Optimal Pacing Strategies in Cycling with Reinforcement Learning

Bryce Davidson, Patrick Dann, Logan Shinde

Introduction

Reinforcement Learning Overview

The goal of this project is to explore the double Q-learning algorithm, as cited by Van Hasselt et al. [2015], as a method for solving Markov decision processes. Our objective is to optimize a cyclist's power output for a race, as inspired by Anna Kiesenhofer's mathematical approach, which won her the 2021 Tokyo Olympics women's road race. We use the concepts of critical power (CP) and Anaerobic Work Capacity (AWC or W') as our physiological constraints and simple physics equations derived from Newton's Second Law of Motion as our mechanical constraints.

Reinforcement learning is a type of machine learning that models a learning algorithm as an agent. This agent seeks to learn about an environment by taking actions based on its current state, receiving a new state and a reward from that action, and then repeating this process to generate the training data necessary to find a policy for the agent which maximizes its expected cumulative discounted reward over all states. Solving for the maximum discounted reward over all states is commonly referred to as solving the Bellmen equation Sutton and Barto [2018]. In the context of our study, the agent is represented by a cyclist, while the environment encompasses the physiological and mechanical constraints imposed on that cyclist.

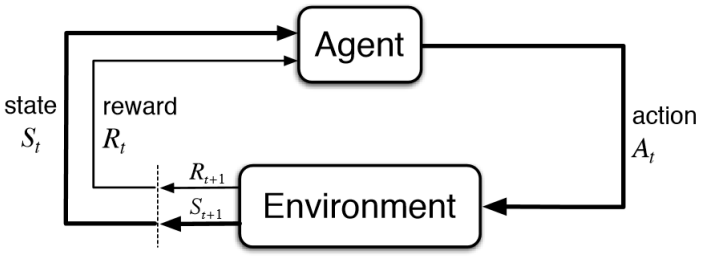

Figure 1: Description of the Reward Hypothesis taken from Sutton and Barto [2018].

Methods and Materials

Markov Decision Processes and the Bellman Equation

From a mathematical perspective, MDPs are discrete-time stochastic processes. They are commonly used to model decision-making in which outcomes are influenced by the dynamics of the environment and the decisions made by an agent. As described by Miller [2022], the most common implementation of an MDP is that of Discounted-Reward Markov Decision Process, which can be defined as a 6-tuple $(S, s_0, A, P, r, \gamma)$ with the following elements,

- **State space (S)**: The set of all possible states.
- **Initial state (s_0)**: An initial state in S.
- **Action set (A)**: The set of actions for which $A(s) \subseteq A$ are the applicable actions our agent can take given it is at state $s \in S$.
- **Transition probability (P)**: The transition probability $P_a(s' \mid s)$ for $s \in S$ and $a \in A(s)$.
- **Reward (r)**: A positive or negative reward $r(s, a, s')$ given as a result of transitioning from state s to state s' using action a.
- **Discount factor (γ)**: A discount factor taking values in the set $[0, 1)$ that determines how much a future reward should be discounted compared to a current reward.

The objective of reinforcement algorithms is to take an MDP as input, and produce an optimal policy function π_* as output, which tells an agent the best action to choose in any given state in order to maximise the expected discounted cumulative reward from that state [Miller, 2022]. The expected value of following a policy π from state s, can be represented as a function $V^\pi(s)$ defined as follows,

$$V^\pi(s) = E_\pi[\sum_i \gamma^i r(s_i, a_i, s_{i+1}) \mid s_0 = s, a_i = \pi(s)]$$

In order to ensure the policy π is the optimal policy π_*, the function $V(s)$ must satisfy the Bellman Equation.

$$V(s) = \max_{a \in A(s)} \sum_{s' \in S} P_a(s' \mid s)[r(s, a, s') + \gamma V(s')]$$

Subsequently, the optimal policy can be extracted by choosing actions which satisfy the following,

$$\pi_*(s) = argmax_{a \in A(s)} \sum_{s' \in S} P_a(s' \mid s)[r(s, a, s') + \gamma V(s')]$$

We can simplify this expression by defining a function which represents the value of choosing action a in state s and then following the policy π Miller [2022].

$$Q(s,a) = \sum_{s' \in S} P_a(s' \mid s)[r(s,a,s') + \gamma V(s')]$$

The optimal policy can then be reformulated as the following which will be the foundation of the Q-learning method,

$$\pi_*(s) = argmax_{a \in A(s)} Q(s,a)$$

Building the Environment

The environment was built around the equations and constants referenced in the appendix [A.2, A.3, A.4, A.5, A.6], to mimic the physiological constraints due to fatigue and recovery, and mechanical constraints due to Newton's Laws of Motion. Additionally, We assumed the rider to have a max AWC of 9758 J, a Critical power of 234 W and mass of 70kg.

