Reinforcement Learning and Backgammon

Marie Spreen
Institute of Cognitive Science
University of Osnabrück
Osnabrück, Germany
mspreen@uni-osnabrueck.de

Stefan Warkentin
Institute of Cognitive Science
University of Osnabrück
Osnabrück, Germany
stwarkentin@uni-onabrueck.de

***Abstract*—We make an attempt to recreate Gerald Tesauro's TD-Gammon and to qualitatively compare its performance to that of other reinforcement learning algorithms. Herein we share the methods of our approach, justifying the choices we make along the way, as well as any complications we encounter.**

I. Introduction

Backgammon is board game for two players. Although there is a strong element of chance—rolling the dice is a prominent feature of the game—the rules allow for strategic depth. Backgammon is played competitively . For information on how the game is played, references the rules on bkgm.com

TD-Gammon is a backgammon program developed by Gerald Tesauro, on which he authored two papers: "Practical Issues in Temporal Difference Learning" (1992) and "Temporal Difference Learning and TD-Gammon" (1995). TD-Gammon utilizes temporal difference learning to learn how to play backgammon, and—with no prior knowledge—achieves performance comparable to or even exceeding that of professional backgammon players. TD-Gammon is considered by many to be a great success in the field of reinforcement learning.

It is our present goal to see if we can recreate TD-Gammon based on Tesauro's descriptions, to train several such agents with varying numbers of hidden units and for different amounts of episodes, and to then let it compete against agents based on different algorithms to see how well they fare in comparison.

II. Methods

A. Environment

1) Representing the board: There already exist several backgammon environments which are both freely available and which we consider suited to our needs. Nonetheless, we choose to create an environment of our own, ensuring that we are familiar with the inner workings of our code and facilitating tweaks and changes further down the line.

Using OpenAI's gym framework as our starting point, we begin by encoding the game board in the manner described in "Practical Issues in Temporal Difference Learning". Since TD-Gammon's source code is yet to be made public, we are forced—in places where insufficient detail is provided—to make conjectures. In the case of the encoding scheme, we supplement the description found in Tesauro's paper with the encoding scheme found in another of Tesauro's backgammon players, namely that of his standardized benchmark opponent. We do this on the assumption that Tesauro would have used the same encoding scheme for both. Unlike Tesauro, we choose to represent the state of the board as a dictionary, only flattening the dictionary into an array when necessary. This makes observations provided by the environment easier to interpret for humans while also providing us with a convenient way of indexing them. Furthermore, we deem it convenient to change the way in which board positions are represented. In TD-Gammon, players move their respective pieces in opposite directions, White starting at position 1 and moving through to position 24, and Black doing it the other way around. This is the way backgammon is played on a typical board, and it has the benefit that the same index corresponds to the same position in both board-arrays. However, we decide instead to make both players start at position 1, having both move their pieces in the direction of increasing indices. This removes the need to differentiate between players in the environment's step function at the cost of merely having to use some simple maths when checking for blots.

We end up with the following arrangement: Within our environment the state is represented by a dictionary. Within, there are two more dictionaries ('W' and 'B'), one for each player. Both contain key-value pairs which encode the number of the respective player's pieces at each position of the board ('board'), on the bar ('barmen'), the number of pieces removed from play ('menoff'), and whether it is that players turn ('turn'). The number of pieces at a given position of the board is stored using four values—a truncated unary encoding conceived by Tesauro for this purpose. The number of pieces on the bar stored divided by two, and the number of pieces removed from play is stored as a fraction of 15, which is the total number of pieces each player starts out with.

2) Moving pieces: The environment's step method takes in a list of moves, each a tuple indicating from where a piece should be removed and where it should be placed again—be that the off the board, the bar, or any position on the board itself. Should a blot occur, the opponent's piece is moved to the bar. Subsequently, the environment determines whether one of four winning conditions have been met and updates the state accordingly.

3) Flattening observations and afterstates: Since we cannot feed our observation dictionary to our neural network, we flatten it into an array and convert it into batch form when necessary.

