Solving MDPs

Reinforcement Learning

Cathy Wu

1.041/1.200 Transportation: Foundations and Methods

Readings

1. Morales, Miguel. *Grokking deep reinforcement learning*. 2020. [URL]

   - Chapter 8: Introduction to value-based deep reinforcement learning.
   - (Optional) Chapter 9: More stable value-based methods
   - (Optional) Chapter 11: Policy-gradient and actor-critic methods

2. Zhang, Aston, et al. *Dive into deep learning*. Cambridge University Press, 2023.

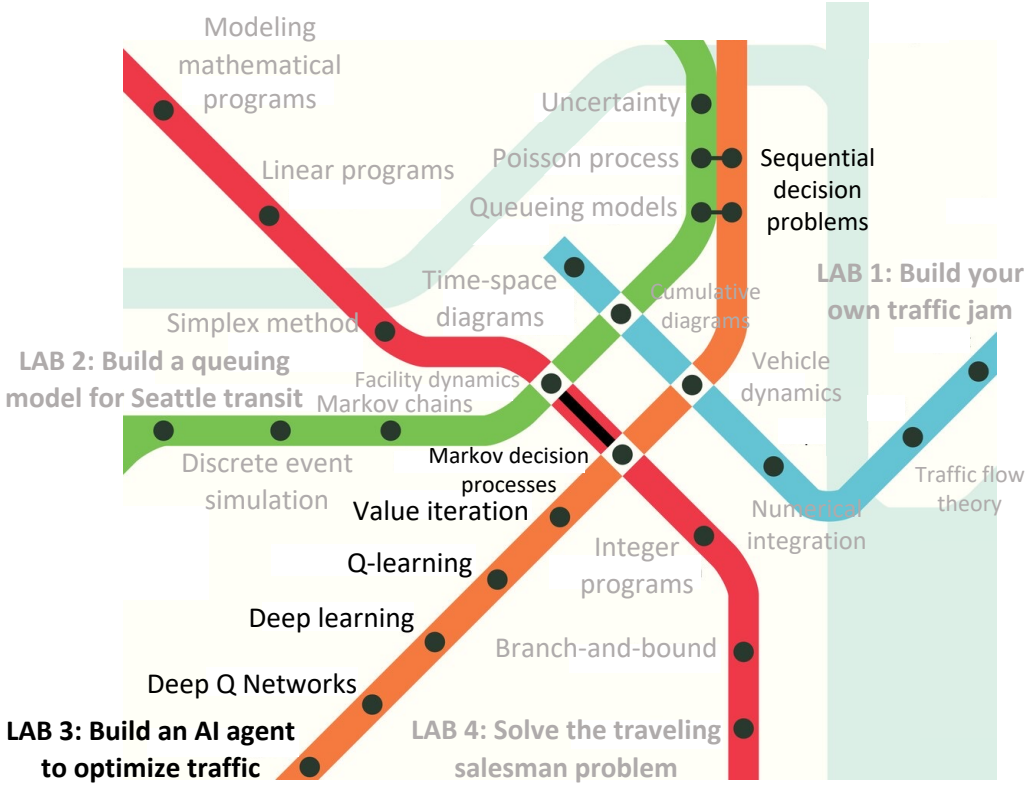
   - Section 2.5 (Automatic differentiation) [link]
   - Sections 5.1-5.3 (Multilayer Perceptrons) [link]
   - Section 12.3 (Gradient Descent) [link]
   - (Optional) Chapters 7-11 (Representations) [link]

Unit 3: Decision Making Under Uncertainty

Unit 3

Optimizing

Multi-stage



Outline

1. Recap Exercise: Parking Problem (Modeling & Solving)
2. Tackling Large State Spaces: Deep reinforcement learning
3. Exploration vs Exploitation
4. Function Approximation
5. Deep learning 101
6. Deep Q Networks (DQN)
7. Policy Gradient
8. Actor Critic

Outline

1. **Recap Exercise: Parking Problem (Modeling & Solving)**
2. Tackling Large State Spaces: Deep reinforcement learning
3. Exploration vs Exploitation
4. Function Approximation
5. Deep learning 101
6. Deep Q Networks (DQN)
7. Policy Gradient
8. Actor Critic

Recall: Dynamic programming algorithm

$V_T(s_T) = r_T(s_T)$
for $t = T-1, \dots, 0$ **do**
  for $s_t \in \mathcal{S}_t$ **do**
    $V_t(s_t) = \max_{a_t \in \mathcal{A}_t(s_t)} r_t(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1} \sim P(\cdot|s_t, a_t)}[V_{t+1}(s_{t+1})]$
end for

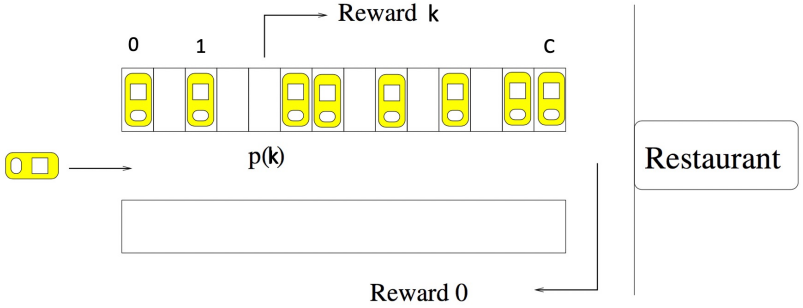
- **Proof**: by induction
- "Efficient": O(|S|^2|A|T)
- With discount $\gamma \in [0,1)$

- For deterministic shortest path routing
  - Equivalent to Bellman-Ford algorithm
  - **Strength**: Generality
  - "Efficient": O(|S||A|T)
  - Much better than naive approach O(T!)
  - **Weakness**: ALL the tail subproblems are solved

Example: Parking Problem

A driver wants to park as close as possible to the restaurant.

Objective: maximize the satisfaction
Problem: Formulate the parking problem as an MDP.



We know (assume) that:

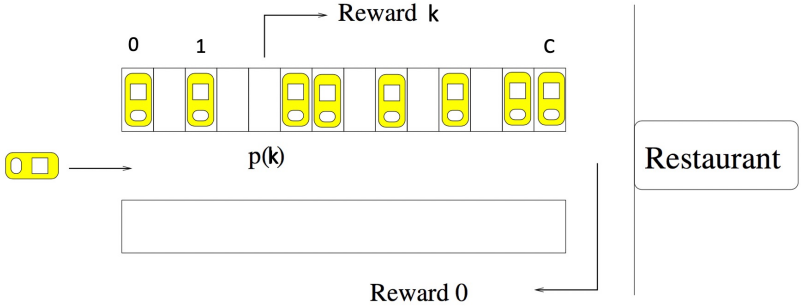
- The driver cannot see if a spot is available unless in front of it.
- There are $C + 1$ parking spots.
- Cannot move backwards. At each place k the driver can either move to the next spot or park (if the place is available).
- Each spot is free with probability $p(k)$, independently of the other spots.
- The closer to the restaurant, the higher the satisfaction. Assume that satisfaction grows inversely with the distance to the restaurant.
- However, parking sooner gives the driver satisfaction. In fact, parking at each later slot gives a factor of 0.9 less satisfaction.
- If the driver doesn't park anywhere, then the driver leaves the restaurant and has to find another place to eat.

Adapted from DPOC vol1 (2005) Chapter 3 and Lazaric (2014) Lecture 2

Example: Parking Problem

A driver wants to park as close as possible to the restaurant.

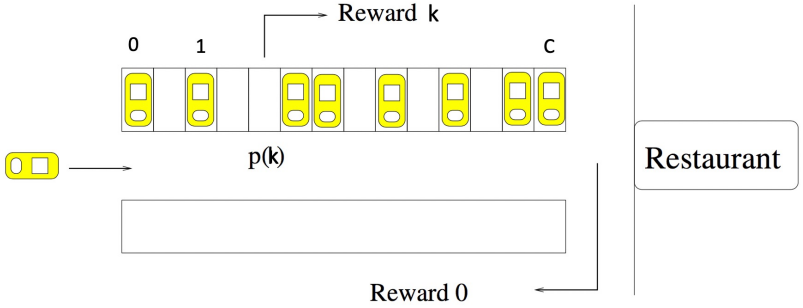
Objective: maximize the satisfaction
Problem: Formulate the parking problem as an MDP.



Example: Parking Problem

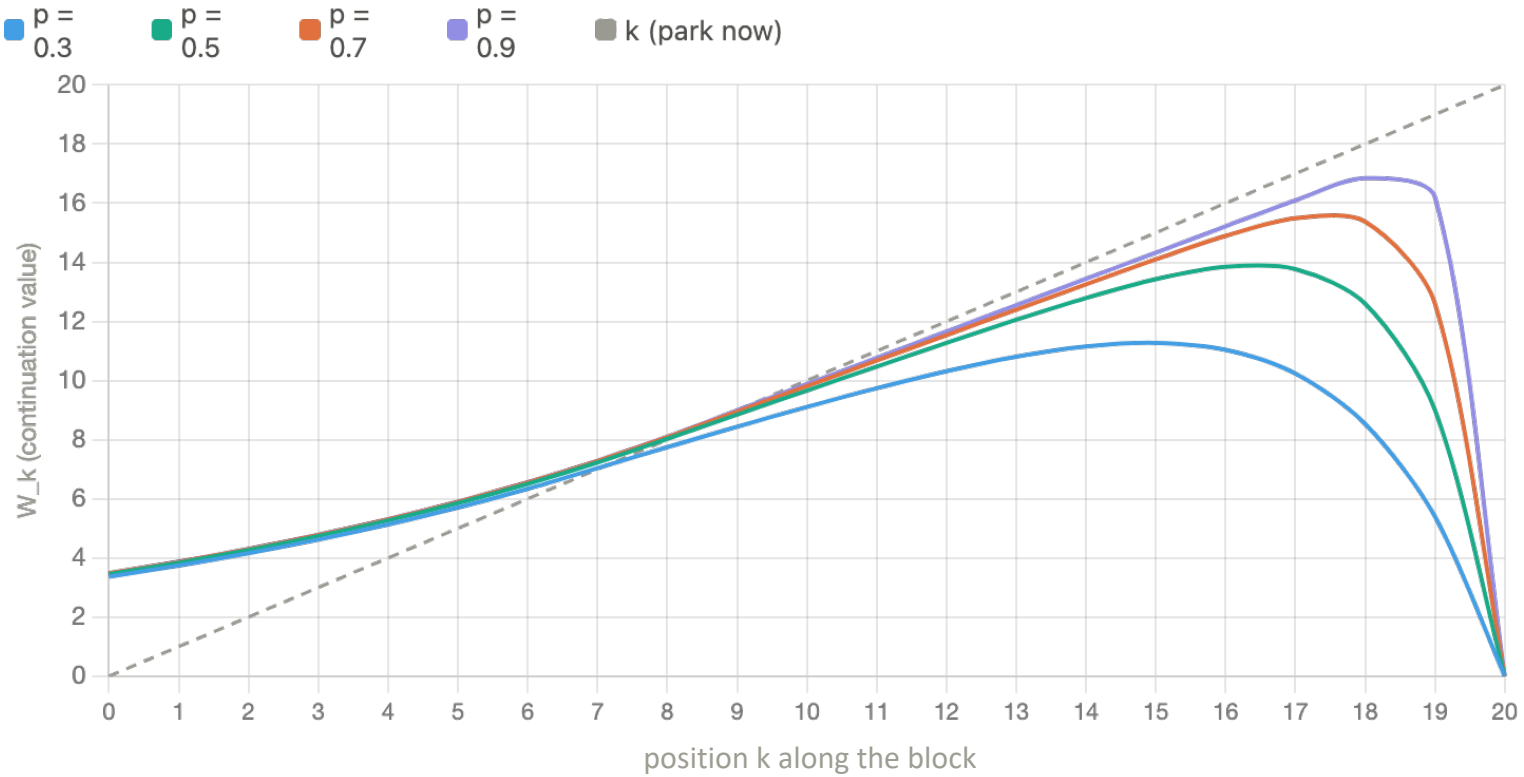
A driver wants to park as close as possible to the restaurant.

Objective: maximize the satisfaction
Problem: Let's solve the parking problem using dynamic programming.



Example: Numerical Solution

$$W_k := V_k^*(T)$$



W_k = continuation value at position k. Intersection with the diagonal (k) marks the threshold k* where parking becomes optimal. γ = 0.9, C = 20.

Applications

- This is an instance of an optimal stopping problem.
- At each stage, the decision maker observes the current state of the system and decides whether to continue the process (perhaps at a certain cost) or stop the process and incur a certain loss.
- Applications in transportation:
  - Curbside management: mitigating effects of mobility-on-demand pick-up & drop-offs
  - Infrastructure investment: when to buy land and build infrastructure
  - Dynamic scheduling / vehicle routing: wait for more information or decide
- Famous application: secretary problem

Further reading

[1] P. N. Stueger, F. Fehn, and K. Bogenberger, "Minimizing the Effects of Urban Mobility-on-Demand Pick-Up and Drop-Off Stops: A Microscopic Simulation Approach," *Transportation Research Record*, vol. 2677, no. 1, pp. 814–828, Jan. 2023, doi: 10.1177/03611981221101894.
[2] J.-D. Saphores and M. Boranet, "Investing in urban transportation infrastructure under uncertainty," in *8th annual real options conference*, 2004.
[3] N. Vodopivec and E. Miller-Hooks, "An optimal stopping approach to managing travel-time uncertainty for time-sensitive customer pickup," *Transportation Research Part B: Methodological*, vol. 102, pp. 22–37, Aug. 2017, doi: 10.1016/j.trb.2017.04.017.

Parking selection in practice

Time-to-Drive (Status Quo)

Time-to-Arrive: Parking Probability-Unaware

Time-to-Arrive: Parking Probability-Aware

Public Transit Time-to-Arrive (Apps Compare w/ Time-to-Drive)

C. Hickert, S. Li, Z. He, and C. Wu, "Probability-Aware Parking Selection," *IEEE Transactions on Intelligent Transportation Systems (T-ITS)*, Feb. 2026, doi: 10.1109/TITS.2025.3649170.

Outline

1. Recap Exercise: Parking Problem (Modeling & Solving)
2. **Tackling Large State Spaces: Deep reinforcement learning**
3. Exploration vs Exploitation
4. Function Approximation
5. Deep learning 101
6. Deep Q Networks (DQN)
7. Policy Gradient
8. Actor Critic

Large state spaces

- DP is inefficient when |S| is large (often exponentially large)
  - $O(|S|^2|A|T)$

$V_T(s_T) = r_T(s_T)$
for $t = T-1, \ldots, 0$ **do**
  for $s_t \in \mathcal{S}_t$ **do**
    $V_t(s_t) = \max_{a_t \in \mathcal{A}_t(s_t)} r_t(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1} \sim P(\cdot|s_t, a_t)}[V_{t+1}(s_{t+1})]$
end for

Key challenge: huge state spaces

- State: location, time, and vehicle type
  - Location is encoded from geohash6 (precise location) [1,600] and geohash5 (neighborhood) [50]
  - Time encoded from hour-of-week categories [168]
  - Vehicle type: standard, luxury, SUV, or handicap accessible [4]
- State space is ≈1600x50x168x4=54M
- For reference: SF Bay Area population is 8M
  - Naïve approach: Would need everyone to take at least 7 rides to gather enough data

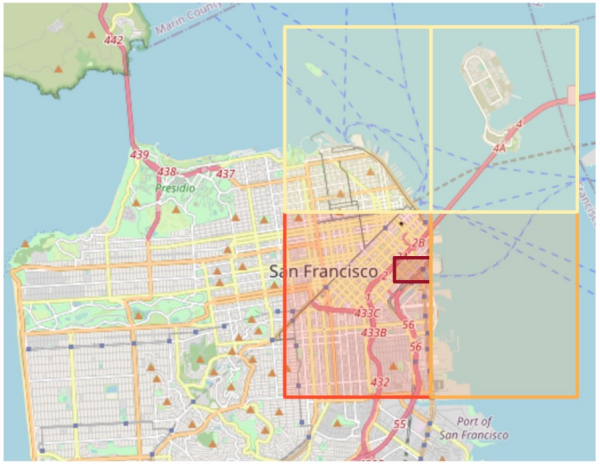
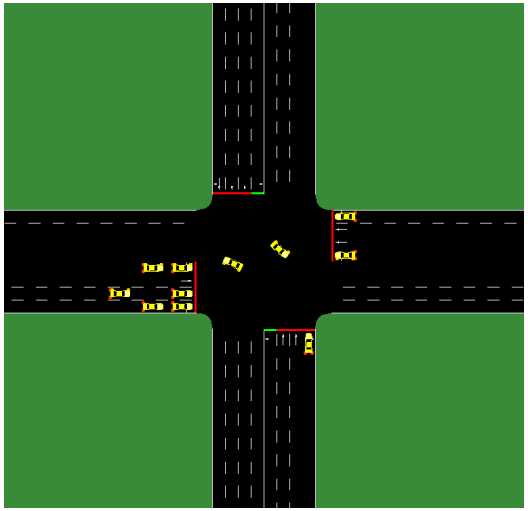


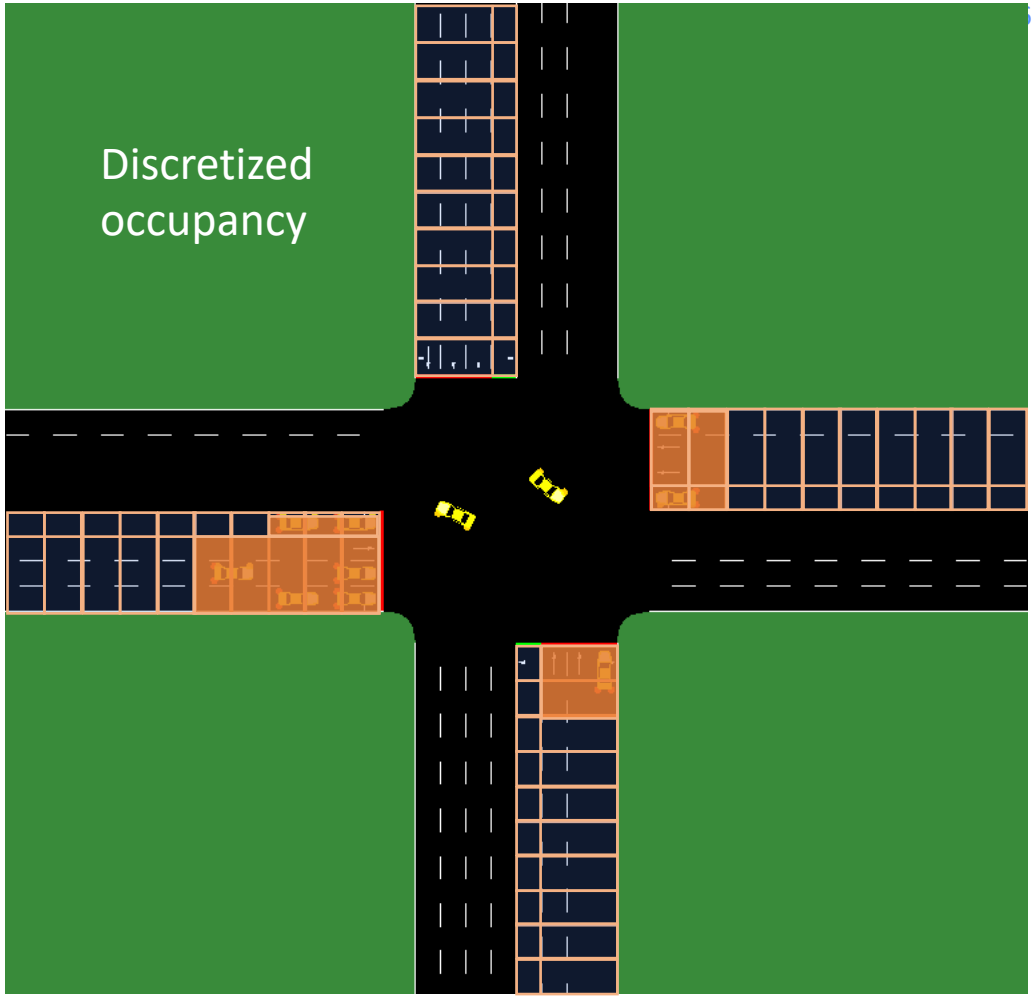
Figure 4: Spatial factor weights are weighted and normalized by the inverse of the distance from the geohash centroid to smoothly interpolate the four closest geohash5 state factors. A similiar interpolation is applied using the two nearest hours of the week, yielding a cross-product of eight spatiotemporal factors and weights. Additional factors also consider the vehicle type, such as standard, luxury, SUV, or handicap accessible.

Cannot only explore. Cannot only exploit.
Must trade off exploration and exploitation.

B. Han, H. Lee, and S. Martin, "Real-Time Rideshare Driver Supply Values Using Online Reinforcement Learning," in *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, in KDD '22. Aug. 2022. doi: 10.1145/3534678.3539141.

State representation





$O(|S|^2|A|T) = 2^{80\times2}\times4\times5400$

Solution: Approximate the value function

$$Q_\theta(s,a) \approx \max_{a' \in A} r(s,a) + \gamma \mathbb{E}_{s' \sim P(\cdot|s,a)}\left[V(s')\right]$$

- Where θ are parameters of some function class (linear, deep neural network, etc.), and timesteps are dropped for simplicity
- Desired: $|\theta| \ll |S|$

- How to obtain Q_θ? Reinforcement learning! Rough idea:
  - Use recursive definition (Bellman's Optimality Equation):

$$Q(s,a) \approx \max_{a' \in A} r(s,a) + \gamma \mathbb{E}_{s' \sim P(\cdot|s,a)}\left[Q(s',a')\right]$$

1. Initialize Q_θ randomly
2. Collect a batch of data D from the MDP, e.g. tuples $(s,a,r,s') \in D$ using Q_θ
3. Use Q_θ to estimate the RHS for each tuple in D and the error $|LHS - RHS|$
   - $\left|Q_\theta(s,a) - \left(\max_{a' \in A} r + \gamma Q_\theta(s',a')\right)\right|$
4. If close, great! If not, update θ to reduce the error over the batch of data
   - → Stochastic gradient descent + Backpropagation algorithm (chain rule)
5. Repeat from step 2 until convergence (not guaranteed).

Terminology

- When $|\theta| = |S| \times |A|$:
  - Setting: Tabular
  - Algorithm: Q-learning

- When $|\theta| \ll |S|$:
  - Setting: Function approximation
  - Basic algorithm: Fitted Q Iteration
  - When θ is parameters of a deep neural network + more algorithmic tricks:
    - Deep Q Networks (DQN), Double DQN, Rainbow DQN, …

- Overall class of methods: **value-based** reinforcement learning
  - When θ is parameters of a deep neural network:
    value-based **deep** reinforcement learning

Potential issues

- What if Q_θ only collects bad data, i.e., bad actions result?
  - After all, Q_θ is initialized arbitrarily
  - Exploration vs Exploitation
- What makes for a good representation Q_θ of Q^*?
  - Function approximation classes
- How to update Q_θ?
  - Stochastic gradient descent & backprop algorithm (chain rule)

Outline

1. Recap Exercise: Parking Problem (Modeling & Solving)
2. Tackling Large State Spaces: Deep reinforcement learning
3. **Exploration vs Exploitation**
4. Function Approximation
5. Deep learning 101
6. Deep Q Networks (DQN)
7. Policy Gradient
8. Actor Critic

Exploration vs Exploitation

- How to ensure that learning agent visits potentially good states?
- General strategy: Use Q_θ to select an action a
- Complete exploitation:

$$\arg\max_{a \in A} Q_\theta(s, a)$$

- Complex exploration:

$$\pi(a|s) = \frac{1}{|\mathcal{A}|}$$

- ϵ-greedy: a simple strategy to balance the two
  - With probability ϵ, explore. Else, exploit

Outline

1. Recap Exercise: Parking Problem (Modeling & Solving)
2. Tackling Large State Spaces: Deep reinforcement learning
3. Exploration vs Exploitation
4. **Function Approximation**
5. Deep learning 101
6. Deep Q Networks (DQN)
7. Policy Gradient
8. Actor Critic

Representations as Function Classes

A representation (or function approximation class) is a family of parameterized functions

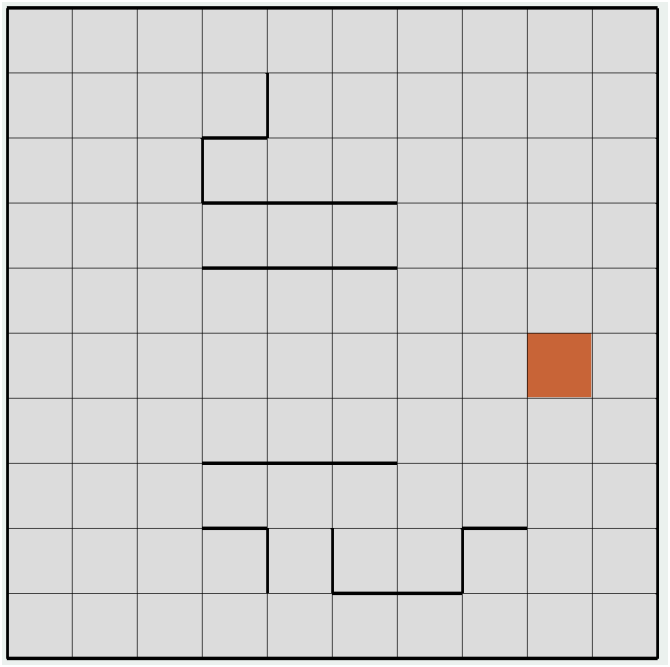
$$\mathcal{F} = \{f_\theta : X \to Y\}$$

Desiderata

- Expressiveness: Can represent the target f^* well
  - i.e., $\exists f \in \mathcal{F}$ such that $f \approx f^*$ (low approximation error)
- Trainability: Can be optimized efficiently
  - e.g., differentiable, scalable optimization
- Inductive biases: Encodes built-in structure that matches the problem
  - Ex: graphs for networks, sequences for temporal processes, utility models for choice, CNNs for imagery
- Data efficiency: Can learn well from the available data
  - Higher-capacity models (more parameters) need more data / inducive biases

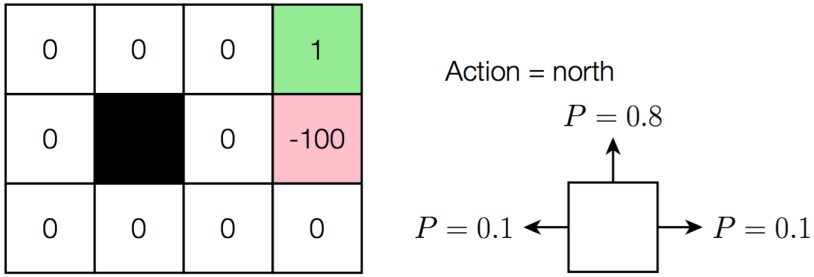
Goal: choose a class that balances expressiveness, trainability, inductive bias, and data efficiency.

The Grid-World Problem



Example: Winter parking (with ice and potholes)

- Simple grid world with a *goal state* (green, desired parking spot) with reward (+1), a *"bad state"* (red, pothole) with reward (-100), and all other states neural (+0).
- *Omnidirectional vehicle (agent)* can head in any direction. Actions move in the desired direction with probably 0.8, in one of the perpendicular directions with.
- Taking an action that would bump into a wall leaves agent where it is.

