WEEK 6, MODULE 1

INTRODUCTION TO ARTIFICIAL INTELLIGENCE

QMIND EDUCATE

This material was created by QMIND Educate, and is for the sole use of students registered in “A Brief Introduction to Artificial Intelligence”. It shall not be distributed or disseminated to anyone other than students registered in this class.

KAELAN LUPTON

TABLE OF CONTENTS

This material was created by QMIND Educate, and is for the sole use of students registered in "A Brief Introduction to Artificial Intelligence". It shall not be distributed or disseminated to anyone other than students registered in this class.

Week 6 Module 1: Reinforcement Learning 2

A Quick Recap…

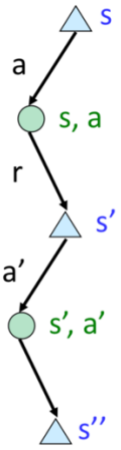
Last week we were introduced to **Reinforcement Learning**, a new method for solving an *MDP* by looking for its *optimal policy π(s).*

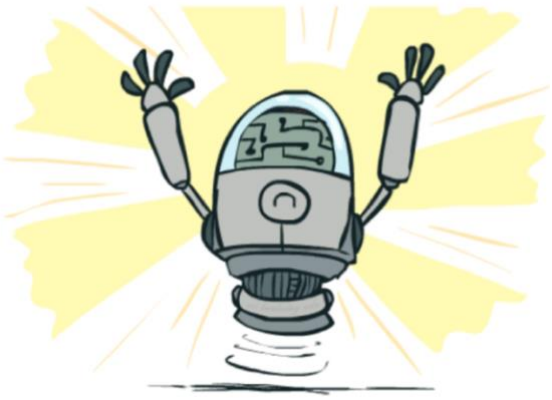
Recall: an MDP is comprised of…

- A set of states $s \in S$
- A set of actions (per state) $a \in A$
- A model (transition function) T(*s,a,s'*)
- A reward function R(*s,a,s'*):

Now, when do we need to use *reinforcement learning*? In many situations, we won't initially know the **model T** or **reward function R**, so our agent must try out actions to understand the world and data around it. The goal is for us to repeatedly observe pairs of states and actions from those states (*s,a*), which we can refer to as **sample outcomes**. As we gain these samples of our environment, we compute averages over **T** using the sample outcomes to gather information about the world around our agent. *Basically, our agent tries as many different actions from as many different states as possible to see how they turn out, and learns from this!*

Model-Free Learning, or **temporal difference learning**, involves some of the practices discussed above. It is used when *we don't know T or R in our MDP* and are looking to find the **optimal values**, **q-values**, and **policy.** Our agent experiences the world through **episodes**, making **transitions** between states using actions and *judging their rewards*. Using our familiar variables, this would look like:





(*s,a,r,s',a',r', s'', a'', r'', s''', a''', r'''*…)

Updates are made to our estimates of values and model after each transition (*s,a,r,s'*) based on the value of the current state (or state-action pair if you're judging a q-value). These updates work by adjusting the utility estimates toward an equilibrium which holds when we've correctly estimated the utility. Eventually, these updates will begin to replicate the **Bellman updates** we've discussed in the past! Recall: due to the **stochasticity** of our environment, these updates *are not always perfect*!

Model-Free learning compares well with Model-Based learning, which won't hold much bearing on the incoming module. If you're looking for a refresher on Model-Based learning, refer to week 5, module 2!

This material was created by QMIND Educate, and is for the sole use of students registered in "A Brief Introduction to Artificial Intelligence". It shall not be distributed or disseminated to anyone other than students registered in this class.

Q-Learning Review

We use **Q-Learning** when estimating the **q-values** for each state. If we knew the values of **T** and **R**, we use the following q-state equation for **q-value iteration**:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right]$$

But, recall that in model-free learning we *don't* know the values of **T** and **R**. Instead, we will continue with our strategy of *computing the average as we go* by receiving sample transitions.

