UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Design of AI Systems

Mini-Project: Introducing Deep Learning to TicTacToe Game

Group 19:

Tim Boleslawsky, 980524-6478, Applied Data Science Master's, gusbolesti@student.gu.se
Mikołaj Nowacki, 20010918-2014, Applied Data Science Master's, gusnowami@student.gu.se

We hereby declare that we have both actively participated in solving every exercise. All solutions are entirely our own work, without having taken part of other solutions.

1. Introduction

1.1 Outline

For our mini-project, we want to extend the Monte Carlo Tree Search (MCTS) algorithm we have implemented in week 6 of the course "Design of AI Systems". The goal of this MCTS algorithm was to play the game of Tic-Tac-Toe. We want to build upon the relatively simple solution we implemented in week 6 and introduce deep learning to improve the performance of this algorithm. As an outcome, we envision a Python script that combines the techniques of MCTS and adds deep learning.

The approach we want to take for this mini-project is to focus on the policy network. For this, we take inspiration from the paper "Mastering the game of Go with deep neural networks and tree search". In the paper, both the improvement of the policy via supervised learning as well as reinforcement learning was described. Because we don't have data to do supervised learning, we will solely focus on the reinforcement learning part. The improvement of the value network is not part of the scope of this mini-project.

First we will describe our implementation of the system. We will go into detail about the architecture of the neural network, how the reinforcement learning was implemented, and how the MCTS algorithm needed to be adjusted to incorporate the trained policy network. Afterwards we want to evaluate how this improved MCTS algorithm performs when compared to our 'simple' MCTS algorithm. In the fourth section of this mini-project, we will explore a different approach which focuses solely on deep Q learning. Finally we will summarize our results and thoughts in a final conclusion.

The code for our solutions is attached as a pdf-file.

1.2 Introduction of the Old Model

To make it easier for the reader, and because our implementation is largely based on this, we quickly want to recap our "simple" MCTS algorithm from module 6.

The basic idea was to divide the script into two classes. The first class (TicTacToe) is the structure of the game, and the second class (MCTSNode) represents a node in a MCTS. Here are the key components of this solution:

- Selection Policy: For the selection policy we chose the UCT formula.
- Roll-out Policy: The roll-out policy for the player is random. When selecting a node of an untried move, we simulate the game from this node by selecting random moves of the available moves until the game is finished.
- Backpropagation: The updates to the search tree in the backpropagation are based on the idea that we "reward" a node with 1 and punish a node with "-1" based on if this node lead to a win or not.

2. Implementation of the System

The general idea behind the implementation is to support the "Selection" step in the MCTS algorithm we have implemented in module 6 with an improved tree policy. The basic functionality of this tree policy is to guide the traversal of the MCTS algorithm and select the most promising nodes. A node in our case represents a move in the game Tic-Tac-Toe. In this section, we will discuss the initial implementation by first looking at the architecture of the neural network we used, then looking at how we implemented reinforcement learning, and finally discussing how we integrate the trained policy network into the MCTS algorithm.

2.1 Architecture of the Neural Network

Our initial implementation of the policy network looked like this:

```
class PolicyNetwork(nn.Module):
    def __init__(self):
        super(PolicyNetwork, self).__init__()
        self.fc1 = nn.Linear(9, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, 9)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return self.softmax(x)
```

There are a few noteworthy decisions we chose to make in order to arrive at this architecture. First, we decided on 3 relatively simple fully connected linear layers with softmax as the activation function for the outputs. Softmax is appropriate here, because we want to classify the possible moves by predicting the move with the highest winning possibility, making this multi-class classification. For now, we decided against convolution layers, because we don't work with spatial data like images or videos. For the input and output size, we obviously have 3*3 input or output neurons with a 3*3 board. For the remaining neurons, we decided on a relatively small number of 64. That is because we don't have a lot of complexity and will have a relatively small sample size (we aim for 100-300 training games).

The forward function defines activation functions, of the activation functions of the hidden layers, which in our case is a standard ReLU activation function.

