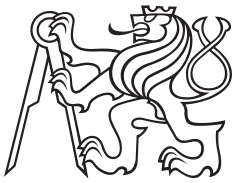
Bachelor's Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science an Engineering

Deep Reinforcement Learning on a Modified Car Racing Environment

Vojtěch Sýkora

Supervisor: doc. RNDr. Lukáš Chrpa, Ph.D.
Field of study: Open Informatics
Subfield: Artificial Intelligence and Computer Science
May 2023

Acknowledgements

Firstly I would like to thank my supervisor, who gave me a great job at the Artificial Intelligence center of the Czech Technical University, which helped me decide which topic to pursue. I would also like to thank my supervisor for their support while writing this thesis. Secondly, I would like to thank my family for helping me concentrate on the work and providing a great environment to live in. Finally, I am grateful for the help of my friends for helping me during my studies and for giving me valuable feedback from a different perspective.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guidelines for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by Act No. 121/2000 Coll., the Copyright Act, as amended, in particular, that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague, 9. May 2023

Abstract

The field of autonomous vehicles has been rapidly growing in recent years, with companies and researchers working on developing vehicles that can navigate complex environments without human intervention. One possible way of achieving such a feat is through the use of reinforcement learning.

To explore this option, OpenAI has developed a 2D car racing environment that can serve as a testbed for autonomous driving algorithms which utilize reinforcement learning techniques. The environment provides researchers with real-life physics and a continuous action space enabling development and testing without the need for physical testing environments and expensive hardware.

One of the current baselines in Deep Reinforcement Learning is the Proximal Policy Optimization (PPO) algorithm created by OpenAI. It has recently gained popularity thanks to its effectiveness in discrete and continuous action spaces and is being used even in models such as ChatGPT.

In this thesis, we will modify the car racing environment by introducing wind that can fluctuate in speed and direction. Wind can affect the movement of the car, and hence the driving agent has to take it into consideration. We will investigate how the PPO algorithm effectively adapts to handle the modified environment (with the wind). We will empirically evaluate PPO in the original and modified car racing environments while investigating pre-trained and non-pre-trained agents.

By investigating the impact of an outside and unpredictable factor such as wind on the learning of our agent, this project aims to contribute to the development of more robust and reliable self-driving vehicles.

Keywords: Artificial Intelligence, Neural Network, Deep Reinforcement Learning, Car Racing, Proximal Policy Optimization, Wind

Supervisor: doc. RNDr. Lukáš Chrpa, Ph.D.

Abstrakt

Oblast autonomních vozidel se v posledních letech rychle rozvíjí a společnosti a výzkumníci pracují na vývoji vozidel, která se dokáží pohybovat v komplexním prostředí bez zásahu člověka. Jedním z možných způsobů, jak takového výsledku dosáhnout, je využití zpětnovazebného učení.

Za účelem prozkoumání této možnosti vyvinula společnost OpenAI prostředí pro 2D automobilové závody, které může sloužit jako testovací prostředí pro algoritmy autonomního řízení využívající techniky posilování učení. Prostředí poskytuje výzkumníkům fyziku reálného světa a spojitý akční prostor umožňující vývoj a testování bez nutnosti fyzických testovacích prostředí a drahého hardwaru.

Jedním ze současných základních postupů v oblasti hlubokého zpětnovazebného učení je algoritmus Proximal Policy Optimization (PPO) vytvořený společností OpenAI. Ten si v poslední době získal popularitu díky své efektivitě v diskrétních i spojitých akčních prostorech a používá se i v modelech, jako je ChatGPT.

V této práci upravíme prostředí automobilových závodů zavedením větru, který může kolísat v rychlosti a směru. Vítr může ovlivnit pohyb automobilu, a proto ho musí řidičský agent brát v úvahu. Budeme zkoumat, jak se algoritmus PPO efektivně přizpůsobí upravenému prostředí (s větrem). Empiricky vyhodnotíme PPO v původním a upraveném prostředí automobilových závodů, přičemž budeme zkoumat předem natrénované a nenatrénované agenty.

Zkoumáním vlivu vnějšího a nepředvídatelného faktoru, jako je vítr, na učení našeho agenta chce tento projekt přispět k vývoji robustnějších a spolehlivějších autonomních vozidel.

Klíčová slova: Umělá inteligence, Neuronová síť, hluboké zpětnovazební učení, Automobilové závody, Optimalizace proximální politiky, Vítr

Překlad názvu: Hluboké zpětnovazební učení na modifikovaném prostředí závodění aut

Contents

1 Introduction 1

2 Car Racing Environment 2

2.1 Gym Library 2

2.2 Description 3

2.2.1 Actions 3

2.2.2 Episode end 4

2.2.3 Rewards 4

2.2.4 Difficulties 5

2.3 Modifications 5

2.3.1 Normalized Observation 5

2.3.2 Normalized Action 6

2.3.3 Wind 6

3 Proximal Policy Optimization (PPO) 8

3.1 Policy Gradient Method 8

3.2 Objective Function 9

3.3 Deep Neural Network Structure 11

4 Hyperparameter tuning 13

5 Training models 17

5.1 Continuous Wind from one Side 18

5.2 Gusty Wind from one Side 20

5.3 Continuous and Gusty winds from both sides 22

5.4 Training a pre-trained model on strong gusty winds 24

6 Evaluating models 26

7 Conclusion 31

Bibliography 33

A Hardware used for Training 35

B Software used & Repository with code 36

Figures

2.1 Left: Render for humans, Right: 96x96 render for agents [1] 3

3.1 PPO algorithm [2] 9

3.2 Clipping [2] 11

3.3 Our Deep Convolutional Neural Network 12

5.1 Continuous left wind 18

5.2 Continuous right wind 18

5.3 Gusty left wind 20

5.4 Gusty right wind 20

5.5 Continuous wind from both sides 22

5.6 Gusty wind from both sides 22

5.7 Pretrained on an environment without wind, trained on a gusty wind from both sides 24

6.1 Average score over 100 episodes . 27

Tables

6.1 The mean score of each model over all environments taken from the heatmap in Figure 6.1 28

6.2 The mean score of each environment/wind over all models taken from the heatmap in Figure 6.1 30

Chapter 1

Introduction

In recent years, Deep Reinforcement Learning (DRL) has been gaining popularity while being applied to a wide range of complex problems. From playing board games and computer games to teaching humanoid robots to walk, this field has achieved astounding results. One of the most promising applications of DRL which we aim to focus on is the development of self-driving vehicles which are finding themselves in increasingly complex environments.

To achieve this task we decided the ideal choice would be the Proximal Policy Optimization algorithm (PPO). PPO has recently gained popularity thanks to its state-of-the-art results and wide range of use cases. Fortunately, OpenAI, one of the leading AI research groups in the world, created the perfect environment for us to learn on using PPO. Driving a car is a massive task, hence it is better to begin with a 2D-controlled environment without all of the moving distractions facing us on the roads. Nevertheless, to make the learning more intriguing and more similar to real-life scenarios, we are adding two types of winds blowing from different directions.

Overall, this thesis aims to provide valuable insights into the development of more robust and reliable systems by investigating the impact of an outside and unpredictable factor such as wind.

Chapter 2

Car Racing Environment

For training agents using Reinforcement Learning, it is recommended to create an environment with which the agent can communicate without human input. We chose a premade environment from the Python library *gym* simulating a car driving on a track in 2D space. This environment intrigued us because of its similarity to real life with its usage of physics and a continuous action space.

