Reinforcement Learning: Trying different training algorithms on Open-AI Gym Cart-Pole environment

Antonis Zikas (sdi2100038), **Panagiotis Papapostolou** (sdi2100142) — DIT University of Athens, Greece

Reinforcement Learning & Stochastic Games

Fall Semester 2024

***Abstract* — *This article is part of an assignment for the Reinforcement Learning & Stochastic Games course at the Department of Informatics and Telecommunications, University of Athens. The primary objective of this work is to investigate the performance of different training algorithms in enabling a machine learning agent to interact effectively with a given environment. The experimental environment used in this study is the* `CartPole-v1` **task from the** `Classic Control` **category of OpenAI Gym.**

KEYWORDS. Reinforcement Learning, Machine Learning, Cart-Pole, DQN, Neural Networks

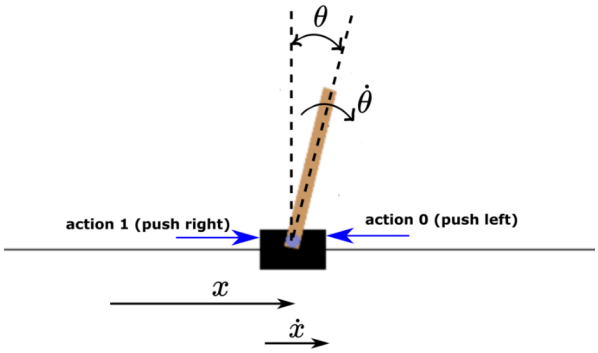


Figure 1: CartPole Environment Structure. Action & Observation Space described visually.

1 Environment Exploration

1.1 The structure of the environment

As previously mentioned, the environment under consideration is the `CartPole-v1`, which belongs to the `Classic Control` category of environments in OpenAI Gym. In this section, we provide a more detailed exploration of its structure, which is also represented in Figure 1.

The action space of the CartPole environment consists of exactly two discrete actions: the cart can either move **left** or **right**. These actions are represented by the integers **0** and **1**, respectively, within the OpenAI Gym framework. Formally, the action space can be expressed as a finite set:

$$A = \{0, 1\}$$

where 0 denotes a leftward movement of the cart and 1 denotes a rightward movement.

Observation	Array Index	Min	Max
Cart Position	0	-4.8	4.8
Cart Velocity	1	$-\infty$	$+\infty$
Pole Angle	2	-24°	24°
Pole Angular Velocity	3	$-\infty$	$+\infty$

Table 1: Observation space of the CartPole-v1 environment.

In addition, the observation space of the environment is a continuous state space consisting of four variables:

1. the horizontal position of the cart,
2. the velocity of the cart,
3. the angle of the pole relative to the vertical axis, and
4. the angular velocity of the pole.

We denote this observation space by S, where each element $s \in S$ is subject to specific lower and upper bounds. These limits are summarized in Table 1.

At each step during the training of the model, the selected agent receives a reward of `+1`, including the termination step. The reward threshold for this environment is `500`.

1.2 Default scores of the environment

As an initial step in this study, we evaluated the environment without applying any training procedure. Specifically, the environment was executed 200 times, and the results are presented in Figure 2. As observed, the total score obtained is very low, with a maximum value of approximately 70, yielding zero wins and 200 failures. Consequently, our objective is to train a model capable of increasing the total score to reach the environment's reward threshold of 500.

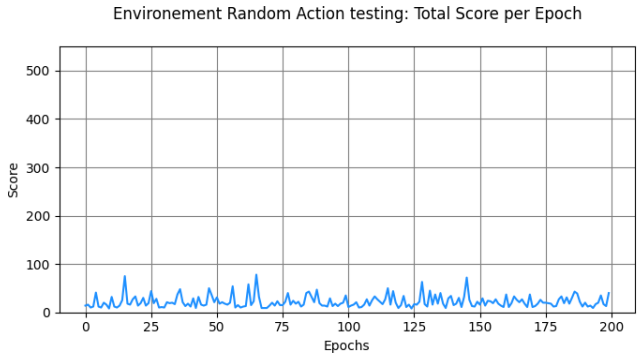


Figure 2: Default scores of the CartPole environment without any training applied to it

2 The DQN Algorithm

2.1 How DQN works

Our first implementation is based on the Deep Q-Network (DQN) algorithm [1]. The key idea of DQN is to approximate the action-value function $Q(s,a)$ with a deep neural network, thereby enabling reinforcement learning in environments with high-dimensional state spaces.

