Introduction to Reinforcement Learning Assignment Deep RL Agents for Simplified Chess

Barış Özakar
Department of Informatics
University of Zurich
baris.oezakar@uzh.ch

***Abstract*—This paper offers two deep reinforcement models (SARSA and DDQN) implemented from scratch without using auto-differentiation libraries to solve the simplified chess problem provided as the course assignment of ”Introduction to Reinforcement Learning” offered at University of Zurich in Spring 2022. The simplified chess environment consists of a 4x4 chessboard and three pieces: 1x King, 1x Queen and 1x Opponent's King, all of them initially placed in random locations on the board.**

***Index Terms*—SARSA, Q-learning, DDQN**

I. Problem Definition

The simplified chess environment consists of a 4x4 chessboard and three pieces: 1x King and 1x Queen for the agent; and 1x King for the opponent, all of them initially placed in random locations on the board. The chess agent interacts with the chess environment by taking series of allowed actions and arriving at new states. The goal of the assignment is to train a chess agent that learn to take actions in order to checkmate the opponent's king, without the game ending in a draw. The opponent king randomly performs actions within its allowed actions at each turn. The simplified chess game ends in a terminal state where the result of the game is either a checkmate in favor of the agent or a draw.

The game of chess, including this simplified version, is a *Markov process*, where the conditional probability distribution of future states of the process depends only upon the present state. In other words, the future is independent of the past given the present [6]. The chess problem lends itself appropriately to the formulation of a *Markov Decision Process* (MDP), where MDPs are an augmented Markov processes extended with action spaces, rewards and the discount factor [9]. Formulating the problem as an MDP assumes that the agent directly observes the entirety of the environment. In such cases, the problem is said to be *fully observable* [4]. The game of chess is a fully observable problem since the agent has access to the state information of all the pieces at each turn of the game.

Formulation of the MDP: The simplified chess MDP can be specified by a tuple (S, A, P, r, μ, γ) of:

- **S**: a finite set of states encoded as a vector of dimensionality $3N^2 + 10 = 58$ where NxN is the size of the board (4x4) and 3 is the number of the pieces involved. Thus, the position of each piece is encoded by a binary chessboard matrix C_{ij} of size 4x4, where $c_{ij} = 1$ if that particular piece is in the position *ij* and c_{ij} = 0 otherwise. The additional 10 dimensions come from the one-hot encoding of the degree of freedom of the opponent King.
- **A**: a finite set of actions which represent the allowed actions of the agent encoded as a 32 dimensional vector concatenated by the degrees of freedom of the agent king and agent queen.
- **P**: state transition probabilities, which represent the transition probabilities from the current state to the next state with the chosen action. For the provided simplified chess environment, the transition probabilities are uniformly distributed across allowed actions of the opponent.
- **r(s, a)**: expected reward function,
- $\boldsymbol{\mu}$: initial state distribution, represents the distribution of the starting states. For the simplified chess environment, the initial state is uniformly distributed with respect to the allowed starting positions of the three pieces.
- γ: discount factor ∈ [0, 1] represents the extent to which the agent considers the rewards in the distant future relative to its immediate future. When the discount factor is 0, the agent will be myopic and concentrate on learning about actions that obtain an immediate reward [3], [8].

For an MDP M = (S, A, P, r, μ, γ) with finite state and action spaces S, A, the size of the deterministic stationary policy set is $|A|^{|S|}$ [9]. Due to the large size of the MDP, it is infeasible to find the exact solution of the problem using dynamic programming solutions that utilize fixed-point iterations such as Value iteration and Policy iteration.

Deep Reinforcement Learning: Due to the large size of the policy set, it is more feasible to train a deep neural network that estimates the action value function Q directly. The Q function denotes the expected return for taking action a starting at state s under policy π.

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\big[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\big]$$

According to the Bellman optimality equation shown below

$$\boxed{Q^*(s,a) = E(r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') \| s_t = s, a_t = a)}$$

if the optimal action-value function Q^* is known, the optimal policy can be found find by acting greedily on it, where the optimal policy π^* is defined as the optimal action to take at every state. By following this optimal policy, the maximum expected return is obtained.

In this setting, the reinforcement learning agent learns to maximize the total rewards from the environment by interacting with it through actions and observing next states and rewards, and updating its belief about the value of the state-action pair. Due to the fact that maximum possible return does not necessarily equate to obtaining the maximum *immediate* rewards, in order to achieve maximum total rewards, this time dependence of future rewards need to be modeled by the discount factor as discounted future rewards. Thus, the agent learns to maximize the sum of discounted rewards.

Data in Reinforcement Learning: In supervised learning, the data is a fixed training and test set consisting of independently and identically distributed data points. In the reinforcement learning setting, the data is collected by interacting with the environment. The observed data consists of trajectories $(s_0, a_0, r_0, s_1, a_1, ...)$. The observed data in reinforcement learning is not i.i.d, because the agent sequentially interacts with the environment, and therefore receives samples which are temporally dependent on previous actions [8].

II. Models

A. *SARSA*

SARSA is a model-free control algorithm that allows the estimation of Q^{π}. The SARSA update rule is shown below:
$Q(s_t, a_t)$ += α_t[r_t + γ Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]
The equation shows that the SARSA estimation of the true Q^{π} is the TD(0) estimate r_t + γ Q(s_{t+1}, a_{t+1}). Moreover, in this *model-free* algorithm, the agent is not directly aware of the state transition dynamics, instead the SARSA agent observes the state transitions and rewards through its interaction with the environment. SARSA is an *on-policy* algorithm since the policy being learnt is the same as the policy being followed. A SARSA agent interacts with the environment and updates the policy based on actions that are actually *taken*. In on-policy learning, the agent has full control over which actions to pick. Therefore, the agent has to tradeoff between exploration and exploitation. SARSA learns a near-optimal policy whilst exploring, therefore in order to learn an optimal policy, we need to carefully pick a suitable exploration strategy [1], [8].

