Reinforcement learning
Assignment N.1

April 7, 2025

Bar, Alon
205476013

---

1 Theory

1.1 Longest Common Subsequence

1. Write a dynamic programming solution for the following problem.

   Given: Two sequences (or strings) $X(1:m)$ and $Y(1:n)$.

   Goal: Return the length of the longest common subsequence of both X and Y (not necessarily contiguous).

Ans:

Algorithm:

1. Create a 2D array L of size $(m+1) \times (n+1)$ and initialize all entries to 0.
2. For $i = 1, \ldots, m$:
   (a) For $j = 1, \ldots, n$:
      i. if $X(i) = Y(j)$, set $L(i,j) = L(i-1, j-1) + 1$.
      ii. Else, set $L(i,j) = \max(L(i-1,j), L(i,j-1))$.
3. Return $L(m,n)$.

Runtime:

The algorithm iterates through an array of size $(m+1) \times (n+1)$, performing $O(1)$ operations per cell. Therefore, the overall time complexity is $O(m \times n)$.

1.2 Moses the mouse

1. Formulate the problem as a finite horizon decision problem: Define the state space, the action space and the cumulative cost function.

Ans:

State Space:

Define the state space as $S = \{(i, j) \mid 1 \leq i \leq M,\ 1 \leq j \leq N\}$,

where the pair (i, j) represents the room in the i^{th} row and j^{th} column.

Action Space:

At any state (i, j), Moses can only move *North* or *East* (provided the move stays within the apartment).

$$A(i,j) = \begin{cases} \{\emptyset\} & \text{if } i = M \text{ and } j = N, \\ \{\text{North}\} & \text{if } i = M \text{ and } j < N, \\ \{\text{East}\} & \text{if } j = N \text{ and } i < M, \\ \{\text{North},\ \text{East}\} & \text{if } i < M \text{ and } j < N. \end{cases}$$

Cumulative Reward(Cost) Function:

Let $r(s)$ be the reward obtained upon entering state s, where

$$c(s) = \begin{cases} 1, & \text{if room } s \text{ contains cheese,} \\ 0, & \text{otherwise.} \end{cases}$$

For a given path (or sequence of states) $\pi = (s_1, s_2, \ldots, s_T)$, the cumulative reward is defined as

$$R(\pi) = \sum_{t=1}^{T} r(s_t)$$

Alternatively, if we view the decision problem in stages with a per-stage reward function $r_t(s_t, a_t)$, we write:

$$R(\pi) = \sum_{t=1}^{T-1} r_t(s_t, a_t) + r_T(s_T),$$

with

$$r_t(s_t, a_t) = \begin{cases} 1, & \text{if } s_t \text{ contains cheese,} \\ 0, & \text{otherwise.} \end{cases}$$

$$r_T(s_T) = \begin{cases} 1, & \text{if } s_T \text{ contains cheese,} \\ 0, & \text{otherwise.} \end{cases}$$

Cumulative cost $(\pi) = -R(\pi)$

2. What is the horizon of the problem?

Ans: The horizon of the problem refers to the total number of steps (or time steps) required to reach the goal.

Since Moses starts at the south-west corner $(1, 1)$ and must reach the north-east corner (M, N), he needs to make:

- $N - 1$ moves to the east.
- $M - 1$ moves to the north.

Therefore, the total number of steps is:

$$Horizon = (M - 1) + (N - 1) = M + N - 2.$$

3. How many possible trajectories are there? How does the number of trajectories behaves as a function of N when M =2? How does it behave as a function of N when M = N? please note that you don't need to calculate the exact number of states, you can give the order number (this also apply to the rest of this question).

Ans: At each step, he makes either a North or East move. Therefore, the number of trajectories is the number of distinct permutations of $M - 1$ North moves and $N - 1$ East moves.

In other words, we need to choose M - 1 places to move north out of N + M - 2 possible options.

$$\#\text{ Trajectories} = \binom{M + N - 2}{M - 1}$$

When M = 2:

$$\#\text{ Trajectories} = \binom{N}{1} = N$$

When M = N:

$$\#\text{ Trajectories} = \binom{2N - 2}{N - 1}$$

4. Aharon, Moses's long lost war-buddy woke up confused next to Moses and decided to join him in his quest (needless to say, both mice suffer the same rare head injury).

(a) Explain what will happen if both mice ignore each other's existence and act 'optimal' with respect to the original problem.

