Value function approximation

Konpat Preechakul
Chulalongkorn University
September 2019

Recap

Learning algorithm

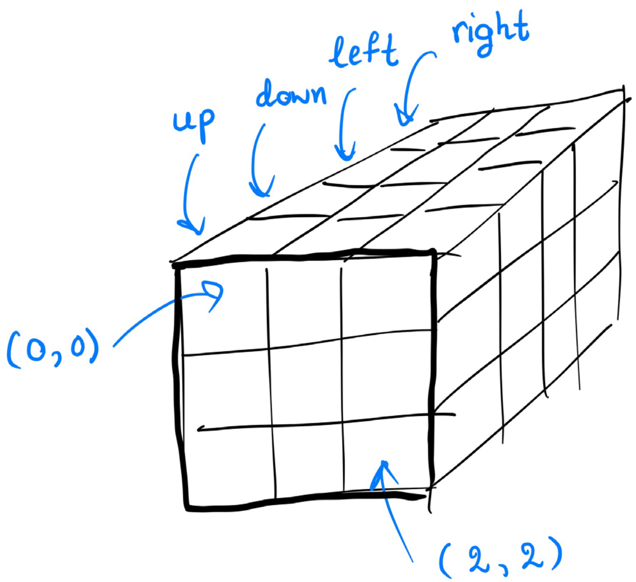
- MC
- TD
- N-step

Implementation

- Tabular
- **Very limited**

Motivation to move away from “tabular”

Limitations of “tabular”



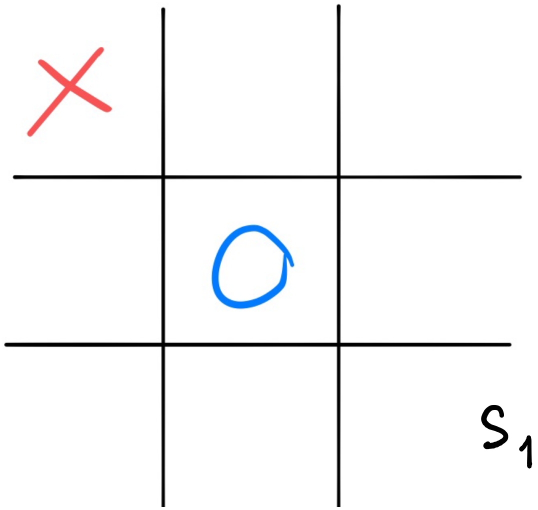
- If we have a grid of 1M x 1M?
  - Memory concern
- If states or actions are continuous?
  - How to “discretize”?

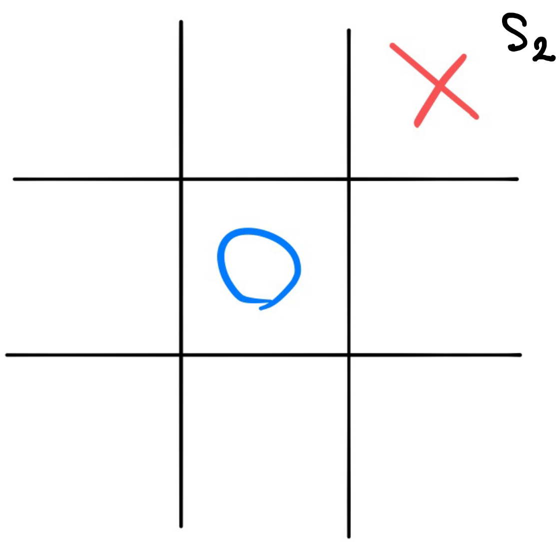
Generalization concern

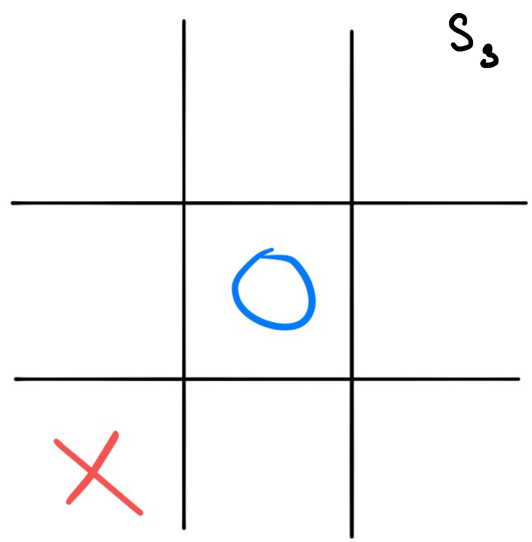
- Grid 1M x 1M = 1 Trillion
- State value for each s? 1 Trillion states
- **How do we even hope to visit them all in limited time?**
- How many times we need to visit each of them to get a sufficient statistic?

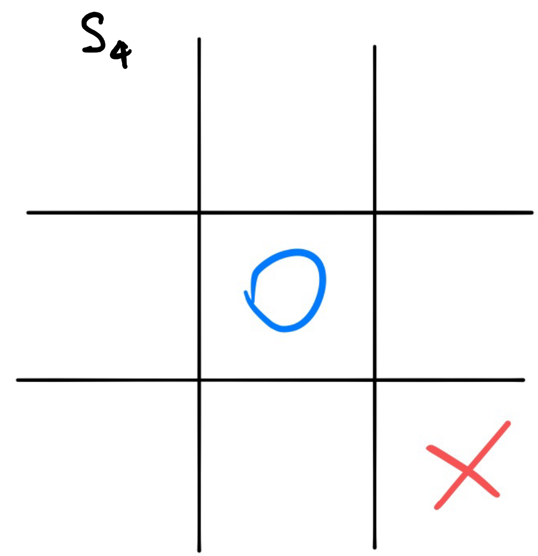
Example of generalization

- These states are equivalent
- Visiting one is the same as visiting one another









Generalization is the only hope

- We cannot deal with very large number of states
- We hope to come up with a “representation” of states
- Such that we only need to keep a “fraction” of the states

What if we represent states differently?

