HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION COMMUNICATION TECHNOLOGY

Capstone project report

Reinforcement Learning Agent for Automatic Car Driving

***Supervised by*:**

Assoc. Prof. Nguyen Hung Son

PhD. Tran Viet Trung

TA. Doan The Vinh

TA. Nguyen Ba Thiem

***Presented by*:**

Trinh The Minh - 20225514

Dao Xuan Quang Minh - 20225449

Nguyen Minh Duong - 20225439

Vu Ngoc Ha - 20225490

Bui Anh Duong - 20225489

Hanoi, December 2024

Contribution

Name	Student ID	Contributions
Trinh The Minh	20225513	Building environment, framework and network structures
Dao Xuan Quang Minh	20225449	PPO algorithm
Nguyen Minh Duong	20225439	Advantage Actor-Critic Algorithm
Vu Ngoc Ha	20225490	Deep Q-Learning Algorithm
Bui Anh Duong	20225489	Deep Q-Learning Algorithm

Contents

1 Introduction **3**
1.1 Problem's Background 3
1.2 Final Objective 3

2 Theoretical basis **4**
2.1 Multilayer perceptron 4
2.2 Convolutional neural network 5
2.3 Reinforcement learning 7
2.4 On-Policy Learning 8
2.5 Off-Policy Learning 8

3 Environment **10**
3.1 Observation Space 10
3.2 Action Space 10
3.3 Reward Structure 11
3.4 Episode Terminated/Truncated 11

4 Methodology **12**
4.1 Policy 12
4.2 Algorithm 13

5 Experimental results **17**
5.1 Policy comparision 17
5.2 Algorithm comparison 17

1 Introduction

In recent years, autonomous driving has emerged as one of the most exciting and impactful areas in artificial intelligence and deep learning research. The development of autonomous vehicles holds the promise of improving road safety, reducing traffic congestion, and optimizing energy consumption. This project focuses on building a reinforcement learning agent for automatic car driving, a problem that combines the challenges of real-time decision-making, dynamic environments, and control optimization. The motivation for this work arises from the increasing relevance of self-driving systems in both academic research and real-world applications, as well as the opportunity to apply state-of-the-art reinforcement learning algorithms to a practical scenario.

1.1 Problem's Background

The problem centers on designing an autonomous agent capable of driving a simulated car within a virtual environment. Specifically, the **Better-Car-Racing** environment - a modified version based on **OpenAI Gymnasium's Car-Racing** - has been chosen as the testing ground. This environment provides a simplified yet effective platform for training and evaluating reinforcement learning agents. In CarRacing, the agent must navigate a race track by controlling actions such as steering, acceleration, and braking. The visual input from the environment consists of frame images that depict the race track and the car's position. To make the problem computationally tractable, the observations undergo a series of transformations, including resizing and grayscale conversion, before being processed by a deep neural network.

To address this problem, several reinforcement learning methods are suggested. **Deep Q-Learning (DQN)**, **Advantages Actor-Critic (A2C)** is explored for discrete control tasks, where actions are broken into finite choices. For continuous control, more advanced algorithms like **Deep Deterministic Policy Gradient (DDPG)** and **Proximal Policy Optimization (PPO)** are utilized. These methods leverage deep learning architectures, such as Convolutional Neural Networks (CNNs), to extract spatial features from input images and optimize the agent's decision-making process.

1.2 Final Objective

The ultimate goal of this project is to develop and train an autonomous driving agent capable of navigating race tracks in a stable and efficient manner. The agent must learn to maximize cumulative rewards by making optimal control decisions in real-time, such as steering precisely and adjusting speed through acceleration or braking. The success of the agent will be evaluated based on its ability to generalize across varying race tracks while maintaining consistent performance. By achieving this objective, the project demonstrates the practical application of reinforcement learning algorithms in solving dynamic and uncertain control tasks, which are central to the field of autonomous driving.

2 Theoretical basis

2.1 Multilayer perceptron

A **Multilayer Perceptron** (MLP) is a class of artificial neural network (ANN) that consists of multiple layers of neurons, where each layer is fully connected to the next. It is one of the most fundamental and widely used models in deep learning and machine learning, especially for tasks such as classification, regression, and feature extraction. MLPs are considered *feedforward networks*, meaning that information flows in one direction—from input to output—without looping back.

Structure of MLP

An MLP consists of three primary types of layer:

- **Input Layer**: The first layer, which receives the input data features. Each node in this layer corresponds to a feature in the input data.
- **Hidden Layers**: Intermediate layers between the input and output layers. These layers are responsible for processing the data. MLPs can have one or more hidden layers, each consisting of nodes (neurons) that apply an activation function to the weighted sum of the inputs.
- **Output Layer**: The final layer that produces the model's predictions or outputs. In classification tasks, this layer typically uses a *softmax* or *sigmoid* activation function to convert the raw output into probabilities or class labels.

2.1.1 Mathematics of MLP

Each layer in an MLP consists of neurons, where the output of each neuron is calculated as:

$$y_j = f\left(\sum_{i=1}^{n} w_{ij} x_i + b_j\right)$$

Where:

- y_j is the output of the j-th neuron in the current layer,
- x_i is the i-th input to the neuron,
- w_{ij} is the weight connecting the i-th input to the j-th neuron,
- b_j is the bias term for the j-th neuron,
- $f(\cdot)$ is the activation function applied to the weighted sum.