2.1 Gym Library

Gym is an open-source Python library for developing and comparing Reinforcement Learning algorithms by providing a standard API to communicate between learning algorithms and environments, as well as a standard set of environments compliant with that API. Since its release, Gym's API has become the field standard for doing this. [1]. It also implements the usage of parallel/vectorized environments which can speed up the training. Its documentation can be found on `https://www.gymlibrary.dev/environments/box2d/car_racing/`.

2.2 Description

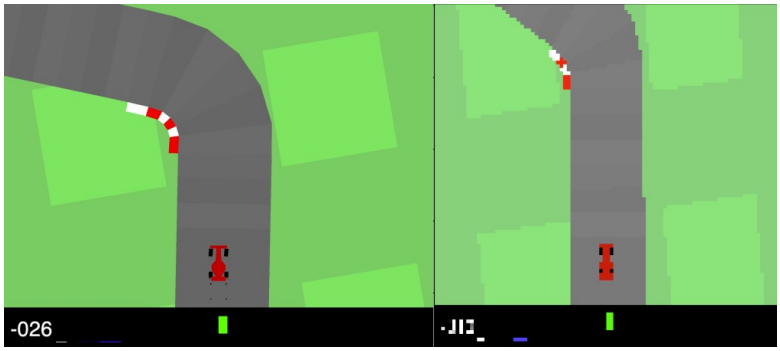


Figure 2.1: Left: Render for humans, Right: 96x96 render for agents [1]

Our chosen environment **CarRacing-v2** is a part of Box2D environments based on real-life physics. A random race track is generated for each episode which the agent can observe as an RGB image of size 96x96, which represents a top-down view of the racetrack with the car centered.

Some indicators are shown at the bottom of the window under the state RGB buffer. From left to right: actual speed, four ABS sensors, steering wheel position, and gyroscope. It is a powerful rear-wheel drive car, making it easy to start uncontrollably drifting.

2.2.1 Actions

- If **continuous**: There are three components of each action that can be combined since they are represented as a vector of size 3 (steering, throttle, break). The action is applied only in one step and does not propagate to the next step. What is quite inefficient and often happened with the model is that it can break and throttle at the same time.
  - **steering**: (0 is full left, +1 is full right). Any number in the range [0,1] is acceptable. 0.5 means no steering is applied. (The original environment has the interval [-1,1], which we normalized as described in subsection 2.3.2).
  - **throttle**: (0 is no throttle, 1 is full throttle). Any number in the range [0,1] is acceptable.
  - **break**: (0 is no break, 1 is full break). Any number in the range [0,1] is acceptable.

- If **discrete**: There are five actions: do nothing, steer left, steer right, throttle, and brake.

We used the more challenging continuous action space for this project because it better represents the real world.

2.2.2 Episode end

There are two types of an episode ending.

- **Termination**: If an error occurs. If the car goes so far off the track, it reaches the outside of the map.
- **Truncation**: If we reach the time limit for an episode. If a lap is finished, which in our case was completing at least 95% of the track. Since the track is generated randomly, the number of tiles in a track is also decided randomly for each episode. The track is a closed loop where the car begins somewhere on the right side driving counterclockwise.

2.2.3 Rewards

The reward for an episode is -0.1 for every step and +1000/N for every track tile visited, where N is the total number of tiles in the track.

$$episode\_reward = \frac{1000}{track\_length} \times tiles\_visited - 0.1 \times frames$$

If the car finishes the whole track, the equation simplifies to

$$episode\_reward = 1000 - 0.1 \times frames$$

For example, if you have finished the whole track in 420 frames, your reward is 1000 - 0.1*420 = 958 points. If the car drives completely off the field, it gets an additional -100.

This means the only policy for gaining a good positive score is staying on the track while driving fast. This equation for rewards also prevents a good score for an agent that cuts corners over grass or even, in extreme cases, one that makes a 360° turn at the beginning to go immediately through the end. The accepted score for solving this environment is 900.

2.2.4 Difficulties

There are an enormous number of different states because the track is generated randomly, so if the agent chooses an action, it cannot be known with certainty which state will follow. If the state could be any image of the given size, the number of possible states would be $256^{3*96*96}$ for RGB colors from 0-255; however, they use only some colors, and the image needs to look like a track. Nevertheless, the number would still be enormous.

The last issue is that the rendered view for a human has much more pixels than the observation returned from the environment, as seen in figure 2.1. This negatively affects the training accuracy, but it speeds it up tremendously.

2.3 Modifications

We decided to apply modifications using environment wrappers since they provided us with a clear division of each modification and the ability to stack them on top of each other easily. Environment wrappers are functions that stand between the agent and the environment and can alter the action or the observation that is being sent. In our case, we always had a wrapper normalizing the observation and clipping the action. Following these, we either had nothing for the environment without wind or added a single wrapper to simulate wind.

2.3.1 Normalized Observation

The first modification was normalizing the observation image vector. By squishing the RGB 0-255 values in the 0-1 range, we offer the Neural Network much friendlier numbers to work with. This also removed any need for normalization later on.

2.3.2 Normalized Action

The second modification was normalizing the action to be between 0 and 1. This removes negative numbers from our equations which is often a well-received change. In our case, it even enabled our code for wind wrappers to be much shorter and clearer.

Here is the formula we used to normalize our actions from a range of [-1,1] to [0,1]:

$$newAction = \frac{oldAction + 1}{2}$$

2.3.3 Wind

We decided to work with multiple different wind wrappers. Due to the implementation of the CarRacing-v2 environment, we decided that working with winds blowing from the left and right sides of the car would be a viable and interesting choice. Furthermore, we implemented continuously blowing winds and also gusty winds, gusty meaning that there are blocks of no wind and blocks of wind that alternate. One set of wrappers was purely for a left wind, another for a right wind, and the last for a mix of both.

All wrappers have parameters to be able to experiment more freely. There is always a strength range from which a number is randomly taken, representing the percentage change due to the wind.

For example, let us say our agent chose the action (0.9,0,0), meaning turn sharply to the right, and we had a wind blowing to the left with a strength range from 40% to 50%. Our random generator chooses a strength of 45% for the current step. The wind would change the first component of the action to $0.9 \times (1 - 0.45) = 0.495$, making the action (0.495,0,0). Lowering the first number of the vector means steering more to the left, and 0.45 is the wind strength.

This is the only parameter of the continuous left and right winds, while the continuous wind from both sides also has a block range. This is a range such as [10,50] from which a number is randomly taken, representing how many steps the current wind should act. For example, we could start with a left wind which generates a block length of 23 which means for the following

23 steps, a left wind will be applied. After this block is finished, we switch to the right wind and generate a new block length from the range for the right wind. If this new block length is 42, we apply the right wind from step 24 to step 24+42. This process repeats until the end of the episode.

Gusty winds have the same strength parameter while having a block range for both blocks when a wind is applied and also blocks when a wind is not applied. We separated these into two different parameters for more flexibility since we wanted to see what happens when the blocks with wind last three times longer than the blocks without wind.

Gusty winds from both sides work on the same principles. They begin with wind from a random side, and after its block is finished, a block without wind begins. These two blocks switch between each other, while the wind block always chooses a random side to blow from. This means that there can be an episode without a right wind if random change chooses to make it so. This episode would be the same as a gusty wind from the left. This is, however, highly unlikely.

All these random choices from ranges are meant to better simulate the real world's unpredictability.

