Reviewing Deep Reinforcement Learning Approaches in Chess

1st Finn Williams

***Abstract*—Reinforcement learning enables an agent to navigate an environment effectively, by interacting with it to learn the optimal set of actions to take. This paper explores and implements deep Q learning and deep SARSA to attempt to solve a simplified version of chess.**

I. Introduction

Q learning [**?**] is a reinforcement learning algorithm aimed with finding the best series of actions to take in an environment. It is "off-policy", meaning it aims to learn its own policy (strategy used to make decisions in an environment) by learning from data. It is also "model-free" meaning it does not rely on the rewards produced by the environment to learn; it uses a prediction of the environment reward with trial and error to learn its own policy. It achieves this by implementing and iteratively building a Q matrix which represents the best actions to take at a given state, with a higher value indicating a better action. The Q matrix is updated using the Bellman equation:

$$Q(s,a) = \alpha(r + \gamma \ * \ max_{a\prime} \, Q(s\prime, a\prime) \ - \ Q(s,a)) \qquad (1)$$

Where s and a are the current state and current action respectively, α is the learning rate, r is the reward obtained (in this instance a "1" for a checkmate, or a "0" otherwise), γ is the discount factor, s' and a' are the next state and action respectively, and $max'_a \ Q(s', a')$ is the maximum Q value for the next state and actions.

One factor that enables Q learning to perform well is the epsilon-greedy policy, which is responsible for selecting an action. It aims to provide an answer to the reinforcement learning principle of "exploration vs exploitation"; this is the compromise of time spent between exploring the environment and taking new actions, compared to learning from the information it has already obtained.

One drawback of Q learning is the need to explore the environment multiple times over to gain a sufficient understanding of the environment and the task. For the experiments discussed in this paper, 100,000 games of chess were played, which took a considerable amount of time – approximately 1 hour 10 mins. In addition to this, the epsilon-greedy policy does not provide an adequate solution to the exploration vs exploitation principle. In particular, the Bellman equation(1) always selects the maximum Q-value in a greedy fashion. As a result, it lacks the exploration factor, meaning an optimal solution may not always be reached, and may become stuck in a local minimum. The SARSA [**?**] algorithm attempts to overcome this. It is a modified version of the Q learning algorithm that aims to motivate the agent to explore more through the modification of the Bellman equation:

$$Q(s,a) = \alpha(r + \gamma \ * \ Q(s\prime, a\prime) \ - \ Q(s,a) \qquad (2)$$

By removing the decision to always take the maximum Q value, the agent relies less on the exploitation of previously learned knowledge and aims to explore the environment more by taking the current proposed action and state. Compared to Q learning, SARSA is an on-policy algorithm. As described by the updated Bellman equation (2), it differs from off-policy algorithm using the same policy that is used to explore the environment. As a result, one drawback of SARSA compared to Q learning is for its tendency to converge more slowly.

A limitation of both algorithms is the reliance on a Q table – a large matrix – to store values in that correspond to desired actions. In particular, both methods suffer from the curse of dimensionality; the Q table grows exponentially in size proportionally to the size of the environment. Specifically for a game of chess, which involves around 10^4 different combinations of moves, the size of the Q table needed, and not to mention the time taken to optimally update said Q table, is infeasible. To overcome this, deep Q learning, and similarly deep Q SARSA, aim to replace the Q table with a deep neural network capable of storing a learned dense representation of the environment.

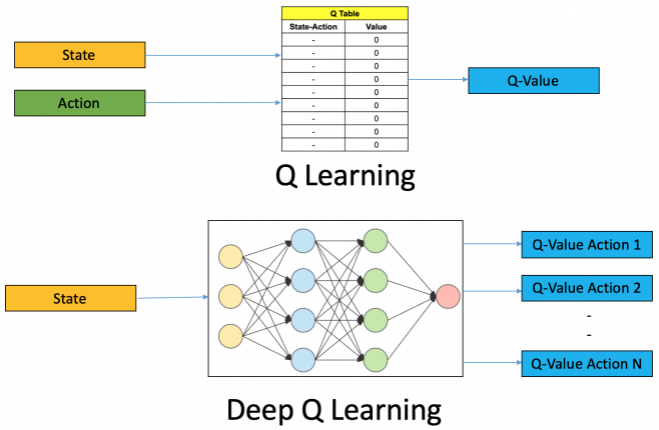

Fig. 1. [**?**] A comparison between vanilla Q learning and deep Q learning

In place of extracting values from the Q table, both algorithms pass the environment data through a dense feed-forward neural network and use the output as the Q values.