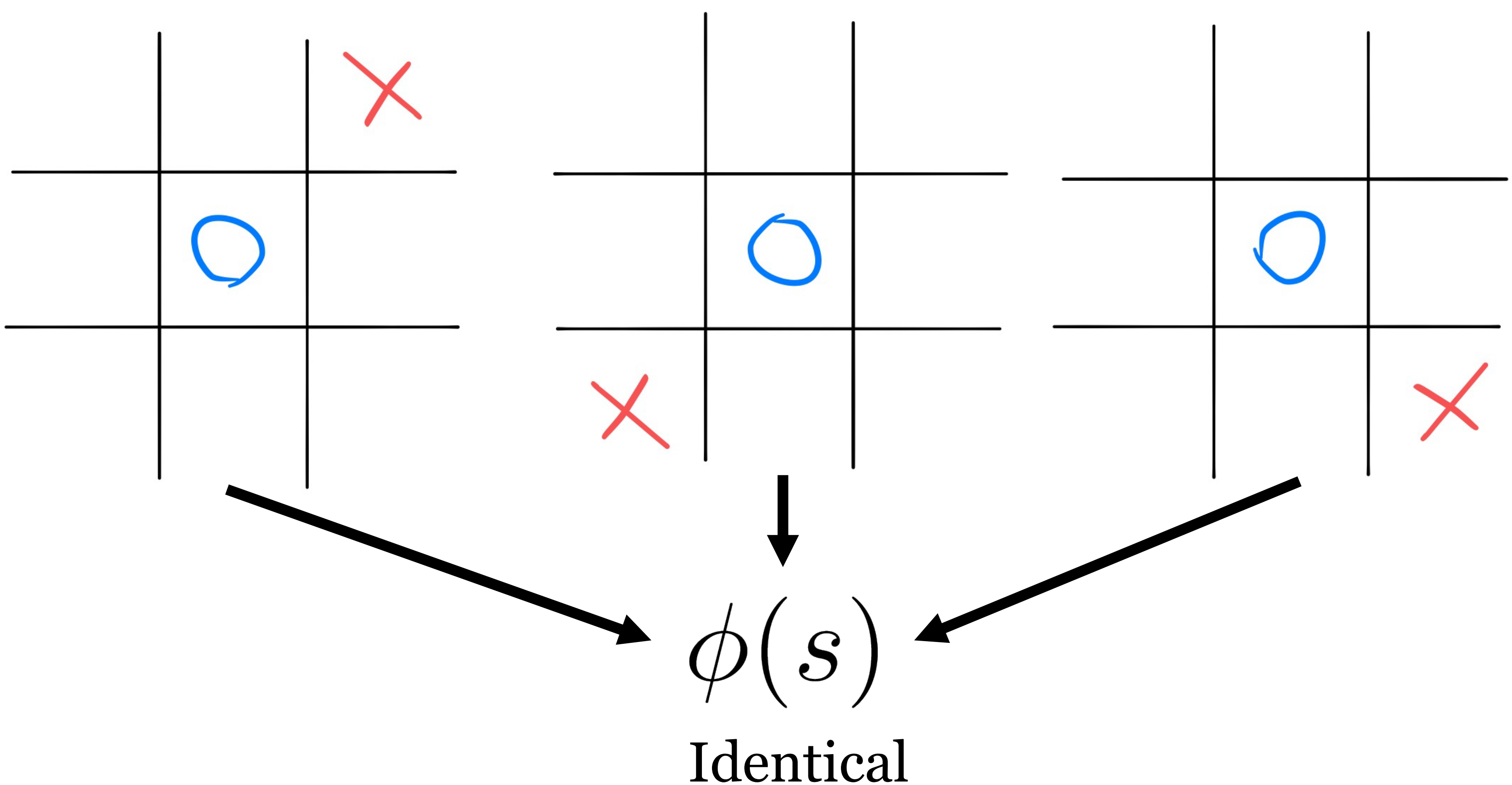
Better state representation

State **feature** function

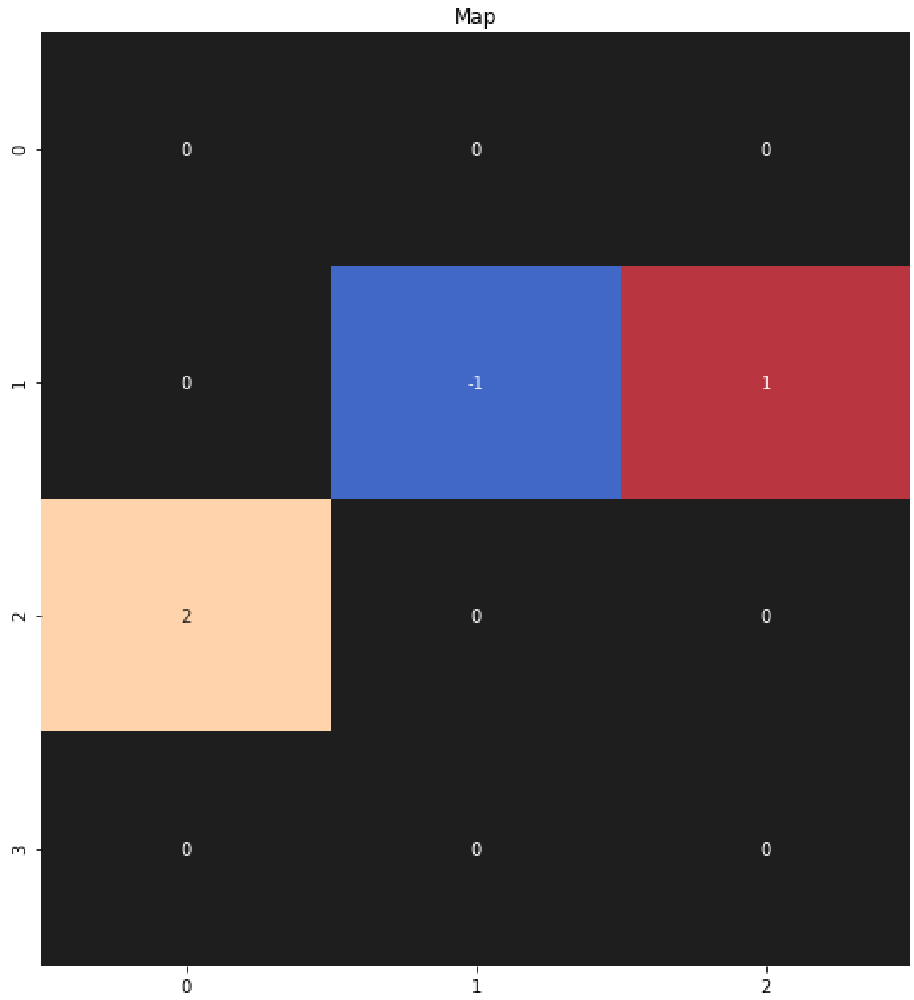
$$\phi(s) = (x^{(0)}, x^{(1)}, \ldots, x^{(k)})$$

- Takes a state as an input
- Outputs a vector representing the state
- **Feature engineering**

