some notes

(Tabular Methods)

- In tabular methods, each q,s,a could be seen as a seperate parameter, which can be set
up during learning. In fact, we have more parameters than states in such methods, since each
state could have n parameters according to # actions available in that particular state.
- It would be ideal to let # parameters be independent from # states; to be better
approximate fn.

(Fn Approximation)

- it's used in RL, because:
    - state & action space may be big or combinatorially enormous, making
    tabular methods impossible to render.
    - in general, we seek the agent to be memory-, space- & data-efficient.

(Monte Carlo)

- Is a model-free method.
- MC approx need too many samples to learn in an env which lasts more than several hundreds
of time steps. This is mainly because of its high variance property; where there are many
random variables with high degree of stochasticity; either from env or from policy.
Consequently, learning will be slow.
- MC targets approximations are just numbers, known at the end of an episode. They do not depend
on any parameter.
- Even though MC sample-based targets are unbiased, they have higher variance than TD sample-based targets.
- In MC, we cannot update the model till the end of the episode!
- MC is an off-line method (updating policy at the end of the episode)

(Temporal Difference)

- Is a model-free method.
- Is more sample-efficient than MC.
- In TD-learning, we approximate 3 times:
    - appr "true value fn" by parametric model
    - using samples as "targets", instead of expected value of targets
    - each sample of a target is itself an appr; since it uses an appr of next state value
- Unlike in Mc, TD targets are biased; in that they are not equal to the true targets in an expectation.
They depend on parameter w.
- TD have targets have lower variance; since they depend on the stochasticity of reward & next state
and NOT on the stochasticity of all rewards & states till the end of the game.
- Update step can be done during the experience (e.g. playing the game itself),
no need till the end of the episode like MC to update the model, which is a huge advantage.
i.e. as few as one step of experience (s, a, r, s_) can be used; to update the model.
- Is an on-line method (updating policy during the episode)

- Is a "bootstrapping" method.

* Given sufficiently flexible parametric model, both MC & TD converge to the same estimate of value-fn.

* Some findings from "TD or not TD" paper:
- Pure MC can outperform TD when the rewards are noisy.
   - TD can outperform pure MC when the return is mostly dominated by the reward in the terminal state.
   - MC tend to degrade less when the reward signal is delayed.
   - MC seems to be on-par with TD when the reward is sparse.
   - MC can outperform TD methods with more complex and high dim perceptual inputs.

(Loss Fn in value-based RL)

- In case of TD, differentiating the grad wrt parameters w will kind of change the natural flow of time!
This will not only make the value est of s,a look more similar to our target, but also will make the subsequent est of return be dependent ont the prev rewards.
That's the main reason of introducing Semi-Gradient Methods.

- TD being a Bootstrapping method; is -in fact- not an instance of true grad descent. Such methods only include part of the grad. That's why they are called 'Semi'-grad.

(Semi-Gradient Updates)

- They disallow adjusting estimate of the next state solely on the basis of info about current
state value & current reward. "time only goes forward" principle.
- they generally simplify the math in the parameters update equation.
- they converge well in practice.
- its update rule does NOT solve the problem of targets "running away" faster,
then the estimates "catching them up" subsequently. other methods have been
developed to smoothen this effect.

(Semi-Gradient SARSA)

- We use sample-based estimate of SARSA target, since it's the only thing possible as we do not know
the env dyn; so we do not know the expected target.

(Approximating Q-fn)

- All 3 methods (SARSA, Expected SARSA, Q-Learning) require R & S_ to perform updates,
however some of these may require other inputs.

SARSA update:
$Q(S,A) := (1-\alpha)\, Q(S,A) + \alpha\, (R + \gamma\, Q(S',A'))$

Expected SARSA update:
$Q(S,A) := (1-\alpha)\, Q(S,A) + \alpha\, (R + \gamma\, E\_{A' \sim \pi(S')}\, Q(S',A'))$

Q-Learning update:
$Q(S,A) := (1-\alpha)\, Q(S,A) + \alpha\, (R + \gamma \max_{A'} Q(S',A'))$

N.B. all the RHS expressions depend on R & S_

* SARSA will be better than Expected SARSA, when:
□- it is impossible to compute an explicit expectation over policy stochasticity.
□- the action space is too large, so approximations can not be integrated over huge action space.