ϵ_t greedy exploration with decay is a method for balancing exploration and exploitation where the agent:

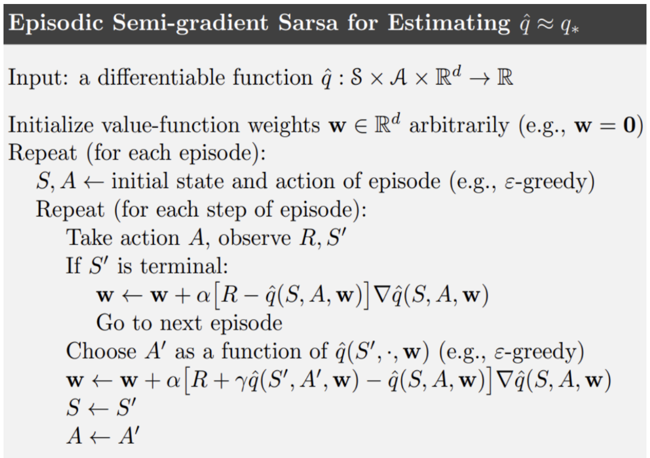
- picks a random action, with probability ϵ_t
- picks the best action with respect to the value estimate, with probability (1-ϵ_t) [8]

I have utilized a decaying ϵ_t greedy approach since exploration is important at the start of the learning for eliminating suboptimal actions by exploring new and better actions. However, as the training continues more exploitation of the already explored optimal actions is needed to maximize the total reward. As can be seen in the code, I applied the decay each episode as such $epsilon_f = epsilon_0/(1 + beta * n)$

SARSA for finite-state and finite-action MDPs converges to the optimal action-value under the following conditions:

- The policy sequence satisfies the condition of GLIE (satisfied by the ϵ_t greedy with decay)
- The step-sizes α_t satisfy the Robbins-Monro sequence

SARSA is a conservative learner since it will tend to avoid a dangerous optimal path if there is risk of a large negative reward close to the optimal path. The conservatism of SARSA is due to the fact that it will calculate values that include the effects of exploration [2]. SARSA will only slowly learn to use the dangerous path when the exploration parameters are reduced. This feature makes SARSA preferable in real life situations where the agent should avoid high risk in order to prevent harm to itself. This feature is demonstrated by the ”Cliff Walking” example shown in class [1].



```
Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable function $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$

Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Repeat (for each episode):
    $S, A \leftarrow$ initial state and action of episode (e.g., $\varepsilon$-greedy)
    Repeat (for each step of episode):
        Take action $A$, observe $R, S'$
        If $S'$ is terminal:
            $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
            Go to next episode
        Choose $A'$ as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., $\varepsilon$-greedy)
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
        $S \leftarrow S'$
        $A \leftarrow A'$
```

Fig. 1. Episodic Semi-Gradient SARSA with Function Approximation [7]

Episodic Semi-Gradient SARSA with Function Approximation: For my SARSA implementation shown in Fig.1 above, I have used the episodic setting where the agent learns over multiple training episodes i, each resulting in a new trajectory, after which the environment resets according to uniformly random initial state distribution. Due to the large state space, I have used function approximation via a neural network with 2 hidden layers with 200 hidden nodes in each. I have made sure to backpropagate the gradients only for the weights relevant to the action taken.

Administration of reward: The dynamics of the learning process is closely tied with the administration of reward in the environment. The provided chess environment originally rewards the checkmate terminal state +1 and the draw state 0, and 0 any other intermediate state. Therefore, in the original environment, there is no dynamic to deter the agent from drawing the game. However, since the agent side is advantageous with a queen at hand, a draw is actually the worst possible outcome for the agent. By introducing a punishment for the draw terminal state of -1, the average percentage of games the agent can be improved drastically. Looking at the Table I, in the original environment, a random agent achieves an average checkmate performance of 19.9%. Here we see that with a draw reward of 0, on average, a random agent gets an ”expected reward” of 0.199, with 80.1% of games ending in draw. Hence, if we were to introduce a draw reward of -1, the ”expected reward” of a random agent would be -0.602.

Table II demonstrates the improvement achieved from utilizing negative reward. Comparing the results of SARSA agents in 0 and -1 draw reward environments, although the average reward is slightly lower for the -1 draw reward agent than that of 0 draw reward (this is expected since we introduce a negative reward that brings down the ”expected reward” of a random agent from 0.199 to -0.602), the percentage of games that ended with a checkmate is significantly higher (from 51.3% to 75.4%). Therefore, we can safely conclude that we achieve a better performing chess agent by training with a negative draw reward. Therefore, all of the remaining agents in this paper have been trained in the negative draw reward environment.