In particular, epsilon-greedy selects the largest output node or a random output node which corresponds to the Q value. To update the Q table, or in this case to optimise the neural network, it is trained using backpropagation. The Bellman equations along with the reward of a 0 or a 1 is used as a loss to measure the performance of the network, of which will be used to update the parameters:

$$L = \frac{1}{2}((r + \gamma \ * \ max'_a \ Q(s', a') \ - \ Q(s, a)) \ - \ \hat{y})^2 \quad (3)$$

$$L = \frac{1}{2}((r + \gamma \ * \ Q(s', a') \ - \ Q(s, a)) \ - \ \hat{y})^2 \quad (4)$$

Where $\hat{y}$ is the output layer of the neural network, and represent the predicted Q values. Some disadvantages of a neural-based approach include an increase in time to train. This is due to the fact that passing values through the network is more inefficient than indexing them from a Q table. In addition to this, the deep learning implementations require an extra step to train the parameters of the network which is time consuming and costly for a single episode. However by using common machine learning practices such as early stopping, there is a potential for the overall training job to not run for as many episodes and terminate faster, despite the time for a single episode to terminate being much greater.

To explore the differences between these approaches and to solve the simplified chess problem, this paper documents the design and implementation of deep Q learning and deep SARSA from a bottom-up perspective, discussing the training methods and differences between implementation in detail. Additionally this paper reports on a range of conducted experiments, testing out various hyperparameters and changes in algorithmic design.

II. Method

The neural network architecture this paper implements consists of an input layer of size 58, which accepts as input the chess board feature matrix of the same size. This is then fed into a hidden layer of size 200 which aims to learn a dense representation of the environment in place of the Q table. Lastly, the network features an output layer of size 32 to extract information from the hidden layer. The benefit of using an output layer smaller than the hidden and input layer means that the network can generalise better, forcing it to capture only the most relevant information from the environment. All relevant layers implement the ReLU (Rectified Linear Unit) nonlinearity as it is a proven high-performing activation function in the machine learning space. Once forward propagation has occurred, the epsilon-greedy policy selects either a random node or the largest value node from the output layer to act as the Q value for the respective state and action. We implement a decaying epsilon value which is set to a larger value at the start of training to favour exploration, and gradually reduces this value over time to focus more on training. The derivation of both the forward propagation and backpropagation stages can be found in the appendix.

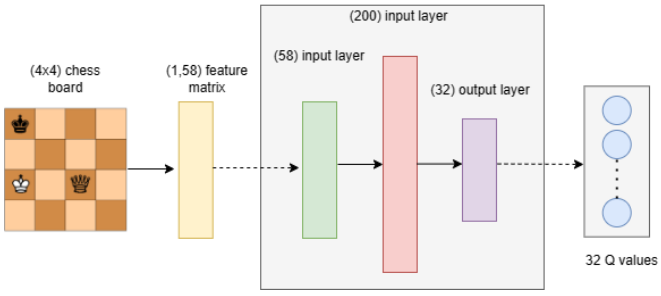

Fig. 2. Network architecture

This implementation presents us with a variety of experimental options to undertake. The first of which is to carry out the training of deep Q learning and deep SARSA using their respective vanilla implementations, as described above. These versions use an epsilon_0 value of 0.2. This initial value is small in effect as it means there is at most a 20% chance of exploring the environment on the first episode (only to reduce later). To explore the epsilon-greedy policy in more detail, we train deep Q learning and deep SARSA again with a higher epsilon_0 value of 0.4, to emphasise exploration more. There is currently no limit within the implementation to how many moves can occur within the game, meaning that many turns can occur in succession without a checkmate or a draw. Not only does this increase the training time, but also means the algorithms are less likely to converge. To overcome this issue, we seek to train both algorithms using a maximum move setting of 20 and 10.

