[Project #2]
Traveling Salesman Problem
소스코드 설명서

[01 팀]

	이름	학번	학년	E-mail	기여도(%)
팀장	선민준	20201273	3	sunminjun89@gmail.com	33
조원 1	남광규	20192898	3	ska00100@naver.com	24
조원 2	황준성	20221820	3	junseong5832@gmail.com	24
조원 3	홍진영	20221785	3	sunaep123@naver.com	19

1 VALUE-BASED POLICY EXTRACTION

2 Q-LEARNING

3 DNN Q-LEARNING

4 DEEP Q-NETWORK

1 VALUE-BASED POLICY EXTRACTION

1.1 TREE_SEARCH_Q_TABLE.IPYNB

1. 주요 함수들

calculate_distance : 0 부터 시작해서 나온 경로들에 대한 거리 계산하는 함수

choose_reward : 각 도시의 거리 평균값 계산 후 거리 평균 값보다 작다면 -log 씌우고 거리 평균값보다 크다면 -지수씌워서 reward 주는 함수

heuristic_fuc : A*에 사용되는 heuristic 함수 해당 도시에서 greedy 하게 탐색해나간다.

a_star_search: 각 도시에 대한 fringe 를 제한해서 도시들을 탐색하는 함수, 제일 처음 나온 경로,거리 반환

greedy_search : 각 도시에 대해 가까운 도시순으로 방문하는 함수, 경로들과 거리 반환

euclidean_distance: 두 도시간의 유클리디언 거리 계산하는 함수

2.과정

2-1: tree 나 유전 알고리즘을 활용한 경로 추출

2-2: value 초기화 : 각 상태에 대한 value table 을 초기화

2-3: value iteration : 초기화된 value talbe 에 대해 V*(s) = max(Reward(s, a) + (discount factor x V*(s))) 과정 진행

2-4 : Q table : value iteration 이 진행된 value table 로 Q[s, a] = Reward(s, a) +discount_factor *v[a]로 Q table 구성

2-5: POLICY EXTRACTION: argmax 로 가장 좋은 Q value 선택한 후 경로 추출

3.필요한 정보

MAX_FRINGE_LIMIT = 8 현재 도시에서 선택가능한 후보들 수

G : 각 도시들에 대한 완전 연결 그래프

sortedNodeList: :각 도시에 대해 거리가 작은 값이 제일 앞에 있는 오름차순으로 정렬된 2 차원 리스트

V(s): 상태 s 에 도달했을 때 얻을 수 있는 기대 보상의 총합

discount_factor = 미래 보상을 떨어트리는 변수

2 Q-LEARNING

2.1 MC_Q_TABLE.IPYNB

1. 하이퍼파라미터 설정

MAX_FRINGE_LIMIT = 8

learning_rate = 0.1 학습률

discount_factor = 0.9 할인율

epsilon = 0.1 탐험률

num_episodes = 1000 에피소드 개수

2. 주요 함수

a_star_search_limit : 각 도시에 대한 fringe 를 제한해서 도시들을 탐색하는 함수, 이때 앞에서 사용한 a_star_search

함수와는 다르게 경로들에 대한 도시들 개수를 제한한다.

heuristic_fuc: 그리디 함수를 heuristic 설정하였다.

generate_random_city : state 도시랑, epath 를 인자로 받고 state 랑 다르고 epath 에 없는 랜덤한 도시 반환

generate_max_city: state 도시랑, epath 를 인자로 받고 epath 에 없는 도시들 중 state 의 Q 값이 가장 높은 도시 찾기

choose_action : state 도시랑, epath 를 인자로 받고 epsilon 을 사용해서 0.1 확률보다 작다면 generate_random_city 의 결과값을 반환하고 0.1 보다 크거나 같다면 generate_max_city 함수의 결과값 반환

generate_episode :cur_state, pi_state, epath 를 인자로 받고 episode 라는 리스트를 만들고 전체 reward 를 저장한 Gt 라는 변수를 만들고 현재 도시 , 다음 도시에 대한 reward 를 저장해서 초기화해주고 while 문에서 epath 에 없는 도시들에 대해 choose_action 함수를 호출해서 다음 도시를 정하고 현재도시,다음도시에 대한 reward 를 정하고 앞에서 초기화한 Gt 값에 reward 를 더해주고 현재 도시를 다음 도시로 업데이트해준다. 또한 while 문 안에서 구한 다음 도시를 방문했다는 걸 경로에 추가한다. 경로가 998 개 도시로 즉 한개의 에피소드가 완성되면 while 문 탈출한다 앞에서 구한 Gt 라는 전체 total 에 reward 를

각 도시, 상대도시, 전체 리워드를 각각 저장한다.

