Universiteit Leiden

Master Computer Science

Optimizing Elite Sports Training: A Reinforcement Learning Approach for Personalized and Effective Training Schedules

Name:	Nikolaos-Maximos Bilalis
Student ID:	s3626776
Date:	28/04/2025
Specialisation:	Data Science: Computer Science
1st supervisor:	prof. dr. Arno Knobbe
2nd supervisor:	prof. dr. Aske Plaat

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Crafting personalized and effective training schedules for elite athletes presents a significant challenge in sports coaching. Traditional methods often rely on coaches' experience and intuition, which may not fully capture the nuances required for optimal training. This research addresses this challenge by applying Reinforcement Learning (RL) to model coaching decision-making using historical training data from top Dutch athletes in Professional Speed Skating. The goal is to develop a data-driven approach that not only replicates but enhances traditional coaching practices. By incorporating feedback from races and physical tests, the model aims to iteratively refine training schedules to improve their effectiveness. This approach seeks to advance the precision of training regimens, potentially leading to better athlete performance, more efficient resource utilization, and a more scientifically grounded coaching methodology. We find that, eventually, the best approach to this set up is the traditional ML models, scoring an accuracy of 85% in predicting the best future training schedules.

April 21, 2025

Contents

1 Introduction

In the domain of elite sports coaching, the formulation of optimal training schedules stands as a hard challenge. Traditional coaching methods often rely heavily on the coach's experience and intuition, which, while valuable, may not always capture the nuances required for individualized training optimization. This paper outlines a research initiative aimed at addressing this challenge through the application of Reinforcement Learning (RL) in the context of sports training. Leveraging historical data from top athletes in Dutch elite sports, specifically in Professional Speed Skating, we attempt to construct a model capable of generating personalized and effective training schedules for individual athletes. The focus extends beyond mere replication of human decision-making, aspiring to iteratively improve training schedules by incorporating limited feedback from races and physical tests. The ultimate goal of this research is to develop a robust, data-driven approach that assists coaches in making informed decisions, thereby enhancing the effectiveness and precision of training schedules. This could potentially lead to significant improvements in athlete performance, better resource utilization, and more scientifically grounded coaching practices.

1.1 Topic

1.1.1 General Overview of the Topic

Elite sports training has long been a critical area of study within sports science, focusing on maximizing athletic performance through optimized training regimens. The precision and customization of training schedules are crucial for athletes aiming to achieve peak performance, reduce the risk of injury, and maintain long-term physical and mental well-being, as well as sleep quality. Brown et al., 2020, Forndran et al., 2012

1.1.2 Specific Context of the Research

In recent years, the complexity of training individualization has increased, driven by advancements in sports science and the availability of vast amounts of performance data Bourdon et al., 2017, Cardinale and Varley, 2017. Elite coaches must consider numerous variables, including the athlete's physiological and psychological states, competition schedules, and recovery needs. However, crafting precise and personalized training schedules remains a significant challenge, often relying heavily on the coach's experience and intuition. Nash and Collins, 2006

A training plan, or exercise prescription, translates sport and exercise science into practice. Similar to medical practice, it involves creating a training plan or prescribing an exercise program based on the best current scientific evidence. However, these plans often include a combination of various interventions, such as exercises and nutritional guidelines, which evolve over time due to factors like periodization or tapering.

This complexity makes it nearly impossible to rely entirely on scientific evidence for long-term training plans. Wackerhage and Schoenfeld, 2021 Another challenge is the significant variability in how individuals adapt to different training programs, such as endurance or resistance training. Until specific biomarkers for trainability are identified, continuous testing and monitoring of athletes, clients, or patients are necessary to determine the effectiveness of a training plan and make adjustments as needed. Unlike the clear-cut "drug or no drug" decisions in medicine, creating a training plan involves many decisions that cannot all be evidence-based. Hence, an evidence-informed approach should be employed, where some decisions are grounded in the best available evidence. Because training adaptations vary widely, practitioners must continuously assess and adjust training plans if they do not produce the desired outcomes.

At this point, the Reinforcement Learning (RL) solution is proposed. Using RL for producing training schedules, while it might not directly include evidence-based elements, follows a systematic and adaptive approach. RL algorithms can learn from continuous feedback and adapt training plans to optimize performance outcomes. This method involves setting specific goals, applying various training interventions, and constantly adjusting based on the athlete's response. Over time, RL can identify patterns and preferences unique to each individual, leading to highly personalized and effective training schedules. By integrating RL, we can complement the evidence-informed approach with a dynamic system that evolves with the athlete, potentially uncovering new insights into optimal training practices and reducing the trial-and-error aspect of current methodologies.

1.1.3 Existing Solutions and Limitations

Traditional approaches to training schedule optimization involve heuristic methods, generic training guidelines, and basic data analysis techniques Raab, 2012 , Raiola and Tafuri, 2015. While these methods have yielded some success, they fall short in providing the fine personalization required for elite athletes. Existing methods lack the ability to dynamically adapt to the changing needs and conditions of the athlete, leading to suboptimal training outcomes and potential overtraining or undertraining scenarios.

1.1.4 Research Focus

This research aims to address these limitations by leveraging historical training data from successful Dutch athletes and employing Reinforcement Learning (RL) to model coaching decision-making. RL, with its capability to learn optimal policies through interaction with an environment, offers a promising approach to creating adaptive and personalized training schedules. In our setting, the role of the environment will be played by the historical dataset, but we will dive deeper into this later in the paper. By developing a data-driven RL-based system, this research seeks to enhance the precision and effectiveness of training

schedules, providing a valuable tool for coaches and contributing to the field of sports science.

1.2 Problem Statement

This research focuses on the challenge faced by elite sports coaches in crafting precise and personalized training schedules for individual athletes. The project utilizes historical training data from successful Dutch athletes, employing Reinforcement Learning to model coaching decision-making.

1.3 Research Questions

At the end of this study, we aim to be able to answer the following research questions:

- How can Reinforcement Learning be effectively applied to model the decision-making process of elite sports coaches in generating personalized training schedules for athletes in Professional Speed Skating?
- To what extent can a Reinforcement Learning model effectively incorporate and adapt to limited feedback from races and physical tests to iteratively refine training schedules, thereby enhancing their effectiveness and aligning with individual athlete performance outcomes?
- How effectively can Reinforcement Learning methods—originally designed for online exploration—be adapted for offline settings with historical data, and how do these adaptations compare in performance and stability to dedicated offline RL algorithms under conditions of sparse rewards and limited exploration opportunities?
- Can Reinforcement Learning (RL) outperform standard Machine Learning techniques when the dataset is limited? Do RL agents generalize well when historical data lacks comprehensive exploration coverage, and if not, do they effectively learn to mimic real-world scenarios where the learned policy would be applied?

2 Related Work

In this section, we present some intriguing papers on the intersection of Sports and Machine Learning that closely align with our research. Recently, Machine Learning (ML) has increasingly been incorporated into Sports to maximize performance. Notably, there is a lack of research aimed at building personalized training schedules with end-to-end Reinforcement Learning (RL) directly from past training data.

2.1 ML intersection with sports

The recent advancements in Computer Science and hardware allow for the analysis of vast datasets, including physiological metrics, motion capture data, and psychological profiles, to provide insights that were previously unattainable. For instance, predictive models can forecast an athlete's future performance or injury risk based on current data trends, allowing for proactive adjustments to training regimens. Moreover, computer vision and wearable technology enable real-time feedback during training sessions, facilitating immediate corrections and adjustments.

Injury prediction

In the work of Van Eetvelde et al., 2021 a systematic review of machine learning methods was conducted, including RL, for sport injury prediction and prevention. Their study highlighted the potential of ML to improve injury prediction and enable effective prevention strategies, thus enhancing athlete safety and performance. They concluded to an almost 85% accuracy on predicting an injury highlighting the importance of AI in sports

Computer Vision

Computer Vision is also a rapidly advancing field with direct application in sports. As sports are being broadcasted and always filmed there is a vast amount of videos and photos that can be used to train CV models. For example, the paper titled "Applications of Computer Vision in Sports" Thomas et al., 2021 explores the diverse and impactful ways in which computer vision (CV) technology is transforming sports. It highlights how CV is used for real-time tracking of players and balls, enhancing performance analysis, and preventing injuries through biomechanical monitoring. The paper also discusses the role of CV in improving refereeing accuracy, enhancing fan engagement through augmented reality, and aiding in talent identification and strategy development. Additionally, CV contributes to broadcasting quality and event security, showcasing its broad applicability in sports.

2.1.1 Personilized training schedule with ML

Crafting personalized training schedules for elite athletes has been a longstanding challenge in sports science. Traditional coaching methods, while valuable, often rely on intuition and experience, leading to subjective decisions that may overlook the intricacies of an athlete's performance capabilities. The integration of machine learning (ML) into sports has revolutionized this process by enabling data-driven decision-making. ML techniques such as predictive analytics, computer vision, and biometric analysis have been employed to create highly tailored training programs that maximize performance and minimize injury risks.

A good example of this is the paper "Athletic Skill Assessment and Personalized Training Programming for Athletes Based on Machine Learning" Qin et al., 2024 which dives into the use of machine learning to enhance athletic performance through personalized training programs. It demonstrates that ML models improve the accuracy of performance predictions, such as sprint times, and significantly reduce injury risks. The study highlights the role of ML in analyzing diverse datasets, including biometric and psychological data, to tailor training to individual athletes, leading to improved performance outcomes and sustained long-term benefits. The research also addresses ethical considerations, emphasizing privacy and data security. Overall, the paper underscores the transformative impact of ML on sports science, enabling more effective and personalized coaching strategies.

Their study emphasizes the need for evidence-based practices in sports coaching, advocating for the integration of ML-driven insights to optimize training and performance at all levels of athletic development. However, as they state in the paper, they use the insights of the predictive models to then formulate a training schedule manually while in our research this it is done end-to-end while also they are being restricted to traditional ML approaches instead of RL.

2.1.2 RL in Sports

Within this broader ML framework, reinforcement learning (RL) has shown particular promise. RL algorithms optimize training regimens by continuously adapting to the athlete's progress and feedback, effectively learning what works best for each individual over time. This dynamic and responsive approach offers a significant advantage over static training plans, making RL a powerful tool in the pursuit of peak athletic performance.

RL in competitive cycling

One illustrative paper in this domain is "Deep reinforcement learning for improving competitive cycling performance" by Demosthenous et al Demosthenous et al., 2022, a sport very similar to speed skating in terms of training schedules. This paper investigates the application of deep reinforcement learning to competitive cycling, where predictive models developed from sensory data collected during bike rides forecast cycling speed and heart rate. These models form the basis of a recommendation system that provides real-time feedback on optimal cycling postures to improve speed while minimizing heart rate impacts. Evaluated in both simulated environments and real-world settings, this system demonstrates potential to significantly enhance performance, albeit adjustments are needed to improve the practicality of the recommendations for real-world application. This research highlights the innovative application of RL in developing dynamic and personalized training strategies in sports, setting a foundational example for further exploration in this area.

RL for personalized recommendations

Apart from the forecast, RL can also be used for personalization, very similar to what it is tried in this research. In the paper "Deep Reinforcement Learning Based Personalized Health Recommendations" Mulani et al., 2020 which is building on the principles of personalized recommendations through machine learning, the application of deep reinforcement learning (DRL) in healthcare, offers an effective parallel to sports science. In this healthcare framework, an advanced three-layer DRL system is employed to not only assimilate extensive health data but also to predict disease probabilities and generate precise health recommendations via an actor-critic model. This model adeptly navigates through the complexities of dynamic decision-making processes, optimizing health outcomes over the long term. Such a methodology closely mirrors the objectives in sports training optimization where historical data is pivotal in shaping future training schedules. Here, the actor-critic model's utility in healthcare—to continuously learn and adapt its recommendations based on real-time data—echoes the adaptive training regimens designed in sports science to enhance athletic performance. This convergence highlights the transformative potential of DRL across diverse fields, underpinning the development of tailored, data-driven strategies that significantly advance both individual and collective outcomes in health and sports.

2.1.3 Our contribution

As we show in this chapter, ML and specifically RL have already been applied in the realm of sports. However, it should be mentioned that this is still a very young area of research, and most papers are pioneering. While it is demonstrated that RL is used for personal recommendations and real-time feedback in training sessions, we now aim to expand its application to crafting personalized and effective training schedules for elite athletes based on past training data. Our research proposes a novel approach that leverages the historical performance data of athletes to dynamically adjust and optimize their training regimens. This methodology not only tries to adapt to the unique needs and progress of each athlete but also anticipates future development potentials, thereby enhancing overall athletic performance. The subsequent chapters will detail the methods and experimental setup that underpin our approach, discussing the specific RL algorithms employed, the data collection and processing strategies, and the validation techniques used to test the efficacy of the proposed training schedules. By bridging the gap between theoretical models and practical applications, our work contributes to the evolving landscape of sports science.

3 Background

This chapter introduces the main concepts behind the tools and methodologies used in this work, as well as a basic description of the model's architecture. If you are already familiar with the fundamentals of Reinforcement Learning (RL) and Machine Learning (ML), you may choose to skip this chapter.

3.1 Machine Learning: Introduction

3.1.1 Definition and Scope of Machine Learning

Machine Learning (ML) is a subset of artificial intelligence (AI) that involves the development of algorithms and models that allow computers to learn from and make decisions based on data. It can be categorized into three main types: supervised learning, unsupervised learning, and reinforcement learning (which will be later discussed in detail). ML systems aim to generalize patterns from historical data, making predictions or decisions without being explicitly programmed for every possible scenario.

In this section, we will explore the foundational principles of machine learning and the specific role deep neural networks (DNNs) play within it, along with the critical building blocks that form these systems. As we see in Fig. 1, we will have a quick look at the main stages since the inception of the artificial neuron back in 1943.

3.1.2 Historical Context and Evolution of ML

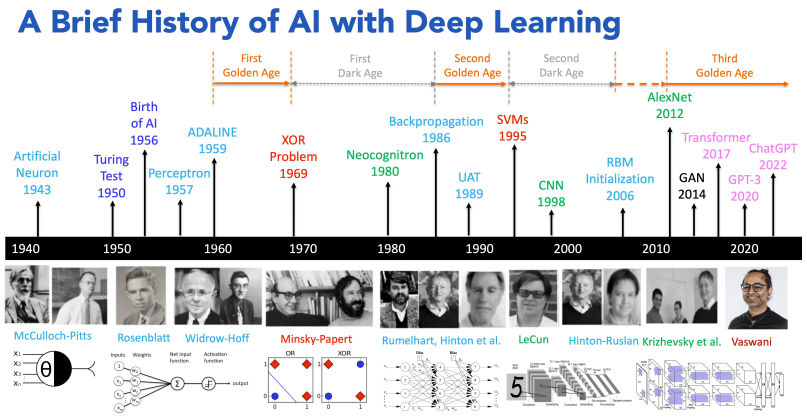


Figure 1: Representation of the ML timeline. Image source: medium.com

Machine Learning (ML) has evolved significantly since its inception, driven by advancements in theory, data availability, and computational power. The

history of ML can be divided into key phases, each contributing to the development of the field as we know it today.

Early Beginnings (1950s - 1970s) ML's foundations were laid in the mid-20th century, with Alan Turing proposing the idea of machines learning from data Turing, 1950. The introduction of the perceptron in 1958 by Frank Rosenblatt Rosenblatt, 1958 was one of the earliest neural network models, although its limitations, as pointed out by Minsky and Papert Minsky and Papert, 1969, slowed the progress of neural networks, leading to the first "AI winter."

