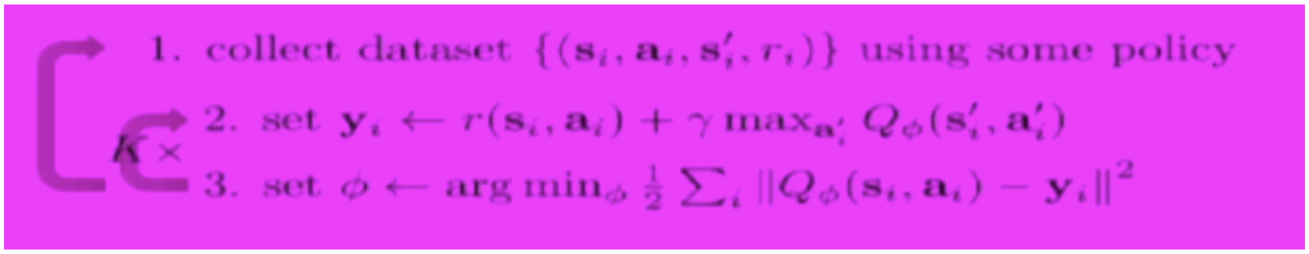
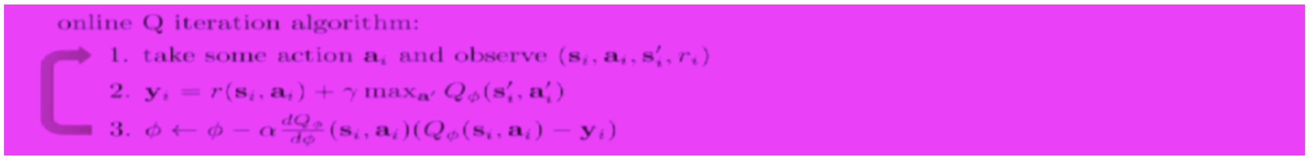
Q learning algorithms

- read the notes on MDPs, value iteration and policy iteration as a background for the notes on value function methods.
- we've seen the modified policy iteration. that was still for discrete state and action spaces.
- now to make value iteration work for continuous state spaces, we will use fit a model to value functions with data samples so that instead of maintaining a table of values, we will have a trained model to predict values.
- now once we do the above (fit a value function for each policy) the problem is we can't choose a policy that maximizes the value to go if we don't know the transition dynamics → because we won't know which action leads to which next state. So instead of learning the value function, if we learn the Q-function, $Q(s, a)$, we can just take the action that has the max Q-value without having to know the transition dynamics. this means that instead of fitting value functions, we will learn q-functions from sample data.
- summary of fitted q-iteration



```
1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy
    $K \times$
    2. set $\mathbf{y}_i \leftarrow r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'_i} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
    3. set $\phi \leftarrow \arg\min_\phi \frac{1}{2} \sum_i \|Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i\|^2$
```

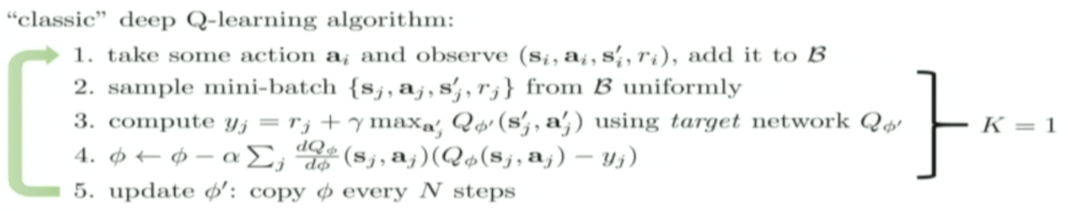
- note that the big advantage of the above algorithm is that it is off-policy. This is because $Q(s, a)$ is the best expected reward after taking action a from state s (for the best policy possible). this means that our data can be collected from any policy → it doesn't matter! all we are collecting is (state, action, next state, reward) tuples. This is the data we use to train our model. This data is independent of any policy so we can collect a large dataset before we begin iterating steps #2 and #3 and not have to update the dataset on each policy.
- now, one caveat is that we will need our policy used during sampling to be "full support" or at least "very wide support". this means that we should sample a wide variety of actions from each state in order to have a diverse set of (state, action, next state, reward) tuples so that we have covered the (state, action) pairs we are actually going to visit when we run the algorithm.
- The online version of the above algorithm samples just 1 data point in step #1. This is called the online q-iteration algorithm or the **q-learning algorithm.** Note that this is an online learning algorithm:



```
online Q iteration algorithm:
    1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
    2. $\mathbf{y}_i = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
    3. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i)$
```

- Again, note that we should make an effort to do a good amount of exploration to increase the support of our policy (to explore more basically)