이후 각 도시, 상대도시, 전체 리워드가 저장된 배열을 출력한다

update_q_mc : epi_data 인자로 받고 여러번의 에피소드를 거친 뒤 해당 정보가 들어있는 epi_data 로 현재도시 다음도시 전체 reward 정보중에 현재도시와 다음도시 겹치는 경우가 있다면 전체 reward 를 평균내서 q value 를 update 해준다.

3.진행 과정

에피소드 개수만큼 for 문을 돌면서 지금찾은 탐색한 도시들을 제외하고 랜덤으로 현재 도시 정하고 경로에 추가해주고 choose_action 으로 epath 에 없는 도시들중에서 next 도시 고르고 generate_episode 함수를 호출해 하나의 에피소드를 만든다 만든다 update_q_mc 에 사용할 각 도시, 상대도시, 전체 리워드의 정보가 담긴 배열을 episode_data 에 추가해준다.

에피소드 개수만큼 for 문을 돌고 난 뒤 update_q_mc 를 호출하고 인자로 중첩된 리스트가 풀린 episode_data 리스트를 전달한다.

argmax 를 통한 Q TABLE 로 경로를 추출한 뒤 calculate_distance 를 호출해서 인자로 그래프, 998 개의 경로가 담긴 순서를 주고 거리를 계산했다.

2.2 TD_Q_TABLE.IPYNB

1. 하이퍼파라미터 설정

learning_rate = 0.1 학습률

discount_factor = 0.9 할인율

epsilon = 0.1 탐험률

num_episodes = 1000 #에피소드 개수

2. 주요 함수

generate_random_city : state 도시를 인자로 받고 state 랑 다르게 랜덤하게 도시 반환

td_generate_max_city : state 도시, next 도시를 인자로 받고 next 도시의 Q table 에서 state 도시, next 도시를 제외한 Q 값이 가장 높은 도시 반환

choose_action : epsilon 을 사용해서 0.1 확률보다 작다면 generate_random_city 의 결과값을 반환하고 0.1 보다 크거나 같다면 현재 state 에서 가장 높은 Q 값을 가지는 도시 반환

td_update_q_table : state, reward, next_state 를 인자로 받고 Q(s, a) = Q(s, a) + learning rate(reward + discount_factor x max(Q(s' a')) - Q(s,a)) 으로 Q table update

3.진행 과정

에피소드 개수만큼 for 문을 돌면서 랜덤으로 현재 도시 정하고 choose_action 으로 next 도시 고르고 choose_reward 함수로 보상 정한 후 td_update_q_table 함수호출해서 Q table update 한다.

argmax 를 통한 Q TABLE 로 경로를 추출한 뒤 calculate_distance 를 호출해서 인자로 그래프, 998 개의 경로가 담긴 순서를 주고 거리를 계산했다.

3 DNN Q-LEARNING

3.1 DNN_Q_LEARNING_1.PY

1. 하이퍼파라미터 설정
   A. EPOCHES : 모델이 전체 데이터 셋을 500 번 반복하여 학습하도록 설정
   B. INT_LR : 학습률을 0.0001 로 설정
   C. PATH_COST_SCALING_FACTOR : 경로 비용을 스케일링하는 데 사용되는 값으로, 여기서는 경로 비용을 0.001 로 스케일링
   D. PCA_COMPONENTS : 주성분 분석(PCA)을 통해 차원을 축소할 때 사용할 주성분의 개수를 설정. 여기서는 100 개의 주성분을 사용
   E. LOG_DIR : TensorBoard 에 기록할 로그 파일의 디렉토리를 설정

