the action as input. And returns the expected future reward of that action and state.

$$\underbrace{\boxed{Q^{\pi}(s_t, a_t)}}_{\text{Q-Values for the state given a particular state}} = \underbrace{\boxed{\underline{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots}}_{\text{Expected discounted cumulative reward}} \underbrace{\boxed{s_t, a_t}}_{\text{Given the state and action}}]$$

How to update this equation?

```
Initialize Q-values (Q(s,a)) arbitrarily for all state-action pairs
Until learning is stopped...
    Choose an action (a) in the current world state (s) based on the current Q-
value estimates (Q(s,_))
    Take the action (a) and observe the outcome state (s') and reward (r)
    Update Q(s,a) = Q(s,a) + learning_rate * [r + discount_rate * (highest Q
value between possible actions from the new state
    s') - Q(s,a)]
```

source

This equation is based on the Bellman Equation. To understand the formula it is much easier to look on a code example. Because we use DQN, deeper understanding of this equation is not much help for the future, our Neural Network will update the Q-Table.

Deep Q Learning

Deep Q Learning has the same foundation as Q Learning; the only difference is a substitution of the Q-Table. During the substitution of the Q-Table the Neural Network will generalize the best moves for every state, to make it more efficient. This differs from Q learning, where you store for every state the best move in a Q-Table. DQN has the benefit over standard neural networks that it can interact with its environment.

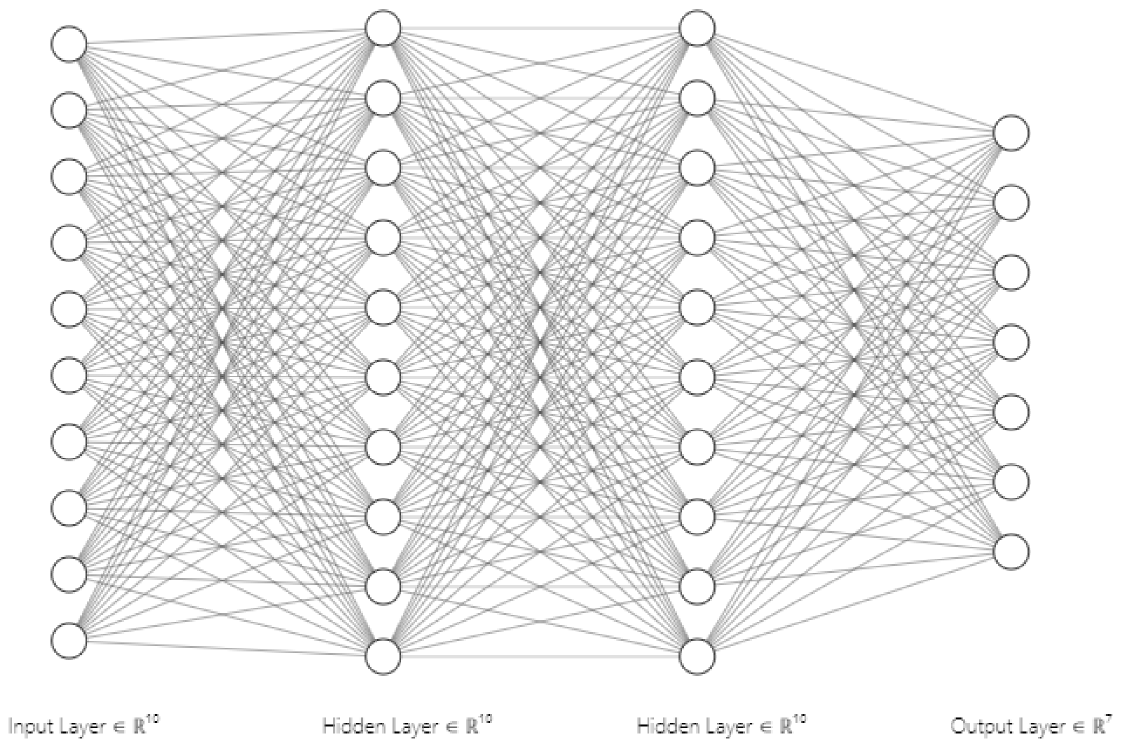
A key decision for designing a DQN-Network is the choice between two different architectures. The first uses the state and the actions as input to predict a scalar value. This scalar prediction is the desired action. For this project this would mean, that the Neural Network would output the scalar column in which the network wants to drop the piece.