The algorithm makes use of two neural networks with identical architectures, illustrated in Figure 3. These are referred to as the *policy network* and the *target network*, denoted as $Q(s,a;\theta)$ and $Q_{\text{target}}(s,a;\theta^-)$, respectively. The policy network is updated continuously during training, while the target network is updated less frequently, typically every fixed number of steps (e.g., every $C = 10$ episodes). This stabilization technique helps to mitigate divergence in Q-learning.

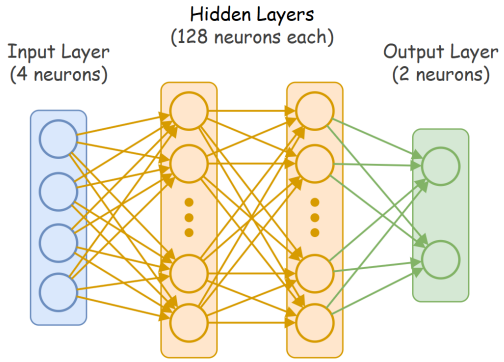


Figure 3: Policy and Target networks architecture for the DQN algorithm

At each time step, the agent selects an action using the ϵ-greedy exploration strategy:

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

Algorithm 1: Deep Q-Network (DQN)

Initialize replay buffer $\mathcal{D}$, policy network $Q(s,a;\theta)$, and target network $Q_{\text{target}}(s,a;\theta^-)$ with $\theta^- = \theta$
for *each episode* **do**
  Initialize state s
  for *each step in the episode* **do**
    Select action a using ϵ-greedy policy
    Execute a, observe r, s', and terminal flag d
    Store (s,a,r,s',d) in buffer $\mathcal{D}$
    Sample minibatch from $\mathcal{D}$
    Compute targets y using Equation (2)
    Update θ by minimizing loss in Equation (3)
    Every C steps update target network: $\theta^- \leftarrow \theta$
    Set $s \leftarrow s'$
    if $d = 1$ **then**
      break

where ϵ is the exploration parameter. Initially, $\epsilon = 1$, ensuring purely random exploration. As training progresses, ϵ is decayed gradually until it reaches a predefined lower bound (e.g., $\epsilon_{\min} = 0.01$), encouraging more exploitation of the learned policy.

To improve sample efficiency and reduce correlation between consecutive updates, DQN employs an *experience replay buffer*. This buffer stores tuples (s,a,r,s',d), where s and s' denote the current and next states, a is the action taken, r the reward received, and $d \in \{0,1\}$ indicates whether the episode terminated. Mini-batches are sampled uniformly from this buffer to update the policy network.

The target for the Q-value update is computed as:

$$y = \begin{cases} r, & \text{if } d = 1, \\ r + \gamma \max_{a'} Q_{\text{target}}(s',a';\theta^-), & \text{otherwise}, \end{cases} \tag{2}$$

where $\gamma \in [0,1]$ is the discount factor. The policy network parameters θ are optimized by minimizing the mean squared error (MSE) loss between the predicted Q-value and the target using Formula (3). The overall procedure is summarized in Algorithm 1.

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s',d)\sim\mathcal{D}}\Big[\big(y - Q(s,a;\theta)\big)^2\Big]. \tag{3}$$

2.2 Experimentation

2.2.1 Code Organization. Having analyzed the functionality of the DQN algorithm, we now proceed to its implementation in our codebase. To maintain clarity and modularity, we organized the implementation into four main components:

1. **Neural Network:** This module defines the architecture of the Q-network, which follows the design illustrated in Figure 3.
2. **Replay Memory Buffer:** This module implements the experience replay mechanism. It is based on a `deque`

data structure that stores transitions collected during training, i.e., (s, a, r, s', d) tuples.

3. **Agent:** This module encapsulates the behavior of the learning agent. It includes the ϵ-greedy exploration strategy and the optimization procedure based on the update rules described in Equations (2) and (3).
4. **Trainer:** The trainer orchestrates the training loop. Its primary role is to train the agent by interacting with the environment, following the procedure outlined in Algorithm 1.

2.2.2 Default Training & Testing DQN. The agent was initialized with the following hyperparameters:

- Discount factor: $\gamma = 0.99$
- Learning rate: $\alpha = 1 \times 10^{-3}$
- Batch size: 64
- Initial exploration rate: $\epsilon_{\text{start}} = 1.0$
- Final exploration rate: $\epsilon_{\text{end}} = 0.01$
- Exploration decay: $\epsilon_{\text{decay}} = 0.995$
- Target network update frequency: $C = 10$

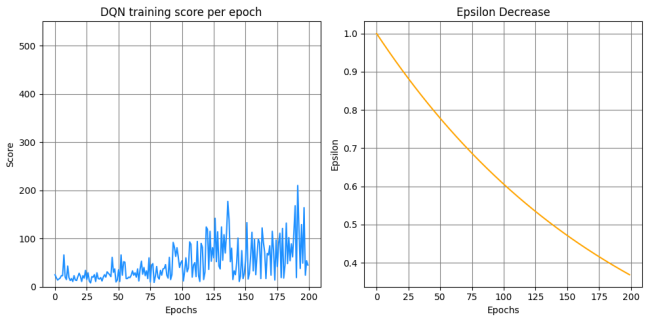


Figure 4: Training results of the DQN agent. The left plot shows the episodic return across training epochs, while the right plot illustrates the decay of the exploration rate ϵ.