Following the implementation of the environment, our next challenge was to devise a reward function that would guide our agent's learning process. Given that our agent's objective is to complete the race as swiftly as possible, we initially set a negative existence reward for each step taken, coupled with a substantial positive reward upon course completion. Furthermore, we aimed to prevent scenarios where the velocity drops below zero, as such situations are undesirable in any racing context. Therefore, we assigned a significant negative reward whenever the current state exhibited a velocity less than zero.

The next step was to train the agent we used the Double Deep Q Networks method which is described in the following section.

Double Deep Q-Networks

Double Deep Q-Networks (DDQN) is an extension of Deep Q-Networks (DQN) that addresses the issue of overestimation bias in the Q-values. This is a significant improvement as the overestimation of bias can lead to sub-optimal policies Van Hasselt et al. [2015]. DDQN uses an online neural network with weights θ to approximate the Q-function, as well as a target network with weights θ^- to maintain the temporal difference target for a certain number of time steps, where the temporal difference target is defined as

$$Y_t = R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta)$$

The update rule for the weights of the online network, θ is as follows

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \boldsymbol{\theta_t}); \theta_t^-) - Q(S_t, a_t; \boldsymbol{\theta}_t) \right] \nabla_{\boldsymbol{\theta_t}} Q(S_{t+1}, a_t; \boldsymbol{\theta_t})$$

Our Implementation

Our model was set up for the agent to have an action of applying a percentage of the available max power at any second. The action space was an integer between 0-10, where each integer represented a 10 percent increment; 10 was 100 percent and 0 was zero percent. The state space comprised of the agent's current velocity, percentage complete, AWC, available power, and gradient of the course. The Q function took into account the state action pair and in our case, the Q function considered the current velocity, percent complete, AWC, available power, and gradient of your agent, and it outputted a percentage of the maximum available power that the agent should apply. In our case, the Q-functions goal was to find the best balance between speed and efficiency so that the agent could complete the task as fast as possible.

We initially created a 10km course with repeating 1km hills at a 15 percent gradient [A.7]. This course was used to run simulations to test our environment. We found that a rider set to 100 percent of available power output (10-rider) completed the course in 2279s, while a rider set to 100 percent on the uphill and 80 percent on the downhill (10/8-rider) completed the course in 2260s. A comparison of the two simulations can be seen in the appendix A.8. This was used as proof of concept that varying power output as a pacing strategy can outperform a maximal power output strategy.

We then created a new 400m testing course to speed up the training time for the DDQN model. We chose this specific course as it provided a significant difference in time between the 10-rider and 10/8-rider, indicating that pacing strategies were effective even on the short course. A profile of the course can be seen the the figure below 2.

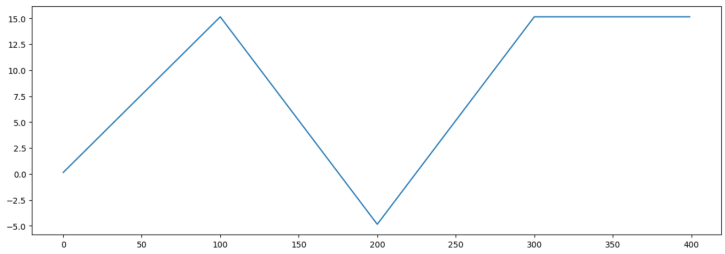

Figure 2: Test course profile.

We ran into the issue of sparse rewards due to the nature of our problem and the structure of the reward function. Sparse rewards arise when the agent does not encounter rewards frequently enough to associate actions with the rewards, Eschmann [2021]. This ultimately, results in inefficiencies in the learning process and prevents the agent from learning the optimal policy. In an attempt to overcome the sparse rewards issue, we implemented the reward-shaping strategy.

