Deep Q-learning. → perform action #1.

First environment: first state

Q-learning as function approximator

while not done:

Deep Q-Learning (DQN) Review

- Basic Q-Learning:

```
s = env.reset()
while not done:
  a = argmax(Q[s, :]) # can also use epsilon-greedy
  s', r, done = env.step(a)
  td_err = (r + gamma * max { Q[s', :] } - Q[s, a])^2
  params = params - learning_rate * grad(td_err, params)
  s = s'
```

env. reset. State 1

action

action

reward

give

given

state

action

all actions

TD-error. Semi gradient - target | ignore target.

true target → Estimate

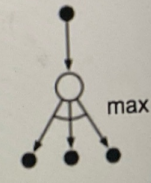
Diff Q once

Details

Estimated target: (r + gamma * max { Q[s', :] }

Prediction: Q[s, a])^2

- It's not true gradient descent (we call it "semi-gradient")
- In supervised learning, we normally have (target - prediction)^2
- In Q-Learning, the target depends on the prediction!
- The target does not tell us the *true* Q value
- Even though Q shows up twice (once in target, once in prediction) we only differentiate wrt prediction
- We pretend the target is fixed
- Q-learning is off-policy
- The target is still $r + \gamma Q(s',a')$ where $a' = \text{argmax}\{Q(s',:)\}$
- Even if we don't take the action a' in the next step



Q-learning

semi-gradient

Target depends on the prediction

Estimate based on own model / model parameters once.

off-policy learning algorithm.

diff. once with Q

Q-learning - Off policy learning.

$$Q = r + \gamma Q(s', a')$$

target

$a' = \text{argmax}(Q(s', :))$

same policy regardless of action I take.

→ fixed feature extract

Q learning function approximator

N.N unstable → cannot work with fixed feature extraction with linear model on top.

Deep Q-learning tricks

N.N → deep learning

(s, a, r, s', done) - tuple | Experienced replay memory buffer

input ⇔ target pair

Target: $r + \gamma * (1 - done) * \max(Q(s', :))$

Pred: $Q(s, a)$

Why $(1 - done) = 1$, then target = r

$(1 - 1) = 0$,

terminal state $s' = 0$ = $\sum_j$ rewards future

$Q(s, a) \rightarrow 0$

$s' = 0$

Q learning tricks

replay buffer: 1 million | most recent sample correlated.

perform sampling on (s, a, r, s', done) or the batch from the replay buffer

call (input (tuple) → grad (s, a, r, s', done))

experience replay buffer strategy

Trick 1: Replay buffer of size 1 million

greedy | s, a, r, s', done | most recent but (sample from batch sequential states episodes are correlated)

Experience replay pseudocode

```
s = env.reset()
while not done:
  a = argmax(Q[s, :]) # can also use epsilon-greedy
  s', r, done = env.step(a)
  replay_buffer.add((s, a, r, s', done))
  inputs, targets = sample data from replay_buffer
  params = params - lr * grad((targets - predictions)^2, params)
  s = s'
```

optimal action (a)

action perform → add replay buffer tuple to s'

Sample data: 1 million past samples to choose from

Trick 2: Use separate target network

$$Old: \; r + \gamma(1-d)\max_{a'}\{Q(s',a')\}$$

$$New: \; r + \gamma(1-d)\max_{a'}\{Q_{targ}(s',a')\}$$

Target is not from target (semi-gradient).

$$Old: \; r + \gamma(1-d)\max_{a'}\{Q(s',a')\}$$

$$New: \; r + \gamma(1-d)\max_{a'}\{Q_{targ}(s',a')\}$$

10k steps

copy

Pseudocode

```
s = env.reset()
step = 0
while not done:
  a = argmax(Q[s, :]) # can also use epsilon-greedy
  s', r, done = env.step(a)
  replay_buffer.add((s, a, r, s', done))
  inputs, targets = sample data from replay_buffer
  params = params - lr * grad((targets - predictions)^2, params)
  s = s'
  step++
  if step % 10000 == 0:
    copy params to target network
```

if steps are multiples of 10000, then copy main params to target network.

