Florida Atlantic University

Reinforcement learning

CAP6629

Project 3

Author:
Temirlan Kdyrkhan
Z23757665
tkdyrkhan2024@fau.edu

December 22, 2024

1 Title: Actor-Critic network on "Galaxian" Atari game.

This report includes project 3 details and analysis with included code explanation. For this project Option B was selected as implementation of Atari game named "Galaxian". We going to provide the Actor-Critic architecture and all the details about how we implemented the game learning. There will not be provided whole code in this PDF file.

Initially we chose "Tetris" game and tried 7 different architectures with Actor-Critic method and couldn't find the best solution, so "Galaxian" game will be discussed here.

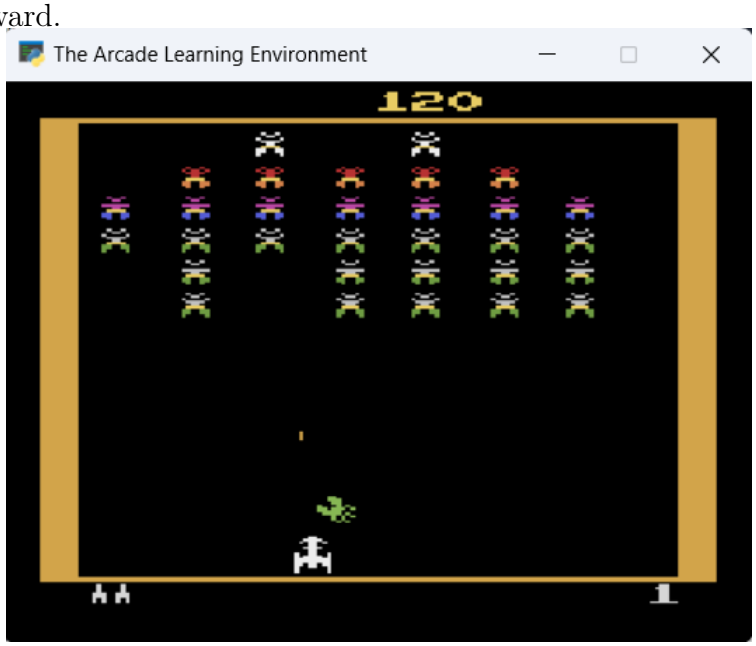
2 Introduction

Reinforcement Learning (RL) is a machine learning paradigm where an agent learns to make decisions by interacting with an environment to maximize cumulative rewards. The Actor-Critic method is a widely used architecture in RL that combines two fundamental approaches: policy-based (actor) and value-based (critic) methods. The actor predicts the optimal action policy, while the critic estimates the value function to evaluate the actor's choices. This dual-role architecture allows for efficient policy updates by leveraging the value function to calculate the advantage and gradients.

In this project, we implement the actor-critic method to train an agent to play the Atari game Galaxian. The goal is to optimize the agent's ability to maximize scores by managing the trade-off between immediate and delayed rewards.

3 Problem Formulation

The task involves training an RL agent to play Galaxian, a classic Atari game where the player controls a spaceship to shoot alien invaders. There is given 3 lives and 1 at the beginning with total 4 lives for each game. There are different types of rewards spaceship can get depending on the target. Out main goal is to make the agent to gain maximum reward.



3.1 State Representation

The state is represented as a series of frames capturing the current environment's visual information. These frames are resized and normalized to fit the input dimensions of the neural networks. Raw Frame Dimensions is (210, 160, 3), which contains a lot of unnecessary information (e.g., background details) that doesn't help the agent. Preprocessing is used to simplify the state representation and make it more meaningful for the agent. So we needed to use preprocessing like normalization and reorder the dimensions for PyTorch.

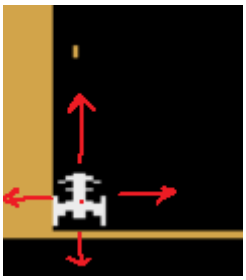
```
state = torch.tensor(state, dtype=torch.float32).unsqueeze(0).permute(0, 3, 1,
    2) / 255.0
```

The processed state is passed through a series of convolutional layers in both the Actor and Critic networks to extract spatial and semantic features.

