Train a smartcab to drive

Mention what you see in the agent's behavior. Does it eventually make it to the target location?

We first update the action to just choose a random action from a list of valid actions (which is actually defined in the Environment file): [None, 'forward', 'right', 'left'], so then it just like a random walk and our agent will eventually get to the destination, given unlimited time. We will be using this purely random model to assess the performance of our final model.

Identify a set of states that you think are appropriate for modeling the driving agent. Justify why you picked these set of states, and how they model the agent and its environment.

We have a lot of freedom when choosing the states. We identify four possible choices of states. The state space for these four choices varies greatly but we feel that the way to reduce the state space is to actually infuse some of our knowledge of driving rules into our choice of states, thus making it easier for the agent to learn since we are already teaching in some way by limiting the number of states. For reference we list our input space:

- **Next waypoint:** 'left', 'forward', 'right'.
- **Traffic light:** 'red', 'green'.
- **Left traffic:** None, 'left', 'forward', 'right'.
- **Oncoming traffic:** None, 'left', 'forward', 'right'.
- **Right traffic:** None, 'left', 'forward', 'right'.
- **Deadline:** List of integers from an environment-generated deadline to 0.

Let us examine the four choices:

- **State space 1:** This first state space makes use of all available inputs. This would be our state space of choice if we knew absolutely nothing about our problem at hand, or if we want to sort of set a baseline of how accurate our agent can get without doing any "state engineering". Our state space would now consist of 384*deadline (between 7500 and 15000, depending on the value of deadline) different states. This space is way too large. In order for our Q matrix (|S| X |A|) to converge to an optimal Q* we need to visit each state action pair many times (ideally an infinite number of times). With just a little tweaking we can reduce the size of the state space.
- **State space 2:** For this second state space we note that we don't care about traffic coming from the right, and can eliminate this input as a feature of our states. Note that in order to do this we have had to apply our knowledge of driving rules, so in a way we are already doing some of the teaching for our agent. We also realize that we don't really care about the exact value of our

deadline or time step, but rather we care if we have more or less than a number of steps λ (a parameter we could play with). We now effectively reduced our state space to 96*2 different states, without making major assumptions about driving rules.

- **State space 3:** We can still reduce the size of the state space by noting that not all possible directions oncoming cars can take matter. We see that for left traffic we only care if it is going forward, and for oncoming traffic we only care if it going forward or right without making a difference between the two. We have now slimmed down the possible values of the ‘left’ and ‘oncoming’ input variables to two each: [None|’left’|’right’, ‘forward’] and [None|’left’, ‘right’|’forward’]. We have again used our knowledge of traffic rules to slim down the space, but we aren’t yet encoding all of the traffic rules in the definition of the states. **This is the space state we will use throughout the rest of the problem**. If we choose not to use the deadline parameter into two groups, we end up with 24 different states. Seeing that this number of states is more manageable, we can even consider splitting the timer into two or even three different groups, say [t<=5, 5<t<=10, t>10] for a total of 48 or 72 states.
- **State space 4:** Finally, we can encode all of our knowledge of traffic rules into the definition of our states by noting that for particular for different values of the ‘next_waypoint’ variable, we care about different values of the traffic variables. By writing them out explicitly (ignoring the deadline for now) we see we only need 8 different states:

State \ Variable	next_waypoint	light	left	oncoming
1	L	R	N | L | R | F	N | L | R | F
2	L	G	N | L | R | F	N | L
3	L	G	N | L | R | F	R | F
4	F	R	N | L | R | F	N | L | R | F
5	F	G	N | L | R | F	N | L | R | F
6	R	R	N | L | R	N | L | R | F
7	R	R	F	N | L | R | F
8	R	G	N | L | R | F	N | L | R | F

If we now choose to consider two or three possible values for the counter we end up with 16 or 24 different states. But we think that this is a little bit too much, since what we want is for our agent to learn on its own, and with this definition of states we have pretty much done all the work!

Implement the Q-Learning algorithm by initializing and updating a table/mapping of Q-values at each time step. Run it again, and observe the behavior.
What changes do you notice in the agent's behavior?

We now implement Q learning in the following way: we first sense our state, perform an action and get a reward and store those values (<s,a,r>). Then we move on to the next iteration, where we sense our state again. With our previous values and our new state s', we have all the necessary information (<s,a,r,s'>) to update our Q(s,a) entry in the following way:

$$Q(s,a) := (1-\alpha)Q(s,a) + \alpha(r(s,a) + \gamma\, max_{a'} Q(s',a'))$$

Where α is the learning rate and ɣ our discount factor. Let's quickly explore these two parameters in their extremes.