We discussed incorporating a sample estimate into a running average for a q-value, the equation for which looked like:

$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha * [\text{sample estimate}]$

Where α is our *learning rate* (how much value we want to give to new samples, a number between 0 and 1) and our *sample estimate* is of the formula $r(s,a,s') + \gamma \max Q(s',a')$.

A spectacular property of *Q-Learning* is that it will **converge** to find *optimal* q-values e*ven while not executing optimal policy*! This is called **off-policy learning**, and happens so long as…

- The agent explores enough (by enough, we mean the agent visits every (*s,a*) pair "infinitely" often, or often enough to converge to the values that would be found if each pair were *actually* visited infinitely often)
- The learning rate α must eventually be made small enough, but also while not decreasing too quickly, or else the initial states will have too great an influence.

The following diagram illustrates which techniques to use in each situation of MDP solving.

Known MDP: Offline Solution

Goal	Technique
Compute V*, Q*, π*	Value / policy iteration
Evaluate a fixed policy π	Policy evaluation

Unknown MDP: Model-Based

Goal	Technique
Compute V*, Q*, π*	VI/PI on approx. MDP
Evaluate a fixed policy π	PE on approx. MDP

Unknown MDP: Model-Free

Goal	Technique
Compute V*, Q*, π*	Q-learning
Evaluate a fixed policy π	Value Learning

For a refresher on any of this material, check out Week 5, Module 2!

This material was created by QMIND Educate, and is for the sole use of students registered in "A Brief Introduction to Artificial Intelligence". It shall not be distributed or disseminated to anyone other than students registered in this class.

Exploration

The simplest method of exploring a state space is to take random actions based on an **ε-greedy** (pronounced epsilon-greedy) function. The idea is to flip a coin, where each side of the coin is weighted with ε and 1- ε. If the coin lands on the side with probability ε, the agent acts randomly; if it lands on the side of probability 1- ε, the agent makes an action based on the current policy. *Therefore, the greater the ε value, the more randomness you allow state selection.*

The ε value can be chosen and altered by the programmer! When first exploring a state space, it makes sense to have a *larger ε* so to encourage *more random actions* to gather some first impressions. As your agent gains information, however, it will probably be beneficial to use a *smaller value of ε* to encourage actions based on the *learned policy*. Random actions are *slow* and often *sub-optimal*, so **choosing actions based on the policy the agent is learning will result in greater value.** We can still improve on this though!

Exploration Functions

The idea behind using an **exploration function** is to explore based on how much uncertainty remains about certain state-action pairs, rather than just exploring a fixed amount. Here is an example of an exploration function:

f*(*u*,*n*) = *u* + *k*/*n

This equation takes a value estimate *u* and a visit count *n* (how many times you've visited the state before) and returns an optimistic utility. Variable *k* is a constant which represents how much exploration you want to do. Recall the regular Q-Update formula:

$$Q(s,a) \leftarrow_\alpha R(s,a,s') + \gamma \max_{a'} Q(s',a')$$

When using exploration functions, we modify this formula a bit. Take a look at the new formula:

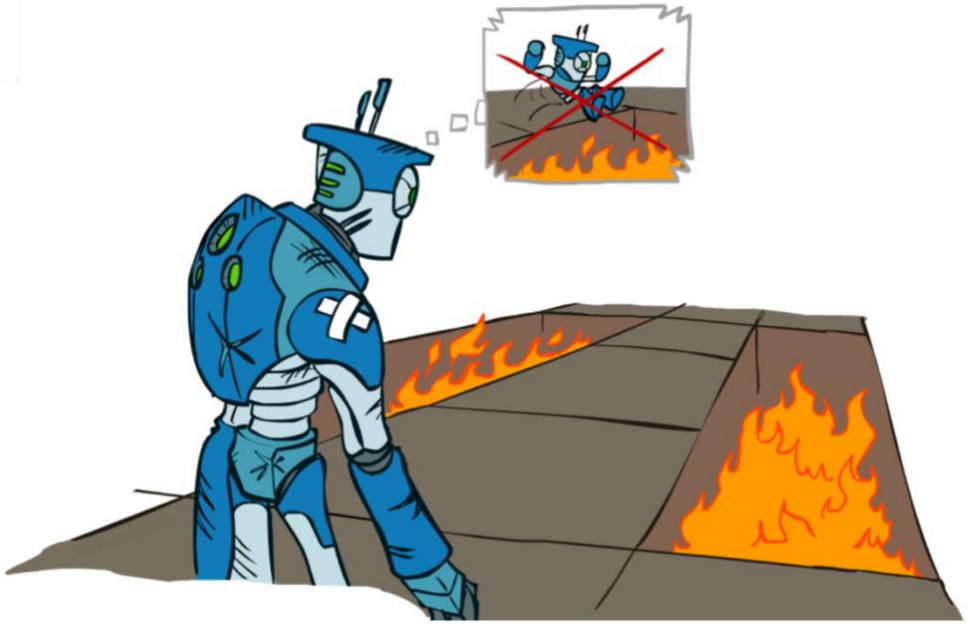
$$Q(s,a) \leftarrow_\alpha R(s,a,s') + \gamma \max_{a'} f(Q(s',a'), N(s',a'))$$

As you can see, there is a change at the formula's end. Instead of making a Q-Update based the q-value of (*s'a'*), we're making a Q-Update based on *the exploration function mentioned above*, which considers both that q-value of (*s',a'*) and the number of times *N* the agent has visited that state (*s',a'*). *This gives a bonus to states you have seen less frequently or never* (lower *n* value = smaller number in the denominator of the *k*/*n* portion of the exploration function), which in turn leads you toward states you haven't seen before to explore everywhere! *A product of this is that as you approach visiting a state infinitely, the exploration bonus will have no more influence.*

This material was created by QMIND Educate, and is for the sole use of students registered in "A Brief Introduction to Artificial Intelligence". It shall not be distributed or disseminated to anyone other than students registered in this class.

Regret

When you don't know what your transition model is, there's no way for you to compute or execute the optimal policy without testing for it. Therefore, even if you do learn the optimal policy, you're still going to make mistakes in the process! The total cost of these mistakes is called your **regret**. To expand, *regret is the difference between your (expected) rewards and your optimal (expected) rewards*. This includes any sub-optimality you may have experienced in your agent's youth while actions were mostly random!



In attempting *to minimize regret*, **you must keep track with each transition about how much regret you are accruing;** it's *not* just about eventually finding the optimal policy. Different methods of finding the optimal policy will lead to different levels of regret. Of the examples we've seen so far…

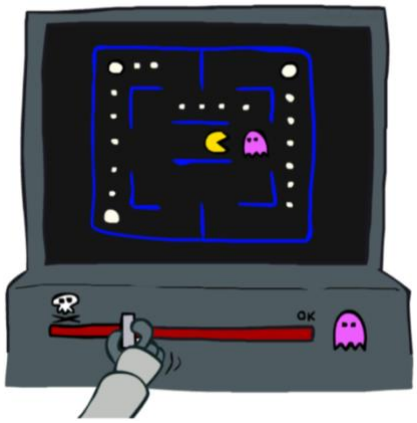
Random Exploration (ε-greedy): eventually optimal, very high regret. Reminder: your ε value is how much randomness you're willing to allow in action selection.

Exploration Functions: also eventually optimal, less regret due to more targeted exploration (though still with significant regret).

So, what then will our next step be to minimize regret?

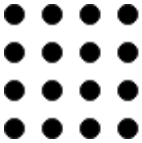
This material was created by QMIND Educate, and is for the sole use of students registered in "A Brief Introduction to Artificial Intelligence". It shall not be distributed or disseminated to anyone other than students registered in this class.