The Resurgence of ML (1980s - 1990s) The field saw renewed interest in the 1980s with the development of backpropagation Rumelhart et al., 1986, enabling multi-layer neural networks to learn complex patterns. Other notable advancements included decision trees (ID3) Quinlan, 1987 and the introduction of Support Vector Machines (SVMs) Cortes, 1995, which became fundamental tools in supervised learning.

Data-Driven ML (1990s - 2000s) The growth of the internet and access to large datasets in the 1990s accelerated ML's application across various domains. Ensemble methods like Random Forests Breiman, 2001 and boosting algorithms like AdaBoost Freund, Schapire, et al., 1996 became popular, improving the accuracy and robustness of ML models. Unsupervised learning methods, such as clustering and dimensionality reduction, also gained traction during this period.

The Deep Learning Revolution (2010s - Present) The 2010s marked the era of deep learning, where neural networks with many layers, known as deep neural networks, became dominant in areas like image and speech recognition. The success of AlexNet Krizhevsky et al., 2012 in the ImageNet competition demonstrated the power of deep learning. Architectures like Convolutional Neural Networks (CNNs) and Transformers Vaswani, 2017 further advanced the field, leading to breakthroughs in computer vision and natural language processing.

Current Trends Today, ML is at the core of many innovations, with ongoing research focusing on improving model interpretability, robustness, and efficiency. Reinforcement learning, unsupervised learning, and self-supervised learning are areas of active exploration, as ML continues to expand into new domains, including healthcare, finance, and autonomous systems.

3.1.3 Core Concepts of Machine Learning

At its core, machine learning involves several key concepts:

- **Data**: ML models learn from data. This data is typically represented as features (input variables) that the model uses to make decisions. Data can be structured (tabular, relational) or unstructured (images, text).

- **Model**: A model is a mathematical representation of the relationship between input data and the desired output. For example, in supervised learning, models map input features to a target label.
- **Training**: The process of feeding data into the model to learn patterns is called training. During training, the model adjusts its internal parameters (weights) to minimize prediction error.
- **Loss Function**: The loss function quantifies the error between the model's prediction and the actual output. It serves as the optimization objective that the model tries to minimize. Common loss functions include Mean Squared Error (MSE) for regression and Cross-Entropy Loss for classification tasks.
- **Optimization**: The process of minimizing the loss function is carried out using optimization algorithms such as gradient descent, which iteratively adjusts model parameters (weights and biases) to minimize the loss function.
- **Generalization**: A crucial aspect of ML is ensuring that the model not only performs well on training data but can also generalize to unseen data. Overfitting occurs when a model becomes too complex and performs well on the training data but poorly on new data.

3.1.4 Supervised, Unsupervised, and Reinforcement Learning

- **Supervised Learning**: In supervised learning, the model is trained on a labeled dataset, where each input has a corresponding target output (label). The goal is to learn a mapping from input to output that can be applied to new, unseen data. Examples include classification (e.g., predicting whether an email is spam) and regression (e.g., predicting house prices).
- **Unsupervised Learning**: In unsupervised learning, the model works with unlabeled data. The goal is to find hidden patterns or groupings within the data. Common applications include clustering (e.g., customer segmentation) and dimensionality reduction (e.g., PCA).
- **Reinforcement Learning (RL)**: RL involves learning by interacting with an environment and receiving feedback in the form of rewards or penalties. The goal is to learn a policy that maximizes cumulative rewards over time.

3.1.5 Deep Learning and Neural Networks

Deep Learning is a subfield of machine learning focused on models with many layers, particularly neural networks. These models are especially powerful for handling high-dimensional data, such as images, audio, and text. The term

"deep" refers to the multiple layers in the neural network that enable it to learn complex representations.

The Structure of Neural Networks Neural networks are composed of layers of interconnected units called neurons. Each neuron receives inputs from other neurons, processes these inputs, and passes an output to the next layer. The basic structure consists of:

- **Input Layer**: The first layer that receives the raw data. Each node in this layer corresponds to one feature in the input data.
- **Hidden Layers**: Layers between the input and output, where the model learns to capture intermediate representations. The more hidden layers a network has, the more abstract patterns it can learn.
- **Output Layer**: The final layer that produces the prediction or classification. For classification tasks, this could be a probability distribution over the classes, while for regression, it could be a continuous value.
- **Weights and Biases**: Each connection between neurons has an associated weight, and each neuron has a bias term. These parameters are learned during training and are adjusted to minimize the loss function.

Activation Functions Each neuron applies an activation function to determine its output based on the weighted sum of its inputs. Common activation functions include:

- **Sigmoid**: Maps input values to a range between 0 and 1, useful for binary classification tasks.
- **ReLU (Rectified Linear Unit)**: The most widely used activation function in deep networks, defined as $f(x) = \max(0, x)$. It helps in mitigating the vanishing gradient problem by allowing gradients to flow through the network during training.
- **Softmax**: Typically used in the output layer of classification networks, it converts raw scores into a probability distribution over predicted classes.

3.1.6 Deep Neural Networks (DNNs)

DNNs are neural networks with multiple hidden layers. These layers allow the network to learn hierarchical representations of data, where each layer captures increasingly complex features. For example, in an image classification task, early layers may learn to detect edges, while later layers may recognize more abstract concepts like shapes or objects.

The power of DNNs lies in their ability to automatically learn feature representations from raw data, reducing the need for manual feature engineering. This

makes them particularly effective for tasks such as image recognition (Convolutional Neural Networks), natural language processing (Recurrent Neural Networks, Transformers), and more.

3.1.7 Training Deep Neural Networks

Training DNNs involves adjusting the weights and biases in each layer using optimization techniques. The typical steps are:

- **Forward Pass**: Data is passed through the network layer by layer, producing an output prediction.
- **Loss Calculation**: The loss function computes the difference between the predicted output and the actual target.
- **Backpropagation**: Gradients of the loss function with respect to each weight are calculated using the chain rule. These gradients indicate how to adjust the weights to reduce the loss.
- **Gradient Descent**: The optimization algorithm updates the weights by moving them in the direction that minimizes the loss. Variants like Stochastic Gradient Descent (SGD), Adam, and RMSProp are commonly used.

Regularization Techniques To prevent overfitting in DNNs, various regularization techniques are employed:

- **Dropout**: During training, randomly drops units (along with their connections) from the network, forcing the model to learn robust features.
- **L2 Regularization**: Adds a penalty proportional to the square of the magnitude of weights, discouraging the model from learning overly complex solutions.
- **Early Stopping**: Stops training when the model's performance on a validation set starts to degrade, preventing overfitting.

The Challenge of Vanishing and Exploding Gradients Deep networks can suffer from vanishing or exploding gradients, where gradients become too small or too large as they are propagated back through the network. This can severely hinder the learning process. Solutions to this include:

- **Batch Normalization**: Normalizes the inputs to each layer, stabilizing the learning process.
- **Residual Networks (ResNets)**: Introduce skip connections that allow gradients to flow directly through layers, mitigating the vanishing gradient problem.

While deep learning has been highly successful in handling high-dimensional data and learning complex representations, its core application has traditionally been in supervised and unsupervised learning settings, where large amounts of labeled or unlabeled data are used to train models. However, many real-world problems require decision-making in dynamic environments where the goal is to learn from interaction, rather than from static datasets. This is where *Reinforcement Learning* (RL) comes into play. Unlike deep learning models that primarily rely on labeled data, RL enables agents to learn optimal behaviors through trial and error, receiving feedback in the form of rewards from their environment. The combination of deep learning and RL, known as *Deep Reinforcement Learning*, has opened new frontiers in areas such as robotics, gaming, and autonomous systems. In the following section, we will delve into the fundamentals of RL and explore how it builds upon the concepts of machine learning to solve sequential decision-making problems.

3.2 Reinforcement Learning: Introduction

3.2.1 Definition and Scope of Reinforcement Learning

Reinforcement Learning (RL) is a branch of machine learning focused on how agents should take actions in an environment to maximize cumulative rewards. Unlike supervised learning, where models learn from a fixed dataset of labeled examples, RL involves learning through interaction with a dynamic environment. The agent makes decisions at each step, observes the consequences, and receives feedback in the form of rewards. Over time, the agent refines its policy (a mapping from states to actions) aiming to optimize the long-term reward. The scope of RL encompasses a wide range of applications, from simple, discrete environments to complex, high-dimensional continuous systems.

3.2.2 Historical Context and Evolution of RL

The concept of reinforcement learning has its roots in behavioral psychology, where it was inspired by the idea of learning through rewards and punishments. Early formalizations of RL appeared in the mid-20th century, with foundational work on Markov decision processes (MDPs) by Richard Bellman and others Bellman, 1957. The field gained momentum in the 1980s with the development of key algorithms like Q-learning Watkins, 1989 and SARSA Rummery and Niranjan, 1994, which provided practical methods for learning value functions and policies from experience.

3.2.3 Importance of RL in Modern AI Applications

Reinforcement Learning plays a critical role in modern AI, enabling the development of systems that can make autonomous decisions in complex and uncertain environments. RL is widely used in various domains, including robotics Kormushev et al., 2013, where agents learn to perform tasks like navigation and

manipulation; finance Deng et al., 2016, where RL algorithms optimize trading strategies and portfolio management; and healthcare Yu et al., 2021, where personalized treatment plans are devised through sequential decision-making. In addition, RL has been instrumental in achieving superhuman performance in games, demonstrating its potential to solve intricate problems that require long-term planning and strategic thinking. As AI continues to advance, RL is expected to be a cornerstone in the development of more adaptive, intelligent systems.

3.3 Reinforcement Learning: Fundamentals

3.3.1 Core Components of RL

As stated before, RL is a subset of ML in which an agent learns to make decisions by performing actions in an environment to maximize cumulative rewards. Unlike supervised learning, which relies on labeled data, RL is driven by the exploration of actions and the subsequent rewards or penalties. The core components of RL include the agent, environment, states, actions, and rewards.

The agent is the learner or decision-maker, which takes actions to achieve a goal. The environment is the external system with which the agent interacts, providing states that represent the current situation or context in which the agent must act. The state is a specific configuration of the environment at a given time, containing all the necessary information the agent needs to make a decision. The action is the choice the agent makes based on the current state, which in turn influences the environment. Finally, the reward is a scalar feedback signal provided by the environment in response to the agent's action, indicating how favorable that action was in achieving the desired outcome. The fundamental objective of the agent is to learn a policy that maximizes cumulative rewards over time, balancing short-term gains with long-term success.

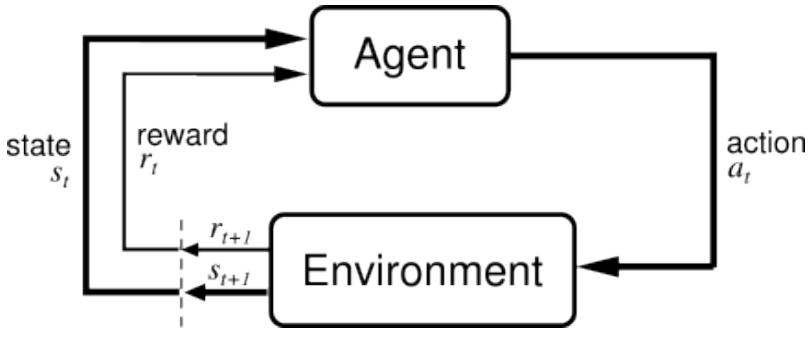


Figure 2: Representation of the interaction of the agent with the environment. Image taken from Sutton and Barto, 2018

As we can see in Fig 2, at each timestep t, the environment presents a state s_t to the agent, which encapsulates the current scenario or context. Based on this state, the agent decides on an action a_t following a specific policy aimed at achieving its objectives. Once the action is executed, the environment responds by providing a reward r_t that evaluates the immediate effectiveness of the action and transitions to a new state s_{t+1}. This reward and the following state serve as feedback for the agent, informing it about the consequences of its actions and guiding its learning process. As discussed, the agent's objective is to learn a policy that maximizes the cumulative rewards over time, training it to make decisions that are increasingly beneficial toward achieving its goal. This cyclical interaction facilitates a dynamic learning environment where the agent continuously adapts and optimizes its behavior based on experience.

The main components can be summarized here:

- **Agent:** The learner or decision-maker.
- **Environment:** The external system with which the agent interacts.
- **State:** A representation of the current situation of the environment.
- **Action:** The set of all possible moves the agent can make.
- **Reward:** The feedback from the environment following an action.

3.3.2 Core Concepts of RL

Several key concepts are central to understanding how reinforcement learning operates. The policy is the agent's strategy or mapping from states to actions; it can be deterministic (a specific action for each state) or stochastic (a probability distribution over actions). The value function estimates the expected cumulative reward associated with each state, helping the agent evaluate the desirability of different states. Closely related is the Q-value (action-value), which estimates the expected cumulative reward for taking a specific action in a given state and then following the policy thereafter. Typically, there are two types of Value functions, the State Value Function which is the expected cumulative reward starting from a state s and following the policy π thereafter and the Action Value Function which is the expected cumulative reward starting from state s, taking action a, and then following policy π thereafter

One of the critical challenges in RL is the exploration-exploitation trade-off. Exploration involves trying out new actions to discover their effects, which is crucial for learning in uncertain environments. On the other hand, exploitation involves choosing actions that are known to yield high rewards, optimizing the agent's performance based on its current knowledge. Balancing these two aspects is key to the success of an RL agent. Finally, the reward function. The reward function is a crucial element that defines the mapping from state-action

pairs to rewards, guiding the agent toward desirable behaviors. The fundamental objective of the agent is to learn a policy that maximizes cumulative rewards over time, balancing short-term gains with long-term success.

The main concepts can be summarized here:

- **Policy:** agent's strategy, essentially the mapping from states to actions.
  - Deterministic: a specific action for each state.
  - Stochastic: a probability distribution over the available actions
- **Value function:** The expected cumulative reward associated with each state. Answering the question: if I follow the policy (selection strategy) from this state onwards, what is my total expected gain?
  - State Value Function
  - Action Value Function
- **Reward function** Mapping that assigns a numerical reward to each state or state-action pair. It is a direct and immediate feedback mechanism that tells the agent how good or bad a particular action was in a specific state
- **Exploration-Exploitation:** A trade-off between exploring new solutions with potentially decreasing your reward short term or being loyal to your initial strategy but with the expected return.

3.3.3 Markov Decision Process (MDP)

A Markov Decision Process (MDP) provides a formalism for modeling decision-making situations where outcomes are partly random and partly under the control of the agent. At the heart of RL lies an MDP and it is defined by the tuple (S, A, P, R, γ), where:

- **S**: A finite set of states, representing all possible situations the agent could encounter.
- **A**: A finite set of actions available to the agent.
- **P**: The state transition probability function, $P(s'|s, a)$, which defines the probability of transitioning to state s' given the current state s and action a.
- **R**: The reward function, $R(s, a)$, which gives the expected immediate reward received after taking action a in state s.
- γ: The discount factor, $\gamma \in [0, 1)$, which determines the importance of future rewards. A γ close to 0 prioritizes immediate rewards, while a γ close to 1 emphasizes long-term rewards.

The objective in an MDP is to find a policy $\pi : S \to A$ that maximizes the expected cumulative reward, often referred to as the return, from each state s:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{1}$$

where G_t is the return at time step t, and r_{t+k+1} is the reward received k steps into the future. The mathematical rigor provided by MDPs allows for the development of algorithms that can systematically optimize the agent's behavior over time.