3.2 Action Space

The action space consists of discrete actions available in the game:

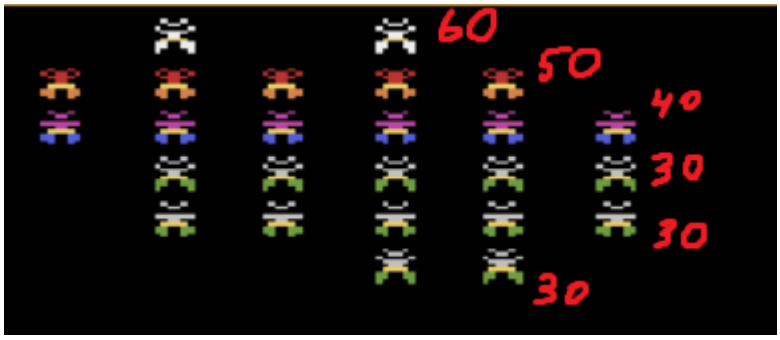
1. Move Left
2. Move Right
3. Shoot
4. Move left and Shoot
5. Move right and Shoot
6. None (means stay in latest position)



3.3 Reward Structure

The reward function is designed to encourage positive behaviors (e.g., hitting enemies) while penalizing negative outcomes (e.g., losing lives or game over):

1. Hit Reward: (hitting the enemy reward) * 0.05, where hitting the enemy reward ranging from 30 to 60 depending which enemy type our agent was able to hit.



2. Step Penalty: -0.01 for each time step to encourage efficiency.
3. Losing One Live Penalty: -5.0 when the agent loses one live.
4. Game Over Penalty: -10.0 when the agent loses all lives.

4 Method Design

4.1 Network Architecture, option A (actor-critic)

The implementation includes two separate neural networks, each having unique weight parameters:

1. **Actor Network**: Outputs a probability distribution over actions using a softmax activation.
2. **Critic Network**: Outputs a scalar value representing the estimated value of the state.

Both networks share a convolutional structure followed by fully connected layers:
Convolutional Layers:

1. **Layer 1**: 32 filters, kernel size 8x8, stride 4, ReLU.
2. **Layer 2**: 64 filters, kernel size 4x4, stride 2, ReLU.
3. **Layer 2**: 64 filters, kernel size 3x3, stride 1, ReLU.

Fully Connected Layers:

1. **Actor**: Fully connected layer (128 neurons) followed by an output layer with softmax activation.
2. **Critic**: Fully connected layer (128 neurons) followed by an output layer producing a scalar value.

```
...
class Actor(nn.Module):
    def __init__(self, action_dim, hidden_dim=128):
        super(Actor, self).__init__()
        ...
    def conv_layers(self, x):
```

```
        ...
    def forward(self, state):
        ...

class Critic(nn.Module):
    def __init__(self, hidden_dim=128):
        super(Critic, self).__init__()
        ...
    def conv_layers(self, x):
        ...
    def forward(self, state):
        ...
```

4.2 Network Architecture, option B (shared network)

Beyond the previous implementation there was implemented another architecture. This implementation uses a single shared network that branches into two distinct heads for actor and critic tasks. This architecture leverages shared feature extraction while maintaining separate outputs for policy and value estimation. It has the same architecture for the Deep NN where actor outputs a probability distribution over actions using a softmax activation and critic outputs scalar value representing the estimated value of the current state, however instead of using 2 different weight parameters, we experimented how the agent's results would show up.

```
class SharedActorCritic(nn.Module):
    def __init__(self, action_dim, hidden_dim=128):
        super(SharedActorCritic, self).__init__()
        ...
    def conv_layers(self, x):
        ...
    def forward(self, state):
        ...
```

4.3 Learning Process

The actor-critic method utilizes two loss functions:

$$ActorLoss = -log(\pi(a|s)) * A(s,a)$$

where $\pi(a|s)$ is the probability of taking action **a** in state **s**, and A(s,a) is the advantage function calculated as:

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

$$CriticLoss = (V(s) - TargetValue)^2$$

where Q(s,a) estimated by the critic and the target value is:

$$TargetValue = R(s,a) + \gamma * V(s^{'})$$

4.4 Weight Update Rules, option A (actor-critic)

Actor Network: Gradients are backpropagated using the actor loss, ensuring the policy probabilities are adjusted to maximize cumulative rewards. **Critic Network**: Backpropagation is applied to minimize the mean-squared error between the estimated and target values.

Forward Path

1. The state is passed through the actor and critic networks.
2. The actor predicts the probability distribution of actions.
3. The critic estimates the value of the state.

Backward Path

1. Advantage and target values are computed.
2. Actor and critic losses are calculated for both actor and critic.
3. Gradients are clipped and applied to update network weights of both actor and critic.