Since our agents rely on afterstates to determine the best course of action, we also implement a way for our environment to compute the outcome of an action without altering the current state.

4) Performance issues: As a consequence of us choosing to represent the state of the environment as a dictionary of dictionaries, we run into unforeseen problems. We find that the easiest way to solve said problems is to create deep copies of our observations wherever necessary, though this simple fix may come at a price. Episodes are slow, and we suspect that our reliance on copies might be inefficient and thus may contribute to said slowness.

B. Network

The neural network which Tesauro trained to create TD-Gammon is described in detail in both Tesauro's work and in "Reinforcement Learning. An Introduction" by Sutton and Barto.

1) Architecture: At first, TD-Gammon was trained using a simple network consisting of only an output layer with four units, one each for every outcome of the game which Tesauro deemed worthy of consideration: White winning, White scoring a gammon, Black winning, Black scoring a gammon. The odds of either White or Black scoring a backgammon was deemed too low to justify reducing the speed at which the network could learn, and so they were subsumed in the cases of either player scoring a gammon. The network would be fed a board state as an input and learn via temporal difference learning to output the approximate probabilities of each outcome coming true. We note that the only activation function used in Tesauro's network was the sigmoid, which—unlike e.g. softmax—does not guarantee that the four output values add up to 1. The network was later expanded by a hidden layer, and the number of hidden units was increased as the program got developed further. Finally, all weights and biases are initialized as random uniform values between -0.5 and +0.5.

We implement the network as described in Tensorflow, creating a class which conveniently generates a network with any number of hidden units we require.

2) Evaluating states: It is not clear to us how exactly Tesauro's agent evaluates the network's output for the purpose of selecting the next action. We decide to sum the four output values weighted by the reward granted to the agent for the associated outcome and we treat the resulting score as a measure of a state's 'goodness'.

C. Agent

1) Basic agent: The basic agent is the super-class all other agents inherit from. As such it contains a method to find all legal actions, which all agents use to choose their actions. Legal actions are dependent on the dice that are rolled and the position the chips are in (i.e. the observation), which means that legal actions need to be evaluated anew for each unique die-roll-observation combination. One legal action consists of a chain of moves. To implement this we decide to conceptualize each action as one branch in the action-tree (each branch being a full action, each node being a single move) that can be searched for legal actions with the help of a recursive function. This function recursively fills an initially empty array with every action or fully traversed branch that fulfills all legal requirements. This means it also has to account for the special cases of having chips barred (i.e. off the board but still in game) and bearing off (i.e. moving chips off the board and out of game). To be able to compare both TD- and DQ-agent to a baseline we also create a random agent that simply chooses a random action.

D. TD-Lambda Agent

1) Selecting actions: When prompted, our TD Agent iterates through all legal actions, evaluates the states which would come about as a result of those actions and selects the one action resulting in the highest score. This is the one-ply approach, so called for the fact that the agent only looks forward in time by one ply i.e. a half step. Tesauro was able to achieve better results using a two-ply approach: The agent evaluates actions based not only on their immediate result, but also on all possible actions its opponent might take as a result. Since training our one-ply agent already takes a long time however, we choose not to pursue this more advanced route.

2) Backpropagation: With no hidden units, we are able to train our agent as is. However, we do not manage to implement TD error backpropagation, meaning that we have to abandon the idea of training our agents using different numbers of hidden units. Though even if we had been able to do so, we suspect that the resulting training times would have been prohibitively long.