3.3.4 Offline and Online Reinforcement Learning

As discussed, RL needs an environment to interact with. This interaction can be done mainly in two different ways namely offline and online. Online RL, or simply RL, involves real-time interactions where the agent continuously updates its policy based on feedback. Common algorithms include Q-Learning, SARSA, and Policy Gradient Methods, with applications in robotics, gaming, and autonomous vehicles. Its strengths lie in adaptability and exploration, but it is sample-inefficient and may face stability issues. Conversely, offline RL, or batch RL Lange et al., 2012, learns from a fixed dataset without further interactions. Algorithms like Batch Constrained Q-Learning (BCQ) Ge et al., 2019 and Conservative Q-Learning (CQL) Kumar et al., 2020a are used, and applications include healthcare, finance, and industrial automation. Offline RL is data-efficient and safe but struggles with generalization and depends heavily on data quality. Comparing the two, online RL excels in dynamic environments needing continuous adaptation, while offline RL is suitable for high-risk or impractical real-time interaction scenarios due to its reliance on pre-collected data. Both approaches have unique strengths and limitations, making them suitable for different applications based on their specific requirements and constraints. As it will be discussed on the next chapter, for the demands of this study, Offline RL will be used.

3.3.5 Model-Free and Model-Based Reinforcement Learning

In RL, the distinction between model-free and model-based approaches can be found in how the agent interacts with and learns from the environment.

Model-free RL involves methods where the agent learns to make decisions only from the experiences it gathers through trial and error, without any explicit understanding of the environment's dynamics and mechanics. These methods, such as Q-learning and policy gradient techniques, directly optimize the policy or value function based on observed state-action-reward sequences. This simplicity makes model-free approaches flexible and easy to implement, though they often require a substantial amount of interaction with the environment to converge to an effective policy.

Another approach is model-based methods which involve the agent first constructing its own internal model of the environment's transitions based on feedback it receives. This internal model allows the agent to explore how different actions might affect states and rewards without actually altering the environment. By using a planning algorithm, the agent can simulate various scenarios and update its policy accordingly, which can lead to higher-quality decisions with fewer interactions needed. The process of refining the policy using the internal model is often referred to as planning or imagination. Plaat, 2022

Model-based methods update the policy in an indirect manner: the agent first learns a model of the environment's transition dynamics, which it then uses to refine its policy. This indirect approach has special implications. On the positive side, once the agent has a reliable internal model of state transitions, it can optimize its policy without further interaction with the environment, potentially reducing the sample complexity. However, the downside is that the learned transition model might be inaccurate, leading to a suboptimal policy. Regardless of how many samples are generated from the model, if the agent's internal transition model does not accurately represent the real environment, the policy derived from it may fail when applied. Therefore, handling uncertainty and model bias is crucial in model-based reinforcement learning. The concept of first learning an internal model of the environment's transitions has been explored for many years, with various methods developed to implement these transition models.

3.3.6 On-Policy and Off-Policy algorithms

Another distinction between the RL algorithms is the distinction between on and off-policy. This distinction is based on the way the algorithms handle the data they acquire through interaction, to learn and improve their policy. In the case of the on-policy algorithms, like SARSA the agent can learn strictly from the experience gathered while following the optimal policy. This means that the learning process directly evaluates and improves the same policy. The exploration-exploitation balance is always ensured but always in the context of the current best policy as on-policy algorithms cannot separate exploration from learning and therefore must confront the exploration problem directly Singh et al., 2000. This results in safer and more stable learning as the reward is gradually getting better however it might be more prone to suboptimal policies.

In contrast, off-policy methods like Q-learning and DQN allow the policy to learn from actions that are outside the current policy. If the learning takes place by backing up the values of another action, not the one selected by the behavior policy, then this is known as off-policy learning Plaat, 2022. This means that off-policy learning can utilize data collected from any behavior policy, not just the one currently being optimized. This is beneficial because the agent can learn from a broader range of experiences. Of course, on the other hand, this might lead to more unstable learning. Moreover, off-policy learning

can utilize training data produced by disparate controllers, including human manual control, or data that was collected in the past. Another advantage of off-policy learning is its ability to learn multiple target policies, such as optimal policies for various subgoals, using data from a single behavior policy Maei et al., 2010.

To provide an example to better illustrate the difference: SARSA is categorized as an on-policy method because it revises its Q-values based on the Q-value of the subsequent state s' and the action a'' dictated by the current policy. It computes the return for state-action combinations with the assumption that the existing policy remains in effect. Conversely, Q-learning is considered off-policy since it modifies its Q-values using the Q-value of the next state s' and the action a' that is optimal according to a greedy policy. This means it projects the returns (total discounted future reward) for state-action pairs as if a greedy policy were consistently applied, even though it actually employs a different policy during learning.

3.4 Tabular Value-Based Reinforcement Learning

Tabular value-based reinforcement learning is a foundational approach in which the agent explicitly maintains a table of values representing the expected returns for state-action pairs or states. This method is particularly suited for environments with discrete and relatively small state and action spaces, where it is feasible to store and update values directly in a tabular format. By leveraging value functions, the agent systematically learns to estimate the long-term rewards associated with different decisions. The process involves iteratively updating the values based on the Bellman equation, gradually improving the policy that guides the agent's actions. Tabular methods, such as Q-learning and SARSA, offer a straightforward implementation of reinforcement learning concepts, serving as a crucial stepping stone toward more advanced techniques that handle larger or continuous spaces.

3.4.1 Value Functions and the Bellman Equation

In RL, value functions are used to estimate the expected return. The state-value function $V^\pi(s)$ under a policy π is defined as:

$$V^\pi(s) = \mathbb{E}_\pi \left[G_t \mid s_t = s \right], \tag{2}$$

which represents the expected return when starting from state s and following policy π thereafter.

Similarly, the action-value function $Q^\pi(s, a)$ gives the expected return for taking action a in state s and then following policy π:

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[G_t \mid s_t = s, a_t = a \right]. \tag{3}$$

The Bellman equation provides a recursive relationship for these value functions. For the state-value function, the Bellman equation is:

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} P(s'|s,a) \left[R(s,a) + \gamma V^{\pi}(s') \right]. \tag{4}$$

For the action-value function, the Bellman equation is:

$$Q^{\pi}(s,a) = R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) \sum_{a' \in A} \pi(a'|s') Q^{\pi}(s',a'). \tag{5}$$

These equations form the foundation for many RL algorithms, including those that use function approximation, such as DQN and DDQN, where the Q-values are approximated using neural networks. And with that said, let's move to the next chapter transitioning from standard RL to deep RL.

3.5 From Reinforcement Learning to Deep Reinforcement Learning

Traditional RL methods, such as Q-Learning and SARSA, have been effective in solving various sequential decision-making problems by learning policies that map states to actions through value functions or direct policy optimization. However, these methods typically rely on handcrafted features and tabular representations, making them impractical for environments with high-dimensional or continuous state spaces, such as those involving visual inputs or complex physical systems. As a result, the transition to *Deep Reinforcement Learning* (Deep RL) became a natural progression, leveraging the representational power of deep neural networks to automatically extract features from raw sensory data. By integrating deep learning with RL algorithms, Deep RL allows for end-to-end learning directly from high-dimensional inputs, such as images or videos, enabling the development of more general and scalable solutions that can tackle complex real-world tasks. This shift has led to significant advancements, exemplified by the success of the Deep Q-Network (DQN) Mnih et al., 2013, which pioneered the application of deep learning to RL, allowing agents to master sophisticated tasks, including playing Atari games at a human-competitive level.

3.6 Deep Reinforcement Learning

3.6.1 Fundamentals of DQN

DQN builds upon the traditional Q-Learning algorithm, which is a model-free RL method. In Q-Learning, the goal is to learn an action-value function $Q(s,a)$, which represents the expected cumulative reward (also called the return) when the agent takes action a in state s and follows an optimal policy thereafter. Formally, the action-value function can be defined as:

$$Q^{\pi}(s,a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a, \pi\right], \tag{6}$$

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_t + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t)\right], \tag{7}$$

where α is the learning rate, s_t and a_t are the state and action at time t, and r_t is the reward received after taking action a_t.

The Q-Learning algorithm updates the Q-values iteratively using the Bellman equation, which provides a recursive decomposition of the action-value function:

3.6.2 Deep Q-Network

In traditional Q-Learning, the Q-values are typically stored in a table, which becomes infeasible when dealing with large or continuous state spaces. DQN addresses this limitation by using a deep neural network (DNN) to approximate the Q-function. The traditional DQN architecture typically consists of convolutional layers followed by fully connected layers, enabling it to process raw pixel inputs and learn relevant features. However, in our case, we are going to use a variation of DQN that will be discussed later on.

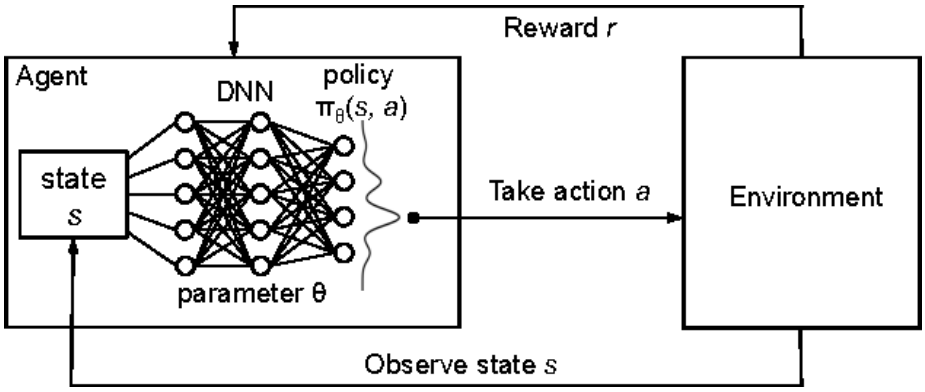


Figure 3: Representation of the interaction of the agent with the environment in a Deep RL setting where the policy is derived from a Deep network instead of a table. Image taken from Mao et al., 2016.

Fig 3 illustrates the interaction between an agent and the environment in a Deep Reinforcement Learning (Deep RL) framework. In this setting, the agent leverages a Deep Neural Network (DNN) to derive its policy, which maps observed states to actions. This contrasts with traditional RL, where policies or

value functions are typically represented using tabular methods. In Deep RL, particularly with techniques like Deep Q-Networks (DQN), the DNN is used to approximate the Q-value function, allowing the agent to handle large or continuous state spaces that would be infeasible with traditional tabular approaches. Unlike traditional RL, where state-action values might be stored in a table, DQN efficiently approximates these values using the network's parameters, enabling more scalable and effective learning in complex environments.

Moving to the notation, let θ represent the parameters of the DNN that approximates the Q-function, denoted as $Q(s, a; \theta)$. The DQN is trained by minimizing the following loss function, which is derived from the Bellman equation:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(y_t - Q(s, a; \theta)\right)^2\right], \tag{8}$$

where the target value y_t is given by:

$$y_t = r + \gamma \max_{a'} Q(s', a'; \theta^-). \tag{9}$$

Here, θ^- are the parameters of a target network, a copy of the Q-network θ that is periodically updated to stabilize training.

3.6.3 Key Innovations in DQN

DQN introduced several key innovations that addressed the instability and divergence issues commonly encountered when using neural networks in RL:

- **Experience Replay:** The agent's experiences (s_t, a_t, r_t, s_{t+1}) are stored in a replay buffer $\mathcal{D}$. During training, mini-batches of experiences are sampled uniformly from this buffer, breaking the correlation between consecutive experiences and thus improving the stability of the learning process.
- **Target Network:** A separate target network, with parameters θ^-, is used to compute the target values y_t. This network is updated less frequently than the Q-network, which helps reduce oscillations and divergence during training.
- **Reward Clipping:** In the Atari 2600 domain, rewards are clipped to the range $[-1, 1]$, which prevents large updates to the Q-values and stabilizes learning.

3.6.4 The DQN Algorithm

DQN represents a significant milestone in the field of deep reinforcement learning, demonstrating that it is possible to learn effective policies from high-dimensional sensory data. The combination of Q-Learning with deep learning, supported by innovations such as experience replay and target networks, allows DQN to overcome the challenges of instability and divergence, leading to

human-level performance in various domains. As a result, DQN has laid the foundation for further advancements in deep reinforcement learning, inspiring numerous extensions and improvements.

The DQN algorithm can be summarized as follows:

Algorithm 1 Deep Q-Network (DQN) Algorithm

Initialize the Q-network with random weights θ.
Initialize the target network with the same weights $\theta^- \leftarrow \theta$.
for each episode **do**
    Initialize the starting state s_0.
    for each time step t **do**
        Select an action a_t using an ϵ-greedy policy based on $Q(s_t, a_t; \theta)$.
        Execute action a_t and observe reward r_t and next state s_{t+1}.
        Store the transition (s_t, a_t, r_t, s_{t+1}) in the replay buffer $\mathcal{D}$.
        Sample a mini-batch of transitions (s, a, r, s') from $\mathcal{D}$.
        Compute the target value $y_t = r + \gamma \max_{a'} Q(s', a'; \theta^-)$.
        Perform a gradient descent step on the loss function $L(\theta) = \left(y_t - Q(s, a; \theta)\right)^2$ with respect to θ.
    end for
    Update the target network parameters periodically: $\theta^- \leftarrow \theta$.
end for

3.7 Double Deep Q-Network (DDQN)

While DQN represents a substantial advancement in the integration of deep learning with reinforcement learning, it is not without its limitations. One of the key issues with DQN is the overestimation bias that arises during the estimation of the Q-values. This bias occurs because the same network is used to select and evaluate the actions, which can lead to an overestimation of the action-value function, potentially resulting in suboptimal policies.

To address this overestimation issue, the Double Deep Q-Network (DDQN) algorithm was introduced. DDQN builds upon the DQN framework by decoupling the action selection and action evaluation processes, which mitigates the overestimation bias and leads to more accurate value estimates.

3.7.1 DDQN Methodology

The key difference between DQN and DDQN lies in how the target value y_t is computed. In DQN, the target value is calculated using the maximum Q-value of the next state s' as follows:

$$y_t^{\text{DQN}} = r + \gamma \max_{a'} Q(s', a'; \theta^-), \tag{10}$$

where $Q(s', a'; \theta^-)$ represents the action-value function approximated by the target network. This approach, while effective, introduces a bias because it uses the same values to both select and evaluate the next action.

In DDQN, this overestimation is mitigated by using the online network to select the action and the target network to evaluate it. Specifically, the target value in DDQN is computed as:

$$y_t^{\text{DDQN}} = r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta); \theta^-), \tag{11}$$

Here, $\arg\max_{a'} Q(s', a'; \theta)$ is the action selected by the online network, but the value of this action is evaluated using the target network $Q(s', a'; \theta^-)$. By separating these two steps, DDQN reduces the overestimation of Q-values, leading to more accurate learning.

3.7.2 Key Contributions and Impact of DDQN

DDQN provides a crucial improvement over DQN by addressing the overestimation bias, which can have significant impacts on the performance of RL algorithms, especially in environments where accurate value estimation is critical. Several empirical studies have demonstrated that DDQN leads to better performance and more stable learning compared to the original DQN, particularly in complex environments such as Atari 2600 games .

