Reinforcement Learning NN Report

AI for Game Technology - Nicolò Loddo 1531697

This report describes the techniques and findings of the journey towards the training of two Neural Network based agents. The agents were trained to play a game similar to the famous tic-tac toe: the players take turns into placing a marker on a 7x7 square board, the first player that manages to create an L shape (in any of its rotations and mirrored versions) without any opponent marker interruption wins the game. If the board gets completed without any player achieving an L shape, it is called a draw.

Part 1 - *Data generation and Neural Network training*

Various types of agents were already created to play a generalization of this game. Extensively describing the implementation of these agents is out of the scope of this report, but I will name them here because they will be cited in the evaluation of the developed agents:

- RandomAgent: an agent that performs random actions;
- BanditAgent: intelligent agent based on solving multi-armed bandit problems;
- ScoringAgent: intelligent agent choosing an action by giving a score to the state that each action would lead to;
- MCTSAgent: intelligent agent based on the Monte Carlo Tree Search algorithm to explore the tree of possible states and actions.

All those agents were developed by me, except the MCTS agent, developed by the colleague Arjen Simmons, to whom I gave feedback: my MCTS agent in fact would not work properly, and even though I developed a new one for this assignment, this new one has at the core a score function trained by a neural network, really different than the one that I am supposed to use for the evaluation.

The first step in the plan was the generation of the data that would have been used for the training of the first Neural Network.
The generation of such was done by recording the state of the board during various games. The players involved in those games were myself, against both BanditAgent and ScoringAgent, but also, to enrich the data, BanditAgent against ScoringAgent. A total of 500 games were recorded this way.
The recording would involve the state of the board at each move of each player: not only the last winning state of the board. The reason for that will be found in the chosen structure of the Network.
To record each state I developed a function that encodes the board state into specific unique and loss-less IDs.

Following, a step by step example of the encoding function on a game of tic-tac toe:

```
-------        Player 1: O
|OXO|          Player 2: X
|XX  |
|O    |
--------
```

The board would be encoded as one-hot encodings for each player mark position:
Player 1: [1, 0, 1, 0, 0, 0, 1, 0, 0]
Player 2: [0, 1, 0, 1, 1, 0, 0, 0, 0]

Then the two would be concatenated:

```
 [1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0]
         Player 1          ,          Player 2
```

The array is then converted into an integer number:
[1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0] = 166064

This way I encoded the board states to be easily and compactly passed between scripts. N.B.: this integer number is not the form of the input of the Neural Network!

Then I proceeded onto the development of the network.
The first intuition on which I based it (and of which I was curious to test performance on) was to have it train on sequences of board states instead of the singular last winning board state. Therefore I thought of using a Recurrent Neural Network.
The input would be the last 9 moves of each game encoded as the binary numbers above, and the training problem would be 3 classes classification for the outcome of the game: wins player1, wins player2 or it was a draw. After the development of such network and hyperparameters tuning I could not go over 70% of accuracy of predicted outcome. I attributed the cause of this upper limit in accuracy to the sparsity of the data caused by the encoding of the board states.
I therefore changed the model and finally settled for what can be called a Time-Distributed Convolutional Neural Network:
The model takes as input for the training a batch of sequences, each one of 9 board states (the last 9 of each game). Each board state is a 3D Matrix of shape (7, 7, 2), with the third dimension being the channel dimensions. We have one channel per player: each channel's matrix has a 1 where the respective player placed its sign, everywhere else it is 0. The board matrix form is therefore kept the same, never encoded in any