4.5 Weight Update Rules, option B (shared network)

Actor Network: Gradients are backpropagated using the actor loss, ensuring that the policy probabilities are adjusted to maximize cumulative rewards. **Critic Network**: Backpropagation is applied to minimize the mean-squared error (MSE) between the estimated and target values.

Forward Path

1. In shared Network the input state is passed through shared convolutional layers, generating features for both actor and critic heads.
2. Actor Outputs a probability distribution over actions using the softmax activation function.
3. Critic outputs a scalar value representing the value of the state.

Backward Path

Backward path is almost the same,but in the shared architecture, the actor and critic share a common network, resulting in a single combined backward pass. In contrast, the non-shared architecture requires separate backward passes for each network, with no shared parameters. Since both actor and critic losses are combined for a single backward pass it reduces computational overhead compared to Option A's independent updates.

5 Experiments

5.1 Option A (actor-critic)

Hyperparameters Learning Rate: 0.0001. Discount Factor (γ): 0.99. Exploration Rate ϵ: Starts at 1.0, decayed by 0.995 per episode, with a minimum value of 0.1. **Number of episodes = 696**.

Pseudo code:

```
initialize_actor_critic_networks()
initialize_optimizer()
for each_episode:
    reset_environment()
    initialize_state, done = env.reset(), False
    initialize_episode_reward, prev_lives = 0, total_lives
    while not done:
        predict_action_probabilities()
        select_action(epsilon_greedy_strategy)
        perform_action_and_observe_reward()
        compute_target_value_and_advantage()
        compute_actor_loss_and_critic_loss()
        update_network_weights()
        accumulate_episode_metrics()
    save_model_weights()
    log_episode_results()
save_final_metrics()
```

While working with any Atari games and trying different architecture I noticed that any simulation requires huge amount of data and time for the game to be trained, so when I was able to find the best meaningful method I was able to run it only with 696 episodes with option A.

5.2 Option B (shared network)

Hyperparameters Learning Rate: 0.0001. Discount Factor (γ): 0.99. Exploration Rate ϵ: Starts at 1.0, decayed by 0.995 per episode, with a minimum value of 0.1. **Number of episodes = 1000.**

```
initialize_shared_actor_critic_network()
initialize_optimizer()
for each_episode:
    reset_environment()
    initialize_state, done = env.reset(), False
    initialize_episode_reward, prev_lives = 0, total_lives
    while not done:
        predict_action_probabilities_and_state_value()
        select_action(epsilon_greedy_strategy)
        perform_action_and_observe_reward()
        compute_target_value_and_advantage()
        compute_combined_actor_critic_loss()
        update_shared_network_weights()
```

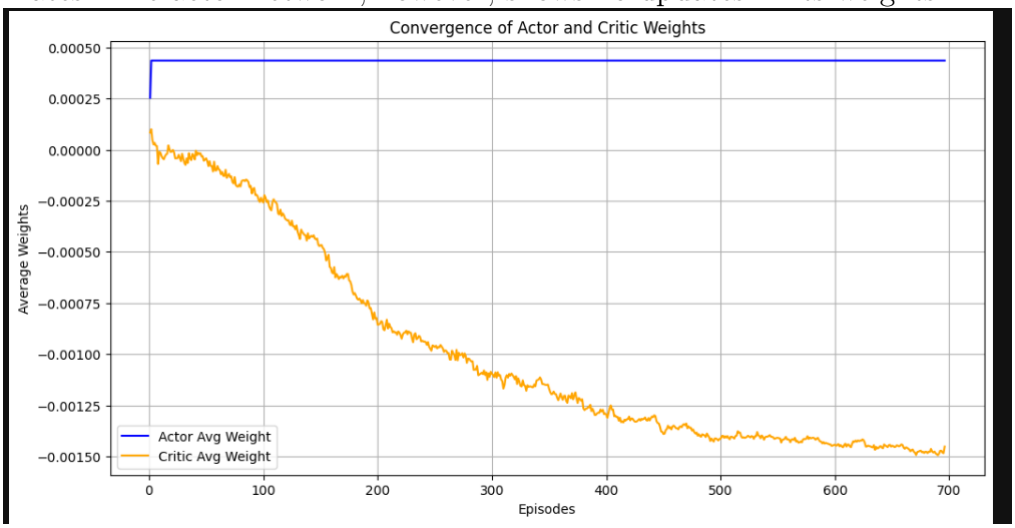
```
        accumulate_episode_metrics()

    save_shared_model_weights()
    log_episode_results()
save_final_metrics()
```

Option B, since there is only one network, worked much faster and I was able to make it run up to 1000 episodes.

