Practical Reinforcement Learning

Week 1:

- **Duct tape approach**

You try to apply data driven approach without data.

1- Generate some data(banners) by initializing with a random strategy.
2- See whether user clicks on this random banner or not.
3- Record this into some kind of flagging system.
4- System runs for a day to get a lot of data containing clicks (rate<0.5) and non clicks
5- Fit using logistic regression model
6- Predict whether the user will click or not on the banner you show

Problems:

1- It's not always a good idea to follow what it suggests. (Click bait problem or false data) to extract money. Can be solved with making users happy so that he would visit you again.
2- You need to train from time to time as the "current optimal" may never discover something better

- **Multi armed bandit**

Collects data and user gives feedback

1- You define user features (age, book he has, gender, social network)
2- Each action recommend a particular book whenever user visits a website
3- Feedback if the user like this recommended book or the revenue

Problems:

1- Click bait problem as it will make you lose trust from users (It recommends most visited ads) therefore either you ruin the user base (data) or decrease revenue

- **Markov decision process**

1- Environment s : in set of States
2- Agent a : in set of Actions
3- Reward r : to formalize the feedback and given every time step
4- We want to optimize the sum of rewards per session to go as fast as possible over the duration of the entire episode
5- We don't need to optimize the immediate reward as this may result in losing the game of a chess which means wrong moves can be taken.
6- Policy is a probability distribution to assign states to all possible actions
7- In a nutshell, go in state – take action according to policy – receive reward.

- **Cross entropy method**

It is a general procedure of stochastic optimization (Not particularly a RL algorithm)
1- Starts with a random policy and play a few games (100 sessions)
2- Pick most 25 effective sessions called elite sessions
3- Change policy so that it prioritizes actions from elite sessions (increase their probabilities)
4- The policy is simply a set of probabilities
5- Should initialize all the probabilities in uniform random distribution to try all actions
6- After making new matrix from elite states and actions, remember how many times you visited a state and how many times you picked each action then divide to make the probability distribution matrix

Drawbacks:
1- You may visit rare states from the 100 which results in weird probability distribution. Can be solved with increasing number of sessions, but it may be hurting the PD. Another solution is to add a small number in the probability distribution for all actions so there would be no zero probability

- **Approximate cross entropy method**

We don't use tabular policy, we don't record all probabilities explicitly but we use any machine learning model to model the PD given state such as neural network with softmax layer, logistic regression or a random forest
1- Initialize NN with random weights and let it pick actions to play 100 sessions
2- Take elite sessions and train to increase the probability of actions in elite sessions
3- Use any optimization technique as Adam or RMS prop, perform updates given best M sessions
4- Nn.fit(elite_states, elite_actions)

Problems:
1- You use only samples from sessions and it gets smaller if you use larger sample sizes. Can be solved by reusing the samples from past iteration, ex. If you want to sample 100, sample 20 only and use 80 from past iterations.
2- Sometimes fall in local optima (Exploding gradient problem getting NAN but can be solved with NN) but for RL action probability might go to zero and it will avoid taking a particular action which may be the optimal one but not discovered yet

Week 2:

- **Reward Design**

Explains for the agent what we want to do, in terms of cumulative sum of scalar signal which is called a return.

Considerations:
1- For a non-stop system (infinite episodes) is called continuing task, we can split it into chunks, such as hours, days and assess agent's performance during these chunks.

2- Positive feedback problem which allows an agent to gain large (almost infinite) rewards, so it will look for the action that gives rewards in short time and ignores the one that gives high reward in long time.
3- Problem of unboundedly increasing sum of rewards. Which can greatly harm our optimization procedure and thus, could break that learning process.
4- Problems in 2,3 can be solved using discounting by gamma (0<g<1) which focuses agent's attention to close rewards and reduces value of distant one
5- Gamma is partially mathematically convenience and partially inspiration of human behavior. And it is very similar to quasi-hyperbolic discounting $f(t) = \beta\gamma^t$ if we assume β=1, it is a discrete approximation for hyperbolic discounting $f(t) = \frac{1}{1+\beta t}$
6- In a nutshell, gamma is a mathematical convenience which makes infinite sums finite and preserve some amount of contribution for each reward, it also allows us to express the return in a recurrent function.
7- Reward Shifting problem (Should not standardize rewards) as it may become positive feedback loop and doesn't go to the end state thus changing the optimal policy.
8- Scaling doesn't affect reward (can divide by non zero constant) it is helpful if you know the most immediate reward and also useful for approximate methods
9- Reward shaping also doesn't affect optimal policy, it adds a value to immediate rewards without changing the optimal policy using potential shaping function $F(s, a, s') = \gamma\phi(s') - \phi(s)$
10- The intuition, consider γ=1, this means if the current = next state it won't add a value, therefore it won't fall in the positive feedback problem