Replay buffer
Target network
} Issues / Stability → } How to use CNN with Q-learning

D.D.P.G : Deep deterministic policy Gradient

Q-learning + Policy gradients

actor Policy

TD

Action

State

Value function

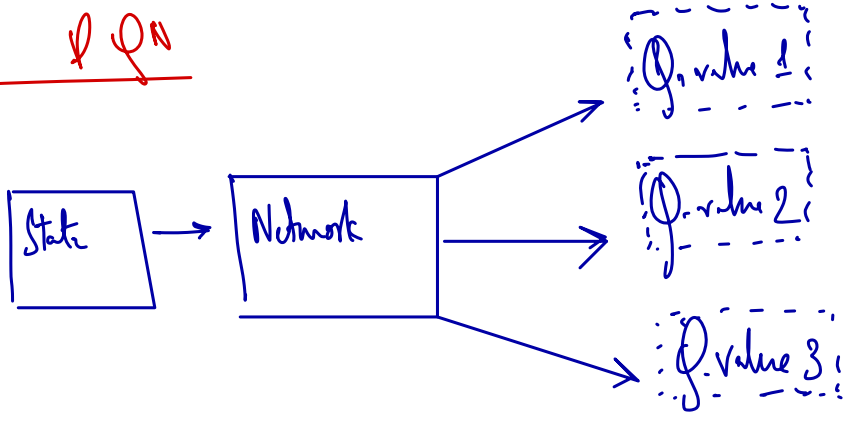
reward.

Environment

Cannot handle continuous action spaces

Limitation of DQN

$Q(s, a)$



Continuous scale
Continuous vector

Binning: $a_0: [1, 0)$
$a_1: [0, 1)$
$a_2: [1, 8]$

$[0.9, 1.1]$ Similar but Binning

Curse of dimensionality

- $[-1, 0, +1]$ → 7 degrees of freedom.
- not expensive anm.

$|A| = 3^7 = 2187$ >> neural network output

Policy as Probability

$\pi(a|s)$: softmax $(W^T s)$ → sample from distrib?

policy parameters : W (shape of W

State, action, Rewards Theoretically nice rewards nice.

- Discrete
- Continuous

0, 1, ... k-1

0 = Pog
1 = Col
2 : nowt

|s| states : 0 ... |s-1|

Target continuous
→ tensors with > 1 dimension

Current state — former state

Current action $\pi(a|s)$: softmax $(W^T s)$

maximize rewards in future

dictionary mapping

Observe?

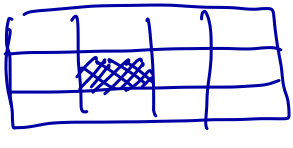
Sub policies (1) (2) (3)

master policy → Action

State (s) vectors → Action (a)

Actions (↑ ↓ ← →)

policy [(0,0) :



Indexing array faster than indexing dictionary.

Policy as Prob.

rand. no. $\rightarrow (\varepsilon) < 0.1$

action from $[A]$

Eg.

a : fixedpolicy(s)

return

Greedy Epsilon algo :

$\pi(a|s)$: softmax $(W^T s)$

Continuous state space : 's' is a D-dimensional vector

W (

$(W^T \cdot S)$ gives rise to a probability of state function. | softmax function.

convert categorical to action space

$\pi(a|s)$ probability for each action space. $\pi(a|s)$

which action to take from the env. | argmax $(\pi(a|s))$

Markov prediction

The guide known for jumping over the lazy POG

Only predict what happens before.

P

Markov Decision Process (MDP)

$p(S_{t+1}, R_{t+1} \mid S_t, A_t)$

probability of getting to state S_{t+1} & get reward (R) at $t+1$ when current state is State t & taking action $A(t)$

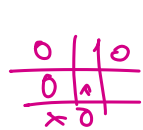
$p(s', r \mid s, a)$ State transition probability.

