Exploring the Deep Reinforcement Learning through PacMan

Lei Cheng
Department of Electrical & Computer Engineering
University of Arizona
Tucson, AZ, USA
leicheng@arizona.edu

***Abstract*—In the contemporary landscape of Artificial Intelligence, reinforcement learning (RL) stands out for its capacity to facilitate complex decision-making through dynamic interactions with the environment. This paper explores deep reinforcement learning, with a specific focus on Deep Q-learning, by utilizing the classic PacMan game as a practical framework for analysis. I compare Deep Q-learning against traditional reinforcement learning methods such as basic Q-learning and Approximate Q-learning to illustrate its effectiveness and enhanced capability in handling complex tasks that standard RL methods struggle with. Through this comparative analysis, we aim to highlight the inherent advantages of Deep RL, particularly in terms of its ability to learn from raw data and adapt to a variety of strategic scenarios without extensive human intervention. The code is available at https://github.com/monroyaume5/Deep_Q_Learning_for_PacMan.git**

***Index Terms*—reinforcement learning, deep reinforcement learning, Q-learning, deep Q-learning, artificial intelligence**

1. Introduction

In the rapidly evolving field of Artificial Intelligence (AI), machine learning stands as a cornerstone, enabling machines to glean knowledge and make decisions with minimal human intervention. Among the prevalent paradigms of machine learning—supervised learning, unsupervised learning, and reinforcement learning—each serves unique purposes and operates under different frameworks, as shown in Fig. 1. Supervised learning relies on labeled datasets to teach models how to predict outcomes, while unsupervised learning seeks to identify patterns and relationships in data without pre-assigned labels.

However, it is reinforcement learning (RL) that distinctly emphasizes interaction with an environment to achieve complex decision-making tasks. Unlike the static nature of datasets in supervised and unsupervised learning, reinforcement learning engages in a dynamic process where an agent learns to perform actions based on continuous feedback aimed at maximizing cumulative rewards [1]. This attribute makes RL particularly powerful for sequential decision-making tasks across diverse applications such as robotics, healthcare, finance, and autonomous vehicles [2].

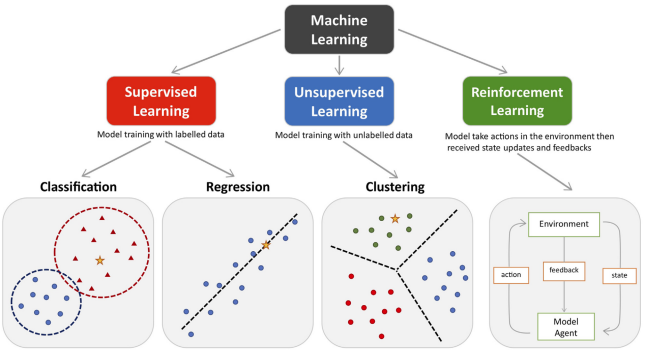

Figure 1: The three main types of machine learning, including classification and regression under supervised learning, clustering under unsupervised learning, and reinforcement learning which enhances the model performance by interacting with the environment. Source: [9]

A quintessential example within reinforcement learning is Q-learning [3], a method that estimates the value of actions in given states to derive optimal strategies. Q-learning aims to approximate the optimal action-value function, but it sometimes learns overly optimistic action values due to its inherent maximization step [4]. This issue can be mitigated by techniques like Double Q-learning [5], which separates the evaluation of action values from their selection to reduce overestimation biases.

Despite the successes of traditional RL methods, they often rely heavily on hand-crafted features coupled with linear value functions or policy representations [6], which require extensive human experimentation to optimize [7], thus limiting their effectiveness and scalability. In contrast, deep reinforcement learning (Deep RL) leverages neural networks to learn from raw data, creating representations at various levels of abstraction (by using more complex, nonlinear function approximations) [8] and lowering the need for prior knowledge. Deep RL has shown remarkable success in complex tasks where traditional methods falter [2]. In this paper, we delve into deep reinforcement learning, specifically focusing on Deep Q-learning, and employ the iconic PacMan game (as shown in Fig. 2) as a testbed. This exploration not only provides an intuitive understanding of

