Reinforcement Learning

Student name and id: Nicklas Hansen (s153077)

Chapter: 4, 5

Exercise 4.1

$q_\pi(11, \text{down}) = 0$ as the reward for transitioning to a terminal state is zero by definition and the state-value function at the terminal state is also zero by definition.
For $q_\pi(7, \text{down})$, we know that $q_\pi(s, a) = \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a]$, so

$$q_\pi(7, \text{down}) = -1 - 14\gamma, \text{ for } k = \infty$$

as shown in Figure 4.1 of the book.

Exercise 4.2

Using equation (4.4) from the book, we can compute the value function for timestep $k = \infty$ (using values from the converged example) for the new state 15 by summing over its possible transitions. As we know the policy is a equiprobable random policy, we get:

$$\begin{aligned} v_\pi(15) &= 0.25(-1 - 22\gamma) + 0.25(-1 - 20\gamma) + 0.25(-1 - 14\gamma) + 0.25(-1 + \gamma\ v_\pi(15)) \\ &= -1 - 14\gamma + 0.25\gamma\ v_\pi(15) \end{aligned}$$

If we change the action `down` for state 13 as well, we now have a system of 2 linear equations that we need to solve. Using equation (4.5) again, for $v_k(15)$ we now have

$$\begin{aligned} v_\pi(15) &= 0.25(-1 + \gamma v_\pi(12)) + 0.25(-1 + \gamma v_\pi(13)) + 0.25(-1 + \gamma v_\pi(14)) + 0.25(-1 + \gamma v_\pi(15)) \\ &= -1 - 9\gamma + 0.25\gamma\ v_\pi(13) + 0.25\gamma\ v_\pi(15) \end{aligned}$$

and for $v_k(13)$ we get

$$\begin{aligned} v_\pi(13) &= 0.25(-1 + \gamma v_\pi(12)) + 0.25(-1 + \gamma v_\pi(9)) + 0.25(-1 + \gamma v_\pi(14)) + 0.25(-1 + \gamma v_\pi(15)) \\ &= -1 + 0.25\gamma(-56) + 0.25\gamma\ v_\pi(15) \end{aligned}$$

which gives us the following system of equations if we isolate the value functions:

$$\begin{aligned} -0.25\gamma\ v_\pi(13) + v_\pi(15) - 0.25\gamma\ v_\pi(15) &= -1 - 9\gamma \\ v_\pi(13) - 0.25\gamma\ v_\pi(15) &= -1 - 14\gamma \end{aligned}$$

which is solved in Maple, so we end up with

$$v_\pi(13) = -\frac{-20\gamma^2 + 224\gamma + 16}{\gamma^2 + 4\gamma - 16}$$

and

$$v_\pi(15) = \frac{4(14\gamma^2 + 37\gamma + 4)}{\gamma^2 + 4\gamma - 16}$$

As these two solutions are rather complicated, we should do a sanity check. Assuming $\gamma = 1$ we get $v_\pi(13) = -20$ and $v_\pi(15) = -20$, which are rather plausible values, so we conclude that this is a solution.

Exercise 4.3

Similarly to what is done in (4.5) of the book, where the value function update rule is formulated based on the Bellman equation, we can do the same for the action-value function:

$$\begin{aligned} q_{k+1}(s,a) &= \sum_{s',r} p(s',r \mid s,a)[r + \gamma \max(q_k(s',a'),a')] \\ &= \sum_{s'} r_{s',a}\ p(s',r \mid s,a) + \gamma \sum_{s'} \sum_{a'} q_k(s',a')\pi(s',a') \end{aligned}$$

where q_k is the action-value at iteration k for state s' and action a' and $\texttt{max}(q,a)$ means that q is maximized for a.

Exercise 4.4

The problem in the code is that if we have two actions that are equally good, the `argmax` function selects any of the two actions with equal probability. For two actions, this leaves us a probability of $1/2^t$ for not terminating in t timesteps. Although the probability approaches 0 as $t \to \infty$, we are not guaranteed to terminate.
One way to fix that issue is to make the `argmax` deterministic, such that it always selects the same action given the same set of equally good actions. If we simply select the first action in the set, we may unintentionally impact the exploration, so another way to solve the problem is to compute the hash of all equally good actions (using any hash function from the family of universal hash functions) and simply pick the action with the lowest hash value.