Deterministic: $R(s, a, s')$ or $R(s)$, $R(s')$ Reward

$p(s' \mid s, a)$

Useful of state transition probability:

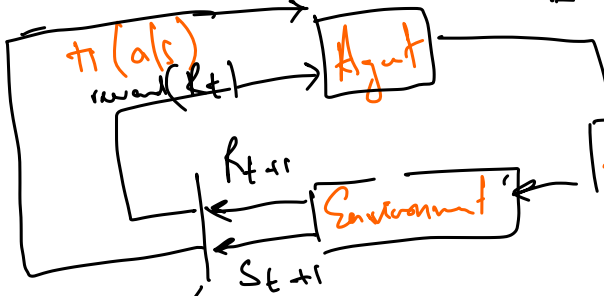
- Build on Q-learning
- Laws of physics: Deterministic? (No)
- nature law of physics: uncertainty
- Learned Environment dynamics



next step probability

(Chaos theory)

1. reach state takes action
2. Environment updated from this next step.
3. Bring agent to next step & next reward.
4. Agent given by prob. $\pi(a \mid s)$ given s.



This part is hazy?

→ Environment State transition probability.

→ well defined problem. ⟶ find solution.

maximize reward.

Agent: **Goal: $\mathbb{E}$ (future rewards)** agent: wants to max

Diff: → next step
→ best/bad rewards

→ plan future max.
→ actions to maximize the reward.
→ Immediate gratification | long-term planning.
→ miss T.V show.

$\mathbb{E}$ (future rewards) = Return

$$G(t) = \sum_{z=1}^{T-t} R(t+1) : \quad R(t+1) + R(t+2) + R(t+3) \dots R(T)$$

Sum upto the terminal state.

Infinite Horizon MDP:

Discounting: used for Return episodic ⟶ ∞ infinite.

Main thesis. Care about rewards now | than future.

$$G(t) = \sum_{z=1}^{T-t} \gamma^{z-1} R(t+z) = R(t+1) + \gamma R(t+2) + \gamma^2 R(t+3) \dots \gamma^{T-t-1} R(T).$$

G_t : reward at $(t+1)$ $+ \gamma$ G_{t+1} recursively.

1. reward 2. return

Value functions & Bellman eqn.

Expressing problems as eqn?

Expected value : mean $= (\mu)$

Probability

bern $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ $(\mu \pm 2\sigma)$

Expected value $= 0.5$ $\neq$ value ("the average of the value you can expect")

Discrete dist. : $E(x) : \sum_{k=-\infty}^{+\infty} p(x=k)\,k$

Continuous dist. : $E(x) : \int_{-\infty}^{\infty} x\, p(x)\, dx$

Expected value is weighted sum of all the possible values of rand. variable where weights = prob of values.

Expected value of Biased coin: $p=(0.6)$ heads.

$E(x) = 0.6 \times 1 + 0.4 \times 0 =$ **0.6** Expected value.

Environment dynamics → probabilistic

policy

maximize Expected sum of future rewards

Expected sum of future rewards : Value function

S

$V(s) = E(G_t \mid S_t = s)$

since $G(t) = R(t+1) + \gamma G(t+1)$ — remaining

conditional : we arrive at state s at time t

$$V(s) = E\left[R_{t+1} + \gamma V(s') \mid S_t = s\right]$$

Problem to solve:

Continue building up problem..

(Bellman equation)

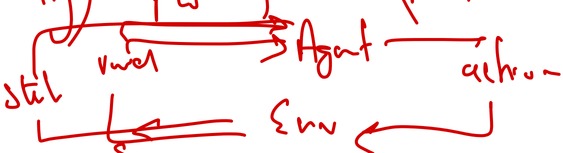
$$V(s) = \sum_a \sum_{s'} \sum_r \pi(a|s)\, p(s', r|s, a) \left\{ r + \gamma V(s') \right\}$$

- Sum over all possible actions (a), possible next state (s'), possible reward r).
- probability performing action a, given we are in state s
- probability of landing in state (s') & getting reward r

Action → Step → reward → reward value.