Chapter 3

Proximal Policy Optimization (PPO)

Deep Reinforcement Learning has experienced immense growth in recent years, and one of the stable baselines has become the Proximal Policy Optimization algorithm (PPO) [2]. PPO learns a policy that minimizes its loss and therefore maximizes its reward. It is part of the Policy Gradient algorithms and is a modified superior version of the Trust Region Policy Optimization algorithm (TRPO) [3]. We will only describe PPO and its objective function; however, we recommend the original PPO paper for a full understanding of where it came from and how it is better than previous algorithms.

The proximal part of the algorithm refers to the fact that the policy is regularized to encourage it to stay close to the previous policy, which helps to improve the stability of the learning process. This regularization can also help avoid overfitting and improve the algorithm's overall performance.

3.1 Policy Gradient Method

This algorithm falls in the family of Policy Gradient methods. These algorithms learn online, which is the main difference between them and Deep-Q Networks (DQN). Policy gradient methods don't store past experiences in a replay buffer; instead, they learn directly from what the agent encounters. Once the batch of experiences is used, it is discarded. This is being called *less sample efficient*. DQN uses their experiences multiple times combined with experiences from other episodes, while Policy Gradient methods use

them from only the current episode and then delete them. Whereas standard Policy Gradient methods perform one gradient update per data sample, PPO has a novel objective function that enables multiple epochs of minibatch updates, which will be explored in a subsection **??** of the hyperparameter Tuning chapter.

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

Figure 3.1: PPO algorithm [2]

There are two alternating threads in PPO, the first of which (inner for loop in 3.1) collects experiences from interacting with the environment. If the hardware allows it, multiple actors can collect experiences at once. The second thread then runs gradient descent on the policy network using the saved experiences.

This division of labor can enable large-scale training with hundreds of CPU workers generating experiences and a couple of powerful GPUs learning from these experiences.

3.2 Objective Function

The objective function in the case of PPO is the loss function the model aims to minimize. Before we get to the objective function, let us define some useful formulae. Let $r_t(\theta)$ denote the probability ratio (also called the likelihood ratio) of new to old policy such that $r(\theta_{old}) = 1$.

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{3.1}$$

Where π is the policy with θ parameters (This will be, in our case, a Deep Neural Network.). a_t is the action to be chosen, and s_t is the current state.

TRPO maximizes the surrogate objective, which can be described as a conservative policy iteration.

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t \left[r_t(\theta) \hat{A}_t \right] \tag{3.2}$$

Where $\hat{A}_t$ is an estimate of the advantage function.

This can be calculated by subtracting the baseline estimate (the value output from the critic part of our Neural Network) from the discounted sum of rewards. In our case, we used a truncated version of the generalized advantage estimation, which is defined as follows

$$\begin{aligned} \hat{A}_t &= \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1} \\ &\text{where } \delta_t = r_t + \gamma V_\theta(s_{t+1}) - V_\theta(s_t) \\ &\qquad t \in [0, T] \end{aligned} \tag{3.3}$$

Where T is our hyperparameter horizon and $V_\theta(s_t)$ is our first output of the Deep Neural Network, also called the Critic (further information in section 3.3). λ is one of our hyperparameters called *GAE lambda* (generalized advantage estimation).

PPO clips the surrogate objective to prevent unreasonably large updates. The following is the clipped loss which is the main part of PPO's objective function.

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right] \tag{3.4}$$

Here ϵ is a hyperparameter which we tune in subsection **??** under the name *clipping range*.

As shown in figure 3.2, when the advantage is positive (meaning the outcome we obtained was better than expected), we clip actions with a high reward to not overly change the policy in one update. When the advantage is negative, our outcome is worse than expected, and we want to undo our previous step by a proportional amount. This represents the linear part of the graph.

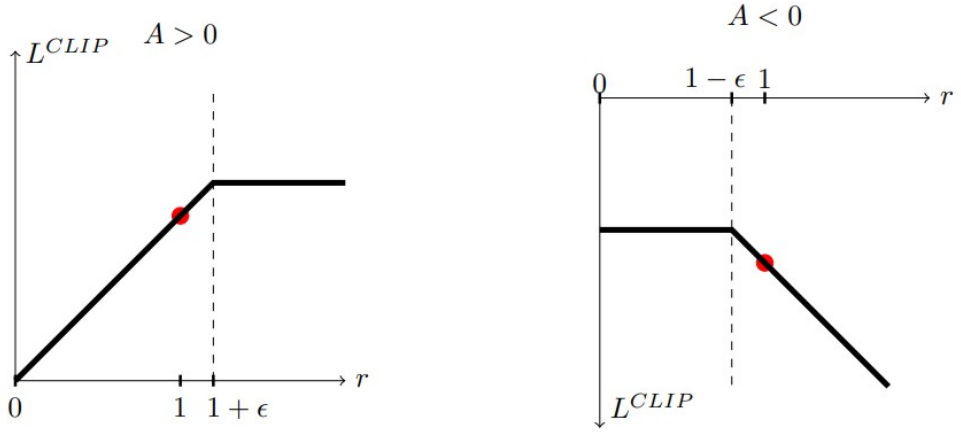


Figure 3.2: Clipping [2]

Finally, the full equation for the objective function of PPO can be defined as follows

$$L^{PPO}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right] \tag{3.5}$$

where c_1, c_2 are the value function coefficient and entropy coefficient. S denotes the entropy which we obtain from a Beta probability distribution created using our other two outputs of the neural network. These other two outputs are called the Actor (further information in section 3.3). L_t^{VF} is the predicted value (from our Neural Network) minus the target value squared.

$$L_t^{VF} = (V_\theta(s_t) - V_t^{\text{target}})^2 \tag{3.6}$$

3.3 Deep Neural Network Structure

The PPO algorithm falls in the family of Actor-Critic methods, which use a Deep Neural Network structure [4] oriented around having an Actor network and a Critic network. Our network has both the Actor and the Critic in a single network because they share the layers processing the observed game state (RGB image 96x96 pixels).

The Critic is responsible for estimating the expected value of a state. The value is the sum of all rewards it expects to receive in the future. This value is a scalar, and we obtain it from a single dense layer with a linear activation

because it can be any number. This layer can be seen on the bottom right of figure 3.3.

The Actor is responsible for generating a probability value for each possible action from the observed state. Since we used a continuous action space, our probability had to be expressed using a probability distribution. For this experiment, we used the Beta probability distribution [5], whose probability density function is defined as

$$f(x, \alpha, \beta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)} \tag{3.7}$$

We obtain the alpha and beta variables from the bottom right dense layers with a slight readjusting using the Lambda layer.

Both Actor and Critic share a convolution network that processes the inputted observation to extract some features and lower the number of total parameters. The current number of parameters of the whole network is 632 999, all being trainable.