Exercise 4.5

Conceptually, the two algorithms are very similar. When considering the action-value function instead, we just insert the formula derived in exercise 4.3:

(1) **Initialization.** $v(s) \in \mathbb{R}$ and $\pi(s) \in A(s)$ arbitrarily for all $s \in S$.

(2) **Policy evaluation.**
Loop until $\Delta < \theta$:
$\Delta \longleftarrow 0$
Loop for each $s \in S$:
Loop for each $a \in A$:
$q \longleftarrow Q(s,a)$
$Q(s,a) \longleftarrow \sum_{s'} r_{s',a}\, p(s', r \mid s, a) + \gamma \sum_{s'} \sum_{a'} Q_\pi(s', a')\pi(s', a')$
$\Delta \longleftarrow \max(\Delta, |q - Q(s)|)$

(3) **Policy improvement.**
policy-stable$\longleftarrow$*true*
Loop for each $s \in S$:
Loop for each $a \in A$:
old-pi $\longleftarrow \pi(s,a)$
$\pi(s,a) \longleftarrow$ `argmax`$(Q_\pi(s,a),(s,a))$
If *old-pi* $\neq \pi(s,a)$ then *policy-stable* $\longleftarrow$ *false*
If *policy-stable* then return $Q \approx q_*$ and $\pi \approx \pi_*$; otherwise go to step 2

Exercise 4.6

As suggested, we consider changes to the policy iteration algorithm steps in descending order.

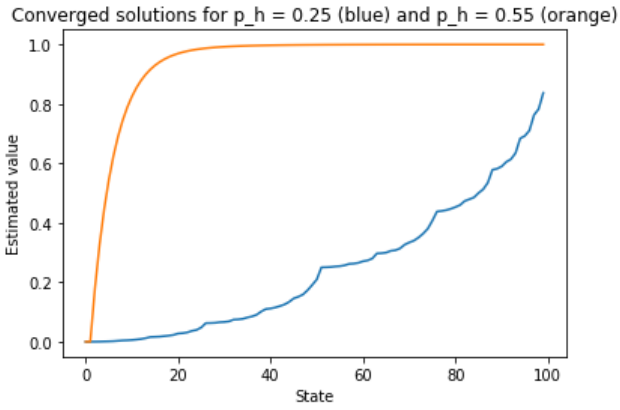
(1) For policy improvement, we want to restrict the algorithm to only produce probabilities that are at least $\epsilon/|A(s)|$. To do so, one could introduce another step in the policy improvement that redistributes probabilities such that the probability of all actions in a given state are at least $\epsilon/|A(s)|$. One way of achieving this is to give the greedy action a probability of selection of $1 - \epsilon + \frac{\epsilon}{|A(s)|}$ and all other actions the probability $\frac{\epsilon}{|A(s)|}$, which can be achieved by defining a piece-wise function.

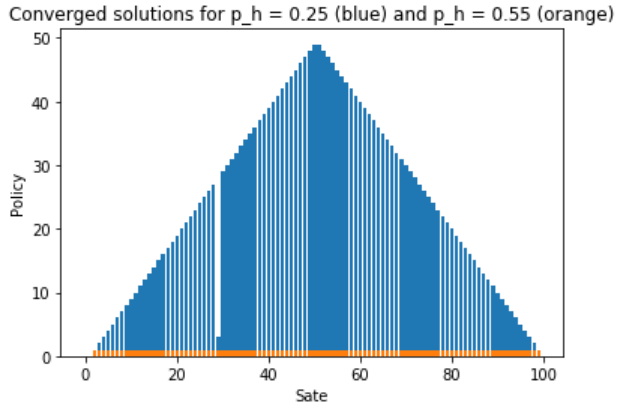
(2) In the policy evaluation loop, we need to consider the probability of selecting each of the possible actions in state s. This is achieved by changing the update rule for $V(s)$ to:
$V(s) \longleftarrow \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, \pi(s))[r + \gamma V(s')]$

(3) In order to satisfy the ϵ-soft requirement initially, we should initialize $\pi(s)$ such that we only consider ϵ-soft policies rather than selecting π arbitrarily.

Exercise 4.9 *(programming)*

Code snippets and outputs from associated notebook is appended to the end of this hand-in. For $p_h = 0.25$ and $p_h = 0.55$ we get the following results:





and we observe that the policy takes great chances when the probability of winning is small, whereas it steadily bets just 1 dollar in all states given a probability of more than half for winning each bet. The reason for the pyramid-shaped policy for $p_h = 0.25$ is that it stakes as much as possible due to bad chances of winning and then gradually bets less as it will achieve 100$ with just one more successful bet.