2.1.2 Activation Functions

Activation functions are applied to the weighted sum of inputs to introduce non-linearity in the network. Some common activation functions used in MLPs are:

- **Sigmoid**: $f(x) = \frac{1}{1+e^{-x}}$
- **ReLU** (Rectified Linear Unit): $f(x) = \max(0, x)$
- **Tanh**: $f(x) = \tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
- **Softmax**: Used in the output layer for multi-class classification problems, where the output is a vector of probabilities.

2.1.3 Advantages

- **Expressive Power**: MLPs are universal approximators, meaning they can represent any continuous function given enough layers and neurons.
- **Flexible**: MLPs can be applied to both supervised learning tasks (classification and regression) and unsupervised learning tasks (like clustering, when combined with other techniques).
- **Optimization**: With the use of modern optimization techniques (like Adam, RMSProp), MLPs can efficiently learn even from large datasets.

2.1.4 Limitations

- **Overfitting**: MLPs are prone to overfitting and also underfitting, especially when there is not enough data. Regularization techniques like dropout and L2 regularization can help mitigate this.
- **Computational Complexity**: MLPs can be computationally expensive, especially as the number of layers and neurons increases.
- **Vanishing/Exploding Gradients**: In deep networks, the gradients may either vanish (get very small) or explode (get very large), making training difficult. This issue is often mitigated using techniques like batch normalization and careful initialization.

In conclusion, the Multilayer Perceptron (MLP) is a powerful and flexible model in machine learning, capable of handling both regression and classification tasks. Its simplicity and flexibility make it a go-to method for many machine learning problems, though for more complex data such as images or sequential data, specialized models like CNNs or RNNs are preferred. Despite its limitations, MLPs are an essential building block in modern deep learning frameworks.

2.2 Convolutional neural network

In recent years, there has been a significant surge in interest in deep learning [1]. The convolutional neural network (CNN) is one of the most well-established algorithms among various deep learning models. It has been a dominant method in computer vision tasks since its remarkable performance in the 2012 ImageNet Large Scale Visual Recognition Competition (ILSVRC) [2], [3].

A convolutional neural network is a deep learning model that processes grid-patterned data, such as images. It draws inspiration from how animals' visual cortex is organized and is designed to learn spatial hierarchies of features automatically [4], [5]. A CNN typically consists of three types of layers: convolution, pooling, and fully connected layers. The convolution and pooling layers extract features, while the fully connected layer maps these features for classification. The convolution layer, in particular, utilizes a stack of mathematical operations, including specialized linear operations, to process pixel values in a 2D grid.

2.2.1 Convolution layer

The convolution layer is a crucial element of the CNN architecture. It performs feature extraction by combining linear and nonlinear operations, namely the convolution operation and activation function. Convolution involves applying a small array of numbers, known as kernels or filters, across the input, represented as an array of numbers called a tensor. At each location of the tensor, an element-wise multiplication between each element of the kernel and the input tensor is calculated and summed to obtain the output value in the corresponding position of the output tensor, known as a feature map Fig. 1. This process is repeated with multiple kernels to

create different feature maps representing various characteristics of the input tensors. Different kernels can be seen as distinct feature extractors. The size and number of kernels are two key hyperparameters that define the convolution operation. The former is typically 3x3 but can sometimes be 5x5 or 7x7.

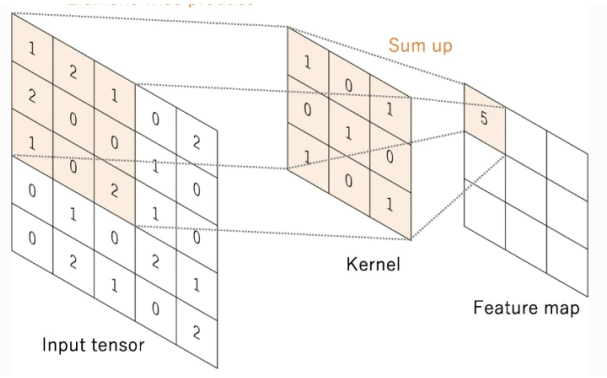

Figure 1: Convolution layer

2.2.2 Pooling Layer

The pooling layer conducts downsampling to decrease the dimensionality of feature maps, introducing translation invariance to small shifts and distortions and reducing the number of learnable parameters. It is important to note that pooling layers do not have learnable parameters but hyperparameters such as filter size, stride, and padding, similar to convolution operations.

Max pooling is the most commonly used type of pooling operation. It involves taking patches from the input feature maps, selecting the maximum value in each patch, and disregarding the remaining values Fig. 4. In practice, a common configuration is to use a 2×2 filter size with a stride of 2, which results in a downsampling of the feature maps' in-plane dimension by a factor of 2 while keeping the depth dimension unchanged.