Varying β and γ for SARSA: I have trained several SARSA agents with varying β and γ parameters. The β parameter controls the decay of the ϵ_t greedy exploration. With a higher β, on average, the agent takes less and less random actions to explore as training progresses. From Fig. 2, it can be noted that as the β increases, both the reward per game and the moves per game generally increases. For the reward per game metric, the increase in performance might be due to the fact that the agent is naturally able to eliminate suboptimal actions fairly quickly, therefore concentrating on the optimal actions early on by acting greedily is advantageous. Table III shows that with an increasing β, the average checkmate percentage significantly increases (from 68.3% to 78.8% for β: 0.00001 and 0.0001 respectively) Looking at Table III, with respect to the average number of moves per game metric, we again observe a significant positive correlation, where the average number of moves per game increase as β increases (from 19.6 to 32.9 for β: 0.00001 and 0.0001 respectively). Observing Fig. 3, it can be noted that in the early stages of training (episodes 0-25000), with a higher γ, the reward per game tends to be higher. However, as training progresses, this difference diminishes and the performance of all agents reaches a plateau, with rewards hovering around similar values. Observing the average checkmate percentage of varying γ's from Table IV also shows that the effect of γ to the performance is not significant. For the majority of the training (episodes 0-70000), a higher γ tends to result in lower number of moves per game. However, Table IV shows that when training completes, the average number of moves tend to be around 30, for all three γ values.

Vanilla SGD and ADAM optimizers: The ADAM optimizer makes use of momentum and RMSProp's decaying average of partial gradients and offers an efficient alternative to the SGD optimization. I have implemented an alternative ADAM SARSA agent in order to test if there would be performance improvements. Looking at Table V and the Fig. 4, although there was no improvement in checkmate performance of the agent (SGD: 75.4% ADAM: 72.7%), there was a significant improvement in number of moves per game (SGD: 31.14, ADAM: 9.43). Further finetuning of the ADAM hyperparameters (beta1 and beta2 and epsilon) might be needed to also improve the agent on the performance front.

Normalized Xavier initialization: For the weight initialization of the hidden layer weights, I have chosen