For the reward function, we implemented a negative existence reward for each step (-1 per second not finished), a large positive reward for completing the course, and a large negative reward if the current state had a velocity of less than zero shown in Figure 3. Additionally, We implemented some reward shaping to penalize unfavourable states and reward favourable states. To incentivise the agent to conserve energy on the downhill and expend energy on the uphill a negative reward equal to 10 percent of the current AWC on uphill sections and a positive reward equal to 10 percent

of the current AWC on downhills was implemented.

```python
def reward_fn(state):
    (
        agent_power_max_w,
        agent_velocity,
        agent_gradient,
        agent_percent_complete,
        AWC,
    ) = state

    reward = -1

    if agent_gradient >= 0:
        reward -= AWC * 0.1

    if agent_gradient < 0:
        reward += AWC * 0.1

    if agent_velocity < 0:
        reward -= 100

    if agent_percent_complete >= 1:
        return 1000

    return reward
```

Figure 3: Reward function used to guide our agent during training

Hyper Parameters

We found that the hyper-parameters of the DDQN model drastically affected the training process and required a lot of trial and error. Ultimately, we found the best success with the hyper-parameter settings found in Figure 4.

```python
# Create the agent
agent = DDQN(
    input_dims=len(env.state),
    output_dims=env.action_space + 1,
    dense_layers=[24, 24, 24],
    dropout=0,
    gamma=0.9,
    epsilon_start=1,
    epsilon_decay=0.9995,
    epsilon_min=0.01,
    target_life=10,
    memory_size=10000,
    batch_size=128,
    lr_start=0.00001,
    lr_decay=1,
    lr_decay_steps=0,
)
```

Figure 4: Hyper Parameters used for final training run.

In Figure 4, the "dense layers" parameter defines the architecture of the neural network. In this case, we have a 3-layer fully connected network each containing 24 nodes. The number of networks

and nodes can influence the model's capacity to represent more complex functions. The number of nodes dictates the bias-variance trade-off as a higher amount of nodes may lead to over-fitting (high complexity model) while a lower amount leads to under-fitting (low complexity model).

The "gamma" or γ parameter is the discount factor that is used in Double Q-learning and it determines how important future rewards are valued (scale of 0 to 1). In our case, A γ value of 0.9 puts a relatively high value on long-term rewards which is important for our model to finish the course.

The "epsilon start", "epsilon decay", and "epsilon min" refer to the epsilon-greedy strategy or ϵ-greedy. The ϵ-greedy value dictates the randomness or how often the agent decides between exploration and exploitation of state-action pairs. In our case, the agent will explore the environment completely randomly at first and then quickly change to maximize the expected Q-value for the state-action pairs.

The "target life" dictates the number of training runs the target network weights remain stationary before being overwritten by the online model weight.

The learning rate α parameter controls the magnitude at which the weights are updated in response to the estimated error. The equation below shows the expression of how the parameters γ and α influence how the Q-value weights are updated at each step.

The weights are updated by the following equation:

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\text{argmax}} Q(S_{t+1}, a; \boldsymbol{\theta_t}); \theta_t^-) - Q(S_t, a_t; \boldsymbol{\theta}_t) \right] \nabla_{\boldsymbol{\theta_t}} Q(S_{t+1}, a_t; \boldsymbol{\theta_t})$$

Where θ_{t+1} represents the updated weight of the Q-function at time $t+1$ and θ_t are the previous weights of the Q-function. α is the learning rate, R_{t+1} is the reward received by taking action a_t at state S_t to S_{t+1}, γ is the discount factor, and $Q(S_{t+1}, \underset{a}{\text{argmax}} Q(S_{t+1}, a; \boldsymbol{\theta_t}); \theta_t^-)$ is the target Q-value using the target network weights θ_t^- . We see that α factors the magnitude of the update and γ factors the importance of future reward.

Physiology of Cycling Overview

Critical power (CP) represents the highest average power you can sustain over a long period of time [Poole et al., 2016]. It represents the threshold power between the anaerobic system and the aerobic system. The aerobic system can generate power for long periods by replenishing energy through the use of oxygen, while the anaerobic system uses finite energy stored in muscles to produce more power than the aerobic system.

Anaerobic work capacity (AWC), also known as W', represents the size of your anaerobic energy system. The larger this tank, the more power you can release above your CP level [Poole et al., 2016]. This is especially important during short, intensive efforts, ie sprinting up a hill. The relationship between pacing, CP, and AWC is intricate. Pacing strategies in time trials often involve working near your CP threshold, while your AWC determines how much power you can release above your CP level. Therefore, understanding and optimizing these three aspects can significantly improve your cycling performance in time trials.