We trained the agent in the specified environment for 200 epochs. The results are shown in Figure 4.

As shown in Figure 4, the left plot presents the total reward obtained by the agent in each episode, while the right plot illustrates the evolution of the exploration parameter ϵ throughout training. The reward curve exhibits an overall increasing trend, consistent with the expected improvement in performance. However, it also displays significant variance, which prevents a smooth and consistent increase. This noise is largely attributable to the ϵ-greedy policy: since actions are selected randomly with probability ϵ, the agent occasionally chooses suboptimal actions, leading to fluctuations in the reward signal.

The right plot demonstrates that ϵ decreases gradually during training, following a convex trajectory from its initial value of 1.0 down to approximately 0.4. This relatively high final value implies that the agent still selects random actions with probability 40% even after 200 epochs. Such a high exploration rate can hinder convergence and limit performance. A potential remedy is to extend the number of training epochs or to adjust the decay schedule of ϵ so that it converges closer to $\epsilon_{\text{end}} = 0.01$, thereby reducing the probability of random action selection.

2.2.3 Training with More Epochs. As mentioned previously, the next step was to train the DQN agent for 500 epochs. The results are shown in Figures 5 and 6. During training, the episodic return only slightly improved, barely surpassing the 200 mark. This limited progress can once again be attributed to the ϵ-greedy exploration strategy. Compared to Figure 4, we observe that ϵ decreases further, reaching a minimum value of 0.08 instead of 0.4.

These results are also reflected in the evaluation phase. As shown in Figure 6, the DQN agent trained for 500 epochs only manages to achieve 5 wins out of 200 episodes, highlighting the limited effectiveness of the training procedure under the current hyperparameter configuration.

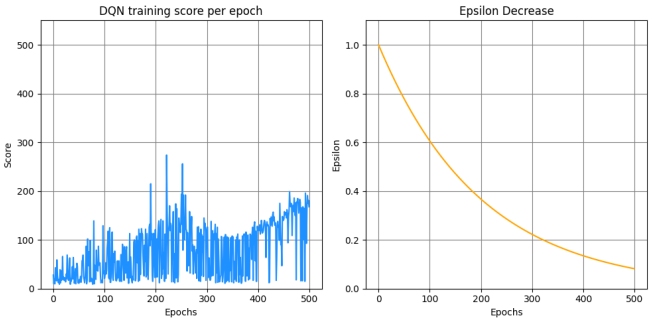


Figure 5: Training results of the DQN agent with 500 epochs. The left plot shows the episodic return across training epochs, while the right plot illustrates the decay of the exploration rate ϵ.

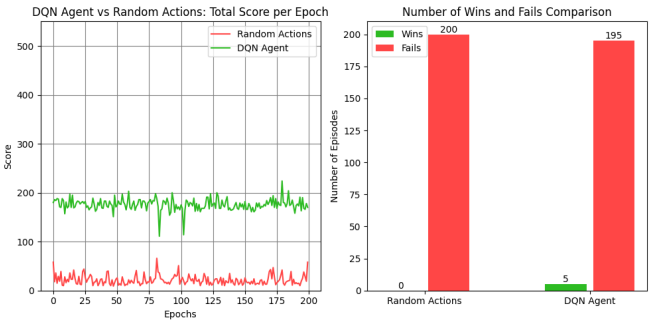


Figure 6: Evaluation results of the DQN agent compared to a random policy. The left plot presents the scores obtained over 200 episodes, while the right plot summarizes the number of wins and losses for both models.

2.2.4 DQN Sensitivity Study. After training the DQN agent with standard hyperparameters and obtaining baseline results, we conducted a **sensitivity study** to examine how variations in key hyperparameters affect performance. Specifically, we

tested the learning rate, discount factor, replay buffer size, and batch size. The values used for each parameter are shown in Table 2.

Learning Rate	1×10^{-4}	5×10^{-4}	1×10^{-3}	5×10^{-3}
Discount Factor	0.8	0.9	0.99	0.999
Replay Buffer Size	1,000	5,000	10,000	50,000
Batch Size	32	64	128	256

Table 2: Hyperparameter values used for the sensitivity study.

