Implementing Actor-Critic Architecture on Grid World

CAP 6629 Reinforcement Learning Project 3

Fanchen Bao

1 Introduction

While tabular method in reinforcement learning can solve problems with a small state and action space, it suffers from the curse of dimensionality when the state or action space becomes large, either due to a lot of discrete choices or being continuous. To address reinforcement learning problems with a large state or action space, one approach is to leverage neural network to model the state or action space. This way, instead of relying on a table to store state information, one can obtain it by passing state as input to a neural network model. Using neural network to model state or action space drastically reduces the burden on memory, and is thus a better method to handle reinforcement learning problems in the real world.

The Actor-Critic architecture is a classic usage of neural network model in reinforcement learning. Generally speaking, the actor is a neural network model of a parameterized policy function, where passing in the current state returns the probability distribution (or density) of all possible actions. The critic is a neural network model of a parameterized state (or action) value function, where passing in the current state (or state-action pair) yields its corresponding state (or action) value. During training, the goal of the actor is to maximize the probability of the action that can maximize the additional return at each state (i.e. advantage, see Section 2 for more detail). The goal of the critic is to align itself with the maximum return at each state. One can view the actor-critic relationship as a competitive one, in which the actor constantly wants to push the return higher than the value estimated by the critic, whereas the critic always wants to catch up with the current return. The two chases each other until the maximum return is approximated by the critic and achieved via the actions predicted by the actor.

The aim of this report is to implement the Actor-Critic architecture for a simple reinforcement learning task. The task chosen is to navigate a grid world, such as the one shown in Figure 1, from top left (start state) to bottom right (end state).

This report is organized in the following manner. Section 2 introduces mathematical notations and concepts of the Actor-Critic architecture. Section 3 provides pseudo-code for the Actor-Critic architecture and describes its implementation details. Section 4 evaluates the performance of the Actor-Critic architecture under different grid worlds. Finally, Section 5 compares the grid world performance under the Actor-Critic architecture with that under the tabular method and concludes the report.

2 Formal Definition

To discuss the mathematical notation of the Actor-Critic architecture, we must start with a parameterized total reward function. In tabular method, there is no total reward function, because it is possible to obtain the discounted total rewards at each step via tabulation. However, this is not possible when the problem involves large or continuous state or action space. Hence, a parameterized total reward function, as defined in Equation (1), is needed.

$$J(\theta) = \sum_{s \in S} d^{\pi}(s) V^{\pi}(s) = \sum_{s \in S} d^{\pi}(s) \sum_{a \in A} \pi_{\theta}(a|s) Q^{\pi}(s, a) \tag{1}$$

Here, θ is the trainable parameters. d^π is the stationary distribution of each state under the parameterized policy function π_θ. V^π and Q^π are the state and action value function following π_θ.

To achieve the maximum total reward, we can use gradient ascent technique to tune θ. To use gradient ascent, we need to compute the gradient of $J(\theta)$. According to the policy gradient theorem (see [1] for its mathematical proof), the gradient of $J(\theta)$ can be written as Equation (2).

$$\begin{aligned}\nabla_\theta J(\theta) &= \nabla_\theta \sum_{s \in S} d^\pi(s) \sum_{a \in A} \pi_\theta(a|s) Q^\pi(s,a) \\ &\propto \sum_{s \in S} d^\pi(s) \sum_{a \in A} \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) \\ &\propto \mathbb{E}_\pi[Q^\pi(s,a) \nabla_\theta \ln \pi_\theta(a|s)]\end{aligned} \tag{2}$$

It is important to note that Equation (2) is not the only way to compute $\nabla_\theta J(\theta)$. According to Schulman [2], $\nabla_\theta J(\theta)$ can also be expressed as Equation (3).

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} (Q^\pi(s_t,a_t) - V^\pi(s_t)) \nabla_\theta \ln \pi_\theta(a|s) \right] \\ &= \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} (G_t - V^\pi(s_t)) \nabla_\theta \ln \pi_\theta(a|s) \right]\end{aligned} \tag{3}$$

Here, we can substitute $Q^\pi(s_t,a_t)$ with G_t, the true discounted reward, because $Q^\pi(s_t,a_t) = \mathbb{E}_\pi[G_t|S_t,A_t]$. The expression $G_t - V^\pi(s_t)$ is called the **advantage**. It describes the difference between the true discounted reward at time t versus the estimated state value.

Equation (4), which is very similar to Equation (3), is the basis for constructing our Actor-Critic architecture.

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} (G_t - V_w(s_t)) \nabla_\theta \ln \pi_\theta(a|s) \right] \tag{4}$$

The only difference between the two is the parameterization of the state value function in Equation (4). We designate $\pi_\theta(a|s)$ as the actor and $V_w(s_t)$ as the critic. They are both approximated by neural network models, parameterized by θ and w, respectively. The reason we choose Equation (4) as the basis for the Actor-Critic architecture is the separation of the actor and critic in terms of training. Since there is no inherent link between the two, they can be trained independently.

However, the antagonistic interaction between the actor and critic is also apparent, because they are linked by the discounted reward G_t. G_t is completely controlled by the actor. The goal of the actor is to predict the best action that can maximize G_t, which in turn also maximizes the advantage. Meanwhile, the goal of the critic is to align itself with the current G_t and reduce the advantage to zero. The antagonistic interaction between the actor and critic can be summarized as follows:

1. The actor increases the advantage.
2. The increased advantage forces the critic to optimize such that the advantage is reduced again.
3. The reduced advantage forces the actor to optimize, and we go back to 1.

This interaction loop does not end until both the actor and critic converge, which leaves us with the parameter θ that maximizes $J(\theta)$. By definition, this will solve any reinforcement learning problem.

3 Implementation

The Actor-Critic architecture is implemented in a grid world as shown in Figure 1.

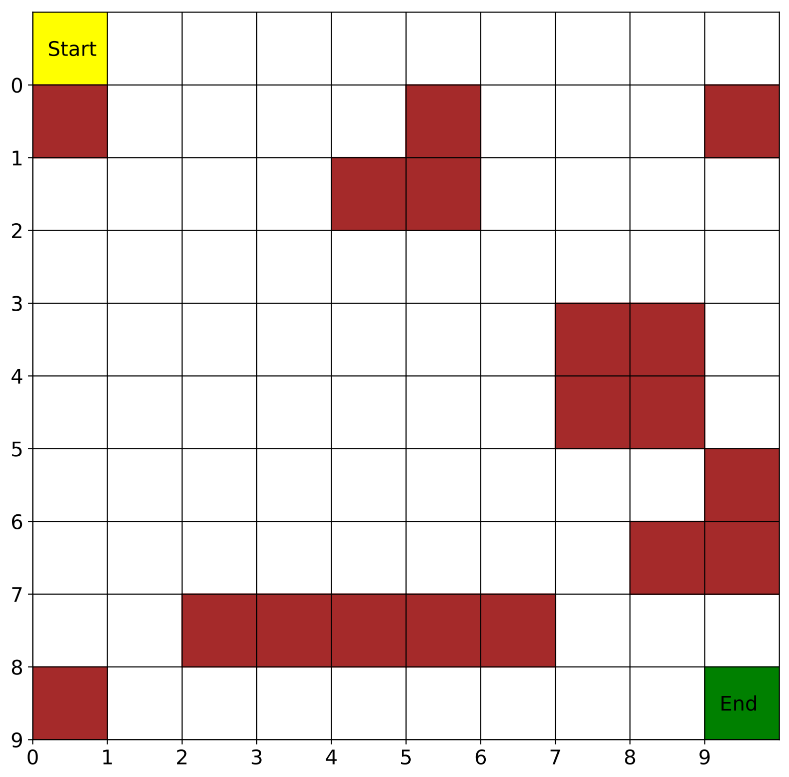

Figure 1: A Grid World with Obstacles

The grid world is 10×10 in size, with the start and end state marked by yellow and green, respectively. The brown cells represent blockages in the grid world where an agent cannot enter. The rule of the grid world is that an agent can only make a move in four directions: up, down, left, and right. Any move incurs a penalty of -1. If an action leads an agent out of bound or into the blockage, the agent remains where it is but still accumulates the penalty. The goal of the grid world is to use the Actor-Critic architecture to learn the optimal path from the start to end state while avoiding all the blockages.

3.1 Pseudo-code

The pseudo-code for the Actor-Critic architecture implementation is provided in Algorithm 1. Intuitively, for each episode, the pseudo-code accumulates C_{val}, P_{act}, and R at each step. Based on R, we can obtain the discounted total reward G at each step. We then use the mean squared error between G and C_{val} as the loss function to tune V_w. This loss function is a reasonable choice because it is widely used when fitting a neural network regressor. That said, it is worth mentioning that the documentation on TensorFlow [3] and Keras [4] use Huber Loss as the critic loss function.