Ans: If both mice act optimally without coordination, they may end up taking the exact same path and arriving at the same pieces of cheese simultaneously. While this might be sociable, it's inefficient: they could leave behind some cheese that neither of them can reach anymore (e.g., rooms that are to the south-west of both paths). As a result, the total amount of collected cheese may be suboptimal.

(b) Assume both mice decided to coordinate their efforts and split the loot. How many states and actions are there now?

Ans: Action space:
Each mouse can choose from two actions: North or East (ignoring edge cases for now). So the joint action space is:

$$A = \{(a_1, a_2) \mid a_1, a_2 \in \{\text{North}, \text{East}\}\}$$

Therefore,

$$|A| = 2 \times 2 = 4.$$

State space:
A state must now describe the positions of both mice, i.e., their row and column indices. So we define the state space as:

$$S = \{(i_1, j_1, i_2, j_2) \mid 1 \leq i_1, i_2 \leq M;\ 1 \leq j_1, j_2 \leq N\}$$

and thus the total number of states is:

$$|S| = M \times N \times M \times N = (M \cdot N)^2.$$

But we an constraint both mice must be in the same step:
So the state space is:

$$S = \{(i_1, j_1, i_2, j_2) \mid 1 \leq i_1, i_2 \leq M;\ 1 \leq j_1, j_2 \leq N;\ i_1 + j_1 = i_2 + j_2\}$$

define t = i + j, P(t) = # (i, j)— i + j = t + 2
$|S| = \sum_{t=0}^{M+N-2} P(t)^2$ where $P(t) = min(t + 1, N)$

(c) Now their entire rarely-head-injured division has joined the journey. Assume there's a total of K mice, how many states and actions are there now?

Ans: Action space:

Each mouse can choose from two actions: North or East (ignoring edge cases for now). So the joint action space is:

$$A = \{(a_1, .., a_k) \mid a_1, ..a_k \in \{\text{North}, \text{East}\}\}$$

Therefore,

$$|A| = 2^k$$

State space:

A state must now describe the positions of both mice, i.e., their row and column indices. So we define the state space as:

$$S = \{(i_1, j_1, ..., i_k, j_k) \mid 1 \leq i_1, ..., i_k \leq M;\ 1 \leq j_1, ..., j_k \leq N\}$$

and thus the total number of states is:

$$|S| = (M \cdot N)^k$$

But we an constraint all mice must be in the same step:
So the state space is:

$$S = \{(i_1, j_1, i_2, j_2) \mid 1 \leq i_1, i_2 \leq M;\ 1 \leq j_1, j_2 \leq N;\ i_1 + j_1 = i_2 + j_2\}$$

define t = i + j, P(t) = # (i, j)— i + j = t + 2
$|S| = \sum_{t=0}^{M+N-2} P(t)^k$ where $P(t) = min(t + 1, N)$

1.3 Language model

1. Find the probability of the following words: 'Bob', 'Ok', 'B', 'Book', 'Booook'

Ans:

$Bob = 0.25 * 0.2 * 0.325 = 0.01625$

Ok = 0 because (a) Every word starts with the letter 'B'

B = 0.325

$Book = 0.25 * 0.2 * 0.2 * 0.2 = 0.002$

$Booook = 0.25 * 0.2^3 * 0.2 * 0.2 = 0.00008$

2. We wish to find the most probable word in the language of length K.

   (a) Formulate the problem as a finite horizon decision problem: Define the state space, the action space and the multiplicative cost function.

   Ans:
   $A = \{B, K, O, -\}$ choosing the next letter
   $S_1 = \{B\}$
   $S_{t\neq 1} = \{B, K, O, -\}$ The current letter
   $C = \prod_{i=1}^{K-1} C_t(s_t, a_t) C_T(s_T)$ where $C_t = p(a_t|s_t) = P(l_{t+1}|l_t)$
   and $C_T(s_T) = -1$

   (b) Bound the complexity of finding the best combination.

   Ans: A brute-force search would require checking all possible sequences of actions over the horizon. If there are K decision steps and the action space has size $|A|$, the total number of sequences is $|A|^K$.
   However, we can use the Viterbi algorithm to find the best sequence in time

$$O\Big(K \cdot |A| \cdot |S|\Big) = O(K),$$

   Viterbi Algorithm Sketch:
   Define $P(i, s)$ as the best cost up to decision step t when the system is in state s. Also, define $Q(t, s)$ to store the state at step $t - 1$ that yielded the best cost for state s at step i.

