Unit 05 - Reinforcement Learning

Lecture 17. Reinforcement Learning 1

17.1. Unit 5 Overview

This unit covers reinforcement learning, a situation somehow similar to supervised learning, but where instead of receiving feedback for each action/decision/output, the algorithm get a overall feedback only at the very end. For example, in “playing” a game like Chess or Go, we don’t give to the machine a reward for every action, but what counts whether the whole game has been successful or not.

So we will start approaching the problem of reinforcement learning by looking at a more simplified scenario, which is called **Markov Decision Processes (MDPs)**, where the machine needs to identify the best action plan when it knows all possible rewards and transition between states (more on these terms soon). Then, we are going to lift some of the assumptions and look at the case of full reinforcement learning, when we experience different transitions without knowing their implicit reward, and collect them only at the end, which is a typical case in life.

In the project at the end of this unit we will implement an agent that can play text-based games.

Summary:

- Lesson 17: How to provide the optimal policy for a Markov Decision Process (using Bellman equations) when you know all the state probabilities/rewards
- Lesson 18: How to do the same while you don’t know them and you need to learn them little by little with experience. Very interesting concept of exploration vs exploitation
- Lesson 19: Qualitative lesson (but with some interesting concepts) on the ML research in Natural Language Processing

17.2. Learning to Control: Introduction to Reinforcement Learning

This lesson introduces reinforcement learning. The kind of tasks we can solve are more extended than supervised learning, as the algorithm doesn’t need te be given an info of the payoff at each single step. And in many real-world problems we cannot get supervision to the algorithm at every point of the progression.

In real world, we go, we try different things, and eventually some of them succeed and some of them don’t. And as intelligent human being, we actually know to trace back an attribute, what made our whole exploration successful.

For example, a mouse learning to escape a maize and get some food, learns how to exit the maize even if he doesn't get a reward at each crossing, but only at the end of exiting the maize.

In many occasions we are faced with the same scenarios: computer or board games, robots learning to arrive to a certain point (Andrew Ng PhD dissertation was about teaching a robot helicopter to fly), but also the business decisions, e.g. call, email, sending gift to lead a client to sign a contract (note that each of this option may have, taken individually, a negative reward, as they imply costs).

Note that in some cases you may also get some rewards in the middle, but still what must count, is the final situation, if the goal has been reached or not.

So these are the type of problems that are related to reinforcement learning, where our agent can take a lot of different actions, and what counts is the reward this agent gets or doesn't get at the end. And the whole question of reinforcement learning is how can we get this ultimate reward that we will get and propagate it back and learned how to behave.

Defined the objectives and the type of problems that can be solved with reinforcement learning, the next point of this lecture will discuss how to formalize these problems, setting the problem in clear mathematical terms.

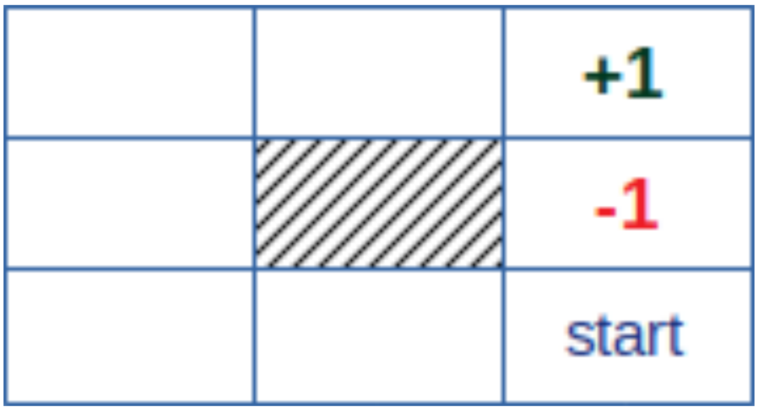
We will first introduce Markov Decision Processes, introducing concepts as state, reward, actions, and so on.

We will then discuss the so called **Bellman equations** that allow us to propagate the goodness of the state from the reward state to all the other state, including the initial one. Based on this discussion we would be able to formulate value iteration algorithms that can solve the Markov Decision Processes.

Across the remaining of the exposition in this lecture we will consider the following, very simple, example.

In the following table of 3 X 3 cells, a “robot” has to learn to reach the top-right corner, getting then a $+1$ reward, while if it ends in the lower cell it gets a negative reward -1. Also there will be some contraints, like being forbidden to go to the central cell.

Our goal is to find a strategy to send the robot on that particular top-right corner



17.3. RL Terminology

The terminology for today:

- **states** $s \in S$: in this lecture they are assumed to be all observed, i.e. the robot know in which state it is (the "observed case")
- **actions** $a \in A$: What we can actually do in this space, in this example going one cell up/down/right or left (and if we hit the border we remain in the same state)
- **transition** $T(s, a, s') = p(s'|s, a)$: The probability to end up to state s' conditional to being in state s and performing action a. For example, starting in the bottom-left cell and performing action "going up" you may have 80% probability of actually going up, 10% of going on the adiacent right cell, and 10% of remaining in the same cell. Note that $\sum_{s' \in S} T(s, a, s') = 1$
- **reward** $R(s, a, s')$: The reward for ending up in state s' conditional to being in state s and performing action a. In the example here the reward can be defined as only a function of s (i.e. the *leaving* cell), but more generically (as in the example of marketing where each action involves a cost) it is a function also of a and s'.

This problem in a deterministic set up would just require planning, nothing particular.

But here we frame the problem in a stochastic way, that is, there is a probability that even if the chosen action was to move to the top cell, the robot ends up instead to an other cell. This is why we introduce transitions. Note: this is very similar to the Markov Chains studied in the Probability course (6.431x Probability–The Science of Uncertainty and Data, Unit 10). There, there is a simple transition matrix. Here the transition matrix depends from the action employed by the agent.