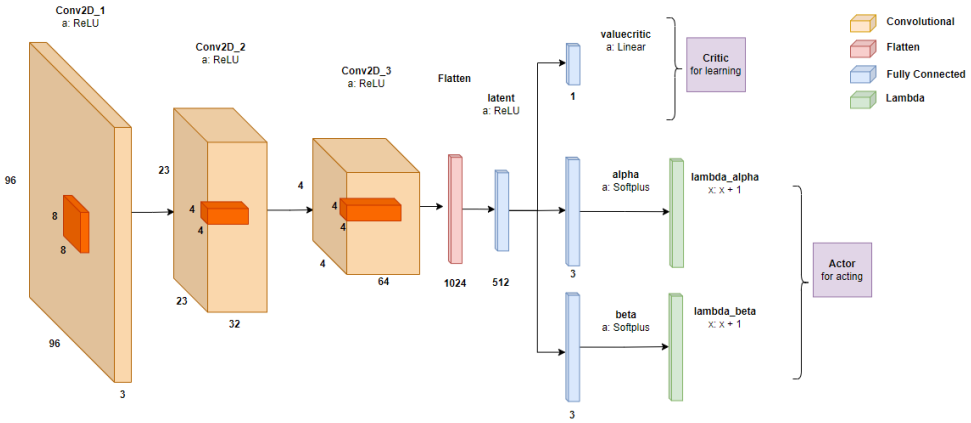


Figure 3.3: Our Deep Convolutional Neural Network

Chapter 4

Hyperparameter tuning

Each Deep Reinforcement Learning program has one issue, which is called hyperparameters. One can perfectly write the algorithm; however, for it to be effective, a good set of hyperparameters has to be found. Where to even begin? We did a detailed study of what each hyperparameter represents and how it affects learning [6]. This chapter will be a summary of our findings.

We decided to initialize the hyperparameters using our previous knowledge [7, 8] combined with the knowledge gained from the original PPO paper [2].

Our initial hyperparameters were

$$\begin{aligned} \text{horizon} &= 128 \\ \text{mini-batch size} &= 256 \\ \text{epochs per episode} &= 3 \\ \text{clipping range} &= 0.2 \\ \text{gamma} &= 0.99 \\ \text{gae lambda} &= 0.95 \\ \text{value function coefficient} &= 1 \\ \text{entropy coefficient} &= 0.01 \\ \text{learning rate} &= 2.5e\text{–}4 \end{aligned}$$

All of the hyperparameters could be found by testing a range of numbers taken from the theoretical description of what each hyperparameter does.

Some did not affect the outcome as much as others. For example, changing the entropy or value coefficients did not change the outcome by a large margin, whereas changing the gamma or the learning rate had some drastic effects. The tested hyperparameters were chosen based on theory; however, the final ones were chosen because of their performance in our experiments.

Firstly, the **horizon** is the number of steps in each episode. A new lap begins if the lap terminates before reaching the horizon. From this information, we can deduce that a low horizon would result in the car exploring the track in small portions before moving further. In contrast, a large horizon would result in the car exploring large portions of the track, slowing its initial learning. We decided on a large horizon to allow more exploration of the track.

The experience we collect during the episode can grow to a number in the higher tens of thousands, and even with great hardware, we cannot optimize our Neural Network with all of the experiences at once. This is where **mini-batches** come in. We optimize using gradient descent using a single batch of experiences simultaneously. Smaller mini-batch sizes are often preferred because they are noisy, which offers a regularizing effect and lowers generalization error. In our case, the best results were surprising, with a substantially large mini-batch size of 1024.

An **epoch** refers to one cycle through the full training dataset. Our training dataset is the experience buffer from one episode. It is recommended to train with more than one epoch, each having randomized mini-batches, because it leads to better generalization with not yet seen batches of experiences. In theory, the more epochs we have, the more information can be learned from one experience buffer; however, it is not the best strategy in practice. Too many epochs can cause overfitting, which may, in extreme cases, lead to almost no increase in rewards. We decided on three epochs per episode.

The **clipping range** is where we deem the surrogate objective acceptable. It is used to ensure that the policy update is not excessively large. A more detailed description was in section 3.2. Shortly, the higher the clipping range, the larger the policy update can be done, which could result in a drastic change in the policy. To keep the policy stable, a smaller number is often used. The number 0.15 worked well for us.

The discount factor represented by the Greek letter **gamma** (γ) accounts for the fact that our agent prefers rewards that it will receive now rather than the same reward further down the line. This can be compared to interest with finances, as a person would rather get the same amount of money now than in a year. To explain this further, if we have gamma=0.9, the reward in 6 steps

is half as important as the immediate reward, whereas, with gamma=0.99, the reward in 60 steps is half as important as the immediate reward. We went for 0.99 since a much more long-term view of the situation is preferred when driving.

GAE lambda (λ) is used to control the bias-variance trade-off. If you want a smoother training curve corresponding to more stable training, choose a λ close to zero. A number close to zero means high bias and low variance, while a number close to 1 means the opposite. Having a high variance in training is beneficial if one does not want to get stuck in a local minimum which is why we determined 0.9 would work best.

The **value function coefficient** controls the impact of the value function loss on PPO's objective function. It decides how influential our prediction of a state's value should be. We tried linearly spaced numbers between 0.5 and 1, and none was a misstep. Our final choice of 0.64 was because it had the highest recorded reward.

The **entropy coefficient** can be called a regularizer because it helps prevent premature dominance of one action probability over the policy, which could prevent exploration. A policy has minimum entropy when a single action has an overly dominant probability. This means that if we always wanted to be greedy and choose the current best action, we would have entropy as low as possible and a high entropy if we wanted to explore the state space. A very small entropy coefficient was the best choice in our case since one wrong action could lead to an uncontrollable drift out of the track.

The **learning rate** is how large of an impact the optimizer should have during a single update. For our experiment, we chose the Adam optimizer [9]. The original PPO paper described using a discounted learning rate [2]; however, we wanted to experiment with both discounted and constant learning rates. After experience collection and before our agent started learning, we applied a function to the learning rate to get a new one for each episode. If we wanted a discounted learning rate, we multiplied the initial learning rate by a decreasing number $(1 - \frac{\text{current episode number}}{\text{final episode number}})$, which decreased linearly from 1 to 0.

A decreasing learning rate is used because, at the beginning of training, it is useful to explore and be able to escape some local minima. As the agent learns and becomes better at gaining a positive reward, it is much less desirable to change the policy significantly in a single update.

The experiments showed that a learning rate of 0.0003 won in both constant and discounted runs, and if compared, the difference between discounted and constant learning rates was nonexistent. For our final learning rate, we took the discounted 0.0003 because of the theory.

Ultimately, we decided to continue with these hyperparameters, which gained a stable score of 900-940 in the environment without wind.

$$\begin{aligned}
\text{horizon} &= 2250 \\
\text{mini-batch size} &= 1024 \\
\text{epochs per episode} &= 3 \\
\text{clipping range} &= 0.15 \\
\text{gamma} &= 0.99 \\
\text{gae lambda} &= 0.9 \\
\text{value function coefficient} &= 0.64 \\
\text{entropy coefficient} &= 0.0071 \\
\text{learning rate} &= 3e\text{–}4
\end{aligned}$$

Chapter 5

Training models

In this chapter, we will focus on training the agents using each of the different winds mentioned in subsection 2.3.3. We decided to train on 4 strength ranges for each wind to be able to compare how the wind affects training when we increase or decrease the strength of the same wind. These strength ranges are between a 10% to 50% change in the incoming action. They are always mentioned in the graph's legend. The graphs will also have a graph with the training curve in the original environment without wind ($noWind$) for comparison.

We also experimented with different wind ranges (the number of steps wind is applied) and found some interesting occurrences. In the end, we took a pre-trained model from the original environment without wind and tried training it in an environment with a very strong and unpredictable wind.