- **Bellman equation ($E_\pi[R_t + \gamma * v_\pi(S_{t+1}) | S_t = s]$)**

State and Action value functions: (Dynamic programming)

To find the optimal policy in a step by step fashion
1- Breaks the problem into small pieces
2- Launches iterative process until there is no small pieces remain
3- Reuses solved pieces to solve the next pieces

The maximum cumulative reward (return) is random due to the randomness of policy and could be also due to the randomness of environment which can have different reward and next state given current state and action and we can solve this by taking an expectation which corresponds to the Value function.

Value function(state value function):

Depends only on current state, it is the mean reward that an agent can get from environment

$$\sum_a \pi(a|s) \sum_{r,s'} p(r, s'|s, a)\,[r + \gamma * v_\pi(s')]$$

$\sum_a \pi(a|s)$: Policy stochasticity

$\sum_{r,s'} p(r, s'|s, a)$: Environment stochasticity

This is called the Bellman expectation equation for value function

Action-Value function (Q(s,a))

It is the expected return given state and action, the Q function is the mean reward that agent can get from environment after acting action a in state s, following policy π.
It isn't connected with the policy, which can have even have 0 probability.
Has no stochasticity in the first action step and we don't have to sum over possible initial actions

$$Q_\pi(s, a) = \sum_{r,s'} p(r, s'|s, a)\,[r + \gamma * v_\pi(s')]$$

Bellman optimality equation

Generalized Policy iteration (evaluation and improvement)

Model based: (Value based approach)

Knows all probabilities of rewards and next states, given current state and action

1- Build value function
2- Extract a policy from the value

Policy evaluation:

Measures how good a given policy in each state

1- Initialize value function to zero or any initialization as long as terminal states are zero
2- Solve bellman expectation equation v(s) until state values do not change anymore
3- Gives us the direction where we use to improve our policy

$$v_\pi(s) = \sum_a \pi(a|s) \cdot Q(s,a)$$

Policy improvement:

1- Recover Q from value function V which can be computed from policy evaluation
2- Find an action that maximizes Q function which is the best possible action
3- Based on Bellman optimality equation

If Q* is known, the optimal policy can be obtained by taking the action which maximizes its function.

$$\pi^*(s) = argmax(Q(s,a))$$

If V* is known, to recover the optimal policy from it, one need to recover first a Q function from this V* .And this can be done with environment probabilities precisely in the same way as we did in policy improvement, and then recover this Q*.

Generalized policy iteration:

1- Doesn't depend on initialization
2- Not susceptible to local optima
3- It doesn't need complete policy evaluation
4- Doesn't need to improve policy in all states at any particular step. As long as the policy is guaranteed to be updated in each and every state every once in a while
5- Updating one state at each policy evaluation step will converge the GPI to global optima
6- Also updating it in random direction will converge to global optimal policy

Value iteration:

1- Performs only 1 single policy evaluation and full policy improvement
2- Doesn't store the probabilities of actions at each state
3- Starts with initializing v(s)
4- Solves Bellman optimality equation $v(s') = \max_a(Q(s,a))$
   Note: Q learning is applying value iteration with Q function instead of value function

Policy iteration:

1- Requires precise policy evaluation before improvement step therefore we do many evaluations until numerical convergence of state values before doing single improvement
2- Starts with initializing v(s) and π(s) arbitrary
3- Perform policy evaluation then policy improvement

Week 3:

- **Model Free Learning**

We don't have information about transition probabilities or the reward function, as you don't know anything about how the opponent can react, in other words you don't know which state will come next.
You can sample from states and rewards from environment, but you don't know the exact probability of them occurring. Therefore we can't compute the expectation of the possible outcome and this prevents using the optimal policy given value function

Q-Learning:

We only have access to the trajectories and not the whole possible states and actions transitions where the trajectory is a sequence of state, action and reward

Using Monte-carlo:

1- Get all trajectories containing particular (s,a)
2- Estimate the value for each trajectory then average them to get the expectation
3- Can be used in simple problem with few states.
4- Uses full trajectory to learn

TD error (temporal difference):

1- Exploits Q values and finds the maximum of them using state, action, reward and next state
2- We solve the expectation using approximation by taking 5 samples if possible, or take 1 sample as in self-driving cars as there is no go back to try other actions. But this approximated Q value is very noisy and we can't depend on it, therefore we use exponentially weighted average (α) which will allow gradually converge to the average.
3- Learns from partial trajectory an works with infinite MDP

$$Q(s,a) = \alpha(r_t + \gamma \cdot \max_a Q(s_{t+1}, a')) + (1-\alpha) \cdot Q(s,a)$$

Q-Learning algorithm:

1- Initialize with zeros
2- Loop : sample <s,a,r,s'> from env and compute the $\hat{Q}(s,a) = r(s,a) + \gamma \cdot \max_a Q(s', a)$
3- Update using TD method $\hat{Q}(s,a) = \alpha \cdot \hat{Q}(s,a) + (1-\alpha) \cdot Q(s,a)$

Problems with Q-learning:

1- It fails miserably and falls in sub-optimal policy (finds the shortest path and leaves the safer path) as for cliff problem, the agent may fall in the cliff from the epsilon (random action) and the problem comes from taking maximum
2- To solve this problem we go to SARSA (<s,a,r,s',a'>)

SARSA:

1- We replace maximization with expectation $\hat{Q}(s,a) = r(s,a) + \gamma \cdot E_{a' \sim \pi(a'|s')}[Q(s', a)]$
2- It discounts the values of the states on the shortest path because we expect over all possible outcomes and we will get the –ve reward of falling of the cliff
3- $\hat{Q}(s,a) = r(s,a) + \gamma \cdot Q(s', a')$
4- The difference is that we take the Q value of the next state instead of the max Q value but this leads to high policy stochasticity in case we have a lot of actions
5- This problem can be solved using Expected SARSA which considers all possible actions and we compute the average over Q of all actions and therefore removes the policy stochasticity

- **On-Policy vs Off-Policy**

On-Policy:

1- For example SARSA their agent tries to improve its policy online (immediately), plays then improve then plays again and wants to get to optimal policy as quick as possible
2- Agents can pick actions and move on its own policy

Off-Policy:

1- For example Q-learning where you explore using large epsilon, not to behave optimally, but you train to find an optimal policy, and then exploration goes away to pick optimal actions
2- Agents can't pick actions and trains on recorder trajectories
3- Expected SARSA when the probability is set to 1

Week 4:

- **Model free approximation**

In model free, we have limitations on number of parameters which is number of states, also memory to store the q values, data and time. Therefore we need an approximate.

One idea is to reduce number of states or make #parameters independent on #states and we use weights $\hat{v}(s,w) \approx v_\pi(s,a)$, $\hat{q}(s,a,w) \approx q_\pi(s,a)$, this could makes us learn infinite environment but now we have coupling between parameters and states. Therefore any parameter can affect estimates in all possible states.