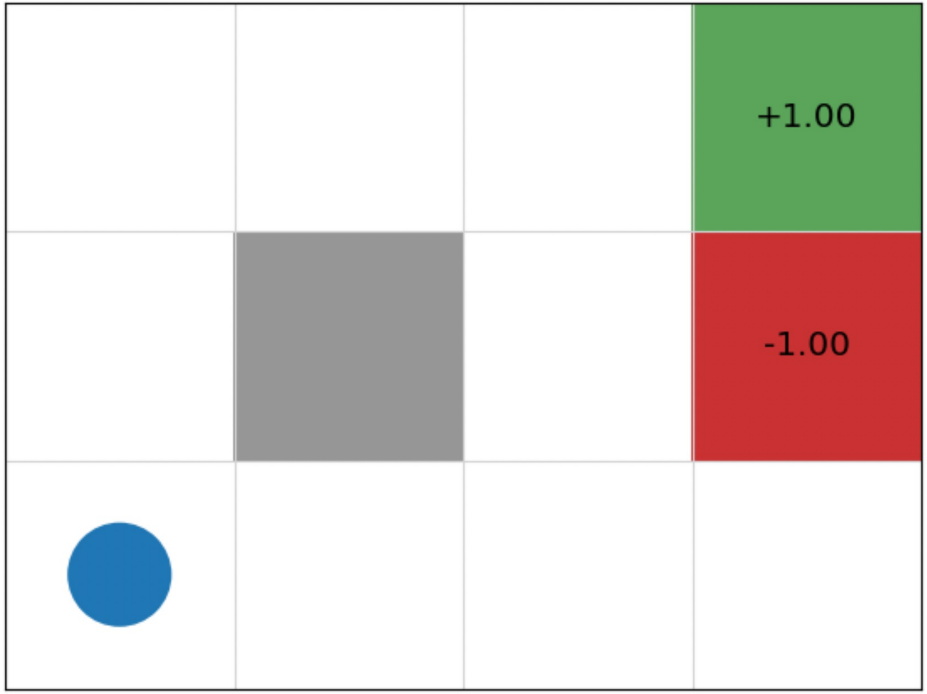


[Source: adapted from Kolter, 2016]

Grid world, revisited

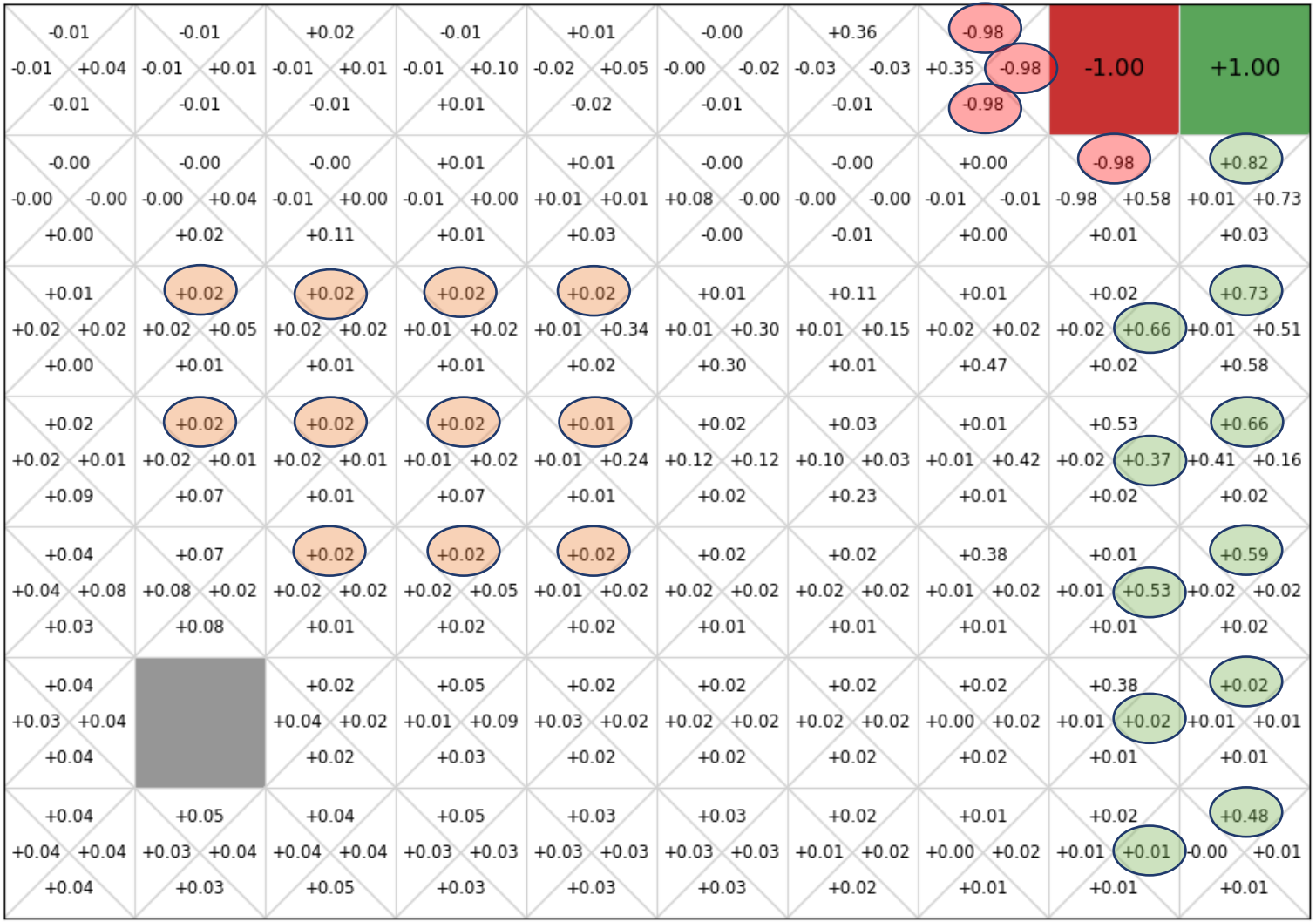
We will revisit the grid world, but now with more states, to make the problem harder. A few items to note:

- An episode terminates when the agent reaches a goal state, either good (+1) or bad (-1).
- By default, we will run Q-learning for 100 episodes. How much can the agent learn in 100 episodes?
- We should expect a random walk from the lower left corner to the top right corner to take a while, especially for large grids.
- In comparison, with sufficient "fake rewards" that nudge the agent in the right direction, we should expect an agent with a shaped reward to quickly find a short path to the good goal state - regardless of how big the grid is.



Q Table

$Q(s, a)$



Example: Grid world with linear representations

- Linear function

$$\tilde{Q}_\theta\big((x,y),a\big) = a_0 + b_1 x + b_2 y + b_3 a$$

  - $\theta = (a_0, b_1, b_2, b_3)$, $|\theta| = 4$

- Linear with polynomial features, 3rd degree

$$\tilde{Q}_\theta\big((x,y),a\big)$$
$$= a_0 + b_1 x + b_2 y + b_3 a + c_1 xy + c_2 xa + c_3 ya + c_4 x^2$$
$$+ c_5 y^2 + c_6 a^2 + d_1 xya + d_2 x^2 y + d_3 xy^2 + d_4 xa^2$$
$$+ d_5 ya^2 + d_6 x^2 a + d_7 y^2 a + d_8 x^3 + d_9 y^3 + d_{10} a^3$$

  - $\theta = (a_0, b_1, b_2, b_3, c_1, c_2, c_3, c_4, c_5, c_6, d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8, d_9, d_{10})$, $|\theta| = 20$

Example: Optimal replacement problem

State: level of wear of an object (e.g., a car).
Action: $\{(R)\text{eplace}, (K)\text{eep}\}$.
Cost:

- $c(x, R) = C$
- $c(x, K) = c(x)$ maintenance plus extra costs.

Dynamics:

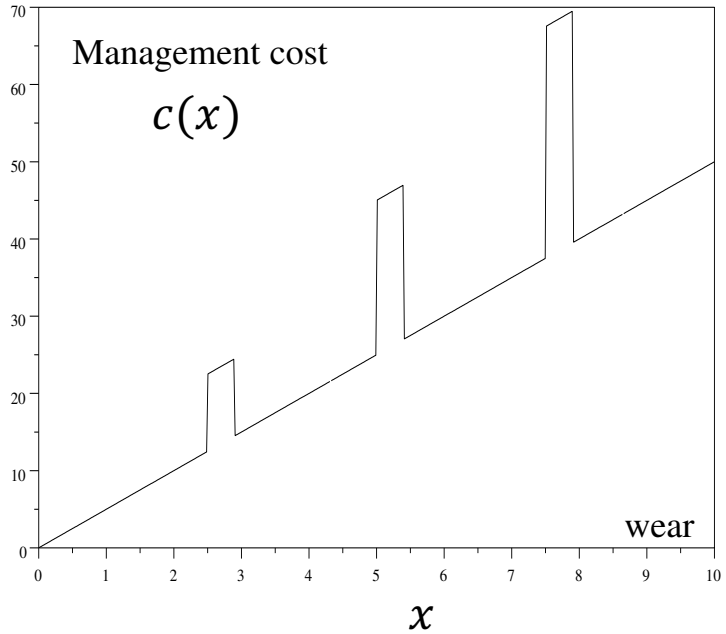
- $p(\cdot|x, R) \sim \exp(\beta)$ with density $d(y) = \beta \exp^{-\beta y} \mathbb{I}\{y \geq 0\}$,
- $p(\cdot|x, K) \sim x + \exp(\beta)$ with density $d(y - x)$.

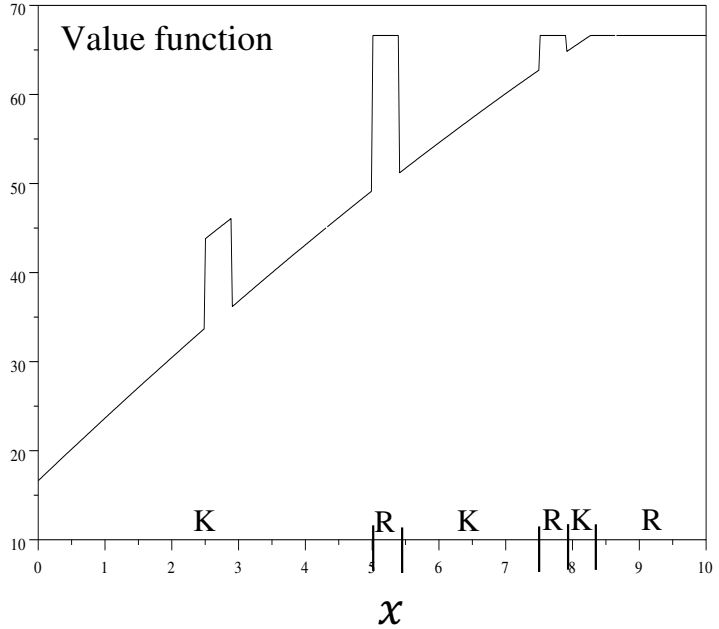
Problem: Minimize the discounted expected cost over an infinite horizon.

Optimal replacement problem

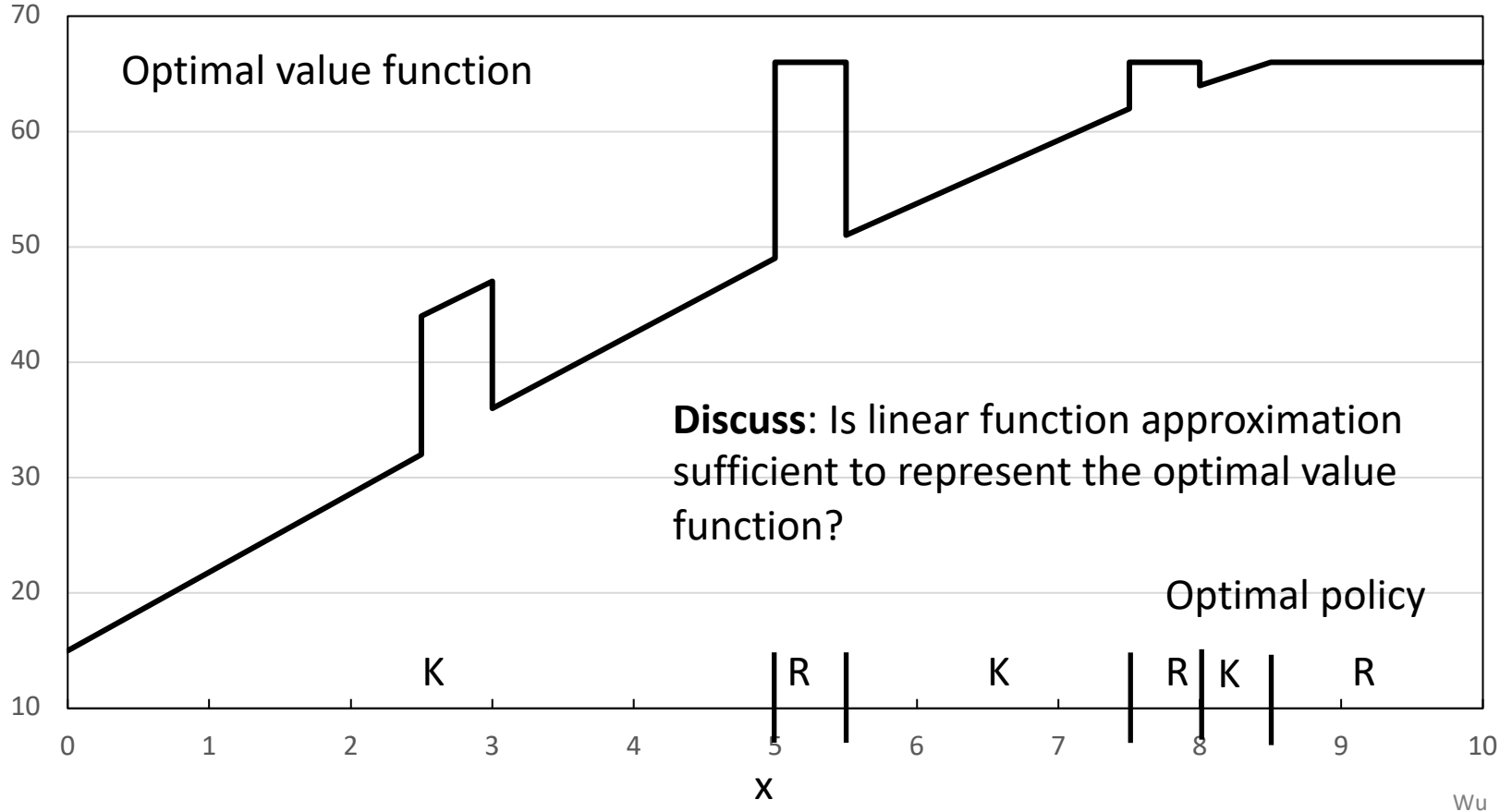
Optimal value function

$$V^*(x) = \min\left\{c(x) + \gamma \int_0^{\infty} d(y-x)V^*(y)dy,\ C + \gamma \int_0^{\infty} d(y)V^*(y)dy\right\}$$

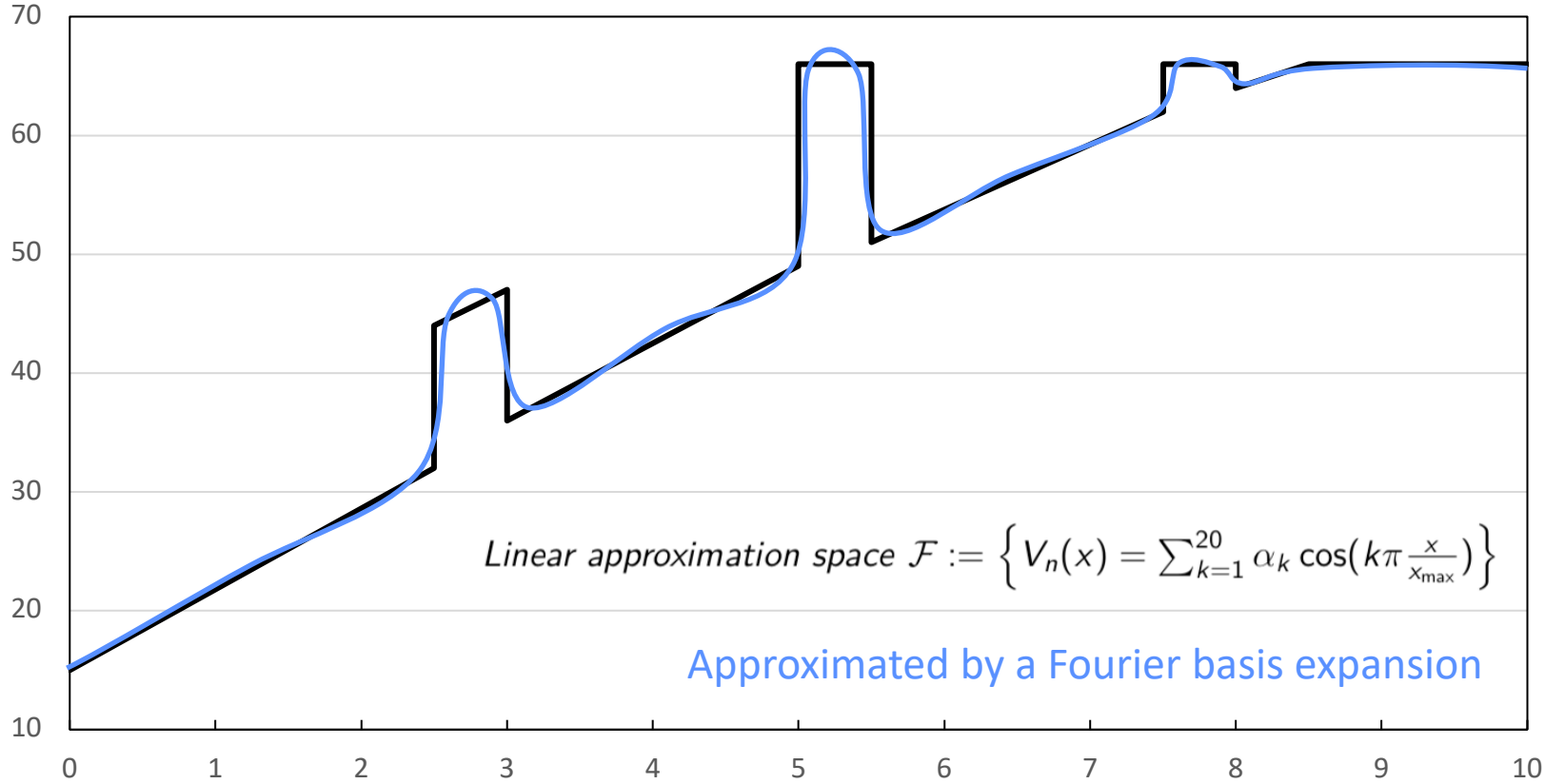




From Exact to Approximate RL



From Exact to Approximate RL



Useful representations

- Linear
  - Polynomial, Fourier basis, radial basis function (RBF) kernel features
- Nonlinear
  - Piecewise linear (e.g., triangular fundamental diagram)
  - Fully connected networks (FCNs)
- Grid-based (CNNs)
- Graph-based (GNNs)
- Attention (Physics-informed ML / Transformers)
- Combinations, e.g., Vision Transformer

Outline

1. Recap Exercise: Parking Problem (Modeling & Solving)
2. Tackling Large State Spaces: Deep reinforcement learning
3. Exploration vs Exploitation
4. Function Approximation
5. **Deep learning 101**
   a. Stochastic Gradient Descent
   b. Backpropagation
6. Deep Q Networks (DQN)
7. Policy Gradient
8. Actor Critic

Gradient descent

Main idea: iteratively minimize an (error) function.

$$\min_{\theta} f_\theta(x) = \min_{\theta} \sum_{k=0}^{N} \left(\tilde{Q}_\theta(s_k, a_k) - R_k\right)^2$$

$$\text{where } x = (s_k, a_k, r_k, s_{k+1}, R_k)_{k \in [N]}$$

Gradient descent algorithm (sketch)

1. Start with some θ
2. Improve it: $\theta' \leftarrow \theta +$ (improvement)

- (Recall) Q-learning update:

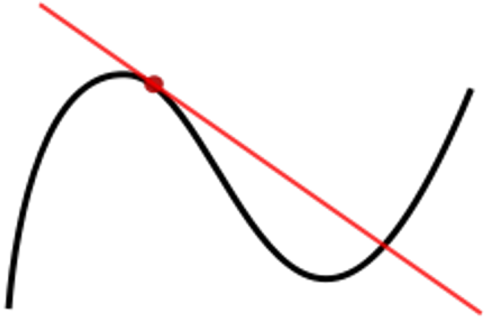
$$Q_{i+1}(s,a) = Q_i(s,a) + \eta_i \underbrace{\left(r + \gamma \max_{a'} Q_i(s', a') - Q_i(s,a)\right)}_{\text{Temporal difference (TD) error}}$$

- We will use $R_k \coloneqq r_k + \gamma \max_a Q_\theta(s_{k+1}, a)$

Recall: single-variate (1D) calculus

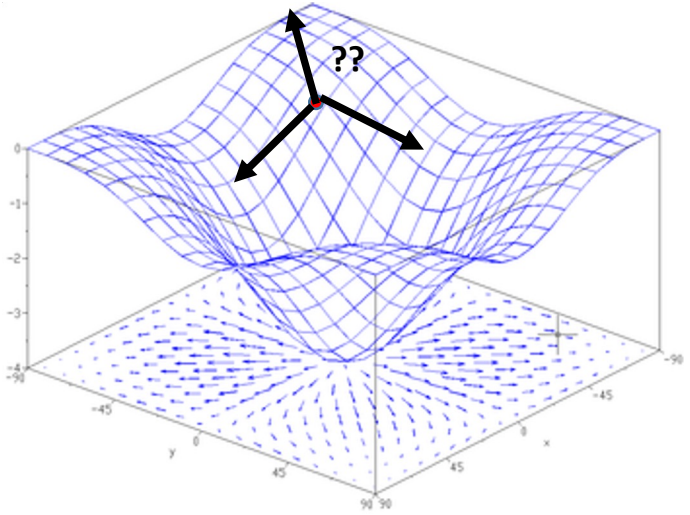
- i.e. $|\theta| = 1$
- Want: $\frac{d}{d\theta} f_\theta(x)$

  [x is the data, constants]
- Rate of change in $+\theta$ direction
- NOT: $\frac{df}{dx}$



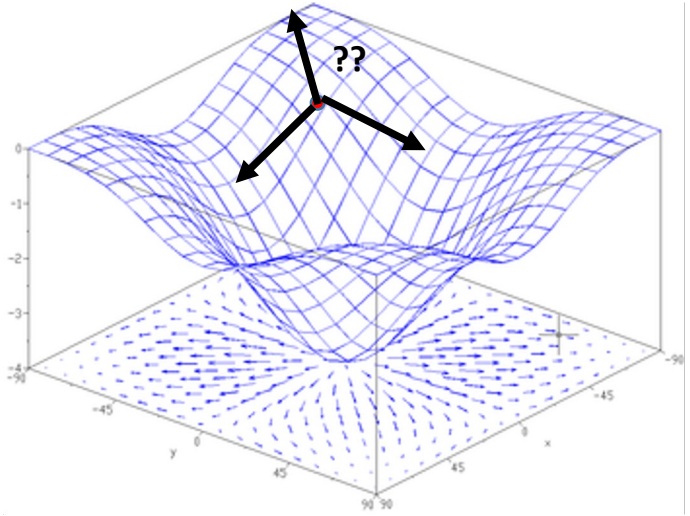
- What about multiple directions to move in?

$$|\theta| > 1$$



Multi-variate gradients

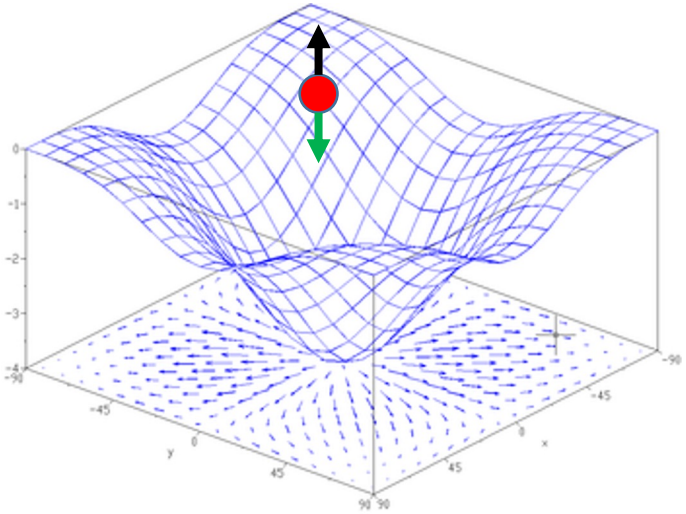
- Define: $\nabla_\theta f_\theta(x) = \begin{bmatrix} \frac{df}{d\theta_1} \\ \frac{df}{d\theta_2} \\ \vdots \\ \frac{df}{d\theta_{|\theta|}} \end{bmatrix}$



- Here, $\frac{df}{d\theta_i}$ is derivative of f by θ_i holding all other variables fixed
- The gradient is the direction of biggest increase **(REMEMBER THIS)**

Multi-variate gradients

- The gradient is the direction of biggest increase **(REMEMBER THIS)**



∇f:Direction of biggest increase

$-\nabla f$:Direction of biggest decrease

Gradient Examples

$$f(\theta_1, \theta_2) = \theta_1^2 + \theta_2^2$$

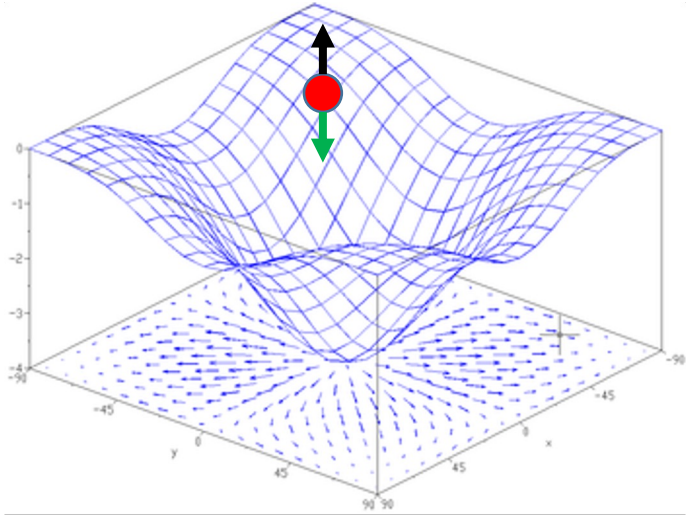
$$\nabla_\theta f(\theta_1, \theta_2) = [2\theta_1, 2\theta_2]$$

$$g(\theta_1, \theta_2) = -2\theta_1\theta_2$$

$$\nabla_\theta g(\theta_1, \theta_2) = [-2\theta_2, 2\theta_1]$$

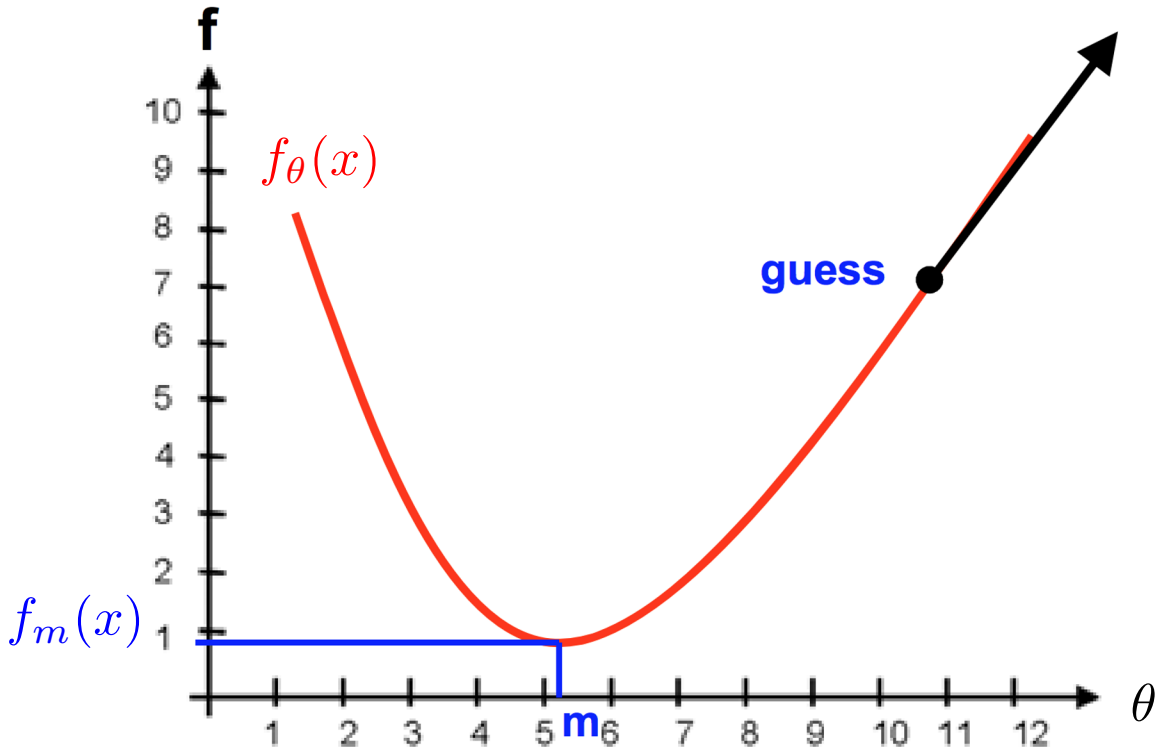
Gradient descent

- How can we find θ in $\min_{\theta} f_{\theta}(x)$?
  - Analytical solution sometimes!
  - **Follow the negative gradient**