Mechanical and Physiological Constraints

The mechanical constraints due to physics are derived similarly as in Feng et al. [2022] and Boswell [2012]. Where we denote velocity as a function of time $v(t)$, the gradient or slope of the road as $\theta(t)$, the power of the rider as $P(t)$, and the mass of the rider and bicycle as M and m respectively.

The following mechanical constraints were built into our environment to mimic the effects of gravity, slope, rolling resistance and wind resistance on our agent to better generalize to the real world.

The power required to overcome gravity is shown in Boswell [2012] to be:

$$\Delta_{\text{gravity}} = (M+m)g\sin(\theta(x))v$$

Where g is the acceleration due to gravity, and is assumed to be a constant $9.81\,\text{m/s}^2$.

The power required to overcome wind resistance is shown in Boswell [2012] to be:

$$\Delta_{\text{wind}} = A(v - w\cos(\varphi(t)))^2 v$$

Where A is the drag coefficient based on the frontal area of the cyclist and w is the wind speed, both of which are assumed to be constant. $\varphi(t)$ is the angle that which the wind is pushing the rider, which we also assume to be constant and head-on. For the sake of simplicity, we will assume the wind to be negligible and equal to 0 for our environment.

The power required to overcome rolling resistance is shown in Boswell [2012] to be:

$$\Delta_{\text{rolling}} = (M+m)gR(x)v$$

Here $R(x)$ represents the rolling resistance from the tires and the road surface. We assume $R(x)$ to be constant and equal to 0.0046 which is in line with Boswell [2012].

To generate a velocity from applied power we take into account the effects of the physics equations above in the equation below:

$$\frac{d}{dt}\left(\frac{1}{2}(M+m)v^2\right) = P_{\text{rider}} - \Delta_{\text{gravity}} - \Delta_{\text{wind}} - \Delta_{\text{rolling}}$$

And, assuming mass is constant:

$$(M+m)v\frac{dv}{dt} = P_{\text{rider}} - [\sin(\theta(x)) + R(x)](M+m)gv - A(v - w\cos(\varphi))^2 v$$

And this differential equation is assumed to have initial conditions $x(0) = 0$, $v(0) = 0$.

Much like Ashtiani et al. [2019], which uses Dynamic programming to optimize power. We discretize the above equation gives us a formula for calculating velocity over each interval to allow the function to be implemented in our RL environment:

$$V_i = \left[\left(P_{\text{rider}} - \Delta_{\text{gravity}} - \Delta_{\text{wind}} - \Delta_{\text{rolling}} - \frac{1}{2}(M+m)v^2\right)\left(\frac{2}{M+m}\right)\right]^{1/2}$$

For the physical constraints of our model, We want equations that show W increasing when working below CP and W decreasing when working above CP. Our equations are similar to the ones used in Feng et al. [2022] which are derived in Ashtiani et al. [2019] on experimental data. The equations used are shown below:

$$\frac{dW}{dt} = \begin{cases} -(P_{\text{rider}} - CP) & \text{for } P_{\text{rider}} \geq CP \\ -(0.0879P_{\text{rider}} + 204.5 - CP) & \text{for } P_{\text{rider}} < CP \end{cases}$$

Here 0.0879 and 204.5 are experimentally determined constants to match realistic human results. These constants were determined by fitting adjusted recovery power vs. actual applied power during interval tests found in Ashtiani et al. [2019]. Where the subject was required to perform repeated bouts of high-intensity intervals. The adjusted recovery power (power level sustained during recovery) was then plotted against the actual applied power (corresponding all-out power level after the recovery period) to generate a linear relationship.

The available max power output as a function of the current AWC level (W), is modelled in the following equation from Ashtiani et al. [2019].

$$\mathrm{P_{max}} = 7 \times 10^{-6} \times \mathrm{AWC}^2 + 0.0023 \times \mathrm{AWC} + \mathrm{CP}$$

Results

We were able to train a DDQN model that outperformed the 10/8-rider on the test course by one second and the 10-rider by 11 seconds, shown in Figure 6. We can see the strategy that is the fastest run during training of the final model in Figure 5.