When α→0 we are simply updating our Q value with its current value: our system is not incorporating the new found knowledge. On the other extreme, when α→1, we are not incorporating any previous knowledge into our update, and we are simply replacing the previous value with our new update. When properly decayed, this value assures the convergence of the Q values table to its optimal value [1].

When ɣ→0 we are essentially looking at the immediate reward for taking a certain action and we do not care of the value of our next state. When ɣ→1, we take into account the values of **all** our next states! We will play with this parameter in the result section below.

In reinforcement learning, we can use the values in our Q table to take the action that maximizes our long term reward. But when we are teaching our agent, how does one balance exploration vs. exploitation? How can we be sure we are choosing the best possible action and not some good action, but not the best, which we happened to choose first? One of the approaches to playing with this tradeoff is called Boltzmann exploration[2]. With this approach we select an action with a probability proportional to its Q value relative to the other Q values for that state:

$$P(a \mid s) = \frac{exp(Q(a,s)/\tau)}{\Sigma_{a'}\, exp(Q(a',s)/\tau)}$$

So for a given state, as we become more and more sure that a particular value is optimal, we will chose that action with a higher probability. When we are not so sure that the action is the best possible (i.e. its Q value is not too different from that of other actions), then we can still choose some other action to keep exploring. The parameter τ, can be understood as a

[1] L.P. Kaelbling et al. Reinforcement Learning - A Survey. Jour. of Art. Int. Res. 4, 1996
[2] Ibidem

temperature (as is common in Boltzmann models). When the temperature is high, exploration is more common since all possible actions are “excited” (i.e with more or less equal probability). However as we cool our model, there is less noise, and we start exploiting our knowledge since now we will pick the action with the highest Q value. We will test these two limits in the results section below.

Note: for our description of results below we set the random seed of all modules in order to replicate results with different agents.

We now run our model with the following parameters:

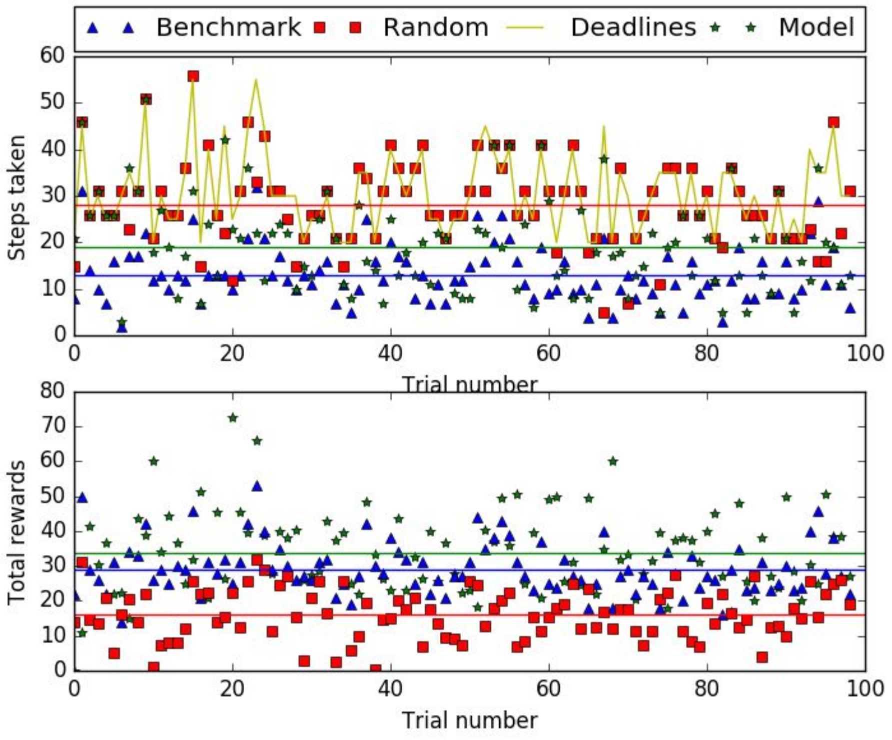
- Boltzmann’s temperature: 0.5
- Discount factor: 0.2
- Learning rate decayed as: $\alpha = (1 - globalCounter/3000)^2$