- the proof of why basic tabular case value and policy iteration work and converge to optimal policy is the contraction proofs we saw for basic tabulated value and policy iteration algorithms in our earlier notes.
- in the case of fitted value iteration and q-iteration/q-learning, in general it does **not** converge and in fact, in practice, there are many cases where it often does not converge. we will, in the next lecture, learn some practical tricks to formulate our problem so that fitted value iteartion/q-learning will converge. in general though, it **does not converge.**
- the batch actor-critic algorithm that we saw in the notes on actor critic algorithms also have the same problem which is that, in general, they do not converge!
- two of the reasons they don't converge is because this steps #2 and #3 are not gradient descent because the target (see step 2) is moving all the time! Secondly, the gradients tend be very correlated from step 2 to step 3. hence no convergence.
- to get around the problem of correlated gradients, we can do parallel sampling from a variety of pre-collected (state, action, next state, reward) tuples. Because these tuples are pre-collected, this technique is called "replay buffers". this increased batch size takes a variety of gradients that are hopefully uncorrelated, unlike collecting a single data point and immediately running gradient descent on that data point etc.
- note the difference in step 3 between q-iteration and online q-learning. in q-iteration step 3 fits the data in step 2, like it were a regression problem. in q-learning, we just take 1 gradient step. in the q-iteration case, even though the whole algorithm may not converge, individual executions of step 3 can converge to fit the target in step 2 (note that this doesn't mean anything good as the overall algorithm still doesn't converge in general).
- we can use a similar idea to make q-learning converge. instead of having a moving target, we will keep our target fixed for a few 100/1000 loops, regress onto that and then update the target network. This technique is called target network. we can combine replay buffers and target networks to get faster convergence. this is called the classic deep q-learning algorithm:



"classic" deep Q-learning algorithm:

```
1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$, add it to $\mathcal{B}$
2. sample mini-batch $\{\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}'_j, r_j\}$ from $\mathcal{B}$ uniformly
3. compute $y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mathbf{a}'_j)$ using *target* network $Q_{\phi'}$
4. $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(\mathbf{s}_j, \mathbf{a}_j)(Q_\phi(\mathbf{s}_j, \mathbf{a}_j) - y_j)$
5. update $\phi'$: copy $\phi$ every $N$ steps
```

step 5 happens only once every few loops, not every loop

- now, fitted q-iteration, online q-learning and deep q-networks are all instantiations of the same algorithm with slightly different settings. that general algorithm has 3 processes running at different frequencies:
  - data collection (along with maybe evicting old data)
  - q-function regression: sampling a batch + gradient descent
  - updating the target network (maybe with something like polyak averaging from previous network → polyak averaging is like target = $\alpha * new + (1 - \alpha) * old$.

  the above steps done at different relative frequencies gives us fitted q-iteration, q-learning and DQN.
- online q-learning does 1 of each of the above step. relative frequency is 1.
- in DQN, the update step is slower but data collection and regression run at the same frequency. (bigger replay buffer)

- in fitted q-iteration: innermost loop is regression, next inner loop is updating the target network and outermost loop is data collection.
- as a rule of thumb, the more we can do regression with a fixed target, the more we can stabilize learning, but the slower we will go since we collect new data about rewards slower this way.
- if you look at the sampling a batch step in the above algorithms, you will see that we set $y = r + max_a Q(s, a)$. Here, if Q is falsely optimistic for certain actions from a given state, we will amplify that noise as we go. so our Q estimates will always be over optimistic. to overcome this, we can use two networks: one to pick an action based on max and another to query the Q to get the sample data $y = r + max_a Q(s, a)$. now, if the noise/errors are uncorrelated, we will do much better! this is called double Q-learning.
- in fact the two networks to use for double Q-learning can be the two networks we already have with vanilla deep Q learning: the current network and the target network.
- we could do multi-step returns with q-learning just like we did with actor-critic algorithms but we lose the theoretical big advantage we have which is that q-learning methods can be done with off policy data. n-step returns theoretically requires sampling from on-policy data (since its a multi step process and we need to have a way to evaluate the action to take at each step). there are ways around this like:
    - just ignore the problem (tends to work quite well!)
    - do importance sampling
- we've talked about doing a $max_a Q(s, a)$ operation while both following a policy and while sampling for training. what if we have a continuous action space? it'd be crazy to do a SGD in the inner loop of training while sampling. something that often works well (with parallelizable GPUs especially) is to do stochastic optimization → which means just sample a few and take the max. this works pretty well, but much less well in higher dimensional action spaces.
- an alternative in continuous action spaces is to use a simpler function to approximate q values such that we can easily max this function over actions → instead of a neural net, maybe a quadratic function, for example → but of course the function is much less expressive so if the q-function is high dimensional/complex we might still need the neural net.
- another alternative is to train a network to estimate the action that maximizes q-function for a continuous space (recent research)
- practical tips for q-learning:
    - test on easy, reliable, small tasks first to make sure your implementation is correct
    - large replay buffers improve stability → more data is good. eg. for atari games, millions of samples is typical.
    - it might take time to train. once you've given it lots of data and are confident your implementation is correct, give it time to train!
    - you should prefer more exploration early on and reduce that as you go
    - bellman error gradients might be really big → gradient clipping might be a good strategy
    - double q-learning works really well → do it always!
    - n-step returns is also often a good idea
    - lots of computational resources helps a lot → a lot of these algorithms tend to have a lot of variance across runs. it is better to run with multiple random seeds → if you are rich or work at google, it helps a lot.