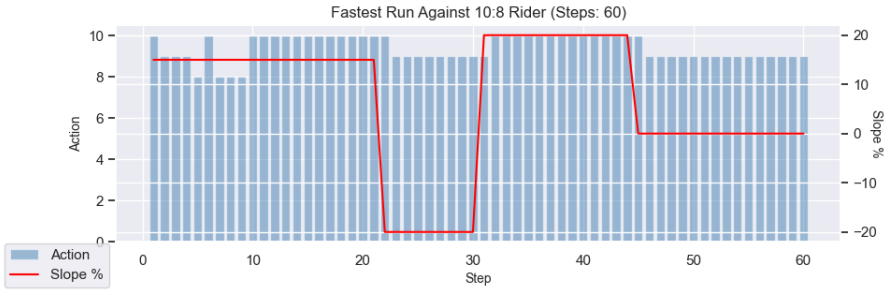

Figure 5: Action space of a run that beat 10/8-rider. The action represents the percent of maximum power applied by the agent at a given state and timestep, shown in A.3.

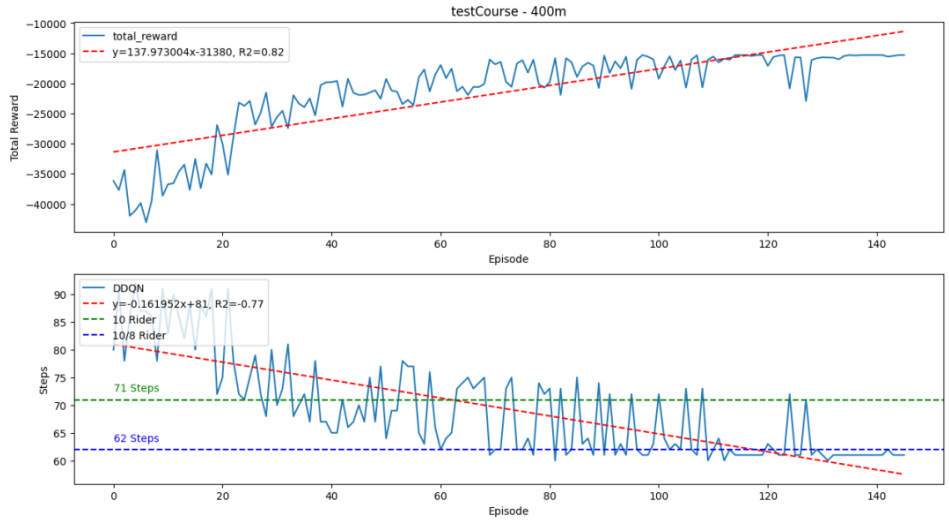

Figure 6: Training of our final model. The number of steps represents the number of seconds required to finish the course on a particular run and the episodes are training runs of the course.

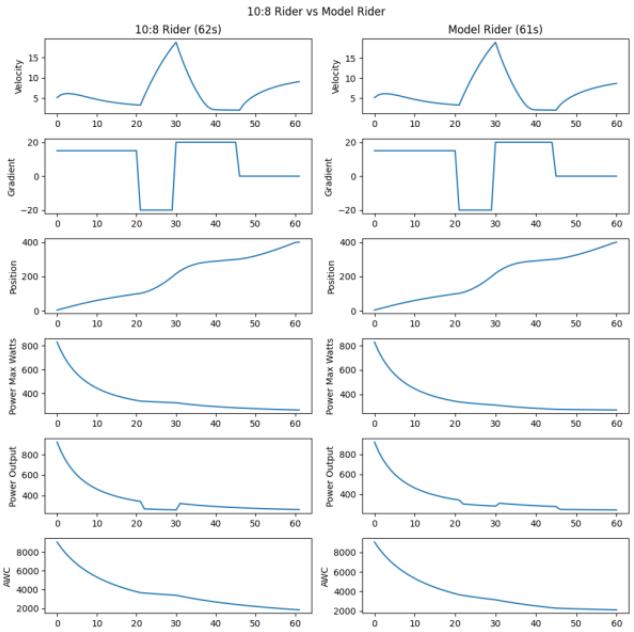

Figure 7: Comparison of 10/8-rider against model rider on the test course.

In Figure 7 we see the comparison of pacing strategies employed by the 10/8-rider compared to the model rider. Notably, the model rider conserves more energy at the beginning of the race and

holds higher power on the downhill section compared to the 10/8-rider. This ultimately resulted in a time that was 1 second faster over the 400m test course.