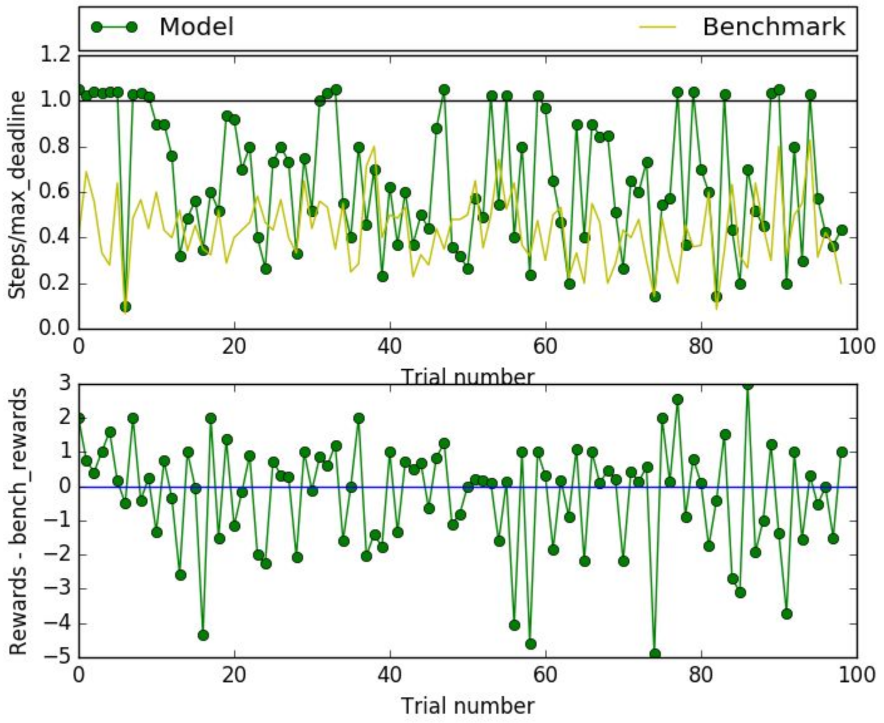
The results are summarized in the graphics below:

- The first graph shows the total steps taken per trial for our model (green stars), our benchmark which always takes the correct action (blue triangles) and our purely random model (red squares) as well as the max_deadline per trial (yellow line). We see that after around 10 trials the model starts reaching the deadline within the allotted time. The overall accuracy (over all trials) is around 78.8% whereas if we look at the last 85 trials we get 85.7%. Note that we find our model sometimes reaches the destination in significantly less steps than the optimal model. This is due to two reasons: first our model may make mistakes and advance quicker even if that incurs in a penalty, and second and most important, the map wraps around so if we are on the top edge and move upwards we end up at the very bottom. However the next_waypoint does not take this into account and will never wrap. Our model may, due to a random choice end up wrapping which may play to its advantage.

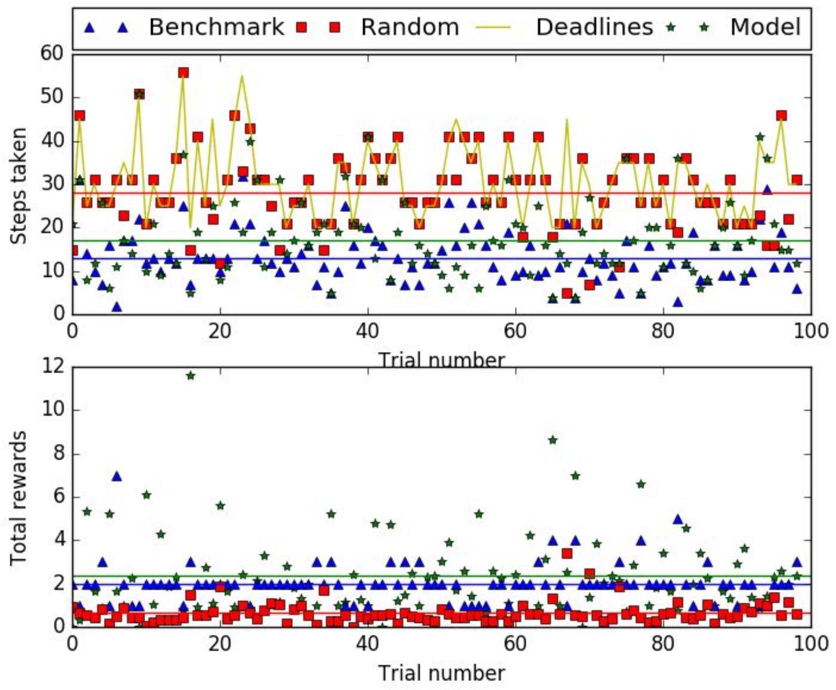
  We can also see the total rewards divided by the total steps taken per trial for the three models described above, and the averages. As we would expect the average random reward is lower than the optimal reward. We note that our model’s average reward is higher but should not be surprising since most of the time our model needs more steps to reach the destination.