Another important thing to point out is the computation of the actor loss. It is computed as $-\ln(P_{act}) \cdot A$. We need to compute the logarithm of the action probability because Equation

Algorithm 1 Actor-Critic Architecture

```
 1: Instantialize actor $\pi_\theta$ with initial random parameters $\theta$
 2: Instantialize critic $V_w(s)$ with initial random parameters $w$
 3: ep $\leftarrow 0$                    ▷ keep track of the number of episodes experienced
 4: max_eps                              ▷ Maximum number of episodes allowed
 5: $T$                                  ▷ Maximum number of steps per episode
 6: $\alpha_\theta \leftarrow 0.01$      ▷ Learning rate of the actor
 7: $\alpha_w \leftarrow 0.01$           ▷ Learning rate of the critic
 8: $\gamma \leftarrow 0.90$             ▷ Discount rate
 9: $C_{val} \leftarrow []$              ▷ Empty array to store the critic value at each step in an episode
10: $P_{act} \leftarrow []$    ▷ Empty array to store probability of a sampled action from the actor at each
    step in an episode
11: $R \leftarrow []$                    ▷ Empty array to store the reward at each step in an episode
12: $G \leftarrow []$      ▷ Empty array to store the discounted total reward at each step in an episode
13: Initialize starting state $s$
14:
15: repeat
16:     for $t = 1...T$ do:                        ▷ Each step in an episode
17:         $a, P_a \leftarrow \pi_\theta(a|s)$          ▷ Sample action and its probability
18:         c $\leftarrow$V$_w(s)$                       ▷ Obtain estimated critic value
19:         $r \leftarrow -1$                            ▷ Obtain the reward, which is always -1
20:         Append $c$ to $C_{val}$
21:         Append $P_a$ to $P_{act}$
22:         Append $r$ to $R$
23:     end for
24:
25:     $g \leftarrow 0$                               ▷ Accumulation of discounted reward
26:     for $r$ in reversed $R$ do:      ▷ The discounted total reward is computed in reverse
27:         $g \leftarrow$r + $\gamma g$
28:         Append $g$ to $G$
29:     end for
30:
31:     A $\leftarrow$G - C$_{val}$             ▷ Obtain advantage via pair-wise subtraction
32:     $L_c \leftarrow$A$^2/2T$                  ▷ Mean squared error, critic loss
33:     $L_a \leftarrow$-ln$(P_{act}) \cdot A$    ▷ Dot product, actor loss
34:
35:     Compute $\nabla_\theta \pi_\theta$ based on $L_a$     ▷ Obtain actor gradient
36:     $\theta \leftarrow \theta + \alpha_\theta \nabla_\theta \pi_\theta$    ▷ Update actor parameters
37:     Compute $\nabla_w V_w$ based on $L_c$                  ▷ Obtain critic gradient
38:     $w \leftarrow$w + $\alpha_w \nabla_w V_w$              ▷ Update critic parameters
39:
40:     ep $\leftarrow$ ep + 1
41: until ep = max_eps
42: return $\pi_\theta$ and $V_w$                ▷ The trained actor and critic
```

(4) explicitly requires so. We multiply by A, because the advantage gives us the signal whether the actor needs substantial update. If A is small, that means the actor needs to push for a better G_t. Hence it will update itself. If A is large, that means the actor is already producing good actions. Hence it will not make major changes. Finally, we need to add a negative sign, because when the loss function is being minimized during training, we are actually maximizing $\ln(P_{act}) \cdot A$, which is exactly the goal of the actor.

3.2 Neural Network Models

Algorithm 1 is written in Python, relying on Keras for constructing the neural network model for the actor and critic. Diagrams of the neural network models are shown in Figure 2. Both neural networks have two input nodes, corresponding to the row and column index on the grid world. Both also contain a single hidden layer with 128 nodes. The activation function for all hidden nodes is “relu”. The critic has one node in the output layer, with the activation function being “relu”. However, since the reward used in the grid world is negative, a negative sign is added to the critic output. The actor has four nodes in the output layer, corresponding to each action. The activation function for the output layer in the actor is “softmax”. It is worth mentioning that the documentation on TensorFlow [3] and Keras [4] let the actor and critic share the same hidden layer. But in our implementation, we have decided to make their training independent of each other.

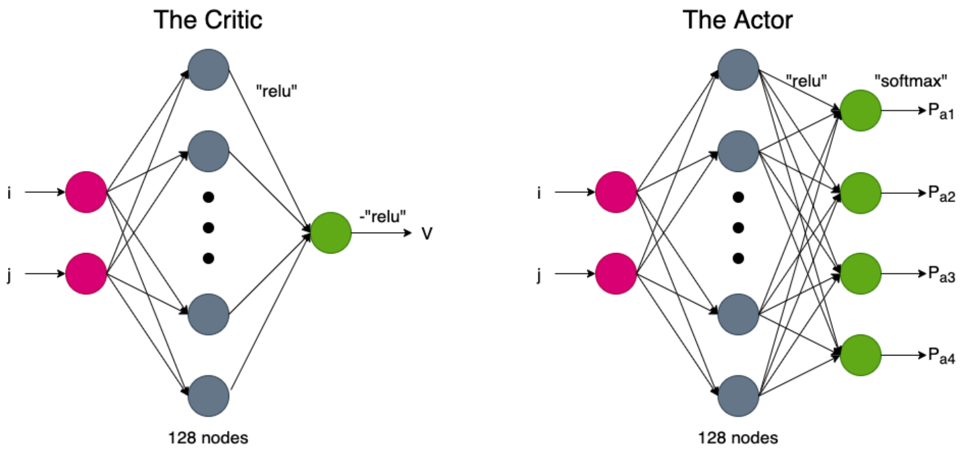

Figure 2: Neural network models for the critic and actor

The gradient of the critic and actor loss function is computed using the `GradientTape` API from TensorFlow. The training consists of 200 episodes, with each episode not exceeding 1000 steps.

4 Evaluation

The evaluation of the Actor-Critic architecture on the grid world is displayed in Figures 3, 4, and 5. These three figures represent three snapshots of the training process at the beginning, middle, and end of the 200 episodes, respectively. In each figure, subplot (A) is the steps-to-go curve at each episode. Since the maximum step per episode is capped at 1000, if an episode fails to reach the end state, the steps-to-go will be shown as 1000. The green horizontal line represents the optimal path with 18 steps. Subplot (B) shows the critic and actor loss in one plot, with the former using the left y-axis and the latter the right. Subplot (C) shows an example path constructed from the currently trained actor. And subplot (D) shows the heat-map of the state values based on the currently trained critic.

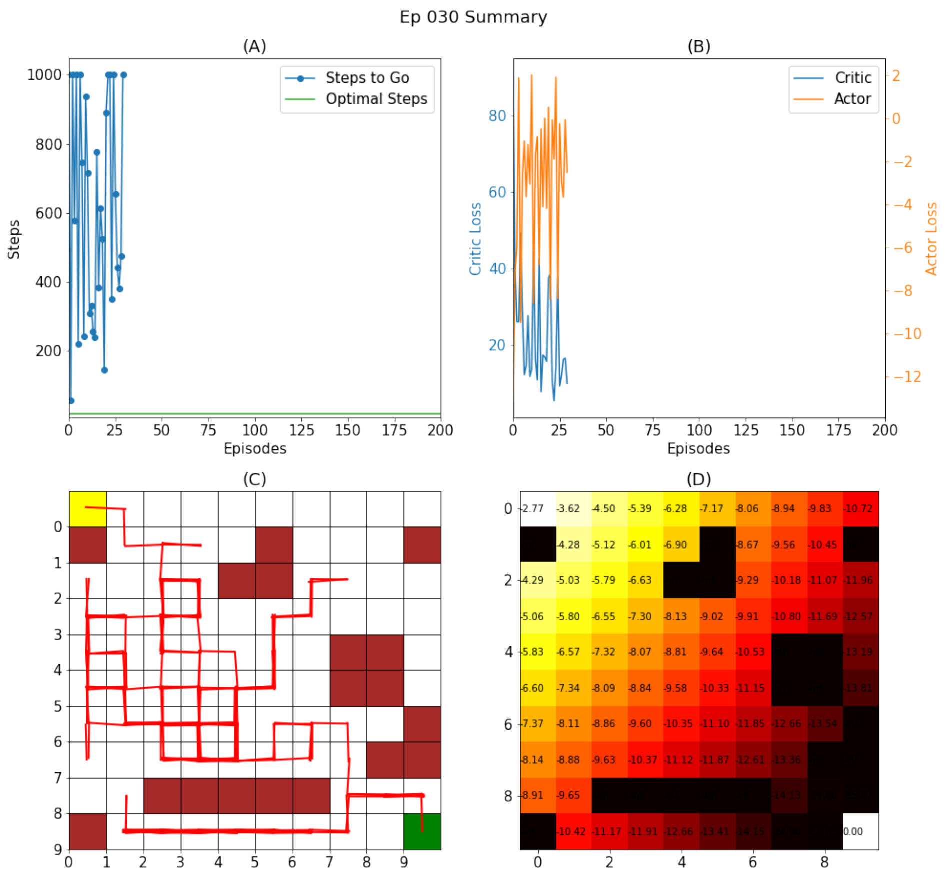

Figure 3: Training summary at 10 episodes, two-opening grid world

There are a few interesting observations. First, quite a few failed episodes occur in the training, even towards the end (e.g. Figure 5. This highlights the stochastic nature of the Actor-Critic architecture, where the action is always sampled from the actor according to their probabilities and there is no guarantee that the action with the highest probability is always going to be selected. That said, if we do not pay attention to the failed episodes, the steps-to-go seems to converge towards the optimal path (Figure 5).