Rotational invariant feature



Example: Gridworld



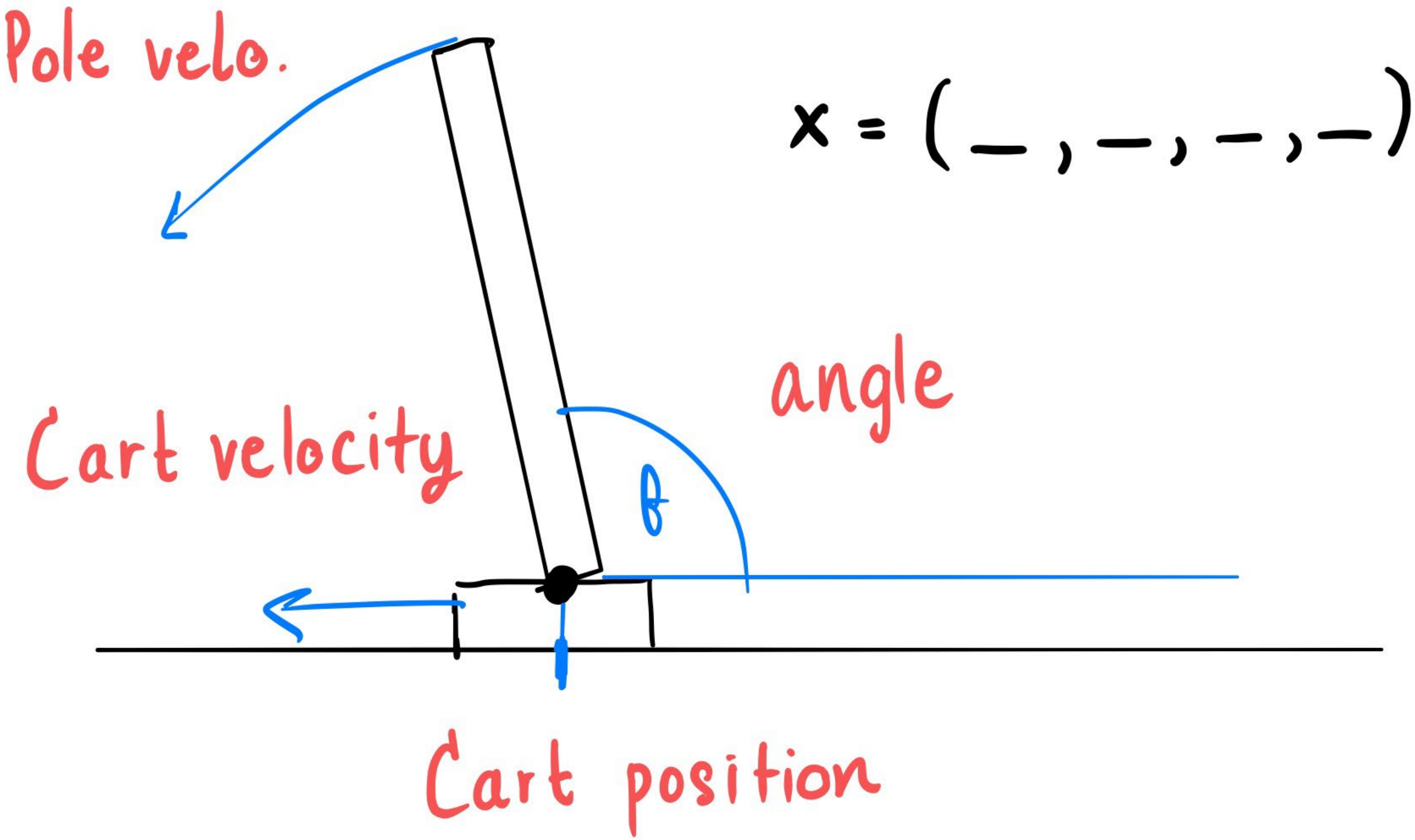
Vector

- Current position
- Distance to traps
- Distance to goal

$$(x, y, \underbrace{T_1, T_2, \dots, T_n}_{\text{Distance to traps}}, G)$$

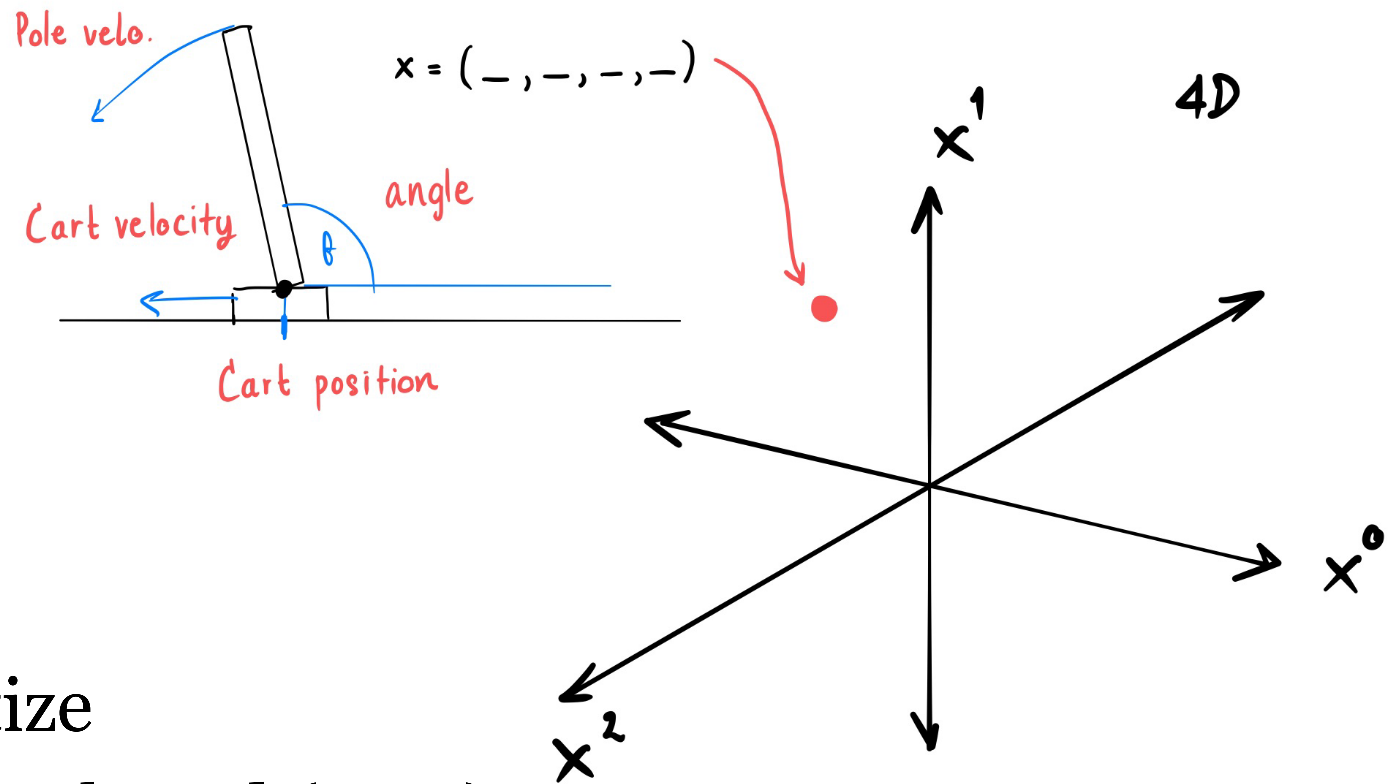
Distance to goal

Extending to continuous space



Cart Pole

Extending to continuous space



- Discretize
- Distance-based (RBF)
- Learned feature function (Neural nets)