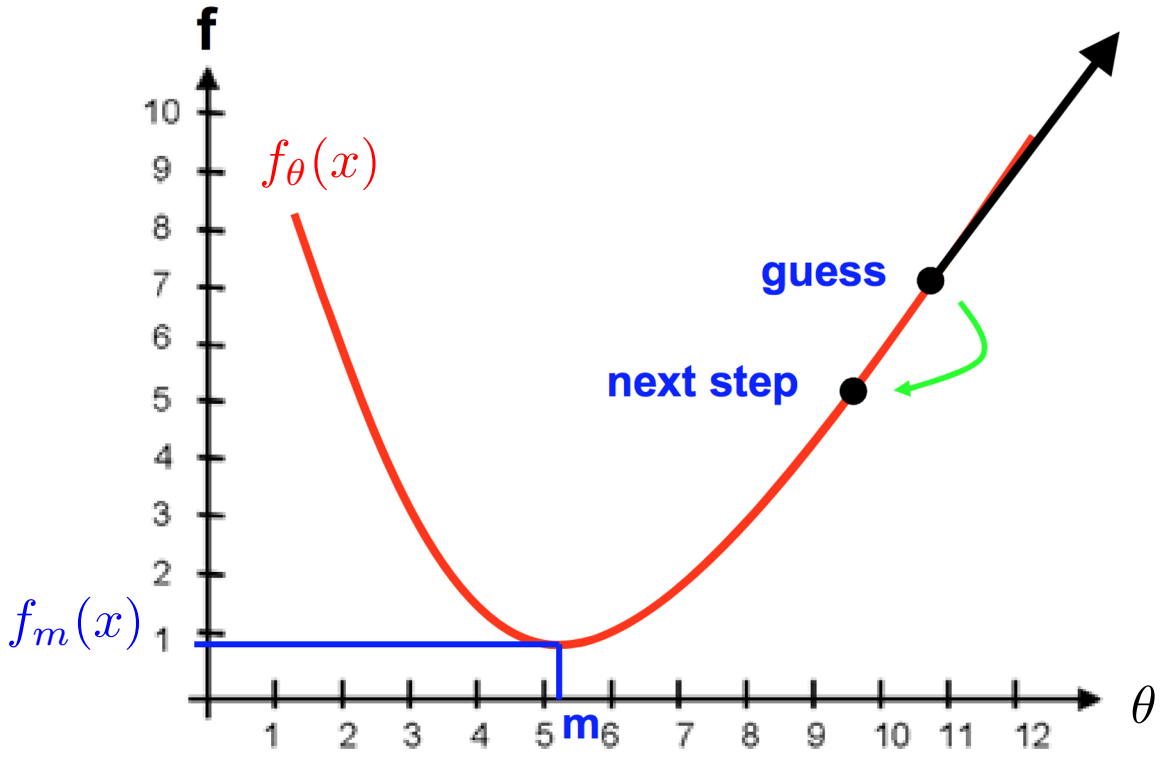
Gradient Descent

Find a minimum by following the slope:



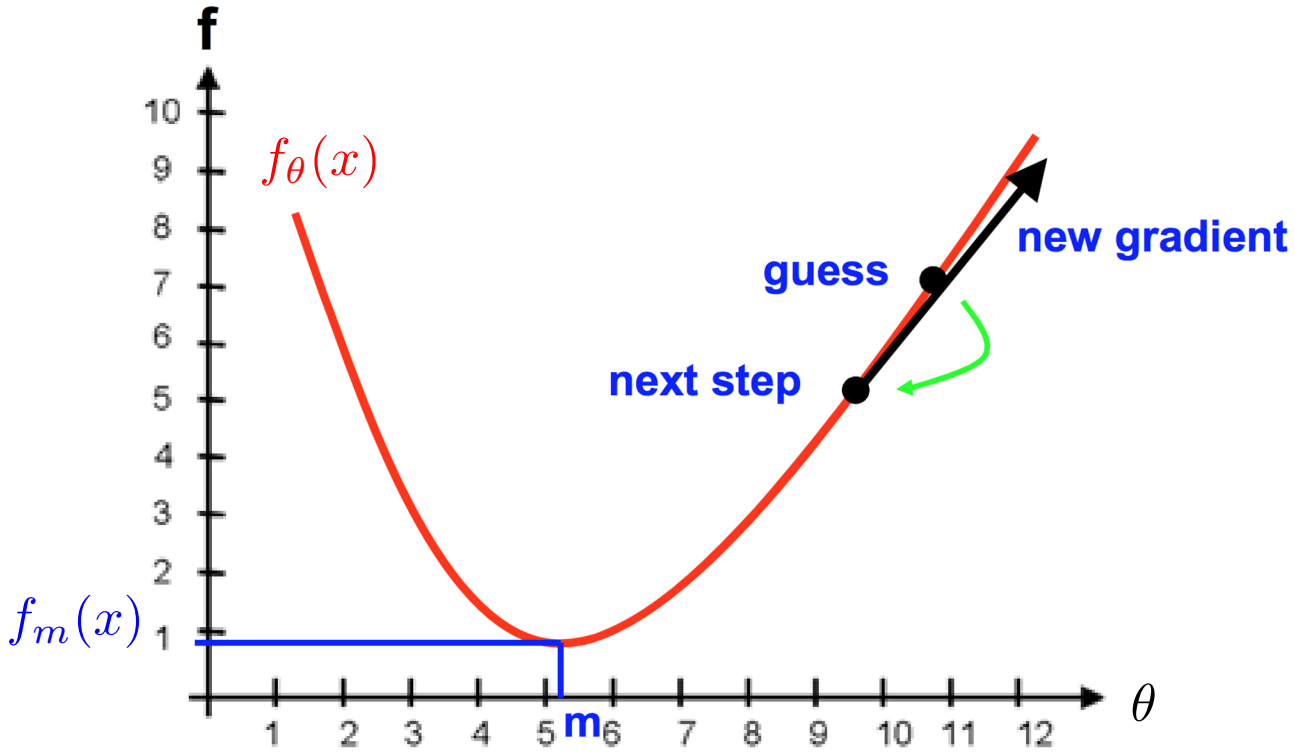
Gradient Descent

Find a minimum by following the slope:



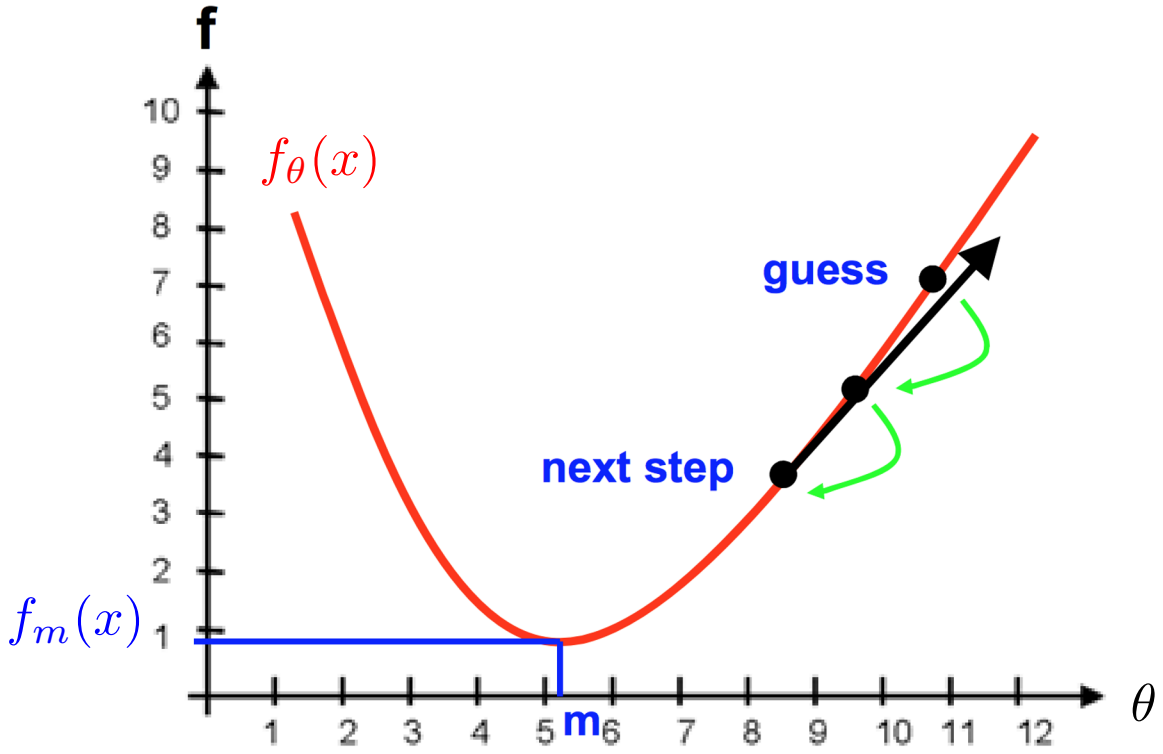
Gradient Descent

Find a minimum by following the slope:



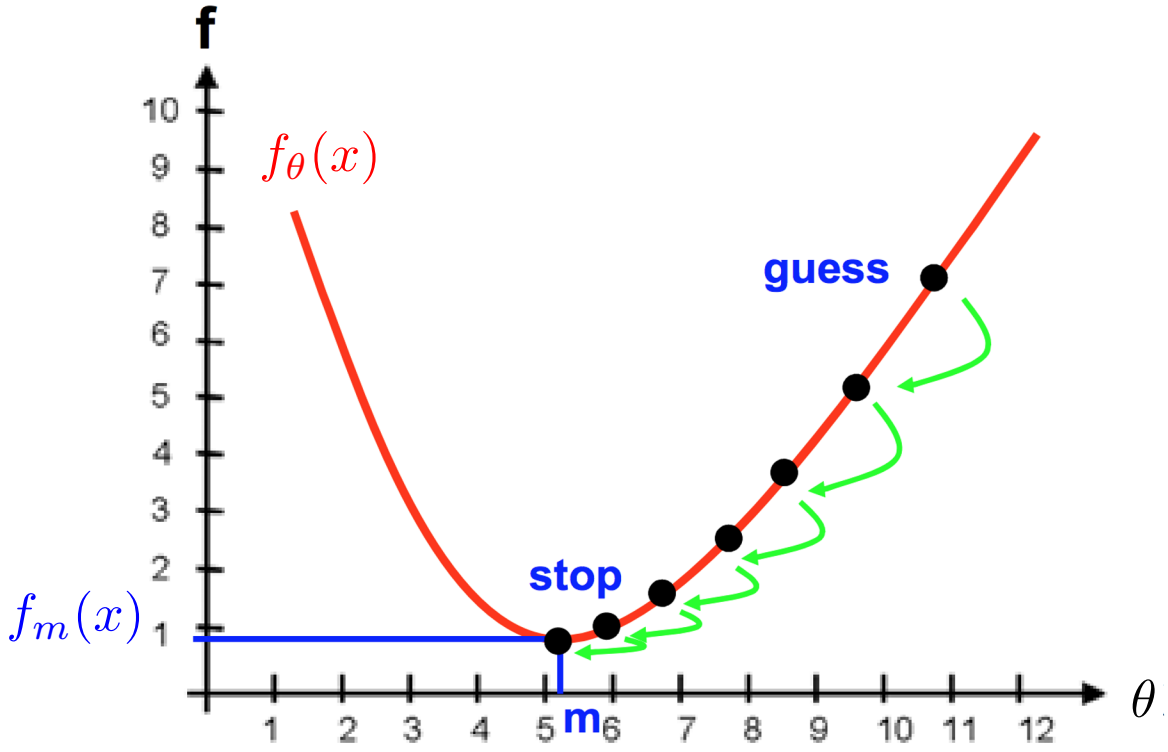
Gradient Descent

Find a minimum by following the slope:



Gradient Descent

Find a minimum by following the slope:



Gradient descent

Main idea: iteratively minimize an (error) function.

$$\min_{\theta} f_\theta(x) = \min_{\theta} \sum_{k=0}^{N} \left(\tilde{Q}_\theta(s_k, a_k) - R_k\right)^2$$

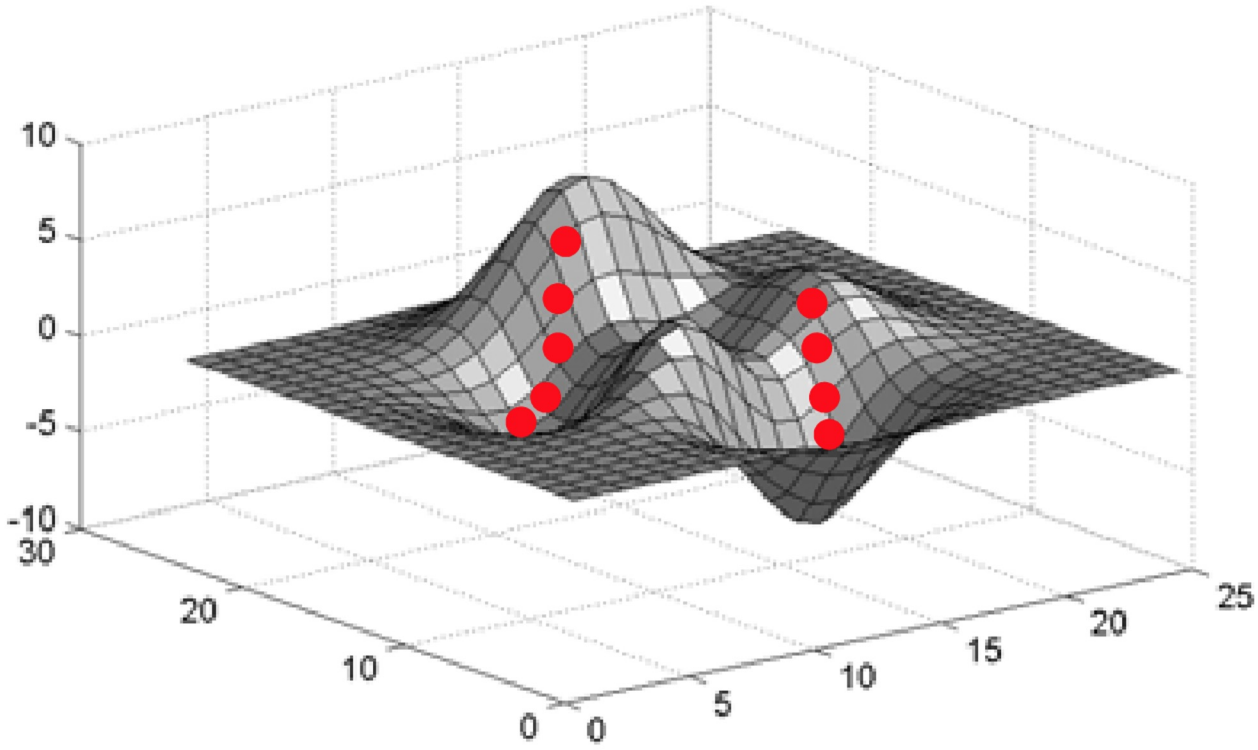
$$\text{where } x = (s_k, a_k, R_k)_{k \in [N]}$$

Gradient descent algorithm:

1. Pick a starting θ_0
2. Repeat:
    1. Compute descent direction: $-\nabla_\theta f_\theta(x)$
    2. Step in the direction: $\theta_{t+1} = \theta_t - \eta \nabla_\theta f_\theta(x)$
    3. Check if we should stop (gradient close to zero)

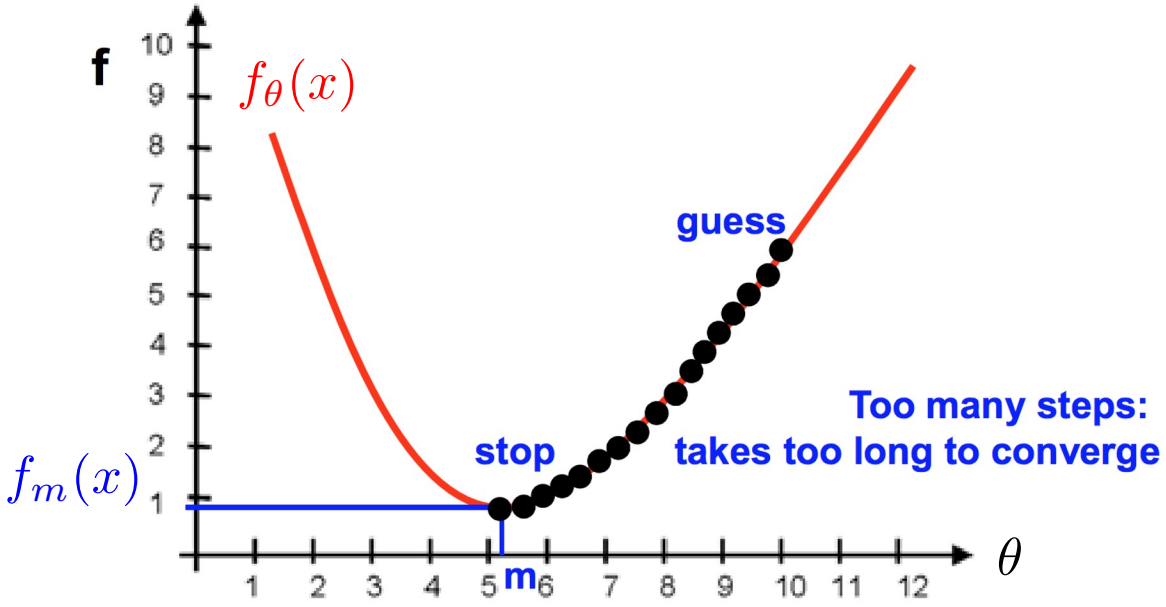
Gradient Descent

- Works just as well in higher dimensions



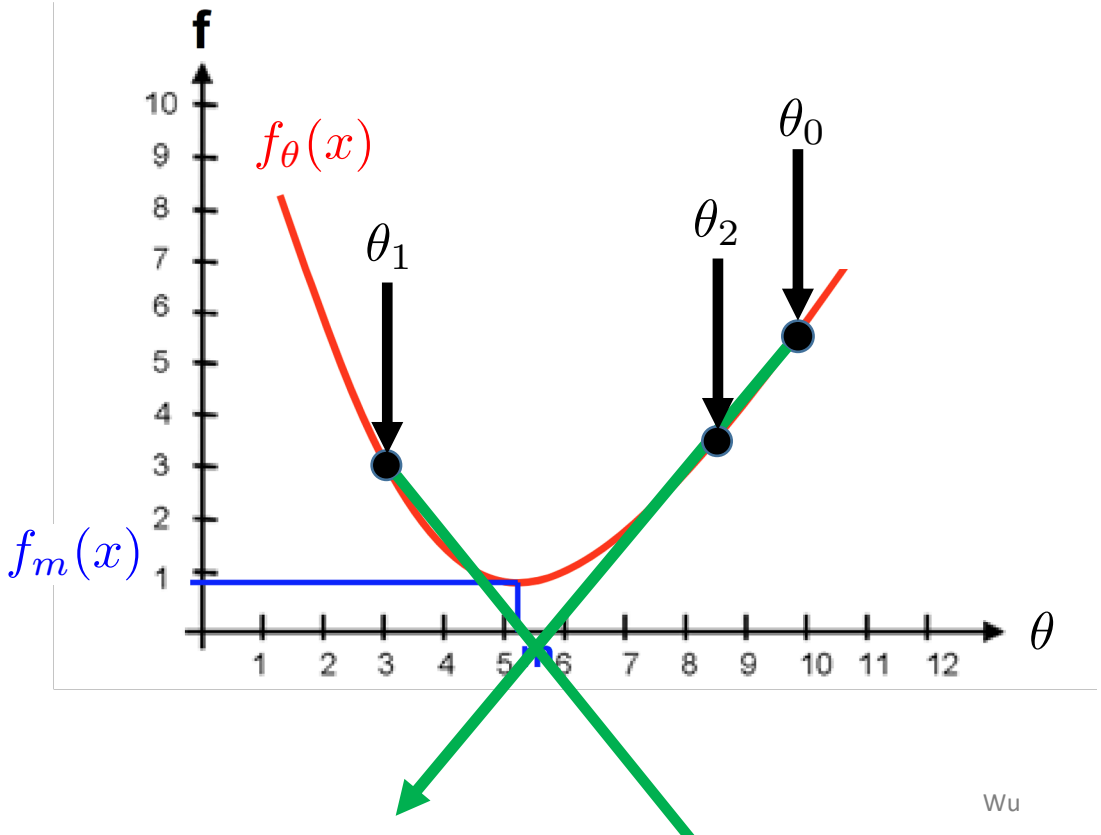
Picking a step

- Small steps: too slow!



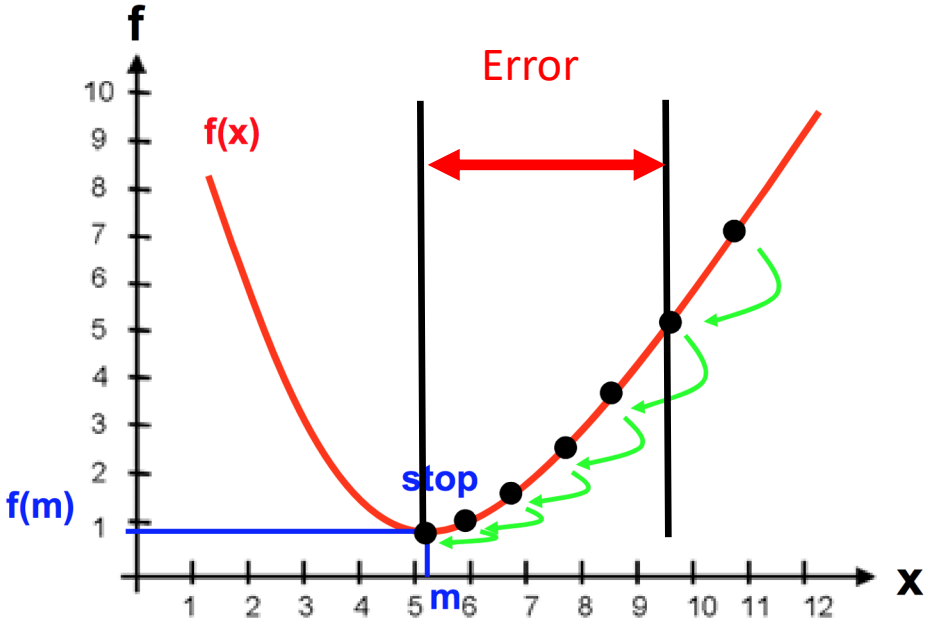
Picking a step

- Large steps: overshoots!
  - Takes a while
  - Possibly unstable! Goes to infinity



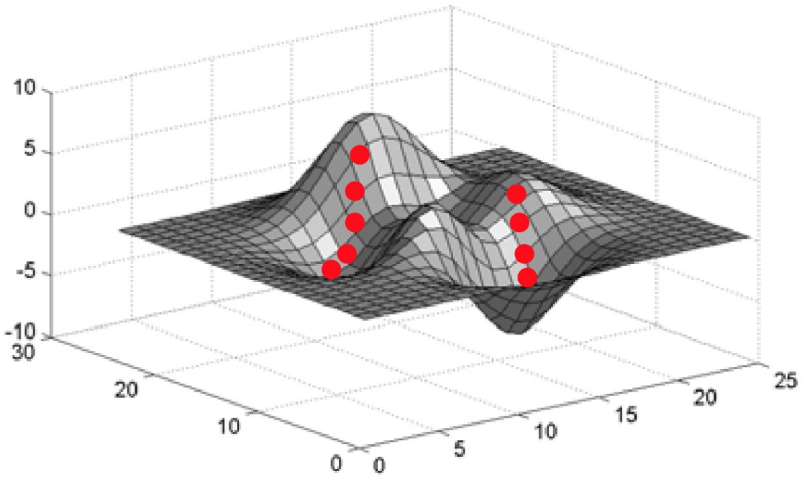
Picking a step

- In practice:
  - Many algorithms & theory to help with this
  - “Just start with .01”
  - Decrease it if too large (unstable updates)
  - Increase it if too small (slow improvement)



Picking a step

- When to stop?
  - Some options:
    - $|f(x_{t+1}) - f(x)| \leq \epsilon$
    - $|\nabla f(x_{t+1})| \leq \epsilon$
  - Could be many minima



Stochastic Gradient Descent (SGD)

- Dataset of size D: $\{(x^1, y^1), (x^2, y^2), \ldots, (x^D, y^D)\}$
- Goal: Fit linear model: $f_\theta(x) = \sum_{i=0}^{n} \theta_i x_i$
- Loss: $L(f(x;\theta), y) = \frac{1}{D}\sum_{j=1}^{D}\left(f_\theta\left(x^j\right) - j\right)^2$
- Suppose we have 10000 samples
- Do we need to use all of them for the gradient?

Stochastic Gradient Descent (SGD)

- Goal: Fit linear model: $f_\theta(x) = \sum_{i=0}^{n} \theta_i x_i$
- Loss: $L(f(x;\theta), y) = \frac{1}{D}\sum_{j=1}^{D}\left(f_\theta\left(x^j\right) - y_j\right)^2$
- Gradient: $\nabla L = \frac{2}{D}\sum_{j=1}^{D}\left(f_\theta\left(x^j\right) - y_j\right)\frac{\partial f_\theta(x^j)}{\partial \theta}$
- Each sample “contributes” one gradient
- Redundancy?
- **Idea:** Using a subset of samples is good enough!