Indeed: *"A Markov decision process (MDP) is a discrete time stochastic control process. It provides a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker. Markov decision processes are an extension of Markov chains; the difference is the addition of actions (allowing choice) and rewards (giving motivation). Conversely, if only one action exists for each state (e.g. "wait") and all rewards are the same (e.g. "zero"), a Markov decision process reduces to a Markov chain."*

In MDP we assume that the sets of states, actions, transitions, and rewards are given to us: $MDP = \ < S, A, T, R >$. MDPs satisfy the Markov property in that the transition probabilities, the rewards and the optimal policies depend only on the current state and action, and remain unchanged regardless of the history (i.e. past states and actions) that leads to the current state. In other words, The state we may end up after taking an action and the reward we are getting only depends on the current state, which encapsulates all the relevant history.

We can consider several variants of this problem, like assuming a particular initial state or a state whose transitions are all zero except those leading to the state itself, e.g. once arrived on the rewarding cell on the top-right corner you can't move any more, you are done.

When we will start thinking about more realistic problem, we will want to have a more complicated definition of transition, function, and the rewards. But for now, for what we're discussing today, we can just imagine them as tables, i.e. constant across the steps.

Reward vs cost difference

What is the difference between *reward* and *cost* ?

The reward generally refers to the value an agent might receive for being in a particular state. This value is used to positively "reinforce" the set of actions taken by the agent to get itself into that state.

The cost generally refers to the value an agent might have to "pay", "expend", or "lose" in order to take an action.

Think about there are things you want to do (e.g. pass this course): doing this would result in some reward (e.g. getting a degree or qualification). But doing this isn't free: you have to spend time studying, etc. which can be seen as "costly". In other words, you're going to have to spend something (in this case, your time) in order to reach that rewarding state (in this case, passing this course).

In the same way, we can model RL settings where an agent is told that a particular state has a reward (completing a maze gives the agent +100) but each action it takes has a cost (walking forward, backward, left, or right gives the agent -1). In this way, the agent will be biased towards finding the most efficient/cheapest way to complete the maze.

17.4. Utility Function

The main problem for MDPs is to optimize the agent's behavior. To do so, we first need to specify the criterion that we are trying to maximize in terms of accumulated rewards.

We will define a utility function and maximize its expectation. Note that this should be a finite number, in order to compare different possible strategies, to find which is the best one.

We consider two different types of "bounded" utility functions:

- **Finite horizon based utility** : The utility function is the sum of rewards after acting for a fixed number n steps. For example, in the case when the rewards depend only on the states, the utility function is $U[s_0, s_1, \dots, s_n] = \sum_{i=0}^{n} R(s_i)$ for some fixed number of steps n. In particular $U[s_0, s_1, \dots, s_{n+m}] = U[s_0, s_1, \dots, s_n]$ for any positive integer m

- **(Infinite horizon) discounted reward based utility** : In this setting, the reward one step into the future is discounted by a factor γ, the reward two steps ahead by γ^2, and so on. The goal is to continue acting (without an end) while maximizing the expected discounted reward. The discounting allows us to focus on near term rewards, and control this focus by changing γ. For example, if the rewards depend only on the states, the utility function is $U[s_0, s_1, \ldots] = \sum_{k=0}^{\infty} \gamma^k R(s_k)$.

While tempting for its simplicity, the finite horizon utility is not applicable to our context, as it would drop time-invariance property of the Markov Decision Model in terms of the optimal behaviour. Indeed, as there is a finite step (time), the behaviour of the agent would become non-stationary: it would not only depends from which state it is located, but also on which step we are, on how far we would be from such ending horizon. So if you arrive to a certain state, and you just have one step to go, you may decide to take a very different step versus if you have still many steps to go and you can do a lot of different things. So for instance, if you just go one step to go, you may go to extremely risky behaviour because you have no other chances.

We will then employ the discounted reward utility, where γ is the **discount factor**, a number between 0 and 1 that measures our “impatience for the future”, how greedy we are to get the reward now instead of tomorrow. In economy it is often given as $\frac{1}{1+r}$ in discrete time applications (as here) or $\frac{1}{e^r}$ in continuous time ones, where r is the “discount rate” and we would then say that the utility is the net present value of this infinite serie of future rewards (a bit like the value of some land is equal to the discounted annuity you can get from that land, i.e. renting it or using it directly.)

But why the discounted reward is bounded?

The discounted utility is $U = \sum_{k=0}^{\infty} \gamma^k R(s_k)$ but we can write the inequality $U = \sum_{k=0}^{\infty} \gamma^k R(s_k) \leq \sum_{k=0}^{\infty} \gamma^k R_{max}$ where R_{max} is the maximal reward obtenible in any state, and then taking it outside the sum and using the geometric series poperties that $\sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma}$ for $|\gamma| \leq 1$, we can write that $U \leq \frac{R_{max}}{1-\gamma}$

Going back to the land example, even if you can use or rent some land forever, its value is a finite one, indeed because of the discounting of future incomes.

17.5. Policy and Value Functions

We now define a last term, “policy”.

Given an MDP, and a utility function $U[s_0, s_1, \ldots, s_n]$, a **policy** is a function $\pi : S \to A$ that assigns an action π to any state s. We denote the optimal policy π_s^* as the optimal action you can take in a given state, in term of maximising the expected utility $\pi_s^* : argmax_{a_s} E[U(a_s)]$.

Note that the goal of the optimal policy function is to maximize the expected discounted reward, even if this means taking actions that would lead to lower immediate next-step rewards from few states.