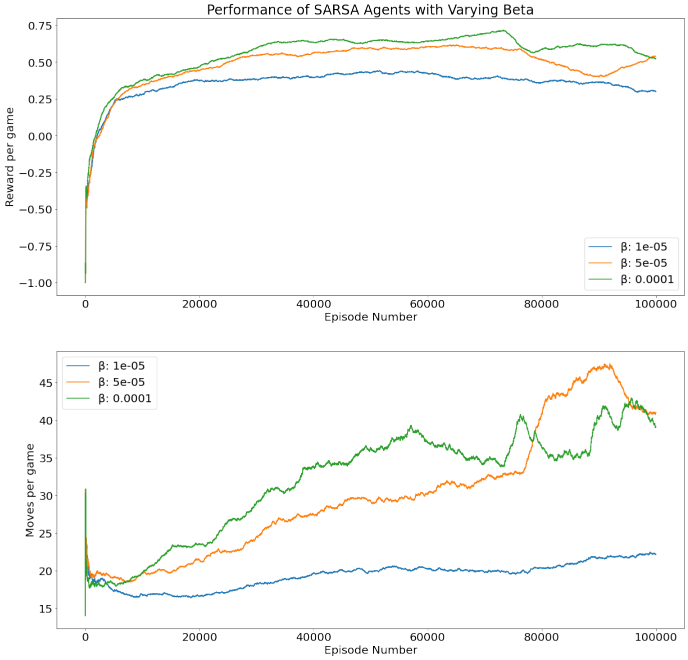


Fig. 2. Performance of SARSA with varying beta

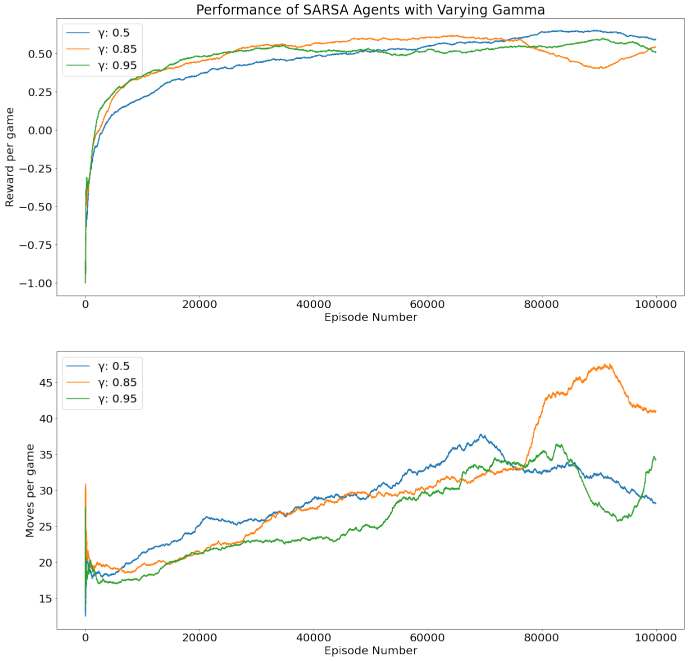


Fig. 3. Performance of SARSA with varying gamma

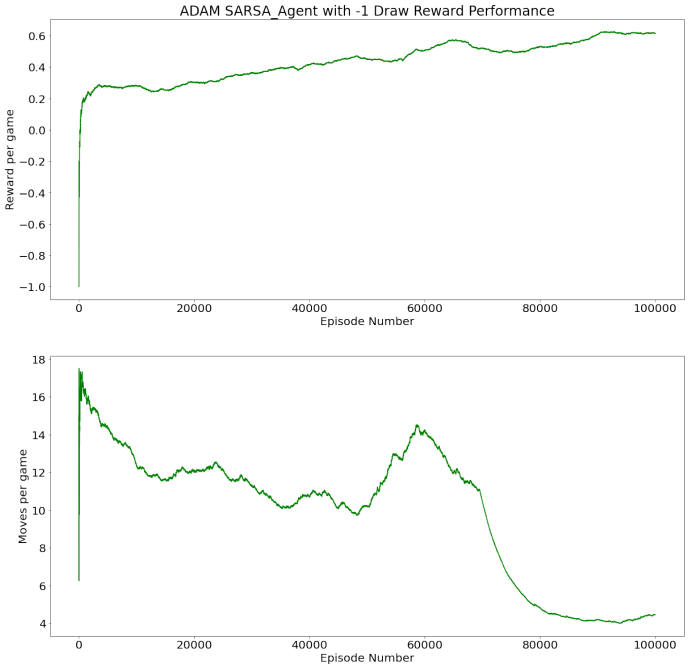


Fig. 4. Performance of ADAM SARSA

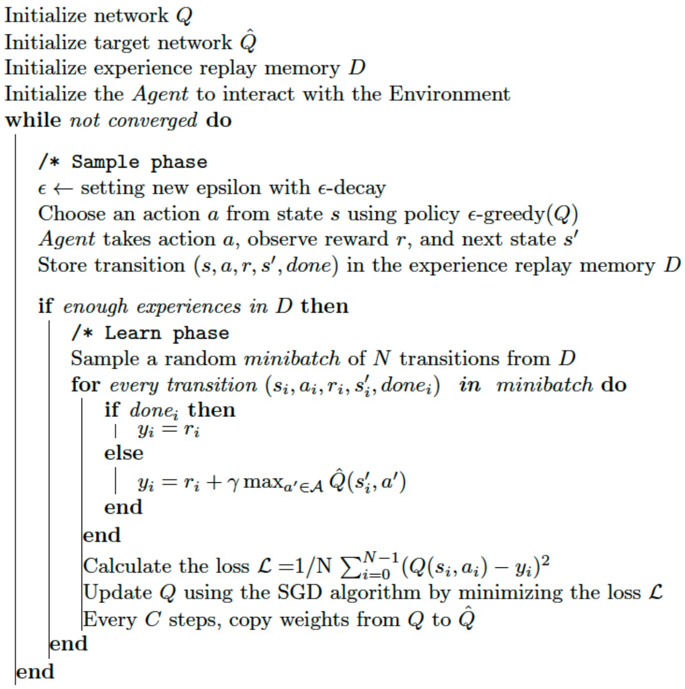
normalized Xavier initialization to prevent layer activation outputs from exploding or vanishing during training. The code snippet demonstrates first layer weights initialized as: $W1 = np.random.normal(0, scale = np.sqrt(6/(N_h + N_{in})), size = (N_h, N_{in}))$

B. *Q-learning*

Q-learning is a model-free control algorithm that allows the estimation of Q^*. Although SARSA and Q-learning are similar in nature, they behave differently. In Q-learning, we directly estimate the optimal Q^*(s,a) with bootstrapping and *off-policy* samples. Thus, Q-learning converges to the *optimal* state-action value function Q^*. Moreover, unlike SARSA, Q-learning is an *off policy* algorithm. Therefore, the observations (s_t, a_t, r_t, s_{t+1}) at time t can come from any behavior policy. The Q-learning update rule is shown below:
$Q(s_t, a_t)$ += α_t[r_t + γ max$_{a'}$ Q(s_{t+1}, a') - Q(s_t, a_t)]
From the update rule, it is apparent that in Q-learning, unlike SARSA, the estimate is updated from the *maximum estimate of all possible next actions*, regardless of which action is taken.
Maximization bias and Double Q-learning: An important problem called the maximization bias arises with the standard Q-learning update rule. This is because standard Q-learning uses same Q-value estimator both to determine the maximization action and to estimate the value of this best action. [8], [9] This results in an upward bias towards the current best action. By using the principle of *Double Q-learning*, we can decouple the choice of maximization action and its estimation by using independent estimates Q_1(s,a) and Q_2(s,a), both of which are neural networks. In the end, two estimators are learnt:

- Q_1(s,a) to select the maximization action
- Q_2(s,a) to estimate the value at the best action

Double Deep Q-learning Network (DDQN): I have used DDQN for my deep Q-learning implementation. DDQN is an algorithm that tackles maximization bias using Double Q-learning by learning two separate deep neural network estimators to select the best action and to evaluate this best action. In my implementation, during training, the target network is fixed for 200 episodes such that the current network has a stationary target. After 200 episodes, the current network is copied into the target network. I have made sure to only backpropagate the gradients for the weights relevant to the action taken.



```
Initialize network $Q$
Initialize target network $\hat{Q}$
Initialize experience replay memory $D$
Initialize the $Agent$ to interact with the Environment
while $not\ converged$ do
    /* Sample phase
    $\epsilon \leftarrow$ setting new epsilon with $\epsilon$-decay
    Choose an action $a$ from state $s$ using policy $\epsilon$-greedy($Q$)
    $Agent$ takes action $a$, observe reward $r$, and next state $s'$
    Store transition $(s, a, r, s', done)$ in the experience replay memory $D$
    if $enough\ experiences\ in\ D$ then
        /* Learn phase
        Sample a random $minibatch$ of $N$ transitions from $D$
        for $every\ transition\ (s_i, a_i, r_i, s'_i, done_i)$ in $minibatch$ do
            if $done_i$ then
                $y_i = r_i$
            else
                $y_i = r_i + \gamma \max_{a' \in A} \hat{Q}(s'_i, a')$
            end
        end
        Calculate the loss $\mathcal{L}$ =1/N $\sum_{i=0}^{N-1}(Q(s_i, a_i) - y_i)^2$
        Update $Q$ using the SGD algorithm by minimizing the loss $\mathcal{L}$
        Every $C$ steps, copy weights from $Q$ to $\hat{Q}$
    end
end
```

Fig. 5. DDQN Algorithm [5]

Similar to the SARSA implementation, I have used a neural network with 2 hidden layers with 200 hidden nodes in each as the estimator.