Second, there is a lot of variation in both loss functions (subplot (B)), yet the critic loss seems to gradually decrease and the actor loss seems to converge towards 0 (Figure 5). This is understandable. For the critic loss, getting smaller means the critic is better at estimating the true discounted reward. For the action loss, converging towards 0 means the probability of the actor's predicted action is approaching 1, which means the actor is more confident in its choice.

Third, the agent apparently explores much more at the early stage of training. This can be seen in subplot (C) of Figure 3 where the agent explores many parts of the grid world. This is within expectation as the actor has not yet been solidified. This allows the agent the freedom to explore the grid world via random walk. Random walk is crucial to training, because the rule of the grid world mandates that the only way for an agent to reach the end state initially is via random walk (i.e. there is no other incentive or cues to guide it there). If a different reward system is given, where intermediate goals are given to guide the agent through the highly indeterministic area of the grid world, there will surely be much less random walk.

Finally, an obvious shift in the state value can be observed in subplot (D) across the three figures. In Figure 3, the high state value region (light color) is near the starting point; it rotates to the bottom left in Figure 4; and finally lands on the end state in Figure 5. This is a good representation of how the Actor-Critic architecture works out the problem. Initially, especially

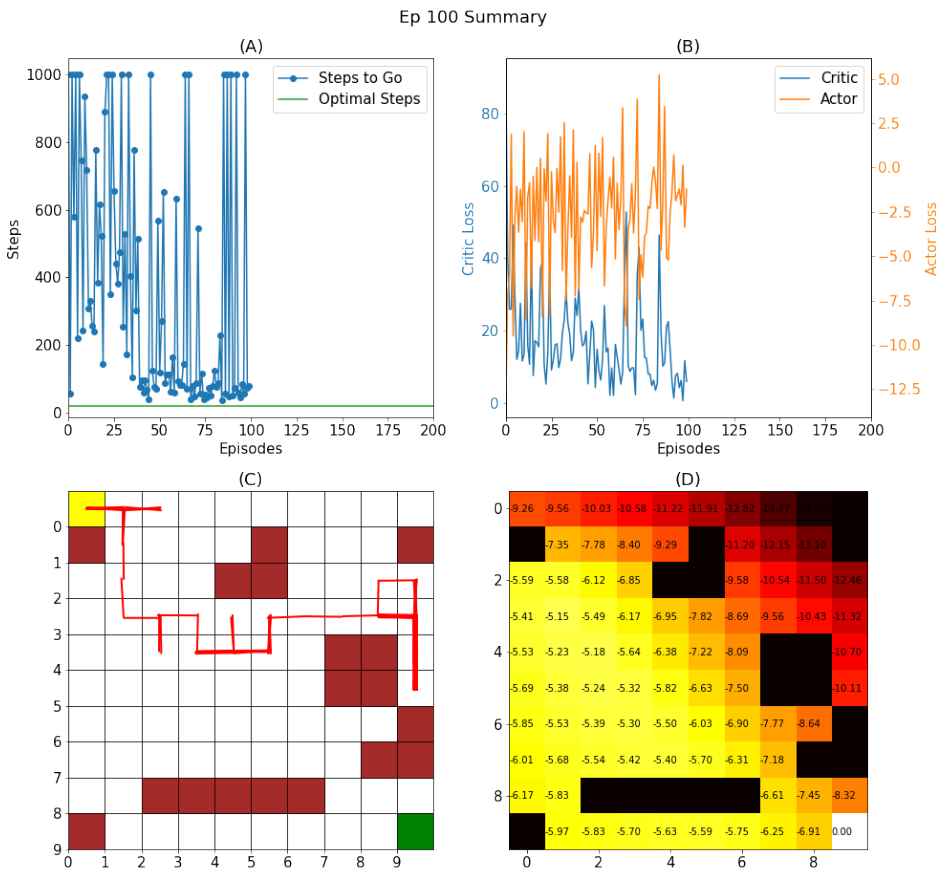

Figure 4: Training summary at 100 episodes, two-opening grid world

before the end state can be reached via random walk, the critic evaluates each state mainly based on how accessible it is. If a state is accessible, i.e. it can be reached from many sides, leaving that state for another will incur fewer penalties, because there is little chance the agent will get stuck. This is the case with the states near the upper left corner. On the contrary, if a state is surrounded by blockages, moving away from that state will likely get the agent stuck and incur a lot of penalties. Hence the states in the bottom right corner have low state values (dark color) at the beginning.

However, the situation starts to change once the end state is reached, because the states leading to the end state will now accumulate the least penalties. From Figure 4, it is apparent that the state values towards the bottom right corner are significantly higher than before (light color). Meanwhile, the low state value region (dark color) moves to the upper right corner as it is a region with good amount of blockages and also quite far from the end state.

Finally, when the critic stabilizes in Figure 5, the high state value region shifts to the bottom right corner. We can even pick out a path based on the color gradient in the heat-map. The top right corner still has the lowest state values (dark color) because it is far from the end state and surrounded by blockages. The top left corner is also low in state value (dark color) because it is also far from the end state. Yet, since there are fewer blockages in the top left corner, its state value is generally higher than the top right corner. The bottom left corner has similar level of state value as the top left. Despite the closeness of the bottom left corner to the end state, the many blockages around it counter this benefit.

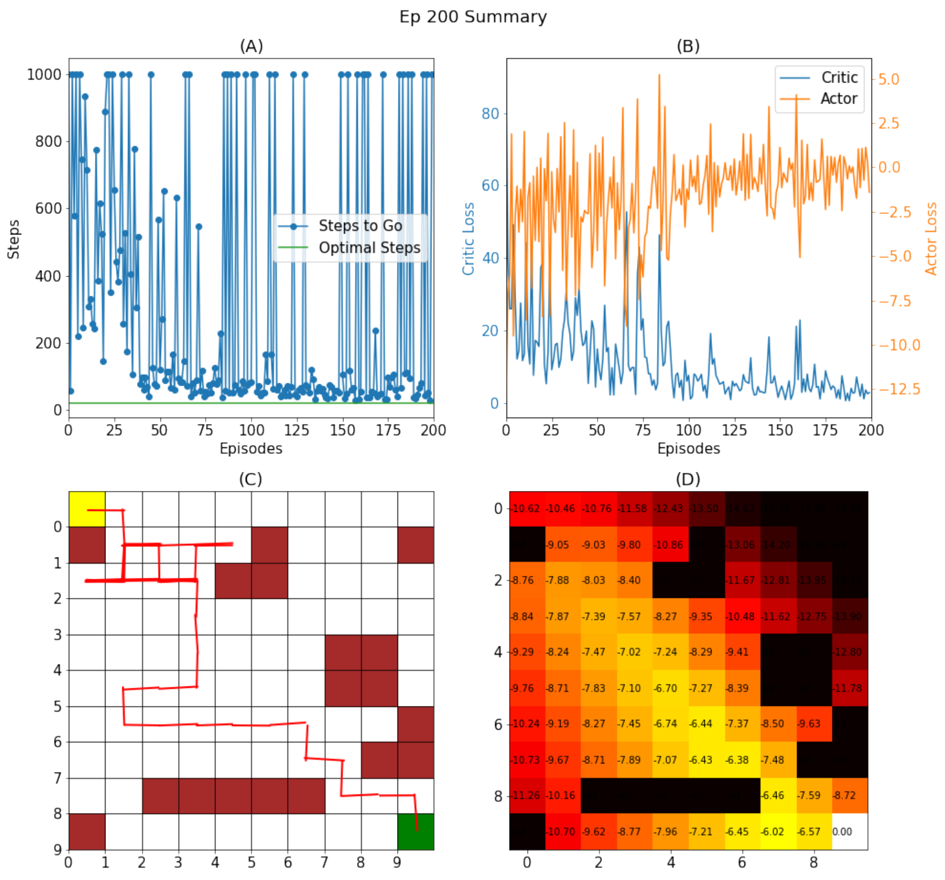

Figure 5: Training summary at 200 episodes, two-opening grid world

4.1 The One-opening Grid World

The evaluation shown above is based on a two-opening grid world. However, it is interesting to report that among all the paths the agent takes during the training process, it almost exclusively uses the right opening. One reason is that, compared to the left opening, the right opening is closer to the center of the grid world, which means it is more likely for the agent to hit it via random walk, especially during the early stage of training. Another reason is that the right opening is much closer to the end state than the left opening, which naturally gives it more importance, especially towards the end of training.