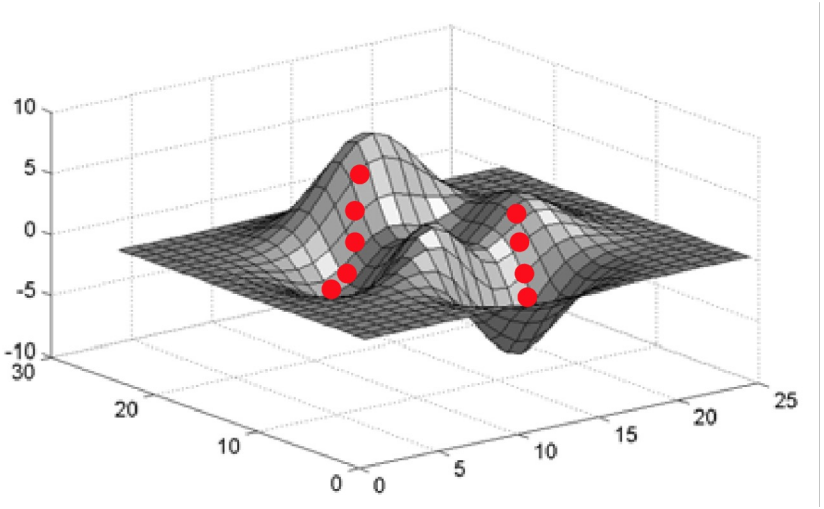
Stochastic Gradient Descent (SGD)

- **Idea:** Using a subset of samples is good enough!

- **Algorithm:**
  - **Repeat:**
    - Sample a subset of size D': $\{(x^{1'}, y^{1'}), (x^{2'}, y^{2'}), \dots, (x^{D'}, y^{D'})\}$
    - Compute the gradient on D'
    - Descend using the gradient

Why SGD?

- Faster
- Noisy
  - Good if many local minima!
  - Can jump out of a shallow minimum



Gradient descent with deep neural networks

Main idea: iteratively minimize an (error) function.

$$\min_\theta f_\theta(x) = \min_\theta \sum_{k=0}^{N} \left(\tilde{Q}_\theta(s_k, a_k) - R_k\right)^2$$

where $x = (s_k, a_k, R_k)_{k \in [N]}$

Gradient descent algorithm:

1. Pick a starting θ_0
2. Repeat:
   1. Compute descent direction: $-\nabla_\theta f_\theta(x)$
   2. Step in the direction: $\theta_{t+1} = \theta_t - \eta \nabla_\theta f_\theta(x)$
   3. Check if we should stop

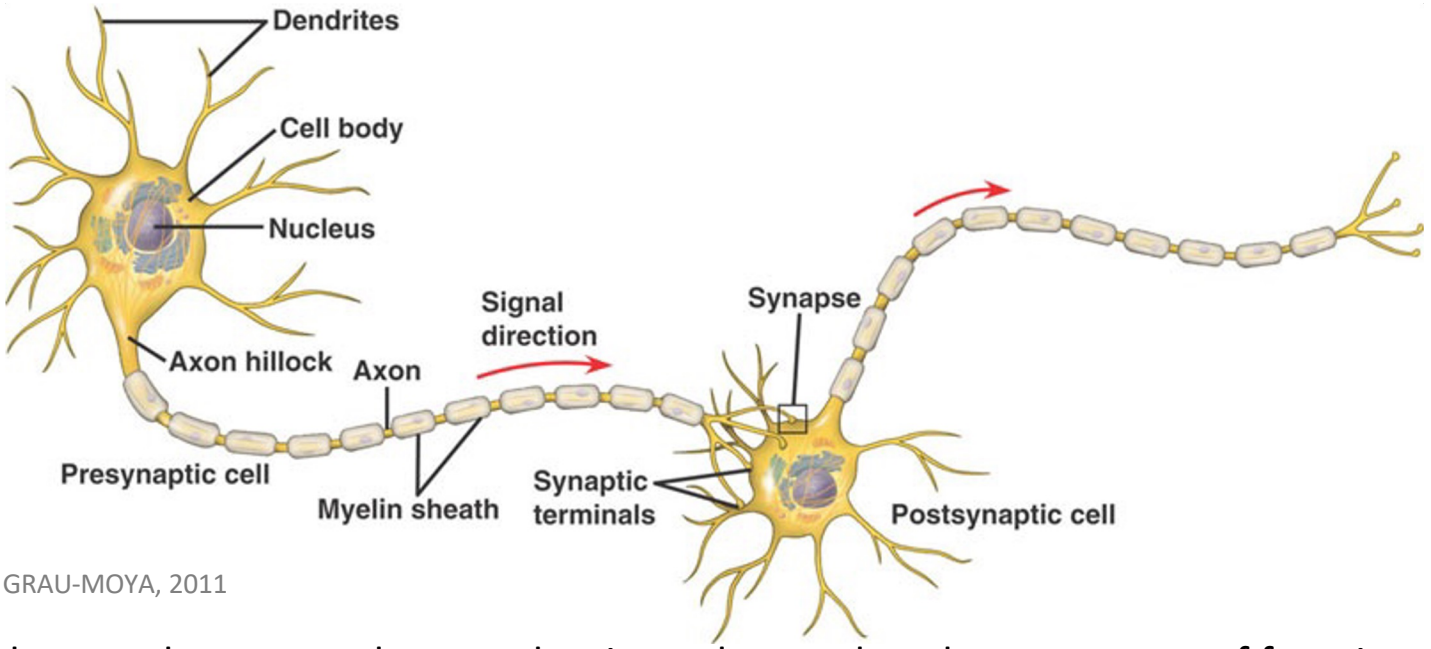
$$\nabla_\theta f_\theta(x) = 2 \sum_{k=0}^{N} \left(\tilde{Q}_\theta(s_k, a_k) - R_k\right) \nabla_\theta \tilde{Q}_\theta(s_k, a_k)$$

???

For complex problems, we wish to leverage advanced function approximation, i.e., deep neural networks. How to update those?

Biological Inspiration

Neuron

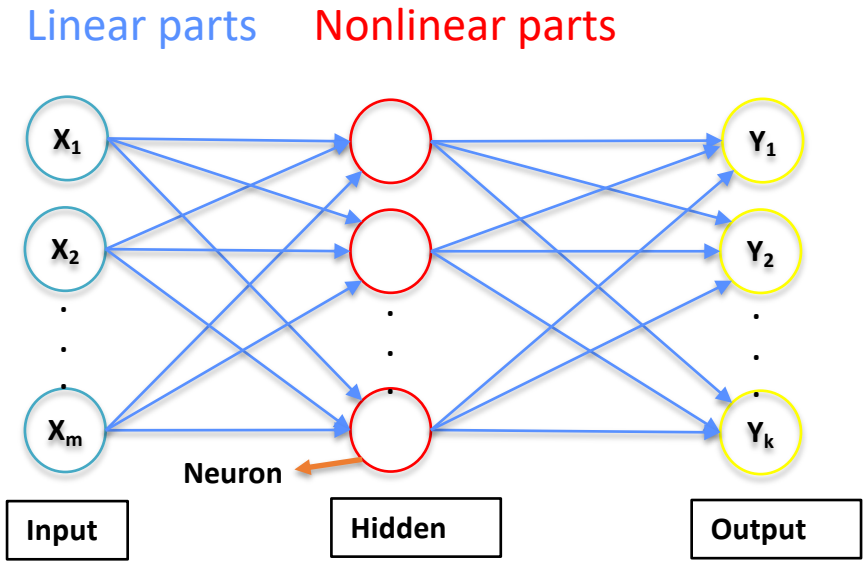


Source: GRAU-MOYA, 2011

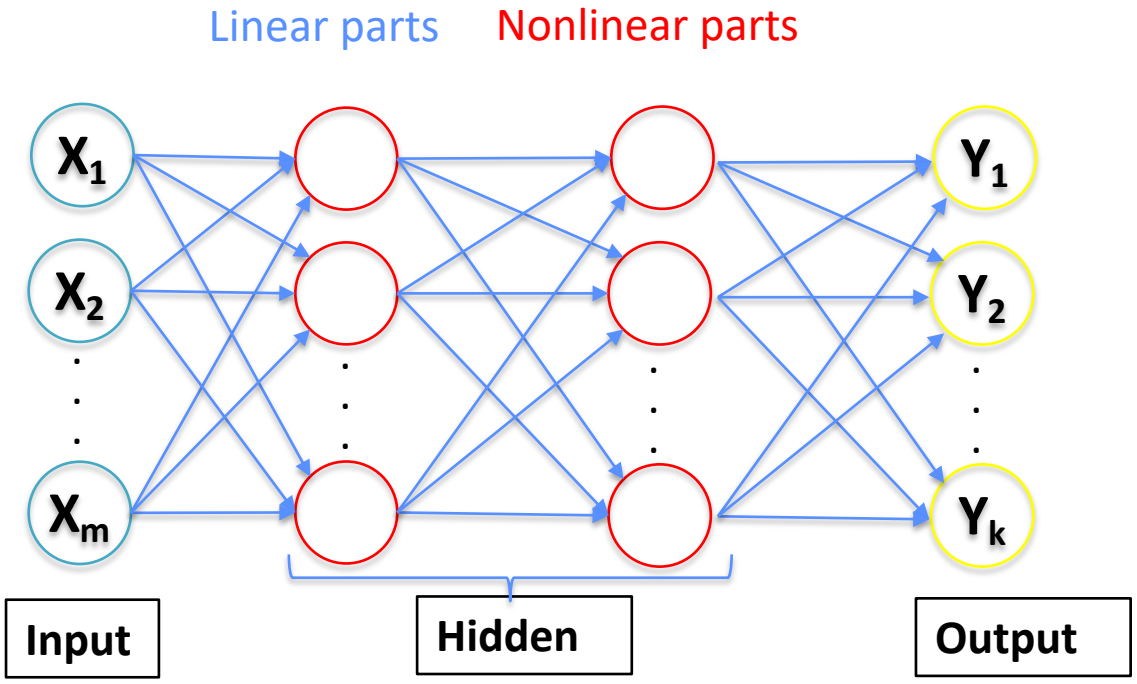
- Neural networks are mostly named so in analogy rather than exactness of function

Single Hidden Layer NN

- It was shown that a sigmoid network with one hidden layer of unbounded size is a universal function approximator.
- NN with single hidden layer (unbounded size) can represent any decision boundary in a classification problem.



Multiple Hidden Layers

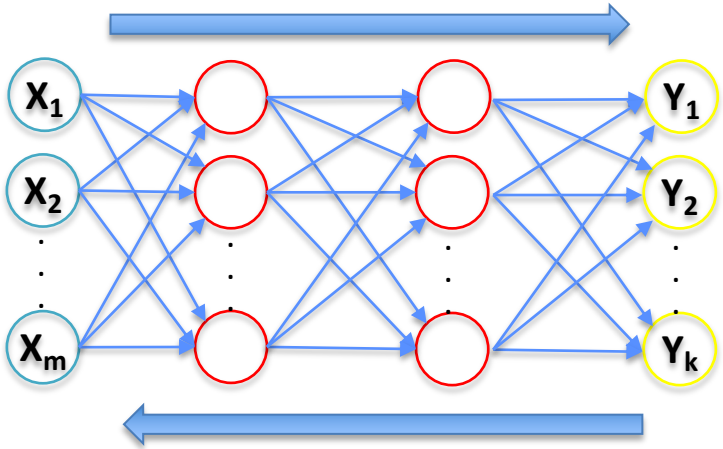


How to estimate the weights and biases of our NN?

Training a NN: Backpropagation Algorithm (Backprop)

0- Initialize the weights and biases

1- Forward pass



2- Compute the error

3- Backward pass, compute the gradients

4- Update the weights

Automatic differentiation in PyTorch

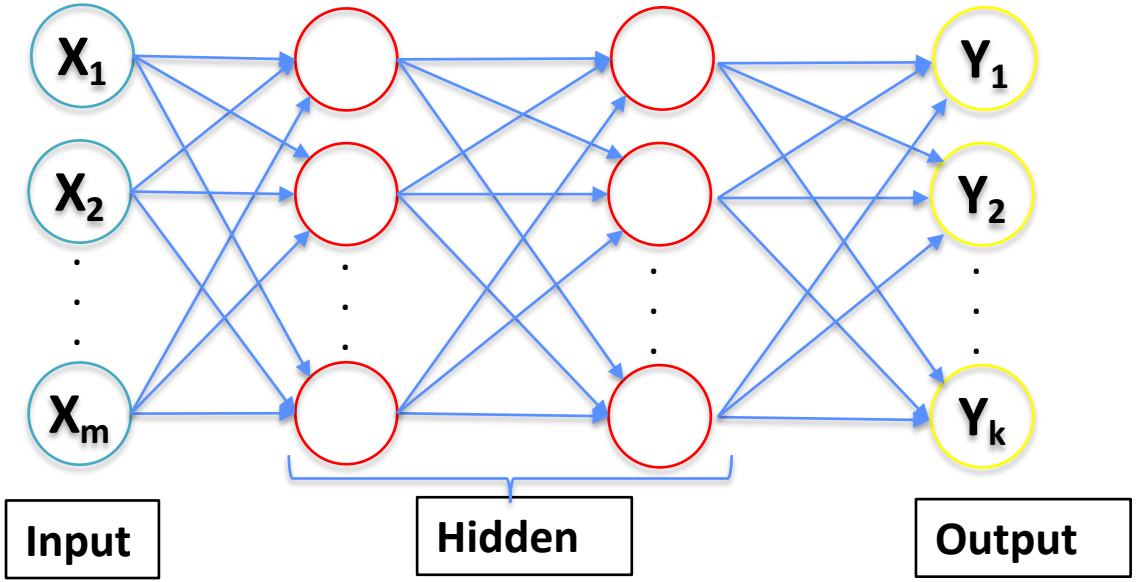
- PyTorch implements backprop via `backwards()` function

```
import torch
import torch.nn as nn

# Define a simple model
model = nn.Linear(2, 1)  # Single layer: y = w * x + b
loss_fn = nn.MSELoss()

# Input and target
x = torch.tensor([[1.0, 2.0], [3.0, 4.0]], requires_grad=True)
y = torch.tensor([[5.0], [7.0]])

# Forward pass
pred = model(x)
loss = loss_fn(pred, y)

# Backward pass
loss.backward()

# Gradients
print("Weight gradient:", model.weight.grad)
print("Bias gradient:", model.bias.grad)
```

Credit: Noel Benji

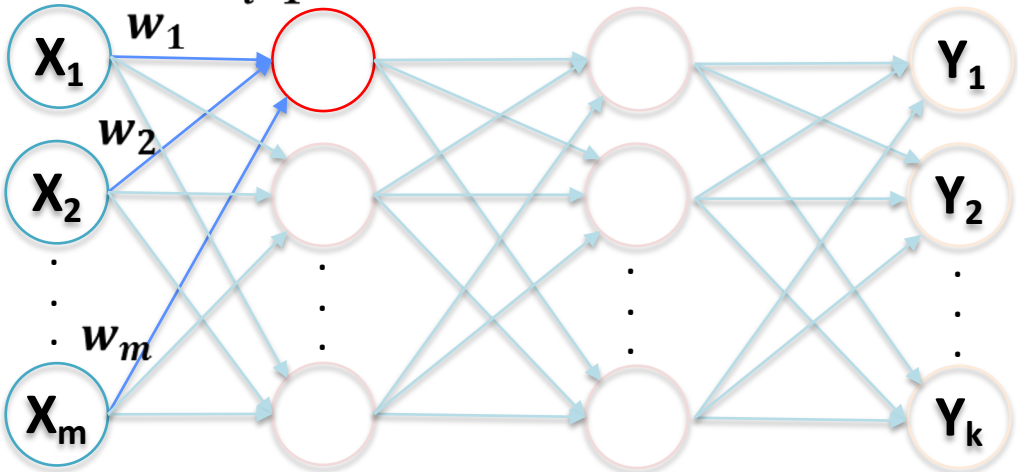
Forward Pass

Objective: compute the output values and the activations of hidden nodes



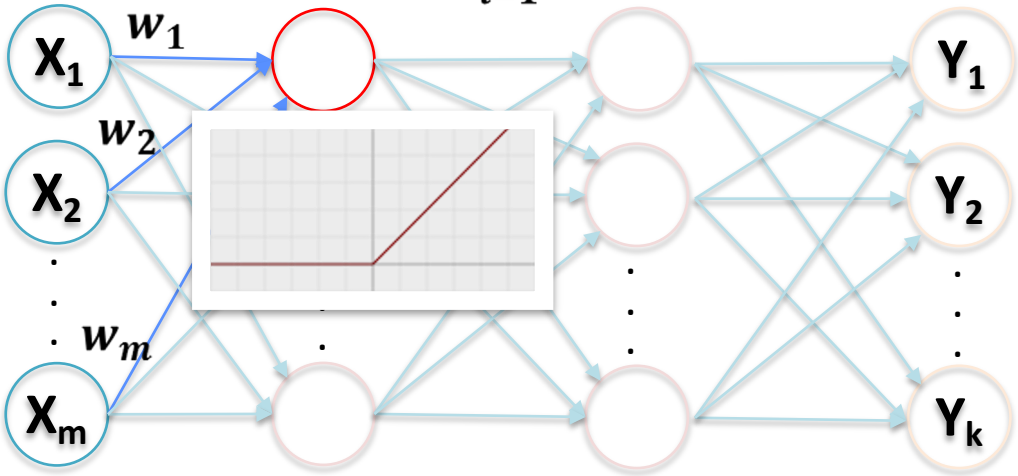
Forward Pass

$$z = \sum_{i=1}^{m} w_i x_i + b$$



Forward Pass

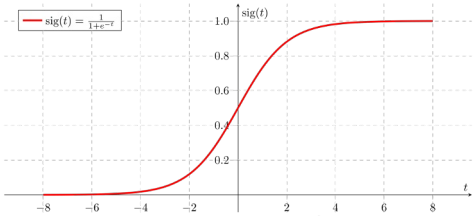
$$a = \varphi(z) = \varphi(\sum_{i=1}^{m} w_i x_i + b)$$



Activation Functions

- Introduce non-linear properties to hidden and output layers

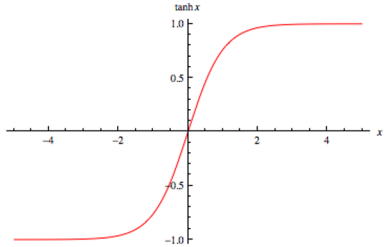
Sigmoid



$$\varphi(u) = \frac{1}{1+e^{-u}}$$

- Historically popular since they have some nice properties (i.e. smooth)
- Saturated neurons kill the gradients

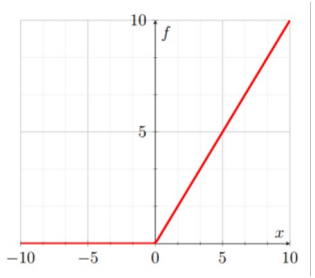
tanh



$$\varphi(u) = \frac{e^{u}-e^{-u}}{e^{u}+e^{-u}}$$

- Saturated neurons kill the gradients

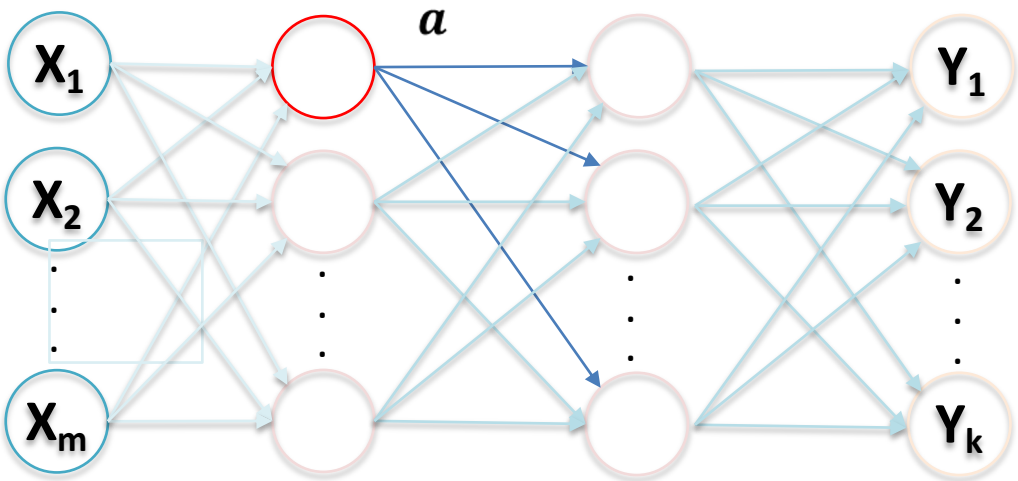
ReLU



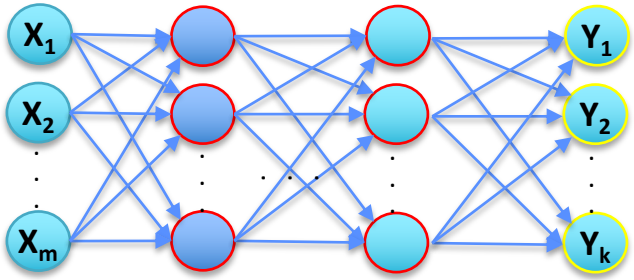
$$\varphi(u) = \max(0, u)$$

- Does not saturate
- Computationally efficient

Forward Pass



Compute the error



N: Sample size

Cost:

Sum of Squared Error:

$$C = \frac{1}{2}\sum_{k=1}^{K}\sum_{n=1}^{N}(\hat{y}_{nk} - y_{nk})^2$$

Cross-Entropy:

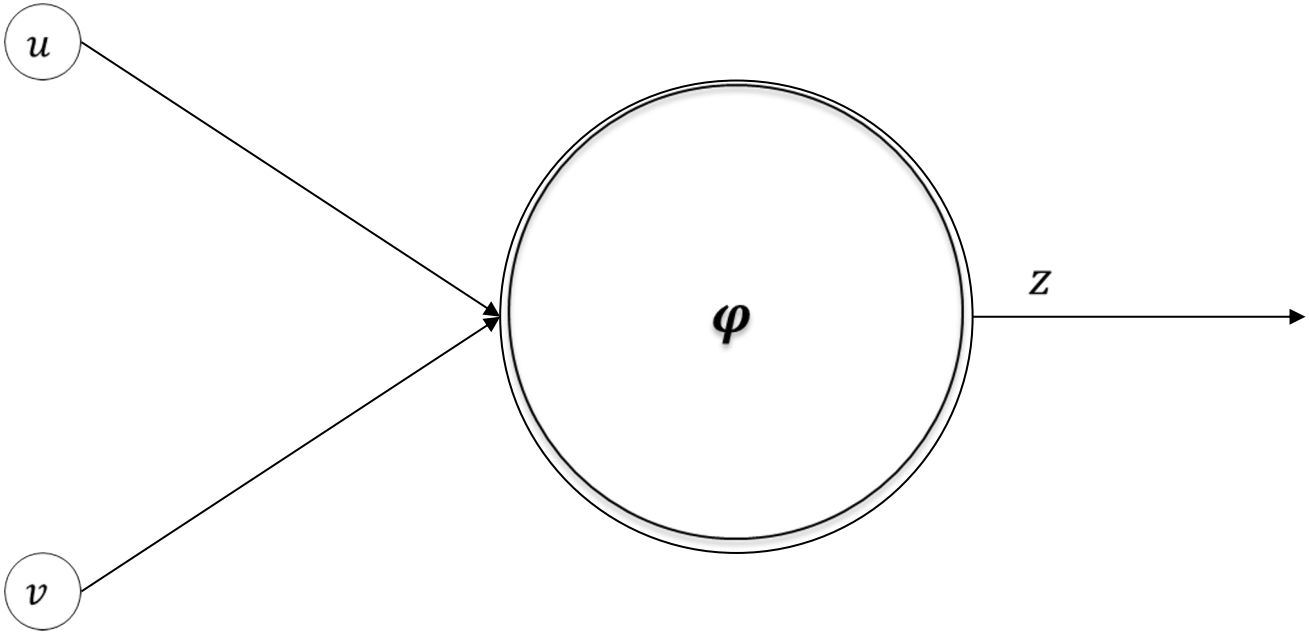
$$C = -\frac{1}{2}\sum_{k=1}^{K}\sum_{n=1}^{N} y_{nk}\log\hat{y}_{nk}$$

For more on cross-entropy and its relation to maximum likelihood estimation:
https://www.quora.com/Whats-an-intuitive-way-to-think-of-cross-entropy

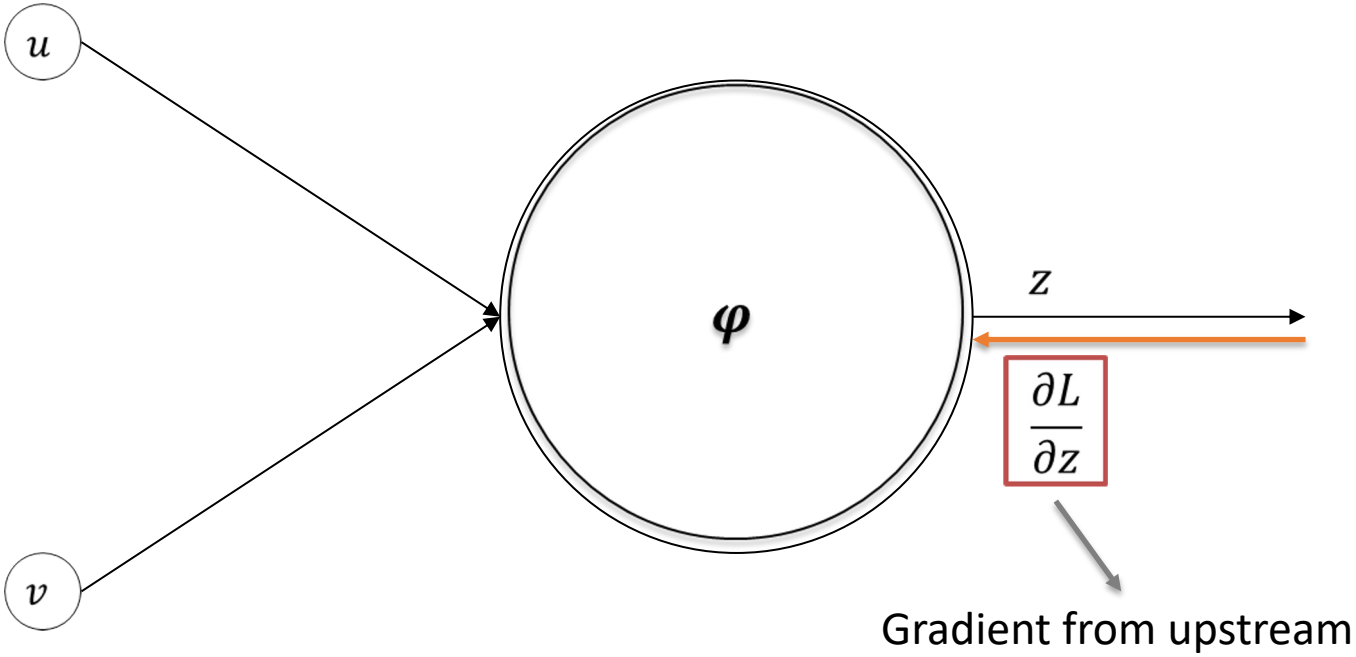
Backpropagation

- Objective: compute the gradients for the weights and the biases
- The gradients will be used in gradient descent or stochastic gradient descent