Finally, other implementations seek to only update the weight values that correspond to the action taken, rather than the entire weight matrix. We carry out final implementations which explore this design choice, using both the mean and the max of the associated weight delta to allow for the matrix shapes to be broadcasted together. To measure the success of these experiments, we plot graphs of the number of actions taken per game over time, and the reward over time, both smoothed using an exponential moving average. All variations are applied to the vanilla implementation, and ran for 100,000 episodes. Code samples demonstrating the implementations of forward propagation, backpropagation and the epsilon-greedy policy can be found in the appendix. The table below summarises the experiments carried out in this paper:

It would be expected that using 0.4 as a value for epsilon would increase the performance. This is because the agent would be able to spend more time exploring at the start of training, and can then optimise later. Updating only the weight value that corresponds to the respective action taken would make the network train faster as only this single value needs to be updated, rather than the entire weight matrix. Using a mean to capture the average delta would be more representative, however using a maximum would allow the weight to be updated with the largest and therefore most relevant gradient. For this reason it is predicted that the max operation will perform better than the mean. Finally, it is expected that setting

TABLE I
EXPERIMENTS

Experiment	Algorithm
Vanilla 0.2 epsilon	Q learning
	SARSA
0.4 epsilon	Q learning
	SARSA
Mean action weight	Q learning
	SARSA
Max action weight	Q learning
	SARSA
Maximum 10 moves	Q learning
	SARSA

the moves to a maximum of 10 will not only reduce the time taken to train, but also improve the results, as it discourages the network to learn to keep playing chess, as to avoid drawing and consequentially avoid a checkmate.

Other noteworthy parameters include γ and β. γ controls the discount factor – the importance of future rewards relative to immediate rewards within the Bellman equations - and β controls the speed at which epsilon decays. As discussed with changing the epsilon_0 value, a larger value for β would result in less time spent exploring the environment, and vice versa. For this reason, it would be expected that a better solution overall would be reached, as the agent has spent more time exploring the environment. In a similar sense, γ controls how much the agent values a short-term reward compared to a long-term future reward. Setting this value higher would allow for the agent to think ahead and receive rewards for exploration. Epsilon greedy is responsible for deciding what actions the agent takes, whereas the Bellman equations decide what reward the agent should receive.

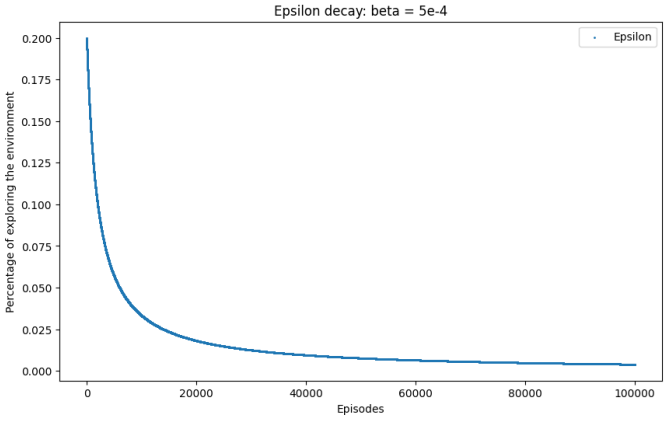

Fig. 3. Rate of decay of epsilon using a value of 5e-4

A. Reproducing Results

The code can be found in the following GitHub: https://github.com/finnwilliams16/adaptive-intelligence. For access please contact the author: finnw70@gmail.com. The same results as this paper can be reproduced by using a seed value of 0.

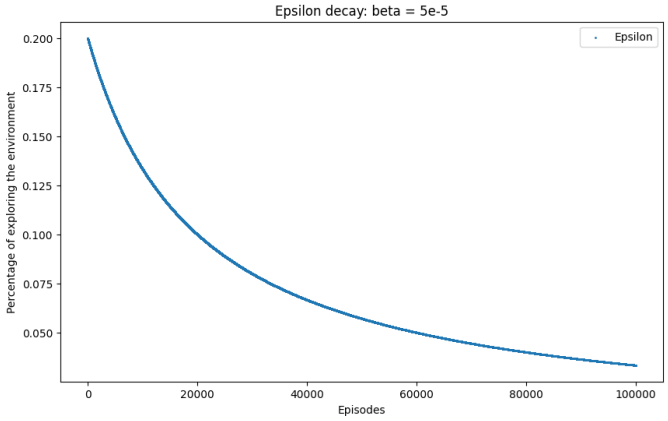

Fig. 4. Rate of decay of epsilon using a value of 5e-5

III. RESULTS

Below is a table displaying the results for the various experiments conducted:

TABLE II
EXPERIMENTS

	Experimental Results		
	Algorithm	*Average Reward*	*Highest moves*
Vanilla 0.2 epsilon	Q learning	0.181	15
	SARSA	0.203	15
0.4 epsilon	Q learning	0.190	14
	SARSA	0.187	14
Mean action weight	Q learning	0.326	9
	SARSA	0.322	9
Max action weight	Q learning	0.316	11
	SARSA	0.316	11
Maximum 10 moves	Q learning	0.192	8
	SARSA	0.192	8

Overall, the results are inadequate. For example, performing vanilla deep SARSA scored an exponentially moving average reward of approximately 0.2; this means that at best, deep SARSA was capable of winning a game of chess 20% of the time. This poor performance could be indicative of an implementation inaccuracy – a mistake in the implementation of the algorithm – as similar implementations have scored an average reward of 80% or higher. However, the improvements suggested by this paper resulted in an increase in performance. Most notably, only updating the weight that corresponded to the action taken saw average scores of over 30%, with the highest being 32.6% achieved by taking the mean average of the delta on the Q learning approach. These results suggest that only updating the weights that correspond to the action taken yields a better performance than updating all of the weights. Slightly higher results using the mean of the weight delta performed better than taking the maximum delta.

Surprisingly, adding a ceiling to the amount of moves the agent was allowed to take did not improve the performance. This could be because underneath, the base method did not change, and as a result was sill subject by the limitations

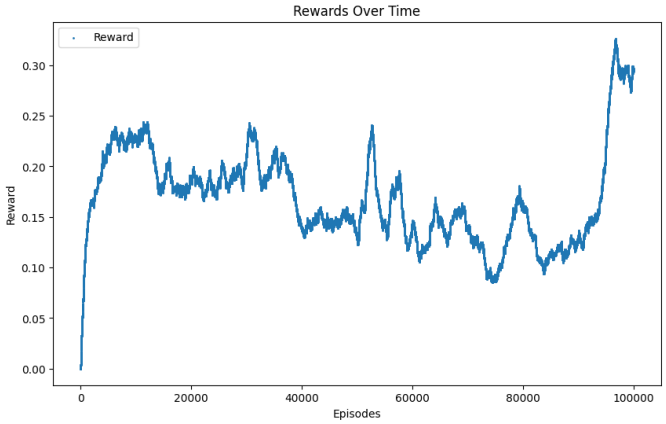

Fig. 5. Exponential moving average reward over time for Q learning mean acton weight update

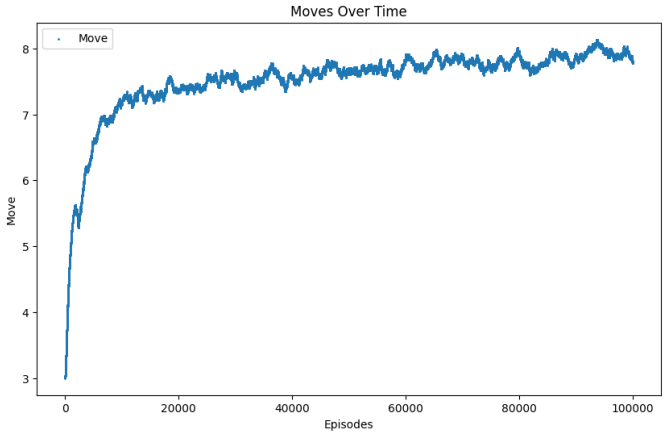

Fig. 6. Exponential moving average moves over time for SARSA 10 move ceiling

of the vanilla implementation. SARSA in general performed significantly better than Q learning on the vanilla implementations, but achieved approximately the same results across all other experiments. When using a value of 0.4 for epsilon, only a slight improvement in the Q learning average reward was observed, whereas the average reward for SARSA dropped from 0.203 to 0.187. An explanation for this could be a mismatch between the γ value used in the Bellman equation and using 0.4 as the epsilon_0 value. Since γ was set to 0.85, this rewards the agent for exploring the environment. Using a higher value of 0.4 would also mean the agent spends more time exploring the environment. These 2 factors, along with SARSA prioritising exploration itself, could mean that not enough time was spent optimising for a solution.