By doing this we reduced our problem to supervised learning but without specified loss function, training data because there is no outcomes without also inputs (what states possible to the environment) but we can solve this using the 2 approaches, MC and TD

For MC: (G_t is the return which is our goal)

Goal is the expectation of value $E_\pi[G_t|S_t = s]$ and action-value functions $E_\pi[G_t|S_t = s, A_t = a]$ and to approximate the expectation we use sample based estimates of goals $G(s) = R(s, \pi(s)) + \gamma \cdot G_{t+1}$, $G(s,a) = R(s,a) + \gamma \cdot G_{t+1}$

So, now we have : Inputs as states and actions

Targets as sample based estimates of our goals

But for MC we need too many samples to learn with very large number of steps, which in turn causes a large variance due to the sum of many random variables (state, rewards, action in each time step) either due to stochasticity in states and actions. Can be solved using TD

For TD:

More sample efficient than MC, they have less variance because they depend of stochasticiy of rewards and next state and not all of them till the end of the game as in MC

V(s) --> $E_\pi[R_{t+1} + \gamma \cdot v_\pi(S_{t+1})|S_t = s]$ Q(s,a) --> $E_\pi[R_{t+1} + \gamma \cdot v_\pi(S_{t+1})|S_t = s, A_t = a]$
But we want to approximate these expectation with its sample based estimates, but the value of next state is unknown to us. Therefore we approximate it with the parametric estimate.

V(s) --> $R(s, \pi(s)) + \gamma \cdot \hat{v}_\pi(S_{t+1}, w)$ Q(s,a) --> $R(s,a) + \gamma \cdot \hat{v}_\pi(S_{t+1}, w)$

We approximated 3 times : The value function with parametric w

Target sample based of the expectation

The value function of the next state with parametric w

Loss function in supervised learning:

MSE, MAE, or Huber loss can be applied

$Loss = \frac{1}{2} * \sum_{s,a} \rho_{\pi}(s,a)[g(s,a) - \hat{q}_{\pi}(s,a,w)]^2$

It is the MSE of our targets(goals) and the estimates. We should sum over all states and actions and compute the weights of importance $\rho_{\pi}(s,a)$ for the states actually visited and actions done by policy.

Note that the Loss is an expectation of MSE and this is if we assume states and actions are distributed according to distribution of ρ_{π}, and this means we are going to approximate the loss in sample based fashion by sampling ρ_{π}.

Samples of states and actions from distribution ρ_{π} can be obtained by collecting visited states and actions done by the policy π in the environment.

For Off-Policy we can sample from experience generated by behavior distribution

Off policy is a set up where there are two policies.

Behavior policy:

1- Makes actions in an environment and, we are interested in and over which we have a control.
2- Policy is out of our control.
3- We only require Behavior policy to have non-zero probabilities of making actions
4- Collects data

Target policy:

1- The Target policy is updated and improved to become closer and closer to optimal policy.
2- We have another probabilities under our Target policy.
3- Subjected to evaluation and improvement to become close to our optimal policy
4- This assumption is required to learn well in off policy scenario

On-policy:

1- Learning is a slightly easier task.
2- Target policy is the same as Behavior policy, so we are in control of acting in that environment, and in control of changing the policy.
3- But these changes, we could influence the data which we are collecting, and explores areas on the environment, which are of our interest.
4- If algorithm can learn off policy, it also can learn on policy, but not the other way around.

Off policy and On policy may seem similar to online and offline learning, however, in varying versions learning works online, and offline are more frequently used as reference to updating policy doings opposite that is online, or updating the policy only when the opposite ends, that is offline. So Temporal difference methods can be described as online methods, while Monte Carlo methods, can be described as an offline methods.

Minimizing the loss using Gradient decent:

$$w = w - \alpha \cdot \nabla_{\mathrm{w}} Loss(w)$$

Since we can only compute the Loss with its sample based estimate, we can approximate the true gradient with its stochastic estimate, thus using stochastic gradient decent

SGD:

we approximate the full gradient with its estimate in a particular state and action which are sampled from ρ_π behavior policy by sampling from agent's experience

$$w = w - \alpha \cdot \nabla_{\mathrm{w}} Loss_{s,a}(w)$$

Where, $Loss_{s,a}(w) = [g(s,a) - \hat{q}_\pi(s,a,w)]^2$

But how we are going to differentiate g(s,a) with w.r.t "w"?. In MC the goal is simply numbers, but in TD it has a dependency on "w"

SGD will make the value estimate of current state-action look more similar to our target, but will also will make the subsequent reward be dependent on the previous reward, thus Semi-SGD

Semi-SGD:

Treats the goals as fixed. Therefore $\nabla_w g(s,a) = 0$, and by this we have reduced our problem to supervised learning where the goals are almost always fixed

$$w = w + \alpha \cdot [g(s,a) - \hat{q}_\pi(s,a,w)] \nabla_{\mathrm{w}} \hat{q}_\pi(s,a,w)$$

Properties:

1- Treats goals g(s,a) as fixed
2- Updating changes parameters to move closer to targets
3- Ignores the effect of update on targets (unlike gradient update)
4- No, SGD convergence properties
5- Converges reliably in most cases
6- More computational efficient than SGD

Targets(goals) which what we want to learn:

SARSA:

$$g(s,a) = R(s,a) + \gamma \cdot \hat{q}_\pi(S_{t+1}, A_{t+1}, w)$$

Expected SARSA:

$$g(s,a) = R(s,a) + \gamma \cdot \sum_a \pi(a|S_{t+1})\, \hat{q}_\pi(S_{t+1}, a, w)$$

Q-Learning:

$$g(s,a) = R(s,a) + \gamma \cdot \max_{\mathrm{a}} \hat{q}_\pi(S_{t+1}, a, w)$$

- **DQN**

Can be used with CNN but without pooling, because pooling makes the network computationally expensive and require time. Instead use strides

The DQN paper uses 4 stacks of layers, and this is to know the direction of the paddle, as it will be impossible to know solely from one picture.

An environment with incomplete information is called POMDP, partial observable markov decision process. Therefore stacking these layers makes us skip the POMDP

Instability problems:

1- Sequential correlated data which may hurt convergence and performance. Solved using experience replay
2- Instability of data distribution due to policy changes. Solved using target network, where you use another network for target with different weight parameters, then update it frequently either hard update or soft update.
3- Unstable gradients due to the high variations of Q-values and unknown scale of rewards. Solved using reward clipping from -1 to 1 for less peaky Q-values

- **Double Q-Learning**

They are two independent tables, we update them one after the other, and use one Q-function to train the other one

Objective for Q1:

$$\hat{Q}_1(s_t, a_t) = r_t + \gamma \cdot Q_2(s_{t+1}, \underset{a^*}{\operatorname{argmax}} Q_1(s_{t+1}, a^*))$$

Objective for Q2:

$$\hat{Q}_2(s_t, a_t) = r_t + \gamma \cdot Q_1(s_{t+1}, \underset{a^*}{\operatorname{argmax}} Q_2(s_{t+1}, a^*))$$

Week 5:

- **Policy gradient method**

As we know we have 2 approaches for solving RL problem.

Value based:

1- We learn state value v(s) or state-action value Q(s,a) and then infer policy given value function but you need perfect Q values for optimal policy.

Policy based:

1- Explicitly learn probability or deterministic policy and they adjust them to maximize the expected reward.

Objective of Policy gradient method:

To compute the expected reward : $J = E[R(s,a,r,s')]$

The agent gets born in some random state, takes one action and observes the reward and then a new session begins. To solve this expectation we should integrate over all possible states and actions because we have infinite continuous amount of options.

$$J = E[R(s,a,r,s')] = \int_s p(s) \int_a \pi_\theta(a|s)\, R(s,a)\, da\, ds$$

The first integral is the state visitation frequency (may depend on the policy if its complicated) but, the second integral is the probability of taking action a in the state which maybe the table of all possible probabilities of all action probabilities.

Approximation of expected reward:

We can approximate this expected reward to $J \approx \frac{1}{N}\sum_{i=0}^{N}\sum_{s,a\in z_i} R(s,a)$ where the first summation is the sampling of N sessions by following $\pi_\theta(a|s)$ and this is called **Monte-Carlo sampling**.

Computing gradient:

But how can we compute the gradient $\frac{dJ}{d\theta}$ when theta isn't in the formula after approximation?

Sol.: We can try some simple duct tape approach called Finite difference $\nabla J \approx \frac{J_{\theta+\varepsilon} - J_\theta}{\varepsilon}$, this will technically work but the policy changes slightly by the small value of epsilon.