2.2.3 Fully Connected Layer

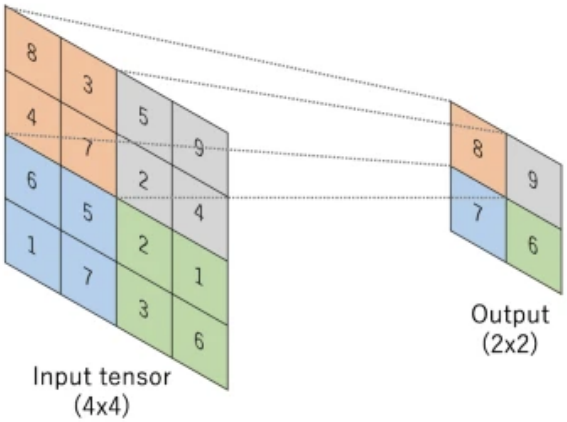

Figure 2: Max Pooling

The final convolution or pooling layer's output feature maps are usually flattened into a one-dimensional array of numbers and connected to one or more fully connected layers, also known as dense layers. These dense layers have learnable weights, and every input is connected to every output. The features extracted by the convolution layers and downsampled by the pooling layers are then processed by a subset of fully connected layers to produce the final outputs of the

network, such as the probabilities for each class in classification tasks. The final fully connected layer typically has the same number of output nodes as the number of classes.

2.2.4 Training

Training a neural network involves identifying patterns in convolution layers and adjusting connections in fully connected layers to minimize the differences between predicted outputs and actual labels in a training dataset. The backpropagation algorithm is commonly used for this purpose, relying on a loss function and gradient descent optimization algorithm. Model performance with specific patterns and connections is evaluated using the loss function through forward propagation, and the adjustable parameters, such as patterns and connections, are updated based on the loss value using the backpropagation and gradient descent optimization algorithm.

The gradient descent algorithm is a crucial optimization technique in machine learning. It works by iteratively adjusting network parameters, such as weights and kernels, to reduce the loss function. The gradient of the loss function indicates the direction of the steepest increase, so parameters are updated in the opposite direction, with the learning rate determining the step size Fig 3. Mathematically, the gradient is the partial derivative of the loss concerning each parameter, and the parameter update is expressed as:

$$w := w - \alpha * \frac{\partial L}{\partial w} \tag{1}$$

- w: learnable parameters
- α: learning rate
- L: Loss

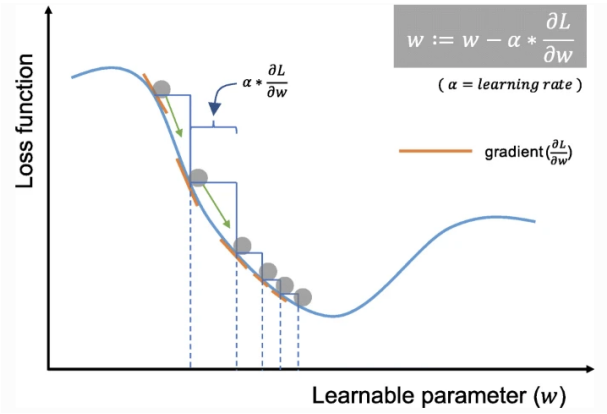

Figure 3: Gradient Descent

Additionally, various enhancements to gradient descent, such as SGD, SGD with momentum, RMSprop, and Adam, have been developed and are widely used. However, the details of these methods are beyond this project's scope.

2.3 Reinforcement learning

Reinforcement Learning (RL) is a subfield of machine learning in which agents learn to make decisions by interacting with an environment. One of the key aspects of RL algorithms is the way they update their policies based on the observed experiences. This document explains the difference between **On-Policy** and **Off-Policy** learning, which are two primary types of RL approaches that define how an agent's policy is updated using past experiences.

In reinforcement learning, an agent interacts with an environment by taking actions based on its policy, observing the resulting states and rewards, and learning from this interaction to improve future decision-making. Two common paradigms in reinforcement learning are **On-Policy** and **Off-Policy** learning. The key distinction between these paradigms is the relationship between the policy used to generate the agent's experiences and the policy that is being improved during learning.

2.4 On-Policy Learning

In on-policy learning, the agent learns a policy that is updated based on the same policy that is being used to interact with the environment. In other words, the agent explores the environment and makes decisions according to its current policy, and then updates that policy using the experiences it gathers. The policy and the behavior policy (the policy used to explore the environment) are the same.

2.4.1 Key Characteristics of On-Policy Learning

- The agent follows its current policy while collecting experience.
- The policy that is improved is the same as the one used for exploration.
- Exploration and exploitation are directly linked through the same policy.

2.4.2 Example of On-Policy Learning: SARSA

One of the most well-known on-policy algorithms is **SARSA** (State-Action-Reward-State-Action). In SARSA, the agent selects actions according to the current policy (typically using an ϵ-greedy strategy) and updates the Q-values based on the action actually taken and the next action chosen under the current policy. The update rule for SARSA is as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right)$$

Where:

- $Q(s_t, a_t)$ is the action-value function at state s_t and action a_t,
- r_t is the reward received after taking action a_t in state s_t,
- γ is the discount factor,
- α is the learning rate,
- a_{t+1} is the action selected in the next state s_{t+1}.

