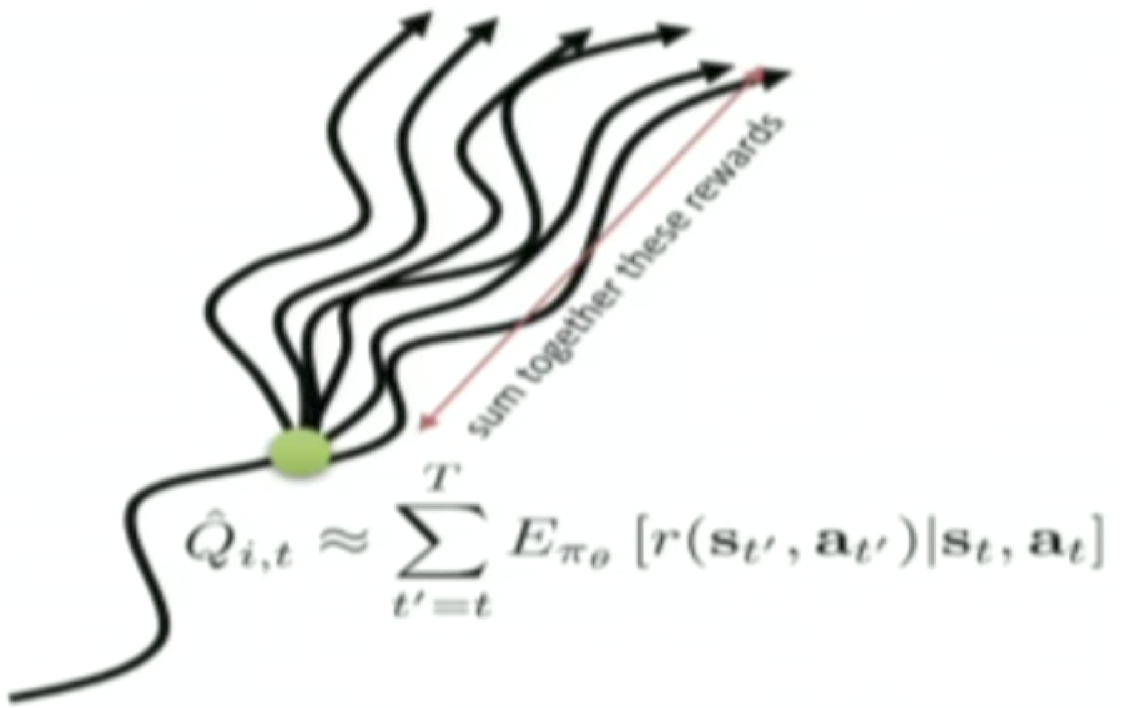
actor-critic

- the on-policy sampling methods we saw in the notes on policy gradient methods all involve summing up rewards from our sampled trajectories. Now we will see better ways to estimate that sum of future rewards term, called "reward-to-go".
- Q-function is a general idea of value functions but for state-action pairs instead of for just states.

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t}) \underbrace{\left(\sum_{t'=1}^{T} r(\mathbf{s}_{i,t'},\mathbf{a}_{i,t'})\right)}_{\substack{\text{"reward to go"} \\ \hat{Q}_{i,t}}}$$

- As discussed previously, one of the biggest challenges with policy gradient methods is getting a good estimate of the reward function. The expected reward to go is the average of all the following trajectories that arise from taking action a_t at state s_t:


sum together these rewards
$\hat{Q}_{i,t} \approx \sum_{t'=t}^{T} E_{\pi_\theta}\left[r(\mathbf{s}_{t'},\mathbf{a}_{t'})|\mathbf{s}_t,\mathbf{a}_t\right]$

- The goal is to estimate the reward to go. We could do this by drawing a single sample → but that's not a good estimate, it has high variance. The more we can reduce the variance, the better our gradient estimates, the faster we can converge.