The second approach is to only feed the state into the neural network for the training process. This requires the architectural change of choosing the number of output neurons the same as the number of possible actions. This approach reveals interesting insights into the networks choice of action. Yet this can only be applied if the number of potential actions is relatively low.

For this project, since the game connect four only has seven different actions, the second approach is used for the architecture of the network with seven output neurons equal to the possible actions.

Creating the Neural Network

There are many convenient ways to build a Neural Network. We decided to use Pytorch for our layers. The output of our network has seven neurons, each corresponding to the action of dropping a piece into one of the seven different columns of the board. Below is an example of a possible network for the connect four environment.



source

DQN Network

For our agents network, we use eight fully connected layers. We originally started of with only three layers, but decided to enhance that number because we wanted a higher approximation accuracy. Since Deep Q Learning uses the property, that neural networks are universal function approximators (universal approximation theorem), we thought that only one hidden layer wouldn't be enough to get a good approximation of the q function. The number of Neurons in every layer, except the output layer, is 42 which is the size of the input vector as the board is of size 6x7. In addition to the fully connected layers, we added some drop out layers to prevent overfitting. This is especially important in the context of Connect Four considering the training process. Overfitting to an opponent would mean worse performance against an opponent with a different strategy. As an optimizer, we decided to use Adam (adaptive moment estimation), given that it adaptively selects a separate learning rate for each parameter, which makes manual tuning less important. Hence performance is less sensitive to hyper-parameters when compared to other optimizers such as SGD (Stochastic Gradient Descent). source

```
class DeepQNetwork(nn.Module):
    def __init__(self, lr, input_dims, fc1_dims, fc2_dims, fc3_dims, fc4_dims,
fc5_dims, fc6_dims, fc7_dims, n_actions):
        super(DeepQNetwork, self).__init__()
        self.input_dims = input_dims
        self.fc1_dims= fc1_dims
        self.fc2_dims = fc2_dims
        self.fc3_dims = fc3_dims
        self.fc4_dims = fc4_dims
        self.fc5_dims = fc5_dims
        self.fc6_dims = fc6_dims
        self.fc7_dims = fc7_dims
        self.n_actions = n_actions
        self.fc1 = nn.Linear(self.input_dims,self.fc1_dims)
        self.fc2 = nn.Linear(self.fc1_dims,self.fc2_dims)
        self.fc3 = nn.Linear(self.fc2_dims,self.fc3_dims)
        self.fc4 = nn.Linear(self.fc3_dims,self.fc4_dims)
        self.fc5 = nn.Linear(self.fc4_dims,self.fc5_dims)
        self.fc6 = nn.Linear(self.fc5_dims,self.fc6_dims)
        self.fc7 = nn.Linear(self.fc6_dims,self.fc7_dims)
        self.fc8 = nn.Linear(self.fc7_dims,self.n_actions)
        #dropout layers against overfitting
        self.dp1 = T.nn.Dropout(0.5)
        self.dp2 = T.nn.Dropout(0.3)
        self.dp3 = T.nn.Dropout(0.2)
        #choose Adam(adaptive moment estimation) as optimizer
        self.optimizer = optim.Adam(self.parameters(), lr=lr)
        #mean squared error as loss function
        self.loss = nn.MSELoss()
        #use graphics card if available
        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')

    #forward pass
    def forward(self, state):
        x = F.relu(self.fc1(state.float()))
        x = F.relu(self.fc2(x))
        x = self.dp1(x)
        x = F.relu(self.fc3(x))
        x = F.relu(self.fc4(x))
        x = self.dp2(x)
        x = F.relu(self.fc5(x))
        x = F.relu(self.fc6(x))
        x = self.dp3(x)
        x = F.relu(self.fc7(x))
        actions = self.fc8(x)
        return actions
```