Although similar results were achieved with 0.4 epsilon compared to a ceiling of 10 moves, there is evidence the latter is a smarter agent. This is because similar scores were achieved with significantly fewer moves – 14 and 8 respectively. These results demonstrate that it only takes the latter a maximum of 8 moves to achieve checkmate at the same rate the epsilon 0.4 agent achieved. The best performing variations, updating the action weight using the mean, were able to achieve this in 9 moves.

IV. Conclusion

Overall, despite poor performance, the suggested changes and variations have successfully improved the performance, in particular updating only the weight corresponding to the action taken. Going forward, there are a range of improvements and motivations this paper has suggested for future research. One of which may be to use a complex architecture. This paper implements only a simple feed-forward neural network with one densely-connected hidden layer; other architectures could provide a boost in performance such as by using more hidden layers or a different schematic entirely, such as an RNN (recurrent neural network).

Another suggested improvement is the use of a smaller output layer. This paper implements an output layer of 32 neurons, however a smaller sized layer could force the network to generalise better, resulting in a reduced chance of overfitting and an easier time converging on a global minimum. Other improvements could be to minimise the moves taken. This is different to ending the game after a certain number of moves; incorporating the number of moves made into the loss function and then enticing the network to minimise this could encourage a smarter agent which can achieve checkmate in only a small number of moves.

Appendix

```
# Epsilon greedy
if np.random.rand() < epsilon_f:
    Action = np.random.permutation(a)[0]
else:
    # Select best action
    valid_actions = yhat[0][a] # Get Q values that
    correspond to best action
    largest = np.argmax(valid_actions) # Get largest
    Q value
    Action = a[largest] # Get best
```

Fig. 7. Implementation of the epsilon-greedy policy. Selects at random whether to take the best action or take a random action

```
# Forward propagation
z1=np.matmul(X,w1) + np.tile(b1,[1,1]) # wx + b
a1 = relu(z1)  # ReLU
z2=np.matmul(a1,w2)+np.tile(b2,[1,1]) # ax + b
yhat = relu(z2)  # ReLU
```

Fig. 8. Implementation of forward propagation

```
dloss = (R_rep-yhat[0][Action])
delta_w1 = eta * np.array([X]).T.dot((dloss * drelu(
    z2)).dot(w2.T) * drelu(z1))
delta_b1 = eta * dloss * drelu(z2) * w2 * drelu(z1).T
delta_b1 = np.sum(delta_b1, axis=1, keepdims=True)
delta_w2 = eta * dloss * drelu(z2) * a1.T
delta_b2 = eta * dloss * drelu(z2)
```

Fig. 9. Implementation of backpropagation

The equations for forward propagation

$$z_1 = X \cdot \ W_1 \ + \ b_1 \tag{5}$$

$$a_1 = f\left(z_1\right) \tag{6}$$

$$z_2 = a_1 \cdot \ W_2 \ + \ b_2 \tag{7}$$

$$\hat{y} = f(z_2) \tag{8}$$

$$L = \frac{1}{2}(y - \hat{y})^2 \tag{9}$$

$$where \ f(x) = max(0, x) \tag{10}$$

Part 1 of the equations for backpropagation

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_2} \frac{\partial z_2}{\partial a_1} \frac{\partial a_1}{\partial z_1} \frac{\partial z_1}{\partial W_1} \tag{11}$$

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_2} \frac{\partial z_2}{\partial w_2} \tag{12}$$

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_2} \frac{\partial z_2}{\partial a_1} \frac{\partial a_1}{\partial z_1} \frac{\partial z_1}{\partial b_1} \tag{13}$$

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_2} \frac{\partial z_2}{\partial b_2} \tag{14}$$

Part 2 of the equations for backpropagation

$$\frac{\partial L}{\partial W_1} = (y - \hat{y}) \times f\prime(z_2) \times w_2 \times f\prime(z_1) \times X \tag{15}$$

$$\frac{\partial L}{\partial W_2} = (y - \hat{y}) \times f\prime(z_2) \times a_1 \tag{16}$$

$$\frac{\partial L}{\partial b_1} = (y - \hat{y}) \times f\prime(z_2) \times w_2 \times f\prime(z_1) \times 1 \tag{17}$$

$$\frac{\partial L}{\partial b_2} = (y - \hat{y}) \times f \ \prime(z_2) \times 1 \tag{18}$$

$$where \ f\prime(x) = \begin{cases} 0 & forx < 0 \\ 1 & forx \geq 0 \end{cases} \tag{19}$$