Approximating Q-Learning



A basic Q-Learning method plans to keep a table of all possible q-values. While that sounds great in its completeness, it's far from reasonable in a realistic situation! *Learning about every possible state simply cannot be done; between the time it would take to visit each state in training, and the memory needed to hold the q-values of each state, it's often impossible.*

To give an idea of how rapidly the number of states increases in a state space, consider a 4 x 4 grid, where each grid region can hold a dot. In each state, each grid has the option to either hold a dot or be empty. This gives us as many as 16 dots in any given state, and as few as 0. When calculating how many possible states exist in this state space, the size numbers quickly gets out of hand. Despite there only being up to 16 dots present, there are 20,229,789,888,000 possible states in this example. **Absurd**! *In the name of transparency, I feel it important to clarify that this math does not fall into the scope of this course – it is strictly to demonstrate a point.*



It quickly becomes obvious that we will need to utilize a shortcut to solve these problems.

Generalizing Across States

To tackle this problem, we will consider **generalizing knowledge across states** to save both time and memory. To do this, we will have our agent learn from experience form a small number of training states, then extrapolate that information to generalize the experience to new and similar situations. *This is one of the first major steps we will take towards developing working machine learning algorithms, as it's a fundamental concept in the field.*

An example of the type of generalizing we will look to do can be found in the three Pacman boards below. Is the left diagram really much different than the center diagram? Or the left from the right, which is only missing one extra dot in an unrelated location? Hardly – these states can likely all be generalized by looking at what is happening: the agent is trapped between two ghosts, and likely facing a loss very soon. This is evaluating a state based on what is really happening in that state – something we will look further into in the next section.

This material was created by QMIND Educate, and is for the sole use of students registered in "A Brief Introduction to Artificial Intelligence". It shall not be distributed or disseminated to anyone other than students registered in this class.

Future-Based Representations

A solution to our problem can be found in using better methods to describe our states, and evaluating our states based on these descriptions. We will move forward by describing our states using **a vector of features or properties,** with **weights** assigned to each of those figures or properties. *This is almost like developing an artificial intuition!*

A **feature** (sometimes called a **basis function**) is a **function** that maps a *state* to *a real number* to capture and describe an important property of that state. The "real number" is often a binary 0/1 Boolean value, describing a true or false scenario. In the Pacman game above, several great examples features would be:

- The distance to the nearest dot (integer or float/double value)
- Distance to the nearest ghost (integer or float/double value)
- Is Pacman in a corner? (Boolean)
- Ghost distances from each other (integer or float/double value)
- Number of ghosts (integer)
- Does the action move Pacman closer to food? (Boolean – interesting to note as well, this feature describes a q-state)
- Etc.…

Another feature example would be asking if the state Pacman is in equals some specified state that may hold some significance, which would also be a Boolean value. However, it is best to limit features of this nature because they increase your feature space, giving you many weights to learn without much gain due to the inability to generalize those features across the state space effectively.

When designing features, the programmer must ensure they create enough difference between these features to be able to distinguish between states. You need to have features that helps you understand the circumstances the agent is in. An example of this would be making sure to include a feature *describing the direction a ghost is travelling as well as the distance your agent is from that ghost*.

Linear Value Functions

Look at the following equations called **linear value functions,** demonstrating future-based representations; the first equation is for calculating value and the second for calculating q-value.

$$V(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$
$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a)$$

In these equations, each variable *f* is a *feature* and each related *w* is that *feature's weight*. Intuitively, the value function is based only on *states*, while the q-value function works with *state-action* pairs. These equations can be very useful as they *sum up an environment in a simple series of numbers* – but as

This material was created by QMIND Educate, and is for the sole use of students registered in "A Brief Introduction to Artificial Intelligence". It shall not be distributed or disseminated to anyone other than students registered in this class.

mentioned previously, the programmer must take care that they observe enough different features to distinguish states from each other.