2.2 Reinforcement Learning

For the implementation of the reinforcement learning, we implement the function *train_policy_networ*() depicted below. The basic idea is to let the policy network play against itself, and after each game, update the network parameters based on gradients we get from calculating a loss function and backpropagating these results.

```python
def train_policy_network(policy_network, optimizer, games):
    for game_num in range(games):
        print("Training game:", game_num)
        game = TicTacToe(board_size)
        states = []
        played_moves = []
        while not game.is_terminal():
            root = MCTSNode(game, policy_network=policy_network)
            move = root.run_mcts(1000)
            states.append(game.board.flatten())
            move_index = move[0] * 3 + move[1]
            played_moves.append(F.one_hot(torch.tensor(
                move_index), num_classes=9).float())
            game.make_move(move)

        winner = game.check_winner()
        if winner == 1:
            rewards = [1 if i % 2 == 0 else -1 for i in range(
                len(states))]
        elif winner == -1:
            rewards = [-1 if i % 2 == 0 else 1 for i in range(
                len(states))]
        else:
            rewards = [0 for _ in range(len(states))]

        optimizer.zero_grad()
        for state, move_prob, reward in zip(states, played_moves
            , rewards):
            state_tensor = torch.FloatTensor(state).unsqueeze(0)
            predicted_probs = policy_network.forward(
                state_tensor)
            loss = -torch.sum(torch.FloatTensor(move_prob) *
                torch.log(predicted_probs)) * reward
            loss.backward()
        optimizer.step()
```

Let's go through our implementation step by step. First, we iterate over the number of training games we want to perform. Then, we define the necessary variables and start a game. We did not change the way the game is played except for the points we discuss in the next subsection. While the network plays against itself, we save the states and the played moves (one hot encoded). We do this to be able to compare the current beliefs of the system for a given state against the outcomes of the training game. When the game is finished, we determine a winner and assign a respective reward. Now we go through each state, look at the current belief of the policy network and compute the gradient loss. The gradient loss serves to answer the question: "Are the moves with a high assigned probability leading to victories?". Our loss function is based on the Policy Gradient Loss method appropriate when dealing with policies. After the loss is calculated, we run backpropagation and update the policy network's parameters using

the computed gradients from the backpropagation. As an optimizer, we use the common "Adam" optimizer with the standard learning rate of 0.001.

2.3 Integrating the Policy Network into the MCTS Algorithm

After we trained the policy network, we have a ready neural network that we can use to improve our MCTS algorithm. To integrate the policy network, we need to make a few changes to the existing solution:

- First, we need to introduce a method to get the prior probabilities for the available moves in a current game state from the policy network. For this with introduce the method *get_prior_probabilities*(). Within this function, we feed the current board tensor to the trained policy network and receive the probabilities which we can then assign to the available moves.

- Second, we need to change the calculation of the UCT formula to include the prior probabilities. In our original solution we use the standard UCT formula for the traversal of the MCTS. To incorporate the believes of our policy network, we need to introduce the prior probabilities to this formula. To do this we multiply the exploration component of the UCT formula to arrive at this formula: $UCT(s,a) = \frac{Q(s,a)}{N(s,a)} + c \cdot P(a|s) \cdot \sqrt{\frac{\ln N(s)}{N(s,a)+1}}$. Here $P(a|s$ are the prior probabilities from the policy network.

3. Evaluation of the Implementation

To evaluate our implementation, we want to let the trained network with the MCTS algorithm play against a completely random opponent. We already had the same approach when testing our solution for module 6. In the end, we want to compare the 'simple' MCTS algorithm and the MCTS algorithm supported by deep learning by comparing the results against the random opponent.

As a reminder, our solution for module 6 had the following results against the random opponent: The AI won 79 times, the opponent won 19 times, and there were 2 draws. These results were relatively stable with a few games plus or minus when running the test multiple times.

For our first test, we wanted to train the model on 100 training games and play 100 test games against a random opponent. Results with variables lr=0.001, training games=100, architecture=64 neurons:

- AI won 65 games.
- Opponent won 21 games.
- 14 games ended in a draw.
- The runtime was 1 minute and 50 seconds.