2.5 Off-Policy Learning

In off-policy learning, the agent learns a policy that can be different from the policy it uses to explore the environment. The behavior policy (which dictates how the agent explores the environment) is separate from the target policy (which the agent seeks to improve). Off-policy learning allows the agent to learn from experiences generated by a different policy, which can be beneficial when it is difficult to explore the environment thoroughly using the current policy.

2.5.1 Key Characteristics of Off-Policy Learning

- The agent may use a different behavior policy to collect experiences than the policy being improved (target policy).
- Exploration and exploitation are decoupled, allowing for more efficient exploration strategies.
- Off-policy methods often leverage experience replay, where past experiences are stored and reused for learning.

2.5.2 Example of Off-Policy Learning: Q-Learning

A popular off-policy algorithm is **Q-learning**. In Q-learning, the agent learns the optimal action-value function regardless of the policy used to generate the experiences. The Q-values are updated using the maximum expected future reward, assuming that the optimal policy is followed. The update rule for Q-learning is:

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

Where:

- $\max_{a'} Q(s_{t+1}, a')$ represents the maximum Q-value at the next state, assuming the optimal action is chosen.

In contrast to SARSA, the update rule for Q-learning uses the maximum value from the next state ($\max_{a'} Q(s_{t+1}, a')$) instead of the Q-value for the next action chosen by the current policy.

2.5.3 Comparison of On-Policy and Off-Policy Learning

The following table summarizes the key differences between on-policy and off-policy learning:

Feature	On-Policy Learning	Off-Policy Learning
Policy for Exploration & Exploitation	Same policy for both	Different policies for exploration and exploitation
Learning from Experiences	Uses experiences generated by the current policy	Can learn from experiences generated by other policies
Flexibility in Exploration	Limited flexibility, as exploration is tied to the policy	High flexibility, can use more diverse exploration strategies
Efficiency	May require more exploration to find optimal policy	Can reuse past experiences and improve efficiency

Table 1: Comparison of On-Policy and Off-Policy

On-policy and off-policy learning are two fundamental paradigms in reinforcement learning. On-policy methods, like SARSA, require the agent to learn from experiences generated by the current policy, while off-policy methods, like Q-learning, allow the agent to learn from experiences generated by different policies. Off-policy methods offer more flexibility and efficiency, especially in environments where exploration can be decoupled from exploitation. Understanding the differences between these approaches is essential for designing reinforcement learning algorithms that balance exploration and exploitation effectively.

3 Environment

The environment used for this project is called **Better-Car-Racing** - a customized environment based on the Car-Racing environment provided by the Gymnasium library with Box2D physics. This section outlines the transformations applied to the environment to incorporate additional state information and modify the input representation for enhanced learning.

The Better-Car-Racing environment is a control task from OpenAI's Gym library, designed to test an agent's ability to navigate procedurally generated racetracks. The environment presents the agent with a top-down view of a race track and requires it to learn how to drive a car effectively through the track while avoiding off-road areas.

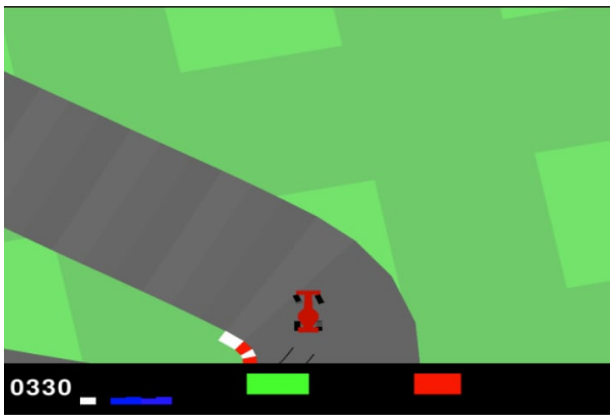

Figure 4: Better-Car-Racing Environment

This environment provides a challenging benchmark for reinforcement learning algorithms, requiring the agent to balance exploration, control, and planning. [6]

3.1 Observation Space

The observation space is designed to include rich and diverse information for the agent. The following components are utilized:

- **Distance Vector:** Using n rays to divide the front vision into n parts, where the rays have equal length. In this project, $n = 7$ and the ray length is 200 units of length. This provides spatial awareness of the track boundaries.
- **Movement Information Vector:** This vector captures 7 dynamic properties of the car, including:
  - Linear speed of the car.
  - Angular velocities of all four wheels.
  - Steering angle of the car.
  - Angular velocity of the car's main body.
- **Top-Down Image:** A top-down 96x96x3 RGB image capturing the car and the race track layout.

3.2 Action Space

The action space is discrete and includes the following actions:

- 0: Do nothing.

- 1: Steer left.
- 2: Steer right.
- 3: Accelerate.
- 4: Brake.

3.3 Reward Structure

The reward function is designed to encourage efficient exploration and penalize undesirable behaviors:

- A reward of -0.1 is given every frame to discourage prolonged episodes.
- A reward of $+\frac{1000}{N}$ is given for every track tile visited, where N is the total number of tiles in the track.
- A penalty of -0.5 is applied each time the car goes into the grass.
- When terminated, reward is -500.

3.4 Episode Terminated/Truncated

Terminated: Reach terminal state.