way, with one singular transformation being the separation of the signs of the two players in two channels.
The sequence length choice of 9 moves was directed by the fact that 9 is the minimum number of moves to form an L shape on the table: 5 moves for the winning player and 4 for the losing one. Those last 9 moves therefore usually have the most important information. Moreover, a fixed size length of the sequences was needed to train the model: this seemed the most accurate way to truncate longer games.
The kernel size of the convolutions is of shape (4, 4) and the pooling as well is done with 4 by 4 kernels, this choice was directed by the fact that the winning shape can be completely included in a 4 by 4 matrix, leading to a correct encoding of the winning information by the convoluting filters.
In game predictions are done with a single given board state by replicating it 9 times.
The agent based on this network picks the move that maximizes its predicted winning probability.

```
 Layer (type)                                        Output Shape          Param #
=================================================================
 time_distributed_27 (TimeDistributed Convolutional)  (None, 9, 7, 7, 20)   (340)

 max_pooling3d_27 (MaxPooling3D)                      (None, 9, 2, 2, 20)   0

 batch_normalization_27 (BatchNormalization)          (None, 9, 2, 2, 20)   80

 flatten_27 (Flatten)                                 (None, 720)           0

 dense_33 (Dense)                                     (None, 1)             721
```

After the convolutions, pooling and batch normalization, one fully connected dense layer with a sigmoid activation function gives the predicted output in a binary classification problem format: a number from 0 to 1. 0 means that Player1 will win the game, 1 means Player2 will win the game, 0.5 would mean that there will be a draw.

This model achieved an accuracy of 92% on the test set (33% of the whole dataset) and this Time-Distributed Convolutional Agent performed well against the other agents (200 iterations):

```
99/1/0 (2074.93 seconds) - Against RandomAgent     [wins/losses/draws]
76/24/0 (2500.37 seconds) - Against BanditAgent    [wins/losses/draws]
40/60/0 (2224.66 seconds) - Against ScoreAgent     [wins/losses/draws]
98/2/0 (2255.72 seconds) - Against MCTSAgent       [wins/losses/draws]
```

Part 2 - *Self training Neural Network MCTS Agent (NN-MCTS)*

Another Neural Network agent was developed, with the feature of being able to play by itself, generating data, to then train further on that same data. This agent Neural Network core is encapsulated inside an MCTS Agent, which uses its training for the development of a performant scoring function that would maximize its winnings.
The core Network of this agent would be a slightly modified version of the Time-Distributed Convolutional Network seen before: sharing the structure but being adapted to its specific goal of self training and of being part of an MCTS agent.

The network weights were randomly initialized, then the training would consist of 3 iterations of playing games against a ScoringAgent and then training.
At each iteration 150 games would be played, giving a total of 450 games. This choice was made for two reasons: time (this agent is really slow in making decisions, I chose the maximum amount of games inside the available time), and because the chosen network had already shown good performance with that amount of data.
Following are the results of the 150 games at each learning iteration, highlighting the learning curve of the network (200 iterations for both NN-MCTS and ScoreAgent):
14/136/0 (20897.69 seconds) - [wins/losses/draws]
31/111/8 (28555.06 seconds) - [wins/losses/draws]
38/102/10 (29245.17 seconds) - [wins/losses/draws]

After all 450 training games the agent was tested against all the others (all agents with 200 in-game iterations, 50 games per evaluation):

50/0/0 (4960.95 seconds) - Against RandomAgent	[wins/losses/draws]
28/22/0 (5438.58 seconds) - Against BanditAgent	[wins/losses/draws]
16/33/1 (5835.16 seconds) - Against ScoreAgent	[wins/losses/draws]
50/0/0 (5127.97 seconds) - Against MCTSAgent	[wins/losses/draws]

Conclusion

As a conclusion, Time-Distributed Convolutional Networks seem like a suitable and well performing choice for the purposes described in the report, with the advantage of not needing much preprocessing and flattening of the board states, and performing well also with a restricted amount of games played by exploiting well the information that lies in the last moves of each game. The NN-MCTS agent depends also on parameters like the in-game iterations and the upper confidence bound ‘c’, making it more difficult to tune. The normal Neural Network agent showed a good wins-losses ratio. Both agents were defeated only by the ScoreAgent. I am overall satisfied with the achieved behavior of the developed agents.