5.1 Continuous Wind from one Side

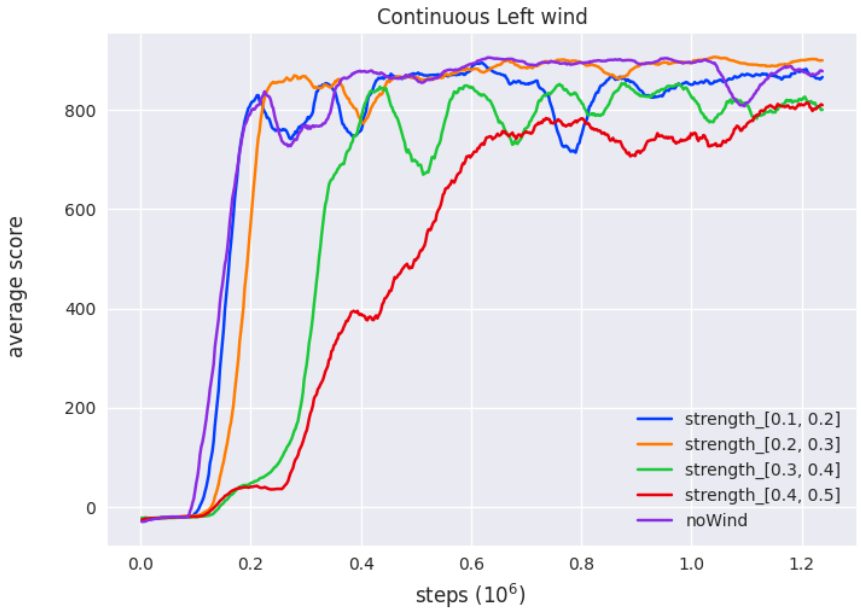


Figure 5.1: Continuous left wind

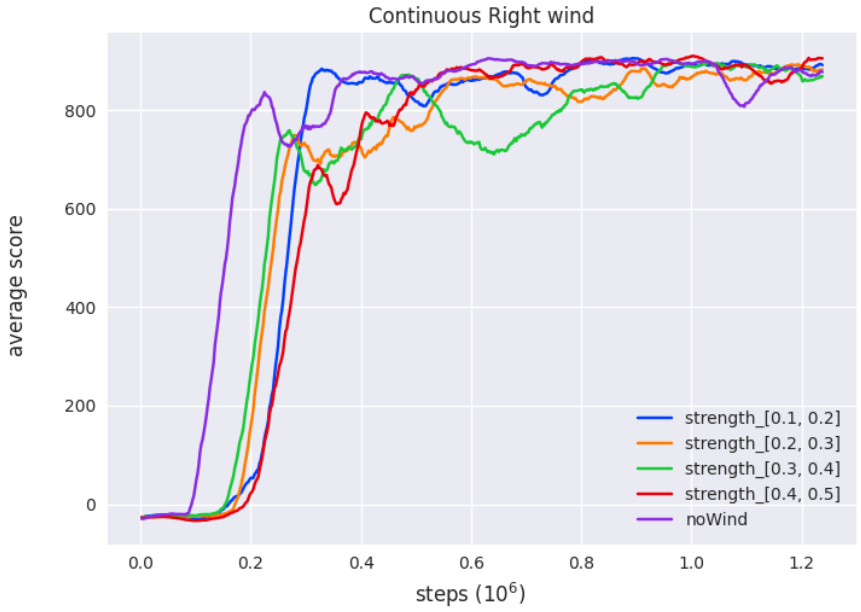


Figure 5.2: Continuous right wind

Let us begin with the most basic wind, being a never-ending wind from a single side, where the only changing state is the strength of the wind. For each step, a random wind strength from the strength range is chosen. We can see these strength ranges in the legends of each graph in this chapter.

It is not surprising how little of an impact the lower strengths have on the speed of training and maximum attainable score. If we think about it, the agent only needs to realize that the actions it is choosing have to be always tweaked in the same direction with a circa same number. What can be better seen in Figure 5.1 is the stronger the wind the slower the training which is quite logical.

What is quite interesting are the fluctuations of the [0.3, 0.4] (green) curve of the left wind (Figure 5.1). Since we saved a model each 10 training episodes (22500 steps), we were able to render these models visually with the same conditions to see what was happening.

Firstly, we focused on the section roughly from 180k steps to 270k steps since there the training slowed and then got exponentially better. At first, the model was driving really slow and always sliding to the left, however, it was already able to go through turns correctly. The next model was still sliding to the left because of the wind but was getting better at driving in the center of the road. The next model got faster, but only when it was in the middle of the road. The final model drove surprisingly well until it reached a right-hand turn which it drove through in a straight line. What can be deduced is that the training slowed down because the agent started going slower and focusing more on its direction. Even though none of these models could successfully overcome the wind for long, they got faster which gained them more positive rewards.

Secondly, we focused on the drop from the first peak above 800, meaning from 450k to 540k steps. What we found out is that the model started slipping a lot when going into sharper turns. It did not press the brakes correctly, it rather stomped on them and made the car drift on the grass. With the later models, we could see that the model somehow got worse and worse at finding its way back on track. After the lowest point, the model realized that it should drive more carefully through sharp turns and use the brakes more gently as not to drift.

5.2 Gusty Wind from one Side

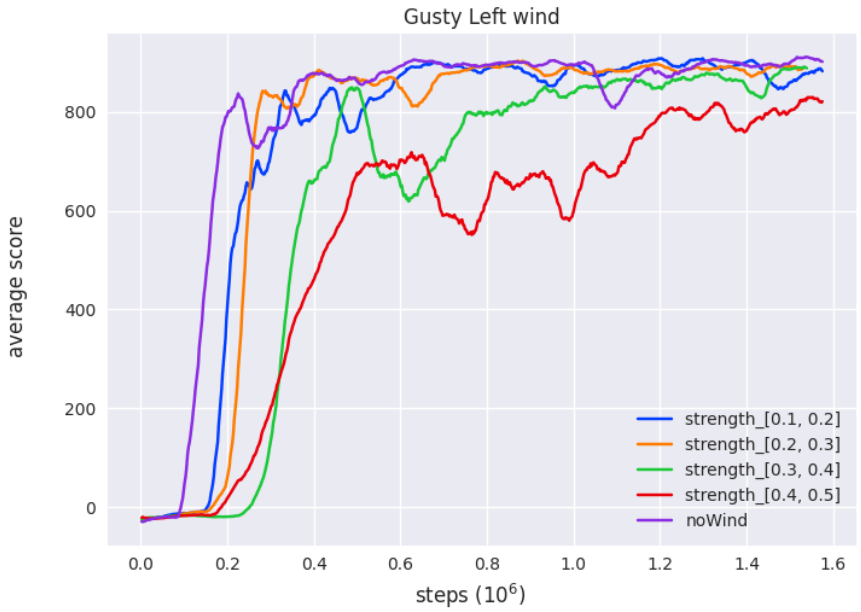


Figure 5.3: Gusty left wind

Figure 5.4: Gusty right wind

Gusty winds had overall less smooth curves compared to the continuous winds since the agent had to learn a combination of driving without wind and driving with wind and how to assess when the wind is active. Nevertheless, the training curves are smoother than if we used algorithms other than PPO since PPO has the clipping part of its loss function which prevents unreasonably large changes in the agent's strategy.