- Just like we used baselines to reduce the variance, we can use the value function (expected value of Q functions over action distribution) to reduce variance estimates of the reward to go using q-functions by using $Q(s_{i,t}, a_{i,t}) - V(s_{i,t})$ instead of just $Q(s_{i,t}, a_{i,t})$. We call $Q(s_{i,t}, a_{i,t}) - V(s_{i,t})$ as $A(s_{i,t}, a_{i,t})$. A for advantage.
- To be clear, Q is the expectation from s by taking action a. V is the expectation from state s.
- We have said why $Q - V$ is a good number to use here but we have not said how to get this number yet. $Q - V$, if we can get it perfectly, is like the perfect term to go here, not just an approximation. The better we can estimate this advantage, the better our gradient estimate will be.
- Now, we will use the same sampling methods as we did in vanilla policy gradient methods and we will do a similar gradient update step. The only thing left to do is to find ways to get good estimates of both Q^π, V^π or A^π directly.
- Which of these do we want to learn a function for? One thing to note is that higher the number of variables, the more the variance in sampling. Ideally we'd like to sample V's instead of Q's for this reason since Q's depend both on state and action, whereas V only on state. Because of the following approximation math, we can:

$$Q^\pi(\mathbf{s}_t, \mathbf{a}_t) = r(\mathbf{s}_t, \mathbf{a}_t) + \sum_{t'=t+1}^{T} E_{\pi_\theta}\left[r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) | \mathbf{s}_t, \mathbf{a}_t\right]$$

$$Q^\pi(\mathbf{s}_t, \mathbf{a}_t) = r(\mathbf{s}_t, \mathbf{a}_t) + E_{\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t)}[V^\pi(\mathbf{s}_{t+1})]$$

$$Q^\pi(\mathbf{s}_t, \mathbf{a}_t) \approx r(\mathbf{s}_t, \mathbf{a}_t) + V^\pi(\mathbf{s}_{t+1})$$

This is the approximation step where we sample one of the states instead of expectation over states.

$$Q^\pi(\mathbf{s}_t, \mathbf{a}_t) \approx r(\mathbf{s}_t, \mathbf{a}_t) + V^\pi(\mathbf{s}_{t+1})$$
$$A^\pi(\mathbf{s}_t, \mathbf{a}_t) \approx r(\mathbf{s}_t, \mathbf{a}_t) + V^\pi(\mathbf{s}_{t+1}) - V^\pi(\mathbf{s}_t)$$

The last step is to get advantages once we've learnt the value functions.

- Note that now, we only need to learn V's and we don't need to learn $Q's$.
- We will use a neural network to learn the value function. Input to network is state, output is value of that state.
- Note that the neural network will need to learn the value function for a given policy. As our policy is updated the value functions keep changing.
- The process of finding the value functions of different states for a given policy is called policy evaluation, cuz we're just evaluating how good the policy is for different states.
- At each policy value, we will sample a bunch of trajectories and get a bunch of (state, reward to go) pairs and use these as training data for a supervised regression neural network. The point of doing training instead of plain sampling like we did in vanilla policy gradient is that the neural network will hopefully learn patterns in the data and give us a lower variance estimate of the true value of a state. Again, note that this neural network as described currently will need to be retrained after each gradient update step because it depends on policy!

- For our training data, in addition to samples from our monte carlo walks, we can also add terms of the form

$$\left\{\left(\mathbf{s}_{i,t}, \underbrace{r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) + \hat{V}^{\pi}_{\phi}(\mathbf{s}_{i,t+1})}_{y_{i,t}}\right)\right\}$$

- In statistics, bootstrapping is any test or metric that relies on random sampling with replacement. Bootstrapping allows assigning measures of accuracy (defined in terms of bias, variance, confidence intervals, prediction error or some other such measure) to sample estimates. The above defined technique of estimating value functions is an instance of bootstrapping.
- Above we did one real reward step and then applied value function. in general, we could do k real samples and then call the value function.
- to start off, we set the neural net to input some very small random number by suitable initialization. if we're using 1-step sampling, one round of training with examples of the form (state, reward + remaining value) will propagate rewards 1-step deep. to train the network for the initial policy we need to train this about horizon number of times or if we discount factors, that will influence this decision.
- one trick to encourage exploration is to set high rewards everywhere so that the policy is lead to undiscovered places and visits them to discover their true value.

batch actor-critic algorithm:

```
1. sample $\{\mathbf{s}_i, \mathbf{a}_i\}$ from $\pi_\theta(\mathbf{a}|\mathbf{s})$ (run it on the robot)
2. fit $\hat{V}^{\pi}_{\phi}(\mathbf{s})$ to sampled reward sums
3. evaluate $\hat{A}^{\pi}(\mathbf{s}_i, \mathbf{a}_i) = r(\mathbf{s}_i, \mathbf{a}_i) + \hat{V}^{\pi}_{\phi}(\mathbf{s}'_i) - \hat{V}^{\pi}_{\phi}(\mathbf{s}_i)$
4. $\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta(\mathbf{a}_i|\mathbf{s}_i)\hat{A}^{\pi}(\mathbf{s}_i, \mathbf{a}_i)$
5. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
```

$$V^{\pi}(\mathbf{s}_{i,t}) = \sum_{t'=t}^{T} E_{\pi_\theta}\left[r(\mathbf{s}_{t'}, \mathbf{a}_{t'})|\mathbf{s}_{i,t}\right]$$