The agent was retrained for each combination of these hyperparameter values. The results are presented in Figure 7, where each subplot corresponds to one hyperparameter and the curves represent the performance obtained under different values. Overall, the results indicate that variations in these hyperparameters produce only minor differences, as the training curves follow very similar patterns.

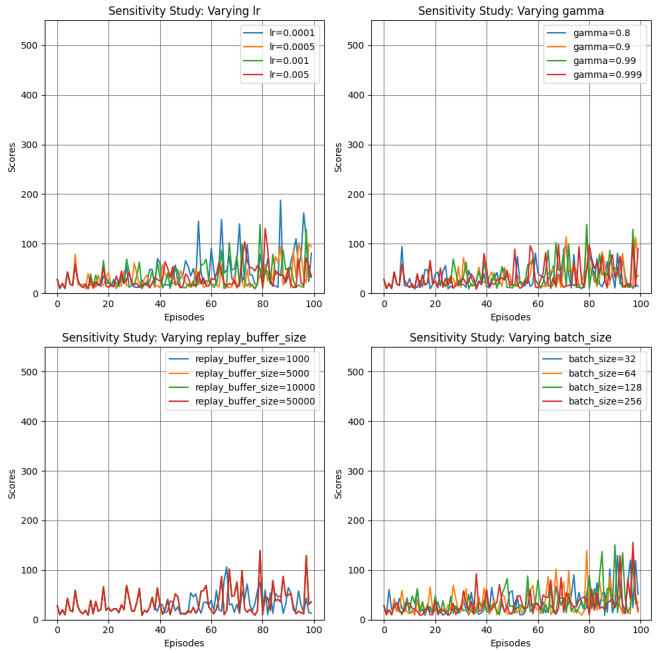


Figure 7: Sensitivity study results for the DQN agent. Each subplot corresponds to one hyperparameter, and the different curves within each subplot illustrate the effect of varying its values.

3 Dueling Architecture

3.1 How Dueling Architecture works

After experimenting with the DQN algorithm, we proceeded to a more advanced technique, specifically the Dueling DQN Architecture. This method affects the neural network architecture of the algorithm and specifically its last part as described in Figure 8. Instead of directly returning $Q(s,a)$, the network splits into two separate streams:

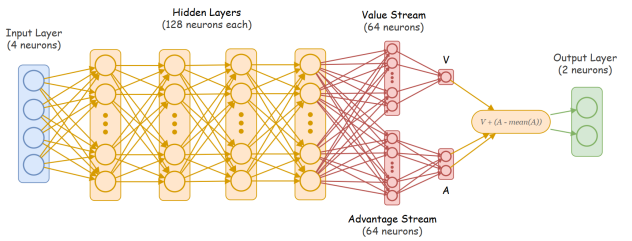


Figure 8: Structure of the Dueling Architecture Neural Network Model.

1. **Value Stream**: Estimates the value of being in a state $V(s)$
2. **Advantage Stream**: Estimates the advantage of each action compared to the average action in that state

$$A(s,a) = Q(s,a) - V(s)$$

After splitting the network and adding those two streams, in the last part they are combined back into Q-values, according to Formula 4

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|A|} \sum_{a'} A(s,a') \right) \tag{4}$$

3.2 Expirimentation

The code organization for the Dueling architecture is identical to the default DQN algorithm, with the exception of the Q-network, which was adapted to fit the above description.

3.2.1 Training & Testing Dueling DQN. The hyperparameters were initialized exactly the same as DQN:

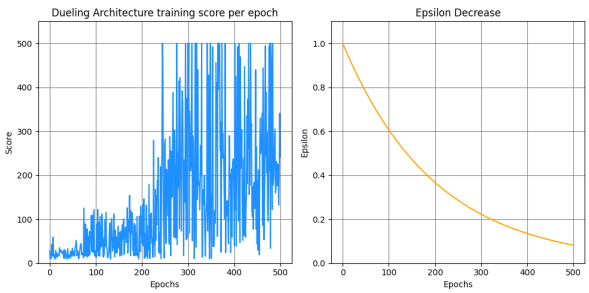


Figure 9: Training results of the Dueling DQN agent with 500 epochs. The left plot shows the episodic return across training epochs, while the right plot illustrates the decay of the exploration rate ϵ.