With the gusty left winds, we decided to focus on the area around the highest peak of the green curve in Figure 5.3. More specifically around 400k to 700k steps of the strength [0.3, 0.4] curve. When we visualized the models, in the beginning, the car wiggled a lot while staying on track and even managed to get back on the right track after doing a double donut while drifting. We could clearly see the parts when the wind hit and the car had issues only when it hit during a sharp turn. Around the peak of the graph, the agent learned that staying on the right side of the road is a good strategy. After that, the agent started having issues with breaking when approaching turns. With these issues came more and more going completely off the track to the point that one could only see grass around. It even got to a point when it started doing a continuous donut in the middle of the grass. This is quite surprising since the agent is using reinforcement learning which should force it to gain as much positive reward as possible. Staying on grass achieves the exact opposite. On top of that if the agent is on grass it does not matter what action it chooses, since all give the same reward. With this in mind, why would the agent learn to drift in a circle? The only explanation we could think of is the agent having some kind of a default action when the rewards over a long period show all actions as equal. This is quite possible since the agent chooses an action based on a sample from a probability distribution. This probability distribution is what is generated from the output of the Critic neural network. If the probability distribution of an action is close to a constant line parallel to the x-axis, all actions bear the same probability. And depending on how the function is coded, we might end up with the same action over and over. In the end, the agent learned again how to use brakes through turns, however, it learned for a short period to drive which in a controlled state of slight drift. Surprisingly, it was quite successful in such a state.

In the gusty right wind environment, we have a parameter of how long should the winds last and wanted to explore if it affects training. What we discovered is that wind range does not significantly affect the training. We tried to compare [10,50] with [50,100] and [100,150] for the wind range and there was no significant difference. Our thinking behind this was that a smaller wind range means the car gets only partially out of its track and has enough time to get back on its original track while with a [100,150] wind range the car can get much further away from its original path and won't be able to make it back in the short time without wind.

5.3 Continuous and Gusty winds from both sides

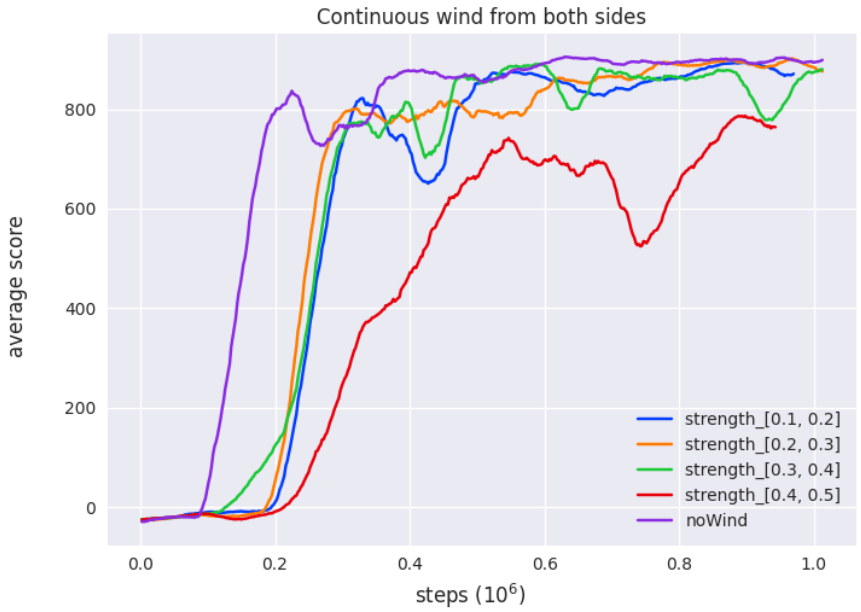


Figure 5.5: Continuous wind from both sides

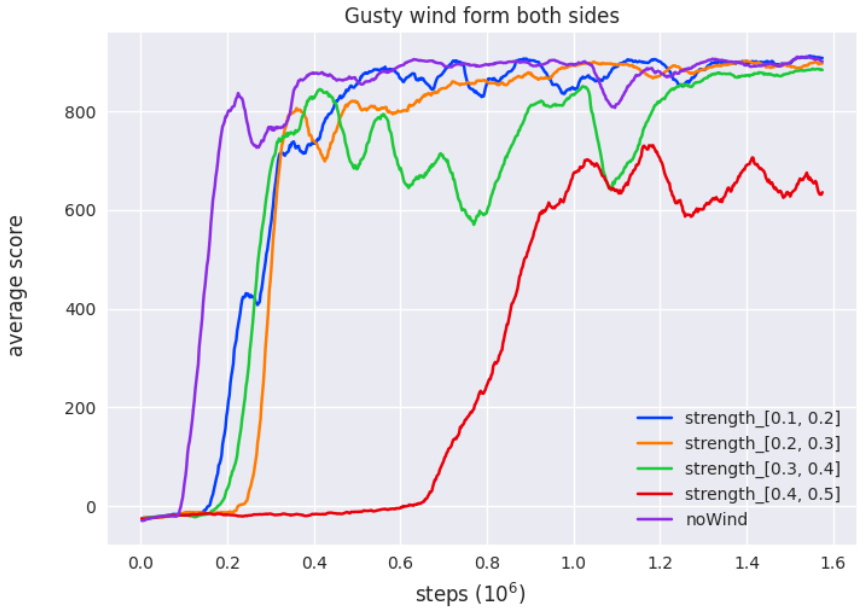


Figure 5.6: Gusty wind from both sides

Looking at Figures 5.5 & 5.6, we can observe very similar learning curves to the previous winds. Logically, the agent would have more difficulties learning under stronger winds, however, that was not the case with the three weaker winds. The only outlier is the strongest version of each wind.

The first anomaly we wanted to explore is why the model training on the strongest gusty winds took so long to see some rapid growth. We visualized our saved models from around the point where the red learning curve started growing just after 600k steps in Figure 5.6. Before the turning point, the car drives straight relatively slowly and does nothing of use, which corresponds to its score. After the turning point, it starts driving faster and quite well, albeit always having its right wheels outside of the track, which is quite unusual since we have winds from both sides. Continuing, the car drives relatively well until it sees two turns close to each other at which point the car freezes and slowly drives off into the abyss. Even when there is a track on the screen that it could see, there is not even a slight movement toward this track.

From this observation, we can deduce that our agent is surprisingly good at conquering even the strongest winds having little issues with staying on the track if there are no difficult turns. The ability to drive through turns comes afterward.

Another thought-provoking segment of the gusty winds training curves is the fluctuations of the second strong wind shown in green. This segment is from 400k to 800k steps. We begin with a car able to navigate anything that comes its way except for a 170° turn. Due to this, the car starts forgetting how to navigate through the sharper turns and uncontrollably drifts into the grass. Following this, the car starts breaking before turns, and if it starts drifting out of the track, it is able to drift back on track. Speed and sharp turns continue to be its most prominent enemies. After this point, the car continued to speed up and failed at turning even 90° turns. There was even a point when the car drove completely off the map, receiving a reward of -100.

This drive reminded us that there was not a single model in our testing which could recognize that it was driving the wrong way even if it was correctly driving on the road instead of the grass. The image the model received contained the current score in the bottom left; however, PPO does not have a memory which is the likely explanation.

5.4 Training a pre-trained model on strong gusty winds

We wanted to explore how a well-trained model in the original environment without wind will handle training in an environment with strong gusty winds from both sides. This would give us insight into the difference between learning from scratch and learning with some preexisting knowledge.