*Equations of this structure are called **linear combinations**. They are filled with interesting properties, many of which are pertinent to this material, but will not fall under the scope of this course. For information on linear combinations and other work in linear algebra, visit…*

Linear Value Functions & Q-Learning

Now we can combine everything we've discussed in this module thus far to do some really interesting work. We will be using **linear value functions** to do **approximate Q-Learning** with *linear Q-functions*! Recall that a transition is modelled by (*s,a,r,s'*), which describes the state, action, reward, and following state of a transition. As well, recall that the difference between the optimal q-values of the two states is modelled by **[*r* + γmaxQ(*s',a'*)] – Q(*s,a*).**

Let's compare the equations for exact Q-Learning (which we determined will ultimately take up too much time and memory to be feasible) and approximate Q-Learning.

Exact: Q(*s,a*) ← A(*s,a*) + α [difference]
Approximate: w_i ← w_i + α [difference] $f_i(s,a)$

These are very similar to the Bellman equations! With the exact Q-Learning equation, with each step we were updating the q-values based on the difference between optimal q-values multiplied by the learning rate α. In our new method of approximating q-values, *what we are updating is the weights of each feature*. The number we add to the weight is that same difference in optimal q-values multiplied by the learning rate α, but this time it is also multiplied by the feature value $f_i(s,a)$.

A more intuitive way of seeing the equation is that we're simply adjusting each of the weights of the features that are active. For example, if a feature is inactive in a given state it will have a value of 0, rendering the entire "α [difference] $f_i(s,a)$" portion of the equation 0 as well. Therefore, there is no update to the weight because the feature is irrelevant at that point. It works in reverse too: if something particularly good or bad happens, this linear combination structure allows us to properly attribute blame to whichever features were responsible.

Of note: now that we're using the numerical values of our features in equations, *it is best practice to ensure all your features are on the same scale*; setting each feature to fall in a range between -1 and 1 is a good standard. This ensures that no feature has too large of an influence on q-value approximation, which makes learning very difficult.

S

S'

This material was created by QMIND Educate, and is for the sole use of students registered in "A Brief Introduction to Artificial Intelligence". It shall not be distributed or disseminated to anyone other than students registered in this class.

Example: look at the diagrams above of example states *s* and *s'*. Say you only have two features, which are distance from nearest dot and distance to nearest ghost. The current representation of these features in the linear value function is $\mathbf{Q(s,a) = 4.0f_{NearestDot}(s,a) - 1.0f_{NearestGhost}(s,a).}$ The feature values we have for the action north are as follows:

$\boldsymbol{f_{NearestDot}(s, North) = 0.5}$

$\boldsymbol{f_{NearestGhost}(s, North) = 1.0}$

Making our linear value function…

Q(*s*, North) = 4.0(0.5) – 1.0(1.0)

Q(*s*, North) = 1

In taking this action, our agent is immediately eaten by a ghost, gathering a reward of -500 for in return. In state *s'* there is no next action to take, so for the sample state (the state we just entered, where we were tragically eaten)…

$\boldsymbol{r + \gamma maxQ(s',a')] - Q(s,a) = -500 + 0 = -500}$

Making the difference between states = -500 – 1 = -501

Finally, we can plug a number into the function to update the weights of both the dot and ghost features.

$\boldsymbol{w_{NearestDot} \leftarrow w_{NearestDot} + \alpha\ [-501] * 0.5}$

$\boldsymbol{w_{NearestGhost} \leftarrow w_{NearestGhost} + \alpha\ [-501] * 1.0}$

Giving us a new linear value function with updated weights…

$\boldsymbol{Q(s,a) = 3.0f_{NearestDot}(s,a) - 3.0f_{NearestGhost}(s,a)}$

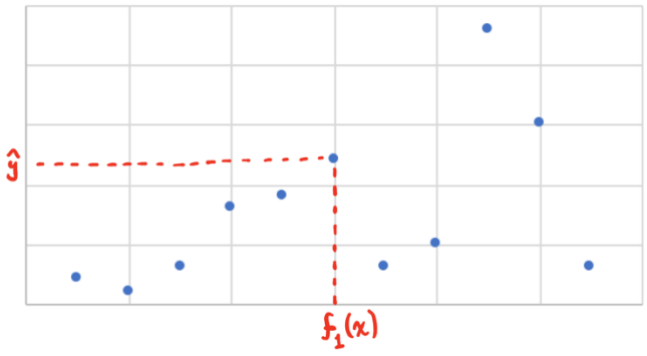
Note: it is good practice to include code that ensures you're exploring all of your features by giving a "bonus" to features you don't know anything about. This way, no features are getting ignored that may prove beneficial to make use of in the long run! We're trying to learn about which combinations of features work well, and which combinations don't.