Exercise 4.10

As the action-value function explicitly states a desired action a rather than maximize over the possible actions like in the state-value function, we update q based on the selected action:

$$\begin{aligned} q_{k+1}(s,a) &= \mathbb{E}[R_{t+1} + \gamma\ q_k(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r \mid s, a)[r + \gamma\ q_k(s', a)] \end{aligned}$$

Difference between policy evaluation, policy improvement, policy iteration and value iteration

Policy evaluation is the task of iteratively evaluating a state-value function v for all states in the state space such that v accurately estimates the value of being in a given state assuming that a policy π is followed from that state onward. At each iteration, the difference between true value and estimated value in all states should decrease and at some point the estimated value function actually converges to the true value function (no matter the initialization of v).
Policy improvement concerns changing the policy π based on a computed state-value function v. If our evaluation of v has converged to the true value of all states, then we can assume that modifying our policy greedily with respect to the value function will produce an improvement over the previous (arbitrarily chosen) policy. In fact, if we know the true value of all states, we can compute the optimal action for all states in the state space and thus produce the optimal policy for a given problem.
In practice, we utilize the *policy iteration* algorithm, which effectively combines policy evaluation and policy improvement into an iteration cycle. In that way, the algorithm iteratively evaluates the current policy, adjusts it according to the new state-value estimates and then repeats the process of evaluation again. At some point in time, the estimated value function will converge to the true value function (like before), which will result in no change for the policy - it is thus already optimal. Therefore, we know that we can run the algorithm until no changes are made to the policy in two consecutive iterations under the guarantee that the algorithm will converge eventually.
It turns out that it is not always feasible to first let the policy evaluation converge to the true state-values for all states and then alter the policy accordingly as it can be computationally expensive to perform a full policy evaluation whenever we adjust our policy. As such, one can apply *value iteration*, which essentially means that we only perform one iteration of the policy evaluation on each cycle, rather than letting it converge first. This obviously means that both the state-value function and the policy changes each cycle without actually knowing the true state-values, but it turns out we actually still can give some guarantees of convergence.

Exercise 5.1

In Figure 5.1 of the book, the estimated value function jumps up for the last two rows in the rear because they correspond to the two states in which the player's sum is 20 or 21. The considered policy determines that the player sticks if the player's sum is 20 or 21, and otherwise hits.
In the event that the player's sum is less than 20, the policy always demands a hit, which potentially makes the player go bust. If the player does not go bust and instead draws another card which does not bring the policy to stick, then the probability of going bust on the following draw increases further. If the player's sum, however, is 20 or 21, the policy demands a stick, which means that the player stops with fairly high chances of winning and no chance of losing (-1). As such, we should expect a value close to 1 in those states.
It is observed that the frontmost values are higher in the upper diagram (usable ace) than the lower diagram (no usable ace). To start off this argument, note that the value also decreases for all scenarios in which the dealer shows either an ace or a face card. If the dealer shows such a card, we know that the probability of the dealer holding a good hand (20 or 21) is higher than if he showed e.g. a 7. Similarly, we can infer that if the player holds a usable ace, then there's only three other aces left and thus a lower probability for the dealer to have one as well. We thus observe a comparatively higher value in states with a usable ace, except for the states in which the dealer also shows an ace.

Exercise 5.2

We have 200 states representing the player's sum, the dealer's shown card and whether or not a usable ace is held. As the player is forced to take an action each turn, which will either increase their sum (transition to a new state) or terminate the sequence of player actions, the probability of visiting the same state twice is zero. Therefore, we should not expect significantly different results using the every-visit Monte Carlo method rather than the first-visit Monte Carlo method shown in Figure 5.1 of the book.

Exercise 5.3

Similarly to the backup diagram for state-value estimation ($v\pi$), the backup diagram for action-value estimation ($q\pi$) is a single, long trajectory starting at a root node representing a state-action pair (s, a) to be updated and ending at the termination of the episode. On each node of the trajectory, we have state-action pairs corresponding to action a_i taken in state s_j. As a Monte Carlo backup diagram shows only the state-action pair transitions that have been taken in a given episode, we end up with a trajectory consisting of single transitions from one state-action pair to another state-action pair, and each state-action pair is updated independently from each other.