(SARSA & Q-Learning)

- About their approximation:
□- both use the regression loss (e.g. MSE, MAE)
□- both can use same NN architecture for approximating Q-fn.
□- they only differ in their form of update, more precisely in the target expression

(Non-Differentiability)

- Non-diff propagates through time because the value-fn in current state depends on the value-fn in the next state.
- Usually, rewards are indicators of some sort of success (reaching some state, finishing some sequence of actions, etc).
Thus, the returns and their expected values can change abruptly within small change of state.
This may cause problems for differentiable approximations of value functions.

(Non-Stationary)

- The interleaving nature of GPI inevitably causes nonstationary; each policy update changes true value estimates.
- The nonstationary nature in RL may be dangerous; because any inconsistency bet true & estimated values may cause the change in data distribution.
- Env is NOT the only source of nonstationary in RL.
- Nonstationary property may arise from TD or any other method.

(DQN - Study Case)

* No pooling was used in the NN:
□- it can be replaced with the strided conv; yielding more-or-less the same effect.
□- it requires additional computation time.

N.B. Pooling layers have no parameters.
- To substitute the effect of dim reduction that pooling produces, stride was introduced in the first 2 conv layrs.

* The aim of introducing 4 stacked frames; is mainly to eliminate the partial observablility i.e lack of info in the observation.
- it allows to infer velocity & acceleration of objects through the finite differences.
- for some tasks, it may turn POMDP into MDP, which greatly reduce the difficulty of task.

* To speed up DQN training process:
- Parallelize learning.
- Use smaller model. Unlike stacking more layers, this actually helps.
- Do NOT use pooling. This is simply empirical!
- Use strides. Conv with strides reduce the size of tensors passed through the net.

(Experience Replay - Buffer)

- Acting as a soln for Sequential Data:
- it increases sample efficiency.
- it helps against the problem of correlated samples.
- it improves learning stability.
- it can only be used in combination with Off Policy learning alg.

- In order to improve experience replay (buffer):
- Use samples more wisely - taking into account their contribution to
the training process.
- Use larger buffer (up to 1 million)

(Target Networks)

- They mainly improve stability of the learning process.
- They are heuristic against the problem of correlated training data.
- They almost completely decouple Q-fn parameters from paratermeters of target net.
- They play NO role transforming RL to Supervised Learning problem; since the RL problem does not change much. Actually the nonstationary property introduced by GPI still persist.
- They got NOTHING to do with uncertainty of next state estimates.

(Different Scales of Reward Signal)

- In order to help agent adapt to different scales of reward signal:
- Adjust the reward signal gamma.
- Give the agent more training samples with lower training rate.
-Adjust learning rate.

(DQN - Statistical Issues)

- Having stochastic actinos might lead to some approx error in the network,

in particular the Q-values of s_ changes.
- Another problem when dealing with such stochastic actions will be that you're NOT picking the max expectation, but the max over samples; expecting over distributions.
N.B. the expectation of max_a Q(s,a) will be greater than the max over expectation.
- Such problem leads to Q-values being too large. i.e. network is over optimistic.

(Double Q-Learning)

- Idea: training 2 Q-fn seperately; considering optimal action from Q1 and taking this action's value from Q2. This solves the over optimisim caused previously.
Thus, the noise in Q1 will be indepedent from noise in Q2.
- We update the target net; getting the Q-values that correspond to the optimal actions.
We use those values with the current net we're training on.
N.B. since the updates is done every 100k iter or so, indepedence is assumed to a great extent.
- Is a heauristic by the end of the end of the day. No guarantees!

(Advantage Fn)

- is measure of how much is a certain action a good or bad decision givena certain state.
In other words, what is the advantage of selecting a certain action from a certain state.