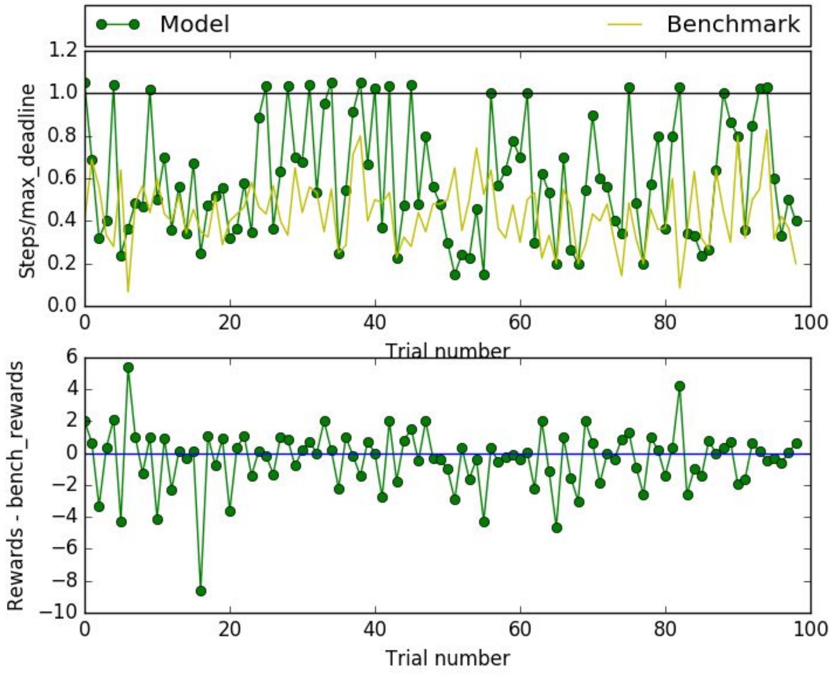
- The second graph shows the number of steps as a fraction of the max deadline and the difference in average reward per trial between our model and the optimum model.





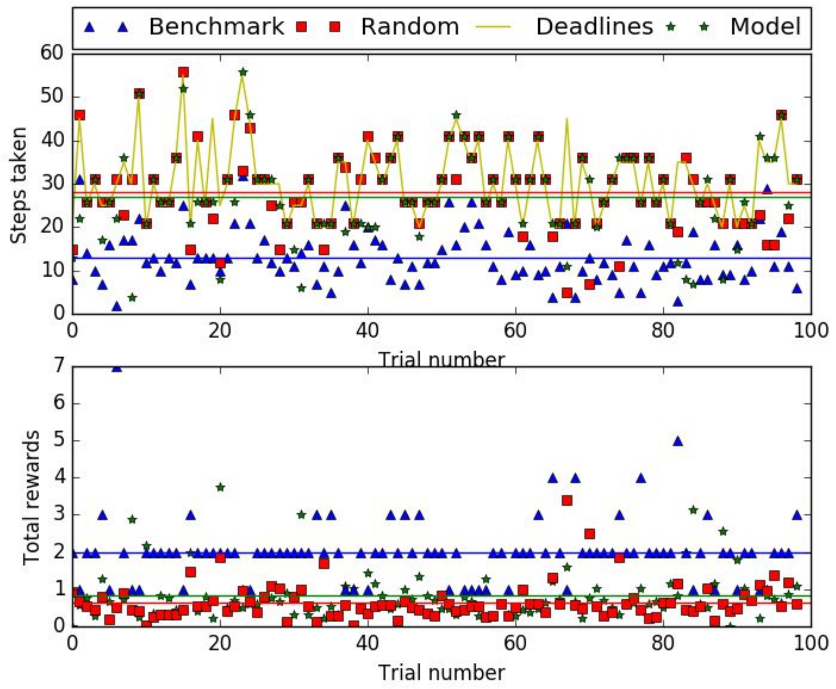
Let's study how different our model performs when the parameter γ is set to 0, that is, when we are taking a purely instantaneous reward approach:

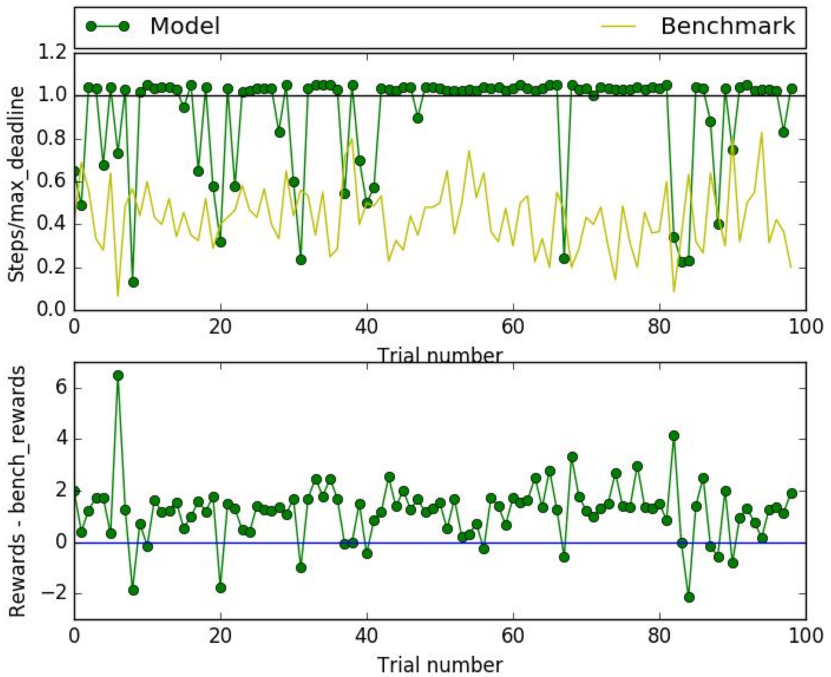




This model has a total accuracy of around 84% and an 85 run accuracy of 85%. So the model learns really fast but then does not improve much over time. But we can see from the bottommost graph that the average reward stays closer to the optimum value than with our previous model where we could observe much more variation.

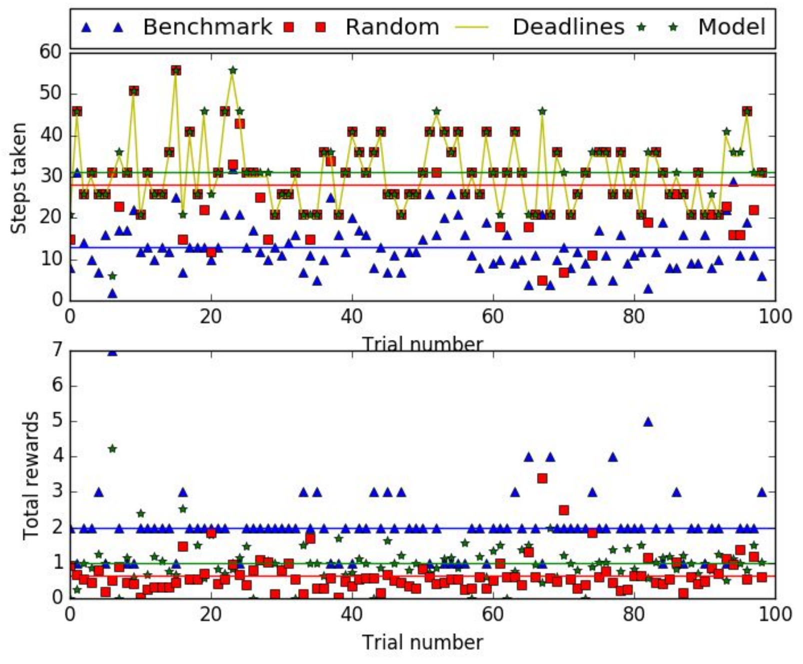
More interestingly, let's try to tune our Boltzmann temperature, since this parameter decides how much we trust our knowledge, or how much we want to explore. Let's first increase the temperature to make our choices more random: we'll set $\tau = 10$.