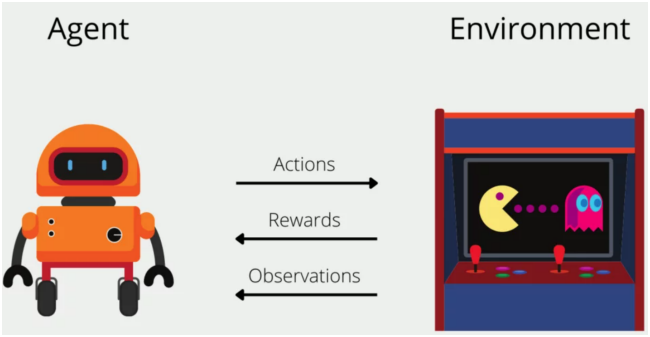

Figure 2: Reinforcement learning with PacMan game. Source: [10]

reinforcement learning but also facilitates a comparative study of Deep RL against conventional RL techniques like basic Q-learning and Approximate Q-learning [8]. Through this comparison, we aim to enrich our comprehension of Deep RL's capabilities and potential applications.

2. Reinforcement Learning

Reinforcement Learning (RL) involves agents interacting within a specific environment where they observe its state, perform actions, and receive rewards. The primary goal of RL is to learn an optimal policy, essentially a strategy that maps observations to actions in a way that maximizes total rewards over time. The entity making decisions and learning within this framework is known as the agent, and it interacts with the environment, which includes everything external to the agent. This interaction is continuous, with the agent choosing actions and the environment responding by changing the state and presenting new scenarios, as well as dispensing rewards—numerical values that the agent aims to accumulate [1].

Such interactions between the agent and the environment, as illustrated in Fig. 3, are structured as a Markov Decision Process (MDP), which is characterized by its state space (S), action space (A), reward function (R), transition operator (T), and the policy (π). At each step, the agent applies the policy to select each possible action and make its decisions. The challenge in solving an MDP lies in finding an optimal policy π^*, a function that for every state $s \in S$ in the state space, identifies an action $a = \pi^*(s)$ in the action space that maximizes expected rewards. An optimal policy is one that ensures the maximum expected total reward or utility when followed by the agent. Solving Markov decision processes using techniques such as value iteration and policy iteration is offline planning, where agents have full knowledge of both the transition function and the reward function, all the information they need to precompute optimal actions in the world encoded by the MDP without ever actually taking any actions. In contrast, reinforcement learning typically involves online planning where the agent has no prior knowledge of transitions and rewards. In this mode, the agent must engage in exploration, trying out

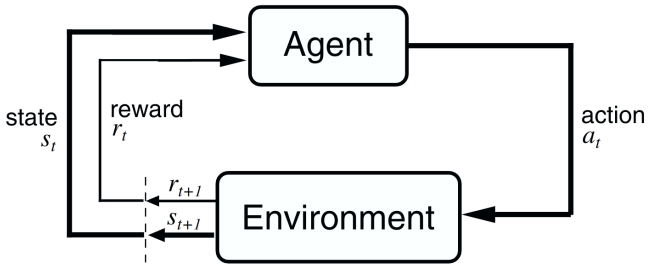

Figure 3: Reinforcement learning with PacMan game. Source: [1]

actions and gathering feedback from the results—namely, the new states and rewards it encounters. This feedback is crucial for the agent to estimate and refine its policy, which it then uses to maximize its rewards through strategic action choices.

3. Q-learning and Approximate Q-learning

Reinforcement learning (RL) algorithms predominantly revolve around estimating value functions, which are functions of states or state-action pairs that predict the benefits for the agent in those states or actions. These benefits, termed as "how good," are quantified in terms of the expected future rewards. The future rewards that an agent anticipates hinge on the actions it opts to take, thus aligning value functions with specific policies. The value of a state s under a policy π, symbolized as $V^\pi(s)$, represents the expected return when the agent starts in s and adheres to π, and can be expressed as:

$$V^\pi(s) = E_\pi \{R_t \mid s_t = s\}$$

where E_π denotes the expected value assuming adherence to policy π. Conversely, the value of executing an action a in state s under policy π, denoted $Q^\pi(s, a)$:

$$Q^\pi(s, a) = E_\pi \{R_t \mid s_t = s, a_t = a\}.$$

The state-value function V^π and the action-value function Q^π, can be deduced from experience. Essentially, solving an RL task means finding a policy that maximizes rewards over time. Optimal policies π^* share a common optimal state-value function V^*, and an optimal action-value function Q^*. The Bellman optimality equations for V^* and Q^* can be expressed as follows:

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^*(s')\right],$$

$$Q^*(s, a) = \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^*(s')\right].$$

Solving these equations yields V^* and Q^*. With V^*, the actions that appear best after a one-step search are deemed optimal. Q^* simplifies the process further; for any state s, the agent can directly choose an action that maximizes $Q^*(s, a)$ without the need for any one-step-ahead search

since the action-value function effectively caches the results of all one-step-ahead searches. Also, the optimal action-value function allows optimal actions to be selected without having to know anything about possible successor states and their values, or about the environment's dynamics. Q-learning, a model-free approach, learns these Q-values directly, circumventing the necessity to know transition functions or reward functions. It employs the following update rule for what's called Q-value iteration:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right].$$

With this update rule, Q-learning is derived by acquiring Q-value samples and incorporating them into an exponential moving average:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \cdot (R(s,a,s') + \gamma \max_{a'} Q(s',a'))$$

By maintaining a robust exploration strategy and appropriately reducing the learning rate α, Q-learning can ascertain optimal Q-values for every state-action pair. This characteristic marks Q-learning as revolutionary.

However, Q-learning needs to store all Q-values for all state-action pairs, and this becomes impractical for applications with vast numbers of states due to memory and computational limitations. To address this, approximate Q-learning employs feature-based representations of states to generalize across similar situations, significantly improving efficiency. Each state and state-action pair in approximate Q-learning is represented by feature vectors, allowing the values of states and Q-values to be expressed as linear combinations of these features:

$$V(s) = \mathbf{w}^T \mathbf{f}(s)$$
$$Q(s,a) = \mathbf{w}^T \mathbf{f}(s,a)$$

where $\mathbf{f}(s)$ and $\mathbf{f}(s,a)$ are the feature vectors for the state and state-action pair respectively, and $\mathbf{w}$ is a weight vector.

The update rule for approximate Q-learning is based on the difference between the expected and the current Q-values:

$$\text{difference} = \left[R(s,a,s') + \gamma \max_{a'} Q(s',a') \right] - Q(s,a)$$

Weights are updated as follows:

$$w_i \leftarrow w_i + \alpha \cdot \text{difference} \cdot f_i(s,a)$$

Rather than storing Q-values for each and every state, with approximate Q-learning we only need to store a single weight vector and can compute Q-values on-demand as needed [5].

4. Deep Q-learning

Traditional Q-learning often relies on linear function approximators to estimate the utility function or action-value function. However, there are compelling reasons to move beyond them. Firstly, a suitable linear approximation for the utility function or action-value function may not exist. Secondly, inventing necessary features can be challenging. Due to these challenges, researchers have long explored more complex, nonlinear function approximators, with deep neural networks now proving particularly effective. These networks can handle raw inputs to learn better representations than handcrafted features. If the final layer of these networks is linear, it allows for an examination of the features used by the network to construct its linear function approximator. Deep Q-learning (DQN) [6] is introduced by leveraging this idea to employ deep neural networks for function approximation. It defines a neural network with weights θ as a Q-network, a function approximator used for estimating the Q-values associated with state-action pairs. The training of a Q-network involves minimizing a sequence of loss functions $L_i(\theta_i)$ that change at each iteration i, described as follows:

$$L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot)} \left[(y_i - Q(s,a;\theta_i))^2 \right]$$

where y_i is the target for iteration i and is calculated by:

$$y_i = \mathbb{E}_{s',a' \sim \rho(\cdot)} \left[r + \gamma \max_{a'} Q(s',a';\theta_i^-) \right]$$

Here, $\rho(s,a)$ represents the behavior distribution, a probability distribution over states s and actions a from which these expectations are sampled. The behavior distribution $\rho(s,a)$ essentially dictates how sequences and actions are chosen during training, reflecting the policy under which the data was collected, often in an exploratory manner to ensure adequate state and action space coverage. This formulation allows for the continuous adaptation of the Q-network by recalibrating the target y_i at each iteration based on the latest policy and network parameters, thus fostering robust convergence of the network weights towards optimal policy estimation.