Exercise 5.4

We know from dynamic programming that the update rule for an iteratively computed mean action-value is

$$Q(S_t, A_t) = Q(S_t, A_t) + \frac{(G - Q(S_t, A_t))}{t+1}$$

which can be used to update the Monte Carlo ES algorithm pseudo-code to avoid storing a list and computing average of that list over an over.
To do so, we simply modify the innermost conditional expression such that it is compressed to the following:

Unless the pair $S_t,\ A_t$ appears in $S_0, A_0, S_1, A_1, \ldots, S_{t-1}, A_{t-1}$:

$Q(S_t, A_t) = Q(S_t, A_t) + \frac{(G - Q(S_t, A_t))}{t+1}$

$\pi(S_t) = \text{argmax}_a\ Q(S_t, a)$

where $Q(S_t, A_t)$ still is the average return and G also still denotes the return experienced in that episode.

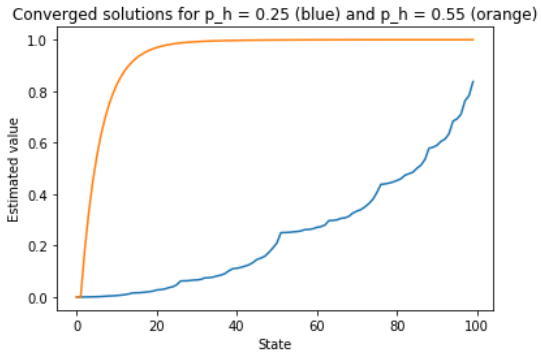
```
states_max = 100

def value_iteration_for_gamblers(p_h, theta=0.0001, gamma=1.0):
    """
    Args:
        p_h: Probability of the coin coming up heads
    """

    def one_step_lookahead(s, V, rewards):
        """
        Helper function to calculate the value for all action in a given state.

        Args:
            s: The gambler’s capital. Integer.
            V: The vector that contains values at each state.
            rewards: The reward vector.

        Returns:
            A vector containing the expected value of each action.
            Its length equals to the number of actions.
        """

        A = np.zeros(s)

        # for each possible action
        for a in range(1, s):

            win  = min(states_max, s+a)
            loss = max(0, s-a)

            # compute  value
            A[a] = p_h * (rewards[win] + gamma * V[win]) + (1 - p_h) * (rewards[loss] + gamma * V[loss])

        return A


    V = np.zeros(states_max+1)
    rewards = np.zeros(states_max+1)
    rewards[-1] = 1
    B = np.zeros(states_max)
```

```
    # Loop for a large, finite number of iterations
    for i in range(2**12):

        # Reinitialize delta
        delta = 0

        # Loop over all states
        for s in range(1, states_max):

            # Compute best action value
            A = one_step_lookahead(s, V, rewards)
            v = np.max(A)

            # Save best action as policy for s
            B[s] = np.argmax(A)

            # Get maximum change in iteration
            delta = max(delta, abs(v - V[s]))

            # Update value function
            V[s] = v

        # If insufficient change, we have converged
        if delta < theta:
            print(f'Converged after {i} iterations using p_h = {p_h}.\n')
            break

    return B, V
    #return policy, V
```

```python
def run(p_h = 0.25):
    print('Running...')

    policy, v = value_iteration_for_gamblers(p_h)
    print(f'Optimized Policy:\n{policy}\n\nOptimized Value Function:\n{v}\n')

    plt.plot(range(states_max), v[:100])
    plt.title(f'Converged solutions for p_h = 0.25 (blue) and p_h = 0.55 (orange)')
    plt.xlabel('State')
    plt.ylabel('Estimated value')

run()
run(0.55)
```

```
Running...
Converged after 6 iterations using p_h = 0.25.

Optimized Policy:
[  0.   0.   1.   2.   3.   4.   5.   6.   7.   8.   9.  10.  11.  12.  13.
  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.   3.
  29.  30.  31.  32.  33.  34.  35.  36.  37.  38.  39.  40.  41.  42.  43.
  44.  45.  46.  47.  48.  49.  49.  48.  47.  46.  45.  44.  43.  42.  41.
  40.  39.  38.  37.  36.  35.  34.  33.  32.  31.  30.  29.  28.  27.  26.
  25.  24.  23.  22.  21.  20.  19.  18.  17.  16.  15.  14.  13.  12.  11.
  10.   9.   8.   7.   6.   5.   4.   3.   2.   1.]