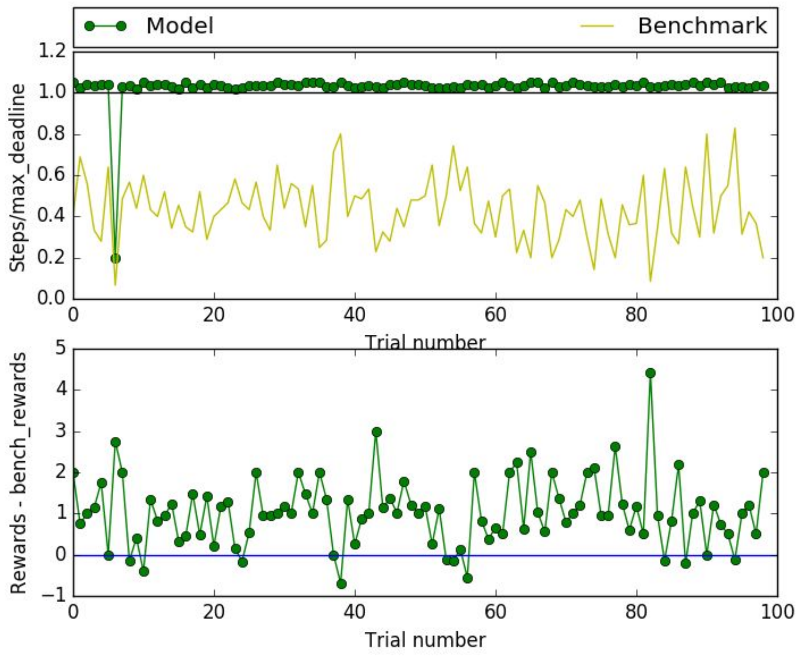




Our accuracies are now around 26% for both cases. We see that our system is too random! We are exploring too much.

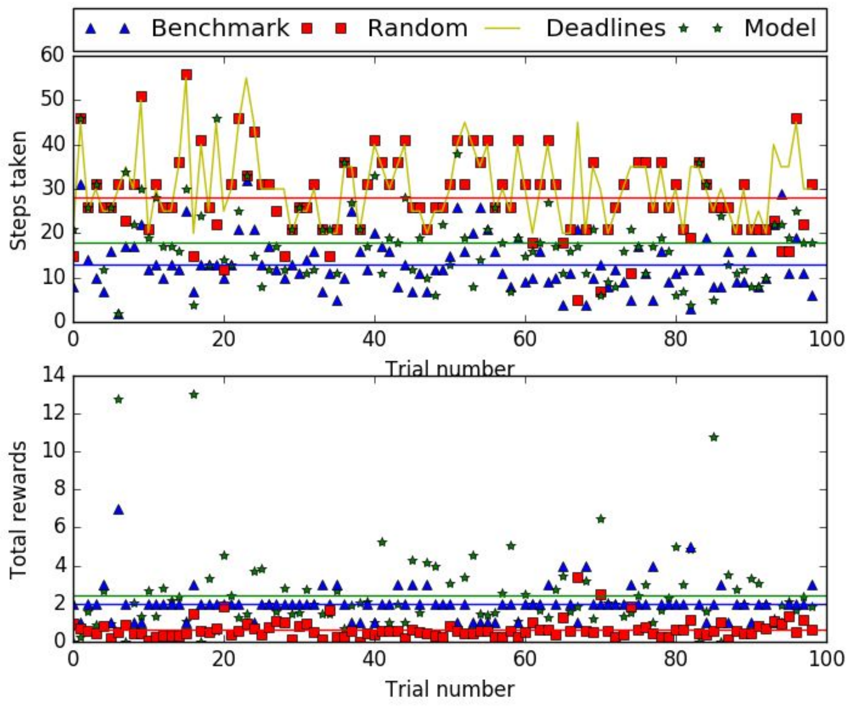
Let’s look at the opposite effect, what would happen if we decrease our value of temperature to $\tau = 0.1$?