× future reward

distinct physically: $\pi(a|s)$, $p(s' \ldots)$ — agent (action, reward), Environment



Problem & Solution

multiple policies → Good
→ Bad.
→ value function

$V_\pi(s)$ = value func: given policy = prediction problem

$\pi(a|s)$ → known (policy). func^n.

$p(s', r | s, a)$ state transition prob | Gridworld. environment.

---

learn meaning : Reinforcement learning problem.

1. Prediction problem.
   given π, find $V(s)$

2. Control problem.
   find π^* which gives max $V(s)$

Greedy distribution

$\Rightarrow a^* = \underset{a}{\text{argmax}}\, Q^*(s,a)$

3 Actions

$Q(s,a_1) \rightarrow$ Q initialized to
$Q(s,a_2) \rightarrow$ $Q(s,a_1) = 0$
$Q(s,a_3) \rightarrow$ $Q(s,a_2) = 0$
$Q(s,a_3) = 1$

Explore - Exploit Dilemma $\rightarrow$ Exploited in bad way

$\rightarrow$ Slot machine best chance of winning

Collect data $\rightarrow \left(\frac{\#\text{win}}{\#\text{play}}\right)$ win rate.

$\frac{0.25 \times 5000}{1250}$

Samples

Samples	Accuracy
10000	
100	
10	

$5{,}000 \times 1 = \$25000$

$\rightarrow$ which slot machine is the best

Epsilon Greedy problem

random no. $\alpha \leq \varepsilon \leq 1$

ε = random value
Q-table.

1) $a = \text{argmax}$ Standard / policy
2) Choose random no. $< \varepsilon$ if
if not, random action
else: $a = \text{argmax}(Q(s,a))$

Greedy Epsilon

Q-learning algorithm

MDP - reward agent, action, environment - State $s \in S$
- take action $a \in A$.

Predict :

Control :

Limitation : $(\Sigma \text{returns})$ not known before episodes / rewards.

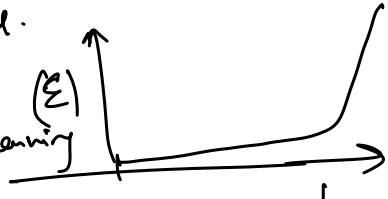
$(\Sigma \text{returns}) =$ Reward.

- Long time for episodes.
- MC sub optimal

Temporal difference

$\Rightarrow G(t) = R(t+1) + \gamma G(t+1)$

return = reward



diminishing returns

Temporal diff $\longrightarrow$ Approx M. Carlo method $\rightarrow$ Expected value problem.

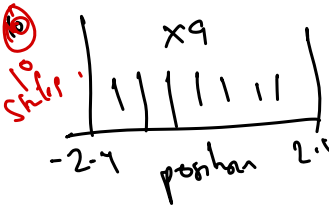
$$Q(s,a) = Q(s,a) + \eta\left(g - Q(s,a)\right) \leftarrow$$

$$J = \left[g - V(s)\right]^2$$

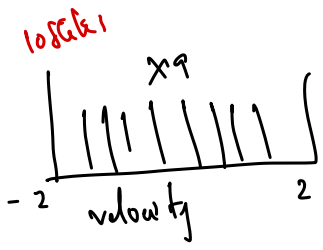
true target & sample (g)

$V(s)$ predicted value for state s'

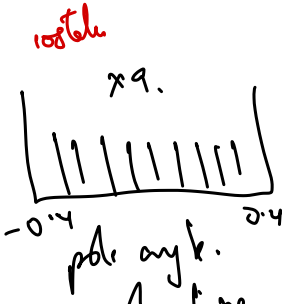
Models from Binning.