By observing the curve on Figure 9 and comparing it with Figure 5, we notice a significant leap in performance after the first 250 epochs, with the Dueling model almost doubling the score of the default one, even though ϵ seems to be decaying at the same rate. This of course can be attributed to the separation of Q into two different streams, allowing the model to

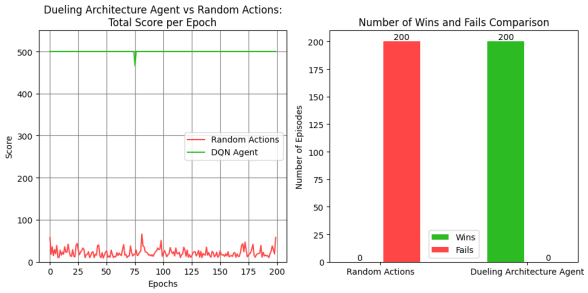


Figure 10: Evaluation results of the Dueling DQN agent compared to a random policy. The left plot presents the scores obtained over 200 **episodes, while the right plot summarizes the number of wins and losses for both models.**

take more educated and efficient actions, as it is also shown in the testing section in Figure 10, where the Dueling agent wins every single run.

3.2.2 Dueling Architecture Sensitivity Study. Following the same procedure, we conducted a sensitivity study using the Dueling Architecture with the same set of hyperparameter values. The results are presented in Figure 11. Similar to the standard DQN case, the training curves exhibit no substantial differences across hyperparameter choices, as all runs follow nearly identical patterns.

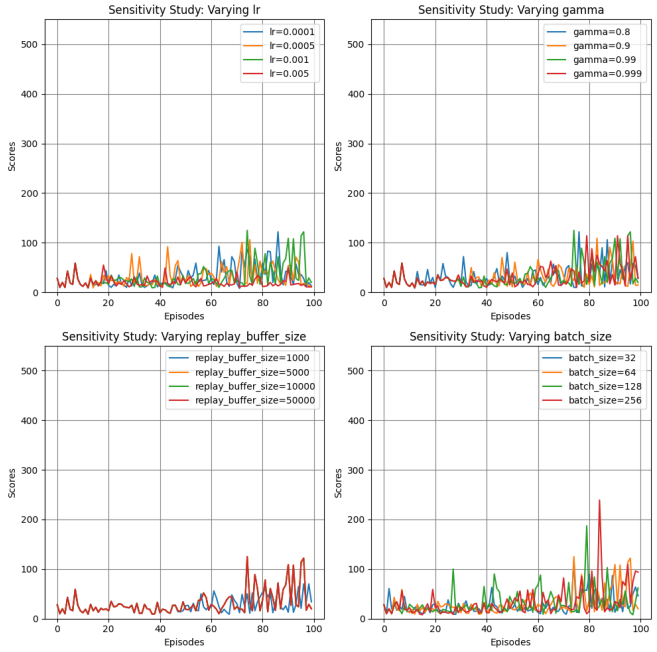


Figure 11: Sensitivity study results for the Dueling Architecture DQN agent. Each subplot corresponds to one hyperparameter, and the different curves within each subplot illustrate the effect of varying its values.

4 Transformers

4.1 How Transformers Work

Our next point of interest is the **Transformer** model and its potential application to the CartPole environment. Transformers are a class of neural networks originally designed for sequence modeling in *Natural Language Processing* (NLP), but they can also be adapted to reinforcement learning by handling sequences of states and actions. The core idea behind Transformers is the mechanism of **self-attention**, which enables the model to attend to all elements of a sequence rather than focusing solely on the most recent input, as in recurrent models. Their architecture [3] is described in Figure 12.

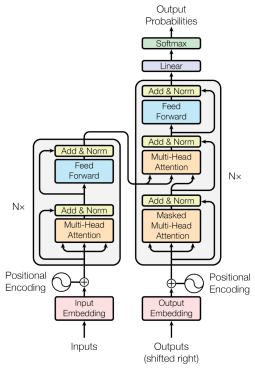


Figure 12: Sensitivity study results for the Dueling Architecture DQN agent. Each subplot corresponds to one hyperparameter, and the different curves within each subplot illustrate the effect of varying its values.

Suppose we have a sequence of T input vectors:

$$X = [x_1, x_2, \ldots, x_T], \quad x_t \in \mathbb{R}^d,$$

where each x_t represents a **state–action** pair. Each x_t is linearly projected into three representations:

- Queries: $Q = XW^Q$
- Keys: $K = XW^K$
- Values: $V = XW^V$

with $W^Q, W^K, W^V \in \mathbb{R}^{d \times d_k}$ denoting trainable weight matrices. The self-attention mechanism is then computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V.$$

Here, the dot product $QK^\top$ measures the similarity between queries and keys, the scaling factor $\sqrt{d_k}$ prevents exploding gradients, the softmax normalizes scores into probabilities, and multiplication with V produces a weighted sum of information. Consequently, each element of the sequence is updated based on all other elements.

In practice, Transformers use **multi-head attention**. Instead of computing a single (Q, K, V) triplet, h independent attention heads are used:

$$\text{MultiHead}(X) = \text{Concat}(head_1, \ldots, head_h)W^O,$$

where each head captures different types of dependencies.