The policy depends from the structure of the rewards in the problem. For example, in our original robot example, let’s assume that for each action we have to pay a small price, like 0.01. Then the “policy” for our robot would likely be (it then depends from the transitions probabilities) to go round the table to arrive to the $+1$ reward without passing for the -1 one. But if instead the reward structure is such that we “pay” each action 10, the policy would likely be to go instead direct to the top-right cell.

Again, we want this policy to be independent on any previous actions or the time step we are, just a function of the state where we are.

When we are talking about solving MDPs, we’re talking about finding these optimal policies.

This problem of solving the MDP is very similar to the reinforcement learning problem, but with the sole difference that when you have an reinforcement learning problem, you’re not provided with a transition function and the reward function until you actually go in the world, experience, and collect it. But for today, we assume these are given.

We will now introduce the Bellman equation. We use it so somehow propagate our rewards.

In our robot example, considering only the $+1$ and -1 rewards as given, we need a tool to formalise the intuition that the cell adjacent to the -1, while not having any reward by itself, it is a “great place to be”, as it is only one step away from the -1 reward, the top-left cell a bit less great place, and the bottom-left cell an even less great place.

In other words, we need to provide our agent with some quantification is, how good is the state, which is its value, even if the reward is coming many steps ahead. So we need to introduce some notion of this value of each state, and what the Bellman equations do, they actually connect this notion of this value of the state and the notion of policy.

The **value function V(s)** of a given state s is defined as the expected reward (i.e. the expectation of the utility function) if the agent acts optimally starting at state s.

Setting the timing in discrete time modelling

One very important aspect of discrete time modelling with discounting is to make clear “when the time starts” and if rewards/costs are beared at the beginning or at the end of the period.

In our case the time start clocking when we leave the departing state. At that time we pay any cost incurred in the action and cash the reward linked to state 1. After one period (so discounted with γ) we move to the second state paying at that moment the action cost and the reward linked to that state and so on.

Mathematically: $U = R(a_{0\rightarrow 1}, s_1) + \gamma * R(a_{1\rightarrow 2}, s_2) + \gamma^2 * R(a_{2\rightarrow 3}, s_3) + \ldots$

17.6. Bellman Equations

Let’s introduce two new definitions:

- the **value function** $V^*(s)$ is the expected reward from starting at state s and

acting optimally, i.e. following the optimal policy
- the **Q-function** $Q^*(s,a)$ is the expected reward from starting at state s, then acting with action a (not necessarily the optimal action), and acting optimally afterwards.

The first of the two Bellman equations relate the value function to the Q function in terms that the value function is the Q-function when the optimal a is followed:

$V^*(s) = \max_a Q^*(s,a) = Q^*(s, a = \pi^*(s))$ where π^* is the optimal policy.

The second Bellman equation relates recursively the Q-function to the reward associated to the action cost and the destination reward (girst term) plus the value function of the destination state (second term):

$$Q^*(s,a) = R(s,a,s') + V^*(s')$$

When we consider the discounting and the transition probabilities it becomes
$Q^*(s,a) = \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V^*(s'))$

Expressed in these terms the value function becomes:

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V^*(s'))$$

This equation encodes that the best value I can take from this state s to the end can be divided into two parts: the rewards that I am getting from taking just one step, this is the reward, plus whatever reward I can take from the following step multiplied by the discounting factor.

17.7. Value Iteration

Given the Bellman equations would be tempting to try to compute the value of a state starting from the values of the reachable states from the optimal action (that, in turn, can be just picked up looking at all the possible actions).

The problem however is that the V(A), the value of state A, then may depends from V(B), but then V(B) may depends as well from V(A) and many more complicated interrelations. Aside very simple cases, this make impossible to find an analytical solution or even a direct numerical one.

The trick is then to initialise the values of the states to some values (we use 0 in this lecture) and then starting iteratively to loop to define the value of a state at iteration (k) from the value of the reachable states *that we did memorised in the previous iteration* from the optimal action :

$$V^*(s_k) = \max_a \sum_{s\prime} T(s,a,s')(R(s,a,s') + \gamma V^*(s'_{k-1}))$$

The algorithm than stop when the differences from the values at k iteration and those at $k+1$ become small enough on each state.

Note that $V^*(s_k)$ can be interpreted also as the expected reward from state s acting optimally for k steps.

Once we know the final values of each state we can loop over each state and each possible action on each state one last time to select the “optimal” action for each state and hence define the optimal policy:

$$\pi^*(s) = argmax_a Q^*(s,a) = argmax_a \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V^*(s'))$$

And pretty much what this algorithm is doing, it is propagating information from these reward states to the rest of the board.

Convergence

While the iterative algorithm described above converges, we are now proving it only for the simplest case of a single state and single action.

In such case the algorithm rule at each iteration would reduce to:

$$V^*(s_k) = (R(s) + \gamma V^*(s_{k-1}))$$

In a fully converged situation the Bellman equation gives us that:

$$V^*(s) = (R(s) + \gamma V^*(s))$$

Putting them together we can write:

$$(V^*(s) - V^*(s_k)) = \gamma * (V^*(s) - V^*(s_{k-1})).$$

That is, at each new iteration of the algorithm, the difference between the converged value and the value of the state at that iteration get multiplied by a factor γ, that being a number between 0 and 1, it means this difference reduces at each iteration, implying convergence.

17.8. Q-value Iteration

Instead of computing first the *values* of the states and then the Q function to retrieve the policy, we can directly, using exactly the same approach, compute the Q function. the algorithm then is named **Q-value iteration** and is the one we will use in the context of reinforcement learning in the next lecture.