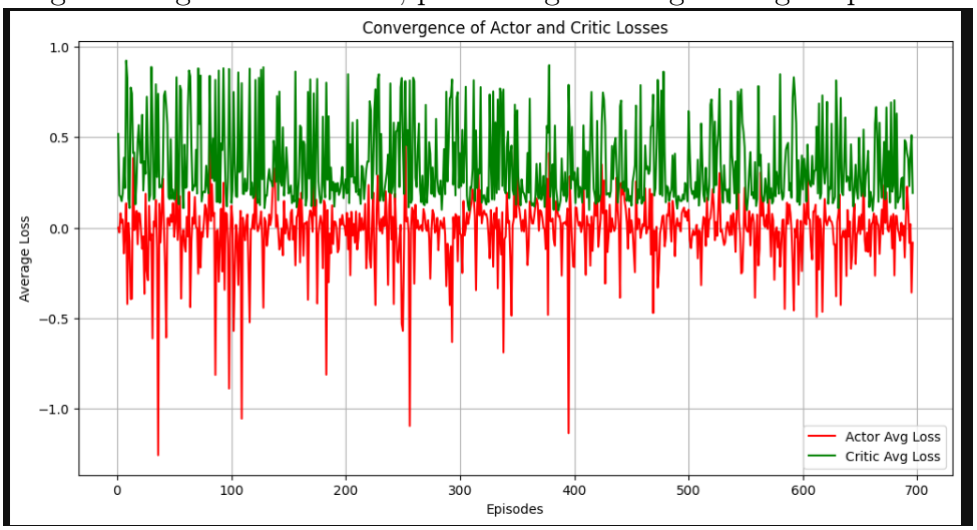
6 Results

6.1 Option A (actor-critic)

Results. The critic network is learning effectively, as evidenced by the consistent updates in its weights. The downward trend indicates the critic is converging towards stable value estimates. The actor network, however, shows no updates in its weights.



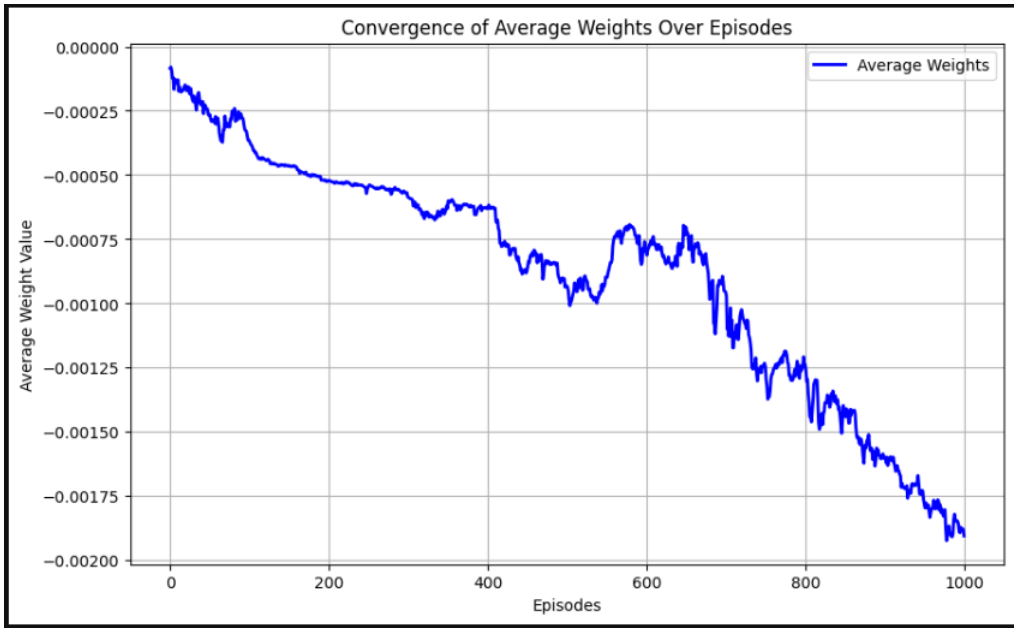
Potential Causes When working on project there were implemented many experiments where the agent didn't show any meaningful results. In this option A actor's learning rate might be too small, preventing meaningful weight updates.