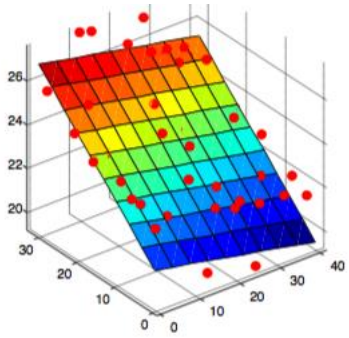
This material was created by QMIND Educate, and is for the sole use of students registered in "A Brief Introduction to Artificial Intelligence". It shall not be distributed or disseminated to anyone other than students registered in this class.

Looking at Least Squares in Q-Learning - Theory

While we won't go too in depth with the mathematics, in this section we will begin talking about optimizing our features using a least squares method.

We can use a **least squares method** to to fit a line on to a set of points. The line will act as a **prediction** for future values, which is a major goal of machine learning. In our situation, those sets of points would be values of a feature; in two dimensions, we would only have one feature and many different values of that feature, but more commonly we will see work with three dimensions – many features, and many values for each.





Predictions in 2D: $\hat{y} = w_0 + w_1 f_1(x)$

Predictions in 3D: $\hat{y} = w_0 + w_1 f_1(x) + w_2 f_2(x)$

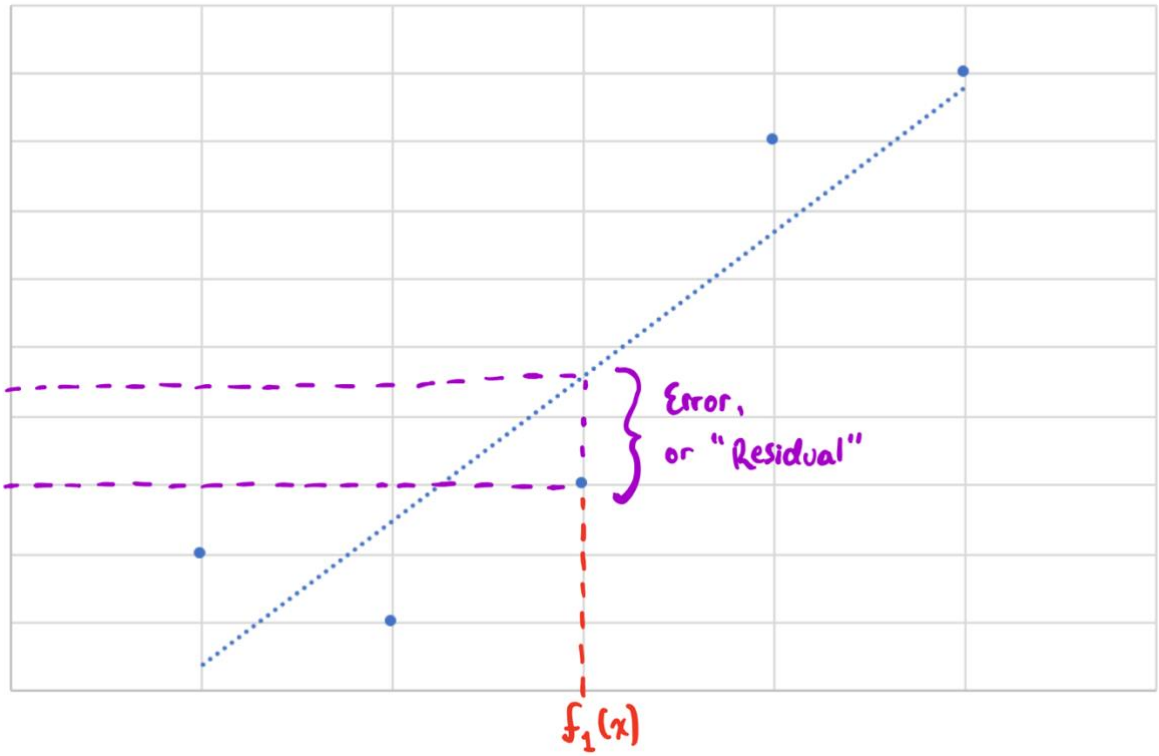
The w_0 value acts as a bit of an offset which raises the prediction line to a point on the y axis. If you're familiar with the equation of a line being $y = mx + b$, you can see this offset plays a similar role to the b value!