Example: RBF Feature

Radial basis function (RBF) is in a form:

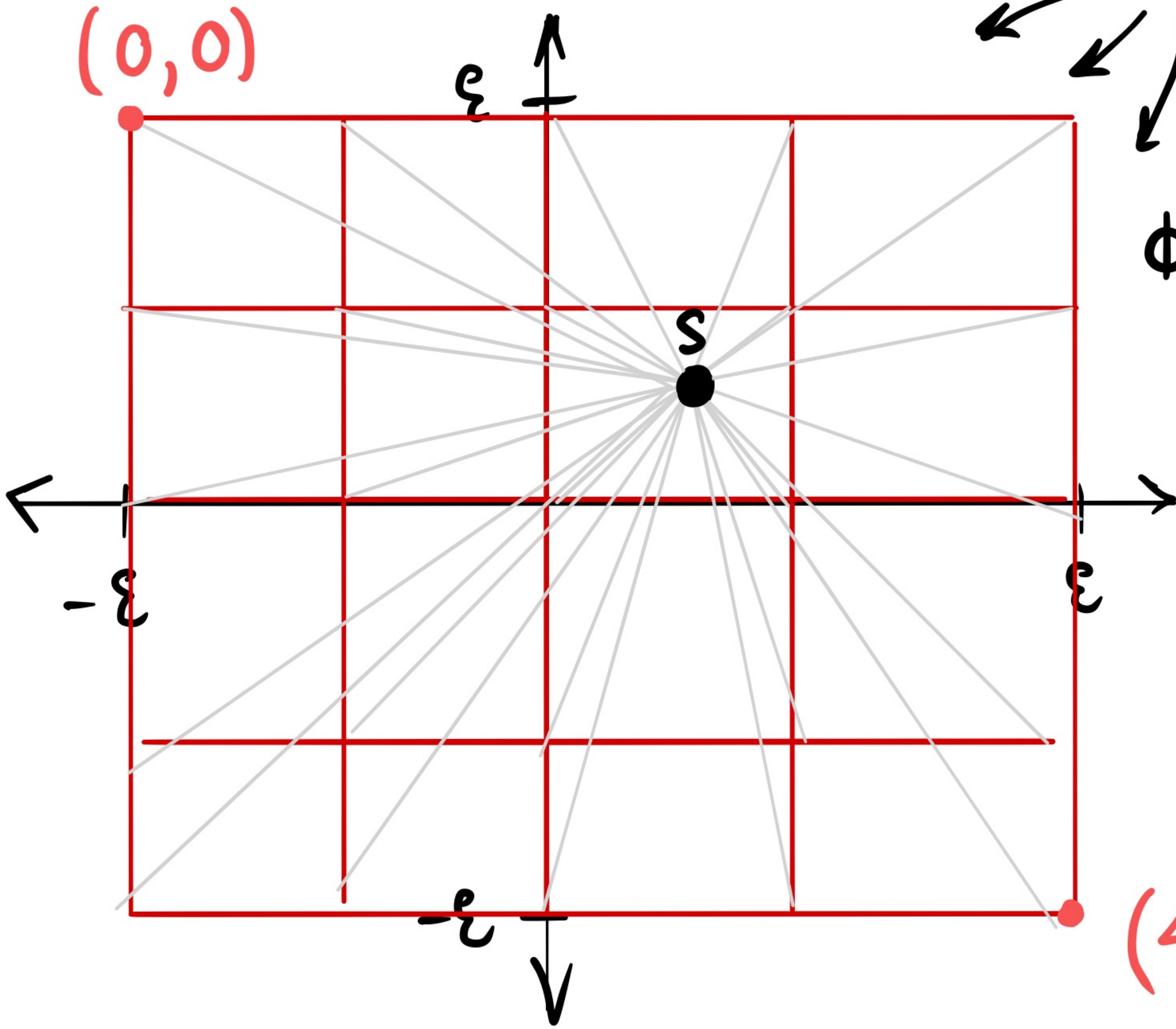
$$\mathrm{x}_i(s) = f(|x - c_i|)$$

Gaussian basis function:

$$\mathrm{x}_i(s) = \exp\left(-\frac{\|x - c_i\|^2}{2\sigma_i^2}\right)$$

c_i are predefined center points

Example: RBF Feature

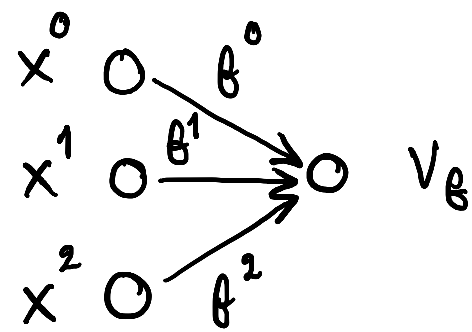


$\phi(s)$ = Distance to each center point

Value is a function of the representation

- Linear function

$$v_\theta(s) = \theta^T \phi(s)$$



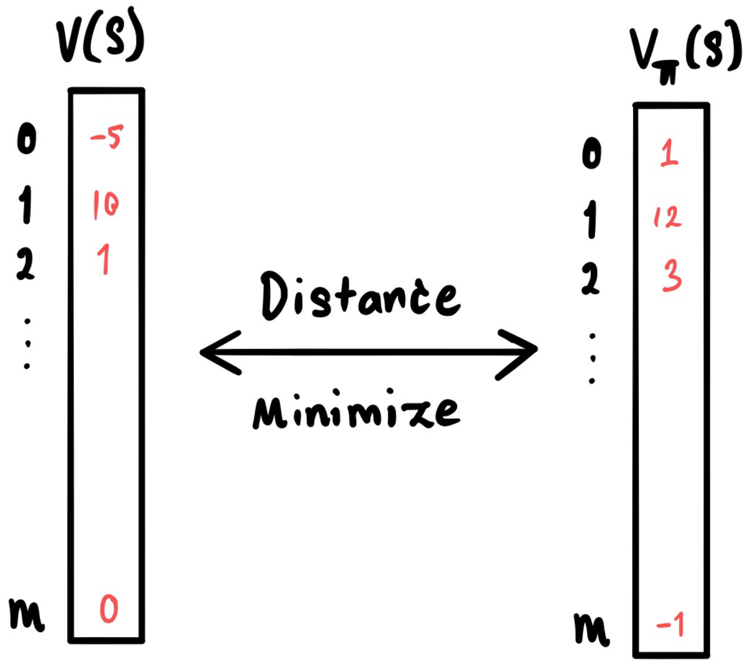
- Non-linear function

$$v_\theta(s) = f_\theta(\phi(s))$$

With parameters **theta**

The goal of approximation

- Minimizing the distance between approximated values and the correct value
- Under some distance function



Value is approximated

- Number of states $|S|$ is large
- We hope to approximate their values with limited "budget" $|\theta|$
- Usually, $|\theta| \ll |S|$

Limitation of approximations

- We cannot hold all values perfectly
- *We do better on one, we do worse on others*
- How to prioritize state values?