E. Deep Q-learning Agent

In deep Q-learning, "Q" refers to the Q-value or state-action-value. The network usually learns the value of each existing state-action pair, taking in a state and an action and returning their value. In order to get a fair comparison between the TD-Lambda and DQ-learning approach and because the random die roll, which is not part of the state, would make it hard to map all possible state-action pairs to a value, we decide to keep the structure of the original network. This means that our "Q-network" is not evaluating state-action pairs, but states. For our DQ-learning step, we decided to use experience replay, meaning that we sample a random minibatch of transitions to perform gradient descent on. For this we kept very closely to this algorithm (Mnih et. al. (2013)):

Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for
```

Since our network evaluates states and not state-action pairs, we have to adjust all uses of the network accordingly. As shown in the algorithm, in each iteration of the game the agent chooses a random action or the action that, in combination with the current state, maximizes the Q-value. Normally we would use a Q-net that outputs Q-values for each possible state-action pair and compare them. In our case we compare each action by the post-state value the current state and action result in. This could be considered a kind of state-action-value, which means we don't run into problems here. For the non-terminal target formation in our learning step we need to use our network to find the maximum Q-value of the post-state (post-state-action-value). This is more difficult, but can be done by choosing the action that would maximize the post-post-state value and using this value as our post-state Q-value.

```python
if len(self.memory.buffer) >= self.batch_size:

    # sample batch
    states, actions, rewards, dones, states_ = \
    self.memory.sample(self.batch_size)

    target = []

    # build targets
    for i in range(self.batch_size):
        if dones[i]:
            target.append(rewards[i] * tf.zeros(4,))
        else:
            # find the max state-action value of the
            # subsequent state
            state__ = \
            self.env.simulate_step(states_[i],
            self.choose_action(states_[i], False))
            state__ = self.env._flatten_obs(state__)
            target.append(rewards[i] + self.gamma *
            self.network(state__.reshape(1,-1))[0])

    states = tf.convert_to_tensor(states)
    target = tf.convert_\to_tensor(target)

    # train network on target batch
    self.network.train_on_batch(states, target)

    # decay epsilon
    self.epsilon = max(self.min_epsilon,
    self.epsilon*self.epsilon_decay)
```

Listing 1. Training Loop DQ-agent

After suspecting that something is wrong with our DQ-learning approach due to the amount of time it takes to train it compared to the TD-agent, we use the cProfile and pstats libraries to analyze the amount of functions called and amount of time per call of both approaches.

```python
import cProfile
import pstats
from pstats import SortKey
from network import Network
from agent import DQNAgent
from env.backgammon import BackgammonEnv
from memory import ReplayBuffer

def main():

    # inititate DQN-agent
    env = BackgammonEnv()
    network = Network()
    gamma = 0.99
    lr = 0.01
    epsilon = 0.9
    min_epsilon = 0.1
    epsilon_decay = 0.99999
    memory = ReplayBuffer(100000)
    batch_size = 20
    episodes = 10
    agent = DQNAgent(env, network, gamma, lr,
    epsilon, min_epsilon, epsilon_decay, memory,
    batch_size)

    # training loop
    for i in range(episodes):
        agent.learn()

if __name__ == "__main__":

    # run main() and collect profiling date
    cProfile.run('main()', "output.dat")

    # sort data by amount of time
    with open("DQN_10_output_time.txt", "w") as f:
        p = pstats.Stats("output.dat", stream=f)
        p.sort_stats("time").print_stats()

    # sort data by amount of calls
    with open("DQN_10_output_calls.txt", "w") as f:
        p = pstats.Stats("output.dat", stream=f)
        p.sort_stats("calls").print_stats()
```

Listing 2. DQ-learning Profiling Loop

We profile both the TD-agent and the DQ-agent for a ten episode training loop, showing that the DQ-agent calls 16 times as many functions as the TD-agent (4672220703 and 293407445) and takes 23 times longer (3844.272 and 167.347 seconds). We realize that in the DQ-learning approach, instead of being able to perform the training step as an update of the state-value through the post-state-value (as we do in the TD-lambda approach), we need to iterate through all post-states, collecting their maximum value action and then predicting the post-post-state-value. This iteration leads to a large increase in total number of functions called, causing the amount of time used for training to go up. We reduce the difference by utilizing tf.keras.Model.predict, which enables us to predict the values of a batch of states. Using the predict-method makes the training loop more efficient, since the network does not need to be called on each state individually but is called on a TF data-set. This improves the amount of calls by 14 percent and the total time by 37 percent.