DQN Agent

Exploration vs. Exploitation

Like in many reinforcement problems, there exists a trade-off between exploration and exploitation. Exploration means, that the agent makes some random decisions or more generally speaking out of character decisions that are not given by the network. This way it tries to find new strategies, that will potentially improve the received reward. Exploitation takes the actions that are calculated by the network. Hence actions that were already learned. Its necessary to find the right balance between these two strategies because an agent that only exploits already learned actions will never be presented with new actions to learn from. On the other hand an agent that always explores will make only random moves and therefore will never reach a high performance level. We deal with this problem by using a technique called epsilon decay. As can be seen in our agents choose_action method, epsilon is the probability that the agent chooses a random action. This epsilon will decrease over time during training as at the end of every call of the learn method a custom eps_dec is multiplied. However it will never vanish completely as we stop decreasing it when eps_min is reached.

```
def choose_action(self,observation):
        if np.random.random() > self.epsilon:
            state = T.tensor([observation]).to(self.Q_eval.device)
            actions = self.Q_eval.forward(state)
            action = T.argmax(actions).item()
        else:
            action = np.random.choice(self.action_space)

        return action
```

Learning Process

After every game all the information the agent gained during that game will be stored in its memory. This memory consists of separate dictionaries, that all save an element of a quintuple with respect to the same index so they can be accessed easier later on. The quintuple consists of the original board state, the action taken, the resulting board state, the corresponding reward and a Boolean depending on if the game was over or not after the taken action. If the agents max_mem_size is reached, old transitions are overwritten. It is important to not choose the memory size to small to prevent overfitting to previous observations.

```
def store_transition(self, state, action, reward, state_, done):
        index = self.mem_cntr % self.mem_size
        self.state_memory[index] = state
        self.new_state_memory[index] = state_
        self.reward_memory[index] = reward
        self.action_memory[index] = action
        self.terminal_memory[index] = done

        self.mem_cntr += 1
```

The agent learns the saved transitions in batches that are chosen randomly from its memory because it would be very time consuming to iterate over all transitions every time the learn method is called (after every game). For learning we use the Q-Learning formula and then use the mean squared error to compute the loss between the q_target value given by the equation and the value q_eval given by the current network state. After that we perform backpropagation and update the networks parameters using the calculated gradients.

```
def learn(self):
    # only start learning if enough memories to fill up batch
    if self.mem_cntr < self.batch_size:
        return

    for i in range(100):
        #reset gradients
        self.Q_eval.optimizer.zero_grad()

        #choose random batch
        max_mem = min(self.mem_cntr, self.mem_size)
        batch = np.random.choice(max_mem, self.batch_size, replace=False)
        batch_index = np.arange(self.batch_size, dtype=np.int32)

        #get batches from memory
        state_batch = T.tensor(self.state_memory[batch]).to(self.Q_eval.device)
        new_state_batch =
T.tensor(self.new_state_memory[batch]).to(self.Q_eval.device)
        reward_batch =
T.tensor(self.reward_memory[batch]).to(self.Q_eval.device)
        terminal_batch =
T.tensor(self.terminal_memory[batch]).to(self.Q_eval.device)
        action_batch = self.action_memory[batch]

        #compute current network output
        q_eval = self.Q_eval.forward(state_batch)[batch_index, action_batch]
        q_next = self.Q_eval.forward(new_state_batch)
        q_next[terminal_batch] = 0.0

        #q-update
        q_target = reward_batch + self.gamma * T.max(q_next, dim=1)[0]
        #compute loss(mse)
        loss = self.Q_eval.loss(q_target, q_eval).to(self.Q_eval.device)
        #backprop
        loss.backward()
        #apply calculated gradients
        self.Q_eval.optimizer.step()

    #update epsilon decay
    self.epsilon = self.epsilon * self.eps_dec if self.epsilon > self.eps_min
else self.eps_min
```

DQN Opponent - Minimax

Due to the tremendous time consumption of self-play reinforcement learning, we chose the quicker option by providing our Neural Network a deterministic Minimax opponent to train on. The strength of Minimax is adjustable by the depth feature. The depth indicates how many time steps the Minimax will estimate into the future. For example, a depth of 3 measure the next two turns of the opponent with respect to our turn in between. For that reason, the Minimax needs to maximize its turn and minimize the opponents turn.

Those procedures are computed by an altering recursion, which can be seen in the pseudocode below.