Using the Bellman equations we can write the Q function as:

$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left(R(s,a,s') + \gamma \max_{a'} Q^*(s',a') \right)$$

The iteration update rule than becomes:

$$Q^*_{k+1}(s,a) = \sum_{s'} T(s,a,s') \left(R(s,a,s') + \gamma \max\nolimits_{a'} Q^*_k(s',a') \right)$$

Lecture 18. Reinforcement Learning 2

18.1. Revisiting MDP Fundamentals

We keep the MDP setup as in the previous model, and in particular we will work in this lecture with the Bellman equation in terms of the Q-value function:

$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left(R(s,a,s') + \gamma \max_{a'} Q^*(s',a') \right)$$

We will however frame the problem as a reinforcement learning problem where we need to find the best policy knowing a priori only the sets of states S and possible actions A but not those of the transition tables T and the rewards R. These will be instead estimated as posteriors when we start "acting in the world" (collecting data), seing where do we actually end up from each (state, action) and the rewards we may ended up with.

In this setup, you see that now actually acting in the world becomes an integral part of our algorithm. We can try as much as we wish, but eventually we need to provide the policy. So how are we doing these trials? Which and what to try will be an integral part of the algorithm.

18.2. Estimating Inputs for RL algorithm

So, out of the tuple $< S, A, T, R >$ that constitutes a Markov Decision Process, in most real world situation we may assume to know the sets of the states S and the possible actions A, but we may not know the transition possibilities T and the rewards R.

We may be tempted to start by letting our agents running with random decisions, observe where they land after performing rule a on state s, and which reward they collect, and then estimate T and R as:

$$\hat{T}(s,a,s') = \frac{\text{count}(s,a,s')}{\sum_{s''} \text{count}(s,a,s'')}$$

$\hat{R}(s,a,s') = \frac{\sum_{t=1}^{\text{count}(s,a,s')} R_t(s,a,s')}{\text{count}(s,a,s')}$ (the average of the rewards we collect)

The problem with this approach is however that certain states might not be visited at all while collecting the statistics for $\hat{T}, \hat{R}$, or might be visited much less often than others leading to very noisy estimates of them.

We will then take an other approach where we will not only passively collect the statistics during the training, but as we start "learning" the parameters we will also start guide our agents toward the action we need more to collect more useful data.

For this instance it is important to distinguish between model based approaches and model free learning approaches.

Instead of defining them, let's take an example.

Let’s assume a discrete random variable X and our objective being to estimate the expected value of a function of it $f(X)$.

Model based approaches would try to estimate first the PMF of X $p_X(x) and then estimate the function$ $f(X)$ using the *Expected value rule for function of Random variables*:

$\hat{p}_X(x) = count(x)/K$ (where K is the number of samples)

$E[f(X)] \approx \sum p(\hat{x}) f(x)$

Model free approaches would instead sample K points from $P(X)$ and directly estimate the expectation of $f(X)$ as:

$$E[f(X)] \approx \frac{\sum_{i=1}^{K} f(X_i)}{K}$$

The “model free” approach is also known with “sample based” approach or “reduced” approach.

18.3. Q value iteration by sampling

We are left hence with two problems. The first one is to actually find how to incorporate in a live fashion our experience to our estimate, updating our estimation of the missing parameters of the model one record after the other, while observing the agent performing, doing an action in a certain state and ending up in a certain state eventually collecting a certain reward.

The second one is actually how to direct the agent while collecting the data, still with limited knowledge of the model.

We will tackle the first problem in this segment, assuming we know how to direct the agent, and we will come to the full picture in the next segment.

Exponential running average

But before we start the segment, let’s quickly introduce a new notion which is called **exponential running average**, which is a way to do averaging, but which is different from the standard averages that we are used to, because it weighs differently the significance of sample. It gives more weight to the current and most recent samples compared to the prior samples that we’ve collected.

If a normal average is $\bar{x}_n = \frac{\sum_{i=1}^n x_i}{n}$, the exponential running average is defined as $\bar{x}_n = \frac{\sum_{i=1}^n (1-\alpha)^{(n-i)} x_i}{\sum_{i=1}^n (1-\alpha)^{(n-i)}}$ with α being a value between 0 and 1.

The exponential running average is normally used in its recursive nature as $\bar{x}_n = (1-\alpha)\ \bar{x}_{n-1} + \alpha\ x_n$.

This formulation of the exponential average gives us a mechanism to take our prior estimate of the running average and add to it the new sample that we are taking.

Sample-based approach for Q-learning

We are now ready to live-updating the estimation for the Q-value algorithm.

As we don’t know the transitions and rewards ahead of time, the Q-value formula will now be based on the samples. We take here a sample-based approach and directly think to what happens to Q when I start taking samples.

Let’s recall the formula from Ballman equations that relate a given Q value to those of the neighbouring (reachable) states:

$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left(R(s,a,s') + \gamma \max_{a'} Q^*(s',a') \right)$$

What so I obtain when I observe my first $sample_1$ of an action done in state s that lead me to state s'_1 ?

The Q-value(s,a) becomes $R(s,a,s'_1) + \gamma \max_{a'} Q^*(s'_1,a')$

More in general, when I observe a k-sample of the same (s,a) tuple (action a on state s) I would observe Q-value$(s,a) = R(s,a,s'_k) + \gamma \max_{a'} Q^*(s'_k,a')$

Of course at this point we don’t yet know what $\max_{a'} Q^*(s'_k,a')$. When we will run the iteration algorithm, it will be the value of the previous iteration.

Now, to estimate my $Q(s,a)$ I can just average over these k samples.

Using normal average it would result in :

$$Q^*(s,a) = \frac{1}{k} \sum_{i=1}^{k} Q_i^*(s,a) = \frac{1}{k} \sum_{i=1} \left(R(s,a,s'_i) + \gamma \max_{a'} Q^*(s'_i,a') \right)$$

Estimating directly the $Q(s,a)$ we don’t need to estimate firstly the T or R matrices.