```python
if len(self.memory.buffer) >= self.batch_size:

    # sample batch
    states, actions, rewards, dones, states_ = \
    self.memory.sample(self.batch_size)

    states__ = []
    for i in range(self.batch_size):
        # find the max state-action value of the
        # subsequent state
        state__ = \
        self.env.simulate_step(states_[i],
        self.choose_action(states_[i], False))
        state__ = self.env._flatten_obs(state__)
        states__.append(state__.reshape(1,-1))

        # reshape dones and rewards
        dones[i] = np.full((1,4), 1 - dones[i])
        rewards[i] = np.full((1,4), rewards[i])

    # convert state__ array to dataset to
    # perform batched prediction
    states__ = tf.data.Dataset.\
    from_tensor_slices(states__).batch(1)
    # build targets
    target = rewards + self.gamma * \
    self.network.predict(states__) * dones

    states = tf.convert_to_tensor(states)
    target = tf.convert_to_tensor(target)

    # train network on target batch
    self.network.train_on_batch(states, target)

    # decay epsilon
    self.epsilon = max(self.min_epsilon, self.
    epsilon*self.epsilon_decay)
```

Listing 3. Improved Training Loop DQ-agent

In addition to lowering the amount of function calls per iteration (and therefore lowering total training time) we also decide to lower our batch-size from 60 to 20, thus lowering the amount of iterations and total training time.

F. *Other agents*

It was our intention to test a greater number of algorithms against our implementation of TD-Gammon. This however proved to be infeasible as a consequence of the issues we encountered while implementing the TD agent and the DQN agent.

G. *Training*

1) Grid: In hopes of shorter training times, we seek to make use of the IKW grid's hardware. We request permission from the admins to remotely connect to the computers in the CIP pool, which allow us to set up Python and GPU-accelerated Tensorflow in our remote directories. While this undoubtedly speeds up our code, the knowledge necessary to make efficient use of these resources is obfuscated by poor documentation. Ultimately, it appears that we lost more time on the grid than we gained from it.

2) Resubmitting jobs: When a job is submitted to the grid, a maximum time span is allotted to it which it cannot exceed lest it is interrupted. Since we expect our training times to exceed that time span many times over, we must find a way to intermittently save our progress and automatically resubmit the job for work to resume. The grid's documentation provides a code example doing just that; however, it relies on Keras' Callback API, which we make no use of in our code. Since we do not wish to rewrite our agents, we modify the code example to work with what we have.

3) DQN Performance: After solving our initial problems with the grid we run into additional performance issues with the DQ-agent. The agent trains very slowly and does not seem to learn very much. After 6158 of 50000 episodes we notice a mistake in the training loop. Since we are using a replay buffer that memorizes all previous experiences (consisting of state, action, reward, done, post-state) and are training on a randomly sampled training batch from that memory, we only start training in the first episode after the memory has been filled to batch-size. As explained previously our job is resubmitted every hour and a half, meaning we re-instantiate agent, environment, replay buffer and network anew every time. To create continuous process we therefore load and save the weights of our network at the beginning and end of each job. Unfortunately we did not account for the same in the replay buffer, which means that with each job