Optimizing our Least Squares Approach

Don't worry about knowing how to actually derive the mathematics in this subsection – this is merely to provide a bit of a backbone of explanation to why a least squares approach works. We won't be calculating any of this just yet!

This material was created by QMIND Educate, and is for the sole use of students registered in "A Brief Introduction to Artificial Intelligence". It shall not be distributed or disseminated to anyone other than students registered in this class.

We can measure how accurate the fit of our line is by measuring the error residual error of our line. If we measure the distance between a point and the prediction for that point on our line, we get a value called the **error** or **residual.** To make sure we're using all positive values, we will then square that residual value.



If we sum up the residuals of all our points, we have a number to represent the total error in our line. The formula for this is as follows:

$$total\,error = \sum_i (y_i - \widehat{y_i}^2)$$

Then, if you take the **derivative** of this error with respect to the weights you can get an idea of direction you need to nudge the function to bring it closer to the optimal solution, *which looks a lot like our q-value updates above. As you see data, you make updates and try to bring your estimate closer to being an accurate prediction.*

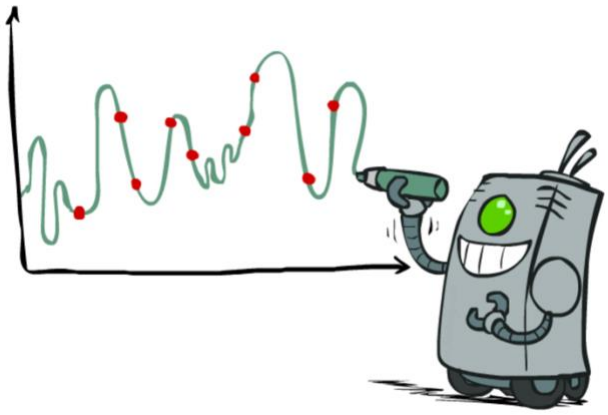
If you're looking for a review of derivatives and their concepts, we'll have the goods in this week's assignment.

This material was created by QMIND Educate, and is for the sole use of students registered in "A Brief Introduction to Artificial Intelligence". It shall not be distributed or disseminated to anyone other than students registered in this class.

Overfitting

One last bit of theoretical work to introduce you to: the concept of overfitting. We will look at this far more in depth when we get to building machine learning algorithms, but it's a subject worth approaching in the meantime!

When fitting a line to your points, it is possibly to "*over-fit*" your line. What I mean by this is you can make your prediction too specifically based on the data points you've seen while not taking care to generalize it across the potential state space.



The grinning robot to the right of this paragraph is demonstrating perfectly what overfitting a line might look like—too many fluctuations and too many curves. A straight line will often preform far better than one demonstrated here, because the predictive values are more accurate and less influenced by ups-and-downs. **A great method to avoid overfitting is to *not train on your entire set of training data***. If you leave about *20%* of your training data out, you can use it as **testing data**, running it through after you finish training to see how well you did. If your model performs admirably, you fit your line well; if not, it is likely your line has been over-fit.

Policy Search

The last topic we're going to look at (not only just in this module, but in this whole unit! Time flies!) **is policy searching**, which is a departure from the more mathematical work we were doing above.