The problem remains however that after we observe $Q(s,a)$ leading to $s\prime$, we may not be able to go bask to state s, it’s not like rolling a dice I can try 1000 times the same experiment.

So we don’t want to have to wait until we can sample it all and then average. What we want to do it to updates of our Q-value each time when we get new sample, to do it incrementally.

And here we can use the exponential average to compute differently our estimates for Q-values, in an incremental fashion.

The Q-value for (s,a) at sample $i+1$ would then be equal to

$$Q_{i+1}(s,a) = (1-\alpha)Q_i(s,a) + \alpha * sample_i(s,a)$$

$$Q_{i+1}(s,a) = Q_i(s,a) - \alpha(Q_i(s,a) - sample_i(s,a))$$
$$Q_{i+1}(s,a) = Q_i(s,a) - \alpha(Q_i(s,a) - (R(s,a,s_i') + \gamma \max_{a'} Q^*(s_i', a')))$$

where s_i is the state I ended up in sample i.

Attention that here $Q_i(s,a)$ is not the Q-value of the sample i, but the average Q-value up to the sample i including. It already embed all the experience up to sample i.

The above equation should recall those of the stochastic gradient descent, where we have our prior estimate, plus alpha, and the update factor. It's just a different form of this update.

The algorithm for Sample-based approach for Q-learning would then be something like

- Initialise all $Q(s,a)$, e.g. to zero
- Iterate until convergence (e.g. until small difference in values between consecutive steps)
  - Collect a sample at each step --> s_i, R(s,a,s_i\prime),
  - Update the Q-value based on the sample as in the above equation.

We can demonstrate that the same conditions on alpha that we used to demonstrate convergence with the gradient descent algorithm are exactly the same that can be shown that apply here. So under this condition, this algorithm is guaranteed to converge (locally).

18.4. Exploration vs Exploitation

Now that we learned how to incormorate our Q-value of the single samples in our $Q(s,a)$ matrix, we are left with the only problem of how to direct our agents during learning, i.e. which policy to adapt.

We are here introducing a very common concept in reinforcement learning (RL), those of Exploration versus Exploitation.

- **Exploration** is when we try to behind the "usual way", when we want to discover new things that may be better than what we already know. In real life is trying a new food, take a new road, experiment with a new programming language. In RL this means trying (e.g. randomly) actions that we don't yet have data for or where our data is limited (like having no institutional memory). As in real life, exploration in RL makes more sense when we don't have yet much experience, to indeed help build it.
- **Exploitation** instead is acting rationally based on the acquired information, try to make the best action based on the available information. It is being very conservative, follow always the same routine. In RL it is simply acting by following the policy.

Epsilon-greedy

To allow in RL more exploitation at the beginning, and more exploitation in successive steps, our model can include a parameter ϵ that gives the probability of tacking an action randomly (vs those maximising the estimated q-value), and have it parametrised such that ϵ is highest at the beginning and reduces as the experiment continues, until becoming negligible when the q-value start to converge, i.e. at that time we follow mostly the policy.

Lecture 19: Applications: Natural Language Processing

19.1. Objectives

- Understand key problems/research areas in Natural Language Processing field
- Understand challenges in various NLP tasks
- Understand differences between symbolic and statistical Approaches to NLP
- Understand neural network based learnt feature representations for text encoding

19.2. Natural Language Processing (NLP)

This lesson deals with Natural Language Processing (NLP), one important application of machine learning technology.

Natural language Processing is all about processing natural languages and it frequently involves speech recognition, natural language understanding, natural language translation, natural language generation etc.

It can be doing something very complicated or something relatively simple, as string matching. An application doesn't need to imply *understanding* of the meaning of the text to be considered doing natural language processing. For example the spam filter in gmail is a "simple" classifier, it doesn't need to understand the text of the email to judge if it is spam.

History

The importance that language processing undergoes in the Artificial Intellingence field is well described by being the core of the famous *imitation game* of the **Turing test** to distinguish (and defining) an intelligent machine from a human:

"I believe that in about fifty years' time it will be possible to programme computers, with a storage capacity of about 10^9, to make them play the imitation game so well that an average interrogator will not have more than 70 percent chance of making the right identification after five minutes of questioning." (Alan Turing, 1950)

In 1966, Joseph Weizenbaum at MIT created a program which appeared to pass the Turing test. The program, known as **ELIZA**, worked by examining a user's typed comments for keywords. If a keyword is found, a rule that transforms the user's comments is applied, and the resulting sentence is returned. If a keyword is not found, ELIZA responds either with a generic riposte or by repeating one of the earlier comments.

Status

The "success" of Natural Language Processing (NLP) technology largely depends on the specific task, with some tasks that are now relatively successful, others that are doing very fast progress, and others that even now remain very complex and hard to be realised my a machine.

Mostly solved tasks

- Spam detection
- Named Entity recognition (NER): extract entities from a text, like location or a person's name of a news
- Part of speech (POS) tagging: recognise in a sentence which word is a verb, which a noun, which an adjective…

Making good progress tasks

They were considered very hard, almost impossible to solve tasks, until a few decades ago.

- Machine translation: while far from perfect, many people now find it useful enough to use it
- Information extraction (IE): translate a free text into a structured representation, like a calendar event from an email or filling structured assessment of a free-text feedback on a medicine
- Sentiment analysis: distinguish, again from a free text, the sentiment, or the rating, over some subject
- Parsing: understanding the syntactic structure of the sentence.

