Deep Reinforcement Learning

1BM120, Decision making with artificial and computational intelligence - GS4 (2023)

Group 21

Full Name	**Student ID**
Sem Draaijer	1713523
Hebe Heij	1837923
Kevin Laemers	0848982
Lore Verstraete	1415492

Tutors: Dr. ir. Laurens Bliek (TU/e) – Dr. Zaharah Bukhsh (TU/e) - Luca Begnardi (TU/e)

Eindhoven, June 27, 2023

Contents

1 | Introduction

Reinforcement Learning (RL) is a powerful approach to solving complex optimization problems by training an agent to interact with an environment and learn optimal strategies through trial and error. This report addresses the Bounded Knapsack problem, a combinatorial optimization problem, using reinforcement learning techniques. The objective is to train an RL agent to solve the Bounded Knapsack problem with 200 items using the *Knapsack-v2* environment implemented in the OR-GYM library version 0.5.0. The environment is considered solved when the agent obtains an average reward of 2000 or above over 100 consecutive episodes. With optimal algorithm and parametric settings, the agent can achieve a reward higher than 2600 over 100 consecutive episodes.

1.1 | Environment

The *Knapsack-v2* environment represents the Bounded Knapsack problem. The RL agent's state is defined as a concatenation of vectors containing the weights, values, and limits of the available items, as well as the current load and maximum capacity of the knapsack. The number of actions is equal to the number of items available. At each timestep, the RL agent can select only one item. The reward is the total value of all items placed within the knapsack. An episode ends when the knapsack is full or no further items can be placed.

Two different Deep Reinforcement Learning (DRL) agents are trained using the Proximal Policy Optimization (PPO) and Advantage Actor-Critic (A2C) algorithms provided in the Stable Baselines3 package. These algorithms use neural networks as function approximators to learn the optimal policy for the *Knapsack-v2* environment.

This report demonstrates the effectiveness of RL techniques in solving combinatorial optimization problems such as the Bounded Knapsack problem. It includes experimentation with different neural network architectures, manual tuning of the algorithms' hyperparameters, and evaluation of the agents. The report provides insights into the performance of different algorithms and hyperparameter configurations in tackling this challenging problem, and compares the best results obtained using the different algorithms. I also shows the sensitive nature of RL algorithms and its hyperparameters.

2 | Reinforcement Learning agents

This section discusses the two chosen reinforcement learning algorithms, Proximal Policy Optimization (PPO) and Advantage Actor Critic (A2C), and the motivation behind their selection. It also details the hyperparameters tuned for each algorithm and their impact on the models' performances. The training process of the agents over the timesteps and the average accumulated reward is visualized and discussed. Finally, the highest gained rewards and the number of training timesteps for each agent are indicated.

2.1 | Proximal Policy Optimization

The first agent for solving the Bounded Knapsack problem is the Proximal Policy Optimization(PPO) agent.
PPO is a model-free, on-policy algorithm that utilizes a policy gradient approach for reinforcement learning. It has been shown to be robust and effective in various RL tasks. One of the key advantages of PPO is its ability to handle discrete action spaces, which makes it suitable for problems like the Bounded Knapsack, where the RL agent needs to select one item from a range of options.

The PPO algorithm has a strong exploration mechanism that allows it to discover better policies by encouraging exploration of the action space. This exploration is essential in the context of the Bounded Knapsack problem, as the agent needs to explore different combinations of items to find the optimal solution within the given weight limit.
Furthermore, PPO incorporates a policy update mechanism that ensures stable and efficient learning. It utilizes a surrogate objective function to optimize the policy while preventing large policy updates that can lead to instability.

Considering these factors, it is clear that PPO probably is well-suited for the Bounded Knapsack problem. Its ability to handle continuous action spaces, coupled with its robust exploration and stable learning mechanisms, makes it a suitable choice for training an RL agent to solve the Bounded Knapsack problem efficiently.

2.1.1 | Used parameters PPO

To optimize the results, the PPO agent is tuned. The best results emerged from the following hyper-paramters.