Truncated: End without reaching terminal state, often indicated by environment, i.e time limit exceeded. The episode terminates under the following conditions:

- When the car goes outside the track, resulting in a large negative reward (optional).
- Linear speed of the car does not change after several timesteps.

The episode truncates under the following conditions:

- When all track tiles are visited, signaling successful completion.
- Reach time limited (1000)

4 Methodology

4.1 Policy

A policy defines the way the learning agent behaves at a given time. Roughly speaking, a policy is a mapping from perceived states of the environment to actions to be taken when in those states. Policies can be deterministic, where a specific action is chosen for each state, or stochastic, where actions are drawn from a probability distribution conditioned on the state.

We defined 3 distinct types of policy: Linear, CNN and the combination CNN + Linear.

4.1.1 Linear Policy

The linear policy is applied to the 3.1 **Distance Vector and Movement Information Vector** - This provides a compact representation of spatial and dynamic information - allowing the model to optimize driving behavior based on linear features of distance and movement information. This approach reduces complexity while maintaining competitive performance in the environment.

By using MLP to transform the input vector step-by-step through hidden layers of increasing dimensions (from size 14 to 256), the MLP learns higher-level abstract features that are critical for making decisions in the driving environment. Finally, the output layer maps the learned representation to the desired output, representing the model's policy actions. The network architecture backbone of linear policy is described as 5.

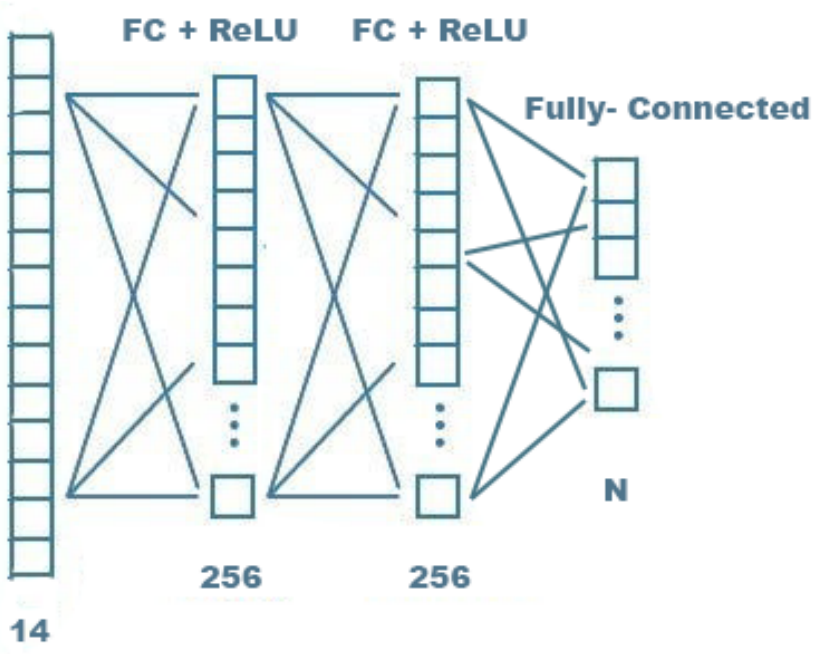

Figure 5: Multilayer perceptron backbone

4.1.2 CNN Policy

The CNN policy processes raw pixel data from the game screen, extracting spatial and temporal features to inform decision-making applied to 3.1 **Top-Down Image**. By leveraging convolutional layers, the policy can capture intricate patterns in the racing track, obstacles, and vehicle dynamics, enabling the agent to make more nuanced and accurate decisions. For this setup, the CNN takes inputs of 4 consecutive 96×96 pixels grayscale images into an image of size 96×96×4. The network architecture backbone of CNN policy is described as 6.

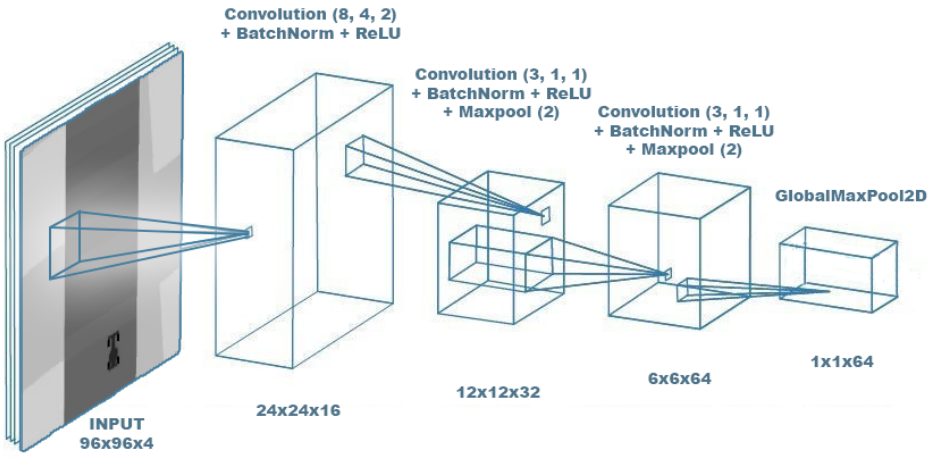

Figure 6: CNN backbone

4.1.3 Combined Policy

Similarly to 4.1.2, this **Combined Policy** takes **Top-Down Image** input into the CNN backbone 6 and then flattens into a one-dimensional vector, then concatenated with the **Distance Vector and Movement Information**, creating a combined feature vector. The combined feature vector is then fed into a fully connected layer before becoming the final output vector of the network, representing the probability distribution over possible actions the car can take. The network architecture backbone of Combined Policy is described as 7.