Still very hard tasks

- (Uncostrained) question answering: except questions for which an almost string-matching approach could solve it (*factoid questions*, like "when Donald Trump has been elected", it is easy find a sentence indicating "when Donald Trump has been elected") , questions that really require analysis and comparison are very hard to solve
- Summarisation: extracting the main points out of a larger context
- Dialog: when we go outside of a specific domain, developing a human/macine dialogue is extremely difficult

19.3. NLP - Parsing

The above classification in three buckets of "readiness" of AI technologies for different tasks is however questionable for three reasons.

- *Measure problem*. First the "performance" depends on the measure we are looking. For example, in an experiment of text parsing, each single relation between words ("arcs") has been found correctly by the machine 93.2% of times. Great. However, if we look at the times the whole sentence had all the relations correct (i.e. all arcs correct), the machine accuracy was below 30%. The beauty is in the eye of the beholder, it depends how you calculate your performance.

- *Data problem*. The second reason is that often it is not the algorithm to perform bad, but the availability of data for the area of the problem that is poor. For example machine translations are highly improving year after year and doing quite a good job, but still a food recipe would be badly translated. This because machine translations are trained mostly on news corpus, and very little on "food recipes" subjects. So it's not that the task is hard, it is hard in the context of training data available to us. The difficulty of the task should always be assessed in the context of the training data available.

- *Decomposing sources of "difficulty"*. Third, when we are assessing the difficulty of the questions, we're thinking about ourselves. How difficult is for us as humans, for example how much time it will take you to discover the answer in a question answering task. But since the machine operates and can very easily perform big searches, powerful word matches, an answer for which somewhere, even remotely, there is a almost exact matching in the data, is an easy task for the machine and could be a difficult one for a human. Conversely, if the task involve connecting parts together, "reasoning", because the information that it needed to connect the dots is not readily available, like in text comprehension (you are given a paragraph, a question about it and a set of plausible answers to choose from) or summation/making highlights, there machines are doing very poorly. But, in connection with the previous point, if we will provide machine with a million news stories of the sort, together with their summaries/highlights (many news agencies, in addition to their stories, provided story highlights, where they extract some sentences or close paraphrases of the sentences as a highlight from the story), the pattern will be there and the machine will find it. So this whole assessment of asking machines of doing processing and compare it with humans. They just have different capacities and capabilities. And we need always to analyse the performances in context.

One area NPL is doing very useful things is in classification, where helps categorise free text from scientific medical articles to help summarise them and building evidence-based medical practices, find the distribution of different opinion on medical issues from the analysis of the individual article and associated medical experiments.

19.4. Why is NLP so hard

In short: due to different kind of ambiguity intrinsic in natural languages.

NLP in the industry (already operational)

- Search
- Information extraction
- Machine translation
- Text generation
- Sentiment analysis

Often we are not even aware we are interacting with NLP tecnologies, like when a machine read GRE or SAT exam essays (still, an human read it too) or a company provides automatised analysis/report on data, and how much this technology impact already our daily life (it is worth however consider some of the criticisms, on the risks of potentially amplifies biases and assumptions in "robo-journalism" software).

There are lots and lots of different opportunities, and it's a very fast-growing area. You can have a job, but also make an important social difference, for example automatising the creation of a US Mass Shooting database from news sources.

We turn now to the question: why machines are not yet perfect? Why they can't understand language perfectly in general settings (opposite to narrow applications)?

Natural language, contrary to programming language, is ambiguous in its nature, with words holding different meaning in different contexts (“Iraqi head seeks arms”) or the scope of the syntactic phase may be ambiguous (“Ban on nude dancing on governor’s desk”).

But because we kind of understand contextually we actually can get to the correct interpretation (nobody is trying to dance nude on governor’s desk).

A machine actually has much more difficulty when it’s trying to disambiguate a sentence.

Note however that these could be ambiguous sentences for humans as well, as when the air traffic control center communicated a “Turn left, right now” leading the pilot to misunderstand and crash… or so was reported by a French linguistic to claim superiority of the French language over the English one…).

An other example of an ambiguous sentence: “At least, a computer that understand you like your mother”

The first is an **ambiguity at the syntactic level** (syntax: how the different pieces of the sentence fit together to create meaning): it is the computer that understand you like your mother does, or it is that the computer understand that you like your mother ? Each of this interpretation would correspond to a different parsing of the original tree.

The second is a **word-sense ambiguity** (aka “semantic ambiguity”): the word “mother” means both your female parent but also a stringy, slimy substance consisting of cells and bacteria (for example the dried mother yeast added to flour to make bread or those added to cider or wine to make vinegar).

The third one is an **ambiguity at discourse level** (*anaphora*)(discourse: how the sentences are tight together): because sentences do not show up in isolation., there could be ambiguity on how the pieces actually show up in a context of other sentences. For example “Alice says that they’ve built computer that understands you like your mother, but she … doesn’t know any details … doesn’t understand me at all”. Here the personal pronoun “she” may co-refer to either Alice (most likely in the first case) or the mother (most likely for the second case) .

Finally, some ambiguities varies across languages. Depending on the specific language that you are operating, some things will be harder and some things will be easier. **Tokenisation** (identification of the units in the sentence, a sub task in any NPL task) is easy in English language, thanks to space separation, puntuation), much harder in some Asian or Semitic languages. Same for **named entity detection**, easy in English due to strong hint given by capitalisation (“Peter”), more challenging in other languages.

19.5. NLP - Symbolic vs Statistical Approaches

Symbolic approaches usually encode all the required information into the agent whereas statistical approaches can infer relevant rules from large language samples.