submission the replay buffer first needs to be filled before training can begin. An additional misplaced return-statement leads to a very inefficient training result. To solve the problem we implement a way for our memory to be saved and loaded as well. This slows down training even more, since we do not skip over the first episode and train on a more varied batch. Since we spent a lot of time on an unusable training result and need to start the DQ-agent training from scratch, we end up only making it to episode 2000 of the initially planned 50000. In order to evaluate our progress we decide to re-train the TD-agent to episode 2000 as well and compare the two resulting models.

H. *Evaluation*

For the evaluation of the two different models we compared their performances in the game by letting them play against each other and the random agent. Since we trained the DQN-agent up to only 2000 of the full 50000 episodes, but have a fully trained TD-agent, this results in four different games: TD-agent (50000) vs Random agent, TD-agent (2000) vs Random agent, TD-agent (2000) vs DQ-agent and DQ-agent vs Random agent. We run a game loop for each respective game that pits the two agents against each other in 1000 games. During the game we collect the amounts of wins and scored gammons of both player and opponent and present them in a pie-chart.

III. Results

Fig. 1. TD-Agent (50000) vs. Random Agent

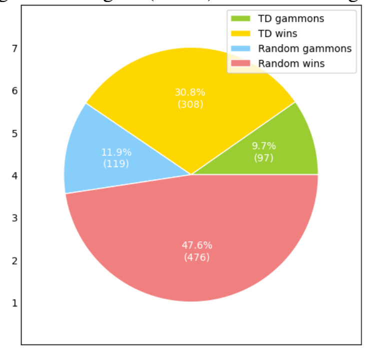


Fig. 2. TD-Agent (2000) vs. Random Agent

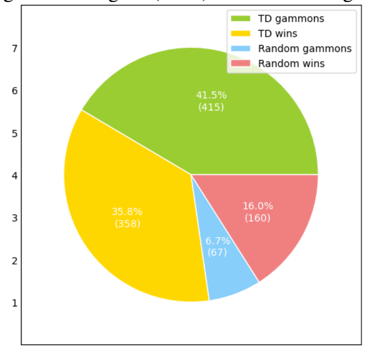


Fig. 3. DQ-Agent vs. Random Agent

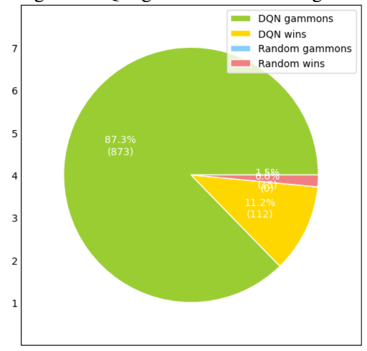
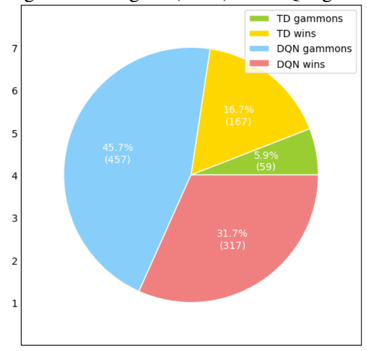


Fig. 4. TD-Agent (2000) vs. DQ-Agent



IV. Discussion

To our surprise the fully trained TD-agent performed very poorly against the random agent. During the training process, intermediary iterations of the agent clearly improved against the random agent as time went on, and the TD agent quite clearly gained the upper hand after only 25000 episodes of training. We believe that some of that progress may have been lost as training progressed further, as is suggested by the fact that the TD agent which trained for merely 2000 episodes did much better. We may have prevented this if we had monitored the agent's performance throughout training, reverting to previous checkpoints when performance worsened. Although we experienced more complications while training the DQ-agent than we did with the TD-agent, it seems to have learned considerably more. After 2000 episodes of training the random agent has barely any chance of winning (15 wins and 0 gammons). In addition, the DQ-agent beats our TD-agent, scoring a gammon almost 50 percent of the time. We assume that this has two reasons. For one our TD-agent performed considerably worse than we had expected, leading us to believe that complications or mistakes might have occurred in the training progress. Another reason could also be that the DQ-learning approach yields better results than the TD-lambda approach, when applied to a problem like backgammon.

Considering the limitations of our work, these results are inconclusive, and further work is required before meaningful statements can be made about which algorithm is superior.

References

[1] Tesauro, G. (1992). ”Practical Issues in Temporal Difference Learning.” doi: 10.1007/BF00992697

[2] Tesauro, G., (1995). ”Temporal Difference Learning and TD-Gammon.”. doi: 10.1145/203330.203343

[3] Sutton, R., Barto, A. (2018). ”Reinforcement Learning. An Introduction.”. Second edition

[4] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M. (2013). Playing atari with deep reinforcement learning. CoRR, abs/1312.5602. doi: 10.48550/arXiv.1312.5602