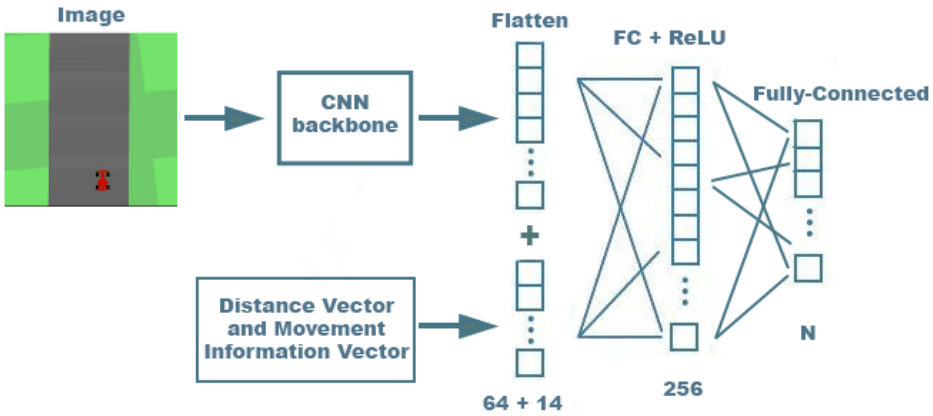

Figure 7: CNN + Linear backbone

4.2 Algorithm

4.2.1 Deep Q-learning

Network Architecture:

- Q_{net}: The Q-network is defined as:

$$Q_{\text{net}} = \text{FC}_{256\to5}\big(\text{Backbone(x)}\big)$$

  where:

  – FC is the fully connected layer, with the subscript $256 \to 5$ indicating 256 input neurons and 5 output neurons.

– Backbone(x) is the feature extractor chosen based on the input type.

- Target Q-Net(x): The weights of the target Q-network are initialized to be equal to the Q-network weights:

$$W_{\text{Q-Target}} := W_{\text{Q-Target}}$$

Loss Function:

$$L_q = \mathbb{E}\left[\text{SmoothL1Loss}\Big(Q_{\text{net}}(x), r + \gamma \cdot \max Q_{\text{target}}(x')\Big)\right]$$

where:

- x is the current state.
- x' is the next state.
- r is the reward.
- γ is the discount factor.

Training Phase:

- Step-wise Training:
  - Batch size: 64
  - Discount factor (γ): 0.99
  - Learning rate (π): 0.00001
  - Training frequency: The network is updated every 4 steps.
  - Soft update coefficient (τ): 0.001
- Exploration Strategy: Use the ε-greedy strategy, where the action a is chosen as:

$$a = \begin{cases} \text{random action} & \text{with probability } \varepsilon \\ \underset{a'}{\text{argmax}}\ Q(s, a'; \theta) & \text{with probability } 1 - \varepsilon \end{cases}$$

  ε Decay: The value of ε decreases gradually from a high value (e.g., 1.0) to a low value (e.g., 0.01) during training.

- Update Target Network: The target network is updated using a soft update rule:

$$W_{\text{Q-Target}} = \tau \cdot W_{\text{Q-Net}} + (1 - \tau) \cdot W_{\text{Q-Target}}$$

  where τ is a small positive value (e.g., 0.001) controlling the update rate.

4.2.2 Advantage Actor-Critic

Network architecture:

The architecture consists of two networks: Actor and Critic.

- Actor: $FC_{256->5}(Backbone(x))$
- Critc: $FC_{256->1}(Backbone(x))$

Here, the Actor is responsible for selecting current actions based on current policy; it outputs a probability distribution over actions or directly selects an action. Besides, the critic evaluates the action taken by the actor by estimating the value function, which represents the expected return (reward) from a particular state or state-action pair.

The *Backbone* used here depends on the shape of the input state. 4.1

The loss function of Advantages Actor-Critic is defined as:

$$L_{\text{Actor}} = -\mathbb{E}_t[A_t \cdot \log(Softmax(\text{Actor}(s_t)))]$$

$$L_{\text{Critic}} = \alpha \cdot \mathbb{E}_t[SmoothL1Loss(\text{Critic}(s_t), R_t)]$$

$$L_{\text{Entropy}} = -\beta \cdot \mathbb{E}_t[Entropy(Softmax(\text{Actor}(s_t)))]$$

$$L = L_{\text{Actor}} + L_{\text{Critic}} + L_{\text{Entropy}}$$

with α, β are critic coefficient and entropy coefficient, respectively. The actor loss, L_{Actor}, assesses the effectiveness of the policy's actions based on the advantage estimates, A_t. The critic loss, L_{Critic}, measures the accuracy of the value function's predictions for the state s_t. The entropy loss, L_{Entropy}, discourages the policy from becoming overly deterministic, encouraging exploration and helping to avoid premature convergence to suboptimal strategies.