Does it even make sense to approximate?

Will we still get a good policy?

- It is unclear how much the policy is hurt with approximated value function
- Approximation makes sense if:

$$v_\pi(s) \propto v^*(s) - \text{error}$$

  - Where π follows $\hat{q}(s, a)$ greedily
- Less error, better policy
- More error, worse policy

Yes, it makes sense!

We can show that:

$$v_\pi(s) \geq v^*(s) - \frac{2\epsilon}{1-\gamma}$$

- $v^*(s)$ is the optimal policy performance
- $v_\pi(s)$ is our policy (using $q_\theta(s,a)$)
- ϵ the maximum error between $q_\theta(s,a)$ and $q^*(s,a)$
- Our policy has a lower bound depending on the error!

Yes, it makes sense!

$$v_\pi(s) \geq v^*(s) - \frac{2\epsilon}{1-\gamma}$$

The term: $\dfrac{2\epsilon}{1-\gamma}$

- If you care more about the future (large gamma)
  - Small error could add up to a large amount
  - Our policy could be much worse than the optimal
- If you care less
  - Error means less

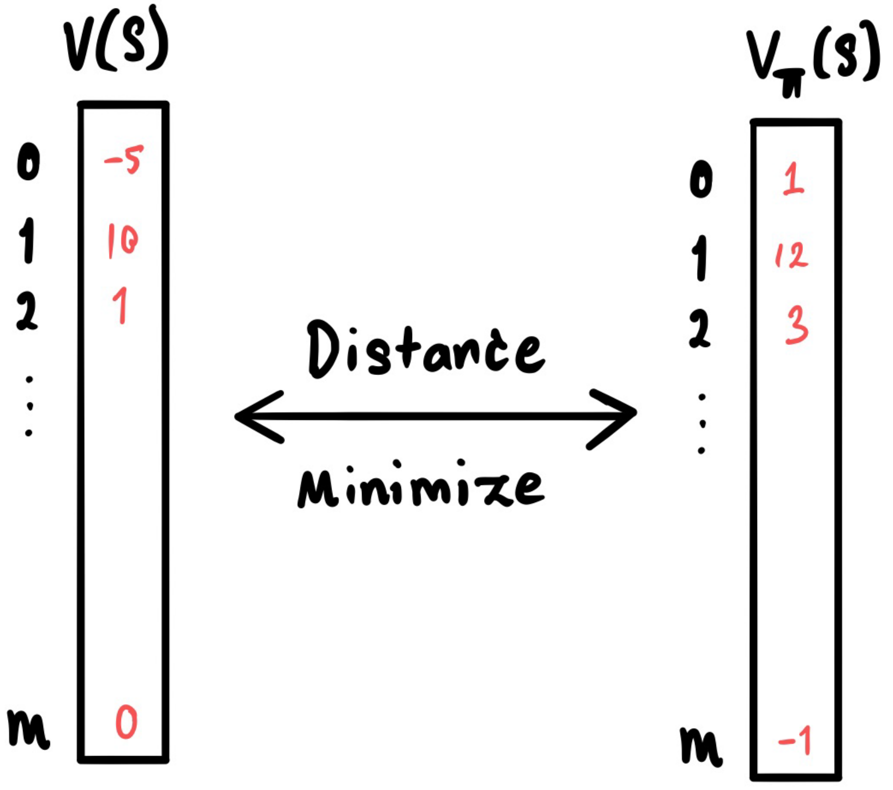
Neural network as a value function

Why neural nets?

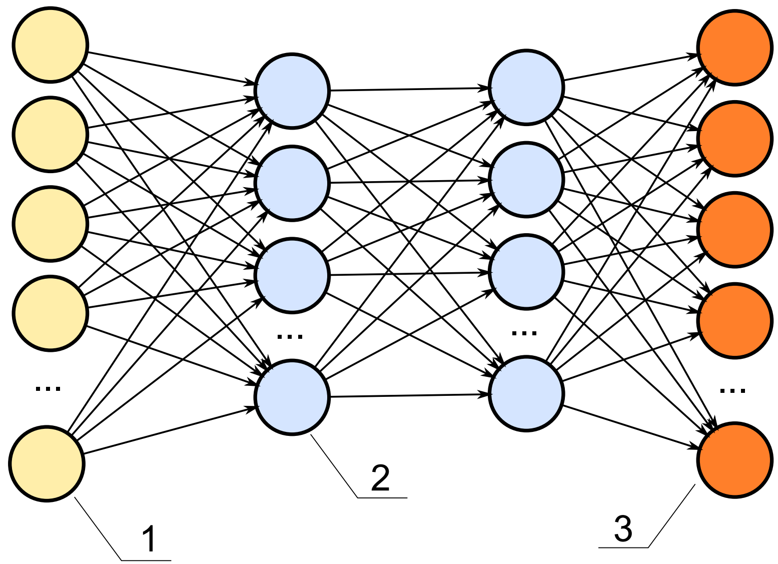
- Universal approximation theorem
  - Deep neural nets can learn any function
- We know it work on many problems
  - Regression
  - Classification
- We know how to train it
  - SGD

What do we want them do?