Discussion

After a significant amount of hyper-parameter tuning, we were able to train a DDQN model that outperformed the 10-rider (maximal output) and 10/8-rider (pacing strategy) simulations on a short test course. We noticed that our model was extremely sensitive to hyper-parameter settings. We found that the hyper-parameters used in Figure 4 had success when training the model.

Notably, a 3-layer network with 24 nodes was effective for our problem. This suggests that it captured a sufficient amount of complexity to learn to approximate the Q-values effectively. Given that our action space was relatively small (11 discrete steps) this is a reasonable interpretation. We saw that a training run with a more complex neural network (4 layers and 200 nodes) did not perform better during training compared to our simpler network mentioned above (results shown in Figures A.9 and A.10 of the appendix).

The generalizability of our DDQN model can potentially be improved by exposing it to more diverse training scenarios. The training course, with its limited features - one uphill, one downhill, and a flat section - may not provide the model with sufficient variety to learn a robust policy. By introducing courses with more features and complexities, the model would encounter a wider range of situations. This could help it learn to adapt its policy to different environments, thereby enhancing its ability to generalize from its training to novel scenarios. However, it's important to note that increasing the complexity of the training environment could also make the learning process more challenging and may require adjustments to the training parameters or algorithm. Thus, the simple test course was sufficient for purposes as a proof of concept.

Our DDQN implementation showed strength in the mitigation of overestimation bias compared to that of DQN, a significant advantage attributed to the decoupling of action selection from Q-value generation when solving the Bellman equation. Additionally, the integration of a target network enhances stability, overcoming some of the issues faced by DQN. Notably, the model's capacity to develop effective strategies for a short course without prior data for training showcases its adaptability. Moreover, existing evidence suggests that DDQN is more robust and exhibits better generalization to new environments. However, the approach is not without its weaknesses, as it proves sensitive to hyperparameters and remains somewhat prone to instability, particularly in the face of sparse rewards. To address these limitations, future work should explore additional reward-shaping techniques and consider the implementation of policy gradient methods. Furthermore, to achieve true generalizability, the model would benefit from training on a more extensive range of courses and scenarios

Conclusion

In summary of this report, we were able to explore the use of double Q-learning algorithms to optimize a cyclist's power output for racing. Through the integration of physiological constraints such as critical power (CP) and Anaerobic Work Capacity (AWC), coupled with mechanical constraints derived from Newton's Second Law of Motion, we have navigated the complexities of modelling cycling performance within the reinforcement learning framework, particularly with the use of Double

Deep Q-Networks (DDQN). Despite facing challenges related to sparse rewards and hyperparameter sensitivity, our trained model was able to outperform maximal and pacing strategy riders on a test course. Looking ahead, future exploration of additional reward-shaping techniques, the introduction of more diverse training and the investigation of policy gradient methods show promise in further enhancing stability and generalizability.

References

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *arxiv.org*, 2015.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL `http://incompleteideas.net/book/the-book-2nd.html`.

Tim Miller. Markov decision processes. 2022.

Jonas Eschmann. *Reward Function Design in Reinforcement Learning*, pages 25–33. Springer International Publishing, Cham, 2021. ISBN 978-3-030-41188-6. doi: 10.1007/978-3-030-41188-6_3. URL `https://doi.org/10.1007/978-3-030-41188-6_3`.

David C Poole, Mark Burnley, Anni Vanhatalo, Harry B Rossiter, and Andrew M Jones. Critical power: an important fatigue threshold in exercise physiology. *Medicine and science in sports and exercise*, 48(11):2320, 2016.

Jiafan Feng, Xiaowen Liu, and Shaochun Deng. Optimal pacing strategy modeling of cycling individual time trials. In *Journal of Physics: Conference Series*, volume 2282, page 012003. IOP Publishing, 2022.

Graeme P Boswell. Power variation strategies for cycling time trials: a differential equation model. *Journal of Sports Sciences*, 30(7):651–659, 2012.

Faraz Ashtiani, Vijay Sarthy M Sreedhara, Ardalan Vahidi, Randolph Hutchison, and Gregory Mocko. Experimental modeling of cyclists fatigue and recovery dynamics enabling optimal pacing in a time trial. pages 5083–5088, 2019.