This observation begs the question: will an agent using the same Actor-Critic architecture be able to reach the end state if the right opening is blocked? To answer this question, we create a second grid world that has only one opening and run the same algorithm on it. For brevity, only the summary of the final episode is shown in Figure 6.

As demonstrated in subplot (C) and (D), the algorithm is not able to find a path through the left opening. The reason for such failure can be sourced from subplot (A) where we can clearly see that there are very few cases where the agent reaches the end state (i.e. successful episodes). Since only the successful episodes provide training information regarding the whereabouts of the end state and left opening, lack of successful episodes means that the actor and critic are under-trained. It is thus not surprising that the algorithm fails to find the correct path.

A deeper question to ask is why only few successful episodes are achieved in Figure 6. The answer is the same as why the right opening is preferred: it is simply not likely for an agent to reach the left opening via random walk. The more often the agent fails to reach the left opening during the early stage of training, the more likely it will not get there in the later stage as the actor and critic are solidified by the training data from the other states. Therefore, it is not

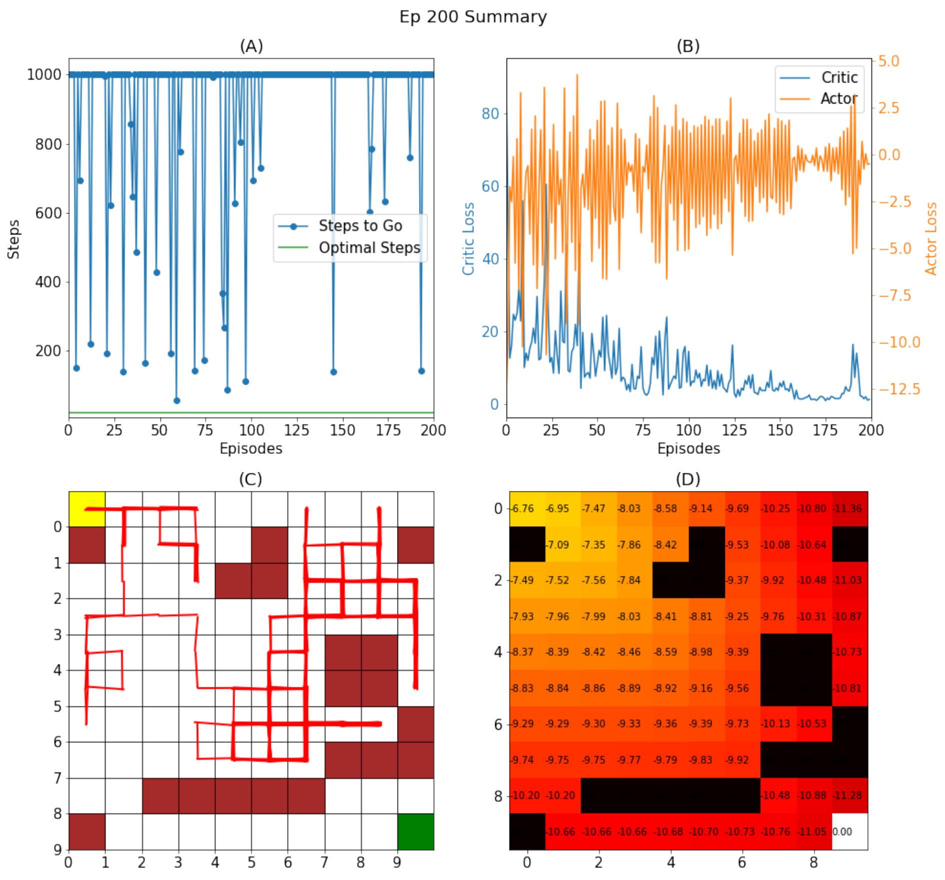

Figure 6: Training summary at 200 episodes, one-opening grid world, top left to bottom right

a stretch to say that given the current algorithm, we cannot solve the one-opening grid world problem.

4.2 Or can we?

The main question is how to get the agent to the left opening via random walk. It is apparently not likely when the agent starts from the top left corner, since there are too many distractions. But how about starting from the bottom right corner? There is no rule preventing us from using the bottom right corner as the starting point. A path found from bottom right to top left is equivalent from a path found from top left to bottom right. Therefore, it is perfectly fine going from the end state backwards toward the start state.

Yet, the benefit of doing so is tremendous. Since the bottom right and bottom left corners are heavily blocked, there is little ambiguity of where to go. With such reduced distraction, it is expected that an agent will be able to reach the left opening easily. Once the left opening is reached, finding the start state shouldn't be too much trouble, since it is located in an accessible area.

To confirm our hypothesis, we run the algorithm on the one-opening grid world backwards, with the start and end state swapped. The results are shown in Figures 7, 8, and 9, representing the beginning, middle, and end of the training process, respectively.

Compared to the one-opening grid world going from top left to bottom right, the reversed direction yields two major benefits. To begin with, at all three training stages, the agent is able to find its way to the start state (see subplot (C) in all three figures). This confirms our hypothesis that with a highly deterministic path from the end state to the left opening, the

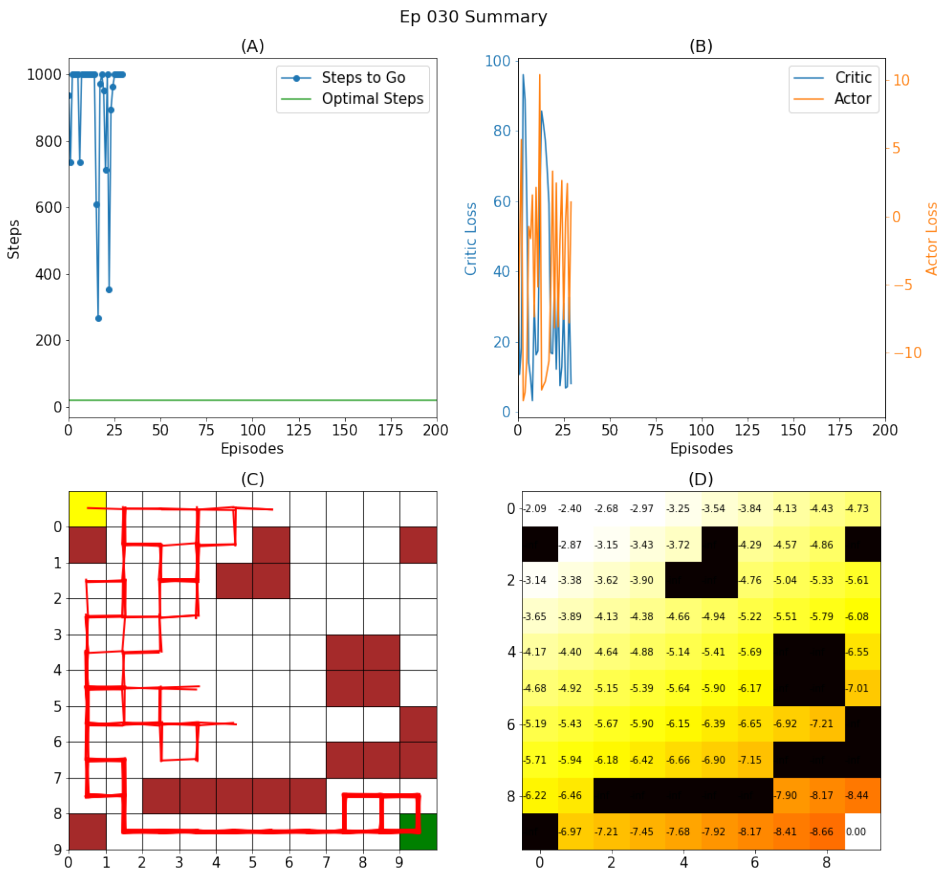

Figure 7: Training summary at 30 episodes, one-opening grid world, bottom right to top left

agent is able to first locate the left opening, and then random walk its way to the start state.

Second, from subplot (A), it is clear that going from the end to the start state results in many more successful episodes than the other way around. This provides the actor and critic more training samples, which further increases the agent's chance of having a successful episode. This virtuous cycle is in stark contrast to the vicious cycle when the agent is trained to go from the start to the end state.