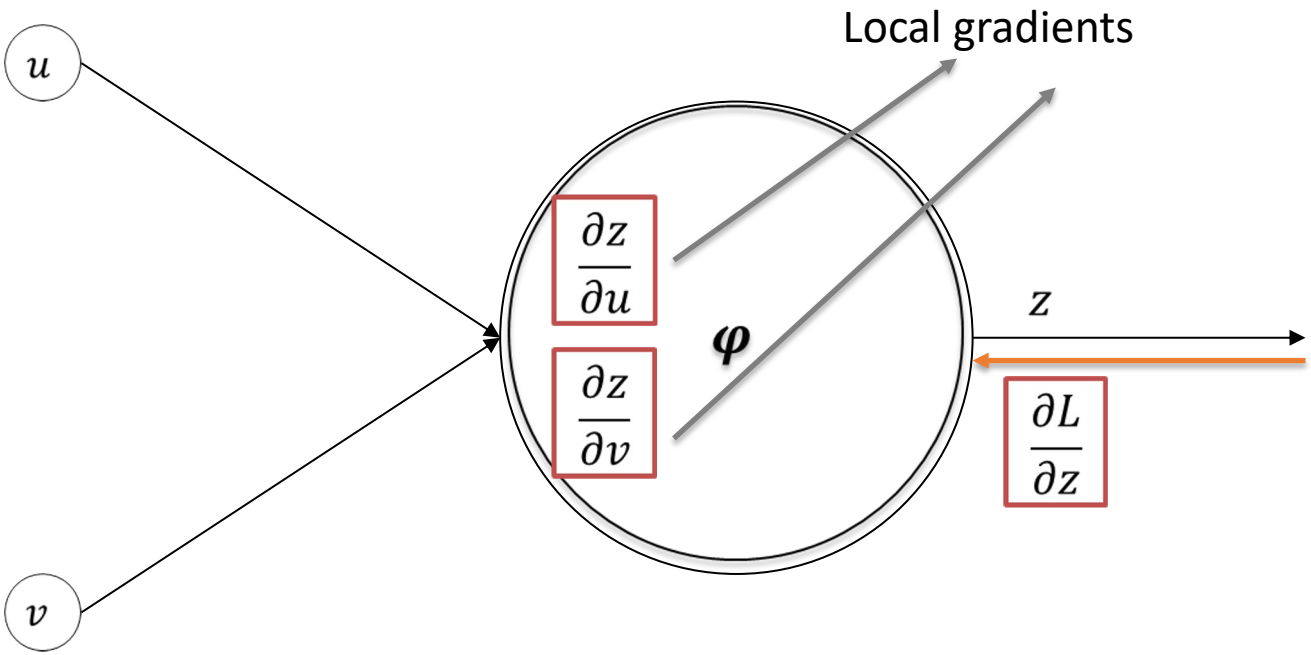
Gradients



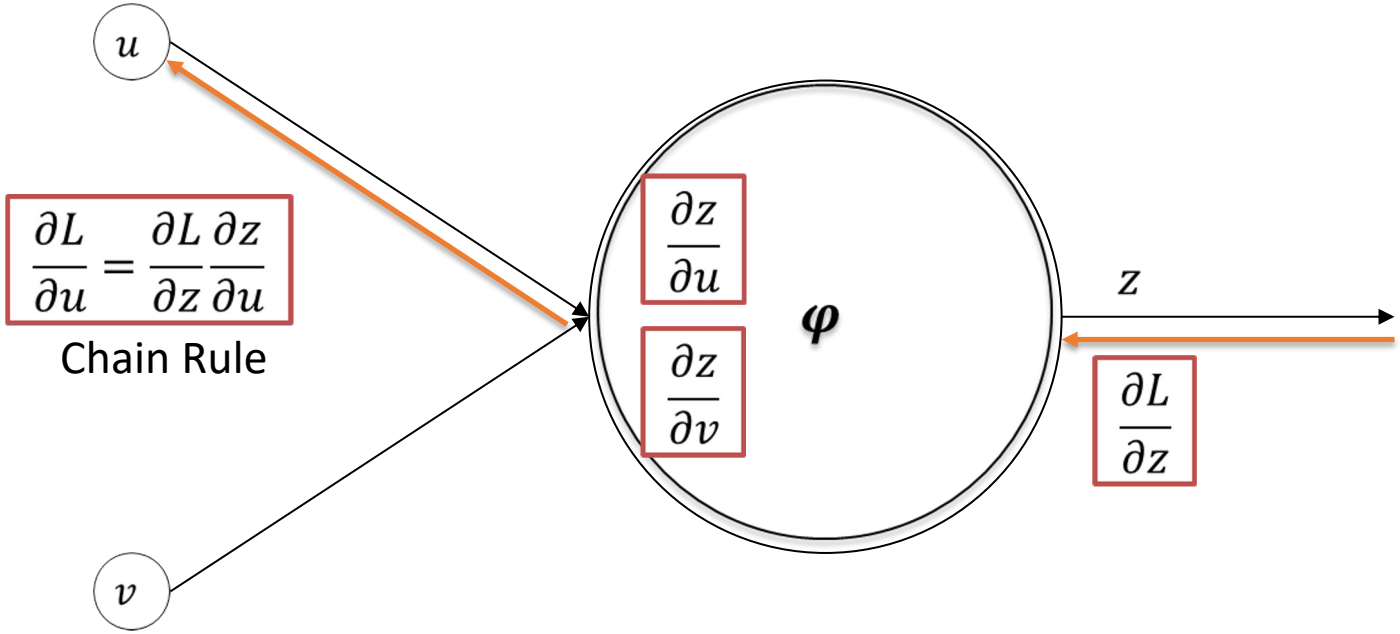
Gradients



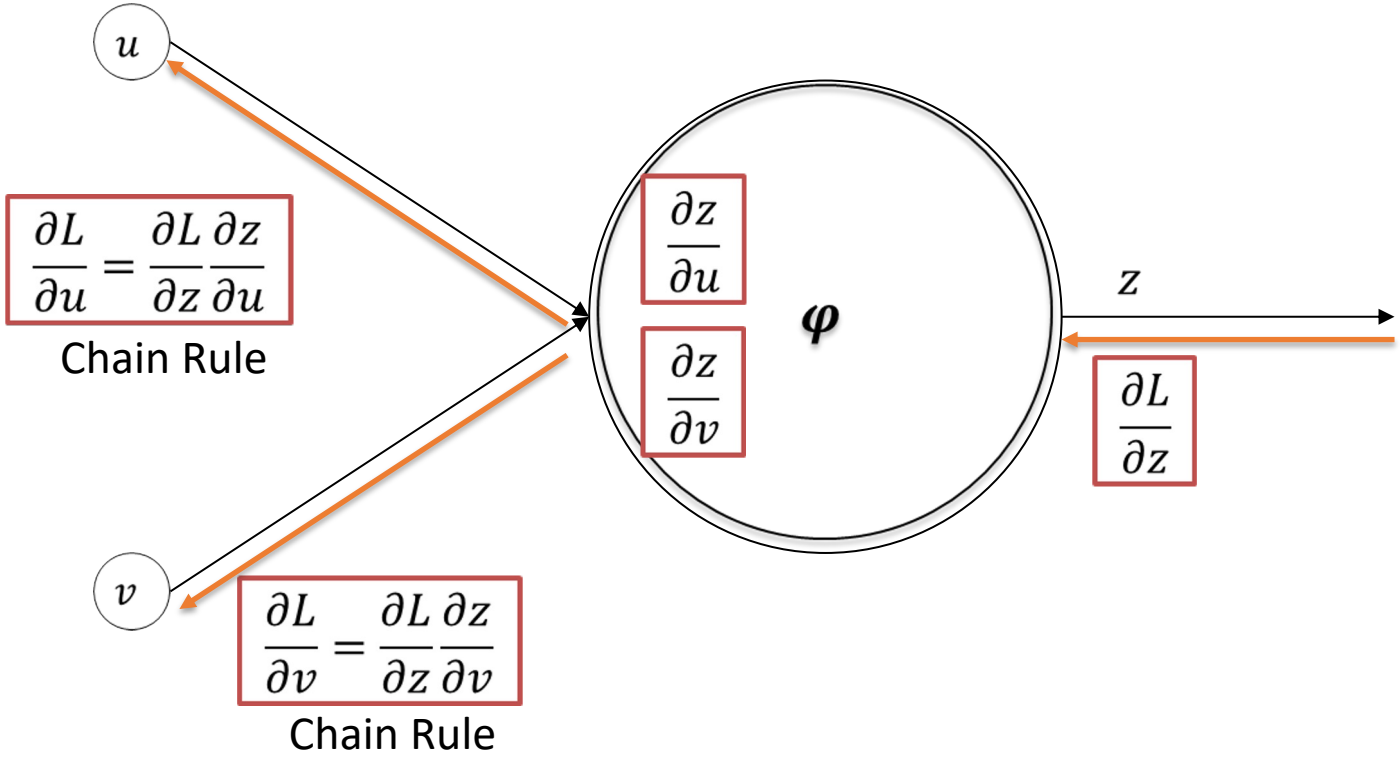
Gradients


Local gradients
u
v
$\frac{\partial z}{\partial u}$
$\frac{\partial z}{\partial v}$
φ
z
$\frac{\partial L}{\partial z}$

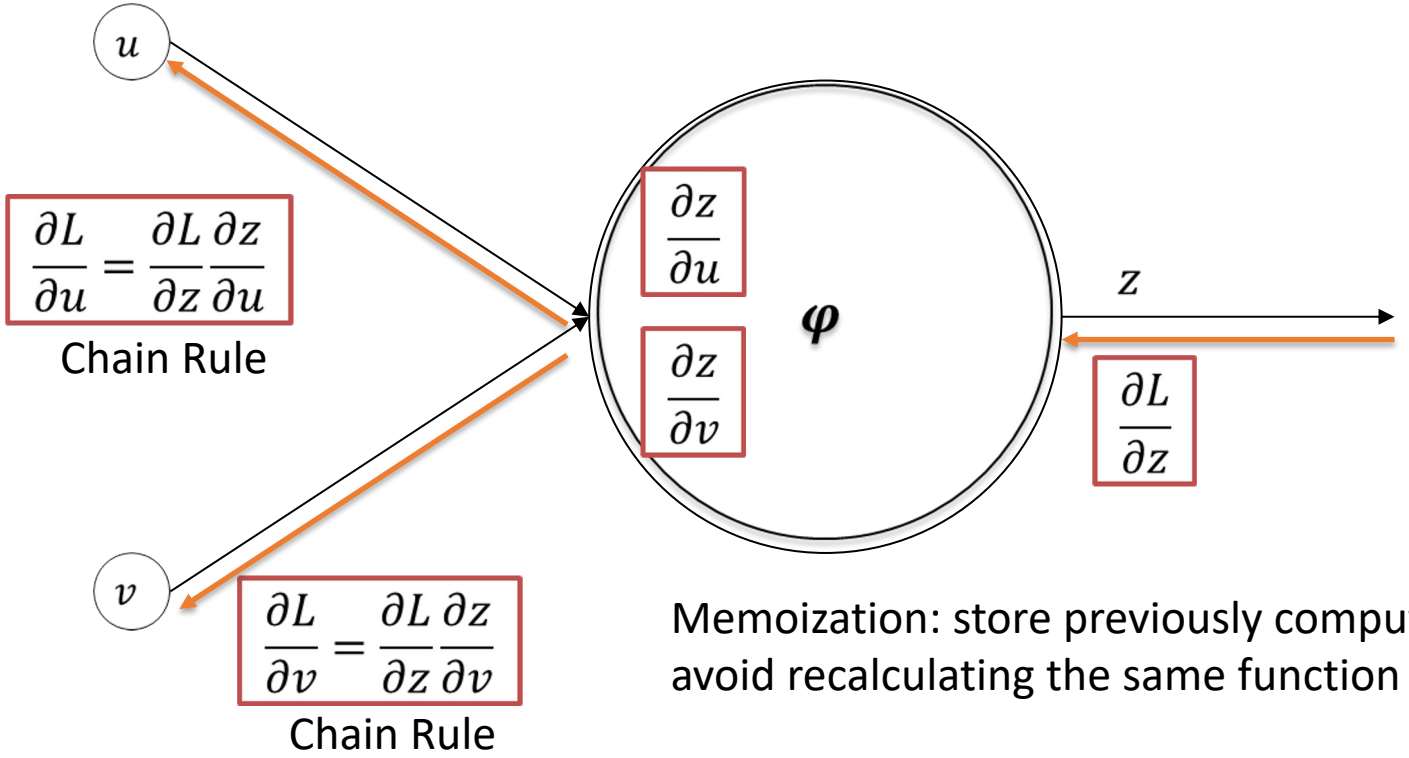
Gradients


u
v
$\frac{\partial L}{\partial u} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial u}$
Chain Rule
$\frac{\partial z}{\partial u}$
$\frac{\partial z}{\partial v}$
φ
z
$\frac{\partial L}{\partial z}$

Gradients

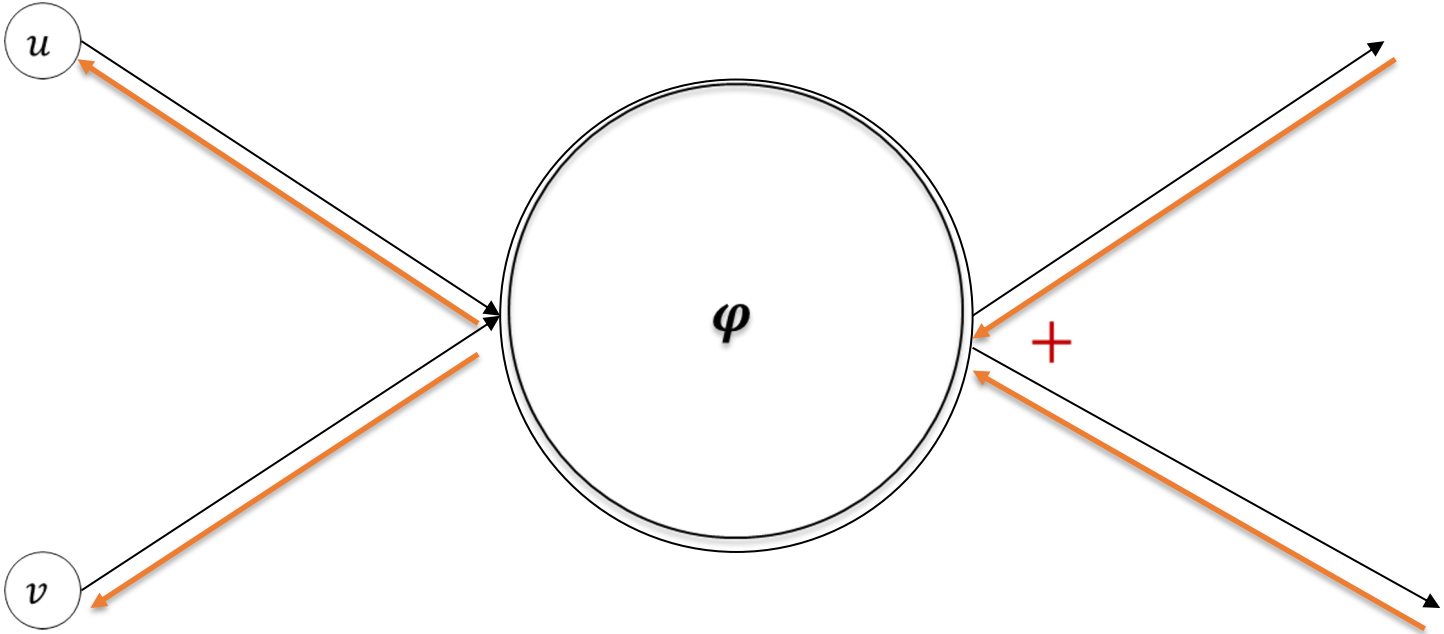

u
v
φ
z
$\frac{\partial z}{\partial u}$
$\frac{\partial z}{\partial v}$
$\frac{\partial L}{\partial z}$
$\frac{\partial L}{\partial u} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial u}$
Chain Rule
$\frac{\partial L}{\partial v} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial v}$
Chain Rule

Gradients



Memoization: store previously computed results to avoid recalculating the same function

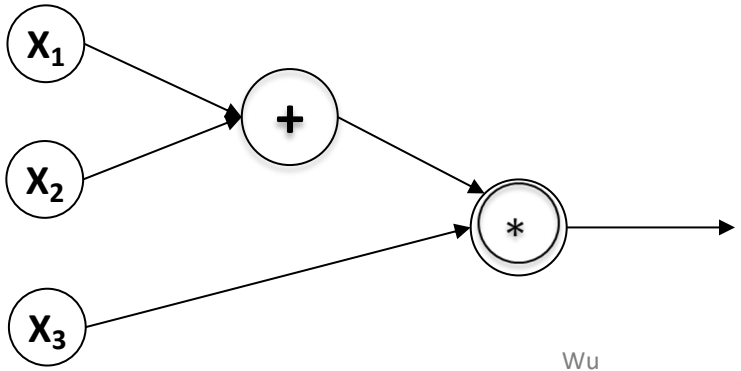
Gradients Accumulate



Computational Graph

- Represents a model as a directed graph expressing a sequence of computational steps
- Each step represents an operation which produces some output as a function of its inputs
- Complex models can be formed by simple functions

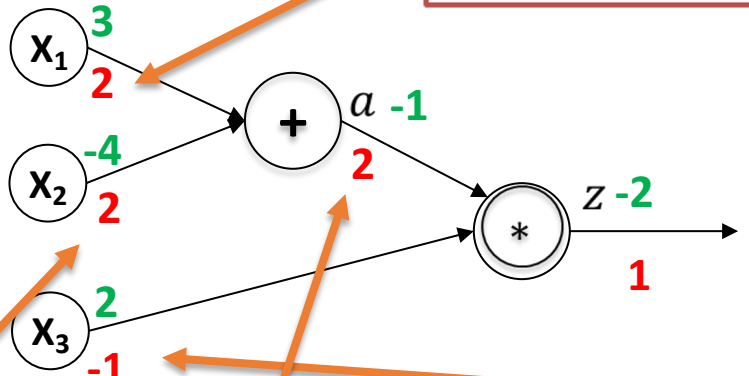
- Example: $z = f(x_1, x_2, x_3) = (x_1 + x_2) * x_3$



Simple Example

- $z = f(x_1, x_2, x_3) = (x_1 + x_2) * x_3$
- Assume: $x_1 = 3, x_2 = -4, x_3 = 2$
- Need: $\frac{\partial z}{\partial x_1}, \frac{\partial z}{\partial x_2}, \frac{\partial z}{\partial x_3}$

- $a = x_1 + x_2, \ \frac{\partial a}{\partial x_1} = 1, \frac{\partial a}{\partial x_2} = 1$
- $z = a * x_3, \ \frac{\partial z}{\partial x_3} = a, \frac{\partial z}{\partial a} = x_3$



$$\frac{\partial z}{\partial x_1} = x_3 * 1 = 2$$

$$\frac{\partial z}{\partial x_2} = x_3 * 1 = 2$$

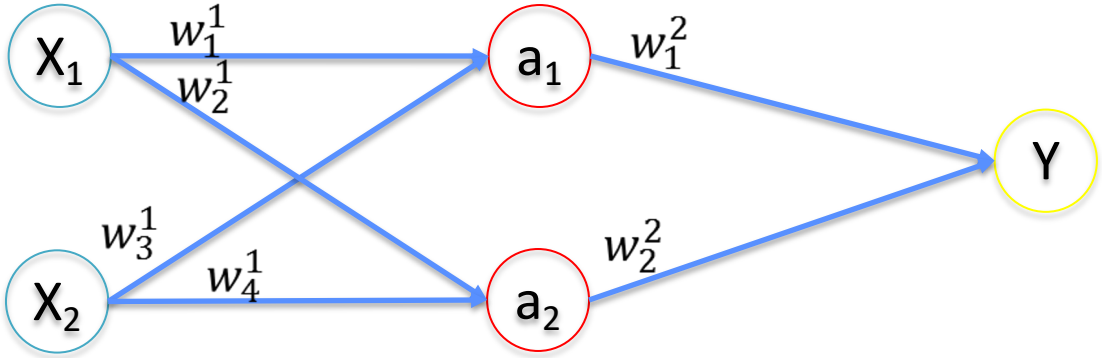
$$\frac{\partial z}{\partial a} = x_3 = 2$$

$$\frac{\partial z}{\partial x_3} = a = -1$$

$$\frac{\partial z}{\partial x_2} = \frac{\partial z}{\partial a} * \frac{\partial a}{\partial x_2}$$

Chain Rule

Example: neural network value function w/ 1 data point



- Assume: $X = \begin{bmatrix} 3 \\ -4 \end{bmatrix}, Y = 1.5, w^1 = \begin{bmatrix} 0.3 \\ 0.9 \\ 0.7 \\ 0.6 \end{bmatrix}, w^2 = \begin{bmatrix} 0.5 \\ 0.8 \end{bmatrix}, b^1 = \begin{bmatrix} 1.5 \\ 1.5 \end{bmatrix}, b^2 = [-2]$, Activation = Sigmoid
- $z_1 = w_1^1 \cdot X_1 + w_3^1 \cdot X_2 + b_1^1,\ a_1 = \sigma(z_1)$
- $z_2 = w_2^1 \cdot X_1 + w_4^1 \cdot X_2 + b_2^1,\ a_2 = \sigma(z_2)$
- $\hat{Y} = w_1^2 \cdot a_1 + w_2^2 \cdot a_2 + b^2$
- Cost function = $\frac{1}{2}\left(\hat{Y} - Y\right)^2$
- See lecture appendix for a worked example.

Automatic differentiation in PyTorch

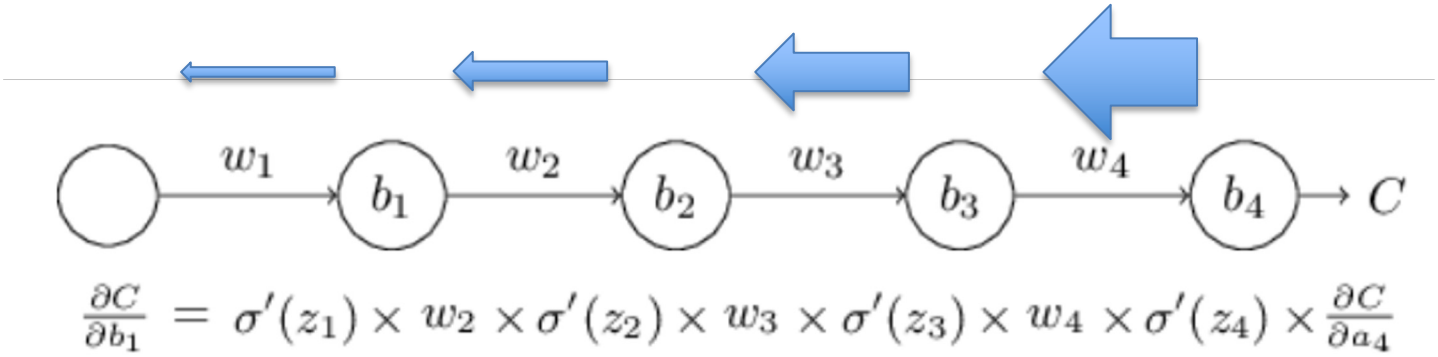
- PyTorch implements backprop via `backwards()` function

```
import torch
import torch.nn as nn

# Define a simple model
model = nn.Linear(2, 1)  # Single layer: y = w * x + b
loss_fn = nn.MSELoss()

# Input and target
x = torch.tensor([[1.0, 2.0], [3.0, 4.0]], requires_grad=True)
y = torch.tensor([[5.0], [7.0]])

# Forward pass
pred = model(x)
loss = loss_fn(pred, y)

# Backward pass
loss.backward()

# Gradients
print("Weight gradient:", model.weight.grad)
print("Bias gradient:", model.bias.grad)
```

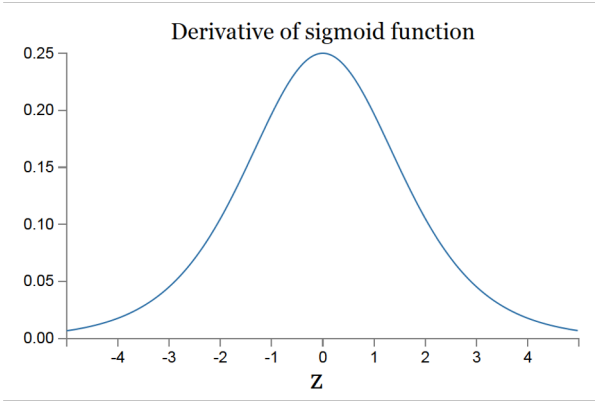
Credit: Noel Benji

Vanishing Gradients Problem

- Deep NN



$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$



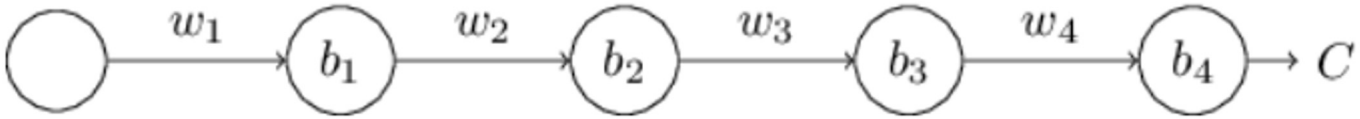
$$w < 1$$
$$\sigma' \leq 0.25 \longrightarrow w \times \sigma' < 0.25$$

In practice:

- Use ReLU
- Don't use sigmoid

Exploding Gradients Problem

Deep NN



$w \times \sigma' > 1$

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$

Be careful with the weight initialization

Hyperparameters

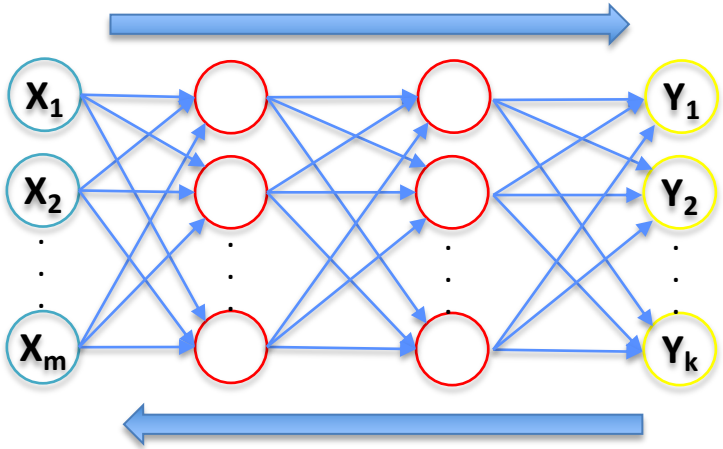
- Hyperparameters:
  - Learning rate
  - Network architecture
  - Regularization

- Use cross-validation for optimizing the hyperparameters

Recap (Backpropagation)

0- Initialize the weights and biases

1- Forward pass



2- Compute the error

3- Backward pass, compute the gradients

4- Update the weights

Training deep neural networks

Main idea: iteratively minimize an (error) function.

$$\min_\theta f_\theta(x) = \min_\theta \sum_{k=0}^{N} \left(\tilde{Q}_\theta(s_k, a_k) - R_k\right)^2$$

$$\text{where } x = (s_k, a_k, R_k)_{k \in [N]}$$

via Backprop

$$\nabla_\theta f_\theta(x) = 2 \sum_{k=0}^{N} \left(\tilde{Q}_\theta(s_k, a_k) - R_k\right) \nabla_\theta \tilde{Q}_\theta(s_k, a_k)$$

Gradient descent algorithm:

1. Pick a starting θ_0
2. Repeat:
    1. Compute descent direction: $-\nabla_\theta f_\theta(x)$
    2. Step in the direction: $\theta_{t+1} = \theta_t - \eta \nabla_\theta f_\theta(x)$
    3. Check if we should stop

Remark: Loss with deep neural networks is not convex

Implication: (S)GD might get stuck in local optima

Loss Landscape Explorer | Explore real loss landscapes of deep learning optimization processes