Experience replay: For the DDQN implementation, I have used the notion of ”experience replay”, where I maintained a replay buffer D of size 10000, consisting of observed transitions (x, a, x', r) obtained by the interactions of the agent with the environment. During training, the agent randomly picks a batch size of 32 observations to learn from. This replay buffer is implemented as a *deque*, where old transitions are replaced by new transitions once the replay buffer is full. Moreover, there is an initial minimum buffer size of 100 such that no learning takes place until this size is reached.

Vectorized implementation In order to improve the execution time of the nonvectorized DDQN, I have vectorized the backpropagation calculations. As a result, the minibatch update is computed more efficiently due to matrix operations. The improvement can be seen in Table V, where the execution time for 100000 episodes has decreased by 48% for my computer specifications (MacBook Pro 2019). Table VII also shows checkmate performance improvement (from 75.7% to 84.1%), possibly due to a more stable gradient update as a result of the minibatch implementation.

Varying β and γ for DDQN: I have trained several DDQN agents with varying β and γ parameters. Similar to the SARSA agent, for DDQN, Fig. 6 shows that as the β increases, both the reward per game and the moves per game generally increases. Table VIII shows that with an increasing β, the average checkmate percentage significantly increases (from 77.6% to 86.8% for β: 0.00001 and 0.0001 respectively) With respect to the average number of moves per game metric, we again observe a significant positive correlation, where the average number of moves per game increase as β increases (from 20.6 to 53.2). Observing the average checkmate percentage of varying γ's from Table IX and Fig. 6 shows that the effect of γ to the performance and number of moves per game is not significant. Table IX shows that the average number of moves tend to be around 42, for all three γ values.

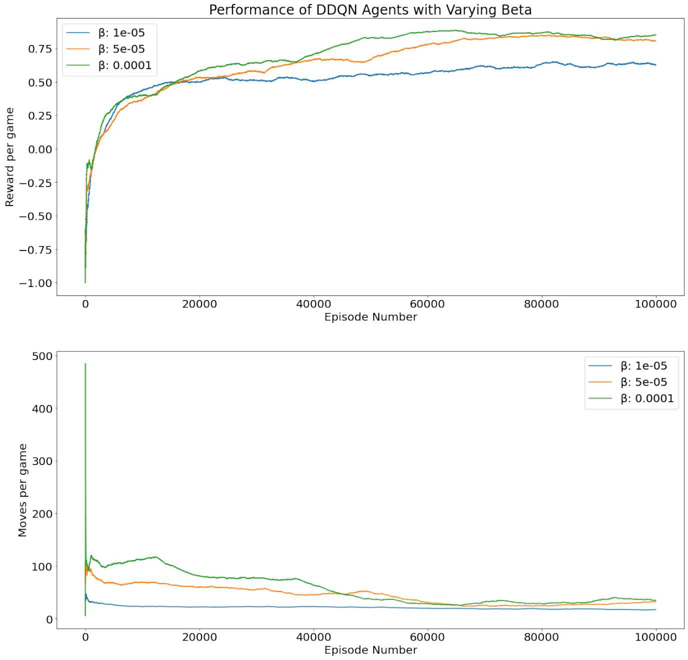


Fig. 6. Performance of Vectorized DDQN with varying beta

III. Results and Reproducibility

This report highlights several agents and their performances according to several metrics; as well as the effects of several hyperparameters to the performance of the trained agents. Since the main objective of the chess agent is to checkmate, Tables III and VIII show that DDQN agents were more successful. This could be due to the fact that Q-learning directly learns the optimal Q value, hence the optimal policy. Since the chess environment is a simulation without real world consequences such as harm to the agent or monetary expenses, deep Q-learning provided a more suitable solution. The DDQN model with the vectorized implementation (β: 0.0001, γ: 0.85) demonstrated a superior overall average checkmate performance of 86.8% and 0.74 average reward. However, ADAM SARSA model provided an alternative fast-paced solution with only 9.4 average number of moves per game yet with an inferior 72.7% checkmate performance. This paper can be improved by further exploring other network topologies as well as further hyperparameter tuning. The code has been implemented such that every result is reproducible by a random seed set to 2022 for each cell. Another essential point for reproducibility is to import the alternative chess environment ”Chess_env_with_draw_punishment.py” I have created with negative draw rewards.