2. FC Layer 모델 정의
   A. fc1 : 첫 번째 fc layer. 입력 데이터를 512 차원으로 변환하는 완전 연결층을 정의
   B. fc2 : 두 번째 fc layer. 512 차원의 입력을 받아 256 차원으로 변환
   C. fc3 : 세 번째 fc layer. 256 차원의 입력을 받아 최종적으로 output_dim 크기의 출력을 생성
   D. forward 함수 : 주어진 입력 데이터를 순전파 방식으로 처리하여 출력 반환
      i. 첫 번째 층:
         1. 입력 데이터 x 를 첫 번째 완전 연결층 fc1 에 통과시킴
         2. torch.relu(self.fc1(x))는 fc1 의 출력을 ReLU 활성화 함수에 적용하여 비선형성을 추가
      ii. 두 번째 층:
         1. 첫 번째 층의 출력을 두 번째 완전 연결층 fc2 에 통과시킴
         2. torch.relu(self.fc2(x))는 fc2 의 출력을 ReLU 활성화 함수에 적용
      iii. 세 번째 층:
         1. 두 번째 층의 출력을 세 번째 완전 연결층 fc3 에 통과시킴
         2. self.fc3(x)는 최종 출력 생성

3. 주요 함수
   A. load_and_normalize_q_table 함수 : 저장된 Q-테이블을 파일에서 로드하고, 데이터를 정규화하여 반환. 정규화를 통해 데이터 범위를 일정하게 만들어 학습의 안정성을 높임.
      i. 입력: Q-Table 이 저장된 파일 경로(pkl 파일)
      ii. 출력: Q-Table
   B. get_distance_matrix 함수 : 거리 행렬을 생성하여 반환. 도시 좌표 데이터를 입력으로 받아 각 도시 간의 유클리드 거리를 계산
      i. 입력: 도시 좌표 데이터 (x), 도시의 수 (num_cities)
      ii. 출력: 각 도시 간의 유클리드 거리를 나타내는 거리 행렬 (distance_matrix)
   C. extract_optimal_path 함수 : Q-테이블을 통해 최적 경로를 구성. 시작 도시에서 출발하여 모든 도시를 방문하는 경로를생성
      i. 입력: Q-테이블 (q_table), 시작 도시 (start_city)
      ii. 출력: 최적 경로 (path)
   D. calculate_path_cost 함수 : 경로와 거리 행렬을 통해 경로의 총 비용을 계산하여 반환. 경로를 따라 이동하는 동안 각 도시 간의 거리 합계를 계산
      i. 입력: 경로 (path), 거리 행렬 (distance_matrix)
      ii. 출력: 경로의 총 비용 (cost)
   E. print_partial_path 함수 : path 출력
      i. 입력: 경로, 출력할 경로 수
      ii. 출력: 도시 경로
   F. save_path_to_csv 함수 : 경로를 solution.csv 파일에 저장
      i. 입력: 도시 경로, 저장할 파일 명(default 는 solution.csv)

4. 학습 준비
   A. TSP 도시 정보 로드: 2024_AI_TSP.csv 파일로부터 TSP 도시 좌표를 불러옴
   B. 거리 행렬 생성: 불러온 좌표로부터 거리 행렬 생성
   C. Q-table 로드 및 스케일링: q_table.pkl 파일로부터 q-table 을 불러온 후 정규화 load_and_normalize_q_table 함수를 통해 정규화

D. 학습 전 path 출력: 네트워크로 q-table 학습 전 path 가 어떻게 나오는 지 확인하기 위해 extract_optimal_path 함수로 경로를 추출하고 print_partial_path 로 추출된 경로를 프린트함. calculate_path_cost 함수를 통해 추출된 경로의 distance 를 계산하여 initial_distance 에 저장

E. 모델 초기화: PCA 를 통해 축소된 거리 행렬의 차원을 입력으로 받고, Q-테이블의 크기와 동일한 출력 차원을 가지는 QNetwork 모델을 초기화. 모델은 GPU 혹은 CPU 에 할당되며, Adam 옵티마이저를 사용하여 초기 학습률(INIT_LR)로 최적화.

5. 학습 루프

A. PCA 로 축소된 거리 행렬을 사용하여 각 도시 간의 특성 벡터 생성: PCA 를 통해 축소된 거리 행렬을 텐서로 변환하여 GPU 또는 CPU 로 이동시키고, 각 도시 간의 평균 특성 벡터를 하나의 입력 벡터로 생성.