- `learning_rate = 3e-4`: The learning rate for the optimizer.
- `n_steps = 2048`: The number of steps to run for each environment per update. This is essentially the size of the rollout buffer.
- `batch_size = 64`: The size of the minibatch for each update. This is the number of samples used to estimate the gradient.
- `n_epochs = 10`: The number of epochs to use when optimizing the surrogate loss. An epoch is a complete pass through the entire training dataset.
- `gamma = 0.99`: The discount factor used in the calculation of the future rewards. This determines the importance of future rewards for the agent.
- `gae_lambda = 0.95`: The factor for trade-off of bias vs variance for the Generalized Advantage Estimator. This is used to balance the bias and variance of the advantage estimator.

The used network architecture is $[128, 128, 128, 128, 128]$, which indicates that the network has 5 layers with 128 nodes each.

2.1.2 | Results for PPO

The PPO agent achieved a reward of approximately 2963 in the last 100 episodes when applied to the Bounded Knapsack problem. The learning curve, as shown in Figure 2.1a, indicates that the agent improves its reward up until around 450 episodes, after which the improvements stagnate and the agent's optimal performance is reached.

2.2 | Advantage Actor Critic

The Advantage Actor-Critic (A2C) algorithm was chosen as the second agent for solving the Bounded Knapsack problem. A2C is a model-free, on-policy reinforcement learning algorithm that combines elements of policy-based and value-based methods.

A2C employs an actor-critic architecture, where the actor learns a policy to select actions, and the critic estimates the value function to evaluate the quality of actions. This actor-critic framework allows for more accurate value function estimation compared to simple policy gradient methods.
The on-policy nature of A2C enables continuous learning from interactions with the environment, ensuring that the agent adapts its policy based on the most recent experiences. This characteristic is particularly useful in the Bounded Knapsack problem, where the agent needs to make sequential decisions while considering the limited capacity of the knapsack.
By selecting A2C as the second agent, the aim of this report is to explore the performance of an actor-critic algorithm in solving the Bounded Knapsack problem. This choice allows for evaluation of the benefits of value function estimation and continuous learning in the context of combinatorial optimization.

Through the evaluation of A2C, insights can be gained into the strengths and limitations of actor-critic algorithms for solving the Bounded Knapsack problem and its performance can be compared with the PPO algorithm.

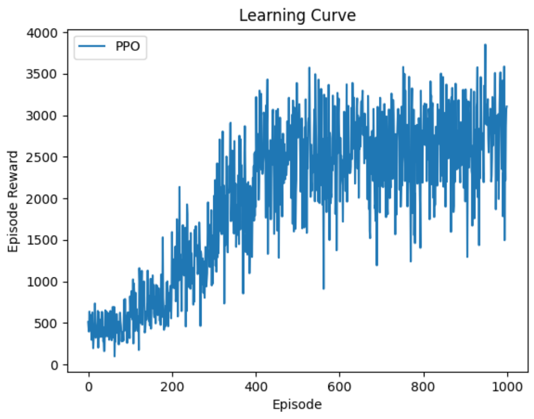
2.2.1 | Used parameters A2C

- `learning_rate = 4e-4`: The learning rate for the Adam optimizer.
- `n_steps = 25`: The number of steps to run for each environment per update.
- `max_grad_norm = 0.6`: The maximum value for the gradient clipping.
- `ent_coef = 0.01`: Entropy coefficient for the loss calculation.
- `vf_coef = 0.500`: Value function coefficient for the loss calculation.
- `gamma = 0.99`: Discount factor.
- `gae_lambda = 1`: Factor for trade-off of bias vs variance for Generalized Advantage Estimator.
- `use_rms_prop = False`: This indicates that Adam is used as optimizer, instead of RMSprop.