- Note that step #2 (fitting the value function) could involve a h/k iteration loop where h is horizon and k is number of samples before you draw from the value function. If $k = 1$, then we might train the network h times (like applying bellman backup h times)
- the way we have described it here, if we do training the value function will increase in an unbounded fashion. in practice, we will use discounts to overcome this problem.
- we have two ways to do discounting:
- the first one is what's used in practice

option 1: $\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})\left(\sum_{t'=t}^{T}\gamma^{t'-t}r(\mathbf{s}_{i,t'},\mathbf{a}_{i,t'})\right)$

not the same!

option 2: $\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})\right)\left(\sum_{t=1}^{T}\gamma^{t-1}r(\mathbf{s}_{i,t'},\mathbf{a}_{i,t'})\right)$

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})\left(\sum_{t'=t}^{T}\gamma^{t'}\textcircled{1}(\mathbf{s}_{i,t'},\mathbf{a}_{i,t'})\right)$$

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\gamma^{t-1}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})\left(\sum_{t'=t}^{T}\gamma^{t'-t}r(\mathbf{s}_{i,t'},\mathbf{a}_{i,t'})\right)$$

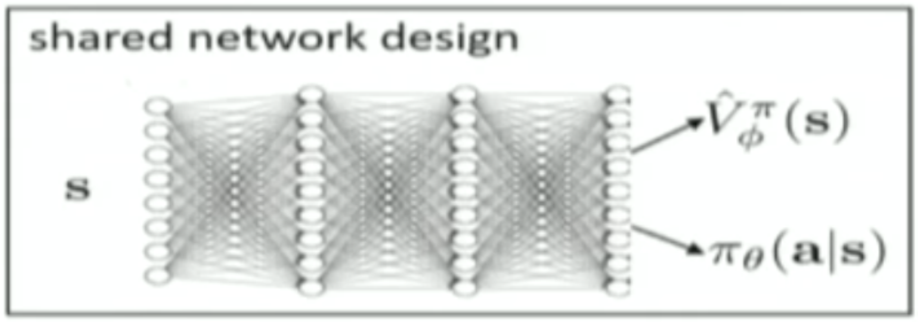
note option 2 is almost the same but the gradients are also discounted

in practice we often use option 1.

- the reason we call these algorithms actor-critic algorithms is because we have an actor (the policy network) and a critic (the policy evaluation or value function).
- we can also make up an online version of the above policy gradient using value functions algorithm:

```
online actor-critic algorithm:
    1. take action $\mathbf{a} \sim \pi_\theta(\mathbf{a}|\mathbf{s})$, get $(\mathbf{s}, \mathbf{a}, \mathbf{s}', r)$
    2. update $\hat{V}^\pi_\phi$ using target $r + \gamma\hat{V}^\pi_\phi(\mathbf{s}')$
    3. evaluate $\hat{A}^\pi(\mathbf{s},\mathbf{a}) = r(\mathbf{s},\mathbf{a}) + \gamma\hat{V}^\pi_\phi(\mathbf{s}') - \hat{V}^\pi_\phi(\mathbf{s})$
    4. $\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(\mathbf{a}|\mathbf{s})\hat{A}^\pi(\mathbf{s},\mathbf{a})$
    5. $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$
```

- since in the above algorithm we only sample once before we update, we are not training on much data so our value estimator has a ton of variance. to address this, we can have many workers in a distributed setting sample trajectories so that we can use a batch of a few samples for each update instead of just 1 sample. this synchronization of parallel steps, computing gradients from n samples and then updating is called synchronized parallel actor critic.
- the asynchronized version of this simply samples trajectories and updates gradients without a synchronizing step. I'm not sure if this is a principled approach but in practice in tends to work → which is what matters!
- until now, we have talked about using separate networks with separate parameters for the actor and the critic.
- if we are in a high dimensional space, like if we use images as input, it would be really nice to share some fundamental features between the actor and the critic. we could use something like:


shared network design
$\mathbf{s}$
$\hat{V}^\pi_\phi(\mathbf{s})$
$\pi_\theta(\mathbf{a}|\mathbf{s})$

the shared network can be nice because it can have shared parameters that are good lower level feature representations, but it can be harder to train and get it to work, but if you can get it to work with the right architecture and hyper parameters, it can learn faster.