B. 모델 예측: 생성된 입력 벡터를 모델에 통과시켜 출력 Q-테이블을 예측하고, 이를 2 차원 형태로 변환.

C. target Q-테이블 설정: 현재 가지고 있는 Q-테이블을 타겟 Q-테이블로 설정하여 텐서로 변환하고, GPU 또는 CPU 로 이동.

D. 예측 Q-테이블로 최적 경로 비용 계산: 예측된 Q-테이블을 사용하여 최적 경로를 추출하고, 해당 경로의 비용을 계산.

E. Loss 계산: 가중치 조정을 위한 loss 로 출력으로 나온 q-table 과 현재 보유중인 q-table 의 MSE 를 통해 loss 계산

F. 네트워크 학습: 손실의 역전파를 통해 모델의 파라미터를 업데이트.

G. TensorBoard 기록: 학습 과정에서 경로 비용과 손실 값을 TensorBoard 에 기록.

H. 학습 과정 출력: 현재 에포크, 경로 비용, 손실 값을 출력하여 학습 과정을 모니터링.

I. 최적 경로 업데이트: 예측된 경로 비용이 현재까지 구한 최적 비용보다 작을 경우, 최적 경로와 최적 비용을 업데이트.

3.2 DNN_Q_LEARNING_2.PY

1. 하이퍼 파라미터 설정 : DNN_Q_Learning_1.py 에 추가
   A. STEP_SIZE : 학습률 감소 주기 설정. 100 epoch 당 0.1 씩 감소
   B. LR_DC_RAT : 학습률 감소 비율 설정

2. FC Layer 모델 정의: DNN_Q_Learning_1.py 와 동일

3. 주요 함수: DNN_Q_Learning_1.py 에 추가
   A. get_alpha_beta 함수: 학습 에포크에 따라 손실 계산을 위한 가중치인 alpha 와 beta 값을 동적으로 조정하여 반환
      i. 입력:
         1. epoch: 현재 에포크
         2. total_epochs: 전체 에포크 수
         3. alpha_start: 초기 alpha 값
         4. alpha_end: 최종 alpha 값
         5. beta_start: 초기 beta 값
         6. beta_end: 최종 beta 값
      ii. 출력: 현재 에포크에 해당하는 alpha 와 beta 값

4. 학습 준비: DNN_Q_Learning_1.py 에 추가
   A. 가중치 스케줄링 비율 파라미터: 학습 과정에서 alpha 와 beta 값을 동적으로 조정하기 위한 시작 값과 종료 값을 설정. alpha 는 초기 1.0 에서 0.01 로 감소하고, beta 는 초기 0.0 에서 0.99 로 증가함. 이는 학습 초기와 후기에서 다른 가중치를 적용하여 안정적인 학습을 유도.

5. 학습 루프: DNN_Q_Learning_1.py 부분 변경
   A. Loss 계산 (MSE + 경로 비용 가중치 계산): 손실을 계산하기 위해 MSE 손실과 경로 비용을 가중치에 따라 조합. 에포크에 따라 가중치(alpha, beta)를 동적으로 조정.

4 DEEP Q-NETWORK

4.1 DEEP_Q_NETWORK_4.PY : DQN1, 2, 3 내용 모두 포함

1. FC Layer 모델 정의
   A. fc1 : 첫 번째 fc layer. 입력 데이터를 4*output_dim 차원으로 변환하는 완전 연결층을 정의
   B. fc2 : 두 번째 fc layer. 4output_dim 차원의 입력을 받아 2output_dim 차원으로 변환
   C. fc2_1 : 추가된 두 번째 layer. 2*output_dim 차원의 입력을 받아 output_dim 차원으로 변환
   D. fc3 : 세 번째 fc layer. output_dim 차원의 입력을 받아 최종적으로 output_dim 크기의 출력을 생성 forward 함수 : 주어진 입력 데이터를 순전파 방식으로 처리하여 출력 반환
   E. forward 함수: 주어진 입력 데이터를 순전파 방식으로 처리하여 출력을 반환
      i. 첫 번째 층:
         1. 입력 데이터 x 를 첫 번째 완전 연결층 fc1 에 통과시킴
         2. torch.relu(self.fc1(x))는 fc1 의 출력을 ReLU 활성화 함수에 적용하여 비선형성을 추가
      ii. 두 번째 층:
         1. 첫 번째 층의 출력을 두 번째 완전 연결층 fc2 에 통과시킴
         2. torch.relu(self.fc2(x))는 fc2 의 출력을 ReLU 활성화 함수에 적용
      iii. 추가된 두 번째 층:
         1. 두 번째 층의 출력을 추가된 두 번째 완전 연결층 fc2_1 에 통과시킴
         2. torch.relu(self.fc2_1(x))는 fc2_1 의 출력을 ReLU 활성화 함수에 적용
      iv. 세 번째 층:
         1. 두 번째 층의 출력을 세 번째 완전 연결층 fc3 에 통과시킴
         2. self.fc3(x)는 최종 출력 생성