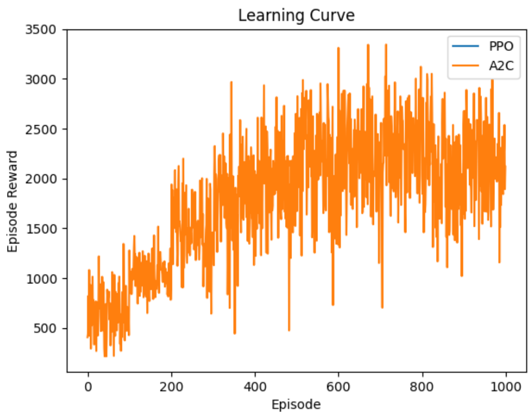
For the A2C agent a network architecture of $[128, 64, 64, 64, 64, 64, 128]$ gave the best results.

2.2.2 | Results A2C

In Figure 2.1b the reward per episode is shown for the A2C agent. The A2C agent achieved a reward of approximately 2000 in the last 100 episodes when applied to the Bounded Knapsack problem. The learning curve indicates that the agent improves its reward up until around 600 episodes, after which the agent hits a local minimum. Setting the seed did not seem to have any effect, since every run was vastly different. The A2C agent also has trouble hitting



(a) PPO results



(b) A2C results

Figure 2.1: Comparison of PPO and A2C results

3 | Conclusion

Upon comparing the performances of the Proximal Policy Optimization (PPO) and Advantage Actor Critic (A2C) agents, the PPO agent emerges as the more favorable choice. This conclusion is not solely based on the superior end result achieved by the PPO agent, but also on other factors that contribute to its effectiveness. Firstly, the PPO agent's reward converges more rapidly to a (local) maximum, which indicates a faster learning rate. Additionally, once the PPO agent reaches this point, it demonstrates greater consistency in maintaining its performance level compared to the A2C agent.

4 | Discussion

The performance of both agents is heavily influenced by the selection of parameters. Suboptimal parameter choices can significantly degrade the agent's performance, underscoring the importance of thorough exploration of the parameter space before making definitive conclusions about the superiority of one agent over another.
Moreover, it's important to note that no two trained agents are identical, even when the same parameters are used. Simulations have shown that the results can vary significantly, particularly for the A2C agent. Therefore, it's recommended to analyze multiple simulations to gain a comprehensive understanding of the agent's performance.
Lastly, it was observed that setting the seed does not seem to have a significant effect on the results, making tuning hyperparameters more challenging. We have had multiple runs in which the A2C will simply not converge in a reasonable amount of time. However, some runs would converge quite quickly.