- **Decoupled Action Selection and Evaluation:** The primary innovation of DDQN is the decoupling of action selection from action evaluation. This modification reduces overestimation, leading to more accurate Q-value estimates and, consequently, more robust policies.
- **Improved Stability:** By mitigating overestimation bias, DDQN improves the stability of the learning process. This stability is particularly important in environments with high variability or where reward structures are sparse or noisy.
- **Broader Applicability:** The improvements introduced by DDQN have made it a preferred choice in various applications of deep reinforcement learning, from game playing to more practical applications like robotics and autonomous systems.

3.7.3 The DDQN Algorithm

DDQN represents a substantial refinement of the DQN algorithm, addressing one of its primary limitations which is the overestimation bias in Q-value estimation. By separating action selection from evaluation, DDQN achieves more

accurate and stable learning, which has led to its widespread adoption in the field of deep reinforcement learning. The development of DDQN emphasizes on the importance of continual refinement, contributing to more robust and reliable systems capable of performing in complex, real-world environments.

The DDQN algorithm follows a similar structure to DQN but with a key modification in the target calculation step:

Algorithm 2 Deep Q-Network (DQN) Algorithm with Double DQN Enhancements

Initialize the Q-network with random weights θ
Initialize the target network with the same weights $\theta^- \leftarrow \theta$
for each episode **do**
    Initialize the starting state s_0
    for each time step t **do**
        Select an action a_t using an ϵ-greedy policy based on $Q(s_t, a_t; \theta)$
        Execute action a_t and observe reward r_t and next state s_{t+1}
        Store the transition (s_t, a_t, r_t, s_{t+1}) in the replay buffer $\mathcal{D}$
        Sample a mini-batch of transitions (s, a, r, s') from $\mathcal{D}$
        Compute the target value $y_t^{\text{DDQN}} = r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta); \theta^-)$
        Perform a gradient descent step on the loss function $L(\theta) = \left(y_t^{\text{DDQN}} - Q(s, a; \theta)\right)^2$ with respect to θ
    end for
    Update the target network parameters periodically: $\theta^- \leftarrow \theta$
end for

3.8 Conservative Deep Q-Network (C-DQN)

The Conservative Deep Q-Network (C-DQN) is an extension of the DQN framework designed specifically for offline reinforcement learning scenarios Kumar et al., 2020b. Offline RL presents unique challenges, particularly when the agent cannot interact with the environment and must learn solely from pre-collected data. This characteristic of this algorithm makes it perfect for our setting. As discussed, the nature of our data is historical so this algorithm is directly applicable to our problem. However, one of the main issues in offline RL is the overestimation of Q-values for actions that are outside the distribution of the training data. This means that the offline algorithms might fail to generalize to data that follow different "trends" than the historic dataset they base their training on. The C-DQN algorithm addresses this by introducing a conservative penalty, discouraging the agent from overestimating the values of unseen or out-of-distribution actions.

3.8.1 C-DQN Methodology

In offline RL, the agent cannot interact with the environment, which can lead to an overestimation of Q-values for out-of-distribution (OOD) actions that were not seen in the training data. The C-DQN algorithm modifies the traditional Q-learning objective by adding a conservative regularization term that penalizes high Q-values for actions that are not well-represented in the offline dataset, leading to more reliable value estimates. The regular DQN algorithm estimates the Q-value as shown in Equation 9

This can cause overestimation in offline RL as unseen actions may get inflated Q-values. To mitigate this, C-DQN modifies the objective function to introduce conservatism in value estimation:

$$\min_{Q} \mathbb{E}_{(s,a)\sim\mathcal{D}} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^{-}) - Q(s,a) \right)^2 \right] + \alpha \left(\mathbb{E}_{a\sim\pi(a|s)}[Q(s,a)] - \mathbb{E}_{(s,a)\sim\mathcal{D}}[Q(s,a)] \right) \tag{12}$$

Here, the regularization term penalizes the Q-values of unseen actions, specifically actions sampled from the learned policy $\pi(a|s)$, while rewarding conservative estimates for actions sampled from the dataset $\mathcal{D}$. The coefficient α controls the weight of the penalty, which can be tuned to achieve the desired level of conservativeness. This ensures that the agent remains cautious in its predictions for out-of-distribution actions that were not observed in the dataset.

3.8.2 Key Contributions and Impact of C-DQN

C-DQN is particularly impactful in the domain of offline reinforcement learning, where interaction with the environment is not feasible. The conservative approach prevents over-optimism in value estimates, which can arise when the agent evaluates actions that are outside the distribution of the training data.

- **Conservative Regularization:** The conservative term in the C-DQN loss function prevents the overestimation of Q-values for actions not present in the dataset. This helps the agent avoid risky actions that it has not seen during training.
- **Improved Stability:** By focusing on actions observed in the dataset and penalizing unseen ones, C-DQN provides improved stability in policy learning, which is crucial in domains where safety and reliability are important.
- **Effective in High-Risk Domains:** The conservative nature of C-DQN makes it suitable for applications like autonomous driving, healthcare, and

robotics, where out-of-distribution actions can lead to unsafe outcomes.

3.8.3 The C-DQN Algorithm

The C-DQN algorithm extends the standard DQN framework by adding a conservative penalty to the Q-value updates. This ensures that the agent remains cautious when evaluating actions not present in the dataset. The C-DQN framework provides a regularization term to handle out-of-distribution actions, mitigating the common overestimation problem in offline reinforcement learning.

The algorithm can be summarized as follows:

Algorithm 3 Conservative Deep Q-Network (C-DQN) Algorithm

1: Initialize the Q-network with random weights θ
2: Initialize the target network with the same weights $\theta^- \leftarrow \theta$
3: **for** each episode **do**
4: Initialize the starting state s_0
5: **for** each time step t **do**
6: Select an action a_t using an ϵ-greedy policy based on $Q(s_t, a_t; \theta)$
7: Execute action a_t and observe reward r_t and next state s_{t+1}
8: Store the transition (s_t, a_t, r_t, s_{t+1}) in the replay buffer $\mathcal{D}$
9: Sample a mini-batch of transitions (s, a, r, s') from $\mathcal{D}$
10: Compute the conservative target value:

$$y_t^{\text{C-DQN}} = r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta); \theta^-) - \alpha \left(\mathbb{E}_{a \sim \pi(a|s)}[Q(s, a)] - \mathbb{E}_{(s,a) \sim \mathcal{D}}[Q(s, a)] \right) \tag{13}$$

11: Perform a gradient descent step on the loss function $L(\theta) = \left(y_t^{\text{C-DQN}} - Q(s, a; \theta) \right)^2$ with respect to θ
12: **end for**
13: Update the target network parameters periodically: $\theta^- \leftarrow \theta$
14: **end for**

Conservative Deep Q-Network (C-DQN) provides a robust framework for offline reinforcement learning by addressing the overestimation of Q-values in out-of-distribution actions. Its conservative regularization ensures that the agent remains cautious when learning from a fixed dataset, improving both the safety and performance of the policy. This makes C-DQN particularly useful for high-stakes environments where the cost of making mistakes is high.

3.9 Soft Actor-Critic (sac)

To understand the Soft Actor-Critic Haarnoja et al., 2018 algorithm someone must first understand the entropy-regularized reinforcement learning framework, which modifies traditional RL equations to incorporate entropy measures.

3.9.1 Entropy-Regularized Reinforcement Learning

Entropy measures the randomness or unpredictability of a random variable. For example, a biased coin that frequently lands heads has low entropy, while a fair coin with an equal chance of landing heads or tails exhibits high entropy.

The following equations are from OpenAI SAC documentation

Consider a random variable x with a probability mass or density function P. The entropy H of x is calculated from its distribution P as follows:

$$H(P) = \mathbb{E}_{x\sim P}[-\log P(x)]. \tag{14}$$

In entropy-regularized reinforcement learning, the agent receives an additional reward at each timestep, which is proportional to the entropy of the policy at that timestep. This modifies the reinforcement learning objective to:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t \left(R(s_t, a_t, s_{t+1}) + \alpha H\left(\pi(\cdot|s_t)\right)\right)\right], \tag{15}$$

where $\alpha > 0$ is a coefficient balancing the trade-off. Assuming an infinite-horizon discounted scenario, we redefine the value functions to incorporate entropy bonuses:

The value function V^π includes entropy bonuses at every timestep:

$$V^\pi(s) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t \left(R(s_t, a_t, s_{t+1}) + \alpha H\left(\pi(\cdot|s_t)\right)\right) \mid s_0 = s\right] \tag{16}$$

The action-value function Q^π, on the other hand, incorporates entropy bonuses from every timestep except the first:

$$Q^\pi(s, a) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) + \alpha \sum_{t=1}^{\infty} \gamma^t H\left(\pi(\cdot|s_t)\right) \mid s_0 = s, a_0 = a\right] \tag{17}$$

The relationship between V^π and Q^π is established as:

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi}\left[Q^{\pi}(s,a)\right] + \alpha H\left(\pi(\cdot|s)\right) \tag{18}$$

and the Bellman equation for Q^{π} becomes:

$$Q^{\pi}(s,a) = \mathbb{E}_{s'\sim P, a'\sim\pi}\left[R(s,a,s') + \gamma\left(Q^{\pi}(s',a') + \alpha H\left(\pi(\cdot|s')\right)\right)\right] \tag{19}$$

or equivalently:

$$Q^{\pi}(s,a) = \mathbb{E}_{s'\sim P}\left[R(s,a,s') + \gamma V^{\pi}(s')\right]. \tag{20}$$

3.9.2 Additional Considerations

The setup of value functions in this entropy-regularized framework is somewhat arbitrary, and other formulations could include the entropy bonus at the first timestep. Definitions may vary slightly across different research papers on this topic.

3.10 Sports Science: a quick overview

Sports science encompasses various disciplines, including physiology, biomechanics, psychology, and nutrition, all aimed at improving athletic performance and reducing injuries. It involves the application of scientific principles to understand and enhance both the physical and mental aspects of athletic training and competition. The precision required in crafting training schedules for elite athletes underscores the importance of personalized approaches tailored to the unique needs and conditions of each athlete.

4 Data

This chapter provides an overview of the data utilized in this research, which consists of historical training schedules of elite athletes provided by a Dutch professional team. Due to competitive and privacy concerns, access to the data is restricted. As the ultimate goal is to apply a Reinforcement Learning (RL) framework, the raw data must be transformed into a structure that resembles Markov Decision Process (MDP) transitions. This transformation process (detailed in Section 4.2) is crucial for enabling RL algorithms to learn meaningful policies.

4.1 Historic Dataset

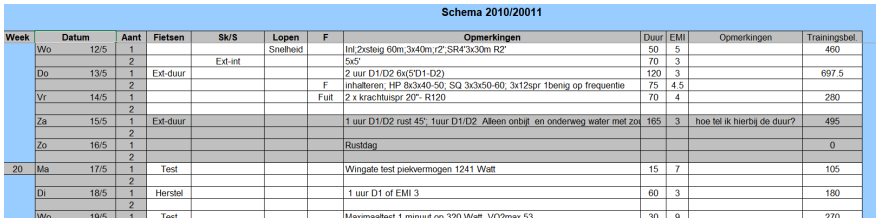
The core dataset is stored in Excel files, each representing a year of training schedules. Within each file, the data are organized such that each day is split into two rows: one for the morning training session and another for the evening session. The dataset contains five primary exercise types:

- **Cycling**
- **Skating**
- **Walking**
- **Strength**
- **Conditioning**

Additionally, several columns capture important attributes related to each training session:

- **Duration:** Time spent on the exercise (in minutes).
- **RPE (Rate of Perceived Exertion):** A numerical value representing the perceived intensity of the exercise, with higher values indicating more challenging sessions.
- **Total Training Load:** The product of *Duration* and *RPE*, representing the overall workload of the session.
- **Comments:** Additional notes provided by the coach for each session.

A snippet of the raw data structure is shown in Figure 4. For this research, data from five consecutive years were combined to form a single dataset. Each row corresponds to a particular session (morning or evening), along with various metrics of interest.



Schema 2010/20011

Week	Datum	Aant	Fietsen	Sk/S	Lopen	F	Opmerkingen	Duur	EMI	Opmerkingen	Trainingsbel.
	Wo 12/5	1			Snelheid		Inl;2xsteig 60m;3x40m;r2';SR4'3x30m R2'	50	5		460
		2		Ext-int			5x5'	70	3		
	Do 13/5	1	Ext-duur				2 uur D1/D2 6x(5'D1-D2)	120	3		697.5
		2				F	inhalteren; HP 8x3x40-50; SQ 3x3x50-60; 3x12spr 1benig op frequentie	75	4.5		
	Vr 14/5	1				Fuit	2 x krachtuispr 20"- R120	70	4		280
		2									
	Za 15/5	1	Ext-duur				1 uur D1/D2 rust 45'; 1uur D1/D2 Alleen onbijt en onderweg water met zo	165	3	hoe tel ik hierbij de duur?	495
		2									
	Zo 16/5	1					Rustdag				0
		2									
20	Ma 17/5	1	Test				Wingate test piekvermogen 1241 Watt	15	7		105
		2									
	Di 18/5	1	Herstel				1 uur D1 of EMI 3	60	3		180
		2									
	Wo 19/5	1	Test				Maximaaltest 1 minuut op 320 Watt, VO2max 53	30	9		270

Figure 4: Sample of the raw data in Excel form, containing training schedules for one year. The dataset includes five main exercise categories: Cycling, Skating, Walking, Strength, and Conditioning.

4.2 Data Preprocessing

Before the dataset could be used in a reinforcement learning pipeline, extensive data cleaning and transformation steps were required. Given that each Excel file had slightly different column layouts and naming conventions, the overarching goal was to harmonize the data into a single DataFrame with consistent columns.

4.2.1 Merging and Cleaning

1. **Merging Multiple Excel Files:** The raw data were provided in five Excel files (`Trainingsstatus2010-2011.xls`, `Trainingsstatus2011-2012.xls`, etc.), each containing training information for one year. We first dropped non-essential or redundant columns, such as additional commentary or unused intermediate metrics, ensuring only relevant columns remained (e.g., *Week*, *Date*, *Duration*, *RPE*, *Cycling*, etc.). The data from the five files were then concatenated row-wise into a single DataFrame.

2. **Forward-Filling Certain Columns:** Columns like *Week* and *Date* sometimes spanned multiple sessions and could contain empty cells. Forward filling (`ffill`) was applied to propagate these values down until the next non-empty row was encountered.

3. **Handling Missing Values:** For numerical columns such as *Duration* and *RPE*, any remaining `NaN` values were set to zero, reflecting that no training was performed or no intensity was recorded. Non-numeric columns were also standardized to avoid undefined session types.

4. **Binary Encoding of Training Types:** The original data contained columns like *Cycling*, *Skating*, *Walking*, and *Str&Cond* indicating whether a particular exercise was performed. After filling empty cells with zeros, each column was converted into a binary indicator (*1* if that exercise was performed, *0* otherwise). An additional *Rest* column was inserted to mark days with no recorded exercise.

4.2.2 Incorporating Performance (Reward) Data

Next, an external file (`Astrand.xlsx`) containing performance metrics was merged with the main dataset. The primary metric (labeled *Reward*) was derived from tests that included heart-rate measures and other athlete-specific evaluations. Key steps included:

1. Converting all relevant date fields to a standard `YYYY-MM-DD` format.

2. Merging the performance file (containing the date and the computed *Reward*) with the main training dataset via a *left join* on the date.