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
   else (* minimizing player *)
       value := +∞
       for each child of node do
           value := min(value, minimax(child, depth − 1, TRUE))
       return value
```

The Minimax expects the opponent to make the best possible move. For calculating the quality of a move, a score function needs to be provided. This score function can be arbitrarily with multiple if-conditions. For example, if the Minimax already has three pieces in a row, the score for placing a fourth piece, the winning piece, should be very high. Furthermore, the score for blocking the opponent from making a winning move should be also rewarded very highly by the scoring function. Those scores are computed for every possible piece position after each players turn.

```python
def evaluate_window(window, piece):
    score = 0
    opp_piece = PLAYER_PIECE
    if piece == AI_PIECE:
        opp_piece = PLAYER_PIECE

    if window.count(piece) == 4:
        score += 100
    elif window.count(piece) == 3 and window.count(EMPTY) == 1:
        score += 5
    elif window.count(piece) == 2 and window.count(EMPTY) == 2:
        score += 2

    if window.count(opp_piece) == 3 and window.count(EMPTY) == 1:
        score -= 4

    return score
```

The evaluate_window function takes as input a window of size four. Every possible game move, vertically, horizontally and diagonal will be considered within the window. So for every single possible winning-window the score will be evaluated. This is part of the recursive process, because the score is returned and then the path with the maximum score is selected as the next turn of the Minimax. The code above shows that we score a winning move with 100. If the Minimax could put a piece next to two of its own pieces, so three in a row, column or diagonal, this is scored by 5. The more pieces put next to each other will increase the score for this possible move. The Minimax will also block the opponent if the Neural Network or we have three pieces next to each other. This is achieved by scoring with a negative value. High negative values will be selected in the minimizing recursion of the Minimax algorithm.

It is obvious that the computational time increases exponentially. For that reason the Minimax can be extended with the so-called Alpha-Beta Pruning. This pruning eliminates low scores from being predicted and so it can save up to 50% computation time. The maximum depth with alpha-beta pruning on our PC was seven. But for this depth, one Minimax turn takes over 10 seconds. This is way too long to be used for the Reinforcement Learning process. The best combination of fast computation and good turns is reached by choosing a depth of three or four.

DQN Training

We started training our DQN agent by letting it play against a random agent. A benefit from that is, that games tended to be very long, since our agent itself had, adding to it being untrained, a high epsilon at the beginning and hence made mostly random moves as well. With the games being long the agent is able to see a lot of different board states and actions and therefore has a high exploration rate. However we also encountered a drawback from that training strategy very early on. Since there are seven columns, vertical wins against a random agent are very easy to achieve because the chance of the random agent blocking a vertical victory is only 3/7. To overcome this issue, we modified the rewards for playing against the random agent. Furthermore in the following code snippet can be seen, that we give a reward with respect to the outcome of the game. Hence every move made by the agent in that game gets the same reward.

```
if g.getWinner() == Tie:
        reward = 0
    winner = AI if turn == PLAYER else PLAYER
    if winner == AI:
        win_cntr += 1
        if vertical_win:
            reward = 5
        else:
            reward = 20
    else:
        reward = -20
```

Before every move of the agent, the current board state is transformed into a 42 dimensional vector to fit the input format required for the agents network. If the agent attempts an invalid action, a high negative reward is placed on that move and its stored to the agents memory. Following that, the agents resulting action is a random move. That follows because the agents learn method is only called at the end of the game hence the network would attempt the same move again. After each action a tuple consisting of the original board state, the action taken, the resulting board state and the Boolean terminal value is saved. At the end of the game all saved transitions are stored to the agents memory together with the resulting reward and its learn function is called.