Additionally, each position is processed through a **position-wise feedforward network (FFN)**:

$$\text{FFN}(x) = \max(0,\ xW_1 + b_1)W_2 + b_2.$$

By stacking layers of multi-head attention and feedforward networks, Transformers are capable of capturing complex dependencies across sequences of arbitrary length.

4.2 Applying Transformers to CartPole

The Transformer model can be adapted to the CartPole training procedure by considering not only the current state, but the entire sequence of past **states, actions, and rewards**. At time step t, this sequence can be written as

$$Z_t = \{(s_1, a_1, r_1), (s_2, a_2, r_2), \ldots, (s_t, a_t, r_t)\}.$$

Each triplet is embedded into a vector representation. The input embedding at position τ is defined as

$$x_\tau = E_s(s_\tau) + E_a(a_\tau) + E_r(r_\tau) + p_\tau,$$

where E_s, E_a, and E_r are the embedding functions for states, actions, and rewards respectively, and p_τ is a positional encoding that preserves sequence order.

The sequence of embeddings is then processed by the Transformer encoder:

$$h_\tau = \text{Transformer}(x_1, x_2, \ldots, x_\tau),$$

producing contextualized hidden states h_τ that incorporate information from the entire trajectory.

Finally, the hidden state at the current time step is used to estimate the action-value function:

$$Q(s_t, a_t) = h_t W^Q,$$

where W^Q is a learnable projection matrix. This formulation enables the agent to condition its decision not only on the present observation, but also on the broader sequence of past interactions.

4.3 Experimentation

4.3.1 Code Organization. While the modular structure that was described in the previous two models remains the same, the Transformer architecture requires making changes to those modules in order for it to work:

1. **Neural Network:** An encoding layer is introduced in order to process the action sequences which the transformer use to inform its future actions.
2. **Replay Memory Buffer:** The main change here is that instead of storing individual actions, we now store sequences of actions, the length of which is decided by a parameter.

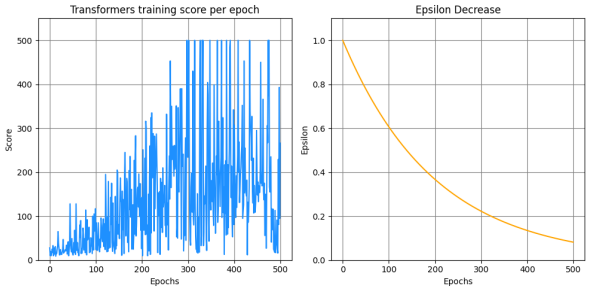


Figure 13: Training results of the Transformer agent with 500 epochs. The left plot shows the episodic return across training epochs, while the right plot illustrates the decay of the exploration rate ϵ.

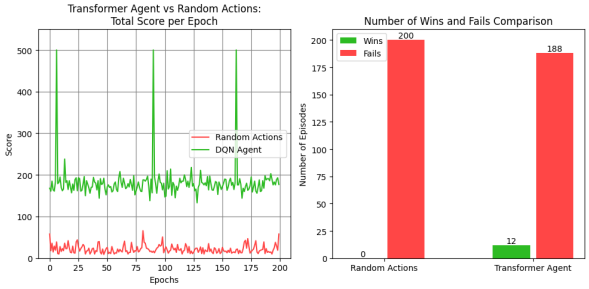


Figure 14: Evaluation results of the Transformer agent compared to a random policy. The left plot presents the scores obtained over 200 episodes, while the right plot summarizes the number of wins and losses for both models.

4.3.2 Training & Testing Transformer. The hyperparameters in this case are the same as before, with the addition of the length of the action sequence, which was initialized at 10:

Comparing the Transformer and Dueling, we notice their training performance is quite similar, with the Transformer agent being slightly worse overall. The testing results emphasize that difference even more, as shown in Figure 14, putting the Transformer's performance in testing just barely above that of the default DQN model. The only explanation we have for this result is that the model perhaps over complicates the problem by evaluating sequences of actions, thus leading to highly inconsistent performance. It is also worth noting that, while the training time of the previous two models was pretty similar(5 minutes for the default DQN and 8 for the Dueling), the Transformer model took more than both of them combined (around 15 minutes), so even if its results where equally as good as those of the Dueling architecture, it wouldn't be a preferable option due to its training time.

4.3.3 Transformer Sensitivity Study. The sensitivity study conducted here is largely similar to those of the two prior models, with the key difference being that the batch size hyperparameter has being changed in favor of the sequence length, since its a parameter more unique to this specific architecture (bottom right plot in Figure 15).