Deep Q-learning also employed a key technique known as experience replay, where the agent's experiences at each time-step $e_t = (s_t, a_t, r_t, s_{t+1})$ are stored in a dataset $D = \{e_1, \ldots, e_N\}$, pooled over many episodes into a replay memory. During training, the algorithm applies minibatch updates to randomly drawn samples from this memory. After performing experience replay, the agent makes decisions based on an ϵ-greedy policy. Since using arbitrarily long histories as neural network inputs is impractical, the Q-function operates on fixed-length representations of histories produced by a function ϕ.

The max operation in Q-learning often leads to overoptimistic value estimates due to using the same values both for selecting and evaluating an action. To address this, the Double DQN (DDQN) algorithm [5] introduces an upward bias correction known as the double estimator method, which decouples the selection and evaluation of an estimator's value, thus allowing for the removal of the positive bias in estimating action values. In DDQN, the target value update rule is modified to:

$$y_i^{DDQN} = r + \gamma Q(s', \text{argmax}_{a \in A} Q(s',a;\theta_i);\theta_i^-),$$

which effectively reduces overestimation in Q-values and enhances stability and performance. Unlike standard DQN,

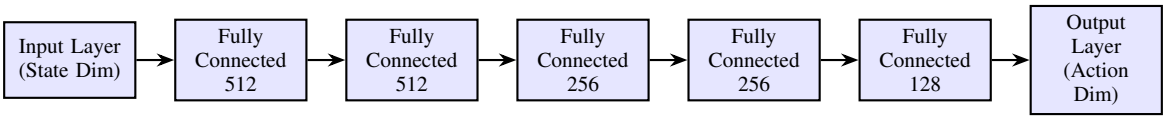

Figure 4: Neural Network Architecture of DeepQNetwork

where a single network estimates the values used to both select and evaluate actions, in DDQN, the target network with weights θ_t^- is solely used for evaluating the current greedy action, ensuring that the policy decisions are based on the values estimated by the current network weights θ.

5. Implementation

In this project, I have developed a deep Q-learning model named `DeepQNetwork`, as shown in Fig. 4, which is a neural network designed to predict the Q values for all possible actions given a state. This deep Q-learning model is trained using the double deep Q-learning method introduced earlier and is encapsulated within an agent class named `PacmanDeepQAgent`, enabling it to play the Pacman game. My implementation closely follows the foundational code provided in the [11]. Specifically, the `DeepQNetwork` was independently developed by me, while the `PacmanDeepQAgent` is largely based on the original code with minimal modifications. Below is a detailed description of my contributions:

The `DeepQNetwork` class, implemented as a subclass of PyTorch's `Module`, forms the foundation of our reinforcement learning framework by approximating the Q-values for all possible actions given a state. It initialized with parameters `state_dim` and `action_dim`, representing the dimensions of the state space and the action space, respectively. The network comprises a sequence of six fully connected (fc) layers with varying sizes, progressively reducing from 512 units in the initial layers to 128 units just before the output layer, which matches the number of actions. Each layer up to the fifth uses ReLU activation to introduce non-linearity, essential for capturing a deep and complex representation of state features (e.g., `self.fc1 = nn.Linear(state_dim, 512)` followed by `relu`). He initialization (Kaiming initialization) is applied (e.g., `nn.init.kaiming_uniform_(m.weight, nonlinearity='relu')`), which is particularly suited for layers followed by ReLU activations, ensuring the variance of activations remains balanced across layers. Initially, I did not use this initialization technique and observed that even after prolonged training, the results were suboptimal. However, upon switching to He initialization, there was a noticeable improvement in performance. The forward pass method `forward` processes batches of states and computes the corresponding Q-values for all actions, efficiently mapping state inputs to action value outputs. For training, the `get_loss` method computes the mean squared error between predicted Q-values and target Q-values, guiding the optimization process (e.g., `mse_loss(Q_predictions, Q_target)`). The gradient update procedure adjusts the network's weights based on the loss, using stochastic gradient descent steps initialized in the constructor (e.g., `self.optimizer.step()`).