2. 메모리 클래스 정의: ReplayMemory

   ReplayMemory 클래스는 DQN 학습에서 경험을 저장하고, 필요 시 배치로 샘플링하여 학습에 활용하는 메모리 버퍼를 구현

   A. push 함수 : 메모리의 현재 길이가 최대 용량보다 작으면 새로운 슬롯을 추가. 현재 위치에 경험을 저장하고, 위치를 업데이트. 위치는 순환 방식으로 관리되어 메모리가 꽉 차면 처음 위치로 돌아옴

   B. sample 함수 : 배치 크기만큼의 무작위 경험 샘플을 반환

3. 주요 함수

   A. train_dqn 함수 : Deep Q-Network (DQN) 모델을 학습시키기 위한 함수, 경험 재생 메모리에서 샘플링한 배치 데이터를 사용하여 DQN 모델의 파라미터를 업데이트. 메모리에 저장된 경험이 배치 크기보다 적으면 학습을 진행하지 않음. 샘플링한 데이터를 각각 batch_state, batch_action, batch_return 으로 분리하고 텐서로 변환하여 GPU 또는 CPU 로 이동. current_q_values 는 모델이 예측한 Q-값을 나타내고, expected_q_values 는 배치의 보상 값을 나타냄. batch_q_table 은 Q-테이블에서 가져온 Q-값으로 구성. Loss 를 계산한 후 역전파를 통해 모델의 파라미터 업데이트

      i. 입력

         1. dqn: DQN 모델
         2. memory: 경험 재생 메모리 (ReplayMemory 인스턴스)
         3. q_table: Q-테이블
         4. batch_size: 배치 크기
         5. log_interval: 로깅 간격 (선택적, 기본값 1000)
         6. log_episode: 로깅 에피소드 (선택적, 기본값 0)

      ii. 출력: 손실 값 반환

   B. select_action 함수 : 현재 상태와 방문한 도시 목록을 기반으로 다음 행동을 선택. ε-탐욕 정책(ε-greedy policy)을 사용하여 랜덤 선택과 최적 선택을 번갈아가며 수행. 랜덤 값을 생성하여 ε 보다 작은 경우, 랜덤으로 다음 도시를 선택. 그렇지 않은 경우, 최적 행동을 선택.

i. 입력
    1. state: 현재 상태를 나타내는 벡터.
    2. visited: 이미 방문한 도시들의 집합.
ii. 출력: 선택된 다음 도시의 인덱스

C. find_optimal_path 함수 : 현재 DQN 모델을 사용하여 최적 경로를 추출. 초기 도시에서 출발하여 모든 도시를 방문하는 경로를 생성. 모든 도시를 방문할 때까지 반복. DQN 모델을 사용하여 Q-값을 예측하고, 이를 넘파이 배열로 변환. 가장 높은 Q-값을 가지는 행동을 선택. 다음 도시를 선택. 선택한 도시를 방문한 도시 리스트에 추가. 현재 도시를 업데이트.
i. 입력
    1. dqn: DQN 모델
    2. num_cities: 총 도시의 수
ii. 출력: 방문한 도시들의 최적 경로

D. extract_indices 함수 : 주어진 상태 벡터에서 현재 도시와 방문한 도시들의 인덱스를 추출
i. 입력: 현재 상태를 나타내는 벡터
ii. 출력: 현재 도시의 인덱스 (current_city)와 방문한 도시들의 인덱스 배열 (visited_city)

E. 나머지 calculate_path_cost, load_and_normalize_q_table, euclidean_distance 함수는 DNN_Q_Learning 코드와 동일

4. 하이퍼파라미터 정의
A. SummaryWriter : TensorBoard 에서 학습 과정을 시각화하기 위한 SummaryWriter 객체를 생성하고 로그 디렉토리를 설정
B. num_episodes : 전체 학습 에피소드 수를 설정
C. batch_size : 학습 시 한 번에 처리할 데이터 샘플의 수를 설정
D. gamma : 할인율 (discount factor)로, 미래 보상의 현재 가치를 계산할 때 사용
E. epsilon_start : ε-탐욕 정책에서 랜덤 행동 선택의 초기 확률을 설정
F. epsilon : 현재 학습 단계에서의 입실론 값을 초기화