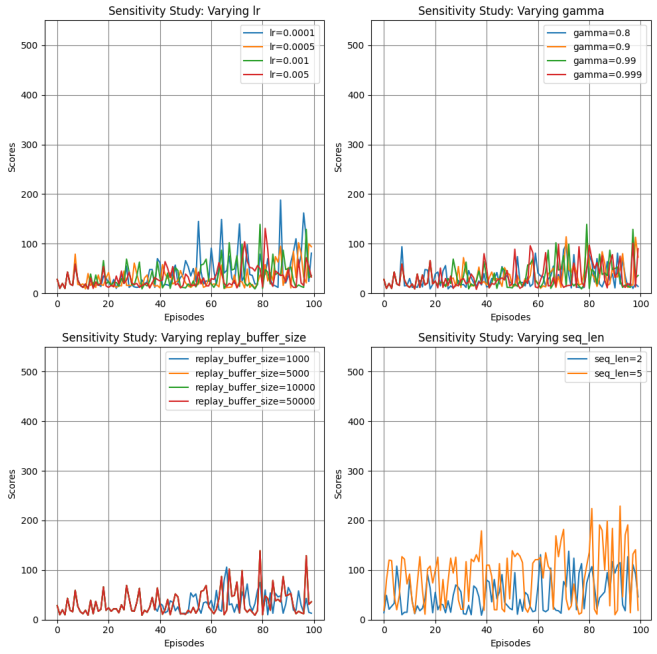


Figure 15: Sensitivity study results for the Transformer agent. Each subplot corresponds to one hyperparameter, and the different curves within each subplot illustrate the effect of varying its values.

5 A2C & PPO

5.1 How A2C and PPO Work

Our final training experiments focus on the PPO (Proximal Policy Optimization) and A2C (Advantage Actor-Critic) algorithms, implemented using the **Stable Baselines 3** library. Both methods belong to the family of **policy gradient algorithms**, which directly optimize a parameterized policy $\pi_\theta(a|s)$. The goal is to maximize the expected return:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty}\gamma^t r_t\right].$$

According to the policy gradient theorem, the gradient of this objective is

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s)\, Q^\pi(s,a)\right].$$

In practice, instead of the action-value function Q^π, we often use the **advantage function**:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s),$$

which measures the relative quality of an action compared to the expected value of the state.

5.1.1 A2C (Advantage Actor-Critic).

A2C is the synchronous version of A3C. The *actor* represents the policy $\pi_\theta(a|s)$, while the *critic* is a value function $V_\phi(s)$ that estimates the state value. The algorithm is described in 2

Algorithm 2: Advantage Actor-Critic (A2C)

Initialize policy network $\pi_\theta(a|s)$ and value network $V_\phi(s)$
for *each episode* **do**
  Initialize state s
  for *each step in the episode* **do**
    Sample action $a \sim \pi_\theta(\cdot|s)$
    Execute a, observe r, s', and terminal flag d
    $A_t = r + \gamma V_\phi(s') - V_\phi(s)$
    $\theta \leftarrow \theta + \alpha_\pi \nabla_\theta \log \pi_\theta(a|s) A_t$
    $\phi \leftarrow \phi - \alpha_V \nabla_\phi \big(V_\phi(s) - (r + \gamma V_\phi(s'))\big)^2$
    Set $s \leftarrow s'$
    if $d = 1$ **then**
      break

Algorithm 3: Proximal Policy Optimization (PPO)

Initialize policy $\pi_\theta(a|s)$ and value function $V_\phi(s)$
for *each iteration* **do**
  Collect trajectories $\{(s_t, a_t, r_t)\}$ by running π_θ
  Compute returns R_t and advantages A_t using GAE or TD estimates
  for *each minibatch from trajectories* **do**
    $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$
    $L^{\text{CLIP}}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t\right)\right]$
    $\theta \leftarrow \theta + \alpha_\pi \nabla_\theta L^{\text{CLIP}}(\theta)$
    $\phi \leftarrow \phi - \alpha_V \nabla_\phi (V_\phi(s_t) - R_t)^2$
  Set $\pi_{\theta_{\text{old}}} \leftarrow \pi_\theta$

The critic is trained by minimizing the value loss:

$$L_V(\phi) = \big(V_\phi(s_t) - (r_t + \gamma V_\phi(s_{t+1}))\big)^2,$$

which corresponds to regression on the Bellman target.