More problems:

1- This Finite difference technique requires lots of episodes for computing $J_{\theta+\varepsilon}$ and same number for computing J_θ and
2- Then, we will suffer from the noise caused from sampling would be still much larger than the difference between the two policies specially if J is sufficiently small
3- If you use large J, the gradients will be useless for anything more complicated than linear model

Another method for computing gradients:

By analogy from $\nabla \log(\pi(s)) = \frac{1}{\pi(s)} \cdot \nabla\pi(s) \;\; \rightarrow \;\; \pi(s) \cdot \nabla \log(\pi(s)) = \nabla\pi(s)$, we are going to apply it to our J to make it more convenient.

$$J = \int_s p(s) \int_a \pi_\theta(a|s)\, R(s,a)\, da\, ds \;\; \rightarrow \;\; \int_s p(s) \int_a \pi_\theta(a|s)\, \nabla \log(\pi_\theta(a|s))\, R(s,a)\, da\, ds$$

The second formula allows us to approximate by sampling over states and actions and it is called the log-derivative trick.

Derivations : https://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.pdf

- **REINFORCE:**

Policy gradient : $\nabla J = E[\nabla \log \pi_\theta(a|s) \cdot Q(s,a)]$

Approximation by Monte-Carlo sampling: $\nabla J \approx \frac{1}{N}\sum_{i=0}^{N}\sum_{s,a\in z_i} \nabla \log \pi_\theta(a|s) \cdot Q(s,a)$

We do Monte-Carlo sampling because we only have access to trajectories of state, action, rewards, next states and next actions

Algorithm:

1- Start by defining the policy (Table, neural network or random forest)
2- Initialize weights with random
3- Sample N sessions under current policy
4- Evaluate policy gradient then update the parameters

Problems:

For easy states, the Q value will increase in easy states where you get a lot of points by the log gradient and we will get high gradients(weights) while the hard states will get low weights. For example translating a small sentence versus a transcript.

Advantage function:

This function computes how good is your algorithm doing, which is the difference between Q function and value function $A(s,a) = Q(s,a) - V(s)$

So we replace the Q with advantage function in our approximated policy gradient using baseline

https://papers.nips.cc/paper/4264-analysis-and-improvement-of-policy-gradient-estimation.pdf

https://arxiv.org/ftp/arxiv/papers/1301/1301.2315.pdf

- **Actor-critic**

1- Has a value based part where it tries to approximate the value function
2- Uses value function to learn policy function approximated same way
3- Learns the difference between current Q function and the average performance in this state

Advantage Actor-Critic:

Compute advantage function and replaces Q function in policy gradient method, so it becomes

$$\nabla J \approx \frac{1}{N} \sum_{i=0}^{N} \sum_{s,a \in z_i} \nabla \log \pi_\theta(a|s) \cdot A(s,a)$$

Where, $A(s,a) = r + \gamma \cdot V(s') - V(s)$, $Q(s,a) = r + \gamma \cdot V(s')$

Of course we need to get expectation over all states for computing the Q function but we can approximate it by taking one sample as in Q learning.

Now, how to get the Value function?

1- Train a network that has two outputs, first it has to learn a policy with units of number of actions and then use softmax to learn the probability distribution and the second it estimates a value function which is a single neuron
2- Update the policy by using the V function to provide better estimate of policy gradient which is the equation above