Cart Bins



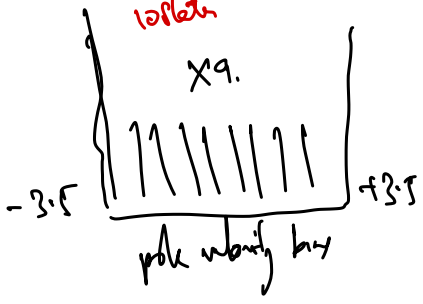
Cart Bins



Feature transformer

transform one obs → at one time.

- not ideal, max probability distribution not addressed.



Model:

conv. —

states — 10*4 — 10 bins for state variables (4)

actions — 2 — num states, x num. actions

Q = uniform

10 * 10

[actions = 2 (left/right)

Build state = [cart pos, cart vel, pole angle, pole vel]

1 2 3 4

4 states

actions

space]

```python
class Model:
  def __init__(self, env, feature_transformer):
    self.env = env
    self.feature_transformer = feature_transformer

    num_states = 10**env.observation_space.shape[0]
    num_actions = env.action_space.n
    self.Q = np.random.uniform(low=-1, high=1, size=(num_states, num_actions))

  def predict(self, s):
    x = self.feature_transformer.transform(s)
    return self.Q[x]

  def update(self, s, a, G):
    x = self.feature_transformer.transform(s)
    self.Q[x,a] += 10e-3*(G - self.Q[x,a])

  def sample_action(self, s, eps):
    if np.random.random() < eps:
      return self.env.action_space.sample()
    else:
      p = self.predict(s)
      return np.argmax(p)
```

Next we have a sample action function.

e 136, Column 33 Spaces: 2

```python
def play_one(model, eps, gamma):
  observation = env.reset()
  done = False
  totalreward = 0
  iters = 0
  while not done and iters < 10000:
    action = model.sample_action(observation, eps)
    prev_observation = observation
    observation, reward, done, info = env.step(action)

    totalreward += reward

    if done and iters < 199:
      reward = -300

    # update the model
    G = reward + gamma*np.max(model.predict(observation))
    model.update(prev_observation, action, G)

    iters += 1

  return totalreward
```

Random Search in parameter space for linear model.

→ >0 action 1
<0 action 2

→ Code
Code:
Code:

Random weights
new_weights = random.
if avg. episode length > Best
weights = new_weights

How good do my best weight do
random perform

random_search.py

```
1  from __future__ import print_function, division
2  from builtins import range
3  # Note: you may need to update your version of future
4  # sudo pip install -U future
5
6  import gym
7  import numpy as np
8  import matplotlib.pyplot as plt
9
10
11 def get_action(s, w):
12   return 1 if s.dot(w) > 0 else 0
13
14
15 def play_one_episode(env, params):
16   observation = env.reset()
17   done = False
18   t = 0
19
20   while not done and t < 10000:
21     # env.render()
22     t += 1
23     action = get_action(observation, params)
24     observation, reward, done, info = env.step(action)
25     if done:
```

If dot product > 0
then val = 1
else val = 0

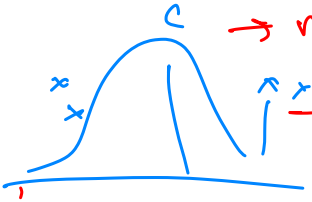
R·B·F networks – Radial Basis func°

1 – layer neural network. (non-linear func° approx).

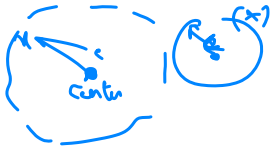
learning hidden units feature rep of input.

Single layer NN with hidden RBF activation functions

→ How many c's?
→ no. of c's = =
→ no. of hidden units in RBF network



$$\phi(x) = e^{\left(-\frac{\|x-c\|^2}{\sigma^2}\right)}$$



max = 1 when

no. of exemplars = no. of hidden units in N. Network.

hyperparameters to be tuned.

~~Skit~~ Sci-kit learn. usage:

```
from sklearn.kernel_approx import RBFSampler.
sampler = RBF sampler()
sampler.fit(raw_data)
features = sampler.transform(raw_data).
```

from sk_learn.pipeline import feature union.