- Learn to remember the state function



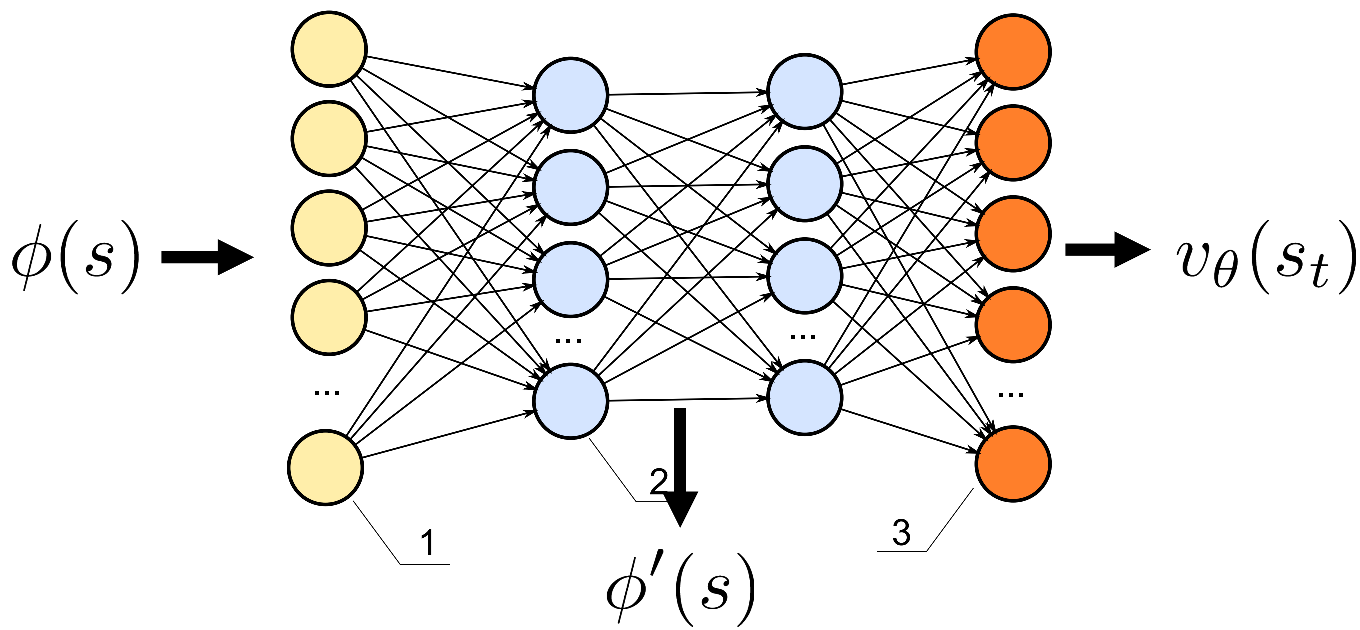
Anatomy of neural nets



- Input $\phi(s)$
- Weights θ
- Output $v_\theta(s_t) = f_\theta(\phi(s))$
- Single layer $v_\theta(s_t) = \theta^T \phi(s)$

https://upload.wikimedia.org/wikipedia/commons/8/8b/Neural_network_bottleneck_achitecture.svg

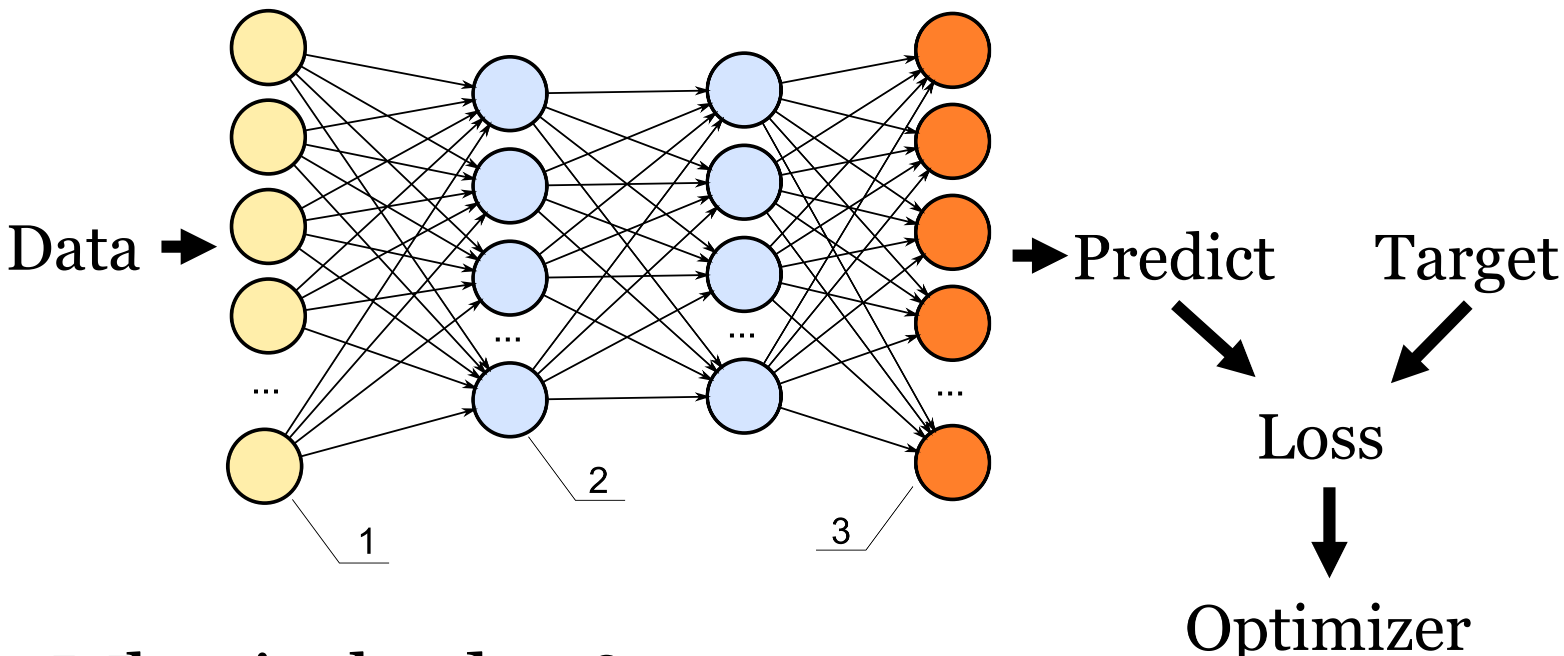
Feature and value in one



- First layers further learn state representation
- Last layers learn value regression

How to train neural nets?

Supervised learning



- What is the data?
- What is the loss function?
- What is the trainer? **Gradient descent**

What is the loss function?