   - **Initialization:** For the starting state s_0 (e.g., B):

$$P(0, B) = -1, \quad P(0, s) = 0 \quad \text{for } s \neq B.$$

   - **Recursion:** For each time step $i = 1, \ldots, K - 1$ and for each state $s \in S$,

$$P(t, s) = \min_{m \in S} \Big\{ C\big(s \mid m\big) \cdot P(t - 1, m) \Big\},$$

and record the optimal previous state:

$$Q(t,s) = \arg\min_{m \in S} \Big\{ C\big(s \mid m\big) \cdot P(i-1, m) \Big\}.$$

Here, $C(s \mid m)$ is the multiplicative cost for moving from state m to state s.

- **Termination:** At the final step, if the terminal state is fixed (say, $s_T = -$), we set

$$\text{Optimal cost} = \min_{s \in S} \Big\{ C\big(- \mid s\big) \cdot P(K-1, s) \Big\}.$$

- **Backtracking:** Starting from the terminal state, use the matrix Q to reconstruct the optimal path.

Since we iterate over K time steps, and at each step we consider all $|S|$ states and each state has $|A|$ possible transitions, the overall time is

$$O\Big(K \cdot |S| \cdot |A|\Big).$$

$|S|$ and $|A|$ are constant (or very small), the algorithm runs in linear time $O(K)$ with respect to the horizon.

(c) Find a reduction from the given problem to an analogous problem with additive cost function instead.

Ans: The original problem is to generate a word with maximum probability, given by the multiplicative cost:

$$C(\text{word}) = -\prod_{t=1}^{K} p(a_t \mid s_t),$$

where $p(a_t \mid s_t)$ is the probability of choosing letter a_t when the current letter is s_t.

By taking the negative logarithm, we can transform this product into a sum. Define the per-step additive cost as:

$$r_t(s_t, a_t) = \log\big(p(a_t \mid s_t)\big).$$

Then, the cumulative cost of a word becomes:

$$R(\text{word}) = \sum_{t=1}^{K} r_t(s_t, a_t) = \sum_{t=1}^{K} \log\big(p(a_t \mid s_t)\big).$$

Hence, maximizing $R(\text{word})$ is equivalent to minimizing $C(\text{word})$.

(d) Explain when each approach (multiplicative vs. additive) is preferable. Hint: Think of the consequence of applying the reduction on the memory representation of a number in a standard operating system.

Ans: In the original multiplicative formulation, the probability of a sequence is given by a product:

$$C(\text{word}) = \prod_{t=1}^{K} p(a_t \mid s_t).$$

This formulation is natural when combining independent probabilities. However, because the individual probabilities are usually small (between 0 and 1), multiplying many of them together often results in extremely small numbers. Standard floating-point representations in operating systems have limited precision, so these products can underflow or lose accuracy.

On the other hand, by taking the negative logarithm, we transform the product into a sum:

$$R(\text{word}) = \sum_{t=1}^{K} \log\big(p(a_t \mid s_t)\big).$$

The additive formulation is numerically more stable since summing avoids the severe underflow issues that can occur with products. Additionally, addition is generally more efficient on standard hardware. As a result, for practical implementations—especially when the horizon K is long—the additive approach (working with log-probabilities) is preferable, both in terms of numerical stability and memory representation.

(e) Write a code in Python which finds the most probable word of a given size using dynamic programming. What is the most probable word of size 5?

Ans: The most probable word of size 5 is "BKBKO" with a probability of 0.00675. Python file named "language_model.py" is submitted separately.

1.4 Minimum mean cost cycle

1. Calculate $d_i(v)$ for every $i \in 0, ..., n = 5$ and every $v \in A, B, C, D, E$

v	d_0	d_1	d_2	d_3	d_4	d_5	$max\frac{d_n(v)-d_k(v)}{n-k}$
A	0	∞	∞	6	10	7	7/5
B	∞	-1	∞	2	-1	5	6
C	∞	∞	2	6	3	2	0
D	∞	∞	1	-2	4	1	3/2
E	∞	∞	1	∞	4	1	0

2. Use Karp's theorem to find the cost of the minimum mean cost cycle

Ans: $\mu* = min_{v \in V} max_{0 \leq k \leq n-1} \frac{d_n(v)-d_k(v)}{n-k} = 0$

3. What is the optimal average cost of this DDP?

Ans: For a Discrete Dynamic Programming (DDP) problem, the optimal average cost is equal to the minimum mean cost cycle value, which we calculated as $\mu^* = 0$

Therefore, the optimal average cost of this DDP is 0.

2 Programming

2.1 MNIST

1. Implement the training and evaluation loop currently marked as "TODO" comments in the code. Use the already defined model and dataset. Make sure you report the loss on the training set each batch and the accuracy on the evaluation set at the end of training.