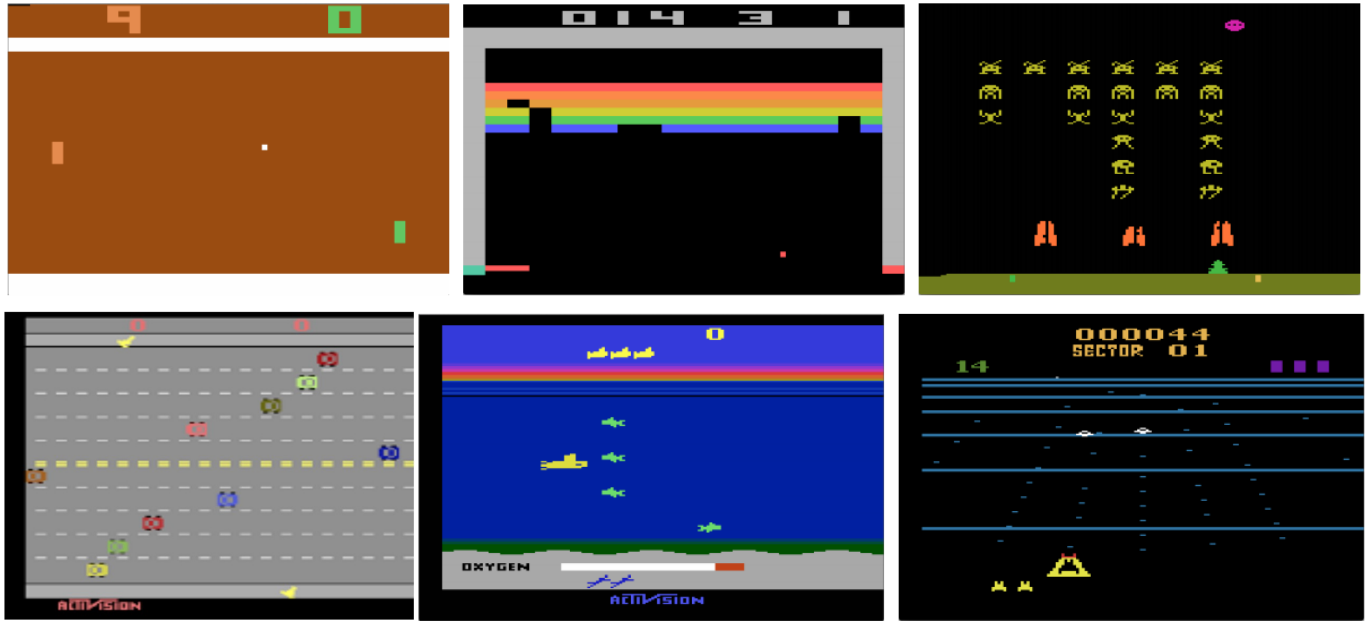
Recap

- Neural networks are powerful function approximators for deep RL
- Use backprop + gradient descent to find good parameters (weights and biases) for neural networks (“DNN training”)
- Use stochastic gradient descent for training efficiency and robustness
- Vanishing (exploding) gradients can slow down learning process and increase inaccuracy in RNNs and Deep NNs
- Use ReLU activation function
- Hyperparameters may need to be tuned for your specific problem

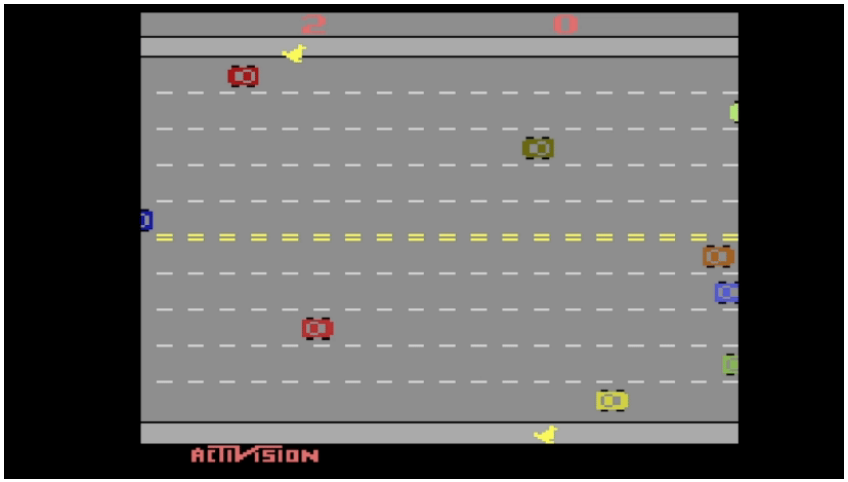
Outline

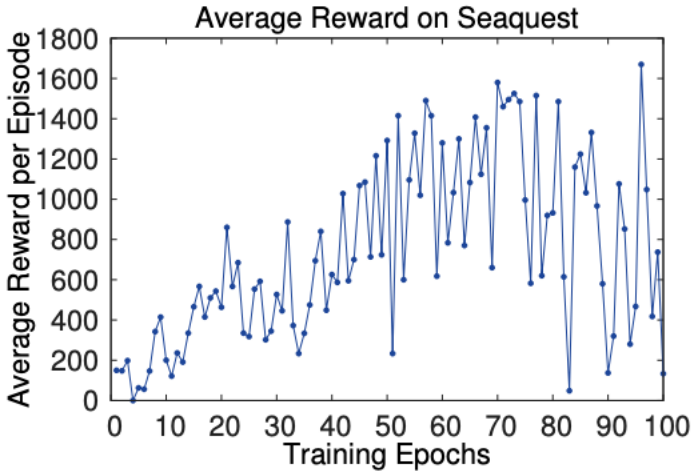
1. Recap Exercise: Parking Problem (Modeling & Solving)
2. Tackling Large State Spaces: Deep reinforcement learning
3. Exploration vs Exploitation
4. Function Approximation
5. Deep learning 101
6. **Deep Q Networks (DQN)**
7. Policy Gradient
8. Actor Critic

Fitted Q-iteration applied to ATARI Games



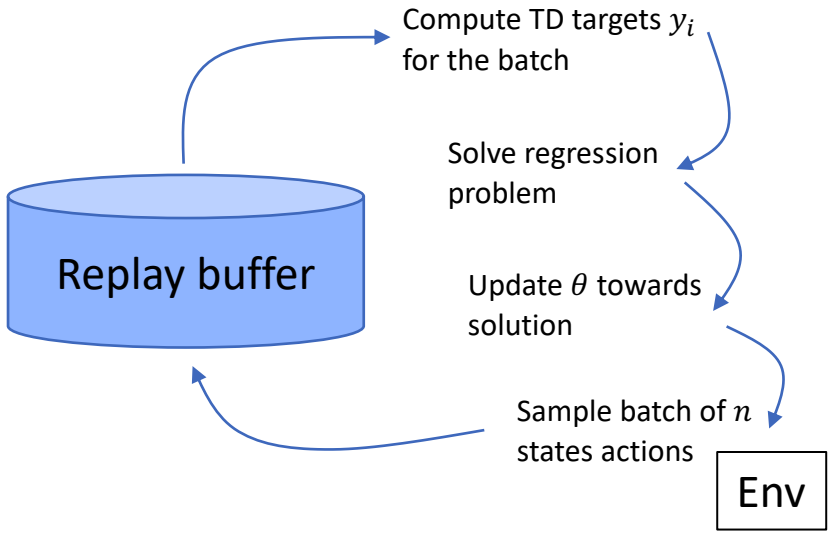
Still doesn't quite work. Why?





- Training is not stable.
- A big reason: the training data is not representative of all state-action pairs.
- Recall: garbage in, garbage out
  - The state-action space is HUGE.
  - The RL agent "collects its own data," wherever it happens to be.
  - If that data is bad, then the result is bad too (and can further worsen future collection of data).

Solution: Replay Buffer



- Increase data diversity in the batch!
- De-correlates samples
- Increased diversity in data → less likely that the data overall is bad for learning

Next issue: chasing a moving target

Q-iteration loss using bootstrapped target

- Iteration i

$$\tilde{L}(s_{1:n}, a_{1:n}, y_{1:n}; \theta)$$

$$= \sum_t \left(Q_\theta(s_t, a_t) - r_t - \gamma \max_{a'} Q_{\theta_i}(s_{t+1}, a') \right)^2$$

- Iteration $i + 1$

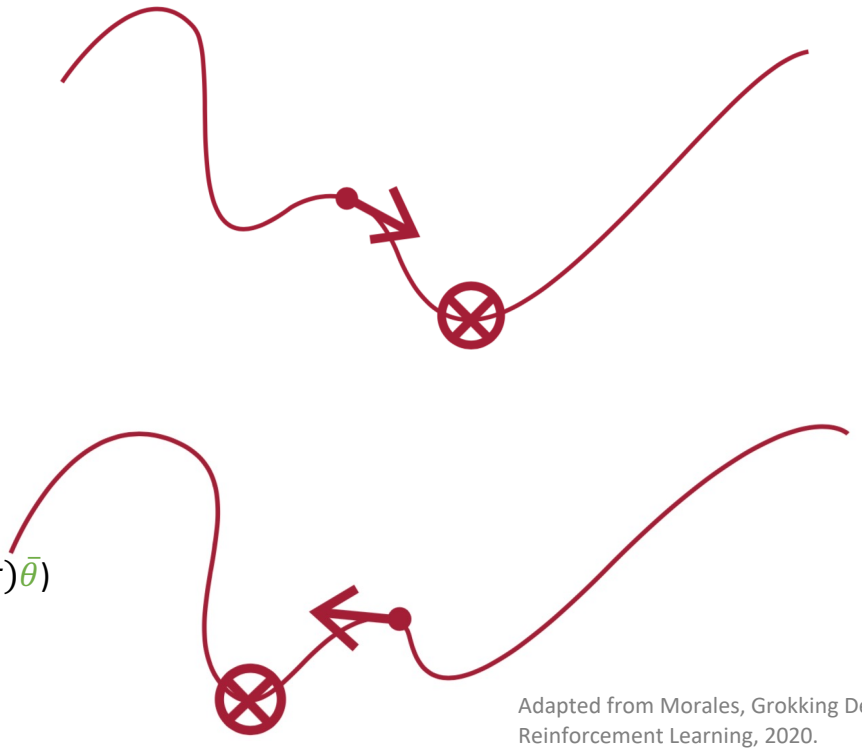
$$\tilde{L}(s_{1:n}, a_{1:n}, y_{1:n}; \theta)$$

$$= \sum_t \left(Q_\theta(s_t, a_t) - r_t - \gamma \max_{a'} Q_{\theta_{i+1}}(s_{t+1}, a') \right)^2$$

- Solution: change the target slowly
(e.g., $\bar{\theta} \leftarrow \theta$ every 1000 steps or $\bar{\theta}' \leftarrow \tau\theta + (1-\tau)\bar{\theta}$)

$$\tilde{L}(s_{1:n}, a_{1:n}, y_{1:n}; \theta)$$

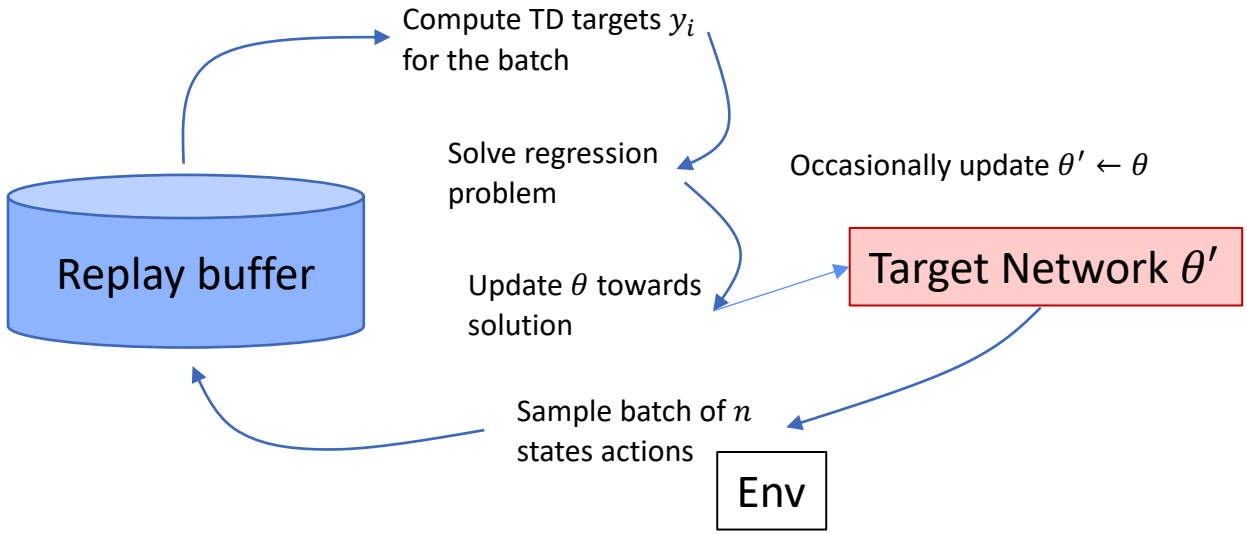
$$= \sum_t \left(Q_\theta(s_t, a_t) - r_t - \gamma \max_{a'} Q_{\bar{\theta}}(s_{t+1}, a') \right)^2$$



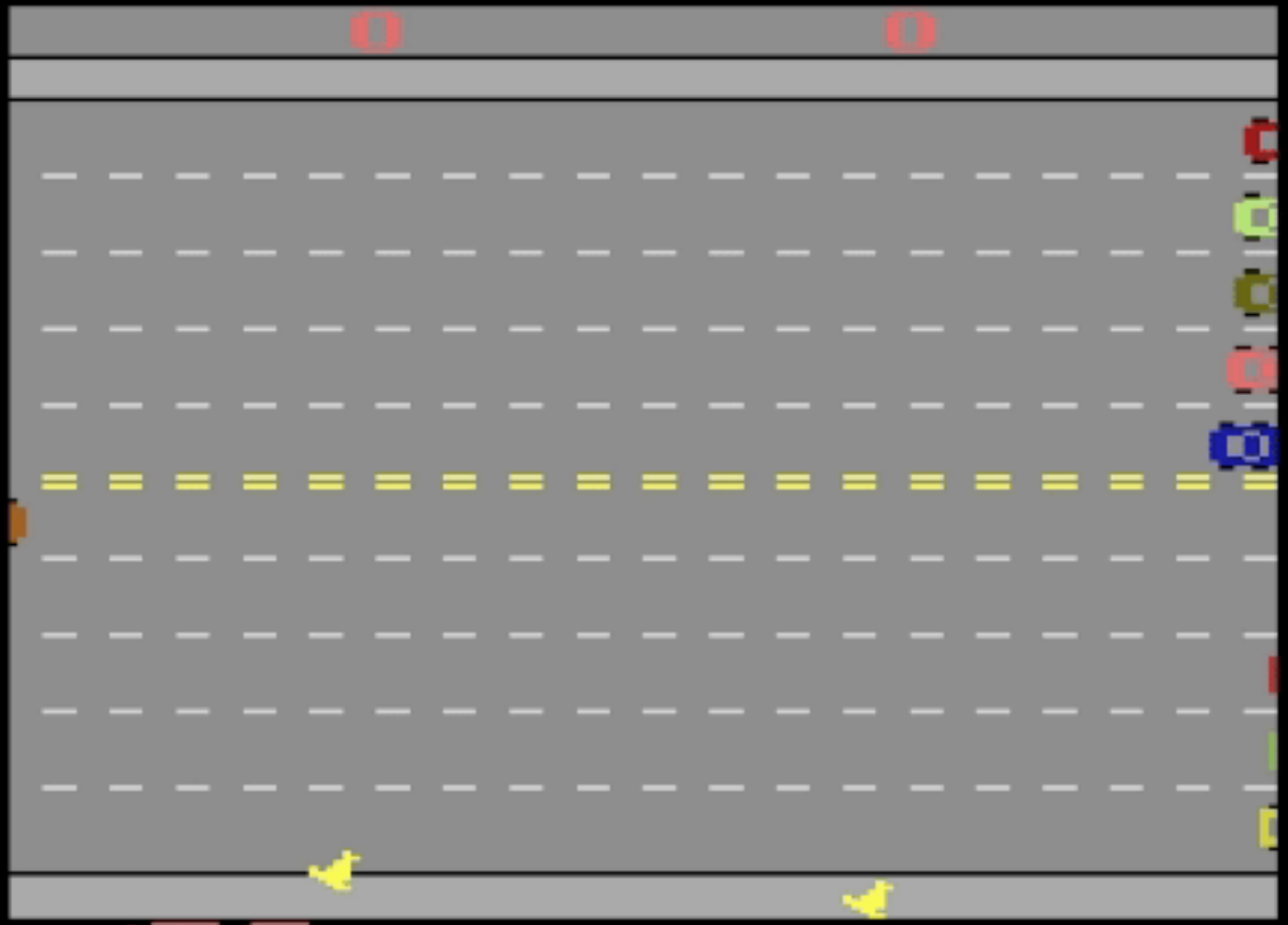
Adapted from Morales, Grokking Deep Reinforcement Learning, 2020.

"Target network"

Solution: Target Network

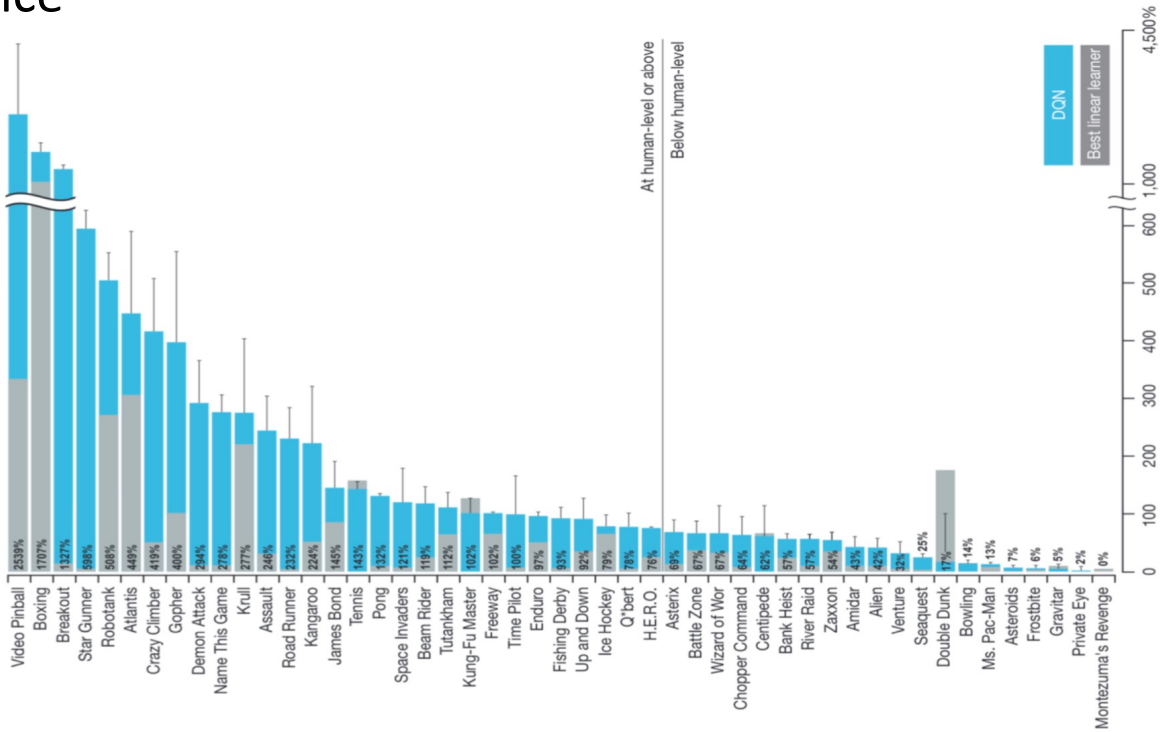


- Slowly update a target network
- Moves the TD target slowly → more stable training


0
0
ACTIVISION

DQN – Atari

Performance



* Human-level is indicated by 75% of the game score achieved by a professional human game tester, not necessarily an expert Atari player.

DQN – Atari

Ablation

DQN

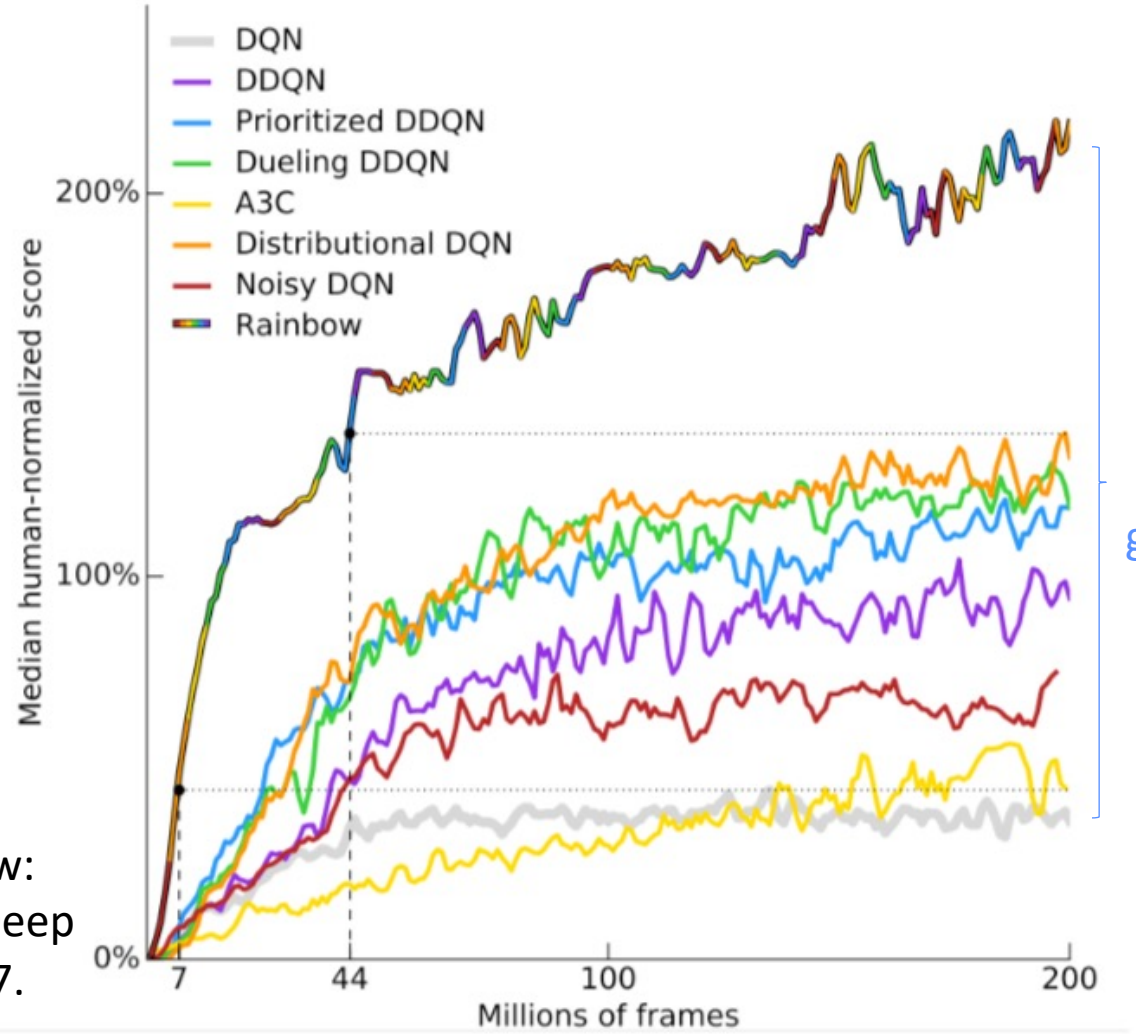
Game	With replay, with target Q	With replay, without target Q	Without replay, with target Q	Without replay, without target Q
Breakout	316.8	240.7	10.2	3.2
Enduro	1006.3	831.4	141.9	29.1
River Raid	7446.6	4102.8	2867.7	1453.0
Seaquest	2894.4	822.6	1003.0	275.8
Space Invaders	1088.9	826.3	373.2	302.0

* Ablations were not hyperparameter tuned, while the full method was, so their performances may underestimate a similarly tuned ablation.

Deep RL Demo

Link

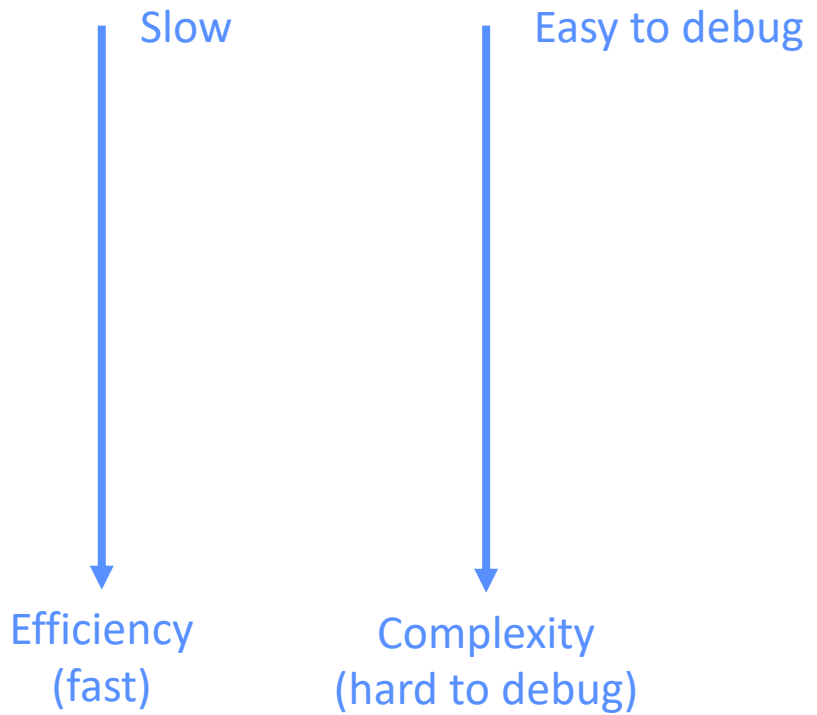
Rainbow DQN



Hessel, Matteo, et al. ”Rainbow: Combining Improvements in Deep Reinforcement Learning.” 2017.