The advantages and returns are computed using Generalised Advantage Estimator (GAE) [7] as in these below formulas:

$$A_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

where the temporal difference error δ_t is defined as:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

Here, A_t is the advantage at time t, r_t is the reward at time t, $V(s_t)$ is the value estimate for state s_t, $V(s_{t+1})$ is the value estimate for the next state, γ is the discount factor, and λ is the GAE hyperparameter.

The return R_t can be computed as:

$$R_t = A_t + V(s_t)$$

where R_t is the return at time t.

The training phase of Advantage Actor-Critic algorithm episode-wise update, meaning the total loss L will be updated after one episode. Below is the hyperparameter set that we found most efficient to training phase:

- Discount Factor $\gamma = 0.99$
- GAE-Lambda $\lambda = 1$
- $\alpha = 1$
- $\beta = 0.01$

4.2.3 Proximal Policy Optimization (PPO)

PPO also use Actor-Critic framework to design its network architecture. However, there are some difference in designing the loss function, leads to unique behaviour of PPO compared to other RL algorithms. PPO make used of a quantity called "probability ratio"

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$$

This quantity prevents the agent from excessively large policy update, therefore makes the training phase more stable. The L_{Actor} is a clipped surrogate loss:

$$L_{\text{Actor\_unclipped}} = -\mathbb{E}_t[r_t(\theta) \cdot A_t]$$
$$L_{\text{Actor\_clipped}} = -\mathbb{E}_t[\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) \cdot A_t]$$
$$L_{\text{Actor}} = min(L_{\text{Actor\_unclipped}}, L_{\text{Actor\_unclipped}})$$

To prevent the Critic from excessively updating the weight that could make it misjustifies action from current policy, we also clip L_{Critic}:

$$V = \text{Critic}(s_t)$$
$$V_{\text{clip}} = V + clip(V - V_{\text{old}}, -clip\_V, clip\_V)$$
$$L_{\text{Critic}} = \alpha \cdot \mathbb{E}_t[SmoothL1Loss(V_{\text{clip}}, R_t)]$$

These modifications make PPO more robust in the training phase. In the tradeoff, training PPO is extremely slow and has a high computational cost.S

In practice, training PPO is too slow, and the agent finds it difficult to explore more since it is too scary to take more risks. To encourage it to explore more, we use dynamic scaling entropy coefficient. The quantity *dynamic_scaling* [8] is calculated based on the mean of recent returns and be normalized by *dynamic_scaling_max* - the maximum expected reward. The formula is:

$$dynamic\_scaling = \frac{\mathbb{E}_t[R_t]}{dynamic\_scaling\_max}$$

The L_{Entropy} becomes:

$$L_{\text{Entropy}} = -\beta \cdot dynamic\_scaling \cdot \mathbb{E}_t[Entropy(Softmax(\text{Actor}(s_t)))]$$

Below is the hyperparameter set used for PPO training phase:

- Discount Factor $\gamma = 0.99$
- GAE-Lambda $\lambda = 0.95$
- $\alpha = 1$
- $\beta = 0.1$
- $dynamic\_scaling\_max = 900$

5 Experimental results

5.1 Policy comparision

After the training procedure, we capture some characteristics of three policy types. The Linear policy struggles with significant instability, as evidenced by high score variance and frequent sharp drops. Although the mean scores gradually improve after extended training, the performance plateaus below its potential, reflecting the policy's limitations in achieving consistent and stable results.

In contrast, the CNN policy demonstrates even greater instability, with frequent sharp drops in scores. While the mean scores show slight improvement over time, they plateau around 0, indicating poor generalization and an inability to achieve consistent performance. The instability and fluctuations highlight the CNN policy's struggle to learn effectively in the environment.

The Combined Policy, however, combines the strengths of both approaches, resulting in a more stable learning process. Scores improve steadily with fewer severe drops than the CNN-only policy, and the mean scores show a clear upward trend. This demonstrates better overall learning, improved stability, and higher consistency, highlighting the benefits of adding a linear layer to enhance performance and mitigate the instability observed in the other policies.

We displays the score graphs of algorithms in the figures below. Note that, the figures of Linear, CNN and Combined Policy are represented from left to right.

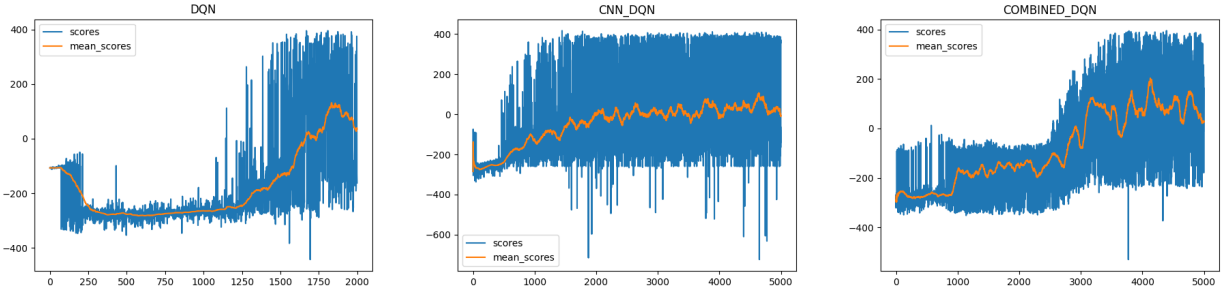

Figure 8: The scores of different policies using DQN.