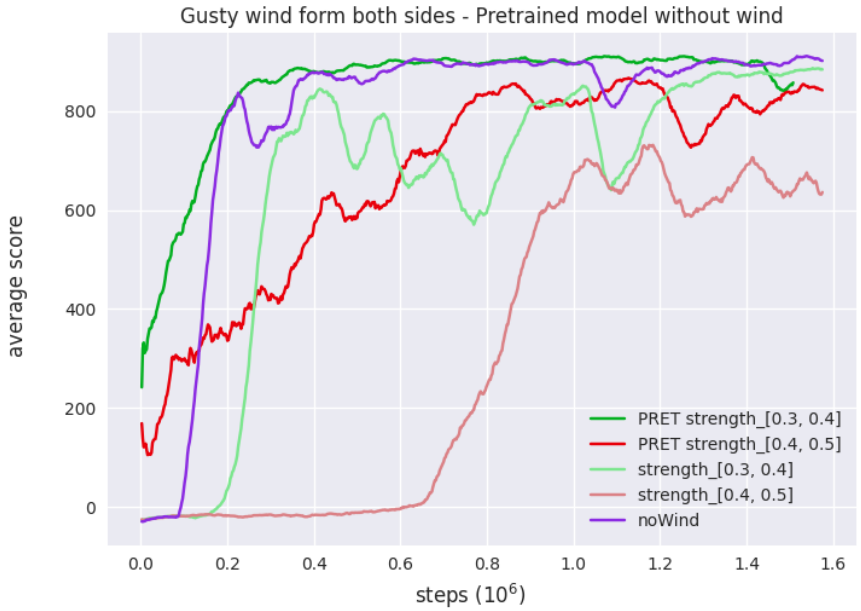


Figure 5.7: Pretrained on an environment without wind, trained on a gusty wind from both sides

In Figure 5.7, we can see the comparison of training a model from scratch (*strength_[0.3, 0.4]*) with training a model which had already been well trained on the same environment, but without wind (*PRET strength_[0.3, 0.4]*). For this, we decided to use a strong gusty wind from both sides which would offer quite a challenge. The same strengths are visualized by the same color, while the models trained from scratch are shown with a more faded version of the color.

As we have seen with the previous graphs, there was always quite a straight beginning followed by a steep rise. This was probably because of the challenge of overcoming the first turn. Once a model learned to drive through one turn, it rapidly achieved a higher score. Now if we compare that to the pre-trained

models, we can see a similar case with the softer wind shown by the green curve. On the other hand, the stronger wind shown by the red curve shows a slower incline.

When visualizing the beginning of the training of the pre-trained [0.4, 0.5] strong wind (vibrant red), we can notice the slight wiggling of the car due to the winds. Nonetheless, this wiggling was quite small if we consider that we are in the first ten episodes of the learning. Another aspect is the car cutting almost every turn. This continues to be an issue, albeit slowly disappearing, while the car learns to navigate the track faster and faster.

What can be taken from this section is the fact that pre-training a model definitely makes a difference since it gains a basic understanding of the environment, what it is looking at, where it should be aiming, and what each action does.

Chapter 6

Evaluating models

After the successful training, we were left with 25 trained models, one for no wind and four per wind type. To get as much information out of these models we decided to evaluate all of them in all the environments we had.

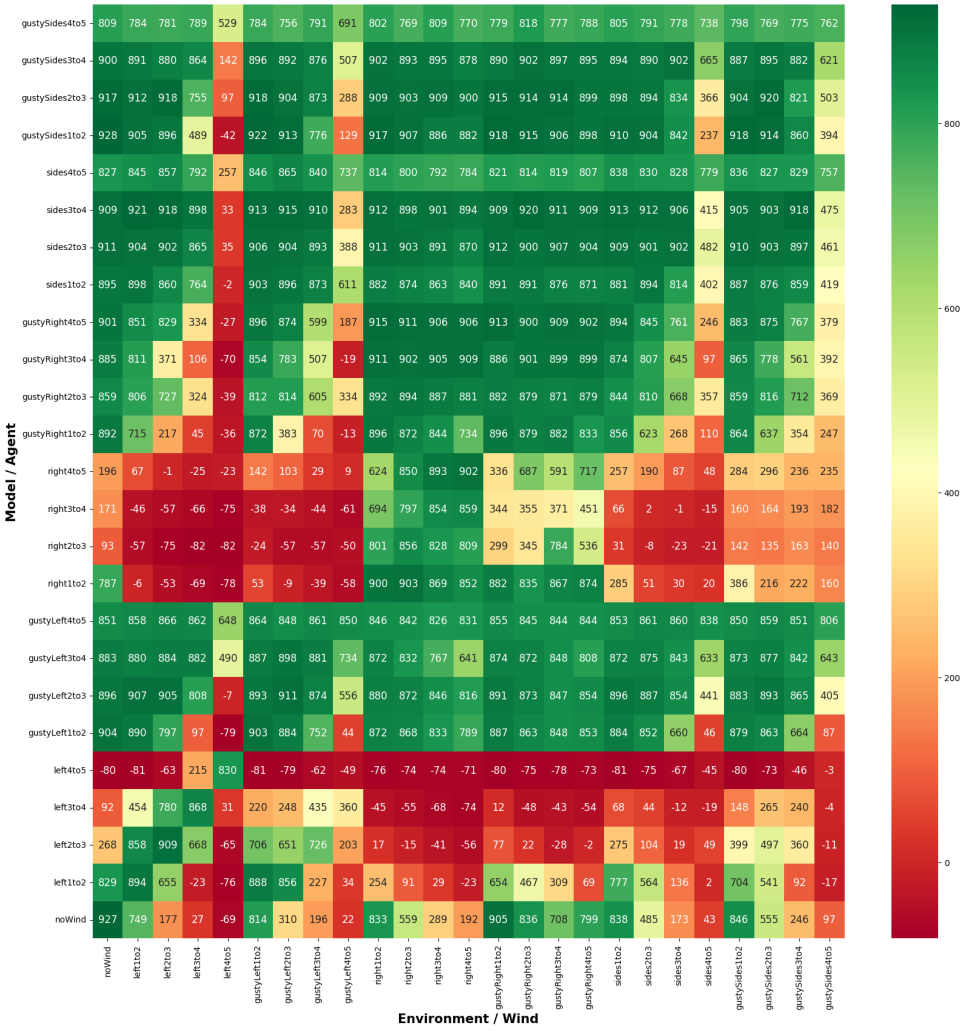


Figure 6.1: Average score over 100 episodes

In this comprehensive heatmap (Figure 6.1) can be seen a lot of information and an explanation of the axes is in place. *left1to2* means a continuous left wind (pushing to the left) with a strength range [0.1, 0.2], in other terms the wind changes the agent's actions by 10% to 20%. Other models follow the same logic of naming. And lastly, there is one model and its corresponding environment representing the original environment without wind.

We let the evaluation run for 100 episodes and took the mean score hoping for a quite accurate representation. The higher the score the better the model's performance.

The heatmap itself presents us with a lot of information and to analyze

it let us begin with finding the overall performance of each model. In Table 6.1 we can see the mean score of each model over all environments. This represents a mean over each row of the heatmap.