Although the path obtained in 200 episodes is far from optimal, swapping the start and end state definitely solves a seemingly unsolvable reinforcement learning problem, without any modification to the core algorithm of the Actor-Critic architecture. It is highly likely that had we extended the training episodes, the reversed method would be able to converge its steps-to-go curve closer to the optimal value.

A good lesson we can learn from this example is that domain knowledge in machine learning is very important. With regard to the grid world, our domain knowledge points out that the only opening in the grid world is closer to the end state and that the path from that opening to the end state is deterministic. This domain knowledge allows us to swap the start and end state, and make the problem easier to solve. There are, of course, many other ways to break down the problem (e.g. set up intermediate goal at the left opening), but the key point is to always remember that one small piece of domain knowledge can be worth hundreds of hours of CPU/GPU time in machine learning.

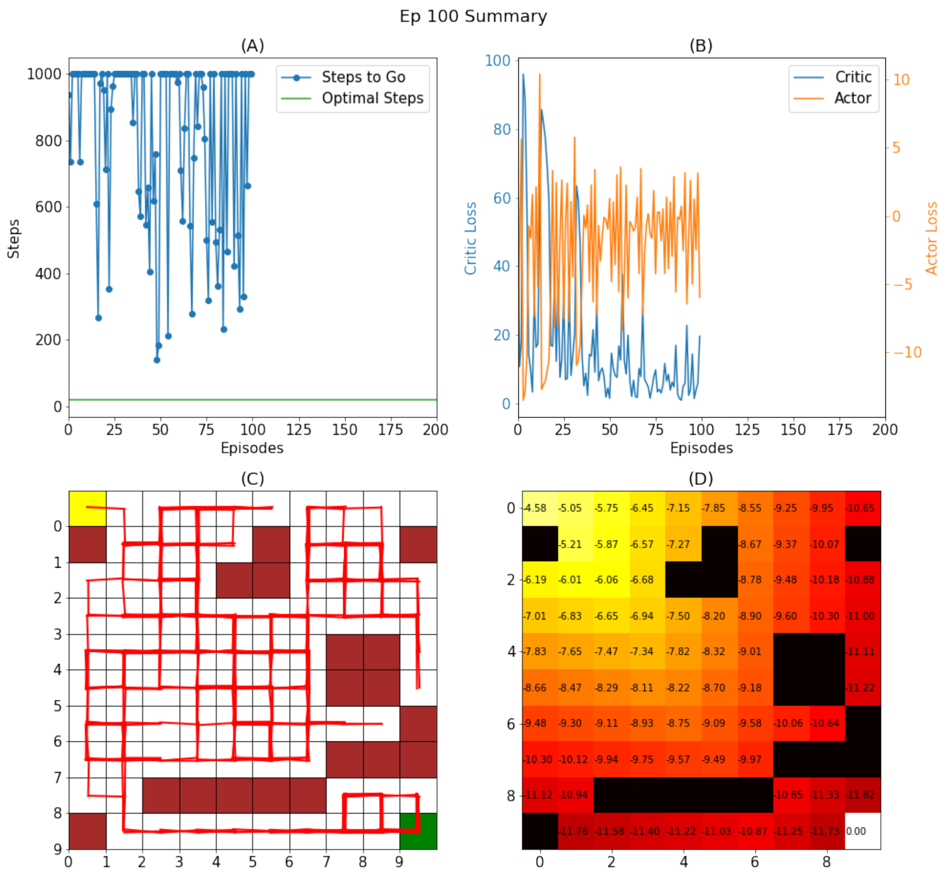

Figure 8: Training summary at 100 episodes, one-opening grid world, bottom right to top left

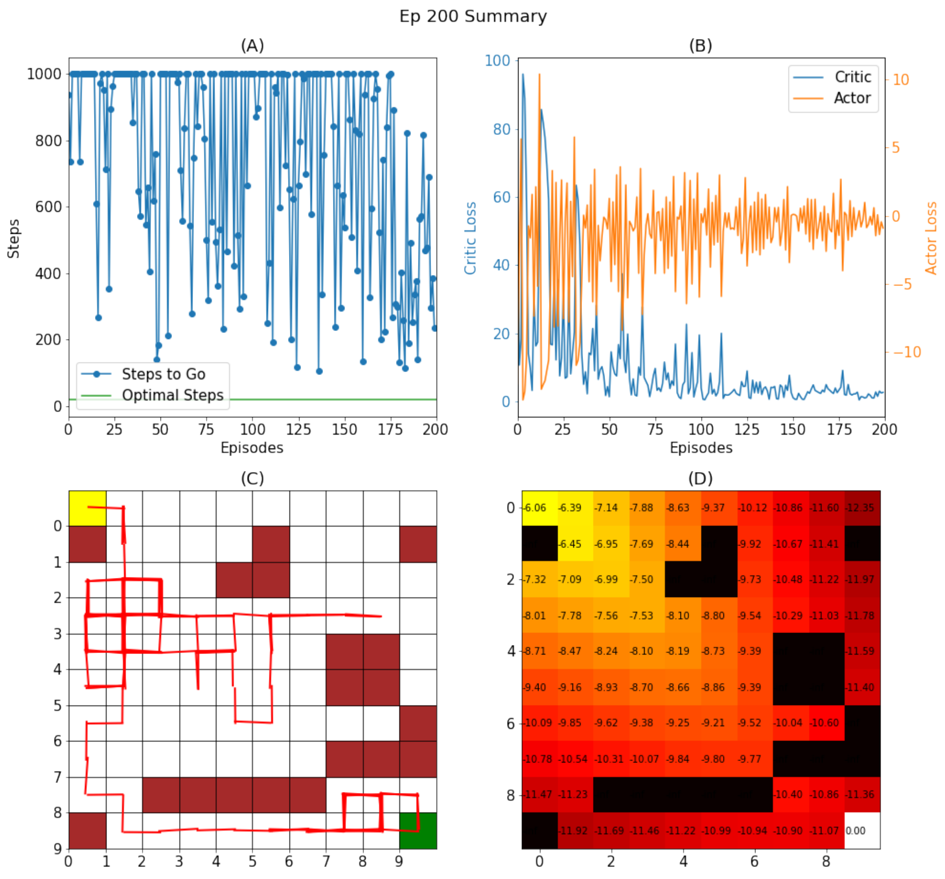

Figure 9: Training summary at 200 episodes, one-opening grid world, bottom right to top left

4.3 Comparison with Project 2

A quick comparison between the performance of the Actor-Critic architecture and that of the TD-λ tabular method from Project 2 (Figure 10) shows that the tabular method is much more stable in the training process (i.e. less fluctuation in steps-to-go), produces deterministic path, and normally converges its solution in shorter period of time. This tells us that the Actor-Critic architecture is not the right tool for use cases where the state and action space is small. It will be interesting to investigate at what magnitude of state and action space is the Actor-Critic architecture a better option than the tabular method.