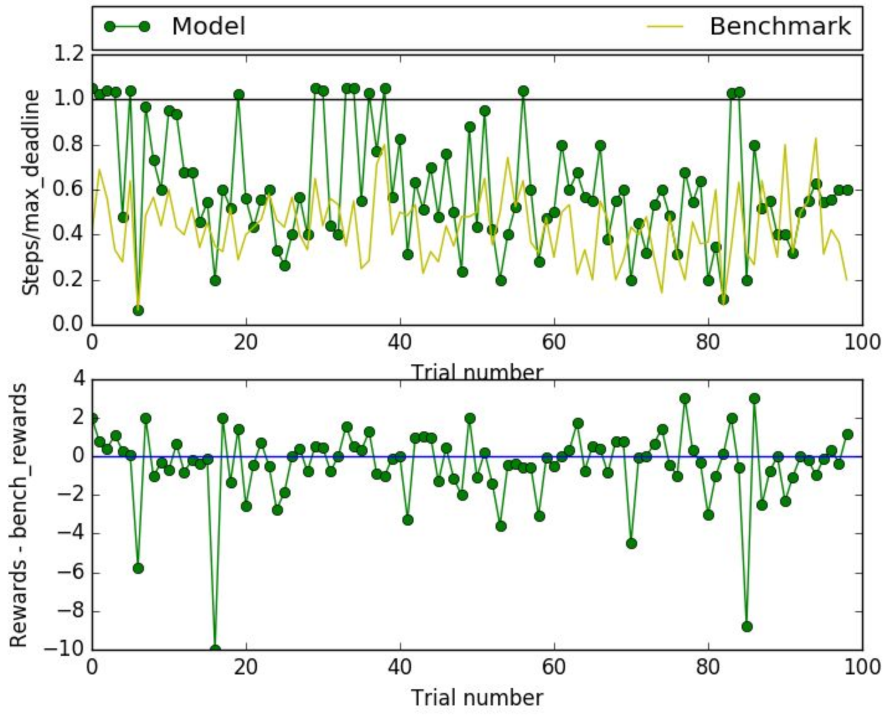




This model isn’t good either! It is even worse than the previous one, with accuracies around 1%. This is because we are blowing up the probabilities with such a low temperature and our model quickly settles at a local maximum (in this particular case the action None).

Let’s try to see if we find a sweet spot for the temperature. This are the results if we lower the temperature just a little bit, to 0.4.

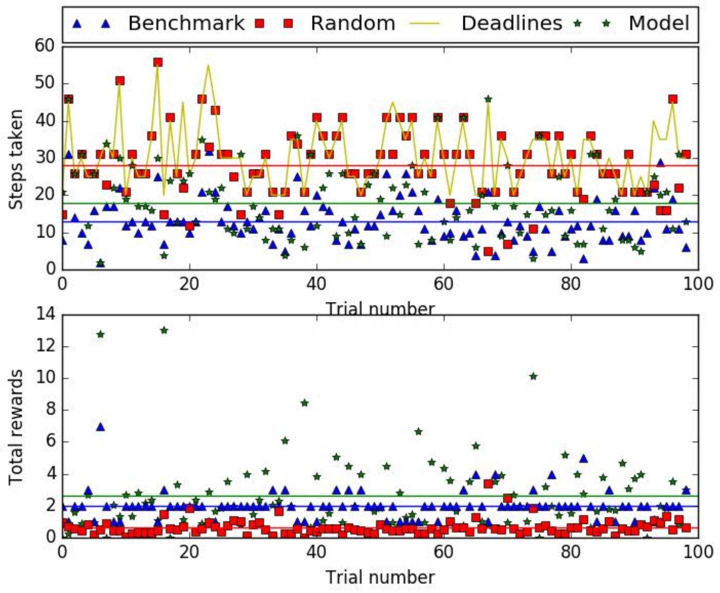


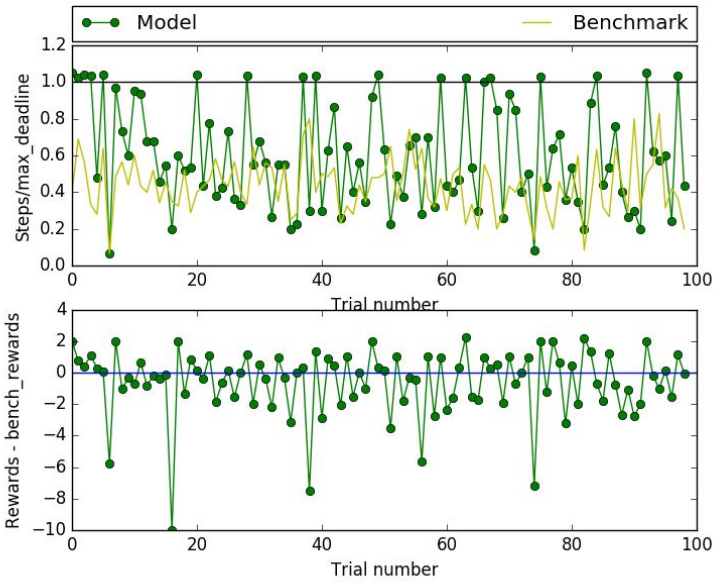


This is our best model so far, with accuracies of 84.9% and 89.1%. The average reward remains pretty close to optimum and in the last 60 trials we only don’t reach the destination 3 times!

Finally, we will study two more things before concluding this report. First we will look at a different alpha decay, and second we will try our model in another set of states, this time with a larger space state.