Model/Agent	Mean of scores
gustyLeft4to5	840
gustySides3to4	825
gustyLeft3to4	815
sides3to4	811
sides2to3	810
gustySides2to3	799
sides4to5	793
gustyLeft2to3	789
sides1to2	788
gustySides4to5	769
gustySides1to2	764
gustyRight4to5	734
gustyRight2to3	709
gustyLeft1to2	677
gustyRight3to4	658
gustyRight1to2	557
noWind	462
left1to2	357
right1to2	355
right4to5	309
left2to3	263
right2to3	217
right3to4	209
left3to4	153
left4to5	-20

Table 6.1: The mean score of each model over all environments taken from the heatmap in Figure 6.1

As can be seen in this table, the best model overall is the gusty wind pushing to the left with strengths between 40% and 50%. This is quite a surprise since we thought that the best models would be the ones trained on the middle ranges of strength from 20% to 40%. Our reasoning for this was that they could better generalize for the other wind strengths. A model trained on 40% to 50% winds should not be better than a model trained on 30% to 40% winds in a 10% to 20% winds environment. And this hypothesis is actually correct. If we look at the heatmap and compare *gustyLeft4to5* and *gustyLeft3to4* we can see that in all instances of *1to2* environments, the *gustyLeft3to4* had a higher score. Even if we compare these two models on every environment, the *gustyLeft3to4* is better at 15/25 environments which is

a majority. The issue here lies in consistency. *gustyLeft4to5* is better in overall performance because it consistently gets high scores in all environments, while *gustyLeft3to4*'s score drops significantly in some environments, which worsens a lot the overall score.

Returning to the overall winner, the second surprising part is that it is a model trained on a wind from one side. We thought that a model trained on a wind from both sides would be more prepared for whatever wind comes it's way due to its higher levels of generalization. By examining the heatmap, if we take the gusty left model and evaluate it on gusty right environments, it is always worse than the gusty sides model. This shows that our thinking was correct, however, there will again be some sort of other explanation as to why *gustyLeft4to5* is the best. And again the explanation seems to be the same as in the previous paragraph. *gustyLeft4to5* has consistently high scores while *gustySides3to4* has some enormous drops such as in the *left4to5* environment.

Speaking of the *left4to5* environment and also the *left4to5* model, they seem to be the absolute worst in everything by a large margin. The heatmap clearly shows this, with the dark red cells being most present in one row and one column representing *left4to5*. If we jump back to Figure 5.1 about the training of the left models, we can clearly see that the model *left4to5* training on the strongest winds had much more difficulty training than its respective right wind model. Reinforcement learning depends a lot on randomness; however, in this case, if we look at the training of the gusty left wind and its right counterpart, we can see the exact same outcome of the comparison of the red curves. This could mean that there is some proof to the models overall having more issues with the left wind.

Table 6.2 shows the mean of scores but this time shows environments instead of models. In this table, we see that *left4to5* was the most brutal environment of them all. We can also see that all right wind environments are above all left wind environments. One reasonable explanation for this phenomenon could be the issue of map generation. It is possible that the creator of the original *CarRacing-v2* environment unknowingly made the generation of sharp right turns more frequent than the generation of sharp left turns. If we add a strong wind pushing to the left, it becomes much more difficult to successfully steer through sharp right turns. This is, however, opposed by the fact that the track is a loop where the car drives counterclockwise, meaning it has to pass through more left turns than right turns. We even triple-checked the implementation of our winds to be sure that a left wind in our code means wind blowing the car to the left.

With all information combined, our only explanation for this phenomenon is the environment having sharper right turns than left turns, making it more

likely for the car to make a mistake while being affected by the strongest left wind.

Environment/Wind	Mean of scores
right1to2	725
right2to3	714
gustyRight1to2	701
gustyRight2to3	700
noWind	697
gustyRight3to4	697
right3to4	693
gustyRight4to5	686
gustySides1to2	679
gustyLeft1to2	669
right4to5	668
left1to2	664
sides1to2	661
gustySides2to3	647
gustyLeft2to3	617
sides2to3	597
left2to3	595
gustySides3to4	566
gustyLeft3to4	535
sides3to4	508
left3to4	447
gustySides4to5	339
sides4to5	276
gustyLeft4to5	268
left4to5	92

Table 6.2: The mean score of each environment/wind over all models taken from the heatmap in Figure 6.1

Chapter 7

Conclusion

In conclusion, this thesis delved deep into the algorithm known as Proximal Policy Optimization. From understanding how it works to testing its limits in unpredictable environments. We can say with confidence that even though PPO does not have a memory, its ability to understand and negate even the strongest of winds was quite unprecedented.

We were able to test a multitude of different winds thanks to the CarRacing-v2 environment from OpenAI's gym. It gave us an opportunity to replicate unpredictable winds while working with real-life physics of driving, continuous actions for a more accurate choice of action, and a random track generation.

Having explored ten hyperparameters, we can say that some had a larger impact than others. With some, the best one could be immediately observed from the learning curve, while with others, it was not so straightforward. We were surprised by how well the training went even when facing the strongest winds.

In the end, we were able to evaluate all models in all environments creating a comprehensive heatmap that helped us uncover that the best model overall was surprisingly trained on the strongest gusty left wind while the worst was trained on the strongest continuous left wind. Quite a contrast. It also helped us show that the most brutal wind was the strongest continuous left wind, while the easiest environment to navigate was the one with the weakest right wind.

The information gained in these experiments could be used to design experiments with real autonomous cars. For future projects, more unpredictable variables could be added into the environment such as narrower roads, obstacles on the track, or even some traffic.

Bibliography

[1] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," *arXiv preprint arXiv:1606.01540*, 2016.

[2] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *CoRR*, vol. abs/1707.06347, 2017.

[3] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust Region Policy Optimization," 2015.

[4] S. J. Russell and P. Norvig, *Artificial Intelligence: a modern approach*. Pearson, 3 ed., 2010.

[5] Çelik and M. Korkmaz, "Beta distribution and inferences about the beta functions," *Asian Journal of Science and Technolog*, vol. 7, pp. 2960–2970, 05 2016.

[6] V. Sykora, "Hyperparameter tuning of the PPO algorithm for OpenAI's CarRacing." 2023.

[7] A. Ng, Y. B. Mourri, and K. Katanforoosh, "Deep Learning Specialization," 2022. `https://www.coursera.org/specializations/deep-learning`.

[8] M. Straka, "Deep Reinforcement Learning," 2022. `https://ufal.mff.cuni.cz/courses/npfl122/2223-winter#home`.

[9] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," 2014.

[10] G. Van Rossum and F. L. Drake Jr, *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[11] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015. Software available from `https://www.tensorflow.org/`.

[12] "Anaconda Software Distribution," 2020. Anaconda Inc., `https://docs.anaconda.com/`.

[13] S. V. Bauer MW, Kurtzer GM, "Singularity: Scientific containers for mobility of compute," *PLoS ONE 12(5): e0177459*, 2017.

Appendix A

Hardware used for Training

The access to the computational infrastructure of the OP VVV funded project CZ.02.1.01/0.0/0.0/16_019/0000765 “Research Center for Informatics” is gratefully acknowledged.

We ran our experiments with a maximum number of 128 parallel threads set in TensorFlow. We found out that the optimal number of parallel environments was 6. With these numbers set, we ran experiments using 32GB per CPU and 1 CPU per task.

Information about the Research Center for Informatics can be found here `http://rci.cvut.cz/`

Appendix B

Software used & Repository with code

This thesis was written in the Python programming language [10]. The main Deep Learning library used was TensorFlow [11]. The rest of the libraries used as well as a tutorial for replicating our environment using an Anaconda environment [12] or a Singularity container [13] are in a GitHub repository.

Following is a link to the GitHub repository of this thesis.

`https://github.com/sykoravojtech/PPOthesis`