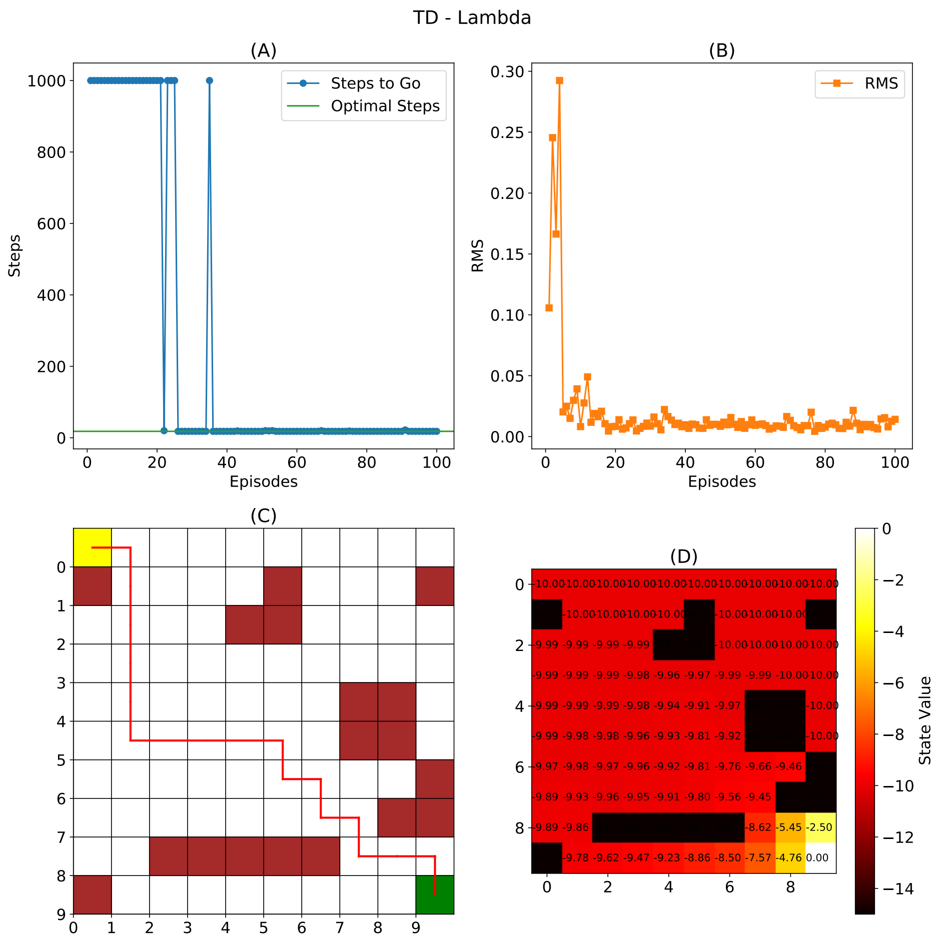

Figure 10: Grid world solved by TD-λ tabular method

5 Discussion And Conclusion

In this report, we have explained the mathematical reasoning behind the Actor-Critic architecture, showed its pseudo-code, and observed the performance of the Actor-Critic architecture over two types of grid world. We have found that our implementation of the actor and critic depends heavily on random walk to reach the end state at the beginning stage of training. This means if the end state or any bottleneck state (e.g. the left opening) is located in a region not easily accessible by random walk, it will be very difficult for the agent to obtain sufficient knowledge about them and the tuning of the actor and critic will fail.

One way to resolve this issue is to leverage the domain knowledge. In the case of the one-opening grid world, swapping the start and end state has significantly simplified the problem and made it solvable.

Finally, we compare the performance of solving the grid world problem using the Actor-Critic architecture and the TD-λ tabular method. The result shows that the tabular method has an advantage over the Actor-Critic architecture when the state and action space in the reinforcement learning problem is small.

References

[1] L. Weng, “Policy Gradient Algorithms,” Apr. 2018. [Online]. Available: https://lilianweng.github.io/2018/04/08/policy-gradient-algorithms.html

[2] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, “High-Dimensional Continuous Control Using Generalized Advantage Estimation,” *arXiv:1506.02438 [cs]*, Oct. 2018, arXiv: 1506.02438. [Online]. Available: http://arxiv.org/abs/1506.02438

[3] “Playing CartPole with the Actor-Critic Method | TensorFlow Core,” Apr. 2021. [Online]. Available: https://www.tensorflow.org/tutorials/reinforcement_learning/actor_critic

[4] A. Nandan, “Keras documentation: Actor Critic Method,” May 2020. [Online]. Available: https://keras.io/examples/rl/actor_critic_cartpole/

p3_Bao

April 9, 2021

[12]:
```python
# Ensure that the root directory is in Python's path. This is to make importing
# custom library easier.
from pathlib import Path
import sys
root = Path('.').absolute().parent.parent
if str(root) not in sys.path:
    sys.path.append(str(root))

# built-in packages
from collections import defaultdict
import json
from typing import List, Dict, Tuple, Callable
import math

# external packages
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
import scipy
import dill
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# matplotlib param
# https://stackoverflow.com/a/55188780/9723036
# SMALL_SIZE = 10
MEDIUM_SIZE = 15
BIGGER_SIZE = 17
TEXT_BOTTOM = 0.9

plt.rc('font', size=MEDIUM_SIZE)          # controls default text sizes
# plt.rc('axes', titlesize=SMALL_SIZE)     # fontsize of the axes title
plt.rc('axes', labelsize=MEDIUM_SIZE)    # fontsize of the x and y labels
plt.rc('xtick', labelsize=MEDIUM_SIZE)    # fontsize of the tick labels
plt.rc('ytick', labelsize=MEDIUM_SIZE)    # fontsize of the tick labels
plt.rc('legend', fontsize=MEDIUM_SIZE)    # legend fontsize
# plt.rc('figure', titlesize=BIGGER_SIZE)  # fontsize of the figure title

np.set_printoptions(formatter={'float': lambda x: "{0:0.5f}".format(x)})
```

1 Initialize Parameters

[13]:
```
ACTIONS = [[-1, 0], [0, 1], [1, 0], [0, -1]]  # N, E, S, W
M = 10  # number of rows
N = 10  # number of columns
NUM_BLK = 15  # number of blocks
GAMMA = 0.9  # discount rate
ACTOR_OPTIMIZER = keras.optimizers.SGD(learning_rate=0.01)
CRITIC_OPTIMIZER = keras.optimizers.SGD(learning_rate=0.01)
OPTIMAL_STEPS = 18  # empirically acquired
RANDOM_SEED = 42
```

2 TensorFlow Models

[14]:
```
INIT_WEIGHT = keras.initializers.RandomNormal(seed=RANDOM_SEED)


def state_value_func(hidden_node: int, num_layers: int):
    """Neural network model simulating the state value function.

    Note that a state value function takes a two dimensional input which␣
 ↪represents
    the current position of the cell in the grid world. For instance, given a␣
 ↪cell
    (i, j), the cell's state value is

    state_value_model(np.array[[i, j]])

    The return value is a tensor like this [[state_val]]. To access the value␣
 ↪itself,
    we can do:

    state_value_model(np.array[[i, j]])[0, 0]

    :param hidden_node: The number of nodes in each hidden layer.
    :param num_layers: The number of hidden layers.
    :return: A keras NN model.
    """
    inputs = keras.Input(shape=(2,))
    x = layers.Dense(hidden_node, activation='relu',␣
↪kernel_initializer=INIT_WEIGHT)(inputs)
    for _ in range(num_layers - 1):
        x = layers.Dense(hidden_node, activation='relu',␣
↪kernel_initializer=INIT_WEIGHT)(x)
    outputs = -layers.Dense(1, activation='relu',␣
↪kernel_initializer=INIT_WEIGHT)(x)
    return keras.Model(inputs=inputs, outputs=outputs)


def action_value_func(hidden_node: int, num_layers: int):
    """Neural network model simulating the state value function.

    Note that a state value function takes a two dimensional input which␣
↪represents
    the current position of the cell in the grid world. For instance, given a␣
↪cell
    (i, j), the cell's state value is

    state_value_model(np.array[[i, j]])

    The return value is a tensor like this [[state_val]]. To access the value␣
↪itself,
    we can do:

    state_value_model(np.array[[i, j]])[0, 0]

    :param hidden_node: The number of nodes in each hidden layer.
    :param num_layers: The number of hidden layers.
    :return: A keras NN model.
    """
    inputs = keras.Input(shape=(3,))
    x = layers.Dense(hidden_node, activation='relu',␣
↪kernel_initializer=INIT_WEIGHT)(inputs)
    for _ in range(num_layers - 1):
        x = layers.Dense(hidden_node, activation='relu',␣
↪kernel_initializer=INIT_WEIGHT)(x)
    outputs = -layers.Dense(1, activation='relu',␣
↪kernel_initializer=INIT_WEIGHT)(x)
    return keras.Model(inputs=inputs, outputs=outputs)


def policy_func(hidden_node: int, num_layers: int, num_actions: int):
    """Neural network model simulating the policy function.
    Note that a policy function takes a two dimensional input which represents␣
↪the
    current position of the cell in the grid world. For instance, given a cell
    (i, j), to obtain the policy distribution, we can call

    policy_model(np.array([[i, j]]))

    The return value of the above call is a tensor like this [[prob_0, prob_1,␣
↪prob_2, prob_3]].
    To access the probability of an action with index a, we can dO:

    policy_model(np.array([[i, j]]))[0, a]

    :param hidden_node: The number of nodes in each hidden layer.
    :param num_layers: The number of hidden layers.
    :return: A keras NN model.
    """
    inputs = keras.Input(shape=(2,))
    x = layers.Dense(hidden_node, activation='relu',␣
↪kernel_initializer=INIT_WEIGHT)(inputs)
    for _ in range(num_layers - 1):
        x = layers.Dense(hidden_node, activation='relu',␣
↪kernel_initializer=INIT_WEIGHT)(x)
    outputs = layers.Dense(num_actions, activation='softmax',␣
↪kernel_initializer=INIT_WEIGHT)(x)
    return keras.Model(inputs=inputs, outputs=outputs)
```