A Appendix

A.1 Code

The code for the model is available on Github at: `https://github.com/Bryce-Davidson/bike-simulation-RL-agent`

A.2 Constants

$$\text{AWC}_{\text{max}} = 9758$$

$$\text{CP} = 234$$

$$\text{mass} = 70$$

$$\text{G} = 9.81$$

$$\mathrm{C_{rolling}} = 0.0046$$

$$\mathrm{C_{drag}} = 0.19$$

$$\mathrm{a} = 0.00879$$

$$\mathrm{b} = 204.5$$

A.3 Power Equations

$$\mathrm{P_{max}} = 7 \times 10^{-6} \times \mathrm{AWC}^2 + 0.0023 \times \mathrm{AWC} + \mathrm{CP}$$

$$P_{\mathrm{agent}} = \mathrm{action} \times \mathrm{P_{max}}$$

A.4 Fatigue Equations

$$P_{\mathrm{adj}} = \mathrm{aP} + \mathrm{b}$$

$$\frac{dW}{dt} = \begin{cases} -(\mathrm{P} - \mathrm{CP}) & \mathrm{P} < \mathrm{CP} \\ (\mathrm{P_{adj}} - \mathrm{CP} & \mathrm{P} > \mathrm{CP} \end{cases}$$

A.5 Resistance Equations

$$\mathrm{slope} = \arctan(\frac{\mathrm{slope\ percent}}{100})$$

$$\mathrm{F_{gravity}} = \mathrm{G} \times \mathrm{mass} \times \sin(\mathrm{slope})$$

$$\mathrm{F_{rolling}} = \mathrm{G} \times \mathrm{mass} \times \mathrm{C_{rolling}}$$

$$\mathrm{F_{drag}} = \mathrm{C_{drag}} \times \mathrm{v}^2$$

$$\mathrm{P_{Resistance}} = (\mathrm{F_{gravity}} + \mathrm{F_{rolling}} + \mathrm{F_{drag}}) \times \mathrm{v}$$

$$\frac{d}{dt}(\mathrm{KE}) = \mathrm{P_{agent}} - \mathrm{P_{resistance}}$$

A.6 Velocity Equations

$$\mathrm{KE_{current}} = \frac{1}{2} \times \mathrm{mass} \times \mathrm{v}^2$$

$$\mathrm{KE_{final}} = \mathrm{KE_{current}} - \mathrm{P_{resistance}} + \mathrm{P_{agent}}$$

$$\mathrm{v_{current}} = \mathrm{sign}(\mathrm{KE_{final}}) \times \sqrt{\frac{2 \times |\mathrm{KE_{final}}|}{\mathrm{mass}}}$$

