TECHNISCHE UNIVERSITÄT MÜNCHEN
Fakultät für Elektrotechnik und Informationstechnik
Lehrstuhl für Datenverarbeitung
Prof. Dr.-Ing. K. Diepold

Approximate Dynamic Programming & Reinforcement Learning

Assignment 3
November 11, 2019

General Information

Submit your code as `zip` archive via Moodle. Detailed instructions are available at the end of this document.

The deadline is 25. November 2019 at the end of the exercise (16:30 pm)

Write in every file your full name and your student ID. The fulfillment of all formalities will also be part of the points for this assignment.

Optimal Scheduling via Dynamic Programming - (Points: 30)

The scheduling problem is a typical challenge arising in High Performance Computing. One has several workers to perform jobs, and based on current capacities of each worker's queue, one must decide where to put the next job. The catch is that all workers are differently well suited to different jobs. The scheduler has to decide whether to put jobs on empty badly suited workers or increase an almost full queue of well suited worker.

The Environment

Hyperparameter: The environment is described by three basic quantities. The number of workers or queues Q, the size of all queues S and the total amount of job types T that exist. The job types always include a placeholder for "no job to process" and is indicated by a zero. Hence, $T = 2$ means there are jobs of type $\{0, 1, 2\}$.

Develop your code first for a tiny problem (which runs very fast). Later solve the normal problem with the parameters $Q = 3$, $S = 3$ and $T = 2$. Define Q, S and T as global variables in the program to change the complexity on the fly. Do not rely on command line arguments but hard code them.

State space: A state in the scheduling problem can be represented as a pair. The first part is a list of queue states, the second is a new job to process. Queue states themselves are tuples of the form $(1, 2, 0)$. This is a queue of size three with two different jobs at position zero and one. The remaining slot is empty as indicated by the placeholder. Queues are always dense that means there can never be a placeholder in between two jobs. New jobs are added in the leftmost empty slot.

With three queues, a complete (random) state could look like $(((1, 2, 0), (0, 0, 0), (1, 0, 0)), 0)$. The first queue is as described above, the second is empty and the third queue has a single element. In this example the second part, i.e. the job to process, is the placeholder. Nothing to do is also possible, otherwise here can occur jobs of type $\{1, 2, \ldots, T\}$.

For visualization and working with the states you might want to draw them in the console similar to:

```
New Job to process: 0
1: 2 1 -
2: - - -
3: 1 - -
```

Implement what you need to work with such states. This includes for example pushing / removing from lists and converting those into tuples of fixed size as shown above. Also think already about how to get the successor states x' for a current state x and control u. The task to process must be appended to a queue, a new task to process must be create and so on.

Action space: Here this space is rather simple. If the job to process is not a placeholder, the algorithm can choose from one of the queues, but only if a queue is not full.

A valid action is to be idle, i.e. it could be better to wait with the new job until the best suited queue has a free spot. Thus there are up to $Q + 1$ actions. Zero is the placeholder for being idle, $1, 2, \ldots, Q$ is the index of the queue to choose. Create a function $U(x)$ to retrieve a set with all allowed actions in a certain state x. Use this set later to iterate over actions in the DP algorithm.

System Dynamics: Now make the system dynamics $f(x, u, w)$ concise. Assume workers have a certain chance to finish the first job in their queue at a certain time step. The chance depends

further on the type of the job, but not on the time or previous iterations. Thus, f as well as the noise space $\mathcal{W}$ are stationary. Create random probabilities for the assignment.

At each time step, i.e. every time you call $f(x, u, w)$, each worker draws a uniformly distributed random number in $[0, 1]$. If this number is bigger than the chance to complete the task it gets removed from the queue. All other tasks to the right in the queue move one slot forward.

After updating the queues an action can be taken. Depending on the control, the next job to process (if it exists) is added to the end of a queue. A queue that has been full can directly get a new job. If a job has been taken care of a new one is sampled, otherwise it stays the same for the next round. Each type (*no job*, $1, 2, \ldots, T$) has the same chance $\frac{1}{T+1}$ to be picked.