3. Forward/Backward filling or zero-filling of missing reward values for dates with no recorded test data, to ensure every row in the training dataset had a numerical reward value.

This *Reward* column functions as the scalar feedback signal for the RL algorithms, informing whether a particular sequence of training actions is beneficial for the athlete's performance.

4.2.3 Feature Engineering and Scaling

To better capture temporal and seasonal effects, the following transformations were applied:

- **Day-of-Year, Day-of-Week:** The *Date* column was split into numerical features such as day of the year (ranging from 1 to 365) and day of the week (0 for Monday through 6 for Sunday). These help capture periodicities in training schedules.
- **One-Hot Encoding of Categorical Variables:** Categorical fields, for instance the session type (morning vs. evening) or day-of-week, were one-hot encoded using `OneHotEncoder` to create independent binary variables for each category.
- **Scaling Numerical Variables:** Columns such as *Week*, *Duration*, *RPE*, and *Day_of_Year* were standardized (using `StandardScaler`) to have zero mean and unit variance, preventing any one feature from dominating the learning process.

A final `DataFrame` was then constructed, containing a mixture of scaled numeric features, one-hot-encoded categorical features, multiple binary action indicators (e.g. *Cycling*, *Skating*, *Walking*, *Str&Cond*, *Rest*), and a *Reward* column. This final structure was saved to a CSV file for reproducibility and serves as the input to the ML/RL pipelines.

4.2.4 From Data to Markov Decision Processes

To apply reinforcement learning, we interpret each row of the final dataset as part of an MDP *transition*:

- **State (S):** A vector of features describing the athlete's status on a given day (e.g., scaled *Duration*, *RPE*, day-of-week, day-of-year, plus any latched features from previous sessions if used).
- **Action (A):** The decision made for that session—in practice, which exercise is performed. In the final DataFrame, this can be represented as one of the binary columns (*Cycling*, *Skating*, etc.) or a combined categorical variable in more advanced setups.
- **Reward (R):** The *Reward* column merged from the performance tests. This serves as feedback for how effective the chosen action was, factoring in the athlete's subsequent test results.

- **Next State (S'):** The features in the following row (i.e., next training session), representing how the athlete's state evolves over time. Because this is historical data, the state transitions are fixed by the dataset, simulating the environment's response.

By aligning the data in chronological order, each pair of consecutive rows forms a (S, A, R, S') tuple, effectively turning the entire historical record into a time-series of transitions. The key assumption is that day-to-day transitions approximate the Markov property, i.e., the next day's condition depends primarily on the current day's state and action, plus any explicitly-engineered features (e.g., session type).

4.2.5 Summary of Processing Steps

Figure 5 outlines the main steps in preparing the raw training data and performance metrics for RL. The preprocessing not only standardizes the formatting but also enhances the data's suitability for sequential decision-making analyses.

1. Load, merge, and clean raw Excel files
2. Forward-fill missing weeks, dates, and other columns
3. Convert exercise columns to binary indicators and add *Rest*
4. Merge external performance file for reward signals
5. Engineer features (day-of-week, day-of-year)
6. Encode categorical variables, scale numeric variables
7. Store final DataFrame for RL

Figure 5: A high-level view of the data preprocessing pipeline used in this study.

This refined dataset serves as the foundation for both the traditional machine learning models and the offline RL setups explored in subsequent chapters. The careful cleaning, merging, and feature engineering steps ensure that the data are coherent, consistently formatted, and representative of an MDP, making it possible to train, validate, and compare different models for crafting personalized training schedules.

4.3 Data Statistics

To provide context for the learning problem, we present statistics on the final dataset used across supervised and RL experiments. The dataset consists of $3,290$ chronologically ordered transitions, each representing a training session annotated with a discrete action (training type), input features (athlete context), and a scalar reward.

Action Space Distribution. The target in all supervised and deep reinforcement learning models is the *training action*, which can be one of five types:

`Cycling`, `Skating`, `Walking`, `Str&Cond`, or `Rest`. Table 1 shows the distribution of these actions in the final dataset.

Action	**Proportion (%)**
Skating	33.0%
Rest	29.5%
Cycling	24.8%
Str&Cond	10.6%
Walking	2.0%

Table 1: Distribution of training actions in the dataset.

Majority-Class Baseline. Given the class imbalance in the dataset, we include a baseline classifier that always predicts the most frequent class (`Skating`). This yields a baseline accuracy of 33.04%, which serves as a reference for evaluating learned policies. For example, our best deep model achieves approximately 85% accuracy—significantly outperforming the baseline and demonstrating its ability to capture meaningful structure in the training data.

Reward Statistics. The reward signal, used in all reinforcement learning settings, is a scalar derived from athlete physiological test results. Table 2 provides summary statistics for this reward.

Metric	**Value**
Mean Reward	11.73
Standard Deviation	40.24
Minimum	0.0
Maximum	168.0

Table 2: Reward statistics in the dataset.

These statistics contextualize model performance and emphasize the importance of learning generalizable decision policies, rather than relying solely on dominant action patterns such as "Skating."

5 Methods and Experimental setup

This section summarizes the methodologies and experimental setup employed in this study to develop optimal and personalized training schedules for elite athletes using mainly Reinforcement Learning techniques. The approach is structured in stages to progressively enhance the complexity of the models and adapt to the nature of the available data, which consists of historical training records without a real-time interactive environment. This means that certain online RL algorithms will be manipulated to fit the offline nature of the data. The technical details of these manipulations will also be discussed.

The study begins with a tabular RL approach, where the problem is initially formulated in a discrete state-action space suitable for simpler, table-based reinforcement learning methods. This stage serves as a foundational step to understanding the dynamics of the training environment and establishing a basic understanding of the performance metrics.

Following the tabular approach, the study advances to deep RL, leveraging both online and offline algorithms to handle more complex state and action spaces. Given that the dataset consists of historical data and a real environment where the agent can interact dynamically does not exist, online RL algorithms are adapted for offline use. This means that the archive dataset will play the role of the environment as the agent will be able to move. learn and sample experiences only within this frame.

Finally, the study incorporates traditional machine learning techniques to predict the outcomes of different training schedules. This predictive approach complements the RL strategies by providing additional insights into the relationships between training variables and athlete performance. It also offers a deeper understanding and serves as a benchmark for comparing how a completely different approach might perform. Additionally, it helps to assess whether RL is the most suitable method given the specific setup.

The following subchapters detail each approach, including the specific algorithms used, their configurations, and the experimental conditions under which they were tested.

5.1 Tabular approach

In the scope of this approach, the problem was formulated as an MDP as follows: The training sessions were categorized into three levels based on training load: Low, Medium, and High intensity. This categorization replaced the five different exercise types present in the raw data with three discrete action buckets, simplifying the action space. To constrain the state space and make the initial model more manageable, a state was defined as a sequence of three consecutive training sessions. For example, a state could be represented as a tuple like ('Medium', 'Medium', 'High'). This formulation resulted in a total $3^3 = 27$ unique states, as there are three possible intensity levels for each of the three consecutive sessions.

Within this simplified setup, a Q-table algorithm was employed to train a Q-table model that learns the optimal next action based on the current state. The reward function was defined such that the agent receives a reward of 1 if the predicted next action matches the action that was actually taken in the dataset, and a reward of 0 otherwise. This approach allows the model to iteratively learn the sequence of actions that most closely aligns with the real-world decisions made by the athletes and coaches.

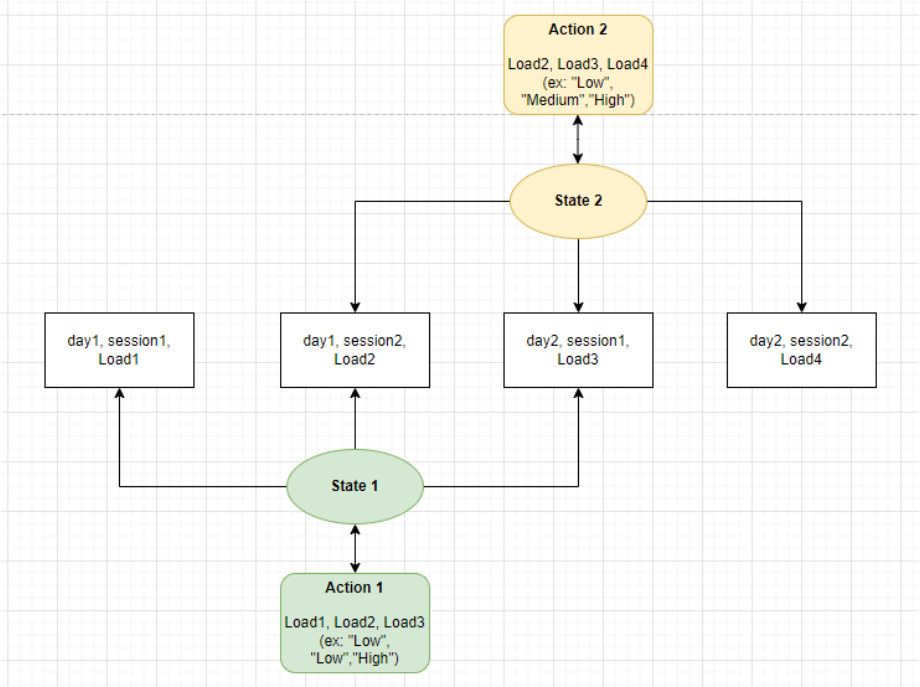


Figure 6: Illustration of state-action pairs in the tabular approach. Each pair follows in chronological order, forming a time-series-like MDP.

In Figure 6 the state-action pairs are displayed as an example. Each state-action pair follows the next one in chronological order, forming a time-series-like MDP. The first state consists of the initial three training sessions, with the corresponding action being a tuple that includes the load categories for these sessions. In this example, the first action (Action 1) is represented by the tuple (”Low”, ”Low”, ”High”).

Since the data are historical, and the agent cannot actively explore different next states, the subsequent state and action are predetermined. The next state, State 2, is defined as the sequence of sessions shifted by one (i.e., the second, third, and fourth sessions), with the corresponding action (Action 2) being another load category tuple, such as (”Low”, ”Medium”, ”High”).

In the same context, multiple experiments were done adjusting every time the state space and experimenting with the Load categorization and consequently the definition of the Action space. The specifics will be discussed in the next

chapter.

The state-action tabular example directly contributes to understanding how RL can be effectively applied to model coaching decision-making. By simplifying the problem into discrete states and actions, the model replicates the thought process of a coach in generating training schedules.

5.2 "Online" deep RL approach

In this section, the online deep RL models that were utilized will be discussed. The online is put on quotes because the traditional online algorithms and methods were modified to fit the offline nature of the dataset.

5.2.1 Deep Q-Network (DQN)

The implementation of this experiment is based on the Deep Q-Network (DQN) model using PyTorch, adapted from the official PyTorch tutorial on reinforcement learning. Originally designed for online reinforcement learning, this DQN model has been modified to function in an offline setting, where the agent learns solely from a fixed dataset of historical data. This is crucial for the nature of this experiment, as the data is archived and does not involve real-time interaction with the environment.

5.2.2 Key Components and Modifications

- **Data Preparation**: The experiment begins with loading a preprocessed dataset consisting of historical training data for athletes. This dataset is divided into training and testing sets to facilitate evaluation. Each entry in the dataset is formatted as a named tuple (`Transition`), which includes the state (representing the athlete's current condition), action (representing the training intensity or type), reward (performance metrics), and next state (resulting condition after the action). This structure mimics the environment-agent interaction in online reinforcement learning but is based entirely on pre-existing data, making it suitable for offline learning.

- **Replay Memory for Offline Learning**: The replay memory component, which is typically used to store an agent's experiences in online settings, is adapted here to store all possible state transitions from the historical dataset. This means that the buffer is filled with the total of historical datasets. Since the agent cannot generate new experiences in an offline setting, it samples from this fixed replay memory to learn from past decisions and outcomes. The replay buffer uses a deque data structure to manage the memory efficiently and supports random sampling during training, a crucial feature for preventing correlation between samples.

- **Deep Q-Network (DQN) Model**: The architecture of the DQN model consists of three fully connected layers with ReLU activation functions,

following the standard DQN design. The input layer takes the state representation, and the output layer predicts Q-values for each possible action. To enhance stability during training, especially with offline data, batch normalization layers are applied. These layers help ensure that the model generalizes better, despite the static nature of the data. The primary goal of this network is to approximate the optimal Q-value function, which predicts the expected future rewards for each action given the current state.

- **Training Setup and Modifications for Offline Learning**: Key hyperparameters, such as learning rate, discount factor γ, and batch size, follow the original DQN setup. However, due to the offline setting, the exploration-exploitation trade-off managed by epsilon-greedy policies in online environments is not directly applicable here. In this offline scenario, the model focuses exclusively on exploiting the existing dataset by learning from transitions stored in replay memory. The target network, which stabilizes training by providing fixed Q-value targets, is periodically updated through a soft update rule. This mechanism ensures that the target network's weights are gradually updated, preventing oscillations in Q-value estimates.

- **Offline Training Optimization**: The DQN model is optimized using a training function that samples random batches of transitions from the replay memory. In each training step, the loss is computed using the Bellman equation by comparing the predicted Q-values with the expected values derived from the target network. A Huber loss function is used to reduce the impact of outliers, which is particularly useful for offline learning with historical data. Gradient clipping is also applied to prevent exploding gradients and ensure stable updates to the network's parameters.

- **Training Execution and Loss Monitoring**: The training loop is modified to run for a fixed number of episodes, during which the policy network is updated based on the available historical data. The target network is updated periodically to maintain stable learning. Throughout the training process, loss values are recorded and smoothed to monitor convergence and performance. This helps track how well the model generalizes from past data to optimize training schedules. Visualizations of the loss over time provide insights into the model's convergence behavior.

5.2.3 Formulation and Pseudocode

In adapting the Deep Q-Network (DQN) algorithm for an offline reinforcement learning setting, several mathematical modifications are necessary. The key mathematical differences between the online Deep Q-Network (DQN) and the offline DQN are highlighted in the following formulations, focusing on how the offline setting alters certain components of the algorithm.

1. Data Distribution

Online DQN: Experiences are collected by the agent interacting with the environment using its current policy π, and stored in a replay buffer.

$$(s, a, r, s') \sim \mathcal{D}_{\text{online}}, \quad \text{where } \mathcal{D}_{\text{online}} \text{ is updated continually} \tag{21}$$

Offline DQN: Uses a fixed dataset $\mathcal{D}$ collected by a behavior policy π_β, with no interaction during training.

$$(s, a, r, s') \sim \mathcal{D}, \quad \text{where } \mathcal{D} \text{ is static} \tag{22}$$

2. Exploration Strategy

Online DQN: Employs an ϵ-greedy policy to balance exploration and exploitation during data collection.

$$a = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \arg\max_{a'} Q(s, a'; \theta) & \text{with probability } 1 - \epsilon \end{cases} \tag{23}$$

Offline DQN: No exploration is possible since the agent does not collect new data; it relies entirely on the actions in the dataset.

3. Loss Function Expectation

Online DQN:

$$L_{\text{online}}(\theta) = \mathbb{E}_{(s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}_{\text{online}}} \left[\left(y_t - Q(s_t, a_t; \theta) \right)^2 \right] \tag{24}$$

Offline DQN:

$$L_{\text{offline}}(\theta) = \mathbb{E}_{(s_i, a_i, r_i, s_i') \sim \mathcal{D}} \left[\left(y_i - Q(s_i, a_i; \theta) \right)^2 \right] \tag{25}$$

- Both use the same form of loss function but differ in the data distribution over which the expectation is taken.