Knowledge bottleneck in NLP

to perform well in NLP tasks we need both:

- Knowledge about the language
- Knowledge about the world

We can employ two possible solutions:

- **Symbolic approach**: encode all the required information about the area into the agent, so that this can act based on these “rules”
- **Statistical approach**: infer language properties from large language samples

Symbolic approach was very common in early AI, and structured rules for “languages” to be used in specific domain were developed. This was in part the result of the Chomsky critic (1957) that a statistical approach could never distinguish between a nonsensal, never used before, grammatically correct sentence and one that other than nonsense would have been also grammatically incorrect. This view lead to the development of systems like SHRDLU or LUNA that, for very limited domain (like instructiong a robot to move boxes around using a natural language) looked promising. These systems however didn’t scale up for larger domains, as to operate these tecnologies you need to be backed by a full model of the world, and then natural language is done on top of this reasoning. The problem was that you really need to encode by hand, all of those relations, there was no learning. Not a scalable solution clearly.

It was only in the early nineties, with teams lead by Prof Jelinek, that the statistical approach really started to take off. In 1993 it was released the “Penn TreeBank”, a parsed and manually annotated corpus of one milion words from 2499 news stories from the Wall Street Journal (WSJ). So now the machine had a sentence and had a corresponding parse tree and concepts as training, learning and evaluation could rise in the NLP domain.

To illustrate the difference between statistical thinking and symbolic thinking, let’s look at the task of determiner placement, i.e. given a text where determiners has been omitted find their correct placement. Determiners in English are, simply put, words that introduce a noun. They always come before a noun, not after, and they also come before any other adjectives used to describe the noun. Determiners are articles, but also demonstrative or possessive pronouns, or quantifiers (“all”, “few”, “many”…)

A symbolic approach would try to follow grammar rules for their placement, but there are so many rules… the placement of determiners depends by the type of noun, whether it is countable or not countable, and then in turn this bring back the concept on what is a number, whether it is unique … end, still, so many exceptions!

It is so hard to encode all this information, and it’s very hard to imagine that somebody would really remember all these rules when they move to a new language.

So now, let’s think about it in a way, the statistical approach, which is the way we’re thinking about this problem after this class. Given many labelled texts, for each noun we can encode features as it is singular or plural, it is the first appearance? We can ask many feature type of question, and then we can just run a classifier and get very good results.

So this illustrates how we can take a problem, which is really hard to explain directly with the linguistic knowledge, with world knowledge, and cast it instead in a very simple machine learning problem and get a satisfactory solution.

Note however that the ML approach is conditional to the scope.

We're not making a claim that machines really understand the whole natural language, but we're just saying we want the particular task to do, let's say, spam prediction. We're going to give a lot of instances of material related to spam, and machines will learn it. They still don't understand language, but that's something useful.

18.6. Word Embeddings

Statistical approaches using classifiers over manually encoded features are great, but the job remains still hard.

One of the problem is that there is still a severe issues related to sparsity problem. If we take at unigram, the coverage improves, it's great. However if we are looking at pairs (bigrams), meaning we're looking at a bigger construct because they have more information, you can see that their frequency of occurrences (of arcs in parsing) is very low.

And even if we increase our training size, we still cannot get good frequency count.

And even if you compare across languages. Some languages, like English, are not very morphologically rich, words will repeat many times. But in others, like in Russian, German or Latin, because of the case system, words get different endings. So if you say the dog, depending how the dog appeared, will have different ending. As consequence, such languages will not have sufficient frequency, will have a lot of sparse sounds, and their performance, keeping equal the amount of training data, would be much lower.

Neural networks have improved earlier classifiers because removed the need to manually encode features for words People, before neural models came into play, were spending a lot of time designing what was a good pattern, a good feature representation. Neural networks actually eliminated the need to do this kind of hand engineering in feature selection, replacing it with a "simple" bag-of-word approach (a one-hot encoding measuring presence or absence of each single word in the sentence using a vector of dimensions equal of the language vocabulary) or with the similar frequency representation.

Still there is a problem with such approach, as the bag of word representation can't encode the "distance" between two words. In such encoding there is no difference in a "distance" between a bulldog and a beagle on one side, and a bulldog and a lipstick (there is not even a metric indeed)

So, we want instead to move from this view of word embedding and translate it instead into a dense continuous vector, in such a way that now the distance (and specifically, the cosine distance) between the word dogs bulldog and beagle will be very close to each other.

Now we don't need in some ways annotated supervision. We can just take large amounts of English text and create models designed for language modelling tasks, where, given some partial sentence, you need to predict which word will come next.

We would like to learn word embeddings that are much less sparse than one hot vector based encoding because reducing the sparsity of input features lowers the sample complexity (number of training examples required to do an accurate task) of the downstream text classification task.

In order to do the above, we should cluster the similar or related words together in the embedding dimension space. For instance, the words “bulldog” and “beagle” must have similar embedding representations than “bulldog” and “lipstick”. One way to learn word embeddings is by maximizing cosine similarity between words with related meaning. Word embeddings are practically very useful because they can be learnt without any significant manual effort and they generalize well to completely new tasks.

If you run a PCA and select two dimensions (for visualisation) over this embedding of the words, you will note that related terms will be clustered together, even if the machine don’t know anything about them.

Using Recurrent Neural Networks we can construct a representation for the whole sentence (rather than individual words) and then use the resulting vector representation, which at that point will embed the meaning of the sentence, for translations, “decoding” it in an other language.

RNN works with stream of data, taking the current context, adding a new word, and then producing a new context. And we train them based on the task at hand, whichever task we have to solve. Because here, we will estimate all this ways based on the classification task we want to resolve. And maybe depending on different tasks, we’re going to get different vectors.

LSTM or Recurrent Neural network based approaches encode an input sentence into a context vector capturing more than just a summation of its constituent parts together.

Having the whole meaning of a sentence as one vector it’s so powerful that you not only can solve a classification task, you can actually stick to it something which is called decoder, which can take this vector and then roll it in a similar fashion and generate a sentence. And in this case, if your decoder is trained on Spanish, you would be able to start with Spanish, translate it into this vector, as so that was English translated into a vector, and then decode it in Spanish or in other language. And that’s exactly how this encoder decoder architectures are used today in machine translation.