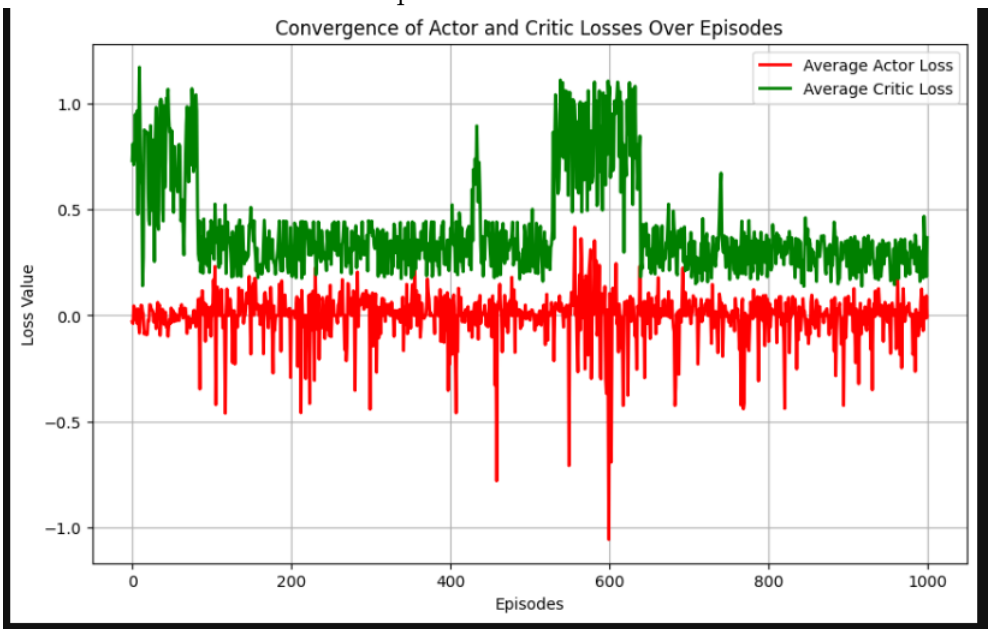
Loss Convergence. The graph shows the convergence of actor and critic losses over 700 episodes. The red curve represents the actor's average loss per episode, while the

green curve shows the critic's average loss. The actor's average loss (red curve) fluctuates widely, including consistent dips into the negative region before the 400 episodes. Later both actor and critic losses **converged over episodes**, indicating stable updates.

6.2 Option B (shared network)



The network weights exhibit a clear downward trend, stabilizing towards a negative value as training progresses. Suggesting that the network is adjusting its weight parameters to minimize the network's prediction error.



The actor loss fluctuates heavily around zero. Negative values indicate that the policy updates are improving (higher likelihood for advantageous actions), while positive spikes might suggest suboptimal policy updates or instability in policy learning. The critic loss stabilizes over time. Both have some noticeable spikes around episodes 500-600. These spikes might correspond to challenging game states or sudden changes, I suggest that around those episodes there were specific weight parameters that made the agent act in

specific way (for example the agent may stay in most left or most right corner of the game) and didn't have much rewards.

7 Challenges

1. In option A the actor network weights appear stagnant, potentially due to low learning rate or sparse rewards (rewards might be too infrequent, making policy updates inconsistent).

2. Option A requires more memory due to maintaining separate networks for actor and critic.

3. Sparse Rewards: Reward sparsity required careful tuning of reward parameters to guide the agent effectively.

4. Maintaining separate actor and critic networks doubles the number of parameters, increasing memory usage and computational cost.

5. The agent must react quickly to multiple simultaneous threats, making it difficult to associate specific actions with positive outcomes.

6. Losing lives penalizes the agent heavily, creating a steep gradient that may overpower the learning signal.

8 Observations on Results

This project explored and implemented two architectures for reinforcement learning: the shared and separate actor-critic networks. Each architecture demonstrated distinct advantages and trade-offs.

Separate Actor-Critic Network (Option A)

For weight Convergence Actor network weights appear stagnant due to a potentially low learning rate, whereas the critic network converges effectively. By maintaining separate networks, the actor and critic could specialize without interference from each other, leading to potentially better policies, however with **Higher Computational Cost and Synchronization Challenges**.

Shared Network (Option B)

The weights show a consistent downward trend over episodes, stabilizing to a negative value. This suggests that the shared network effectively optimizes its weights to minimize prediction errors.

Actor loss oscillates around zero, with spikes representing suboptimal updates or challenging game states. Critic loss stabilizes as training progresses. The spikes between episodes 500-600 might indicate specific game states influencing the network's predictions and actions.

References

[1] Reinforcement Learning: Deep Q-Learning with Atari games

[2] OpenAI Gym Documentation

[3] ALE (Arcade Learning Environment) Documentation

[4] Actor-Critic Reinforcement Learning for Breakout-v4 (Atari)