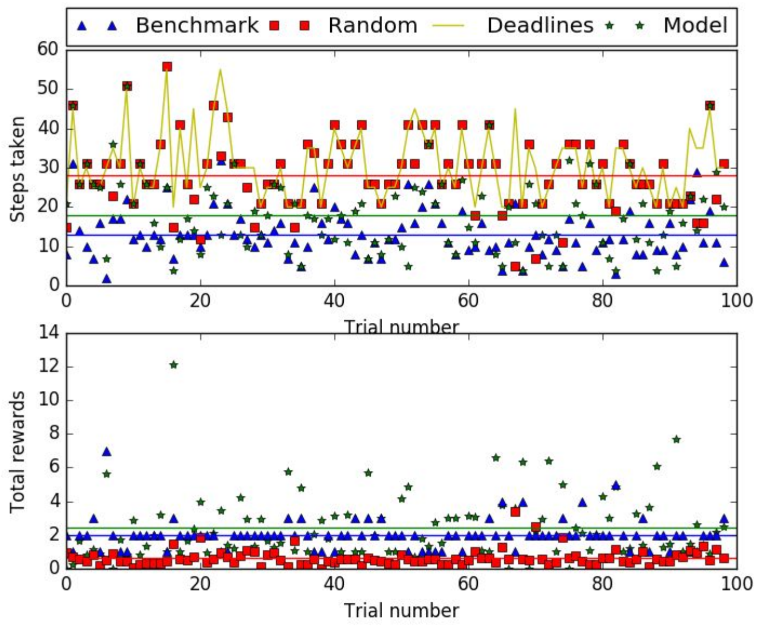
So far we have used a learning rate decay rule of the form: $\alpha = (1 - globalCounter/3000)^2$. We could still play around with this form by changing the power but we want to focus on another type of decay. We use the same model as above but with $\alpha = cos(\pi * globalCounter/2 * 3000)^3$. Here again we could experiment with different powers.

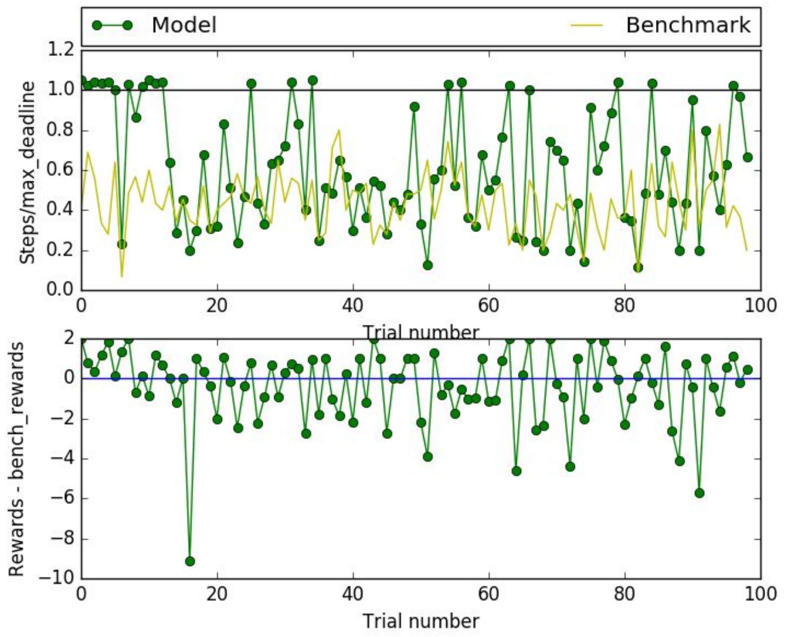




Our accuracies are now 82.8% and 85.7% so even though the effect isn't dramatic we see that varying the learning rate decay has a significant impact, and it is something to take into account when polishing a model.

Finally we will try our best model so far in a different set of states. We will go back to the states in state space 2 at the beginning of the report but ignore the deadline. So now our state is made out of: [next, light, left, oncoming]. Let's see how our model performs in a bigger state space:





Our accuracies are now 80.1% and 89.3%. With a wider state space it takes our model a little longer to learn, as we can see from the larger number of mistakes early on. However, the model finally learns in this state space as well, reaching an accuracy similar to that obtained with the smaller state space.

References:

L.P. Kaelbling et al. Reinforcement Learning - A Survey. Jour. of Art. Int. Res. 4, 1996.
M.L. Littman. Algorithms for sequential decision making. PhD dissertation, Brown Univ.
Udacity videos and forums.
https://studywolf.wordpress.com/2012/11/25/reinforcement-learning-q-learning-and-exploration/
http://mnemstudio.org/path-finding-q-learning-tutorial.htm