4. Target Value Computation

Online DQN: Computes the target Q-value using the maximum over all possible actions at the next state s'.

$$y = r + \gamma \max_{a'} Q(s', a'; \theta^-) \tag{26}$$

Offline DQN:

$$y_i = \begin{cases} r_i, & \text{if } s_i' \text{ is terminal} \\ r_i + \gamma \max_{a'} Q(s_i', a'; \theta^-), & \text{otherwise} \end{cases} \tag{27}$$

- In offline DQN, the next state s_i' comes from the fixed dataset, and actions are limited to those present in $\mathcal{D}$.

5. Policy Update Dynamics

Online DQN: The policy is continually updated based on new experiences, with the data distribution evolving as the policy improves.

Offline DQN: The policy update relies solely on the fixed dataset, which can lead to distributional shift if the learned policy deviates from the behavior policy that generated $\mathcal{D}$.

6. Gradient Update

Both Online and Offline DQN: The gradient of the loss with respect to the network parameters θ is computed similarly, but the expectation is over different data distributions.

$$\nabla_\theta L(\theta) = \mathbb{E}_{(s,a,r,s')} \left[(Q(s, a; \theta) - y) \nabla_\theta Q(s, a; \theta) \right] \tag{28}$$

Gradient clipping

$$\nabla_\theta L(\theta) \leftarrow \delta \cdot \frac{\nabla_\theta L(\theta)}{\max(\delta, \|\nabla_\theta L(\theta)\|)} \tag{29}$$

- Gradient clipping is applied to stabilize training by preventing excessively large updates.

Summary of Differences

The main differences between online and offline DQN can be summarized below:

- **Data Source**: Online DQN collects new data during training; offline DQN relies on a fixed dataset without environmental interaction.
- **Exploration**: Online DQN uses exploration strategies; offline DQN cannot explore and must work within the provided data.
- **Target Network Update**: Online DQN updates the target network periodically; offline DQN employs a soft update at each training step.
- **Action Selection**: Offline DQN may limit action considerations to those present in the dataset to prevent overestimation.

- **Policy Evaluation**: The expectation in the loss function for offline DQN is over the static dataset, not the agent's experience.
- **Overestimation Bias**: Offline DQN is more susceptible to overestimation errors due to the inability to gather new data to correct overestimated Q-values.

To illustrate better how this offline version of the DQN works you below is the pseudocode.

Algorithm 4 Offline Deep Q-Network (DQN) Algorithm

Initialize replay buffer $\mathcal{D}$ with transitions from the offline dataset.
Initialize policy network $Q(s, a; \theta)$ with random weights θ.
Initialize target network $Q(s, a; \theta^-)$ with weights $\theta^- \leftarrow \theta$.
Set hyperparameters: batch size B, learning rate α, discount factor γ, target update rate τ, and maximum gradient norm δ.
for each training step **do**
    Sample a mini-batch of B transitions $\{(s_i, a_i, r_i, s'_i)\}_{i=1}^{B}$ from $\mathcal{D}$.
    Compute the target Q-values for each transition:

$$y_i = \begin{cases} r_i, & \text{if } s'_i \text{ is terminal} \\ r_i + \gamma \max_{a'} Q(s'_i, a'; \theta^-), & \text{otherwise} \end{cases}$$

    Compute the current Q-values for actions taken:

$$q_i = Q(s_i, a_i; \theta)$$

    Compute the loss over the mini-batch:

$$L(\theta) = \frac{1}{B} \sum_{i=1}^{B} (y_i - q_i)^2$$

    Perform a gradient descent step on $L(\theta)$ with respect to θ:

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta)$$

    Apply gradient clipping:

$$\nabla_\theta L(\theta) \leftarrow \delta \cdot \frac{\nabla_\theta L(\theta)}{\max(\delta, \|\nabla_\theta L(\theta)\|)}$$

    Update the target network using a soft update:

$$\theta^- \leftarrow \tau\theta + (1 - \tau)\theta^-$$

end for

5.2.4 Double Deep Q-Network (DDQN)

The Double Deep Q-Network (DDQN) algorithm is an enhancement of the standard DQN architecture that addresses the issue of *overestimation bias*, which is prevalent in Q-learning algorithms. In the standard DQN, the same network is used for both action selection and action evaluation, which can lead to overestimation of Q-values for certain state-action pairs. DDQN mitigates this issue by decoupling the action selection and action evaluation processes, leading to more accurate and stable value estimates. This is particularly important in offline re-

inforcement learning scenarios, where overestimation could result in suboptimal policy learning based solely on historical data.

In this experiment, DDQN has been adapted to work in an offline reinforcement learning setting, utilizing historical training schedules to learn optimal policies for future decision-making. The implementation is largely based on the official PyTorch example, with modifications to incorporate the DDQN framework for action evaluation.

5.2.5 Key Components and Modifications

- **Data Preparation**: The dataset for training the DDQN model is the same as that used for DQN. It consists of a named tuple, `Transition`, which encapsulates the state (representing the athlete's condition), the action (training intensity or type), the reward (performance metric), and the next state (the condition after the training). The goal is to learn from these transitions in order to optimize future training schedules.
- **Replay Memory for Offline Learning**: Similar to DQN, DDQN uses a replay memory to store all the transitions from the fixed dataset. As the model operates in an offline setting, it does not interact with a live environment. Instead, it learns by sampling mini-batches of transitions from the replay memory. The *SequentialReplayMemory* class is used to manage this process, ensuring efficient sampling of transitions during training.
- **DDQN Model Architecture**: The neural network architecture for DDQN is similar to that of DQN, consisting of three fully connected layers, each followed by ReLU activation functions. To improve stability during training, especially given the offline nature of the dataset, batch normalization and dropout layers are included. The network outputs Q-values for each possible action, given the state input.
- **Key Modification - Decoupling Action Selection and Evaluation**: The main difference between DDQN and DQN is in how actions are selected and evaluated. In DDQN, the *policy network* is used to select the action, while the *target network* is responsible for evaluating the value of the selected action. This decoupling helps prevent overestimation by ensuring that the evaluation is based on the more stable values provided by the target network. Specifically, the action with the highest Q-value is selected by the policy network, but its value is evaluated using the target network, reducing the bias inherent in using a single network for both purposes.
- **Training Setup**: The training process follows the same general structure as DQN, with similar hyperparameters such as learning rate, discount factor γ, and batch size. However, in DDQN, the target Q-value computation involves selecting the best action using the policy network and evaluating it with the target network. Since the model is trained offline, the

focus is on exploiting the fixed dataset rather than balancing exploration and exploitation, which is more relevant in online reinforcement learning settings.

- **Offline Training Optimization**: During training, the Bellman error is calculated using the expected Q-values, with the action selection and evaluation performed by separate networks. The Huber loss function is used to minimize the impact of outliers in the dataset, and gradient clipping is applied to prevent unstable updates. The key difference in DDQN is the computation of the Q-value target. The action is selected by the policy network and its value is evaluated using the target network. This decoupling significantly reduces overestimation bias.

- **Training Execution**: The training loop is similar to that of DQN, running for a fixed number of episodes. The loss values are recorded throughout the process to monitor convergence. The target network is updated periodically using a hard update rule, in which the policy network's weights are copied to the target network every fixed number of episodes (e.g., every 10 episodes). The convergence of the model is monitored by tracking the loss over time and plotting the smoothed loss values.

5.2.6 Key Contributions and Impact of DDQN

- **Reduction of Overestimation Bias**: By separating the action selection and evaluation processes, DDQN significantly reduces the overestimation of Q-values, leading to more stable and accurate learning. This is particularly important in offline settings, where the agent must rely entirely on past data without the opportunity for further exploration.

- **Improved Stability**: The use of a target network provides more stable updates during the learning process, reducing oscillations in Q-value estimates. The decoupling of action selection and evaluation improves the model's ability to generalize from the fixed dataset, minimizing the risk of overestimating the value of actions that have not been seen before.

- **Broader Applicability**: DDQN is well-suited for environments with high uncertainty, where accurate value estimation is critical. The reduction of overestimation bias makes it a robust algorithm for offline learning tasks, such as optimizing training schedules based on historical data.

5.2.7 Soft Actor-Critic (SAC)

The Soft Actor-Critic (SAC) algorithm is an advanced reinforcement learning method designed primarily for continuous action spaces. However, in this experiment, SAC has been adapted for discrete action spaces, using offline reinforcement learning based on pre-collected training schedules. This implementation of SAC is heavily based on the open-source repository cleanrl-SAC. SAC is distinguished by its use of a stochastic policy, encouraging exploration through

entropy maximization while aiming to optimize the expected return.

In contrast to Q-learning algorithms like DQN and DDQN, SAC incorporates a policy network (actor) that selects actions, and two Q-networks (critics) that evaluate those actions. The agent balances exploration and exploitation by introducing stochasticity in action selection, leading to more robust learning.

5.2.8 Key Components and Modifications

- **Data Preparation**: The historical training data used in SAC is preprocessed similarly to the other reinforcement learning models. Each transition in the dataset is stored using a `Transition` named tuple, which includes the state, action, reward, and next state. The dataset is split into training and test sets, and the training set is used to populate a replay buffer. This buffer will be used to sample mini-batches of transitions during training.

- **Replay Buffer for Offline Learning**: SAC relies on a replay buffer to store past transitions. In the offline learning setup, this buffer is populated with pre-collected historical data. No new experiences are added during training, as the agent is learning solely from this fixed dataset. The *CustomReplayBuffer* class manages the replay buffer, allowing for efficient random sampling of mini-batches during training.

- **Soft Q-Networks (Critics)**: The SAC implementation includes two Q-networks, commonly referred to as critics, which evaluate the value of state-action pairs. These networks are fully connected neural networks with two hidden layers, employing ReLU activations. Each Q-network outputs the Q-value for each action in the discrete action space. To prevent overestimation of Q-values, the two Q-networks are updated separately, and target Q-networks are updated periodically to maintain stable learning. The target networks are soft updates of the main Q-networks, which means that their weights are slowly adjusted towards the main network weights.

- **Policy Network (Actor)**: SAC's policy network, or actor, selects actions based on the input state. Unlike deterministic policies, SAC uses a stochastic policy that outputs a probability distribution over actions. The policy network is a fully connected neural network with two hidden layers. During training, the policy network updates less frequently than the Q-networks, which helps maintain stability and balance between learning from exploration and exploitation. The output of the actor network uses a softmax function to generate probabilities over the discrete action space.

- **Entropy and Exploration**: One of the key features of SAC is its entropy term, which encourages the agent to explore a wide range of actions. This is crucial in offline settings, where the dataset is fixed, and the agent

must learn to generalize from the available data. The entropy coefficient, denoted as α, determines the balance between exploration (via entropy) and exploitation (via reward maximization). In this implementation, α can either be fixed or dynamically tuned during training. When automatic entropy tuning is enabled, the target entropy is determined based on the number of actions, and α is adjusted automatically to maintain this target entropy.

- **Training Setup and Target Network Updates**: During training, the Q-networks are updated by sampling batches of transitions from the replay buffer. The Q-networks learn by minimizing the difference between the predicted Q-values and the target Q-values, which are computed using the Bellman equation. The target Q-values are computed by taking the minimum value from the two Q-networks, which reduces the risk of overestimation. The actor network is updated to maximize the expected Q-value, while also incorporating the entropy term to ensure sufficient exploration.

  The target networks for the Q-functions are updated periodically using a soft update rule. In this rule, the target network weights are slowly adjusted toward the main network weights, ensuring stable updates without abrupt changes in the target Q-values.

- **Policy and Q-Network Optimization**: Both the policy network and the Q-networks are optimized using the Adam optimizer. The optimization process for the Q-networks is based on the difference between the predicted Q-values and the target values, while the policy network is optimized to maximize the expected return. The optimization process includes minimizing a policy loss that accounts for both the Q-values and the entropy term.

- **Policy Evaluation**: After training, the policy network is evaluated using the test dataset. The evaluation involves comparing the actions predicted by the policy with the actual actions from the dataset. The accuracy of the policy is measured by the percentage of correctly predicted actions. This allows us to assess how well the learned policy generalizes to unseen data.

5.2.9 Key Contributions and Impact of SAC

- **Entropy-Driven Exploration**: SAC's entropy term ensures that the agent explores a broader range of actions, which is particularly useful in offline learning environments where the dataset is fixed. This encourages the agent to avoid prematurely settling on suboptimal policies.

- **Stabilization Through Two Critic Networks**: The use of two separate Q-networks (critics) helps stabilize the training process by minimizing overestimation of Q-values. By taking the minimum of the Q-values from

the two networks, SAC ensures that the agent makes more conservative and reliable decisions.

- **Adaptive Entropy Tuning**: SAC can adaptively tune the entropy coefficient α, allowing the agent to adjust the balance between exploration and exploitation dynamically. This makes the learning process more flexible, particularly in complex environments where the optimal exploration-exploitation balance may vary over time.

5.3 Offline deep RL

5.3.1 Conservative Q-Learning (CQL)

Conservative Q-Learning (CQL) is a reinforcement learning algorithm tailored for offline settings, where the agent learns solely from pre-collected data rather than interacting with a live environment. One of the key challenges in offline reinforcement learning is the overestimation of Q-values for out-of-distribution (OOD) actions—actions not well-represented in the dataset. CQL addresses this problem by introducing a conservative regularization term that penalizes overly optimistic Q-value estimates for actions outside the data distribution.

In this experiment, the CQL algorithm is applied to a fixed dataset of historical training schedules, where the objective is to learn a reliable policy without generating new data. The focus is on making conservative and safe decisions, avoiding overestimation of unseen actions.

5.3.2 Key Components and Modifications

- **Data Preparation and Replay Buffer**: The dataset consists of transitions that represent the athlete's state, the action taken (the type or intensity of training), the resulting reward (performance metric), and the next state after the action. These transitions are stored in a replay buffer, from which mini-batches are sampled during training. In the context of offline reinforcement learning, this replay buffer is static, meaning it contains all the data the agent will learn from, with no new data being generated during the training process.

- **Q-Networks (Critic Networks)**: The CQL algorithm employs two separate Q-networks, referred to as Q_1 and Q_2, to estimate the value of state-action pairs. These networks take the state as input and output the Q-values for each possible action. The architecture of the Q-networks consists of three fully connected layers. Each Q-network is updated during training, and the use of two separate networks helps reduce the overestimation of Q-values, a common issue in traditional Q-learning.

- **Conservative Regularization**: A key modification introduced by CQL is the conservative regularization term, which is added to the Q-network loss function. This term penalizes the Q-networks for assigning high Q-values to actions that are not well-represented in the dataset. By doing

so, the Q-values for out-of-distribution actions are reduced, encouraging the agent to focus on actions that have been observed frequently in the dataset. This conservative regularization helps prevent the agent from learning policies based on unrealistic or risky actions.

- **Target Networks**: Similar to other Q-learning-based algorithms, CQL utilizes target networks to stabilize learning. The target Q-networks, Q_1' and Q_2', are periodically updated using a soft update rule, where the parameters of the target networks are slowly moved towards the parameters of the main Q-networks. This soft update mechanism ensures that the target values used in the training process change more smoothly, reducing instability.