Let’s start simple with **“mean squared error”**

Prediction $v_\theta(s)$

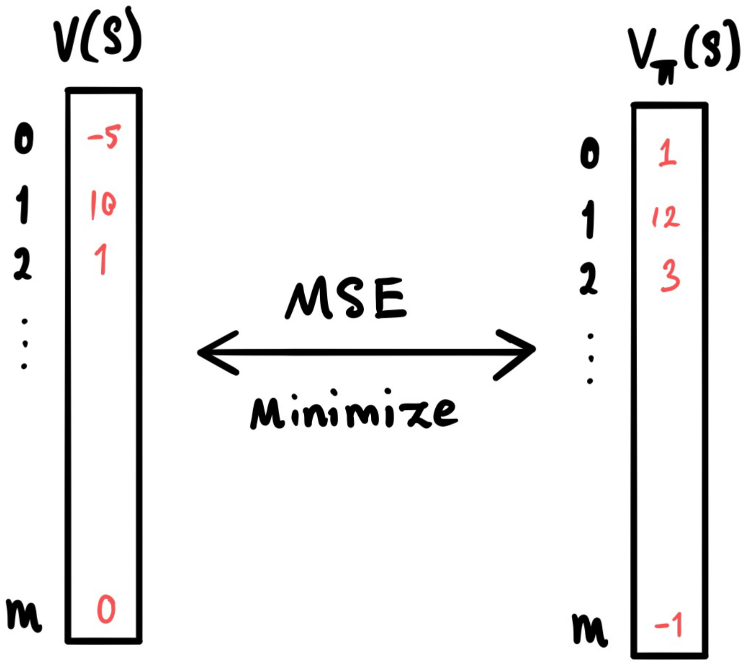
Target $v_\pi(s) = \mathbb{E}_\pi\left[G | S_0 = s\right]$

Loss function

$$\mathcal{L}(\theta) = \sum_s \mathrm{P}^\pi(s)\left[\frac{1}{2}\left(v_\pi(s) - v_\theta(s)\right)^2\right]$$

- P(s) is how frequent s is visited under on-policy
- P(s) is how importance is the state; weight

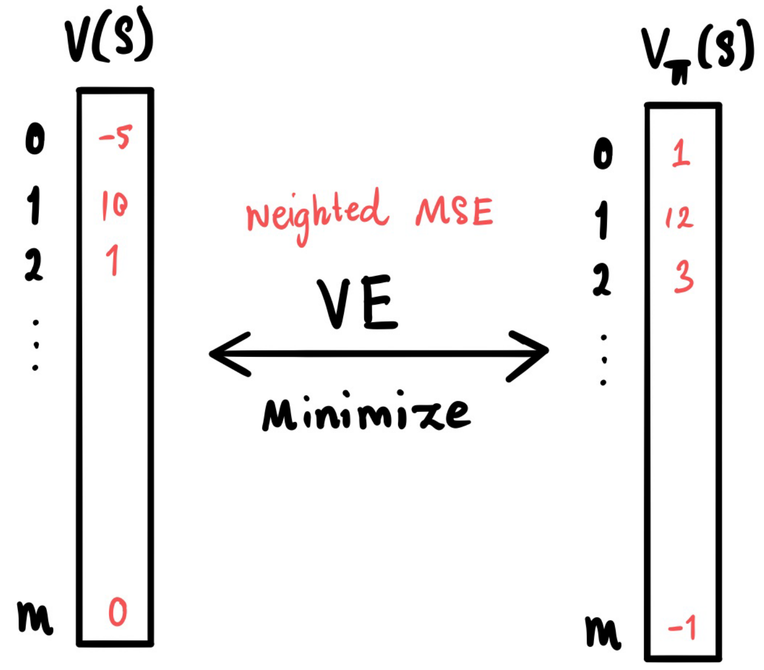
Why do we need state importance?



- What is the desirable solution?
- Equal importance?

$$\mathcal{L}(\theta) = \frac{1}{m} \sum_{s} \left[\frac{1}{2} \left(v_{\pi}(s) - v_{\theta}(s) \right)^2 \right]$$

Why do we need state importance?



- We can visit only a few states? We should not care much of the rest

$$\mathcal{L}(\theta) = \sum_{s} \mathrm{P}^{\pi}(s) \left[\frac{1}{2} (v_{\pi}(s) - v_{\theta}(s))^2 \right]$$

Value Error (VE)

$$\mathcal{L}(\theta) = \mathrm{VE}(\theta) = \sum_{s} \mathrm{P}^{\pi}(s) \left[\frac{1}{2} (v_{\pi}(s) - v_{\theta}(s))^2 \right]$$

- We call this **Value Error (VE)**
- Weighted by P(s) is only a reasonable **heuristic**
- Best loss function should be the one that helps improve the policy the most
- But it is unclear which one is

Surrogate target

- Usually we don't have the target:

$$v_\pi(s) = \mathbb{E}_\pi \left[G | S_0 = s \right]$$

- Because we cannot compute expectation
- We use a return instead

$$v_\pi(s) \approx G \sim \pi | S = s$$

- As long as it is an **unbiased** estimate

What is the gradient

$$\mathcal{L}(\theta) = \mathrm{VE}(\theta) = \sum_s \mathrm{P}^\pi(s) \left[\frac{1}{2} (G(s) - v_\theta(s))^2 \right]$$

$$\nabla_\theta \mathcal{L}(\theta) = -\sum_s \mathrm{P}^\pi(s) \left[(G(s) - v_\theta(s)) \nabla_\theta v_\theta(s) \right]$$

- Calculate return on all states
- Calculate difference on all states
- Average by **on-policy state distribution**

What is the optimizer

Batch Gradient Descent (GD)

- Gradients are calculated from all states

Stochastic Gradient Descent (SGD)

- Gradients are calculated from a small set of states
- Depends on the experience

Value Error + GD

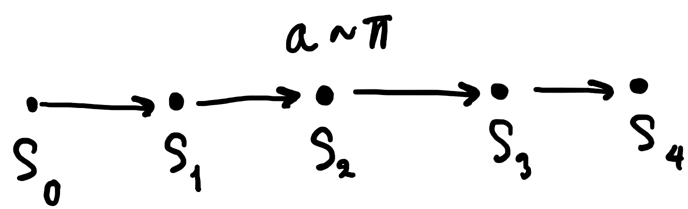
$$\nabla_\theta \mathcal{L}(\theta) = -\sum_s \mathrm{P}^\pi(s) \left[(G(s) - v_\theta(s)) \nabla_\theta v_\theta(s) \right]$$

$$\theta \leftarrow \theta + \alpha \nabla_\theta \mathcal{L}(\theta)$$

- Calculate the gradient for all states
- Update the parameters slowly by the learning rate
- But we don't have $\mathrm{P}^\pi(s)$
- Fortunately we can "sample" from it!