SGD Regressor.
partial fit (X, y) -> one step of grad. descent
predict (X)
partial fit → starting values / initialize?

SVR:

# of SVM's = no of points
don't scale.

ex: of examples :: hyperparameters (Scikit learn implements)

* call partial fit SGD regressor target = op
* optimistic initial values method
instead of linear model.
use X ← transform(s)
for every model, Q(s) represents different actions.
3 actions Q(s) - Left
Q(s) - Right
Q(s) - nothing

Policy gradient — Modelless

Optimal policy π^* → (find π^*)

1. find value given current policy
2. improve policy given the current value (Argmax).

→ policy iteration

parameterized policy : score for action $a(j)$

$\text{score}_j = f(a_j, s, \theta) = \phi(s)^T \theta_j$ if (linear).

Softmax : $\pi(a_j|s) = \frac{\exp(\text{score}_j)}{\sum_j \exp(\text{score}_j)}$

performance.

initial state (s_0)

maximize the performance over the state s_0 : $V_\pi(s_0) = \eta(\theta_p)$ = ↓ performance eta.

parameterize (1) Policy (2) value function

1- Policy : $\pi(a|s, \theta_p)$ = $f(s, \theta_p)$ (action over state s, given policy θ_p)

policy approximation

2- Value func? = $V_\pi(s) = f(s, \theta_v)$

val fn | approximation

θ_p = parameters with 'p'

(θ_v) not included in the further optimal approximation

Policy gradient theorem:

$$\nabla J(\theta_p) = \mathbb{E}\left[\sum_a q_\pi(s,a)\,\nabla_{\theta_p}\pi(a|s,\theta_p)\right]$$

$$\nabla\eta(\theta_P) = E\left[\hat{Q}_\pi(s,a)\nabla_{\theta_P}\pi(a\mid s,\theta_P)\frac{1}{\pi(a\mid s,\theta_P)}\right]$$

$$\nabla \log f(x) = \frac{\nabla f(x)}{f(x)}$$

$$\nabla\eta(\theta_P) = E\left[Q_\pi(s,a)\nabla_{\theta_P}\log\pi(a\mid s,\theta_P)\right]$$

? Expected value.

$$\nabla\eta(\theta_P) = E\left[\sum_a Q_\pi(s,a)\nabla_{\theta_P}\pi(a\mid s,\theta_P)\right]$$

gradient of performance = Sum of returned rewards of action over states S × gradient of policy parameterized with p

$$\nabla\eta(\theta_P) = E\left[G\,\nabla_{\theta_P}\log\pi(a\mid s,\theta_P)\right]$$

G: return we get from playing an episode

parameterized ~~pdf~~ policy

Steps

1. Play episode & cal. the return
2. perform gradient ascent

→ Much different than gradient descent.

→ Since goal is to max. not minimize reward.

Batch gradient descent

By the time batch is over, have the rewards.

$$E(X) = \frac{1}{N} \sum_{n=1}^{N} x_n \longrightarrow \text{Expected value : mean of sample}$$

$$\nabla \eta(\theta_p) \approx \frac{1}{T} \sum_{t=1}^{T} G_t \nabla_{\theta_p} \log \pi(a_t, s_t, \theta_p)$$

$$E(X) \approx \frac{1}{N} \sum_{n=1}^{N} X_n$$

$$\nabla \eta(\theta_P) \approx \frac{1}{T} \sum_{t=1}^{T} G_t \nabla_{\theta_P} log\pi(a_t \mid s_t, \theta_P)$$

$$\frac{1}{T} \sum_{t=1}^{T} G_t \nabla_{\theta_P} log\pi(a_t \mid s_t, \theta_P) = \frac{1}{T} \sum_{t=1}^{T} \nabla_{\theta_P} G_t log\pi(a_t \mid s_t, \theta_P)$$

constant

$$\frac{1}{T} \sum_{t=1}^{T} \nabla_{\theta_P} G_t log\pi(a_t \mid s_t, \theta_P) = \frac{1}{T} \nabla_{\theta_P} \sum_{t=1}^{T} G_t log\pi(a_t \mid s_t, \theta_P)$$

derivative of sum = sum of derivative

$$minimize: \; -\sum_{t=1}^{T} G_t log\pi(a_t \mid s_t, \theta_P)$$

of an episode because this involves the sum

minimize is opp of maximize

T = length of episode

t = 't' time step of episode

Stochastic Grad. method.