Additionally, I have implemented both Q-learning and Approximate Q-learning algorithms to benchmark their performance against the deep Q-learning model. These implementations were adapted from the code I developed in Homework 3.

6. Results and Discussion

In this study, three methods—Q-Learning, Approximate Q-Learning, and Deep Q-Learning—were tested on SmallGrid and MediumGrid Maps. Each method was run three times: Run 1 consisted of 10 test games following 2000 training games, Run 2 involved 10 test games after 4000 training games, and Run 3 included 10 test games subsequent to 6000 training games.

For Deep Q-Learning on the SmallGrid map, the training results from Run 3 are displayed in Fig. 5. It is important to note that the first 1000 games are used for exploration in Deep Q-Learning and do not contribute to updates in the Q network. The learning parameters were set as follows: epsilon=0.05, alpha=0.2, and gamma=0.8. The experimental outcomes are assessed based on the average score of PacMan over 10 test games, as presented in Table 1.

The table comparing the performance of Q-Learning, Approximate Q-Learning, and Deep Q-Learning across SmallGrid and MediumGrid Maps shows distinct patterns in how these methods adapt to varying complexities and sizes of environments. In the SmallGrid scenarios, both Q-Learning and Approximate Q-Learning demonstrate stable and consistent performance with scores clustering around 500. This indicates that these methods effectively converge to a good policy in simpler or smaller environments where the challenges are relatively contained. However, the performance of Deep Q-Learning on the SmallGrid map is highly variable in the initial runs, beginning with notably low scores and improving dramatically by the third run. This pattern suggests that Deep Q-Learning may require a longer period of adaptation and more extensive training to overcome initial instability but has the potential to achieve competitive performance once adequately trained.

On the MediumGrid map, which likely presents a more complex and challenging environment, Q-Learning shows a slight improvement over runs, albeit with scores remaining negative. This increment might be too marginal to be effective for practical applications in complex environments. Approximate Q-Learning does not fare well either, with its performance slightly degrading over time, which might

TABLE 1: Performance Comparison Across Three Runs of Q-Learning, Approximate Q-Learning, and Deep Q-Learning on SmallGrid and MediumGrid Maps

Method	SmallGrid			MediumGrid		
	Run 1	Run 2	Run 3	Run 1	Run 2	Run 3
Q-Learning	498.2	500.2	500.2	-528.3	-523.8	-515.6
Approximate Q-Learning	499.8	499.8	498.2	-514.7	-522.2	-519.7
Deep Q-Learning	-513.2	-241.2	489.4	-114.8	520.9	500.7