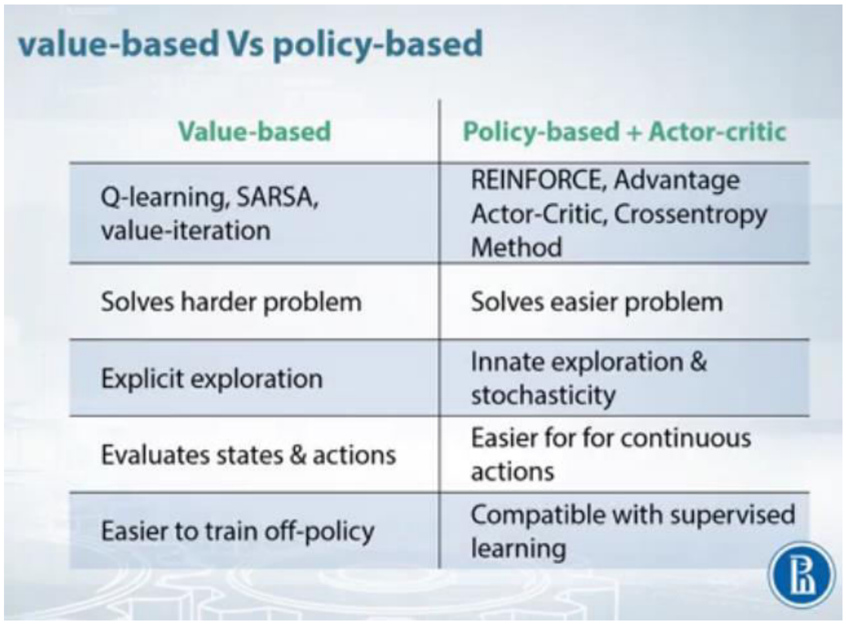
You have to refine your value function by computing MSE and reducing the error and this way we can converge the expectation of value function

$$\mathrm{L_{critic}} \approx \frac{1}{N} \sum_{i=0}^{N} \sum_{s,a \in z_i} (\mathrm{V}_\theta(s) - [r \cdot \gamma \cdot V(s')])^2$$

More on Advantage Actor-Critic:

We have two losses. The first loss is the policy based loss (policy gradient), the second one is you minimize temporal difference loss (value based).

We can consider the second loss to be less important. For a perfect critic but random actor, the critic will estimate how well the random agent perform, but if a good actor but random critic, actor still learns as good as the Reinforce



value-based Vs policy-based

Value-based	Policy-based + Actor-critic
Q-learning, SARSA, value-iteration	REINFORCE, Advantage Actor-Critic, Crossentropy Method
Solves harder problem	Solves easier problem
Explicit exploration	Innate exploration & stochasticity
Evaluates states & actions	Easier for for continuous actions
Easier to train off-policy	Compatible with supervised learning

The problem with actor critic that it is on-policy because you have to train it on the actions taken under its own policy and we can't use experience replay.

So, if we want to use experience replay, we have to overcome the problem of having independent sessions with non-identical distribution. There is a study says if you run parallel sessions, then you can consider it ideal. So, we can spawn multiple agents playing independent replicas of the environment with different weights, where they have to take actions by sampling from the policy independently. This way we make it on-policy.

A3C (Asynchronous Advantage actor-critic):

They train or more than one server, training each agent on a server then synchronize weights periodically to make them not diverge too far.

Famous for being used with LSTM, this means that the agent here uses some recurrent memory. Which asks your agent to memorize some of the observations, so that it can improve its policy based not only what it can see right now, but what it have seen previously. the asynchronous advantage actor-critic.

More details : https://arxiv.org/abs/1602.01783

https://deepmind.com/blog/impala-scalable-distributed-deeprl-dmlab-30/

https://blog.openai.com/evolution-strategies/

https://arxiv.org/pdf/1507.04296.pdf

Supervised with Reinforcement learning problem:

We can initialize the RL agent with heuristic experience, which will yield a good initial guess from which you can then go on reinforcement learning.

For example, if we have NN for machine translation, we can pre-train the NN with the existing data (human translation), then we can follow up with policy gradient to improve the advantage.

Supervised learning equation: $\nabla llh = E_{x,y\sim D}[\nabla \log P_\theta(y|x)]$

By changing it to reinforcement learning notation : $\nabla llh = E_{s,a\sim D}[\nabla \log \pi_\theta(a|s)]$

Policy gradient equation: $\nabla J = E_{s,a\sim D}[\nabla \log \pi(a|s) \cdot Q(s,a)]$

1- Initialize the policy at random
2- Take a few epochs to train the policy to maximize the probability of the heuristic
3- Order your algorithm to train for a few more iteration in the policy gradient mode to get better reward

The difference between supervised and RL equations is that the supervised is a reference expectation which samples from a fixed dataset (you don't let it do something and see what happens). Instead you train it with heuristic data.