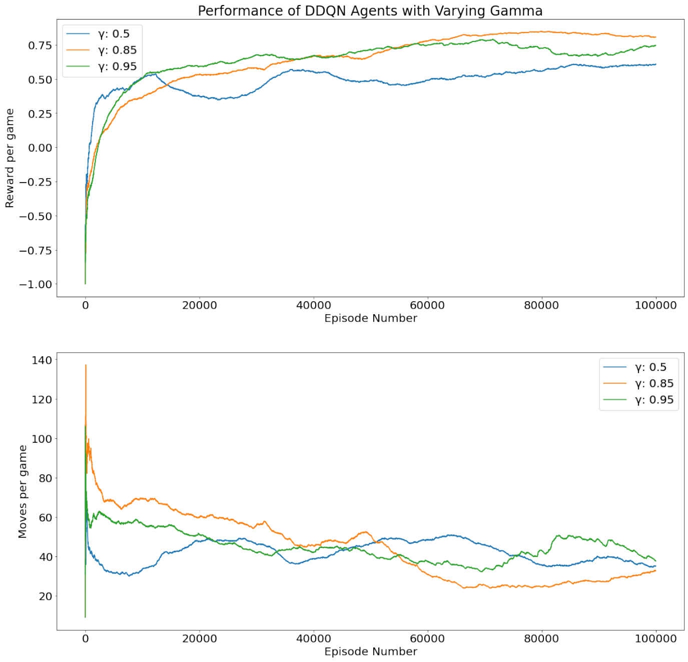


Fig. 7. Performance of Vectorized DDQN with varying gamma

Appendix

A. Tables

TABLE I
Random Agent Draw Reward Comparison

	Random Agent (R_{draw}=0)	Random Agent (R_{draw}=-1)
Average reward	0.199	-0.602
Average number of steps	6.9736	6.9736
Draw %	80.1%	80.1%
Checkmate %	19.9%	19.9%

TABLE II
SARSA Agent Draw Reward Comparison

	Vanilla SGD SARSA (R_{draw}=0)	Vanilla SARSA (R_{draw}=-1)
Average reward	0.51348	0.50888
Average number of steps	10.88195	31.14282
Draw %	48.7%	24.6%
Checkmate %	51.3%	75.4%

β: 0.00005
γ: 0.85

TABLE III
PERFORMANCE OF SARSA AGENT WITH VARYING BETA

	beta: 0.00001	beta: 0.00005	beta: 0.0001
Average reward	0.36664	0.50888	0.57584
Average number of moves	19.56304	31.14282	32.94545
Draw %	31.7%	24.6%	21.2%
Checkmate %	68.3%	75.4%	78.8%

γ: 0.85

TABLE IV
PERFORMANCE OF SARSA AGENT WITH VARYING GAMMA

	gamma: 0.5	gamma: 0.85	gamma: 0.95
Average reward	0.4939	0.50888	0.50116
Average number of moves	29.19442	31.14282	26.82057
Draw %	25.3%	24.6%	24.9%
Checkmate %	74.7%	75.4%	75.1%

β: 0.00005

TABLE V
ADAM AND VANILLA SGD PERFORMANCE

	Vanilla SARSA	ADAM SARSA
Average reward	0.50888	0.45452
Average number of moves	31.14282	9.42893
Draw %	24.6%	27.3%
Checkmate %	75.4%	72.7%

β: 0.00005
γ: 0.85

TABLE VI
EXECUTION TIME OF VECTORIZED AND NONVECTORIZED DDQN IMPLEMENTATIONS

	DDQN SGD Nonvectorized	DDQN Minibatch Vectorized
Number of episodes	100000	100000
Iteration per second	8.99 it/s	15.59 it/s
Total execution time	3:05:24	1:46:55

β: 0.00005
γ: 0.85

TABLE VII
DDQN PERFORMANCE

	DDQN SGD Nonvectorized	DDQN Minibatch Vectorized
Average reward	0.51496	0.68298
Average number of moves	32.13215	42.00092
Draw %	24.3%	15.9%
Checkmate %	75.7%	84.1%

β: 0.00005
γ: 0.85

TABLE VIII
PERFORMANCE OF DDQN AGENT WITH VARYING BETA

	beta: 0.00001	beta: 0.00005	beta: 0.0001
Average reward	0.55162	0.68298	0.73636
Average number of moves	20.63585	42.00092	53.17906
Draw %	22.4%	15.9%	13.2%
Checkmate %	77.6%	84.1%	86.8%

γ: 0.85

TABLE IX
PERFORMANCE OF DDQN AGENT WITH VARYING GAMMA

	gamma: 0.5	gamma: 0.85	gamma: 0.95
Average reward	0.50748	0.68298	0.66872
Average number of moves	41.66203	42.00092	43.82187
Draw %	24.6%	15.9%	16.6%
Checkmate %	75.4%	84.1%	83.4%