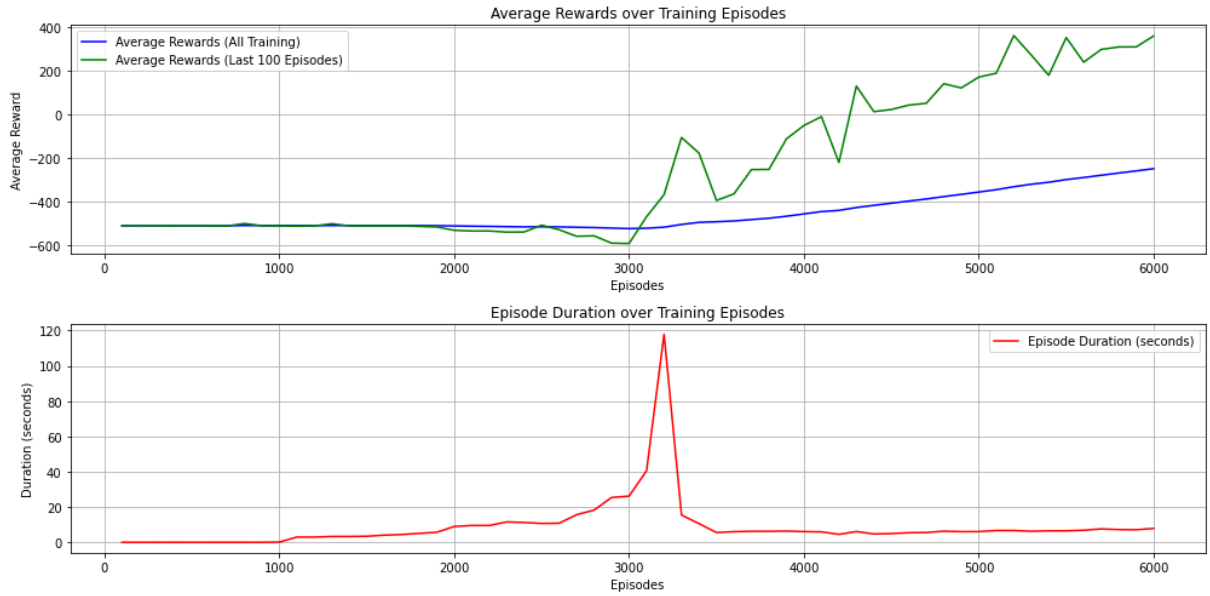

Figure 5: Average Rewards and Duration over Training Episodes

indicate its limitations in more complex scenarios where the feature space can't capture the environment's dynamics adequately. Deep Q-Learning stands out significantly in the MediumGrid environment, starting with poor results but showing remarkable improvement in subsequent runs. By the third run, it surpasses the other methods substantially, suggesting its superior ability to learn and generalize from complex, high-dimensional data. This indicates that Deep Q-Learning, despite its need for extensive training and fine-tuning, is ideally suited for complex environments where traditional methods struggle.

In conclusion, the choice of learning method should be guided by the specific characteristics of the environment and task. For tasks with limited complexity, traditional Q-Learning methods are adequate and provide stable performance with less computational overhead. In contrast, for more dynamic and challenging environments, Deep Q-Learning is more appropriate, albeit at the cost of longer training times (as shown in Fig. 5) and greater computational resources. This method's ability to eventually outperform others in complex settings underscores its utility for real-world applications that involve intricate and unpredictable dynamics.

7. Conclusion

This study has extensively investigated the nuances of deep reinforcement learning through the lens of the PacMan game, demonstrating Deep Q-learning's superior performance over traditional Q-learning and Approximate Q-learning. Our findings corroborate the hypothesis that Deep RL, by leveraging neural networks, can significantly outperform conventional methods in complex environments. The use of Deep Q-learning not only improved decision-making accuracy but also showcased the potential for generalization across different gaming scenarios. This adaptability makes Deep RL especially suitable for real-world applications across sectors like robotics, healthcare, and autonomous navigation, where decision-making in dynamic environments is crucial. Our research also reaffirmed the challenges associated with traditional RL methods, which often depend heavily on the quality of hand-crafted features and are limited by their linear policy representations. In contrast, Deep RL's ability to autonomously learn and

abstract from raw input data reduces the reliance on human-engineered features and opens new avenues for autonomous system design. In conclusion, the advancements in Deep RL not only enhance our understanding of complex decision processes but also pave the way for more sophisticated and autonomous systems in the future. Future research should continue to explore these methods in more diverse settings to fully realize their potential and refine their capabilities for broader applications.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[2] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, J. Pineau *et al.*, "An introduction to deep reinforcement learning," *Foundations and Trends® in Machine Learning*, vol. 11, no. 3-4, pp. 219–354, 2018.

[3] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[4] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.

[5] H. Hasselt, "Double q-learning," *Advances in neural information processing systems*, vol. 23, 2010.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[7] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, "Deep reinforcement learning from human preferences," *Advances in neural information processing systems*, vol. 30, 2017.

[8] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Pearson, 2016.

[9] J. Peng, E. C. Jury, P. Dönnes, and C. Ciurtin, "Machine learning techniques for personalised medicine approaches in immune-mediated chronic inflammatory diseases: applications and challenges," *Frontiers in pharmacology*, vol. 12, p. 720694, 2021.

[10] DataBaseCamp. Reinforcement learning – simply explained! [Online]. Available: https://databasecamp.de/en/ml/reinforcement-learnings

[11] UCBerkeley. Cs 188 spring 2024, project 6: Reinforcement learning. [Online]. Available: https://inst.eecs.berkeley.edu/~cs188/sp24/projects/proj6/