Optimized Value Function:
[  0.00000000e+00   0.00000000e+00   6.10351562e-05   2.89916992e-04
   7.44819641e-04   1.15966797e-03   1.78837776e-03   2.97927856e-03
   4.08935547e-03   4.67300415e-03   5.67722321e-03   7.17885792e-03
   9.09274817e-03   1.19171143e-02   1.57051086e-02   1.63574219e-02
   1.70869827e-02   1.87187465e-02   2.03933486e-02   2.27088928e-02
   2.74810791e-02   2.87377092e-02   3.12662125e-02   3.63709927e-02
   3.96584154e-02   4.76684570e-02   6.25000000e-02   6.28204346e-02
   6.35644794e-02   6.54296875e-02   6.63222491e-02   6.83624148e-02
   7.42187500e-02   7.49345624e-02   7.69197345e-02   8.15759722e-02
   8.49647522e-02   9.08500552e-02   1.01741096e-01   1.09950066e-01
   1.11777857e-01   1.14950837e-01   1.19812012e-01   1.25090197e-01
   1.33711755e-01   1.45483971e-01   1.51233789e-01   1.58633662e-01
   1.73072234e-01   1.90718015e-01   2.09285846e-01   2.50045776e-01
   2.50558615e-01   2.51341283e-01   2.53067017e-01   2.54257917e-01
   2.56819561e-01   2.61778831e-01   2.62815237e-01   2.65295011e-01
   2.70610809e-01   2.73449659e-01   2.79743812e-01   2.96875000e-01
   2.97673360e-01   2.99741687e-01   3.05664062e-01   3.07689801e-01
   3.13723564e-01   3.26305822e-01   3.33833393e-01   3.39859009e-01
   3.50283816e-01   3.63425341e-01   3.79804175e-01   4.06964385e-01
   4.37918961e-01   4.39800262e-01   4.42614671e-01   4.47111428e-01
   4.52958107e-01   4.59807859e-01   4.73255020e-01   4.79248047e-01
   4.85292673e-01   5.00375045e-01   5.12712862e-01   5.34853132e-01
   5.78439221e-01   5.81961003e-01   5.89718580e-01   6.04941265e-01
   6.13969505e-01   6.34534647e-01   6.83829416e-01   6.92288935e-01
   7.10477129e-01   7.62872062e-01   7.82857846e-01   8.37143385e-01
   0.00000000e+00]
```

```
Running...
Converged after 508 iterations using p_h = 0.55.

Optimized Policy:
[ 0.  0.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]

Optimized Value Function:
[ 0.          0.          0.17908897  0.32566168  0.44564614  0.54388899
  0.62435269  0.6902767   0.74430941  0.78861593  0.82496629  0.85480748
  0.87932246  0.89947831  0.91606566  0.92973091  0.94100253  0.95031266
  0.95801465  0.96439746  0.96969744  0.97410793  0.97778713  0.98086448
  0.98344598  0.98561839  0.98745282  0.98900753  0.9903303   0.99146035
  0.99242988  0.99326534  0.99398853  0.99461738  0.99516668  0.99564865
  0.99607342  0.99644938  0.99678349  0.99708157  0.99734847  0.99758825
  0.99780435  0.99799965  0.99817659  0.99833727  0.99848346  0.9986167
  0.99873831  0.99884943  0.99895108  0.99904413  0.99912935  0.99920744
  0.99927901  0.9993446   0.99940472  0.99945981  0.99951026  0.99955646
  0.99959875  0.99963741  0.99967275  0.99970503  0.99973447  0.99976132
  0.99978576  0.99980799  0.9998282   0.99984653  0.99986316  0.9998782
  0.9998918   0.99990408  0.99991514  0.9999251   0.99993405  0.99994207
  0.99994926  0.99995568  0.9999614   0.9999665   0.99997103  0.99997505
  0.9999786   0.99998173  0.99998449  0.9999869   0.99998902  0.99999087
  0.99999247  0.99999386  0.99999506  0.99999609  0.99999697  0.99999772
  0.99999835  0.99999889  0.99999933  0.9999997   0.        ]
```




```
def run2(p_h = 0.25):

    policy, v = value_iteration_for_gamblers(p_h)

    plt.bar(range(100), policy)
    plt.title('Converged solutions for p_h = 0.25 (blue) and p_h = 0.55 (orange)')
    plt.xlabel('Sate')
    plt.ylabel('Policy')


run2()
run2(p_h=0.55)
```

```
Converged after 6 iterations using p_h = 0.25.

Converged after 508 iterations using p_h = 0.55.
```