β: 0.00005

B. Code snippets

```
for n in tqdm(range(N_episodes)):
    # DECAYING EPSILON
    epsilon_f = epsilon_0 / (1 + beta * n)
    # SET DONE TO ZERO (BEGINNING OF THE EPISODE)
    Done=0
    # COUNTER FOR NUMBER OF ACTIONS
    i = 1
    # INITIALISE GAME
    S, X, allowed_a = env.Initialise_game()
    # START THE EPISODE
    while Done==0:
        # Find the Qvalues corresponding to that state
        # Neural activation: input layer -> hidden layer
        h1 = np.dot(W1,X)+bias_W1
        # Apply the sigmoid function
        x1 = 1/(1+np.exp(-h1))
        # Neural activation: hidden layer -> output layer
        h2 = np.dot(W2,x1)+bias_W2
        # Apply the sigmoid function
        Qvalues = 1/(1+np.exp(-h2))
        # Make an action
        action = EpsilonGreedy_Policy(Qvalues, allowed_a.reshape(-1), epsilon_f)
        S_next, X_next, allowed_a_next, R, Done = env.OneStep(action)
        # THE EPISODE HAS ENDED, UPDATE...BE CAREFUL, THIS IS THE LAST STEP OF THE EPISODE
        if Done==1:
            if R==1:
                Checkmate_save_SARSA[n] = np.copy(1)
                Draw_save_SARSA[n] = np.copy(0)
            else:
                Checkmate_save_SARSA[n] = np.copy(0)
                Draw_save_SARSA[n] = np.copy(1)
            R_save_SARSA[n] = np.copy(R)
            N_moves_save_SARSA[n] = np.copy(i)
            # Compute the error signal
            e_n = R - Qvalues[action]
            # Backpropagation: output layer -> hidden layer
            delta2 = Qvalues*(1-Qvalues) * e_n
            dW2 = np.outer(delta2, x1)
            dbias_W2 = delta2
            # Backpropagation: hidden layer -> input layer
            delta1 = x1*(1-x1) * np.dot(W2.T, delta2)
            dW1 = np.outer(delta1, X)
            dbias_W1 = delta1
            #Update W1 and W2 only for the action taken
            W2[action, :] += eta*dW2[action, :]
            W1[action, :] += eta*dW1[action, :]
            bias_W2[action] += eta*dbias_W2[action]
            bias_W1[action] += eta*dbias_W1[action]
            break;
        # IF THE EPISODE IS NOT OVER...
        else:
            # Neural activation: input layer -> hidden layer
            h1 = np.matmul(W1, X_next)+bias_W1
            # Apply the sigmoid function
            x1 = 1/(1+np.exp(-h1))
            # Neural activation: hidden layer -> output layer
            h2 = np.dot(W2,x1)+bias_W2
            # Apply the sigmoid function
            next_Qvalues = 1/(1+np.exp(-h2))
            next_action = EpsilonGreedy_Policy(next_Qvalues, allowed_a_next.reshape(-1), epsilon_f)
            # Compute the error signal
            e_n = (R + gamma * next_Qvalues[next_action] - Qvalues[action])
            # Backpropagation: output layer -> hidden layer
            delta2 = Qvalues*(1-Qvalues) * e_n
            dW2 = np.outer(delta2, x1)
            dbias_W2 = delta2
            # Backpropagation: hidden layer -> input layer
            delta1 = x1*(1-x1) * np.dot(W2.T, delta2)
            dW1 = np.outer(delta1, X)
            dbias_W1 = delta1
            #Update W1 and W2; b1 and b2 only for the action taken
            W2[action, :] += eta*dW2[action, :]
            W1[action, :] += eta*dW1[action, :]
            bias_W2[action] += eta*dbias_W2[action]
            bias_W1[action] += eta*dbias_W1 [action]
            # NEXT STATE AND CO. BECOME ACTUAL STATE...
            S = np.copy(S_next)
            X = np.copy(X_next)
            allowed_a = np.copy(allowed_a_next)
        # UPDATE COUNTER FOR NUMBER OF ACTIONS
        i += 1
```

Listing 1. Vanilla SARSA

```
## Here we define the ADAM optimizer
class Adam:
    def __init__(self, Params, beta1):
        # It finds out if the parameters given are in a vector
        N_dim=np.shape(np.shape(Params))[0]
        (N_dim=1) or a matrix (N_dim=2)
        # INITIALISATION OF THE MOMENTUMS
        if N_dim==1:
            self.N1=np.shape(Params)[0]
            self.mt=np.zeros([self.N1])
            self.vt=np.zeros([self.N1])
        if N_dim==2:
            self.N1=np.shape(Params)[0]
            self.N2=np.shape(Params)[1]
            self.mt=np.zeros([self.N1,self.N2])
            self.vt=np.zeros([self.N1,self.N2])
        # HYPERPARAMETERS OF ADAM
        self.beta1=beta1
        self.beta2=0.999
        self.epsilon=0.001
        # COUNTER OF THE TRAINING PROCESS
        self.counter=0
    def Compute(self,Grads):
        self.counter=self.counter+1
        self.mt=self.beta1*self.mt+(1-self.beta1)*Grads
        self.vt=self.beta2*self.vt+(1-self.beta2)*Grads**2
        mt_n=self.mt/(1-self.beta1**self.counter)
        vt_n=self.vt/(1-self.beta2**self.counter)
        New_grads=mt_n/(np.sqrt(vt_n)+self.epsilon)
        return New_grads
```