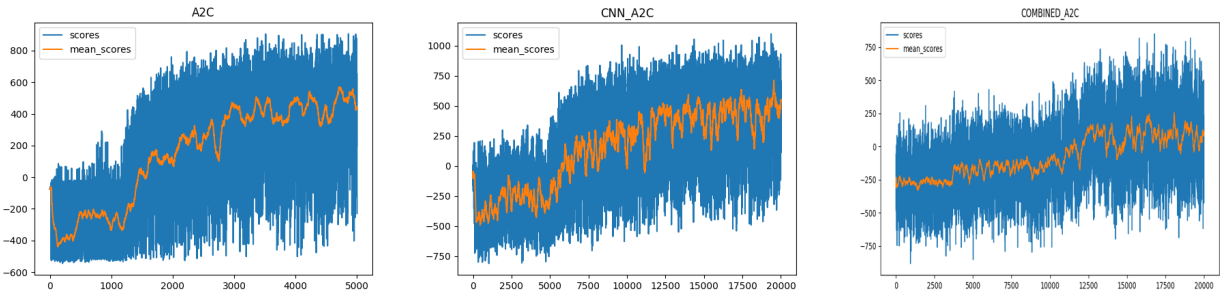

Figure 9: The scores of different policies using A2C.

5.2 Algorithm comparison

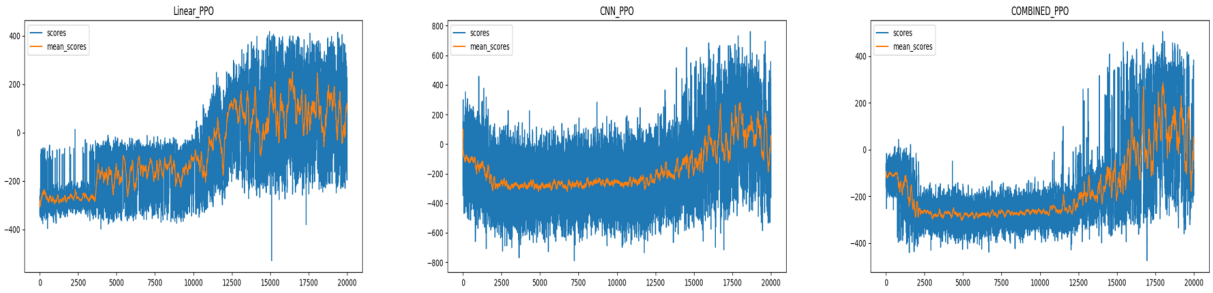

Figure 10: The scores of different policies using PPO.

Feature	**DQN**	**A2C**	**PPO**
Type	Value-based	Actor-Critic	Policy-based
Action Space	Discrete	Discrete/ Continuous	Discrete/ Continuous
On/Off-Policy	Off-Policy	On-Policy	On-Policy
Sample Efficiency	Moderate	Moderate	Moderate
Stability	Moderate	Moderate	High
Exploration	ϵ - greedy	Stochastic Policies	Stochastic Policies
Strength	Simplicity	Balance of Actor/Critic	Stability and Scalability
Limitation	Limited to Discrete Actions	Requires Balancing	High Computational Cost

Table 2: Comparison of DQN, A2C, and PPO

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015, ISSN: 1476-4687. DOI: 10.1038/nature14539. [Online]. Available: https://doi.org/10.1038/nature14539.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25, Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[3] O. Russakovsky, J. Deng, H. Su, *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, Dec. 2015, ISSN: 1573-1405. DOI: 10.1007/s11263-015-0816-y. [Online]. Available: https://doi.org/10.1007/s11263-015-0816-y.

[4] D. H. Hubel and T. N. Wiesel, "Receptive fields and functional architecture of monkey striate cortex," *The Journal of Physiology*, vol. 195, no. 1, pp. 215–243, 1968. DOI: https://doi.org/10.1113/jphysiol.1968.sp008455. eprint: https://physoc.onlinelibrary.wiley.com/doi/pdf/10.1113/jphysiol.1968.sp008455. [Online]. Available: https://physoc.onlinelibrary.wiley.com/doi/abs/10.1113/jphysiol.1968.sp008455.

[5] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, no. 4,

pp. 193–202, Apr. 1980, ISSN: 1432-0770. DOI: 10.1007/BF00344251. [Online]. Available: https://doi.org/10.1007/BF00344251.

[6] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017. arXiv: 1707.06347. [Online]. Available: http://arxiv.org/abs/1707.06347.

[7] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," Jun. 2015. DOI: 10.48550/arXiv.1506.02438.

[8] A. Lixandru, *Proximal policy optimization with adaptive exploration*, 2024. arXiv: 2405.04664 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2405.04664.