There are lots and lots of exciting things that are happening in NLP everyday. This is a very fast evolving field nowadays!

Some other resources on reinforcement learning:

- Sutton and Barton (2018) Reinforcement Learning: An Introduction, 2nd decision(Free textbook)
- Kaelbling, Littman and Moore (1996) Reinforcement Learning: A Survey (Survey paper)

Homework 6

Project 5: Text-Based Game

P5.

High level pseudocode of the provided functions in agent_tabular_ql.py:

- main():
    - load game constants
    - for NUM_RUNS:
        - run() --append–> epoch_rewards_test (list of list of floats):
            - q_func = 0
            - for NUM_EPOCHS:
                - run_epoch() --append–> single_run_epoch_rewards_test (list of floats):
                    - for NUM_EPIS_TRAIN:
                        - run_episode(for_training=True) # update q_func
                    - for NUM_EPIS_TEST
                        - run_episodes(for_training=False) --append–> reward
                    - return mean(reward)
            - return single_run_epoch_rewards_test
    - transform list of list of float epoch_rewards_test in (NUM_RUNS,NUM_EPOCHS) matrix
    - plot NUM_EPOCHS agains mean(epoch_rewards_test) over NUM_RUNS

P5.7. Introduction to Q-Learning with Linear Approximation

Intuition for the approximation of the state/action spaces

MDQ and Q-learning are great tools, but they assume a categorical nature of the states/action spaces, i.e., each state/action is treated categorically, where each item is completely independent from the other. In learning, observing the result of state/action $(s1, a1)$ doesn't influence $(s2, a2)$. Indeed, there is no concept of "distance" between states or actions. The sets of states/action itself is not ordered.

In many problems however this is a suboptimal representation of the world. For example, if I am in a state `1000€ saving` and do action `100€ consumption`, intuitively I could borrow the learning from that state/action pair to the (state `1001€ saving`, action `101€ consumption`) pair, as the outcome shouldn't be too much different, or at least should be less different than (state `200€ saving`, action `199€ spending`).

And their (linear or non-linear) approximation try to exactly first learn, and then exploit, this relation between elements of the state/action spaces. In project 5, setting a feature vector based on bag of words allows to introduce a metric on the sentences, where sentences that change by just a word are closer than completely different sentences.

This is really cool, as it allows to map continuous spaces to discrete features and apply a MDP on top of it, well intended, if the nature of the problem admits a metric across these spaces.

Q-Learning with Linear Approximation

We need to create a vector representation for any possible states. In our case, being the states textual description, one immediate way is to use a bag of words approach, that, given a dictionary of size nV (i.e. of the vocabulary for the language of the sentences), returns for each concatenated description of room + description of quest, a fixed length representation as a vector of size nV, with `1` if the word in the dictionary is present in the concatenated sentence, `0` otherwise. We call this `state_vector`.

The idea is that nV should be much smaller of the possible number of original states (descriptions).

Given nA the number of actions, we then set a matrix θ of (nA, nV) that represents the weights we need to learn, such that the q_values associated to all the possible actions of a state s is $q\_value(s) = \theta * \text{state\_vector}_s$, that is the matrix multiplication between the weights θ and the (column) vector representation of the specific state. The resulting vector has length nA, where each individual element is the q_value for the specific (state,action) pair $q\_value(s, a_i) = \sum_{j=1}^{nV} \theta_{i,j} * \text{state\_vector}_s$.

Our tasks is then to learn the θs, and we can use a squared loss function applied to our flow of data coming from the game, where the value of the “estimated” y is the q_value so computed (from the thetas), and the one acting as the “real” one is those computed using the Bellman equation for the q_value:
$y = q\_value(s, a) = R(s, a) + \gamma \max_{a'} Q(s', a', \theta)$ where s' is the destination state observed as result of the action.

The loss function is then $L(\theta; s, a) = \frac{1}{2}(y - Q(s, a, \theta))^2$.

The gradient with respect to the thetas *of that specific action row* is then
$g(\theta) = \frac{\partial}{\partial \theta} L(\theta) = (Q(s, a, \theta) - y) * \text{state\_vector}_s = (Q(s, a, \theta) - R(s, a) - \gamma \max_{a'} Q(s', a',$
(the thetas in the max q_value for the destination state are here as given).

Finally the update rule of the gradient descent is then:

$$\theta \leftarrow \theta - \alpha g(\theta) = \theta - \alpha \left[Q(s, a, \theta) - R(s, a) - \gamma \max_{a'} Q(s', a', \theta) \right] * \text{state\_vector}_s$$

Note that the `state_vector` so described encodes only the states, but one can choose an other form where both states and actions are represented, making it a matrix rather than a vector, and still using a matrix multiplication with a weight matrix in order to get a linear approximation of it.

Q-Learning with non-linear Approximation

And we can now think of why do we even have a linear approximation for the q_value ? Can we not parametrize it by using a deep neural network, where we plug in the state and the action as a specific vector representation, this vector representation goes through a neural network, which gets not necessarily just an inner product with the (unknown) parameters, and the out comes as the corresponding q_value ? And at this point we can train the parameters associated with that deep neural network in the same way as we did for the linear case, with a loss function where we compare the q_value obtained by the neural network in evaluating (s,a) with those arising from the Bellman equation.

This is then why we don’t want to store potentially all possible (state,action) pairs. Instead, we will just be storing a parameter theta, and every time we want to compute the q_value, we kind of come up with a vector that represents the (state,action) pair, and then feed it through either a linear function or a neural network, and get the q_value(s,a) that we’re supposed to get.

[MITx 6.86x Notes Index]