3 Plot Related

[45]:
```
def plot_step_to_go(ax, steps, optimal_steps: int, max_episodes, title: str):
    """Plot step-to-go vs. episode

    :param ax: An axis object of matplotlib.
    :param steps: Number of expected steps to reach the target for each episode.
    :param optimal_steps: The optimal number of steps to reach the target.
    :param max_episodes: The maximum number of episodes.
    :param title: Title for the step-to-go plot
    """
    ax.plot(np.arange(len(steps)), steps, marker='o', color='C0', label='Steps␣
↪to Go')
    ax.axhline(optimal_steps, color='C2', label='Optimal Steps')
    ax.set_xlabel('Episodes')
    ax.set_ylabel('Steps')
    ax.set_xlim(left=0, right=max_episodes)
    ax.set_title(title)
    ax.legend()
```

```
def plot_state_value_heatmap(ax, V, fig, title: str, color_vmin: int = -15,␣
 ↪color_vmax: int = 0):
    """Plot state value as a heatmap.

    :param ax: An axis object of matplotlib.
    :param V: The final estimated state values.
    :param fig: The fig parameter, acquired when calling plt.subplots().
    :param title: Title for the state value heatmap plot.
    :param color_vmin: Min value for the color map indicator, default to -15.
    :param color_vmax: Max value for the color map indicator, default to 0.
    """
    hm = ax.imshow(
        V,
        cmap='hot',
        interpolation='nearest',
        vmin=color_vmin,  # colorbar min
        vmax=color_vmax,  # colorbar max
    )
    # cbar = fig.colorbar(hm)
    # cbar.set_label('State Value')

    m, n = V.shape
    for i in range(m):  # add V value to each heatmap cell
        for j in range(n):
            val = '-inf' if V[i, j] <= -1000.0 else f'{V[i, j]:.2f}'
            ax.annotate(val, xy=(j - 0.5, i + 0.1), fontsize=10)
    ax.set_title(title)


def plot_path(
    ax,
    actor,
    start: Tuple[int, int],
    end: Tuple[int, int],
    title: str,
    max_steps: int,
    rng,
    grid=GRID,
):
    """Plot a route from start to end on the grid world.

    Yellow is start; green is end; brown is obstacle.

    :param ax: An axis object of matplotlib.
    :param actor: The policy function NN simulation.
    :param start: The start state in the format of (row_idx, col_idx)
    :param end: The end state in the format of (row_idx, col_idx)
    :param title: Title for the path plot.
    :param max_steps: Max number of steps allowed when drawing a path. If
        max steps are reached, the path is left as is.
    :param rng: A random number generator.
    :param grid: The grid world. Default to GRID.
    """
    m, n = grid.shape
    ax.grid(True)
    ax.set_xticks(np.arange(n))
    ax.set_xlim(left=0, right=n)
    ax.set_yticks(np.arange(m))
    ax.set_ylim(top=m - 1, bottom=-1)
    ax.invert_yaxis()  # invert y axis such that 0 is at the top
    i, j = start
    step = 0  # note that each invalid action counts as a step as well.
    while (i, j) != end and step < max_steps:
        a, _ = next_action(i, j, actor, rng=rng)
        di, dj = ACTIONS[a]
        ni, nj = i + di, j + dj
        if is_valid_state(ni, nj, grid=grid):
            ax.plot(  # jiggle on each path plot to visualize paths taken␣
↪multiple times vs. once
                [j + 0.5 + (rng.random() - 0.5) / 10, nj + 0.5 + (rng.random()␣
↪- 0.5) / 10],
                [i - 0.5 + (rng.random() - 0.5) / 10, ni - 0.5 + (rng.random()␣
↪- 0.5) / 10],
                color='red',
                lw=2,
            )
            i, j = ni, nj
        step += 1
    # label start, end and blocks
    ax.add_patch(Rectangle((0, -1), 1, 1, fill=True, color='yellow'))
    ax.add_patch(Rectangle((n - 1, m - 2), 1, 1, fill=True, color='green'))
    for i, j in zip(*np.where(grid == 1)):
        ax.add_patch(Rectangle((j, i - 1), 1, 1, fill=True, color='brown'))
    ax.set_title(title)


def plot_loss(ax, critic_losses: List, actor_losses: List, title: str,␣
↪max_episodes: int):
    """Plot critic and actor loss in a single graph.

    Critic loss takes the left y-axis. Actor loss takes the right.
    :param ax: An axis object of matplotlib.
    :param critic_losses: A list of critic loss per episode.
    :param actor_losses: A list of actor loss per episode.
    :param title: Title for the path plot.
    :param max_episodes: The maximum number of episodes.
    """
    num_loss = len(critic_losses)
    twin = ax.twinx()  # put actor loss on a separate y axis
    critic_plot, = ax.plot(np.arange(num_loss), critic_losses, color='C0',␣
↪label='Critic')
    actor_plot, = twin.plot(np.arange(num_loss), actor_losses, color='C1',␣
↪label='Actor')
    ax.set_xlim(left=0, right=max_episodes)
    # configure critic loss axis
    ax.set_ylabel('Critic Loss')
    ax.yaxis.label.set_color(critic_plot.get_color())
    ax.tick_params(axis='y', colors=critic_plot.get_color())
    # configure actor loss axis
    twin.set_ylabel('Actor Loss')
    twin.yaxis.label.set_color(actor_plot.get_color())
    twin.tick_params(axis='y', colors=actor_plot.get_color())
    ax.set_xlabel('Episodes')
    ax.set_title(title)
    ax.legend(handles=[critic_plot, actor_plot])


def plot(
    title: str,
    critic,
    actor,
    critic_losses: List,
    actor_losses: List,
    steps: List,
    start: Tuple[int, int],
    end: Tuple[int, int],
    max_steps: int,
    max_episodes: int,
    optimal_steps: int = OPTIMAL_STEPS,
    grid=GRID,
    V=V,
):
    """One-stop shop to plot all performance metrics during training.

    :param title: The suptitle of the entire figure.
    :param critic: The state value function NN simulation.
    :param actor: The policy function NN simulation.
    :param critic_losses: A list of critic losses per episode.
    :param actor_losses: A list of actor loss per episode.
    :param steps: A list of steps taken per episode. A failed episode has the␣
↪max
        number of steps allowed.
    :param start: The start state in the format of (row_idx, col_idx)
    :param end: The end state in the format of (row_idx, col_idx)
    :param max_steps: Maximum number of steps per episode.
    :param max_episodes: Maximum number of episodes.
    :param optimal_steps: The optimal number of steps to reach the target.
    :param grid: The grid world. Default to GRID.
    :param V: The final estimated state values.
    """
    rng=np.random.default_rng()
    plt.rc('grid', linestyle="-", color='black')
    fig, axes = plt.subplots(2, 2, figsize=(14, 13))
    axes = axes.flatten()
    # First three plots
    plot_step_to_go(axes[0], steps, optimal_steps, max_episodes, '(A)')
    plot_loss(axes[1], critic_losses, actor_losses, '(B)', max_episodes)
    plot_path(axes[2], actor, start, end, '(C)', max_steps, rng, grid=grid)
    # Compute V table
    for i in range(grid.shape[0]):
        for j in range(grid.shape[1]):
            V[i, j] = -1000 if grid[i, j] else critic(state(i, j))[0, 0]
    V[grid.shape[0] - 1, grid.shape[1] - 1] = 0
    plot_state_value_heatmap(axes[3], V, fig, '(D)', color_vmin=-15,␣
↪color_vmax=-3)

    fig.suptitle(title)
    plt.tight_layout()
    filename = '_'.join(title.split())
    if filename:
        plt.savefig(filename + '.png')
        plt.ioff()
    else:
        plt.show()
```

4 Initialize Grid and State Value Table

[17]:
```
def initialize_grid(num_rows: int, num_cols: int, num_blocks: int, random_state:
↪ int = RANDOM_SEED):
    """Initialize a grid world, with obstacles.

    :param num_rows: Number of rows in the grid world.
    :param num_cols: Number of columns in the grid world.
    :param num_blocks: Number of RANDOM blocks that agent cannot cross. The␣
↪final grid world
        could include more blocks as additional  a manually
    :param random_state: For reproducing random values.
    :return: The grid itself and the tabulation for state values.
    """
    # Create a grid
    grid = np.array([[0] * num_cols for _ in range(num_rows)])
    V = np.array([[0.0] * num_cols for _ in range(num_rows)])
    # randomly generate blocks
    rng = np.random.default_rng(random_state)
    block_i = rng.integers(low=0, high=num_rows, size=num_blocks)
    block_j = rng.integers(low=0, high=num_cols, size=num_blocks)

    ###################################################
    # hardcodede additional blocks, such that we can  #
    # limit the possible number of ways to reach the  #
    # terminal state                                  #
    ###################################################
    block_i = np.append(block_i, [8, 8, 8])
    block_j = np.append(block_j, [4, 5, 6])

    # Update grid and state values by identifying the blocks
    grid[block_i, block_j] = 1
    return grid, V


GRID, V = initialize_grid(M, N, NUM_BLK, random_state=10)
```

