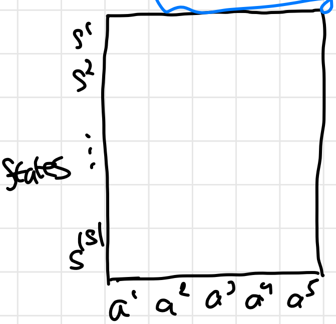
11/14/2023: Reinforcement Learning Contd.

Modeling



Data

At every timestep:
Start at some state s
Take an action a
Get a reward r
Transition to new state s'
This is "1 training example" for Q-Learning

Tabular Q-Learning:
each $\hat{Q}(s,a)$ is 1 parameter to learn
Total # of params is

$$\boxed{\#\text{States} \times \#\text{actions}}$$

← Can be very very large

Q-Learning Update Rule:
Input: (s, a, r, s')
Update

$$\underbrace{\hat{Q}(s,a)}_{\substack{\text{our guess of} \\ \text{Q-value for action} \\ \text{we just took}}} \leftarrow (1-\eta)\underbrace{\hat{Q}(s,a)}_{\text{old guess}} + \eta\Big(\underbrace{\underbrace{r}_{\substack{\text{immediate} \\ \text{reward}}} + \gamma \underbrace{\hat{V}(s')}_{\substack{\text{anticipated} \\ \text{future} \\ \text{reward}}}}_{\text{Estimate of total future reward}}\Big)$$

η: learning rate (e.g. 0.1)

$$\text{where } \hat{V}(s) = \begin{cases} \max\limits_{a \in Actions(s)} \hat{Q}(s,a) & \text{if NOT IsEnd}(s) \\ 0 & \text{else} \end{cases}$$

↑ Our guess of how good a state is based on $\hat{Q}$

One more consideration: How to choose actions during training?

Obvious answer (wrong):
At state s, choose $a = \arg\max_{a'} \hat{Q}(s, a')$

- Optimal if $\hat{Q}$ values are accurate
- Bad idea early in training!

Suppose we do action a in state s once, receive large reward.

$\Rightarrow$ $\hat{Q}(s,a)$ will update to be large (larger than other actions at same state)

$\Rightarrow$ Naive policy always chooses a in state s forever

All exploitation, no exploration

use knowledge we've learned

try different things to see what's best

__Simple Solution: ε-greedy__

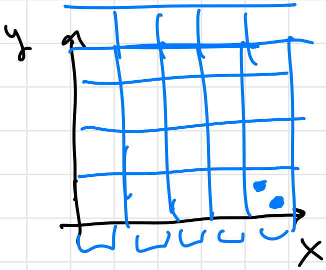
At each timestep: At state s
- with probability $1-\varepsilon$, choose $\arg\max_a \hat{Q}(s,a)$ (Exploitation)
- with probability ε, choose random action (exploration)

Usually choose small but nonzero ε during training (eg $\varepsilon = 0.1$)

At test time, use $\varepsilon = 0$

How to deal with very large state spaces?

Option #1: Discretize the state
- States might be continuous (eg. location)
- Divide a continuous dimension into buckets



Discretize xy plane to
5x5 grid = 25 states

$\# \text{states} = (\# \text{buckets})^{\text{dimensions}}$ Bad in high dimensions

Option #2: Q-learning with linear function approximation

Idea: Q-learning is kind of like regression:
input = (s,a), output = Q-value

So let's learn a linear model:
1. Need feature function $\phi(s,a) \in \mathbb{R}^d$
2. Learn parameter vector $w \in \mathbb{R}^d$
   to predict $\hat{Q}(s,a) = w^T \phi(s,a)$

How to learn w? Revisit Q-learning update rule

For tabular Q-L: $\hat{Q}(s,a) \leftarrow (1-\eta)\hat{Q}(s,a) + \eta(r + \gamma \hat{V}(s'))$

$$= \hat{Q}(s,a) + \eta(\underbrace{r + \gamma\hat{V}(s')}_{\text{"target"}} - \underbrace{\hat{Q}(s,a)}_{\text{current prediction}})$$