A.7 Figure of 10km Course Profile

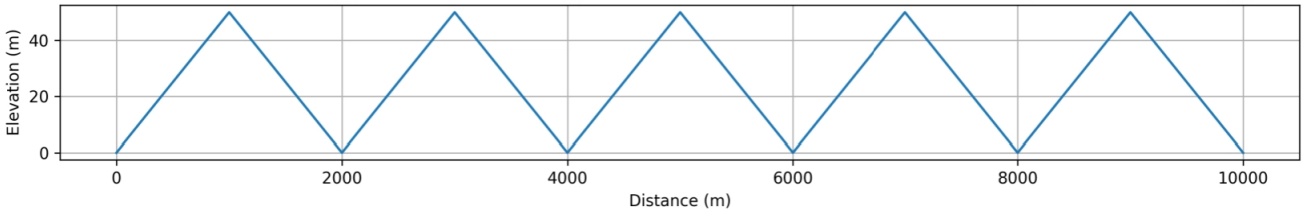

Figure 8: 10km test course used for simulation and environment testing

A.8 Figure of Simmulations on 10km Course

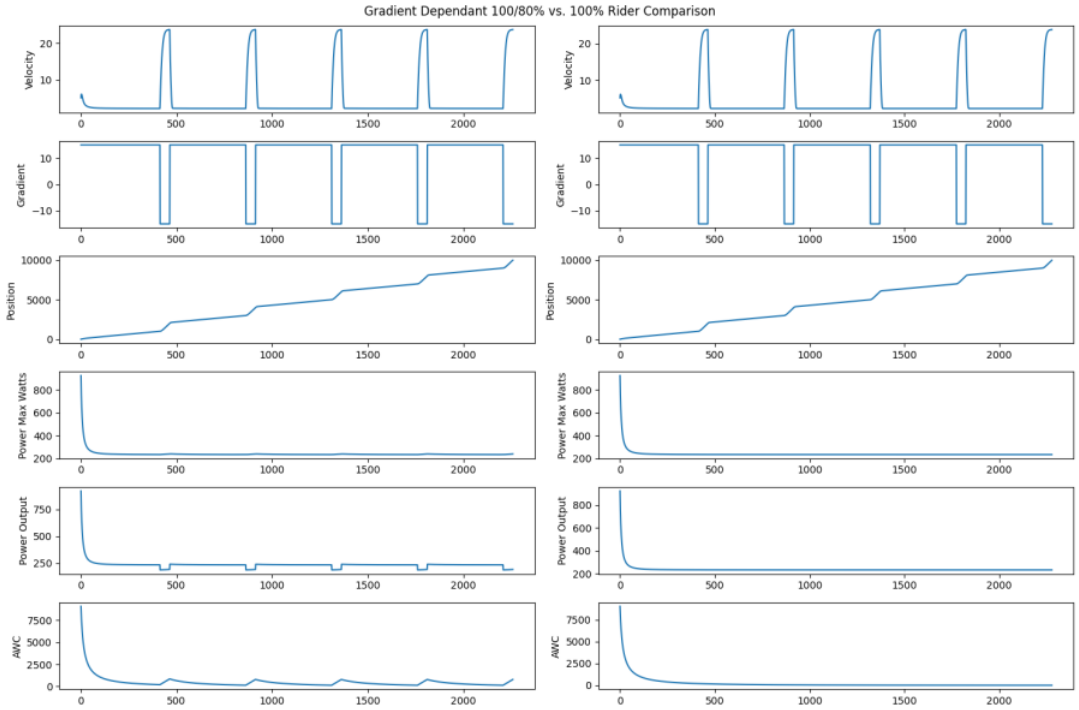

Figure 9: Comparison between maximal effort (10-rider) and pacing strategy (10/8-rider) simulations on the 10km course. The 10-rider finished the course in a time of 2279s while the 10/8-rider finished in a time of 2260s showing a proof of concept that pacing can be superior to maximal effort

A.9 Figure of Hyper-parameter setting with complex network

```
# Create the agent
agent = DDQN(
    input_dims=len(env.state),
    output_dims=env.action_space + 1,
    dense_layers=[200, 200, 200, 200],
    dropout=0.2,
    gamma=0.9,
    epsilon_start=1,
    epsilon_decay=0.9999,
    epsilon_min=0,
    target_life=100,
    memory_size=20000,
    batch_size=64,
    lr_start=0.01,
    lr_decay=0.9,
    lr_decay_steps=2000,
)
```

Figure 10: Hyper-parameter setting with a complex Neural Network architecture

A.10 Figure of Training run with a Complex Neural Network architecture

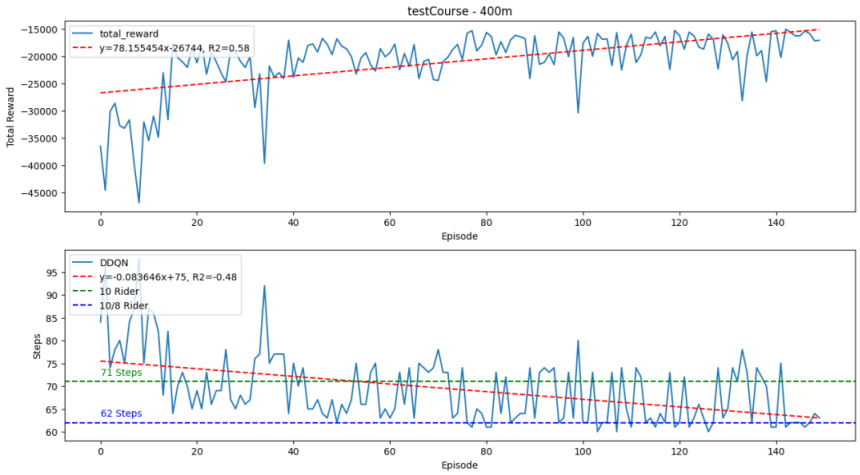

Figure 11: Training run with different hyperparameter settings.