5 Training Helper Functions

[18]:
```
def next_state(i: int, j: int, a: int, grid=GRID) -> Tuple[int, int]:
    """Obtain the next state given the current state and action.

    The peculiarity of this is that if the next state is invalid, i.e.
    it is easier outside the grid world or in a blockage, we return the
    original state, signifying that the result of the illegal action
    results in the agent being stuck at the same location.

    :param i: Row index of the agent on the grid world.
    :param j: Column index of the agent on the grid world.
    :param a: Index of ACTIONS, signifying the specific action to take.
    :param grid: The grid world. Default to GRID.
    :return: The next cell's row and column index.
    """
    di, dj = ACTIONS[a]
    ni, nj = i + di, j + dj
    if is_valid_state(ni, nj, grid=grid):
        return ni, nj
    return i, j


def next_action(i: int, j: int, actor, rng) -> Tuple[float, float]:
    """Obtain the next action from the publicy function simulation.

    Note that we also obtain the probability of the action as reported
    by the actor.

    :param i: Row index of the agent on the grid world.
    :param j: Column index of the agent on the grid world.
    :param actor: The policy function simulation.
    :returns: The action and its associated probability.
    """
    a = rng.choice(len(ACTIONS), p=np.squeeze(actor(state(i, j))))
    return a, actor(state(i, j))[0, a]


def state(i: int, j: int):
    """Generate a numpy array version of the state.

    This is used as the input value for the state and policy function
    NN.

    :param i: Row index of the agent on the grid world.
    :param j: Column index of the agent on the grid world.
    :return: A numpy array of shape (1, 2)
    """
    return np.array([[i, j]])


def state_action(i: int, j: int, a: int):
    """Generate a numpy array version of the state action pair.

    This is used as the input value for the action value function NN.

    :param i: Row index of the agent on the grid world.
    :param j: Column index of the agent on the grid world.
    :param a: Index of ACTIONS, signifying the specific action to take.
    :return: A numpy array of shape (1, 3)
    """
    return np.array([[i, j, a]])
```

```python
def is_valid_state(i: int, j: int, grid=GRID) -> bool:
    """Check whether the current state is valid.

    A valid state must be within the grid world and NOT on any of the
    blocks.

    :param i: Row index of the agent on the grid world.
    :param j: Column index of the agent on the grid world.
    :param grid: The grid world. Default to GRID.
    :return: A boolean value, true => state is valid, false otherwise.
    """
    m, n = grid.shape
    return 0 <= i < m and 0 <= j < n and grid[i, j] == 0
```

6 Actor Critic Architecture

[50]:
```python
MEAN_SQUARED_ERROR = tf.keras.losses.MeanSquaredError()

def actor_critic_monte_carlo(
    actor,
    critic,
    grid,
    start: Tuple[int, int],
    end: Tuple[int, int],
    include_all: bool = True,
    max_episodes: int = 200,
    max_steps: int = 1000,
    actor_opt = ACTOR_OPTIMIZER,
    critic_opt = CRITIC_OPTIMIZER,
    gamma: float = GAMMA,
    random_seed: int = RANDOM_SEED,
):
    """Find the shortest path in the grid world using actor critic architecture␣
 ↪with Monte Carlo.

    Since Monte Carlo is used, each episode extends until the terminal state is␣
 ↪reached or the
    max number of steps achieved, whichever comes first. Based on the path, we␣
 ↪can compute the
    actual discounted reward at each step, and use that to compute the error on␣
 ↪critic. This
    error is then used to compute the loss function for both the critic and␣
 ↪actor NN for gradient
    descent/ascent optimization.

    :param actor: The policy function NN simulation.
    :param critic: The state value function NN simulation.
    :param grid: The grid world.
    :param start: The grid coordinates of the starting position.
    :param end: The grid coordinates of the end position.
    :param include_all: A boolean flag to determine whether all episodes,␣
↪successful or failed,
        are included in the training data. Default to True.
    :param max_episodes: Maxinum episodes to use in training the critic and␣
↪actor. If include_all
        is set to False, significantly more episodes might be needed overall to␣
↪achieve max_episodes
        number of successful expisodes. Default to 200.
    :param max_steps: Maximum steps to take in the grid world in each episode.
        Default to 1000.
    :param actor_opt: Optimizer for the actor NN. Default to ACTOR_OPTIMIZER.
    :param critic_opt: Optimizer for the critic NN. Default to CRITIC_OPTIMIZER.
    :param gamma: Discount rate. Default to GAMMA.
    :param random_seed: To seed a random number generator.
    """
    rng = np.random.default_rng(random_seed)
    m, n = grid.shape
    all_critic_losses = []
    all_actor_losses = []
    all_steps = []
    ep = 0
    total_ep = 0
    while ep < max_episodes:
        critic_vals = []
        actor_log_probs = []
        rewards = []
        with tf.GradientTape(persistent=True) as tape:
            i, j = start
            step = 0  # record steps taken in the current episode
            while step < max_steps and (i, j) != end:
                a, aprob = next_action(i, j, actor, rng=rng)
                cval = critic(state(i, j))[0, 0]
                critic_vals.append(cval)
                actor_log_probs.append(tf.math.log(aprob))
                rewards.append(-1)  # reward is always -1.
                i, j = next_state(i, j, a, grid=grid)
                step += 1
            # Compute discounted returns
            returns = []
            discounted_return = 0
            for r in rewards[::-1]:
                discounted_return = r + gamma * discounted_return
                returns.append(discounted_return)
            returns = returns[::-1]
            # Convert to Tensors
            critic_vals = tf.convert_to_tensor(critic_vals, dtype=tf.float32)
            actor_log_probs = tf.convert_to_tensor(actor_log_probs, dtype=tf.
↪float32)
            returns = tf.convert_to_tensor(returns, dtype=tf.float32)
            # Compute critic and actor NN loss
            advtange = returns - critic_vals
            actor_loss = -tf.math.reduce_mean(actor_log_probs * advtange)
            critic_loss = MEAN_SQUARED_ERROR(critic_vals, returns)

        # Compute gradient and update NN weights
        # However, NN weights update ONLY if the current episode is successful,␣
↪or
        # it is instructed to include all expisodes
        if step < max_steps or include_all:
            actor_gradient = tape.gradient(actor_loss, actor.
↪trainable_variables)
            actor_opt.apply_gradients(zip(actor_gradient, actor.
↪trainable_variables))
            critic_gradient = tape.gradient(critic_loss, critic.
↪trainable_variables)
            critic_opt.apply_gradients(zip(critic_gradient, critic.
↪trainable_variables))
            ep += 1
            # Record losses and steps
            all_critic_losses.append(critic_loss)
            all_actor_losses.append(actor_loss)
            all_steps.append(step)
            if ep % 10 == 0:
                print(f'Episode: {ep}, critic loss: {critic_loss}, actor loss:␣
↪{actor_loss}', end=' ')
                if (i, j) == end:
                    print(f'Terminal State, steps: {step}')
                else:
                    print('Max steps exhausted')
                plot(
                    f'Ep {(ep):03} Summary',
                    critic,
                    actor,
                    all_critic_losses,
                    all_actor_losses,
                    all_steps,
                    start,
                    end,
                    max_steps,
                    max_episodes,
                    grid=grid,
                )
        total_ep += 1
    print(f'Total episodes: {total_ep}')
```

7 Grid World with Two Openings

[1]:
```
# Train the actor critic architecture
policy_model_1 = policy_func(128, 1, len(ACTIONS))
state_value_model_1 = state_value_func(128, 1)
actor_critic_monte_carlo(
    policy_model_1,
    state_value_model_1,
    GRID,
    (0, 0),
    (GRID.shape[0] - 1, GRID.shape[1] - 1),
)
# Save trained models
policy_model_1.save('policy_model_1')
state_value_model_1.save('state_value_model_1')
```

8 Grid World with One Opening

8.1 Top left to bottom right

[2]:
```
# Add one more block such that there is only one opening to the terminal state
GRID[7, 7] = 1

# Train the actor critic architecture
policy_model_3 = policy_func(128, 1, len(ACTIONS))
state_value_model_3 = state_value_func(128, 1)
actor_critic_monte_carlo(
    policy_model_3,
    state_value_model_3,
    GRID,
    (0, 0),
    (GRID.shape[0] - 1, GRID.shape[1] - 1),
)
# Save trained models
policy_model_3.save('policy_model_3')
state_value_model_3.save('state_value_model_3')

# Remove the block
GRID[7, 7] = 0
```

8.2 Bottom right to top left

[3]:
```python
# Add one more block such that there is only one opening to the terminal state
GRID[7, 7] = 1

# Train the actor critic architecture
policy_model_4 = policy_func(128, 1, len(ACTIONS))
state_value_model_4 = state_value_func(128, 1)
actor_critic_monte_carlo(
    policy_model_4,
    state_value_model_4,
    GRID,
    (GRID.shape[0] - 1, GRID.shape[1] - 1),
    (0, 0),
)
# Save trained models
policy_model_4.save('policy_model_4')
state_value_model_4.save('state_value_model_4')

# Remove the block
GRID[7, 7] = 0
```