This result was obviously pretty disappointing because we could not see any improvement when compared to the roughly 80% win rate of the 'simple' MCTS algorithm. We had a few ideas of why that could happen. First, maybe 100 training games are just not enough. Second, maybe a 3x3 grid is too easy and we can't unlock the full potential of the deep learning neural network. Third, maybe the architecture of the network is faulty and we need to change the number of neurons. Fourth, maybe the learning rate of 0.001 is not optimal. Because we want to ensure clarity and transparency, we want to change only one of these variables at a time. First we tried to increase the learning rate to 0.01. Results with variables lr=0.01, training games=100, architecture=64 neurons:

- AI won 71 games.
- Opponent won 16 games.
- 13 games ended in a draw.
- The runtime was 1 minute and 54 seconds.

So we see a slight improvement but still worse than the 'simple' MCTS algorithm. That a higher learning rate leads to better results indicates that the task is not too complex. The second obvious step is to increase the training games. So we increased the training games from 100 to 300. Results with variables lr=0.01, training games=300, architecture=64 neurons:

- AI won 77 games.
- Opponent won 22 games.

- 1 games ended in a draw.
- The runtime was 3 minute and 30 seconds.

This time around we again see a clear improvement but also an increase in runtime. So the next step was to look at the architecture of the network. We started with a network that was connected by 64 neurons. Because Tic-Tac-Toe on a 3x3 grid is not the most complex game, we decided to try a network that is connected by 32 neurons. Results with variables lr=0.01, training games=300, architecture=32 neurons:

- AI won 65 games.
- Opponent won 14 games.
- 21 games ended in a draw.
- The runtime was 4 minute and 1 second.

This unfortunately did not lead to any improvement and actually increased the runtime (although not by much). From this, we conclude that the 64-neuron architecture captures the complexity of the problem sufficiently.

This means that by changing the variables and testing different approaches to the 3x3 Tic-Tac-Toe game, we still could only be roughly as good as the 'simple' MCTS algorithm. So the next idea was that maybe the 3x3 Tic-Tac-Toe game is just too simple and the advantages of a trained policy network cannot be seen. Therefore, we change our code so that we can set dynamic game sizes (for full code see attachment). Because we need to have a baseline to check our deep learning enhanced MCTS algorithm, we first tried the 4x4 grid with a 'simple' MCTS algorithm. Results for 4x4 grid with 'simple' MCTS algorithm:

- AI won 26 games.
- Opponent won 45 games.
- 29 games ended in a draw.

This shows the obvious, that a 4x4 grid Tic-Tac-Toe game is way harder than a 3x3 grid. So now we want to take the best parameters from the 3x3 grid Tic-Tac-Toe game for our new implementation and run 100 test games against a random opponent on a 4x4 grid. Results for 4x4 variables lr=0.01, training games=300, architecture=64 neurons:

- AI won 71 games.
- Opponent won 21 games.
- 8 games ended in a draw.
- The runtime was 14 minutes and 30 seconds (only training).

We can see that this result is a significant improvement to the 'simple' MCTS algorithm. It also needs to be mentioned that we maybe could improve this result by again experimenting with a lower learning rate, more training games, or a different architecture, but this would all likely increase the runtime which already is fairly high.

Although we could show the advantages of the integration of a policy network trained with reinforcement learning into the MCTS algorithm, we still were not satisfied with the long runtime and results. This led us to explore an approach based purely on deep learning. We describe this approach in the next section.

4. Deep Q Learning Approach

The implementation of the second approach focuses solely on deep reinforcement learning. Utilizing the Q-learning algorithm to train an agent, the model learns the rules and tactics of the game by playing against a player who chooses its moves randomly. This approach allowed us to achieve 95% win rate for 10 000 samples. Such sample size allows for easier identification of the *learning curve* as shown in figure 3 later on. It is also more computationally efficient than the MCTS + RL solution as it takes less than 2 minutes to fully train the model on 10 000 samples.