```
observation = []
for sublist in g.board:
    for i in sublist:
        observation.append(i)
observation = np.asarray(observation)
action = agent.choose_action(observation)
if g.check_if_action_valid(action):
    g.insert(action,AI_PIECE)
else:
    while g.check_if_action_valid(action) == False:
        agent.store_transition(observation, action, -100, observation, done)
        action = np.random.randint(7)
    g.insert(action,AI_PIECE)
observation_ = []
for sublist in g.board:
    for i in sublist:
        observation_.append(i)
observation_ = np.asarray(observation_)
transitions_agent += [(observation, action, observation_, done)]

for i in range(len(transitions_agent)):
    agent.store_transition(transitions_agent[i][0], transitions_agent[i][1],
reward,
                           transitions_agent[i][2], transitions_agent[i][3])
agent.learn()
```

The learning process is the most time consuming part. We checked related work e.g. Alpha Zero Go how long they needed to train their model. We also did research about other training processes of reinforcement learning. We concluded that for a game like connect four, with a single GPU we need about three weeks of pure training in order to get a really strong model. The best results can be achieved by letting the network learn with self-play Reinforcement Algorithm. Then it could sub-sequentially check its strength against e.g. a Monte Carlo Tree Search model. This is an analogical approach as ours, but a bit more complex and stronger.

However, this approach could not be realized by us due to time constraints. As we decided to let the network train against the minimax, we still faced many problems. Many hyperparameters like the decay (randomness), number of episodes, depth of the minimax, batch size or learning rate needed to be changed and then optimized.

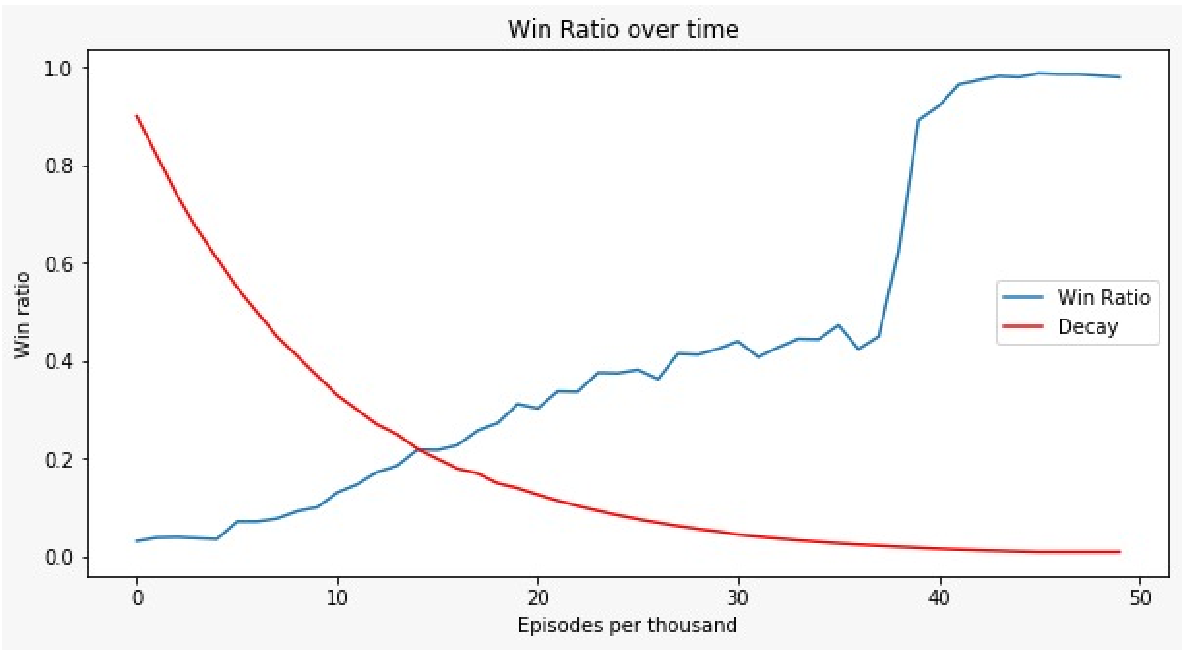
We tried out many different approaches, until we found a good set of hyperparameters for us.

We use:

- Starting decay of 0.9999 which decreases exponentially over time until 0.01
- Episodes 50000 (15 hours training time)
- Depth of 1 and additional 10000 on 2 for validation
- Batch size of 64
- Learning rate of 0.01

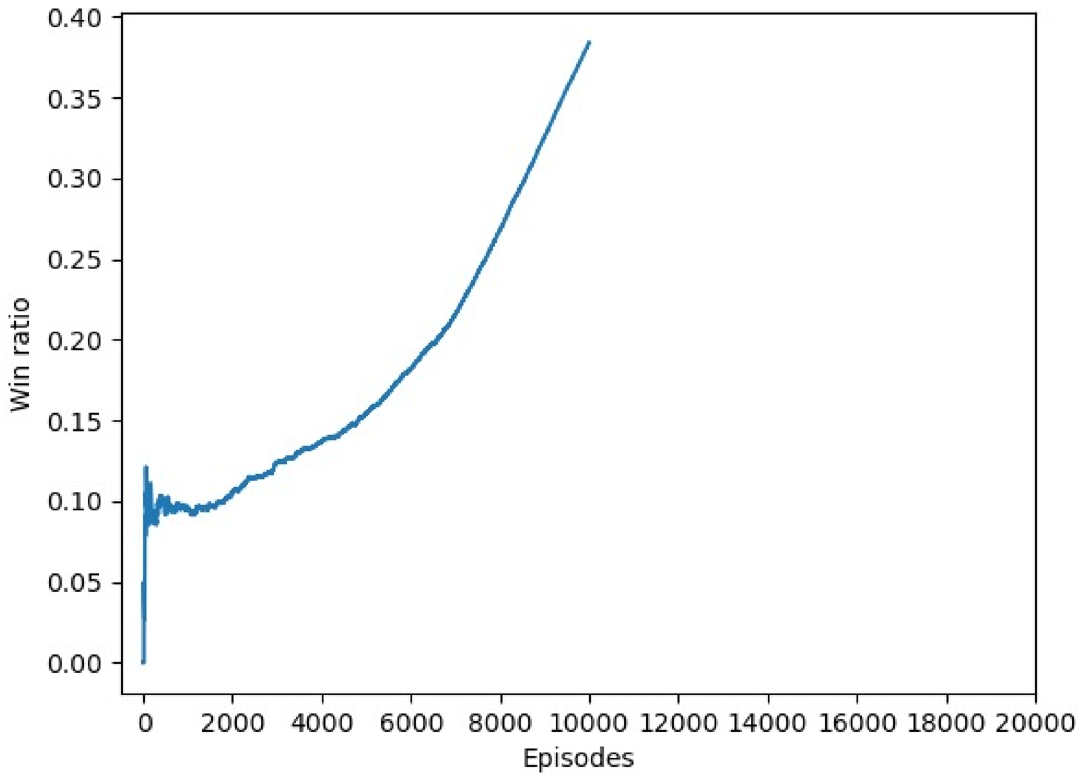
Simulation