$$\theta_{P,t+1} = \theta_{P,t} + \alpha G_t \frac{\nabla \pi(a_t \mid s_t)}{\pi(a_t \mid s_t)} \left(= \theta_{P,t} + \alpha G_t \nabla log \pi(a_t \mid s_t)\right)$$

$\theta(a_t|s_t)$

maximize reward & value of 'G'

π & grad $\nabla(\pi)$

↓ direction of max reward.

(π) → : prob of choosing action(a)

grad (π) → direction of gradient increase in (π)

$$\theta_{V,t+1} = \theta_{V,t} + \alpha(G_t - V_t)\nabla V_t$$

Faster learning rate.

Baseline of V(s)

$V(s_t)$

Policy gradient method:

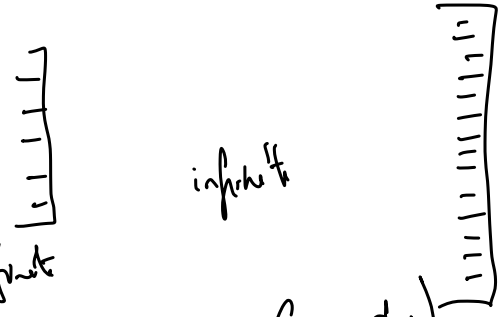
Continous action spaces. ⟷ vs Discrete action spaces.

What distribution appropriate?

$\mathcal{E}(\mu)$ σ (σ) parameters $(wx+b)$

$$\pi(a|s) = \frac{1}{\sqrt{2\pi v}} e^{\left(1 - \frac{1}{2v}(a-\mu)^2\right)} = N(a; \mu, v)$$

Maintain ea. continuous gaussian.



μ: $\theta_\mu \cdot \phi(s)$ (mean) [feature vector]

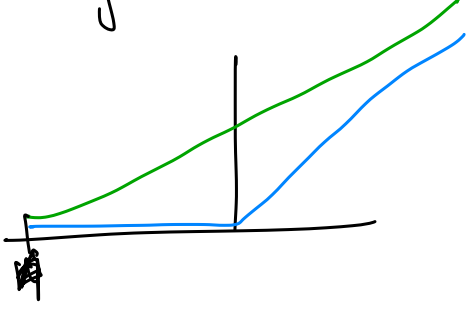
variance: $\exp\left(\theta_v \cdot \phi(s)\right)$ [feature vector.]

Variance: $\text{softplus}(\theta^T, \phi(s)) = (+ve)$

make sure variance is always positive.

Better

grad-descent θ: unbounded. Softplus.



Policy gradient

$$\theta : \theta + \alpha \sum (G_t - V(S_t)) \nabla \log \pi(a_t | S_t)$$

$$\pi(a \mid s) = \frac{1}{\sqrt{2\pi v}} e^{-\frac{1}{2v}(a-\mu)^2} = N(a; \mu, v)$$

$$\mu(s; \theta_\mu) = \theta_\mu{}^T \varphi(s)$$

$$v(s; \theta_v) = exp(\theta_v{}^T \varphi(s))$$

$$Alternative: \; v(s; \theta_v) = softplus(\theta_v{}^T \varphi(s))$$

$$softplus(a) = ln(1 + exp(a))$$

→ Parameterize the mean & variance.

$$\theta = \theta + \alpha \sum_{t=1}^{T} (G_t - V(s_t)) \nabla log \pi(a_t \mid s_t)$$

$$\theta_{t+1} = \theta_t + \alpha (r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \nabla log \pi(a_t \mid s_t)$$

→ How to update (1).
→ define model (1)
→ update model (2)

Work updates:

→ update - Adagrad, RMS prop.