A visualization for an exemplary transition would be the following:

```
New Job to process: 2             New Job to process: 0
1: 2 1 -                          1: 2 1 -
2: - - -              -> u=2 ->   2: 2 - -
3: 1 - -                          3: - - -
```

Queues 1 and 3 run the lottery to see if a job is finished, a possible outcome is that queue 1 is still working while queue 3 removes its single element. The control $u = 2$ means the new job goes in the middle queue ($u = 0$ is being idle, thus the index is shifted by one).

Implement the system dynamics as a function that accepts a state x, a control u and also the noise parameter w. Use w to specify deterministically the outcome of f and provide another method that calculates for a certain x and u all possible values for w and the corresponding probability. Thereby you will make your live really easy when you have to calculate expectations inside the DP algorithm.

Local cost-to-go: The next part to define is the local cost-to-go function $g(x, u, w)$. The cost consists of two additive components. The basis is given by the amount of jobs in the queue, the more the worse. At most $Q \cdot S$ jobs can be in the queue, so this is the largest value. On top comes a cost if a job should be processed but the action is being idle. This is bad and adds a value 5 to the cost.

Terminal cost: At the end of the planning horizon the jobs should be all finished and the queues empty. Thus for $g_N(x_N)$ use just the fill level.

Solution with Dynamic Programming

Now all ingredients for solving the problem with Dynamic Programming are ready. The last thing that is missing is the cost function $V(x)$ for all $x \in \mathcal{X}$ itself. The cost function will be a simple array in your code, thus the tuples forming a state must be properly enumerated to index this array.

Find a method for representing the state space in your program and realize the DP-Algorithm as shown on the last slide of the second lecture.

Solve the problem for a horizon $N = 10$ and store the resulting V_k for all $k = 0, 1, 2, \ldots, N$ and π_k only for $k = 0, 1, 2, \ldots, N - 1$.

As a reference, the problem with parameters from above has 10125 states and 4 different controls, sometimes not all are allowed. In the DP Algorithm one stage takes around 10 seconds on my computer without putting extra effort in optimizing the code. For these hyperparameters memory is not a restriction, but can change them and observe how easily your memory becomes a bottleneck.

Investigating the Policy

For a quick and intuitive verification of the policy at stage $k = 0$ pick a random state with a job to process and almost empty queues (e.g. the left state in the transition example) and look at the action defined by the policy. Compare this action with the chances to complete a task of each worker. You should observe that both match. The opposite can be seen with almost full queues, the policy should often pick a free slot, even if the free queue is not the best for this task. Show the random state you choose, the output of the policy and the probabilities.

To get a feeling for the impact of the horizon compare the policies of different stages. The DP-Algorithm produces at each stage a working policy, one with no stage look ahead, the next with one stage, then with two until the maximum of N.

Take a π_k, select a random start state and let the system run for 250 iteration (i.e. longer than the horizon) and thereby create a trajectory. Calculate all local cost-to-go and sum them up for the 250 states in the trajectory. To compensate the randomness average the costs over at least 15 repetitions.

Do this for all k and plot the costs for all horizons as mean, min and max cost. Describe what you can see.

Final Instructions

Summarize your results in a short report and save it as a PDF file:

- The report contains the answers to the last section "Investigating the Policy".
- Also attach the plot and describe what you can see.

And concerning your code:

- You have to use Python for the implementation, other languages are not accepted.
- Python 3.x on a standard Ubuntu System is used for testing your submissions.
- Provide "plain" python scripts, we will **not** evaluate Jupyter Notebooks.
- You are allowed to use basic modules like Numpy for representing arrays or Matplotlib for plotting, but no existing RL/DP frameworks (e.g. the Reinforcement Learning Toolkit by Richard Sutton).
- The performance will not be graded, but your code must finish within a reasonable time.
- Provide one file called *main.py*, that can be run from the command line without arguments to produce **exactly** the plots you have in your report.
- Add proper titles to identify them.
- You can either show the plots directly on the screen, or you can use `savefig()`. In the second case make sure to use a file format, that can be opened without effort (e.g. a simple `.png`).
- Zip all your python scripts and the report to a single `.zip` archive. As filename use *lastname-firstname-matriculationnumber.zip*. Hand in your archive on Moodle in time.