A menu of value-based (RL) algorithms

- Dynamic programming
- Value iteration
- Q-value iteration
- Q-learning
- Fitted Q-iteration
- DQN
- DDQN
- Prioritized experience replay
- Rainbow DQN

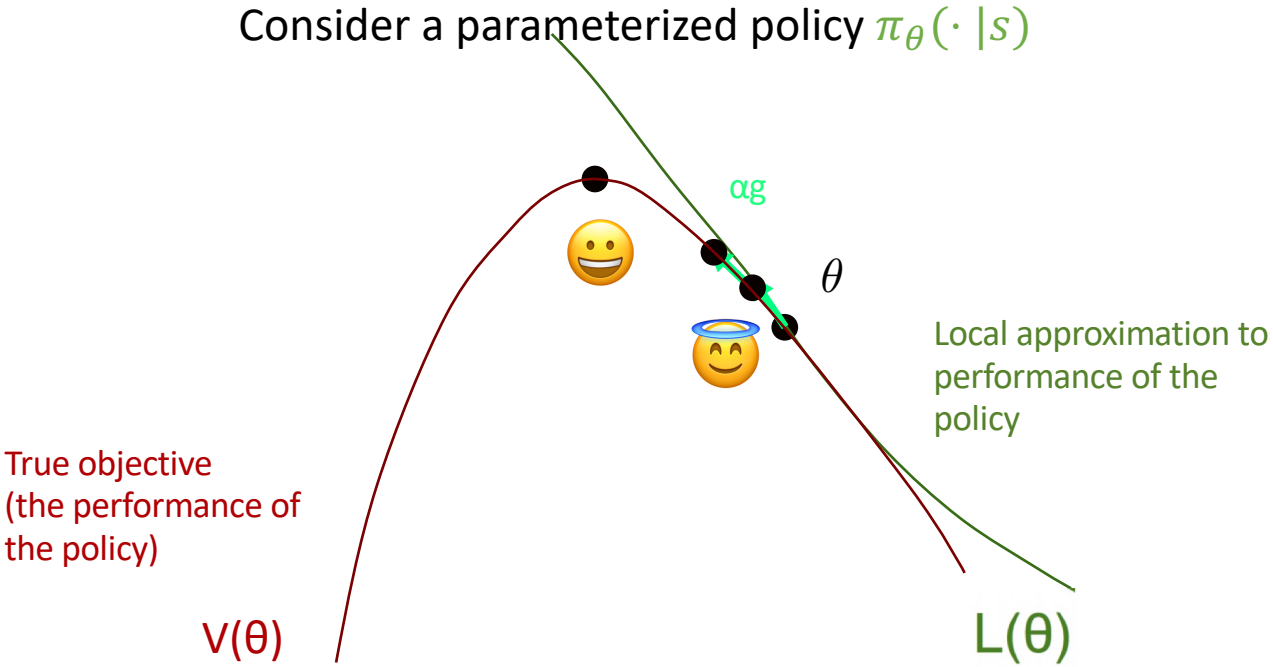


Tip: Use the simplest algorithm the solves your problem

Outline

1. Recap Exercise: Parking Problem (Modeling & Solving)
2. Tackling Large State Spaces: Deep reinforcement learning
3. Exploration vs Exploitation
4. Function Approximation
5. Deep learning 101
6. Deep Q Networks (DQN)
7. **Policy Gradient**
8. Actor Critic

Policy gradient = gradient ascent for MDPs



Adapted from Matteo Pirotta

Policy Gradient = gradient ascent for MDPs

$$V(\pi_{\theta_k}) = \mathbb{E}\left[\sum_{t=0}^{T-1} r_t | \pi_{\theta_k}, M\right] = \mathbb{E}_{\tau \sim \mathbb{P}(\tau | \pi_{\theta_k}, M)}[\mathcal{R}(\tau)]$$

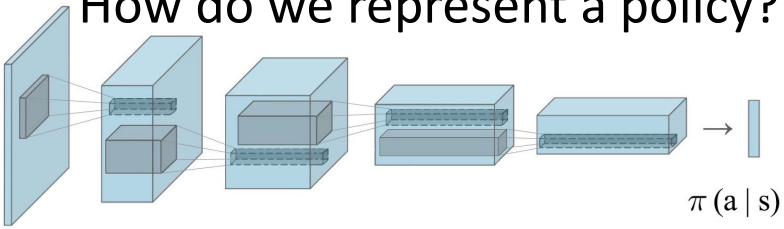
Policy Gradient

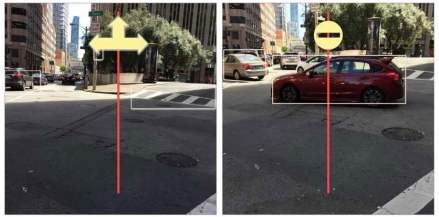
$$\theta_{k+1} = \theta_k + \alpha_k \nabla_\theta V(\theta_k)$$

How do we compute $\nabla_\theta V(\theta)$?

Example: Parameterized Policy

How do we represent a policy?





Normal Policy

$$\pi(a|s) = \frac{1}{\sigma_\omega(s)\sqrt{2\pi}} e^{-\frac{(a-\mu_\theta(s))^2}{2\sigma_\omega^2(s)}}$$

Continuous actions

Gibbs (softmax) Policy

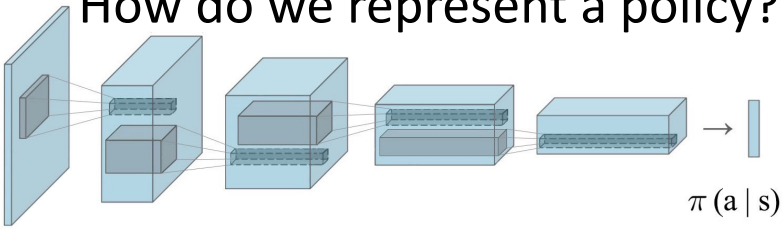
$$\pi(a|s) = \frac{e^{\mathcal{K}Q_\theta(s,a)}}{\sum_{a' \in \mathcal{A}} e^{\mathcal{K}Q_\theta(s,a')}}$$

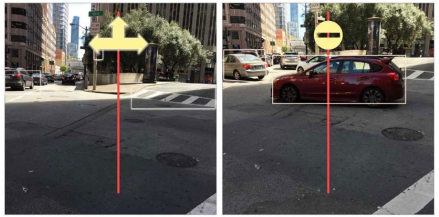
Discrete actions

Differentiable! → autodiff via PyTorch

Example: Parameterized Policy

How do we represent a policy?





Normal Policy

$$\pi(a|s) = \frac{1}{\sigma_\omega(s)\sqrt{2\pi}} e^{-\frac{(a-\mu_\theta(s))^2}{2\sigma_\omega^2(s)}}$$

Then:

$$\nabla_\theta \log \pi(a|s) = \frac{(a-\mu_\theta(s))}{\sigma_\omega^2(s)} \nabla_\theta \mu_\theta(s)$$

$$\nabla_\omega \log \pi(a|s) = \frac{(a-\mu_\theta(s))^2 - \sigma_\omega^2(s)}{\sigma_\omega^3(s)} \nabla_\omega \mu_\omega(s)$$

Continuous actions

Gibbs (softmax) Policy

$$\pi(a|s) = \frac{e^{\mathcal{K} Q_\theta(s,a)}}{\sum_{a' \in \mathcal{A}} e^{\mathcal{K} Q_\theta(s,a')}}$$

Then:

$$\nabla_\theta \log \pi(a|s) = \mathcal{K} \nabla_\theta Q_\theta(s,a) - \mathcal{K} \sum_{a' \in \mathcal{A}} \pi(a'|s) \nabla_\theta Q_\theta(s,a')$$

Discrete actions

Policy Gradient (Finite-Horizon)

Given an MDP $M = (\mathcal{S}, \mathcal{A}, p, r, T, \mu)$ and a policy π_{θ_0}. For k = 1,2,…

1. Use π_{θ_k} to collect data τ.
2. Use τ to approximate gradient of:

Maximizing this is ultimately what we desire

$$V(\pi_{\theta_k}) = \mathbb{E}\left[\sum_{t=0}^{T-1} r_t \middle| \pi_{\theta_k}, M\right] = \mathbb{E}_{\tau \sim \mathbb{P}(\tau|\pi_{\theta_k}, M)}[\mathcal{R}(\tau)]$$

where

- μ is an initial state distribution
- $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_{T-1}, a_{T-1}, r_{T-1}, s_T)$ (includes terminal reward) is a trajectory
- $\mathcal{R}(\tau)$ its return (sum of rewards).

Main issue: MDP is a complex object to differentiate through, i.e. $\nabla_\theta \mathbb{P}(\tau|\pi_\theta, M)$. (Example: Frozen Lake)

3. Update $\theta_{k+1} = \theta_k + \alpha_k \widehat{\nabla_\theta V}(\pi_{\theta_k})$

How?

Policy Gradient (Finite-Horizon)

Policy Gradient Theorem [Williams, 1992; Sutton et al., 2000]

For any finite-horizon MDP $M = (\mathcal{S}, \mathcal{A}, p, r, T, \mu)$ and differentiable policy π_θ

$$\nabla_\theta V(\pi_\theta) = \mathbb{E}_{\tau \sim \mathbb{P}(\cdot|\pi, M)} \left[R(\tau) \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(s_t, a_t) \right]$$

Gradient is now on the inside! We can differentiate through (differentiable) policies.

- Model-free! Why?
- Compare: taking gradient through trajectory-space is difficult

$$\nabla_\theta V(\pi_\theta) = \nabla_\theta \mathbb{E}_\tau[R(\tau)] = \nabla_\theta \int \mathbb{P}(\tau|\pi_\theta, M) R(\tau) d\tau$$

REINFORCE [Williams, 1992]

1. Let π_{θ_1} be an arbitrary policy.
2. At each iteration $k = 1, \dots, K$
   - Sample m trajectories $\tau_i = (s_0, a_0, r_0, s_1, \dots, s_{T-1}, a_{T-1}, r_{T-1}, s_T)$ following π_k
   - Compute unbiased gradient estimate:

$$\widehat{\nabla_\theta V}(\pi_{\theta_k}) = \frac{1}{m} \sum_{i=1}^{m} \left(\sum_{t=0}^{T-1} r_t^i \right) \left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_{\theta_k}(a_t^i | s_t^i) \right)$$

   - Update parameters:

$$\theta_{k+1} = \theta_k + \alpha_k \widehat{\nabla_\theta V}(\pi_{\theta_k})$$

3. Return last policy π_{θ_K}

Monte Carlo approximation of policy gradient

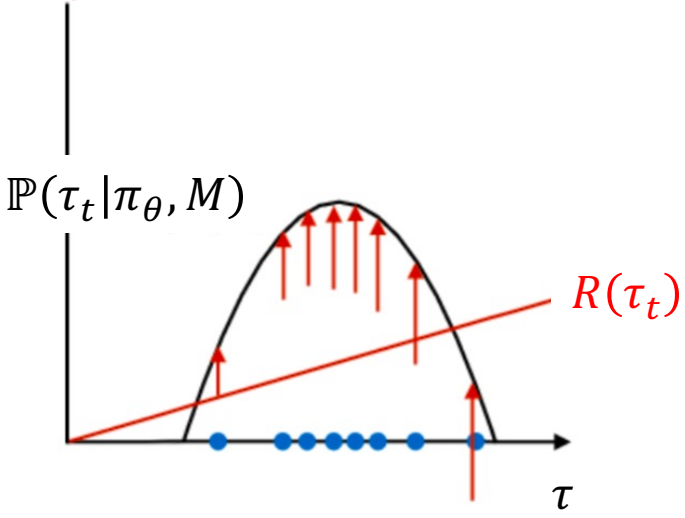
REINFORCE as Supervised Learning

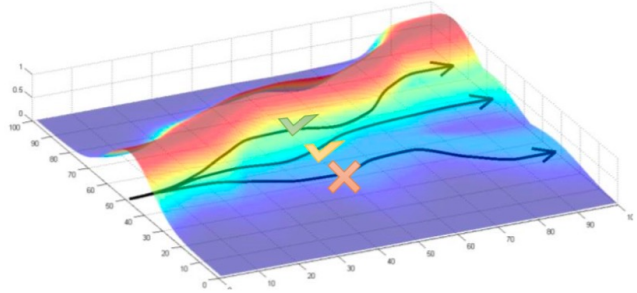
$$\hat{g}_t = R(\tau_t)\nabla_\theta \log \mathbb{P}(\tau_t|\pi_\theta, M)$$

- $R(\tau_t)$ measures how good is sample τ_t
- Moving in the direction of $\hat{g}_t$ pushes up the log probability of the sample in proportion to how good it is.

Interpretation: uses good trajectories as supervised examples

- Like maximum likelihood in supervised learning
- Good stuff are made more likely while bad less
- Trial and Error approach

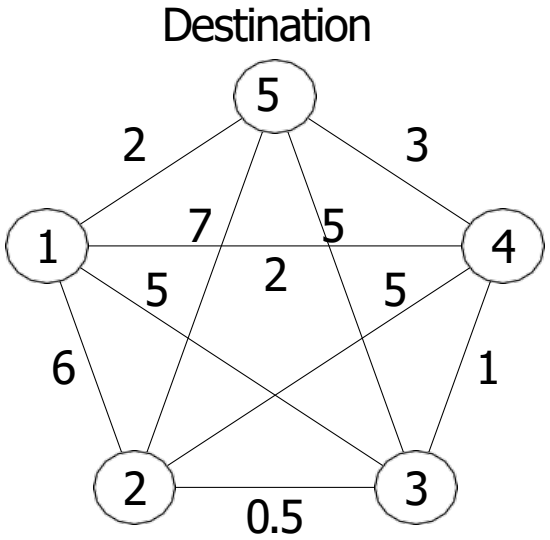




From "CS 294-112: Deep Reinforcement Learning" slides by S. Levine

Dynamic programming vs policy gradient

How would policy gradient solve shortest path?



Destination is node 5.

REINFORCE

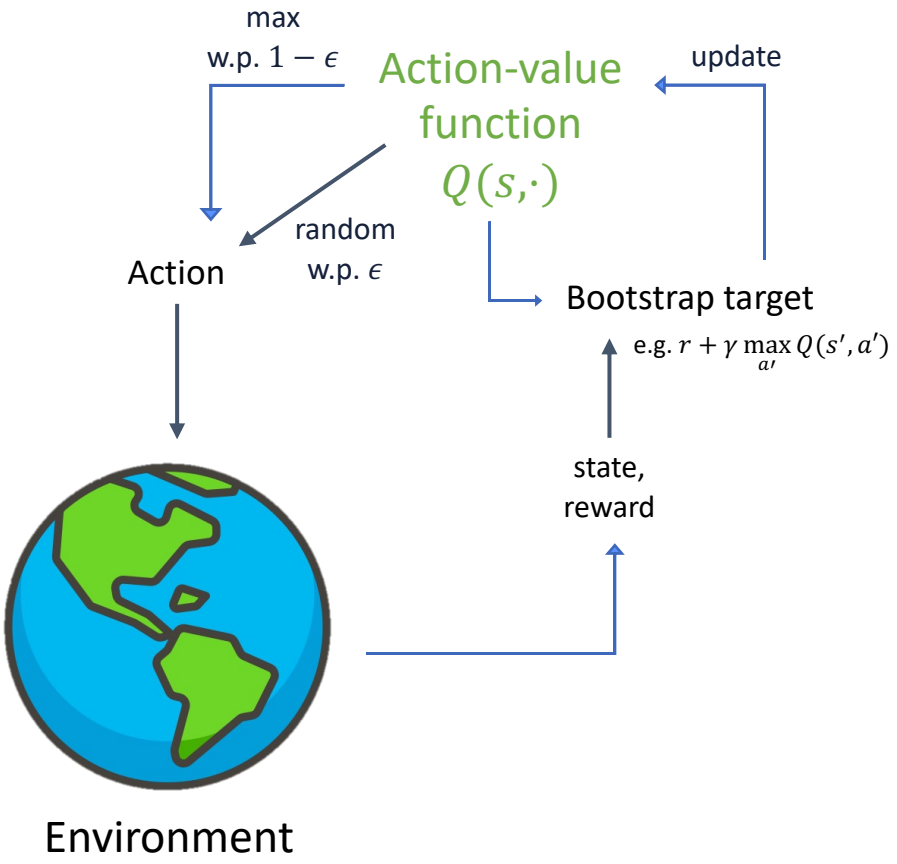
Pros

- Easy to compute
- Does not use Markov property!
- Can be used in partially observable MDPs without modification

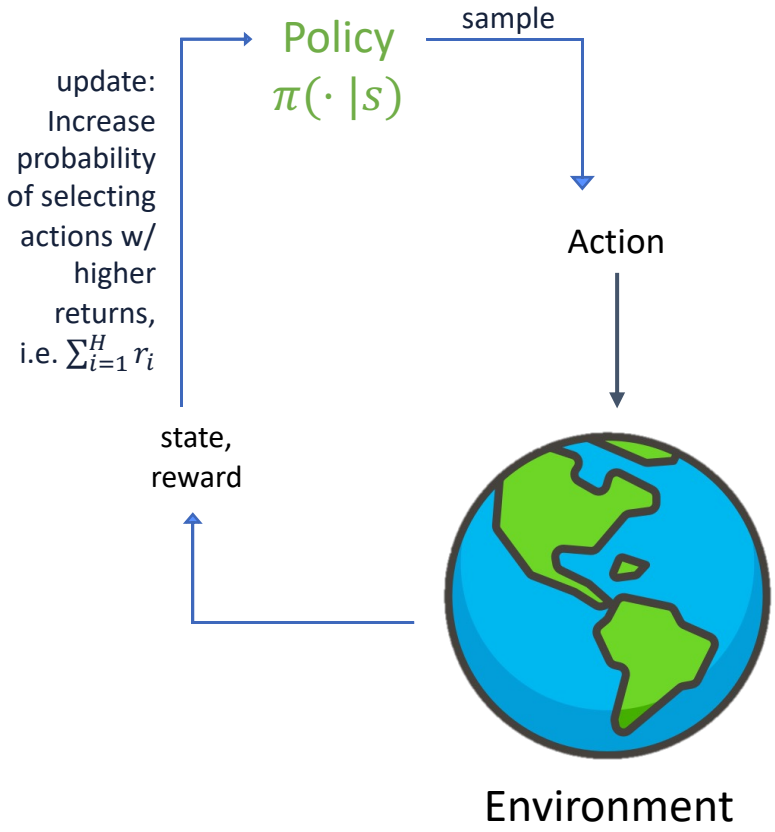
Issues

- Use a MC estimate of $Q(s, a)$
- It has possibly a very large variance
- Needs many samples to converge

Value-based methods



Policy-based methods



Outline

1. Recap Exercise: Parking Problem (Modeling & Solving)
2. Tackling Large State Spaces: Deep reinforcement learning
3. Exploration vs Exploitation
4. Function Approximation
5. Deep learning 101
6. Deep Q Networks (DQN)
7. Policy Gradient
8. **Actor Critic**

Policy- and value-based methods → actor-critic

- Monte-Carlo policy gradient is unbiased but still has high variance

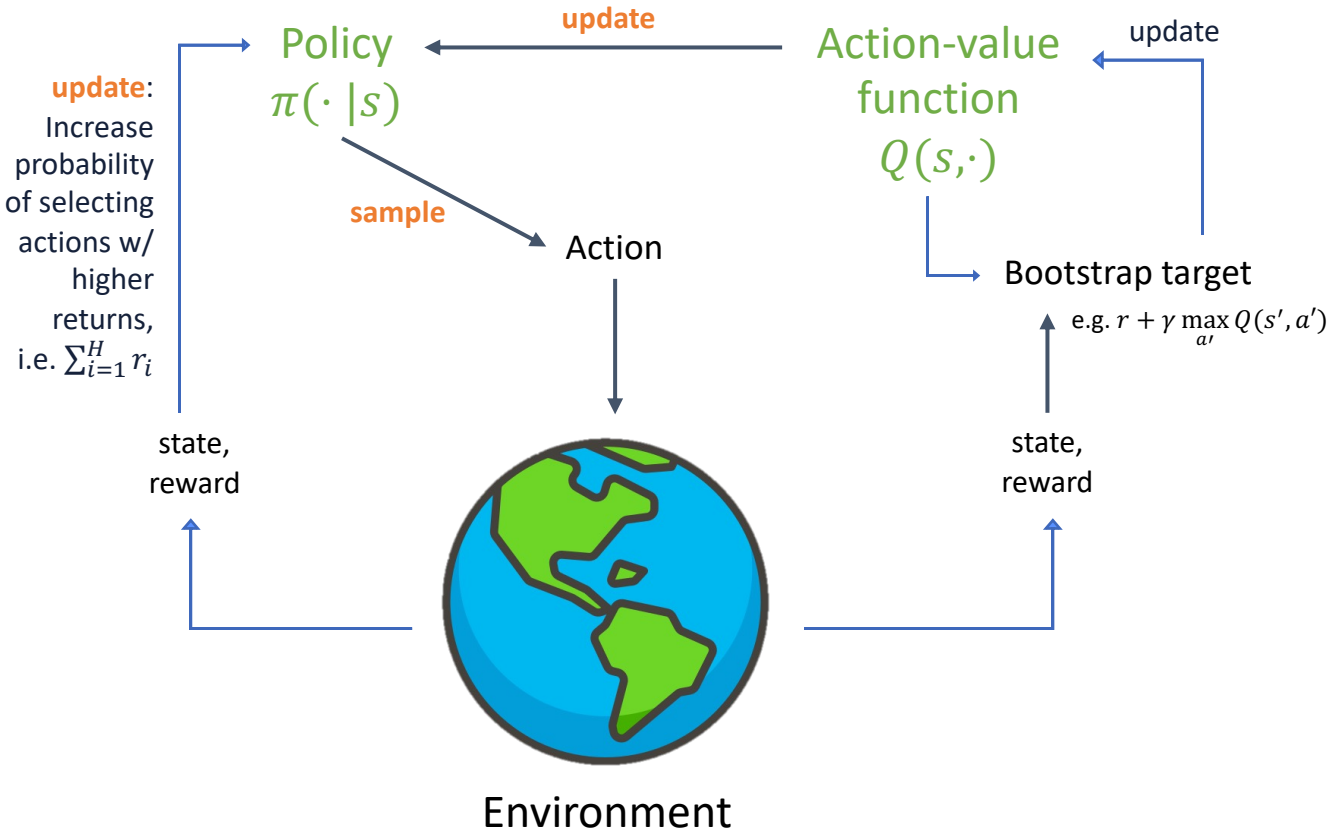
$$\nabla_\theta V(\pi_\theta) = \mathbb{E}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T-1} r_{t'}\right]$$

- Incorporate an estimate of $Q^\pi(s,a) \Longrightarrow$ actor-critic
  - Critic: estimate the value function
  - Actor: update the policy in the direction suggested by the critic
- Actor-critic

$$\nabla_\theta V(\pi_\theta) = \mathbb{E}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, Q^{\pi_\theta}(s_t,a_t)\right]$$

- These are equivalent (see HW).

Actor-critic methods



Actor-Critic

- Algorithm maintains two sets of parameters: $\theta \longmapsto \pi_\theta, \omega \longmapsto Q_\omega$
- Critic can use $TD(0)$

---

for $t = 0, \dots, T-1$ **do**

$a_t \sim \pi_\theta(s_t, \cdot)$ and observe r_t and s_{t+1}

Compute temporal difference

$$\delta_t = r_t + \gamma Q_\omega(s_{t+1}, a_{t+1}) - Q_\omega(s_t, a_t)$$

Update Q estimate

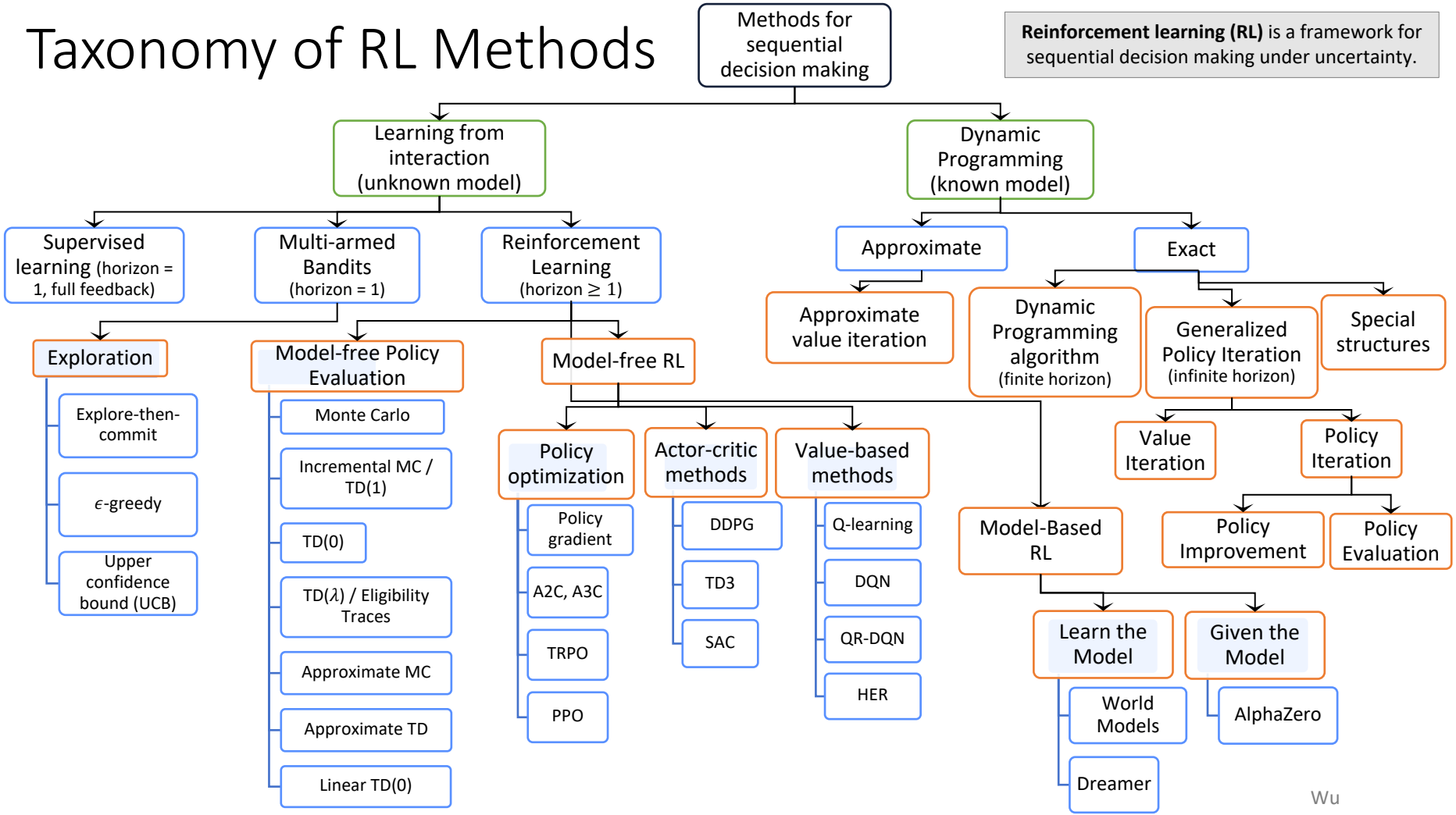
$$\omega = \omega + \beta \delta_t \nabla_\omega Q_\omega(s_t, a_t)$$

Update policy

$$\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(a_t | s_t) Q_\omega(s_t, a_t)$$

end

Taxonomy of RL Methods



Reinforcement learning (RL) is a framework for sequential decision making under uncertainty.

Wu

Stable Baselines3 (SB3)

- A set of reliable implementations of reinforcement learning algorithms in PyTorch for solving sequential decision problems.
- Compatible with `gym` environments.

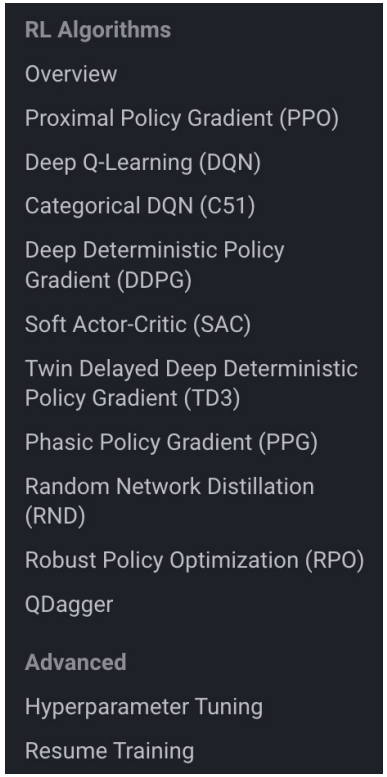
Name
ARS[1]
A2C
CrossQ[1]
DDPG
DQN
HER
PPO
QR-DQN[1]
RecurrentPPO[1]
SAC
TD3
TQC[1]
TRPO[1]
Maskable PPO[1]

https://github.com/DLR-RM/stable-baselines3

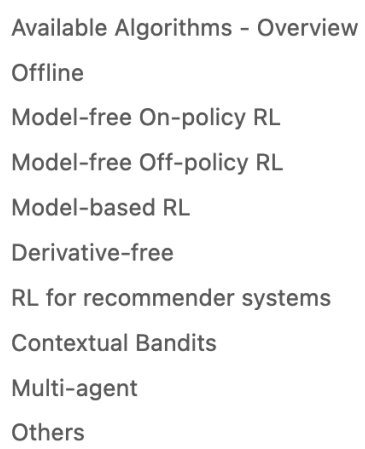
More Implementations of RL algorithms

- For learning and research prototyping:
  - CleanRL: https://docs.cleanrl.dev/
    - A Deep Reinforcement Learning library that provides high-quality single-file implementation with research-friendly features
  - Mastering RL online textbook + code: https://uq.pressbooks.pub/mastering-reinforcement-learning/
    - Implementations of classic algorithms & concepts
- For scaling up:
  - RLlib: https://docs.ray.io/en/latest/rllib/index.html
    - Industry-grade reinforcement learning
    - Built on distributed execution engine Ray

CleanRL



RLlib



References

1. Bertsekas, D. P. (2005). Dynamic programming and optimal control, vol 1. *Belmont, MA: Athena Scientific*, 3rd Edition.
2. Lazaric, A. (2014). Master MVA: Reinforcement Learning.
3. With many slides adapted from Alessandro Lazaric and Matteo Pirotta.