- Now we have seen two variants of policy gradients. the first we learnt about samples using monte carlo (random trajectories) and the second learns a value function for each policy using a k-step backup. the first approach in unbiased on expectation but has a lot of variance. the second has much lesser variance but is likely quite biased (especially when it just starts off).

Actor-critic: $\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})\left(r(\mathbf{s}_{i,t},\mathbf{a}_{i,t}) + \gamma\hat{V}_\phi^\pi(\mathbf{s}_{i,t+1}) - \hat{V}_\phi^\pi(\mathbf{s}_{i,t})\right)$

+ lower variance (due to critic)

- not unbiased (if the critic is not perfect)

Policy gradient: $\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})\left(\left(\sum_{t'=t}^{T}\gamma^{t'-t}r(\mathbf{s}_{i,t'},\mathbf{a}_{i,t'})\right) - b\right)$

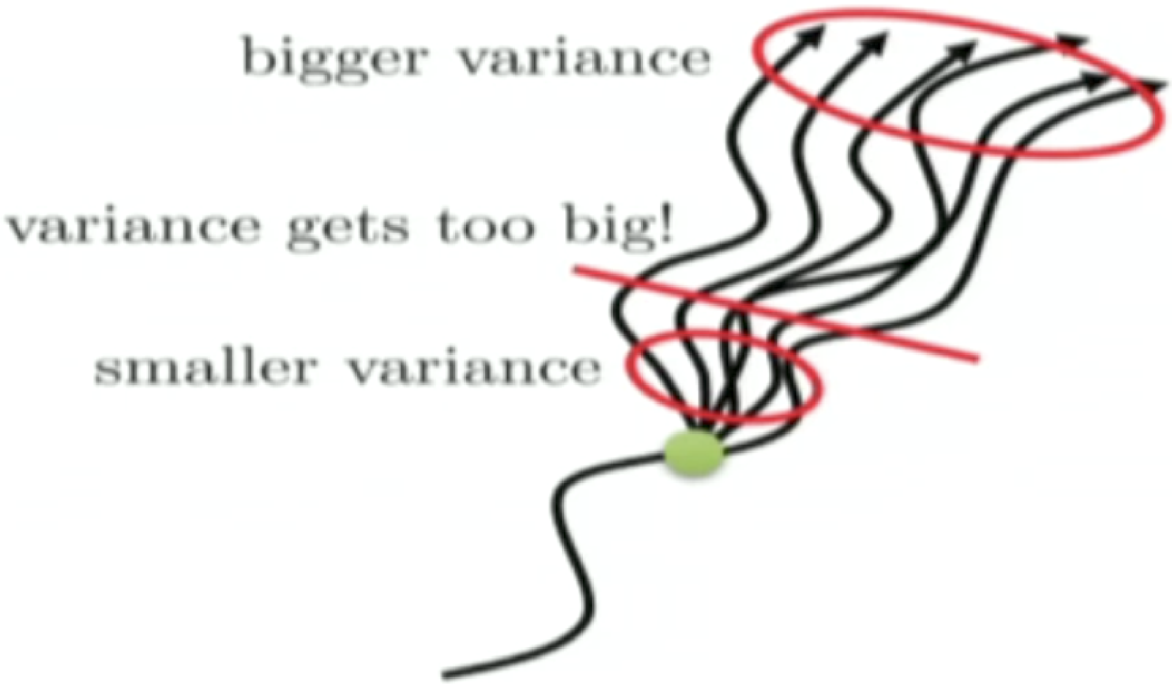
+ no bias

- higher variance (because single-sample estimate)

- a way to get the best of both worlds is to do something like: (low bias and variance)

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})\left(\left(\sum_{t'=t}^{T}\gamma^{t'-t}r(\mathbf{s}_{i,t'},\mathbf{a}_{i,t'})\right) - \hat{V}_\phi^\pi(\mathbf{s}_{i,t})\right)$$

- Earlier we saw that we can use a k-step backup (also called k-step return). This k is kind of a direct way to control bias/variance tradeoff. The higher the k the more the variance and lesser the bias. A justification for that is this diagram:



- Especially earlier, when the critic network is randomly initialized is most definitely biased, we can do higher values of k and then lower k as we go.
- A3C → asynchronous advantage actor critic.