- **Training Process**: During training, the algorithm samples mini-batches of transitions from the replay buffer. For each batch, the current Q-values for the actions taken in the batch are computed by the two Q-networks. The target Q-values are computed using the target networks, ensuring more stable learning. The training loss is calculated as the difference between the predicted Q-values and the target Q-values, with the addition of the conservative regularization term. This loss is then used to update the Q-networks using gradient-based optimization techniques. The soft update of the target networks occurs at fixed intervals during training, gradually aligning the target network weights with the Q-network weights.

- **Policy Learning and Evaluation**: The policy is implicitly learned from the Q-networks in CQL. After training, actions are selected based on the learned Q-values. During evaluation, the trained policy is applied to the test dataset, and the accuracy of the policy is assessed by comparing the predicted actions to the actual actions observed in the test data.

5.3.3 Key Contributions and Impact of CQL

- **Robustness to Out-of-Distribution Actions**: In offline reinforcement learning, it is crucial to avoid overestimation of actions that are not well-represented in the training dataset. CQL achieves this by penalizing high Q-values for such actions, ensuring that the learned policy is more conservative and focused on actions that have been observed frequently in the dataset.

- **Improved Stability in Offline Settings**: The use of conservative regularization and target networks improves the stability of the learning process, especially in offline settings. This reduces the likelihood of the policy overfitting to unrealistic or risky actions and results in a more reliable decision-making process.

- **Applicability to High-Stakes Domains**: The conservative nature of CQL makes it particularly well-suited for high-stakes domains such as healthcare, autonomous driving, and robotics. In these domains, the cost

of making mistakes can be high, and a conservative policy ensures safer and more reliable decisions.

5.4 ML approach

The problem of crafting personalized training schedules can be framed as a supervised learning task, where the goal is to predict the optimal training schedule (i.e., the sequence of actions or decisions) based on the athlete's current condition and past training history. Supervised learning offers a straightforward way to model the decision-making process of coaches, using labeled historical data to train the model.

Data Representation: Each training decision can be viewed as a sample in a supervised learning context, where the input features represent the athlete's physiological state (e.g., fatigue level, performance in prior tests, race results), and the target label corresponds to the optimal training action (e.g., training intensity, rest, specific exercises). Historical data would include states (athlete condition), actions (training decisions made by coaches), and outcomes (performance in races or tests).

5.4.1 DNNs

In alignment with our research objectives, Deep Neural Networks (DNNs) are utilized as a benchmark to evaluate the effectiveness of Reinforcement Learning in modeling the decision-making process of elite sports coaches. In this approach, DNNs are employed to model the complex and nonlinear relationships between athletes' performance and the optimal training decisions. In this set up we address this problem as a supervised learning problem where a model maps directly a training schedule with performance. Again the performance is measured by how well the athlete performed in official measurement days and races. DNNs are well-suited for capturing patterns in high-dimensional data, making them an appropriate choice for this supervised learning task.

Model Architecture:

The DNN model is designed to capture complex, nonlinear relationships between input features and training decisions. The architecture comprises:

- **Input Layer**: The same elements that were used in the RL to describe the state are used as input features
- **Hidden Layers**: Several fully connected layers with ReLU activation functions to capture the nonlinearities in the data. Batch normalization and dropout are employed to enhance generalization and prevent overfitting.

- **Output Layer**: The output layer consists of a single node representing the predicted performance outcome, which could be a continuous variable such as a heart-rate-based reward or a categorical value representing fitness levels.

Training Procedure:

- **Loss Function**: As the reward is measured based on the heart rate, the Mean Squared Error (MSE) loss is applied.
- **Optimization Algorithm**: Adam optimizer is utilized for efficient parameter updates, ensuring stable convergence.
- **Regularization**: Dropout layers and L2 regularization are applied to prevent overfitting, especially given the limited size of the dataset.
- **Training Process**: The model is trained on the historical data, learning to predict the athlete's performance based on their training schedule.

Key Contributions:

- Serves as a **baseline** for evaluating the predictive capacity of Reinforcement Learning approaches.
- Helps in identifying **patterns in training data** that contribute to improved athlete performance, providing insights into the relationships between training load and outcomes.
- Provides a benchmark to assess whether traditional machine learning methods, like DNNs, can match or outperform RL approaches in predicting performance.

5.4.2 Decision Trees

Decision Trees were also used to predict athlete performance outcomes, delivering an interpretable and straightforward model to assess how different training parameters influence performance metrics.

Model Description:

- **Tree Structure**: The tree splits the data based on feature values (such as training duration, type, RPE, and load), progressively narrowing down the possible performance outcomes at each split.
- **Splitting Criteria**: Gini impurity or entropy is used to determine the best splits, ensuring that each branch moves towards homogenous performance outcomes.
- **Tree Pruning**: Maximum tree depth and minimum samples per leaf are controlled to prevent overfitting, especially given the limited size of the historical training data.

Training Procedure:

- **Data Preparation**: Each training session is represented as a feature vector, with the performance outcome as the target variable.
- **Model Fitting**: The tree is trained to split the dataset based on the input features, learning how training schedules affect performance outcomes.
- **Cross-Validation**: K-fold cross-validation is used to ensure the model generalizes well to unseen data.

Key Contributions:

- Acts as a **benchmark** to compare against more complex models, such as DNNs and Reinforcement Learning.
- Provides **interpretability**, allowing for an easy understanding of how different training parameters contribute to athlete performance.
- Highlights the **limitations of simpler models**, showing whether traditional Decision Trees can capture complex patterns in training data.

5.4.3 XGBoost

XGBoost (eXtreme Gradient Boosting) is a high-performance ensemble learning method that is used here to model the relationship between athletes' physiological states and optimal training decisions. XGBoost is particularly adept at handling structured data and capturing complex interactions among features, making it suitable for this supervised learning task. For this task **XGBClassifier** function was used of the **xgboost** library out of the shelf.

Model Description:

- **Ensemble Framework**: XGBoost constructs an ensemble of decision trees sequentially, with each subsequent tree correcting the errors of previous ones. This gradient boosting approach enhances predictive accuracy by modeling residual errors.
- **Regularization**: Incorporates both L1 (Lasso) and L2 (Ridge) regularization to penalize complexity, thereby mitigating overfitting and improving generalization.
- **Handling Missing Data**: Inherently manages missing values by determining optimal splits, ensuring robustness for incomplete datasets.

Training Procedure:

- **Feature Engineering**: Input features mirror those used in the Reinforcement Learning setup, including fatigue levels, prior performance indicators, and other physiological metrics.

- **Data Splitting**: A chronological train-test split is used to respect the temporal nature of data, preventing information leakage. Same as in the previous examples.
- **Model Training**: The XGBoost classifier is trained to optimize multi-class classification accuracy. Hyperparameters such as number of estimators, maximum tree depth, learning rate, subsample ratio, and column sampling ratio are tuned to maximize performance.
- **Evaluation**: Performance is assessed using accuracy on the test set, validating the model's generalization capability.

Key Contributions:

- Provides **interpretability** through feature importance scores, revealing influential variables in training decision-making.
- Offers high **predictive accuracy and computational efficiency**.

6 Results

In this section, the results of the different approached will be presented and briefly discussed. Starting with the Online RL methods, moving to the Offline and finally closing with the traditional ML methods.

6.1 Online RL

Below the in Table 3 we see the parameters used and the respective performance of the Online RL agents. As we see the best-performing agent is SAC with and accuracy of almost 62%.

Online agents	epochs	buffer sampl.	batch_size	gamma	tau	lr Qnet	lr policy	Accur.
DQN	500	rand	32	0.99	0.05	0.001	-	57.93%
DQN	500	seq	32	0.99	0.05		-	51.52%
DDQN	400	rand	32	0.95	0.05	0.001	-	54.57%
DDQN	400	seq	32	0.95	0.05	0.001	-	59.45%
SAC	5000	rand	32	0.99	0.05	0.001	0.0001	**62%**

Table 3: Parameters and performance of the online RL agents

6.1.1 Discussion

Contrary to initial expectations, we observed that more complex models performed better in this task, despite having only a small amount of data and a relatively simplified action space. It is important to note that all experiments were seeded for consistency.

SAC, being considerably more complex than the other models, initially performed poorly. However, hyperparameter optimization was necessary to achieve the observed performance level. On the other hand, the simpler models (DQN and DDQN) were less sensitive to hyperparameter changes and showed minimal improvements when parameters were varied. In the early steps of the training, the performance seemed to fluctuate but in the end, both of the agents seemed to converge.

Another interesting observation was that sequential sampling in the replay buffer led to lower performance for DQN but higher performance for DDQN. This may indicate that more advanced methods like DDQN can better handle correlated data, while DQN relies more heavily on randomized sampling to mitigate overfitting and stabilize learning.

6.2 Offline RL

Below in Table 4 we see the parameters used and the respective performance of the Offline RL agents. As we see the best-performing agent is Conservative DQN with random sampling with an accuracy of almost 79.30% introducing a significant improvement over the online agents.

Offline RL agents	epochs	buffer sampling	BATCH_SIZE	GAMMA	Accuracy
CDQN	100	random	32	0.99	**79.30%**
CDQN	100	sequential	32	0.99	71.23%

Table 4: Parameters used and the respective performance of the Offline RL agents

6.2.1 Discussion

The results demonstrate that conservative DQN trained in an offline setting with random sampling achieved higher accuracy than the "online" methods discussed earlier. This performance difference can be attributed to the inherent characteristics of CDQN, which make it well-suited for offline learning.

In offline settings, the entire dataset is static and predefined, removing the need for exploration. CDQN excels in this scenario because its Q-value updates rely entirely on the data provided, allowing it to efficiently learn policies without the risk of exploration-exploitation trade-offs. Interestingly, similar to the simple DQN the performance of the random sampling seemed to be better compared to the sequential sampling.

Despite the sequential nature of the athlete's training data, the random sampling approach yielded better performance than sequential sampling. This observation can be attributed to several factors.

Firstly, breaking temporal correlations is crucial for effective learning in neural networks. Sequential sampling presents highly correlated data to the model, as consecutive training days for an athlete are likely to have minimal differences. This high correlation can lead to inefficient learning, as the model may make only small updates to its weights, slowing down convergence. Random sampling, on the other hand, breaks these temporal correlations by providing a more diverse set of experiences in each batch. This diversity enhances the model's ability to generalize and stabilizes the training process by preventing the model from overfitting to specific sequences in the data.

Secondly, the data characteristics and model capacity play a significant role. The lack of improved performance with sequential sampling suggests that the sequential dependencies in the data may not be strong or informative enough for the model to leverage. Additionally, the model architecture used in this study may not be designed to capture complex temporal dependencies, especially since it lacks recurrent components like Long Short-Term Memory (LSTM) networks or Gated Recurrent Units (GRUs). As a result, the model might not benefit from sequential data, and random sampling becomes more effective by exposing the model to a wider variety of states and actions.

Lastly, from a theoretical perspective, random sampling aligns better with the assumptions underlying many machine learning algorithms. Specifically, these algorithms often assume that data samples are independently and identically distributed (IID). Sequential sampling violates this assumption due to the inherent dependencies between consecutive samples. Random sampling reduces the bias introduced by the order of data, leading to better generalization. Additionally, it helps balance the bias-variance trade-off by increasing the variance within batches, which can help the model escape local minima and find more optimal solutions.

In summary, the superior performance of random sampling in the CDQN training highlights the importance of considering both the data characteristics and the model architecture when choosing a sampling strategy. Random sampling appears to be more effective in this context due to its ability to break temporal correlations, accommodate the model's capacity, and satisfy theoretical assumptions about data distribution, ultimately leading to improved learning and generalization.

6.3 Traditional ML

Here the results of the traditional ML models will be presented. As we can see in 5 Random Forest is the best-performing model (along with a standard DNN classifier) in predicting future actions based on past data with an accuracy of 85 percent. Second is XGBoost with almost the same score. All models have quite similar performance indicating that we've reached a performance plateau

given our current dataset and features.

ML model	n_estimators	max_depth	min_samples_leaf	lr	Accuracy
RandomForest	200	20	2	-	**85%**
XGBoost	200	5	-	0.01	84%
DNN	-	-	-	0.001	**85%**

Table 5: Parameters used and the respective performance of the Traditional ML models

6.3.1 Discussion

As observed in Table 5, traditional machine learning methods outperformed both online and offline reinforcement learning approaches in terms of predictive accuracy. This result suggests that supervised learning methods, when applied to structured historical training data, can capture and generalize decision-making patterns more effectively than reinforcement learning methods trained in either an online or offline setting.

The most important factor and the big difference between RL and traditional Ml I think it si the limited need for exploration. Since the dataset is fixed and complete, supervised learning models do not suffer from the exploration-exploitation trade-off inherent in RL. While RL must balance discovering new strategies and exploiting known good ones, traditional ML models can directly learn optimal mappings without the need for exploration. And just to be fair, the RL agents did not even get the chance to explore.

Although reinforcement learning methods—especially offline RL—show promise in modeling training schedules, they still lag behind traditional supervised learning approaches in predictive accuracy. However, RL methods may provide additional benefits in dynamic environments where new data can be collected, or when generalization beyond the dataset is required.

7 Discussion

In this RL setting, the state consists of some attributes that describe the fitness of the athlete. The duration, the RPE, and other attributes that are included in the state representation have something to tell about how fit a person is. Each action, representing a training session, transitions the athlete to a new state—either improved or deteriorated. What we are interested in, is predicting those actions-trainings that will bring the specific athlete to the best possible state-fitness level. However, to know what is good and what is bad we need the reward to penalize bad trainings and reinforce good ones. Fortunately, we have access to rewards derived from official measurements and race results, which serve as ground truth indicators of the athlete's actual performance level.

In an online setting, the athlete would train, collect rewards based on the training outcomes, and adjust the policy accordingly through continuous learning. However, this research is confined to past data; our environment is effectively the dataset itself. The dataset becomes a surrogate for the environment, and the model's performance is constrained by the diversity and qrepresentativeness of this dataset. Additionally, as we displayed multiple times in the methods, there is no notion of exploration, the most important tool of an RL agent to learn and generalize. Now, let's say that we have the perfect agent trained on this dataset and ready to predict the best actions according to the dataset. As we don't have the luxury of using an actual person to perform the suggested actions how can we really evaluate this agent? The only meaningful thing that is left to measure is the accuracy of the predicted actions versus the actions that were actually taken.

To do this, we have to use a part of the dataset, unseen to the agent, to evaluate. Now let's say that this perfect model scored an accuracy of 100 %. That's perfect, right? As good as it gets! However, someone might be surprised if we told them that an agent that scores 40 % could actually be way better than the one that scored 100? What do these percentages tell us? Since we are predicting actions and evaluating based on historical actions there is no reassurance that the target actions are optimal. In that sense, we can not really tell if the agent is better or worse than the coach, we can only tell at what percentage it mimics the coach's behavior.

The only definitive way to understand if the agent has genuinely learned effective strategies would be to test with a real athlete who physically performs the trainings suggested by the agent. Given that an individual's fitness is influenced by numerous factors such as age, climate, and personal physiology, the ideal benchmark for evaluating the four RL algorithms would involve having identical athletes follow the suggested actions from each of the four agents and the coach. Then, after a couple of months of training, run some evaluation tests and finally decide who is really better. The coach or one of our trained agents. However, this scenario is, of course, not feasible.