Review: linear regression:
$\nabla_w (w^T x - y)^2$
$= 2(w^T x - y) \cdot x$

$$= \hat{Q}(s,a) - \eta\underbrace{(\hat{Q}(s,a) - r - \gamma\hat{V}(s'))}_{\text{Basically the linear regression gradient}}$$

So, for Q-learning w/ function approx:

minimize squared error between

$$\underbrace{\hat{Q}(s,a)}_{\text{prediction}} \quad \text{and} \quad \underbrace{r+\gamma \hat{V}(s')}_{\text{"target"}}$$

$$\text{loss}(w) = \frac{1}{2}\Big(r+\gamma\hat{V}(s') - \underbrace{w^T\phi(s,a)}_{=\hat{Q}(s,a)}\Big)^2$$

Gradient:

$$\nabla_w \text{loss}(w) = \cancel{\frac{1}{2}} \cdot \cancel{2} \cdot \left(r+\gamma\hat{V}(s') - w^T\phi(s,a)\right) \cdot -\phi(s,a)$$

Update Rule:

$$w \leftarrow w - \eta \nabla_w \text{loss}(w)$$
$$= w + \eta\left(r+\gamma\hat{V}(s') - w^T\phi(s,a)\right)\phi(s,a)$$

Option 3: Deep Q Network (DQN)

Idea: $\hat{Q}(s,a)$ is a neural network that maps (s,a) to estimate of $Q_{opt}(s,a)$

Let θ be parameters of network:

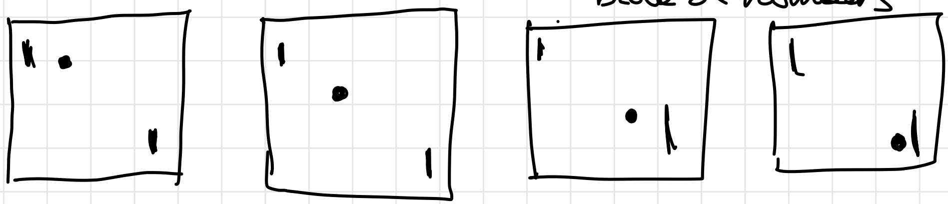
$$\text{loss}(\theta) = \frac{1}{2}\Big(\underbrace{r+\gamma\hat{V}(s')}_{\text{"target"}} - \underbrace{\hat{Q}_\theta(s,a)}_{\text{prediction}}\Big)^2$$

$$\nabla_\theta \text{loss}(\theta) = \cancel{\frac{1}{2}} \cdot \cancel{2} \cdot \underbrace{\left(r+\gamma\hat{V}(s') - \hat{Q}_\theta(s,a)\right)}_{\text{compute directly}} \cdot -\underbrace{\nabla_\theta \hat{Q}_\theta(s,a)}_{\text{compute by backpropagation}}$$

Run gradient descent to update θ

Example DQN to play Pong

Represent state of game with last k frames
- Each frame is 84×84 image
- Set $k = 4$ → Input to DQN is $84 \times 84 \times 4$ block of numbers



↓ Feed to CNN, generates a vector $u(s)$

Has lots of parameters

$u(s)$

3 actions: Learn 1 vector per action

up: w_{up}

down: w_{down}

stay: w_{stay}

parameters

Predictions:

$\hat{Q}(s, up) = w_{up}^T u(s)$

$\hat{Q}(s, down) = w_{down}^T u(s)$

$\hat{Q}(s, stay) = w_{stay}^T u(s)$

Taxonomy of RL methods

Model-Based RL
Learns transitions + Rewards

Model-Free RL
Dont try to directly learn transition & reward probabilities

Q-Learning
Learn $\hat{Q}(s,a)$

Infer good policy by maximizing $\hat{Q}(s,a)$

Policy Gradient
Directly learn a policy

≈ classifier to predict action given state