G. epsilon_end : ε-탐욕 정책에서 랜덤 행동 선택의 최소 확률을 설정
H. epsilon_decay : 각 에피소드마다 입실론 값을 감소시키는 비율을 설정
I. max_memory : 경험 재생 메모리의 최대 크기를 설정
J. learning_rate : 옵티마이저의 학습률을 설정
K. reward_scale : 리워드 값을 스케일링하여 조정

5. 학습 준비
   A. 도시 정보 불러오기 : 도시 정보는 '2024_AI_TSP.csv' 파일에서 불러오며, 각 도시의 좌표를 읽어 리스트에 저장한 후, 유클리드 거리를 계산하여 도시 간 거리 행렬을 생성
   B. Q-Table 로드 및 스케일링 : 'q_table.pkl' 파일에서 Q-테이블을 로드하고, 이를 정규화하여 학습의 안정성을 높임
   C. 모델 초기화 : 도시 수와 방문한 도시의 집합을 포함하는 상태와 현재 상태에서의 Q-값을 출력하는 액션을 정의하여 DQN 모델을 초기화하고, 경험 재생 메모리를 설정한 후, Adam 옵티마이저와 MSE 손실 함수를 사용하여 모델을 최적화
   D. 마지막 layer 의 weight 를 distance matris 로 초기화 : 마지막 레이어의 가중치를 도시 간 거리 행렬로 초기화하여 모델의 초기 상태가 도시 간의 실제 거리를 반영하도록 함. 이 과정에서 가중치와 바이어스는 학습되지 않도록 고정

6. 학습 루프
   A. 현재 도시 정보 초기화 : 각 에피소드의 시작에서 현재 도시를 무작위로 선택하고, 이 도시는 방문한 도시 집합에 추가됨. 상태 벡터를 초기화하고 현재 도시와 방문한 도시를 상태 벡터에 표시. epsilon 값을 갱신하여 탐색과 이용의 균형을 조절.
   B. 다음 action 선택 및 reward 계산 : ε-탐욕 정책을 사용하여 현재 상태에서 다음 행동(도시)을 선택. 선택한 행동에 따라 다음 도시로 이동하고, 이동한 거리의 음수를 보상으로 계산. 현재 상태, 행동, 보상, 다음 상태의 경험을 에피소드 메모리에 저장. 방문한 도시 집합에 다음 도시를 추가하고 상태 벡터를 갱신. 이 과정을 모든 도시를 방문할 때까지 반복.

C. 마지막에 시작 도시고 돌아가기 : 모든 도시를 방문한 후, 마지막 도시에서 시작 도시로 돌아가는 경로를 추가로 계산. 이 때의 보상을 계산하고, 마지막 상태-행동-보상-상태 쌍을 에피소드 메모리에 저장. 전체 에피소드의 총 보상을 계산.

D. target 계산 : 에피소드 메모리에 저장된 각 상태-행동-보상-다음 상태 쌍에 대해 타겟 Q-값을 계산. 다음 상태에서의 최대 Q-값을 추출하고, 이를 할인된 보상과 현재 보상을 더하여 타겟 값을 만듦. 이 타겟 값은 미래 보상을 고려한 현재 Q-값의 기대치.

E. target 스케일링 및 메모리에 추가 : 타겟 Q-값을 [0, 1] 범위로 스케일링하여 안정성을 높임. 타겟 값이 모두 동일한 경우, 모든 타겟 값을 0.5 로 설정하여 스케일링을 방지. 스케일링된 타겟 값을 경험 재생 메모리에 추가.

F. 네트워크 학습 : 경험 재생 메모리에서 배치 샘플을 가져와 DQN 모델을 학습. train_dqn 함수를 호출하여 현재 배치에 대한 손실을 계산하고, 역전파를 통해 모델 파라미터를 업데이트. 학습이 끝난 후, 현재 네트워크를 사용하여 최적 경로를 찾아 경로 비용을 계산.

G. TensorBoard 에 기록 : 현재 에피소드의 손실, 총 보상, 경로 비용을 TensorBoard 에 기록

H. 학습 결과 출력 : 현재 에피소드의 손실 값, 총 보상, 경로 비용을 출력. 10 에피소드마다 최적 경로를 요약하여 출력하여 학습 진행 상황을 확인. 최종적으로 모든 에피소드가 끝난 후, TensorBoard 기록을 마침.