But why the traditional ML algorithms are better at mimicking and could potentially be a better fit for this problem? The answer about the fitness of ML algorithms is "no" they can't. As they were trained based on the actions that the specific coach instructed and the reward is not somehow utilized to give a realistic ground truth we can clearly see that these algorithms could never break the pattern and do better than the coach. Now let's discuss the increased performance of these algorithms in replicating the coach's training schedule focusing on Random Forest which scored the highest performance percentage. These reasons can be summarized as such:

- Firstly, the problem aligns more naturally with supervised learning, where

the goal is to predict historical actions based on labeled data. Random Forests excels at modeling direct mappings from input features (states) to target outputs (actions), optimizing for predictive accuracy without the complexities of RL's reward structures.

- Random Forests handle static datasets effectively, avoiding challenges inherent in offline RL, such as distributional shifts and extrapolation errors. They are less prone to overfitting in low-data regimes and require fewer hyperparameters, making them easier to train and tune.
- RL is generally built to work with plenty of data and have ample trial and feedback sessions before it starts learning meaningful policies. This small dataset might be enough for a simple RF algorithm but for the complex RL agents this is an overkill.
- Finally, the evaluation metric—accuracy in predicting historical actions—perfectly aligns with the objective of supervised learning but not necessarily with that of RL, which seeks to discover optimal policies that may deviate from historical actions. RL's goal and objective was never to mimic the coach.

This means that, in scenarios where the objective is to replicate or predict past actions from existing data, traditional machine learning approaches like Random Forests are more suitable and yield higher accuracy than RL algorithms. However, if the goal is to learn optimal policies that through maximizing the rewards can craft optimal personalized training schedules then probably RL is the way to go.

In conclusion, while our models aim to predict optimal training actions based on historical data, limitations arise due to the inability to validate these predictions in a real-world setting. The reliance on past actions as a benchmark means that high accuracy in mimicking historical actions does not necessarily equate to optimal performance. Future work could explore alternative evaluation metrics or incorporate simulated environments to better assess the agent's effectiveness in enhancing athletic performance.

8 Conclusion

This research investigated the applicability and efficacy of Reinforcement Learning (RL) methods in modeling and optimizing the decision-making process involved in crafting personalized training schedules for elite athletes, specifically focusing on Professional Speed Skating. The primary goal was to determine how historical training and performance data, combined with RL algorithms such as Deep Q-Network (DQN), Double Deep Q-Network (DDQN), and Conservative Q-Learning (CQL), could replicate or potentially enhance the existing coaching strategies. Complementarily, the study also evaluated the performance of these RL approaches against conventional Machine Learning (ML) methods, specifically Random Forests and deep neural networks.

The exploration began by establishing a robust RL framework tailored explicitly to the elite sports coaching context, formulating the scheduling problem as a Markov Decision Process (MDP). States were constructed to reflect athlete physiological status, recent training load, perceived exertion (RPE), and time-related contextual factors. Actions represented the specific types of training interventions, and rewards were grounded in objective athlete performance metrics derived from race outcomes and physical tests. This careful formulation ensured alignment with realistic coaching practices and physiological probability.

Empirical results demonstrated that RL models could indeed capture substantial aspects of the decision-making patterns exhibited by professional speed skating coaches. In particular, the RL approach successfully identified and mimicked complex scheduling and decision-making trade-offs, suggesting the potential viability of RL-based systems to complement traditional coaching practices. However, the practical performance of the RL agents was limited by the inherent limitations of the historical data set: mainly its limited scope, sparse performance feedback, and the absence of active exploration opportunities. As a consequence, while RL models achieved respectable accuracy (approximately 79% with Conservative DQN), they fell short of surpassing the performance of simpler ML models, which achieved accuracy up to 85%.

Below, we revisit and answer each of the research questions explicitly, integrating the findings and insights from our empirical analysis:

RQ1: How can Reinforcement Learning be effectively applied to model the decision-making process of elite sports coaches in generating personalized training schedules for athletes in Professional Speed Skating?

The study confirmed that the efficacy of RL in modeling elite sports coaching relies heavily upon the precise definition of states, actions, and rewards within the MDP framework. The chosen formulation included athlete physio-

logical data (duration, RPE), temporal elements, and carefully designed reward signals reflecting performance outcomes. This representation allowed RL models to effectively mimic real-world coaching behaviors, demonstrating a solid foundation for sequential decision-making modeling. By aligning the MDP with the actual details and constraints faced by coaches, RL agents learned realistic and meaningful scheduling policies from historical data.

RQ2: To what extent can a Reinforcement Learning model effectively incorporate and adapt to limited feedback from races and physical tests to iteratively refine training schedules, thereby enhancing their effectiveness and aligning with individual athlete performance outcomes?

Our RL models successfully incorporated limited and sparse feedback to iteratively adjust training schedules. For instance, following suboptimal performance indicators, the agents consistently adjusted training intensities or volumes appropriately, reflecting real-world coaching responses. However, this adaptive capability had clear limitations; due to infrequent and sparse feedback points, the scope of iterative refinement remained restricted. The RL agents managed incremental improvements and meaningful alignment with individual performance, but we were unable to evaluate or demonstrate extensive or transformative schedule optimization. Thus, while RL can effectively integrate sparse feedback, substantial iterative refinement demands richer, more frequent feedback signals or augmented data sources, and concerning evaluation, an online setup is required.

RQ3: How effectively can RL methods—originally designed for online exploration—be adapted for offline settings with historical data, and how do these adaptations compare in performance and stability to dedicated offline RL algorithms under conditions of sparse rewards and limited exploration opportunities?

Now, the important question becomes: can these adapted "online" agents still perform? The answer is: yes, but not great. Take SAC for example. When we removed the entropy and allowed it to only learn from the existing dataset, it still managed to reach around 62% accuracy. Respectable, but far from optimal. Enter Conservative Q-Learning (CQL)—a method built from the ground up for offline learning. CQL incorporates regularization to avoid overestimating unseen actions and directly addresses the core risks of offline training. It outperformed the rest with almost 79.3% accuracy.

So what's the takeaway? Trying to force online RL into an offline mold kind of works, but only up to a point. It gave us useful comparisons, showed us how much performance comes from exploration, and highlighted the structural limitations of certain architectures. But at the end of the day, offline-specific algorithms like CQL are better suited for this problem. They're designed to

be cautious, to handle limited and potentially biased data, and they do this without trying to reach outside the dataset.

In the broader context, this experiment confirmed that the problem formulation was valid and instructive. It allowed us to observe meaningful behaviors and compare methodological choices. It also confirmed that while adapting online agents is a valid research direction, performance-wise and stability-wise, specialized offline RL methods are currently the way to go when you're locked into historical data.

RQ4: Can Reinforcement Learning (RL) outperform standard Machine Learning techniques when the dataset is limited? Do RL agents generalize well when historical data lacks comprehensive exploration coverage, and if not, do they effectively learn to mimic real-world scenarios where the learned policy would be applied?

Based on the results, the answer here is pretty straightforward: RL didn't beat traditional ML when it came to predictive accuracy. Random Forests and DNNs scored up to 85%, while even the best RL agent, CQL, topped out at about 79%. That's a decent score, but not enough to claim superiority. What's interesting is that the RL agents weren't totally off—they did manage to mimic historical decision patterns and make reasonable predictions. But that's exactly the point: they learned to imitate, not innovate.

Why? Because the dataset itself didn't give them the space to explore or try something different. When your data is fixed, and your reward signal is sparse, it's hard for an agent to figure out what "better" even looks like. It just sticks to what it sees, which means it ends up reproducing the same decisions as the coach who generated the data. So yes, RL can learn to generalize within what it sees, but not beyond. Or even if it does how could we tell? The agents might indeed learned underlying features and dynamics of the athlete and the sport but there is not a feasible way to evaluate this. The only thing we can do is evaluate the agent for the actions it produced and how well they copy the coach instead of evaluating with real "rewards" coming from the athlete performance. Who knows? If the athlete really did the actions "training schedule" that the model was suggesting the reward could be superior.

This makes traditional ML the more natural fit for this task—at least for now. It doesn't try to explore or optimize rewards over time. It just learns the mapping from input to output, and in a fixed-data setup, that's actually all you need. RL might shine later, when we have richer datasets, better feedback mechanisms, or simulated environments where exploration becomes safe and meaningful. Until then, it's good to see that RL holds promise—but it's not quite ready to take over yet.

Reflection and Future Directions

This research highlights both the potential and constraints of applying RL to personalized elite athlete training. While RL's conceptual advantages for adaptive policy learning remain appealing, practical limitations—particularly data scarcity, infrequent performance feedback, and exploration constraints—currently hinder its full efficacy. Traditional ML methods, being inherently less sensitive to these constraints, proved more effective within our specific data context.

Future research should explore the integration of more frequent, detailed performance measurements, simulated athlete responses, or expert-informed synthetic data augmentation to enable safer and more extensive RL exploration. With these enhancements, RL's ability to identify innovative and optimized training schedules could be effectively unlocked, potentially surpassing traditional ML approaches. Therefore, this thesis provides a foundation and a clear pathway toward future improvements in employing RL for more sophisticated and optimized training schedule generation in elite sports.

References

Bellman, R. (1957). A markovian decision process. *Journal of Mathematics and Mechanics*, *6*(5), 679–684. Retrieved August 14, 2024, from http://www.jstor.org/stable/24900506

Bourdon, P. C., et al. (2017). Monitoring athlete training loads: Consensus statement. *International Journal of Sports Physiology and Performance*, *12*(s2), S2–161.

Breiman, L. (2001). Random forests. *Machine learning*, *45*, 5–32.

Brown, G. A., Veith, S., Sampson, J. A., Whalan, M., & Fullagar, H. H. (2020). Influence of training schedules on objective measures of sleep in adolescent academy football players. *Journal of Strength and Conditioning Research*, *34*(9), 2515–2521. https://doi.org/10.1519/JSC.0000000000003724

Cardinale, M., & Varley, M. (2017). Wearable training-monitoring technology: Applications, challenges, and opportunities. *International Journal of Sports Physiology and Performance*, *12*(s2), S2-55-S2–62. https://doi.org/10.1123/ijspp.2016-0423

Cortes, C. (1995). Support-vector networks. *Machine Learning*.

Demosthenous, G., Kyriakou, M., & Vassiliades, V. (2022). Deep reinforcement learning for improving competitive cycling performance. *Expert Systems with Applications*, *203*, 117311. https://doi.org/10.1016/j.eswa.2022.117311

Deng, Y., Bao, F., Kong, Y., Ren, Z., & Dai, Q. (2016). Deep direct reinforcement learning for financial signal representation and trading. *IEEE transactions on neural networks and learning systems*, *28*(3), 653–664.

Forndran, A., et al. (2012). Training schedules in elite swimmers: No time to rest. In *Sleep of different populations* (pp. 6–10).

Freund, Y., Schapire, R. E., et al. (1996). Experiments with a new boosting algorithm. *icml*, *96*, 148–156.

Ge, Y., Zhu, F., Ling, X., & Liu, Q. (2019). Safe q-learning method based on constrained markov decision processes. *IEEE Access*, *7*, 165007–165017. https://doi.org/10.1109/ACCESS.2019.2952651

Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *International conference on machine learning*, 1861–1870.

Kormushev, P., Calinon, S., & Caldwell, D. G. (2013). Reinforcement learning in robotics: Applications and real-world challenges. *Robotics*, *2*(3), 122–148.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, *25*.

Kumar, A., Zhou, A., Tucker, G., & Levine, S. (2020a). Conservative q-learning for offline reinforcement learning. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, & H. Lin (Eds.), *Advances in neural information processing systems* (pp. 1179–1191, Vol. 33). Curran Associates,

Inc. https://proceedings.neurips.cc/paper_files/paper/2020/file/0d2b2061826a5df3221116a5085a6052-Paper.pdf

Kumar, A., Zhou, A., Tucker, G., & Levine, S. (2020b). Conservative q-learning for offline reinforcement learning. *Advances in Neural Information Processing Systems*, *33*, 1179–1191.

Lange, S., Gabel, T., & Riedmiller, M. (2012). Batch reinforcement learning. In M. Wiering & M. van Otterlo (Eds.), *Reinforcement learning: State-of-the-art* (pp. 45–73). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-27645-3_2

Maei, H. R., Szepesvári, C., Bhatnagar, S., & Sutton, R. S. (2010). Toward off-policy learning control with function approximation. *ICML*, *10*, 719–726.

Mao, H., Alizadeh, M., Menache, I., & Kandula, S. (2016). Resource management with deep reinforcement learning. *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, 50–56. https://doi.org/10.1145/3005745.3005750

Minsky, M., & Papert, S. (1969). An introduction to computational geometry. *Cambridge tiass., HIT*, *479*(480), 104.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mulani, J., Heda, S., Tumdi, K., Patel, J., Chhinkaniwala, H., & Patel, J. (2020). Deep reinforcement learning based personalized health recommendations. In S. Dash, B. R. Acharya, M. Mittal, A. Abraham, & A. Kelemen (Eds.), *Deep learning techniques for biomedical and health informatics* (pp. 231–255). Springer International Publishing. https://doi.org/10.1007/978-3-030-33966-1_12

Nash, C., & Collins, D. (2006). Tacit knowledge in expert coaching: Science or art? *Quest*, *58*(4), 465–477. https://doi.org/10.1080/00336297.2006.10491894

Plaat, A. (2022). Deep reinforcement learning. *CoRR*, *abs/2201.02135*. https://arxiv.org/abs/2201.02135

Qin, H., Qian, S., Cai, X., & Guo, D. (2024). Athletic skill assessment and personalized training programming for athletes based on machine learning. *Journal of Electrical Systems*, *20*(9), 1379–1387.

Quinlan, J. R. (1987). Generating production rules from decision trees. *ijcai*, *87*, 304–307.

Raab, M. (2012). Simple heuristics in sports. *International Review of Sport and Exercise Psychology*, *5*(2), 104–120. https://doi.org/10.1080/1750984X.2012.654810

Raiola, G., & Tafuri, D. (2015). Teaching method of physical education and sports by prescriptive or heuristic learning. *Journal of Human Sport and Exercise*, *10*(1), S377–S384.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, *65*(6), 386.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, *323*(6088), 533–536.

Rummery, G. A., & Niranjan, M. (1994). *On-line q-learning using connectionist systems* (tech. rep.). University of Cambridge, Department of Engineering.

Singh, S., Jaakkola, T., Littman, M. L., & Szepesvári, C. (2000). Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, *38*(3), 287–308. https://doi.org/10.1023/A:1007678930559

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. A Bradford Book.

Thomas, G., Gade, R., Moeslund, T. B., Carr, P., & Hilton, A. (2021). Computer vision for sports: Current applications and research topics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, *59*(236), 433–460. Retrieved October 13, 2024, from http://www.jstor.org/stable/2251299

Van Eetvelde, H., Mendonça, L., & Ley, C. (2021). Machine learning methods in sport injury prediction and prevention: A systematic review. *J EXP ORTOP*, *8*(27). https://doi.org/10.1186/s40634-021-00346-x

Vaswani, A. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*.

Wackerhage, H., & Schoenfeld, B. (2021). Personalized, evidence-informed training plans and exercise prescriptions for performance, fitness and health. *Sports Med*, *51*(9), 1805–1813. https://doi.org/10.1007/s40279-021-01495-w

Watkins, C. J. C. H. (1989). *Learning from delayed rewards* [Doctoral dissertation, King's College, Cambridge].

Yu, C., Liu, J., Nemati, S., & Yin, G. (2021). Reinforcement learning in healthcare: A survey. *ACM Computing Surveys (CSUR)*, *55*(1), 1–36.