The actor is updated using the advantage:

$$L_\pi(\theta) = -\mathbb{E}_t\big[\log \pi_\theta(a_t|s_t)\, A_t\big],$$

with

$$A_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t).$$

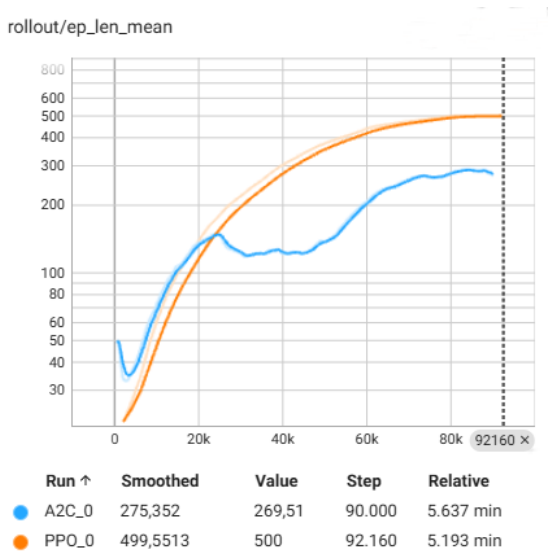
This formulation encourages the actor to increase the probability of actions with positive advantage and decrease the probability of those with negative advantage.

To promote exploration, an entropy bonus is added:

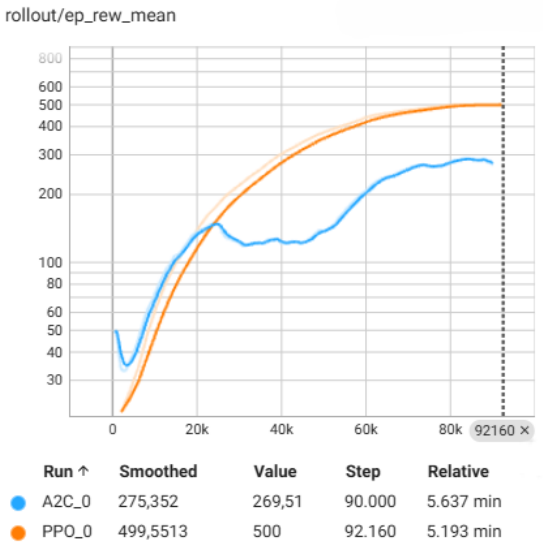
$$L_{\text{entropy}} = \beta\, H\big(\pi_\theta(\cdot|s_t)\big).$$

Thus, the final A2C loss combines these components:

$$L(\theta,\phi) = L_\pi(\theta) + c_1 L_V(\phi) - c_2 L_{\text{entropy}}.$$



(a) Average episode length. Orange: PPO, Blue: A2C.



(b) Average reward. Orange: PPO, Blue: A2C.

Figure 16: Training results of PPO and A2C over 90,000 timesteps averaged across 9 models.

5.1.2 PPO (Proximal Policy Optimization). PPO is also an actor–critic algorithm [2], but it improves stability by preventing excessively large policy updates [Algorithm 3]. This is achieved by defining the probability ratio between the new and old policies:

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

This ratio measures how much the new policy deviates from the old one.

The PPO loss is defined as

$$L^{\text{PPO}}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)A_t,\ \text{clip}\big(r_t(\theta), 1-\epsilon, 1+\epsilon\big)A_t\right)\right].$$

If $r_t(\theta)$ remains close to 1, the update proceeds normally. However, if the ratio deviates outside the range $[1-\epsilon,\ 1+\epsilon]$, it is clipped to avoid destructive updates. This mechanism ensures **trust region-like updates** without the expensive optimization required in TRPO.

As with A2C, the total PPO loss also includes a value loss and an entropy bonus:

$$L(\theta, \phi) = L^{\text{PPO}}(\theta) + c_1 L_V(\phi) - c_2 L_{\text{entropy}},$$

where the value loss is

$$L_V(\phi) = \big(V_\phi(s_t) - R_t\big)^2.$$

5.2 Results and Comparison

We trained both algorithms for 90,000 timesteps, saving the model every 10,000. The training was implemented using **Stable Baselines 3**, and the results were visualized in **TensorBoard** (Figure 16 and Figure 17).

From the diagrams in Figure 16, we observe that the curves are nearly identical. This outcome is expected, since in the CartPole environment the episode length is directly proportional to the reward.

When comparing the two algorithms, **PPO** demonstrates superior performance. This can be attributed to its mechanism of constraining policy updates, preventing large deviations between successive policies. As a result, PPO achieves more stable and efficient learning. In contrast, **A2C** does not include such safeguards, and is therefore more prone to collapsing into suboptimal policies—an effect that we observed in our experiments.

References

[1] Jianqing Fan, Zhaoran Wang, Yuchen Xie, and Zhuoran Yang. A theoretical analysis of deep q-learning, 2019. Accessed: 2025-09-28.

[2] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Figure 17: Metadata of PPO and A2C training