Ans: Code implemented under `mnist.py`, Figure 1 reports the loss on the training set each batch (red lines) for each step (x-axis), test accuracy reported on the top of the figure.

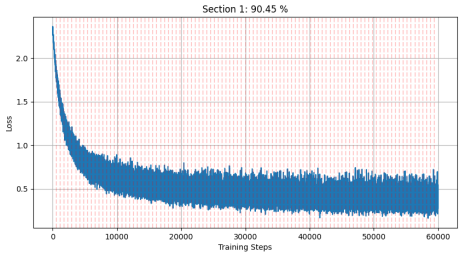

Figure 1: MNIST Section 1 Training Loss

2. Find a better optimization configuration that works well (learning rates, mini-batch size, no. of epochs and so on). You are provided with some starter configuration options for SGD. However, more options are possible and the values provided are not necessarily optimal. With a better choice of parameters, you will converge much faster and achieve better accuracy.

   The new configuration should result faster (in terms of #epochs) convergence. Optional: You are not limited to SGD and are encouraged to try other optimization methods, see PyTorch optim package.

Ans:

Implemented under section 2 in the code, first we tried to change the learning rate to 1e-2 instead of 1e-3 (Figure 2) and then we tried adam optimizer (Figure 3).

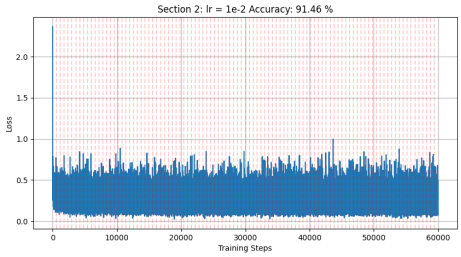

Figure 2: MNIST Section 2 - learning rate change

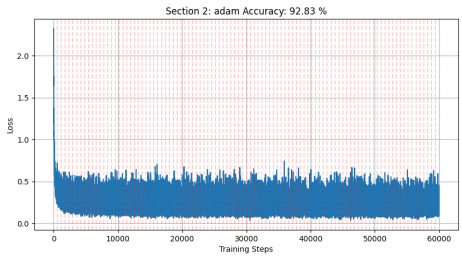

Figure 3: MNIST Section 2 - Adam optimizer

3. Train a deeper model by introducing a ReLU non-linearity and another linear layer. The size of the hidden layer should be 500.

Ans: Implemented under section 4 in the code (Figure 5).

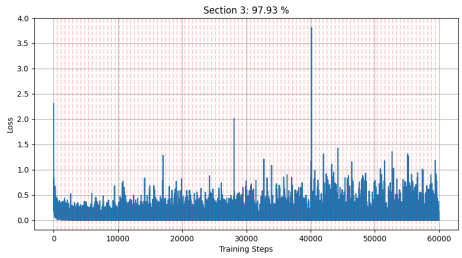

Figure 4: MNIST Section 2 - ReLU + hidden layer

2.2 OpenAI Gym

1. Familiarize yourself with OpenAI Gym by reviewing the short introduction provided.

Ans: Completed in `cart_pole.py`.

2. Implement the agent.

Ans: Implemented in the function `weighted_action` in `cart_pole.py`.

3. Evaluate your agent by running an episode of the environment. An episode ends when the pole drops or after 200 steps. The agent's score should be the accumulated reward over all steps in the episode.

Ans: Implemented in the function `evaluate_episode` in `cart_pole.py`.

4. Train your agent using random search. This involves sampling different weight vectors multiple times and greedily choosing the one that achieves the highest reward. You should sample at least 10,000 times.

Ans: Implemented in the function `random_search` in `cart_pole.py`.

5. Evaluate the random search scheme. Run the search 1,000 times and record, for each run, the number of episodes required for the agent to achieve a score of 200. Plot a histogram of the number of episodes required and report the average.

Ans: Implemented in the function `evaluate_random_search` in `cart_pole.py`.

Average number of episodes needed: 12.35

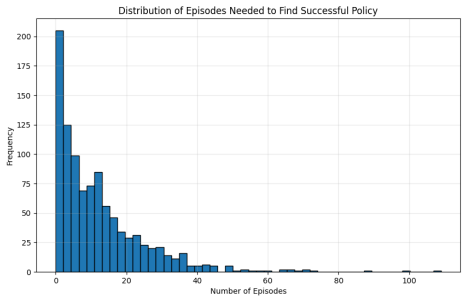

Figure 5: Histogram of number of episodes required until reaching a score of 200