Listing 2. Adam optimizer

```
min_buffer_size= 100
D = collections.deque(maxlen = N_experiences)
counter_to_copy = 0
time_to_copy = 200
episodes_elapsed = 0
S, X, allowed_a = env.Initialise_game()
## COUNTER FOR NUMBER OF ACTIONS
i = 0
pbar = tqdm(total = N_episodes)
while episodes_elapsed < N_episodes:
    # Find the Qvalues corresponding to that state
    # Neural activation: input layer -> hidden layer
    h1 = np.dot(W1,X) + bias_W1
    # Apply the sigmoid function
    x1 = 1/(1+np.exp(-h1))
    # Neural activation: hidden layer -> output layer
    h2 = np.dot(W2,x1)+bias_W2
    # Apply the sigmoid function
    Qvalues = 1/(1+np.exp(-h2))
    # Make an action
    action = EpsilonGreedy_Policy(Qvalues, allowed_a.reshape(-1), epsilon_f)
    S_next, X_next, allowed_a_next, R, Done = env.OneStep(action)
    #Store (s,a,r,s',Done) into buffer
    D.append((X, action, R, X_next, Done))
    # NEXT STATE AND CO. BECOME ACTUAL STATE...
    S = np.copy(S_next)
    X = np.copy(X_next)
    allowed_a = np.copy(allowed_a_next)
    i+=1
    if Done:
        if R==1:   #if checkmate
            Checkmate_save_DDQN_V[episodes_elapsed] = np.copy(1)
            Draw_save_DDQN_V[episodes_elapsed] = np.copy(0)
        else:     #if draw
            Checkmate_save_DDQN_V[episodes_elapsed] = np.copy(0)
            Draw_save_DDQN_V[episodes_elapsed] = np.copy(1)
        R_save_DDQN_V[episodes_elapsed] = np.copy(R)
        N_moves_save_DDQN_V[episodes_elapsed] = np.copy(i)
        episodes_elapsed += 1
        S, X, allowed_a = env.Initialise_game()
        i=0
        epsilon_f = epsilon_0 / (1 + beta * n)    ## DECAYING EPSILON
        pbar.update(1)
    if len(D)<min_buffer_size:
        continue
    #This code runs after the buffer has enough samples to start training
    counter_to_copy += 1
    #Sample a random minibatch of experiences from D
    indices = np.random.randint(0, len(D), size=batch_size)
    sampled_minibatch = [D[i] for i in indices]
    batch_array = np.array(sampled_minibatch, dtype=object)
    X_batch = np.transpose(np.stack(batch_array[:, 0]))
    action_batch = np.array(batch_array[:, 1], dtype=int)
    R_batch = np.array(batch_array[:, 2], dtype=int)
    X_next_batch = np.transpose(np.stack(np.array(list(map(lambda row: row if len(row)>0 else np.zeros(X_batch.shape[0], dtype=int), batch_array[:, 3])), dtype=int)))
    Done_batch = np.array(batch_array[:, 4], dtype=int)
    #Compute the Q values
    h1 = np.dot(W1,X_batch) + np.tile(bias_W1, reps=[batch_size,1]).T
    # Apply the sigmoid function
    X1 = 1/(1+np.exp(-h1))
    # Neural activation: hidden layer -> output layer
    h2 = np.dot(W2,X1) + np.tile(bias_W2, reps=[batch_size,1]).T
    # Apply the sigmoid function
    Qvalues_batch = 1/(1+np.exp(-h2))
    #Calculate one-hot-encoding of the actions
    one_hot_action_batch = np.eye(32)[np.array(action_batch).reshape(-1)].T
    #Find the vector of Q values of the actions taken for each sample in the batch
    Q_taken_actions_batch = np.max(one_hot_action_batch * Qvalues_batch , axis=0)
    #Calculate Q_target using X_next_batch
    h1_next = np.dot(W1_target,X_next_batch)+np.tile(bias_W1_target, reps=[batch_size,1]).T
    # Apply the sigmoid function
    X1_next = 1/(1+np.exp(-h1_next))
    # Neural activation: hidden layer -> output layer
    h2_next = np.dot(W2_target,X1_next)+np.tile(bias_W2_target, reps=[batch_size,1]).T
    # Apply the sigmoid function
    Qvalues_target_batch = 1/(1+np.exp(-h2_next))
    #Calculate the error signal (combines Done=0 and Done=1 computations)
    e_n = one_hot_action_batch * (R_batch + gamma * (1-Done_batch) * np.max(Qvalues_target_batch, axis=0)  - Q_taken_actions_batch)
    #Realize: we use apply the backpropagation to the online Q network not the target network
    delta2 = e_n * Qvalues_batch * (1-Qvalues_batch)
    dbias_W2 = delta2
    dW2 = np.dot(delta2, X1.T)
    delta1 = X1*(1-X1) * np.dot(W2.T, delta2)
    dbias_W1 = delta1
    dW1 = np.dot(delta1, X_batch.T)
    #Update W1 and W2 only for the action taken
    W2 += eta*dW2/batch_size
    W1 += eta*dW1/batch_size
    bias_W2 += eta*np.mean(dbias_W2, axis=1)
    bias_W1 += eta*np.mean(dbias_W1, axis=1)
    #copy current network to target network
    if counter_to_copy % time_to_copy ==0:
        W1_target = np.copy(W1)
        W2_target = np.copy(W2)
        bias_W1_target = np.copy(bias_W1)
        bias_W2_target = np.copy(bias_W2)
pbar.close()
```

Listing 3. DDQN Vectorized implementation

REFERENCES

[1] N. Slater, ”When to choose SARSA vs. Q Learning”, Cross Validated, 2022. [Online]. Available: https://stats.stackexchange.com/questions/326788/when-to-choose-sarsa-vs-q-learning. [Accessed: 29- Mar- 2022].

[2] N. Slater, ”Why update SARSA with S’A’ at all if the goal is a less aggressive exploitation policy?”, Cross Validated, 2022. [Online]. Available: https://stats.stackexchange.com/questions/361485/why-update-sarsa-with-sa-at-all-if-the-goal-is-a-less-aggressive-exploitation. [Accessed: 31- Mar- 2022].

[3] Wright, ”Understanding the role of the discount factor in reinforcement learning”, Cross Validated, 2022. [Online]. Available: https://stats.stackexchange.com/questions/221402/understanding-the-role-of-the-discount-factor-in-reinforcement-learning. [Accessed: 29- Mar- 2022].

[4] Techniques in Artificial Intelligence — Markov Decision Processes. MIT OpenCourseWare, 2002.

[5] J. Torres, ”Deep Q-Network (DQN)-II”, Medium, 2022. [Online]. Available: https://towardsdatascience.com/deep-q-network-dqn-ii-b6bf911b6b2c. [Accessed: 29- Mar- 2022].

[6] ”Introduction to Reinforcement Learning : Markov-Decision Process”, Medium, 2022. [Online]. Available: https://towardsdatascience.com/introduction-to-reinforcement-learning-markov-decision-process-44c533ebf8da. [Accessed: 29- Mar- 2022].

[7] R. Sutton and A. Barto, Reinforcement learning. 2020, p. 244.

[8] A. Krause, ”Probabilistic Artificial Intelligence”, 2021.

[9] N. He, ”Foundations of Reinforcement Learning”, 2021.