A | Code

A.1 | assignment3.py

Note that this code is converted from a Jupyter Notebook.

```python
#!/usr/bin/env python
# coding: utf-8

# ### Knapsack Problem
# #### Description
# The knapsack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
# #### Environment
# The environment is a vector of 1000 items, each with a weight and a value. The goal is to maximize the total value of the items in the knapsack, while keeping the total weight below a certain threshold. The environment is considered solved when the agent achieves an average reward of 2000 over 100 consecutive episodes.
#

# In[5]:


import gym
import or_gym
from stable_baselines3 import PPO, A2C, A2C
from stable_baselines3.common.env_util import make_vec_env
from stable_baselines3.common.evaluation import evaluate_policy
import numpy as np
from stable_baselines3.common.monitor import Monitor
import matplotlib.pyplot as plt
import random

random.seed(42)

# Function to create environment
# Create vectorized environment
env = or_gym.make("Knapsack-v2", max_weight=300, mask=False)
seed = 0
np.random.seed(seed)
env = Monitor(env)


# In[17]:


policy_kwargs_ppo = dict(
    net_arch=[128, 128, 128, 128, 128]
)
# Wrap the environment
state_space = env.reset()  # Get the first state
action_space = env.action_space.n # Get the number of actions
ppo_agent = PPO('MlpPolicy', env, policy_kwargs=policy_kwargs_ppo, learning_rate=3e-4, n_steps=2048,
                batch_size=64, n_epochs=10, gamma=0.99, gae_lambda=0.95, verbose=1) # Create the agent
ppo_timestep_rewards = [] # Keep track of the training episode rewards for the PPO agent

n_episodes = 10 # Number of training episodes


# In[92]:


for i in range(n_episodes):
    # Train agents
    ppo_agent.learn(total_timesteps=10000) # Train the agent on the environment
    ppo_timestep_rewards += [ep_info["r"] for ep_info in ppo_agent.ep_info_buffer] # Get the training episode rewards for the current iteration
    env.reset() # Reset the environment
```

```python
print("PPO Agent performance (average reward, std. dev): ", evaluate_policy(ppo_agent,
    env, n_eval_episodes=100)) # Evaluate the agent


# Plot the learning curve (100 evaluation episodes after every training episode (so for
    now 10 * 100 = 1000 episodes in the plot))
plt.plot(ppo_timestep_rewards, label="PPO") # Plot the PPO learning curve
plt.xlabel("Episode")
plt.ylabel("Episode Reward")
plt.legend()
plt.title("Learning Curve")
plt.show()

ppo_agent.save("PPO_Agent_Knapsack") # Save the agent


# In[93]:


print(np.mean(ppo_timestep_rewards[-100:])) # Print the average reward over the last 100
    episodes


# In[19]:


#Evaluate the PPO agent
ppo_agent.load("PPO_Agent_Knapsack") # Load the agent
ppo_rewards = [] # Keep track of the rewards for the PPO agent
for _ in range(100): # Run 100 episodes
    obs = env.reset() # Reset the environment
    done = False
    total_reward = 0
    while not done: # Run until the episode is done
        action, _states = ppo_agent.predict(obs) # Get the action from the agent
        obs, reward, done, info = env.step(action) # Take the action in the environment
        total_reward += reward # Add the reward to the total reward
    ppo_rewards.append(total_reward) # Add the total reward to the list of rewards
ppo_success = np.mean(ppo_rewards) >= 2000 # Check if the agent was successful (average
    reward over 100 episodes >= 2000)
print(f"PPO Success: {ppo_success}")


# In[6]:


policy_kwargs_A2C = dict(net_arch=[128,64,64,64,64,64,128]) # Define the policy
    architecture for the A2C agent

A2C_agent = A2C('MlpPolicy', env, use_rms_prop=False,  learning_rate=4e-4, n_steps=25,
    max_grad_norm=0.6,
                ent_coef=0.01, vf_coef=0.500, gamma=0.99, gae_lambda=1, verbose=1) #
    Create the A2C agent
state_space = env.reset() # Get the first state
action_space = env.action_space.n # Get the number of actions


# In[7]:


n_episodes = 10 # Number of training episodes
A2C_timestep_rewards = [] # Keep track of the training episode rewards for the A2C agent


# In[12]:


for i in range(n_episodes):
    # Train agents
    A2C_agent.learn(total_timesteps=15000) # Train the agent on the environment
    # Get the training episode rewards for the current iteration
    A2C_timestep_rewards += [ep_info["r"] for ep_info in A2C_agent.ep_info_buffer]
    env.reset() # Reset the environment
```

```python
print("A2C Agent performance: ", evaluate_policy(A2C_agent, env, n_eval_episodes=100))
# Plot the learning curve (100 evaluation episodes after every training episode (so for
    now 10 * 100 = 1000 episodes in the plot))
plt.plot(A2C_timestep_rewards, label="A2C")
plt.xlabel("Episode")
plt.ylabel("Episode Reward")
plt.legend()
plt.title("Learning Curve")
plt.show()

A2C_agent.save("A2C_Agent_Knapsack")


# In[13]:


print(np.mean(A2C_timestep_rewards[-100:])) # Print the average reward over the last 100
    episodes


# In[15]:


A2C_agent.load("A2C_agent_weights") # Load the agent
A2C_rewards = [] # Keep track of the rewards for the A2C agent
for _ in range(100):
    obs = env.reset()
    done = False
    total_reward = 0
    while not done:
        action, _states = A2C_agent.predict(obs) # Get the action from the agent
        obs, reward, done, info = env.step(action) # Take the action in the environment
        total_reward += reward # Add the reward to the total reward
    A2C_rewards.append(total_reward)
A2C_success = np.mean(A2C_rewards) >= 2000  # Check if the agent was successful (average
    reward over 100 episodes >= 2000)
print(f"A2C Success: {A2C_success}")
```