We let the network train with the set of hyperparameters mentioned above. We can clearly see the learning progress the longer we let the network train.



After around 40.000 episodes we see a jump, which indicates a winning strategy over the Minimax with depth one was found. As the Minimax plays the same moves for every state our agent will exploit this knowledge and win every time if the decay is low.

In this plot we trained against the Minimax with depth two. We can see an exponential learning progress - to reach 100% it will need a lot more time and computational power...



Using this model, we can also see a dramatic increase of wins against the depth two Minimax starting from epoch 4000. Unfortunately this training took already very long.

Another interesting point is, that the neural network has such a high win ratio, because it found a way to outsmart the Minimax. Since the network is only trained against the Minimax and not humans nor itself, it mostly learned how to beat the Minimax. This is one of the reasons why we used Dropout layers to prevent the neural network as much from overfitting as possible.

Conclusion

In conclusion we can say that we reached our goal in successfully designing and implementing a deep reinforcement model. Due to a lack of time and computational powers we were not able to train it to the point, were it can compete against humans or more skilled opponents in general. However the learning effect can clearly be seen against weaker opponents. Our initial struggle was to let the neural network play properly against the Minimax and a random agent. In order to get better results we had to find the right balance between exploration and exploitation. Like in RL the learning process itself consists out of trial and error. During this process we could properly define the hyperparameters and let the neural network play against the minimax.

With the attached files it is left to the reader to train and play against our model to improve its performance.

Authors

- **Tim Löhr** - Mavengence
- **Simon Hölck**
- **Florian Cimander** - Flocimander

Kaggle Competition

To explore more the world of RL and Connect Four, there is an Connect X environment instantiated by Kaggle in one of their Kaggle competitions.