4.1 Model

The model is a deep neural network with 3 fully connected layers and 256 hidden layers ($9 \times 256 \times 9$). It uses the *ReLu* function for the activation function. The agent uses the model to predict an action based on a given state. It uses the epsilon-greedy policy to maintain balance between exploration and exploitation. We have chosen the exponential decay for the epsilon value to decrease the randomness of action over time. In the initial phase, the random action has 90% chance of occurrence, which after approximately 2000 games levels out at 0.05% chance.

4.2 Q-learning

The fundamental learning principle of our model is the Q-learning formula in Figure1, commonly used in reinforcement learning applications. This technique is used to learn

$$Q^{new}(S_t, A_t) \leftarrow (1 - \underbrace{\alpha}_{\text{learning rate}}) \cdot \underbrace{Q(S_t, A_t)}_{\text{current value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \Big(\underbrace{\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(S_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} \Big)$$

Figure 1: Q-learning formula (source: Wikipedia)

the value of an action in a particular state.

- The $Q(s_t, a_t)$ is the current estimate of the Q-value for taking an action in a given state, it is predicted by our model.
- α is the learning rate hyperparameter set to LR = 0.001 and determines how much new information overrides old knowledge
- r is the immediate reward: +10 for a win, +5 for a draw, 0 for an ongoing move, -5 for an invalid move (if the chosen grid field is already marked, penalize and choose again), -10 for a loss.
- γ is the discount factor that determines the importance of future rewards. We set it to 0.9 which means the algorithm prioritizes long-term rewards.
- $maxQ(s_{t+1}, a$ is the maximum Q-value for the next state s_{t+1}, representing the best possible future reward.

- $(R_{t+1} + \gamma \cdot maxQ(S_{t+1}, a)$ is the training update rule which derives from the the Bellman equation for optimality.

4.3 Memory replay and optimization

Our model experiences memory replay with two functions updating the short-term and long-term memory. It stores the previous state, action taken based on that state, reward for the action, new state after the action was taken, and if it was the final move in an episode. It is used to calculate the loss function and calculate the weights of the neural network based on experience.

- Short-term memory is invoked after each move of the agent. It updates the model based on current results.
- Long-term memory is invoked at the end of each episode. It uses a batch size of 100 samples which are chosen randomly from the memory. It ensures the actions are decorrelated, which greatly stabilizes the training procedure.

The model aims to minimize the temporal difference error:

$$\delta = Q(s, a) - (r + \gamma \cdot maxQ(s', a')) \tag{1}$$

using Mean Squared Error function. The weights of the neural network are then optimized by Adam optimizer with 0.001 learning rate.

4.4 Demo

The reinforcement learning model learns by playing against randomly generated moves in an offensive strategy (The agent starts first). We trained the algorithm on 10 000 samples (episodes) and achieved the following results:

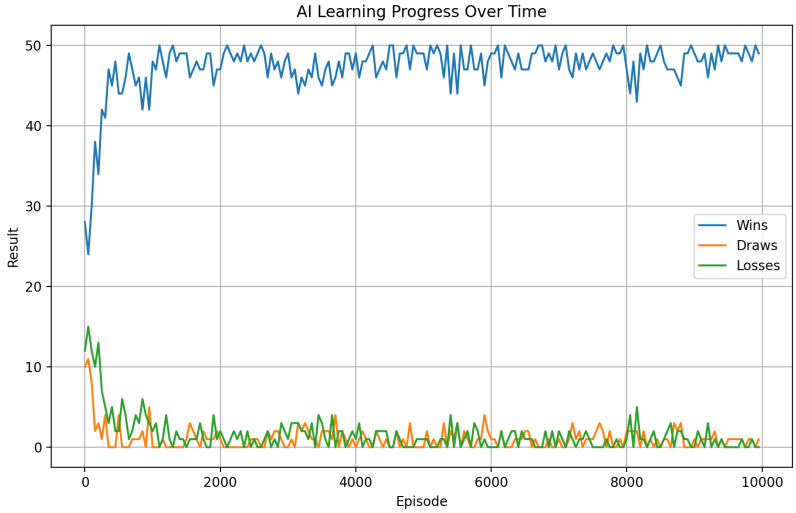

Figure 2: Learning progress per 50 games

- AI wins: 9515
- Random player wins: 310
- Draws: 175

After running multiple learning sessions, the model averages at 95% win rate. Figure 3 shows the learning progress of our model. The *learning curve* is clearly visible and symbolizes the exponential learning rate. In the initial 0 - 2000 games the AI is focused on exploration to find out different ways of winning a game. After ϵ decays exponentially to 0.05 the algorithm then sets to rely on its experience (exploitation). This can be seen as the curve between 0 - 2000th episode.

4.5 Evaluation against the MCTS extended model

The MCTS extended model achieved 70% win rate on average while the DQN model averaged 95% win rate. The main reason for this difference is the memory replay ability of the DQN. When it was disabled, DQN's win rate decreased significantly and equaled MCTS performance.

- AI wins: 580
- Random player wins: 284
- Draws: 136

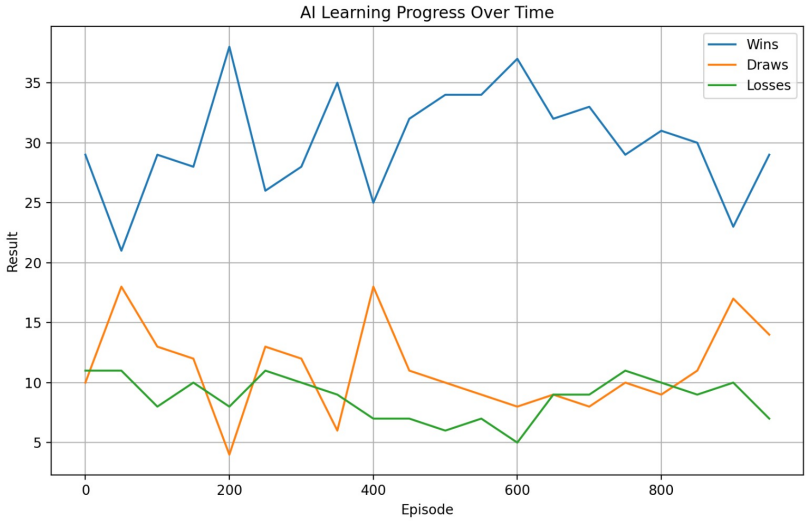

Figure 3: Memoryless DQN performance (1000 samples)

DQN algorithm is also computationally more efficient, as it can run through 10 000 samples in less than 2 minutes compared to extended MCTS 300 games in 3 minutes.

5. Conclusion

In this mini-project, we explored two approaches to more advanced game-playing agents for Tic-Tac-Toe: a hybrid MCTS with deep reinforcement learning and a standalone Deep Q-Network (DQN). Both methods aimed to better the performance of the original MCTS algorithm with different results.

The MCTS+RL approach exceeded the original solution when applied to a 4x4 grid, achieving a 71% win rate compared to the baseline MCTS 26%. However, its performance on the 3x3 grid closely matched the original MCTS scoring 77% vs 79%. The integration of the policy network made this approach also computationally heavy, showcasing limited ability to large-sample training.

In contrast, the DQN approach achieved a 95% win rate on a 3x3 grid outperforming both previous variants. The key to its success was the use of memory replay and epsilon-greedy policy. The DQN's efficiency over 10000 games represents the advantage of direct policy learning over simulation-heavy methods like MCTS. After disabling the memory replay, DQN's performance degraded to MCTS levels, showing the power of experience reuse in reinforcement learning.

This project underscores the potential of deep reinforcement learning by showing the power of the DNQ approach. But it also showed that a balance between computational efficiency and complexity is important when discussing machine learning approaches. It is important to be aware of the complexity of the current task and what approaches and methods are appropriate. We could, for example, see that for a 3x3 Tic-Tac-Toe game, the approach based purely on MCTS performs just as well as the approach combining MCTS with a neural network. Making this implementation questionable for this particular problem.