What is on-policy state distribution

- Start from $S_0 \sim P_{s_0}$ initial state distribution
- Follow π until termination
- We get $\tau = (s_0, s_1, s_2, \ldots, s_{T-1})$
- Counter the number of times each state is visited
- Repeat
- The normalized frequency is the on-policy state distribution $\mathrm{P}^{\pi}(s)$



Value Error + SGD

$$\nabla_\theta \mathcal{L}(\theta)$$

$$= -\sum_s \mathrm{P}^\pi(s)\left[\left(G(s) - v_\theta(s)\right)\nabla_\theta v_\theta(s)\right]$$

$$= -\mathbb{E}_{s\sim \mathrm{P}^\pi}\left[\left(G(s) - v_\theta(s)\right)\nabla_\theta v_\theta(s)\right]$$

$$\approx -\left(G - v_\theta(s)\right)\nabla_\theta v_\theta(s) \quad S \sim \mathrm{P}^\pi, G \sim \pi|S$$

$$= -\left(G - v_\theta(s)\right)\nabla_\theta v_\theta(s)$$

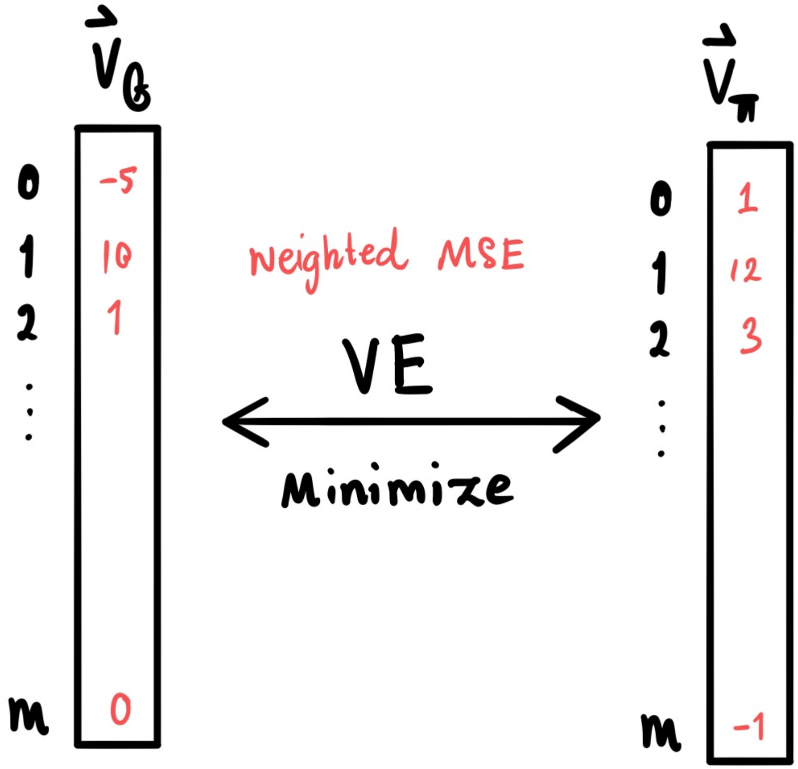
$$S_0 \sim \mathrm{P}_{s_0}, S \sim \pi|S_0, G \sim \pi|S$$

$$\theta \leftarrow \theta + \alpha\left(G - v_\theta(S)\right)\nabla_\theta v_\theta(S)$$

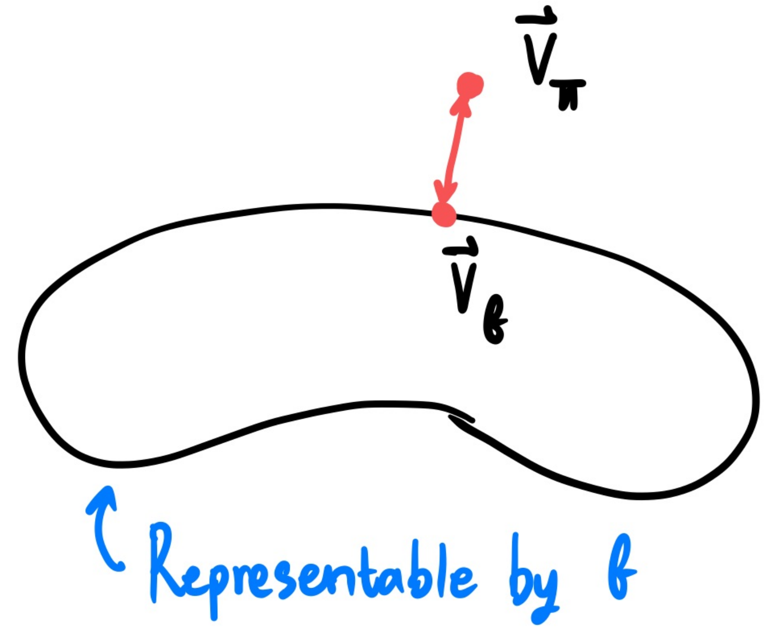
Value Error + SGD Algorithm

for until v is stable **do**
  collect a trajectory τ using π
  for s_t in τ **do**
    $\theta \leftarrow \theta + \alpha \left(G_t - v_\theta(S) \right) \nabla_\theta v_\theta(S)$
  end for
end for

Value Error Solution



$$\underset{\theta}{\operatorname{argmin}} \; \| \vec{V}_\pi - \vec{V}_\theta \|^2_{\rho^\pi}$$



Convergence and fixed point

Convergence

- A training process converges to a fixed point where there is no further progress

Fixed point

- The solution when the training process converges

Value Error limitations

- A full-return needs to wait for an episode
- We cannot implement TD on VE
- We will later learn how to apply approximation onto TD methods
    - We will see that it gives rise to a lot of problems
    - Which are still at the bleeding edge of research

Assignments

- Learn Tensorflow 2 (notebook in github)
- Proof that:

$$\underbrace{v_\pi(s)}_{\text{our policy performance}} \geq v^*(s) - \frac{2\epsilon}{1-\gamma}$$

Given:

$$\max_{s,a} |q_\theta(s,a) - q^*(s,a)| \leq \epsilon$$

$$\pi(s) = \underset{a}{\mathrm{argmax}}\ q_\theta(s,a) \text{ (our approx.)}$$

$$\pi^*(s) = \underset{a}{\mathrm{argmax}}\ q^*(s,a)$$

$$q^*(s,a) = q_{\pi^*}(s,a)$$

Proof guide

$$v_\pi(s) \geq v^*(s) - \frac{2\epsilon}{1-\gamma}$$

- Step 1 prove that:

$$v^*(s) - q^*(s, \pi(s)) \leq 2\epsilon$$

- Step 2 prove (using Step 1):

$$v_\pi(s) \geq v^*(s) - \frac{2\epsilon}{1-\gamma}$$

Hints step 1

$$q_\theta(s, \pi(s)) \geq q_\theta(s, \pi^*(s))$$

- Because π maximize q_θ, hence it selects a greedy action with respect to q_θ
  - Any other policy will have "lower" value (including true optimal policy)
- Greedy policies are deterministic

Hints step 2

- Look for the recursive structure of the equation