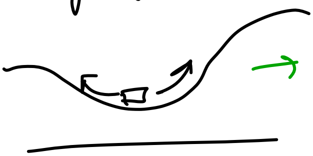
Any output distribution parametrize → policy gradient method

function approximation → argmax/ action to be at step

Q-learning with func? approximation?

- Handle infinite spaces of action space

Policy gradient : Hill climbing



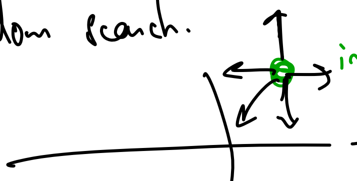
→ Hill climbing

reward. (−x action?)

penalised for taking floppy actions

Hill Climbing

→ special type of random search.



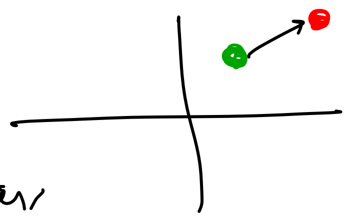
initialised params (randomly)

2 parameters

→ more episodes : variance less

→ less episodes : variance high.

(new)



N.N. -

Linear models → (RBF's)

parameters

Deep learning techniques

$$\theta = \theta + \alpha(r + \gamma max_{a'} Q(s', a') - Q(s, a)) \nabla_\theta Q(s, a)$$

→ not converge to good solution

→ automatic differentiation

Deep Q network

func? approx for $Q(s, a)$.

instead of $f(s, a)$, use $f(s)$.

→ State into input feature, each representing value for state.

→ Experienced Replay

(State, action) ← Tuples

(s, a, r, s')

(State, action, reward, next step)

→ Create env.

↳ Main network

↳ Target network

↳ loop for predetermined # steps

'x'
'G'
} shielded condition

fully connected layer. | hidden layer 'M'

Kontpath.

Actor critic method

making it parameterized.

Actor → Policy

Critic - Value.

G - V(s)

return

reward

- 2 neural networks to parameterize policy and value

$$\pi(a \mid s, \theta_p) = NeuralNet(input : s, weights : \theta_p)$$

→ parameterized policy

$$V(s, \theta_v) = NeuralNet(input : s, weights : \theta_v)$$

→ value.

- For policy loss, work "backwards" from policy gradient, for value loss, use squared error

$$L_p = -(G - V(s)) \log \pi(a \mid s, \theta_p)$$

→ loss of policy

$$L_v = (G - V(s, \theta_v))^2$$

→ loss of value.

Pseudocode:

while not done:

$a = pi.sample(s)$

$s', r, done = env.step(a)$

$G = r + \gamma * V(s')$

$L_p = -(G - V(s)) * \log(pi(s))$

$L_v = (G - V(s))^2$

$\theta_p = \theta_p - lr * \frac{dL_p}{d\theta_p}$

$\theta_v = \theta_v - lr * \frac{dL_v}{d\theta_v}$

ENTROPY: (Policy loss)

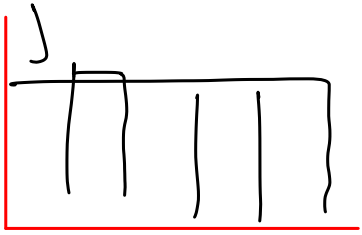
$$H = -\sum_{k=1}^{K} \pi_k \log \pi_k$$

π_k → probability of event

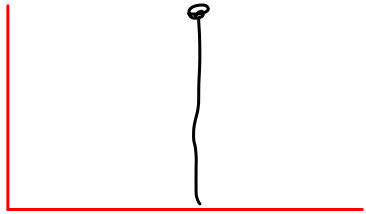
$*$ log probability of event

new loss:

$$L_p' = L_p + (+1$$



max entropy
= uniform
all same prob.



min. entropy
= single event
max prob.