If we think about it, what we're really looking to learn when solving an MDP is the *optimal policy*. However, what we're measuring for with *Q-Learning* are the *policies* which best *optimize* values or q-values. While they are directly related, often the *best feature-based policies aren't the ones focused explicitly on approximating optimal values*. How do you know the features you chose were perfect estimates of obtaining future rewards? That's just it – you don't! *Policy search* attempts to address this.

While Q-learning is looking to minimize error in Bellman equations, *policies* are looking to *maximize rewards*, not the values that predict them. In a *policy search*, we'll start with a decent solution (which perhaps you find by Q-learning!) then fine tune the values of the feature weights by **hill climbing**, a tactic you may remember from Week 3 Module 1. This hill climbing is not done on the error from our update equations, but rather on the expected rewards in sample runs while using a particular policy.

Steps to Utilizing a Simple Policy Search

- Begin with your initial linear value function.
- Nudge each feature weight up or down to see if the changes reflect *positively* in your policy by running your program a several times and find the *average sum of rewards*.

This material was created by QMIND Educate, and is for the sole use of students registered in "A Brief Introduction to Artificial Intelligence". It shall not be distributed or disseminated to anyone other than students registered in this class.

    - This is best done by fixing a random seed, and using that same random seed each time to ensure fair comparisons across different weight changes.
- If the policy improves: keep those changes. If not: discard them.
- Repeat until satisfied. Make smart choices with your weight increases – if one direction of nudging a weight seems to be working, try to continue nudging it in that direction.

This material was created by QMIND Educate, and is for the sole use of students registered in “A Brief Introduction to Artificial Intelligence”. It shall not be distributed or disseminated to anyone other than students registered in this class.

ASSIGNMENT

As mentioned, if you're feeling rusty about derivatives or feel like delving further into linear algebra, feel free to take a look at the following links.

First up is a YouTube channel titled 3Blue1Brown – honestly, I may have just been looking for an excuse to link his work. The man makes incredible content, breaking down big concepts into the exact ideas they're built on and in ways that are easy to understand. This is a link to his series on the Essence of Calculus:

https://www.youtube.com/playlist?list=PLZHQObOWTQDMsr9K-rj53DwVRMYO3t5Yr

A more mathematical breakdown of how to take derivatives can be found here, at this Khan Academy YouTube video series:

https://www.youtube.com/watch?v=rAof9Ld5sOg

And lastly, a link to 3Blue1Brown's Essence of Linear Algebra series:

https://www.youtube.com/playlist?list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab

If you dig his style, check out his work on Neural Networks as well. Super cool, equally relevant to the artificial intelligence field of course!

This week's assignment work can be found in the following medium article written by Arthur Juliani. It uses a brilliant platform built by OpenAI called Gym, which is essentially an environment filled with various standard machine learning problems to solve with various factors built in. Work along with his tutorial (up to the portion for Neural Networks) to get a beginning handle on Q-Learning and its applications. You will be solving a similar grid-style problem as discussed in this module:

https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-0-q-learning-with-tables-and-neural-networks-d195264329d0

And here is a link directly to the gym website, which is provided in the article as well:

https://gym.openai.com/envs/FrozenLake-v0/

If you're keen, feel free to explore further down the article and play around with its built in Neural Network functions using TensorFlow. TensorFlow is an enormous player in the machine learning community, used by nearly any major company you can think of – this is the real deal!

'Til next time – best of luck!

This material was created by QMIND Educate, and is for the sole use of students registered in "A Brief Introduction to Artificial Intelligence". It shall not be distributed or disseminated to anyone other than students registered in this class.

REFERENCES

Artificial Intelligence: A Modern Approach, Global Edition. (2018). 3rd ed. Stuart Russell and Peter Norvig.

Ai.berkeley.edu. (2015*). Berkeley AI Materials*. [online] Available at: http://ai.berkeley.edu/course_schedule.html [Accessed 25 Mar. 2018]

This material was created by QMIND Educate, and is for the sole use of students registered in "A Brief Introduction to Artificial Intelligence". It shall not be distributed or disseminated to anyone other than students registered in this class.