References

References:

1. Deep Learning by “Goodfellow, Bengio, Courville”
2. http://neuralnetworksanddeeplearning.com/index.html

With slides adapted from:

1. Eugene Vinitsky (Berkeley EE290O)

Appendix: DP proof

Dynamic programming algorithm

$V_T(s_T) = r_T(s_T)$
for $t = T-1, \ldots, 0$ **do**
  for $s_t \in \mathcal{S}_t$ **do**
    $V_t(s_t) = \max_{a_t \in \mathcal{A}_t(s_t)} r_t(s_t, a_t) + \mathbb{E}_{s_{t+1} \sim P(\cdot|s_t, a_t)}[V_{t+1}(s_{t+1})]$
end for

- **Proof**: by induction
- "Efficient": O(|S|²|A|T)

- For deterministic shortest path routing
  - Equivalent to Bellman-Ford algorithm
  - **Strength**: Generality
  - "Efficient": O(|S||A|T)
  - Much better than naive approach O(T!)
  - **Weakness**: ALL the tail subproblems are solved

Proof of the induction step

Denote tail policy from time t onward as $\pi_{t:T-1} = \{\pi_t, \pi_{t+1}, \dots, \pi_{T-1}\}$

Assume that $V_{t+1}(s_{t+1}) = V_{t+1}^*(s_{t+1})$. Then:

$$V_t^*(s_t) = \max_{(\pi_t, \pi_{t+1:T-1})} \mathbb{E}_{s_{t+1:T-1}} \left\{ r_t\big(s_t, \pi_t(s_t)\big) + r_T(s_T) + \sum_{i=t+1}^{T-1} r_i\big(s_i, \pi_i(s_i)\big) \right\}$$

$$= \max_{\pi_t} r_t\big(s_t, \pi_t(s_t)\big) + \max_{\pi_{t+1:T-1}} \left[\mathbb{E}_{s_{t+1:T-1}} \left\{ r_T(s_T) + \textstyle\sum_{i=t+1}^{T-1} r_i\big(s_i, \pi_i(s_i)\big) \right\} \right] \qquad \text{[exchange]}$$

$$= \max_{\pi_t} r_t\big(s_t, \pi_t(s_t)\big) + \mathbb{E}_{s_{t+1} \sim P(\cdot|s_t, \pi_t(s_t))} \left\{ \max_{\pi_{t+1:T-1}} \left[\mathbb{E}_{s_{t+2:T-1}} \left\{ r_T(s_T) + \textstyle\sum_{i=t+1}^{T-1} r_i\big(s_i, \pi_i(s_i)\big) \right\} \right] \right\}$$

$$= \max_{\pi_t} r_t\big(s_t, \pi_t(s_t)\big) + \mathbb{E}_{s_{t+1} \sim P(\cdot|s_t, \pi_t(s_t))} \{ V_{t+1}^*(s_{t+1}) \} \qquad \text{[definition]}$$

$$= \max_{\pi_t} r_t\big(s_t, \pi_t(s_t)\big) + \mathbb{E}_{s_{t+1} \sim P(\cdot|s_t, \pi_t(s_t))} \{ V_{t+1}(s_{t+1}) \} \qquad \text{[induction hypothesis]}$$

$$= \max_{a_t \in \mathcal{A}_t(s_t)} r_t(s_t, a_t) + \mathbb{E}_{s_{t+1} \sim P(\cdot|s_t, a_t)} \{ V_{t+1}(s_{t+1}) \}$$

$$= V_t(s_t) \qquad \text{[DP algorithm]}$$

Interpretation as optimal reward-to-go (cost-to-go) function.

Proof of the induction step

For the =, we have:

$$\max_{\pi'} \sum_{s'} p(s'|s,a) V^{\pi'}(s') \leq \sum_{s'} p(s'|s,a) \max_{\pi'} V^{\pi'}(s')$$

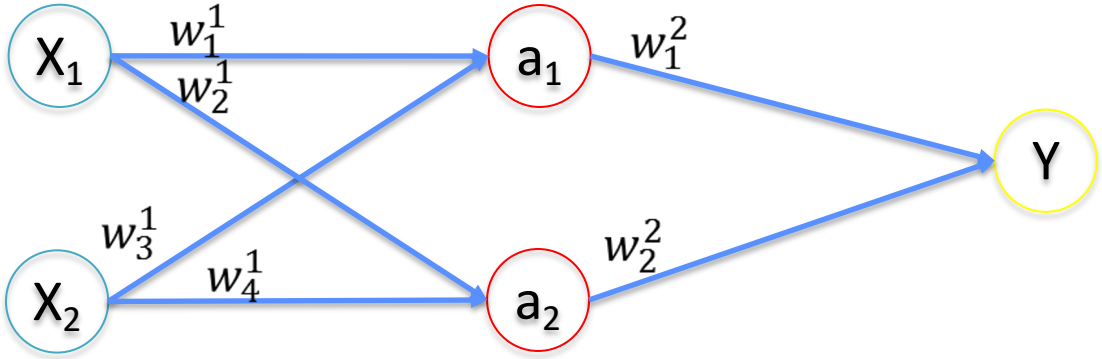
But, let $\bar{\pi}(s') = \operatorname{argmax}_{\pi'} V^{\pi'}(s')$

$$\sum_{s'} p(s'|s,a) \max_{\pi'} V^{\pi'}(s') \leq \sum_{s'} p(s'|s,a) V^{\bar{\pi}}(s') \leq \max_{\pi'} \sum_{s'} p(s'|s,a) V^{\pi'}(s')$$

■

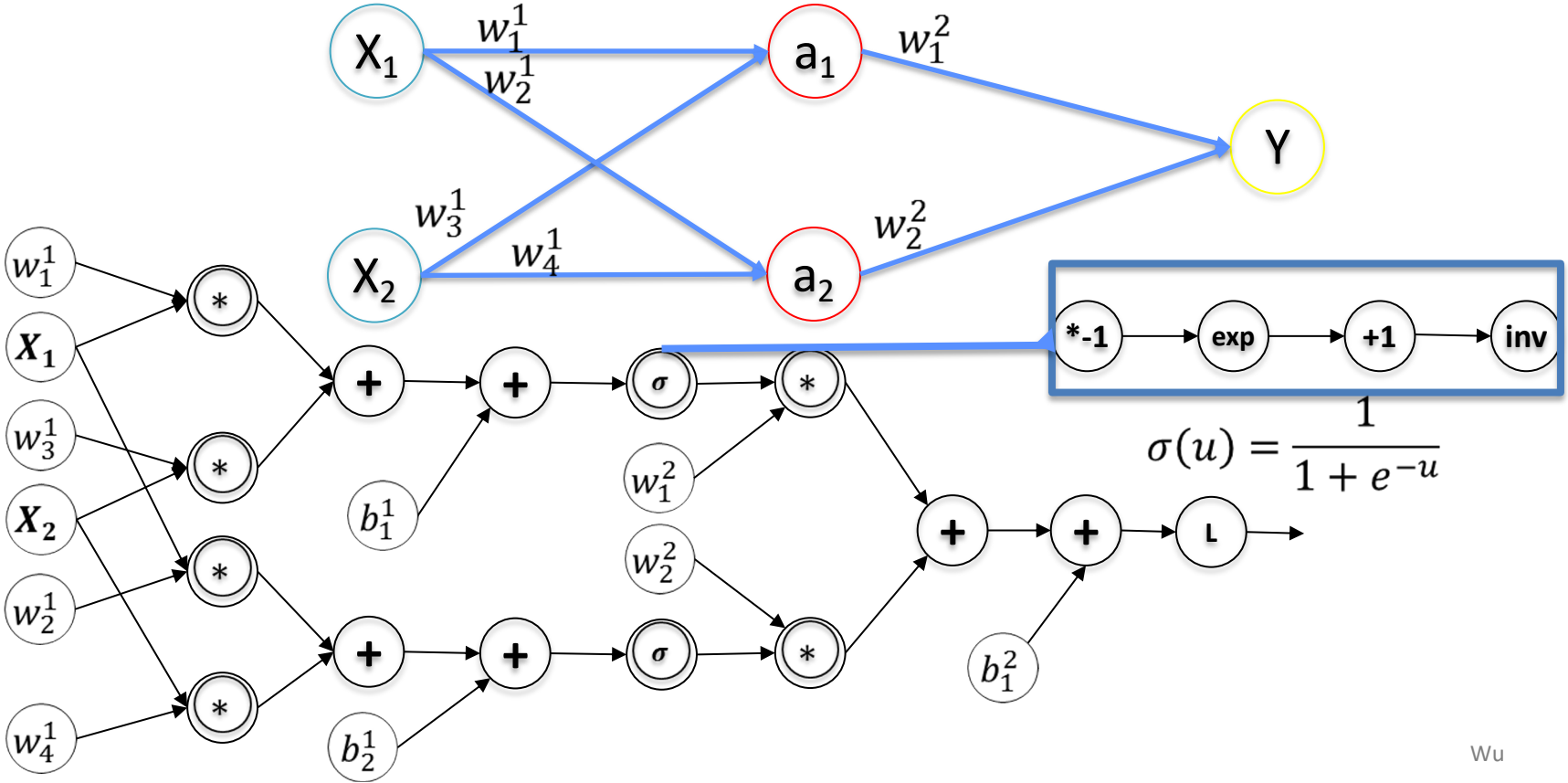
Appendix: Backprop example

Example: neural network value function w/ 1 data point

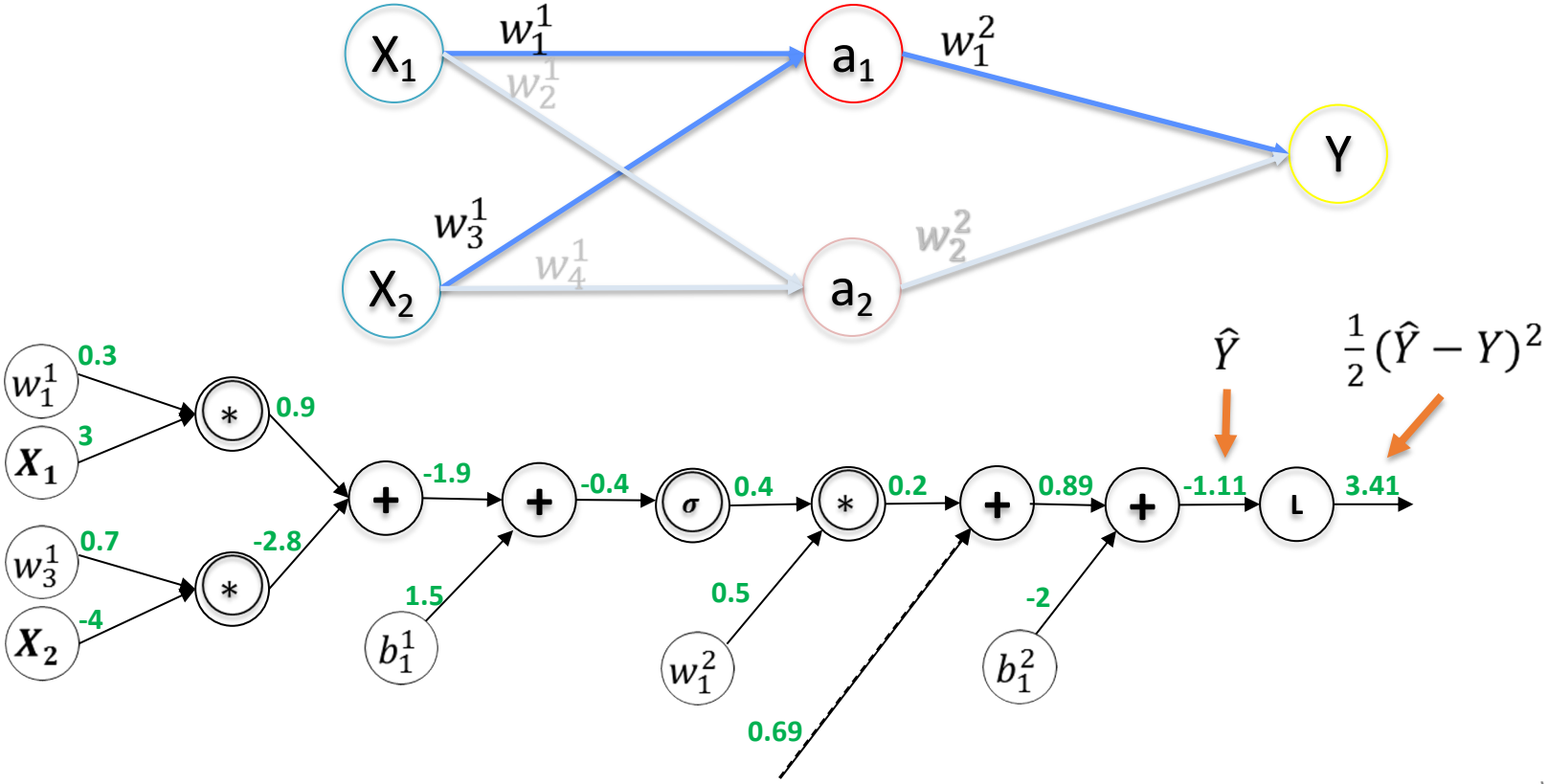


- Assume: $X = \begin{bmatrix} 3 \\ -4 \end{bmatrix}, Y = 1.5, w^1 = \begin{bmatrix} 0.3 \\ 0.9 \\ 0.7 \\ 0.6 \end{bmatrix}, w^2 = \begin{bmatrix} 0.5 \\ 0.8 \end{bmatrix}, b^1 = \begin{bmatrix} 1.5 \\ 1.5 \end{bmatrix}, b^2 = [-2]$, Activation = Sigmoid
- $z_1 = w_1^1 \cdot X_1 + w_3^1 \cdot X_2 + b_1^1,\ a_1 = \sigma(z_1)$
- $z_2 = w_2^1 \cdot X_1 + w_4^1 \cdot X_2 + b_2^1,\ a_2 = \sigma(z_2)$
- $\hat{Y} = w_1^2 \cdot a_1 + w_2^2 \cdot a_2 + b^2$
- Cost function = $\frac{1}{2}\left(\hat{Y} - Y\right)^2$

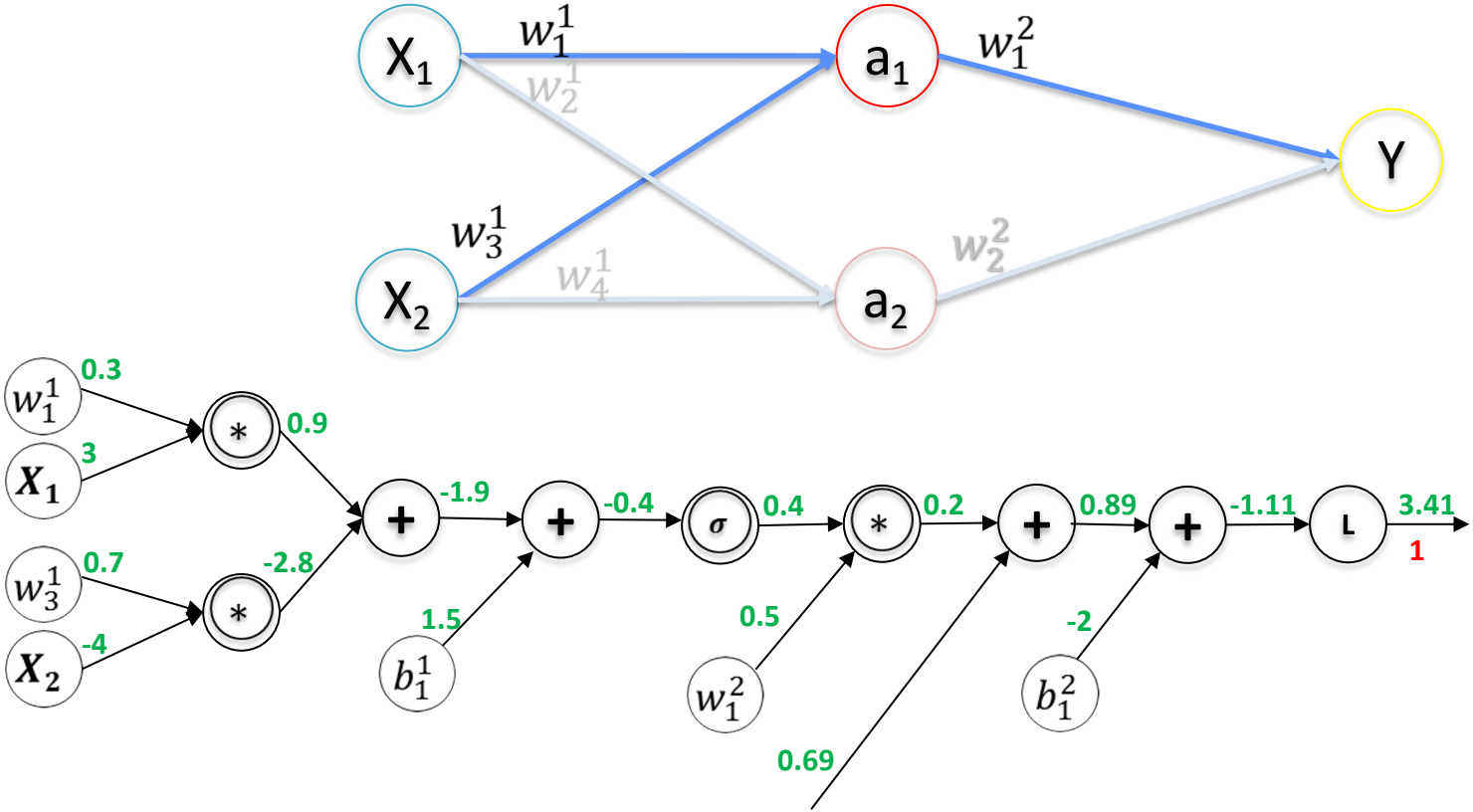
Example


X1
X2
a1
a2
Y
w_1^1
w_2^1
w_3^1
w_4^1
w_1^2
w_2^2
*-1
exp
+1
inv
$\sigma(u) = \frac{1}{1+e^{-u}}$
L
b_1^1
b_2^1
b_1^2

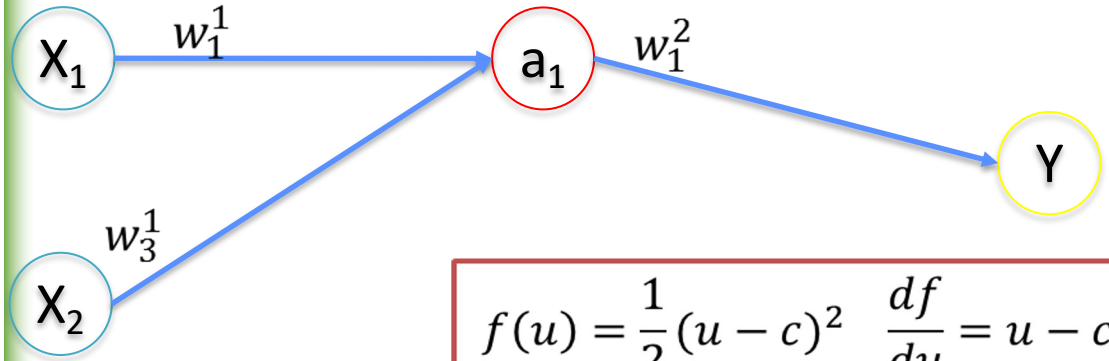
Example Forward Pass



Example Backward Pass



Example Backward Pass

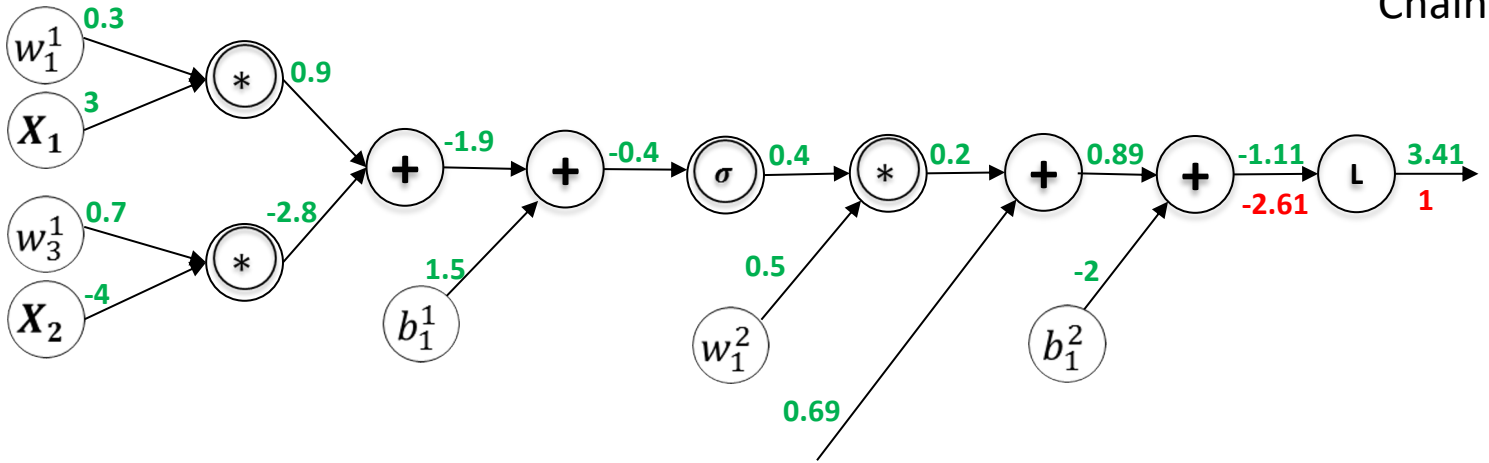


$$f(u) = \frac{1}{2}(u-c)^2 \qquad \frac{df}{du} = u - c$$

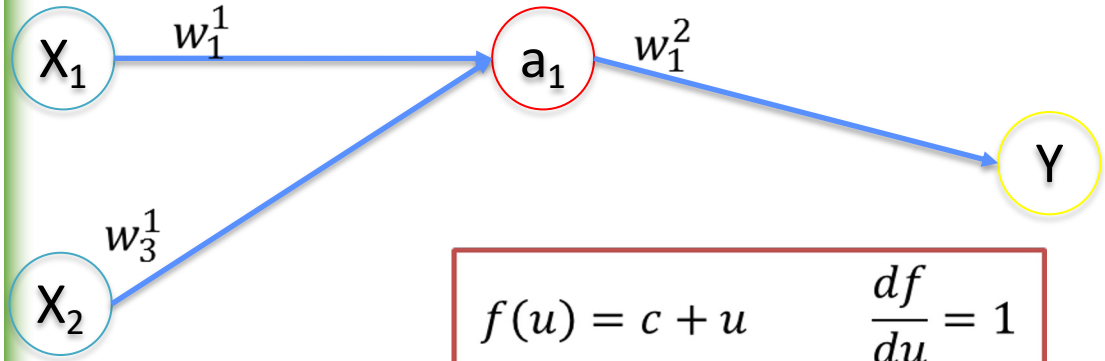
Y=1.5

$$(-1.11 - 1.5) * 1 = -2.61$$

Chain Rule



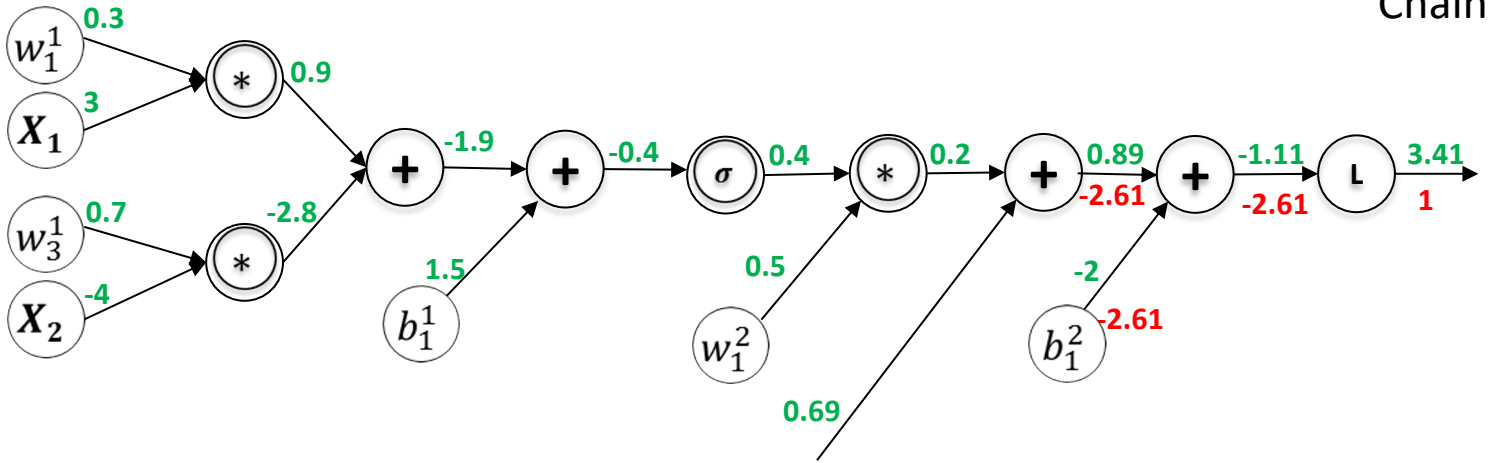
Example Backward Pass



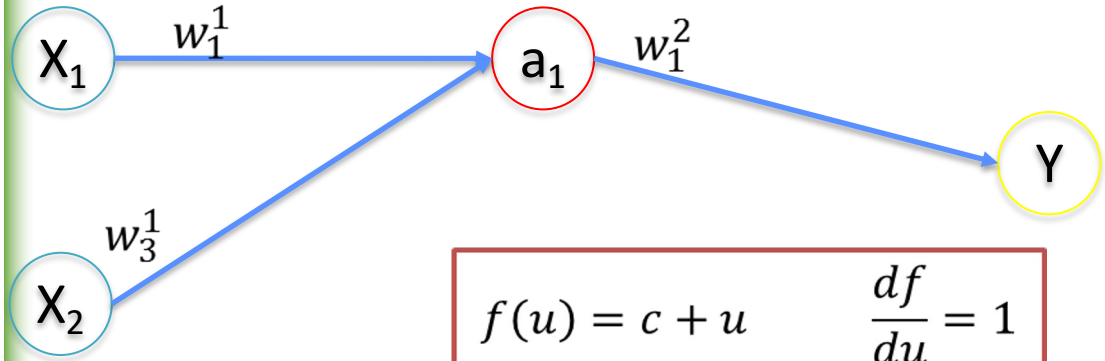
$$f(u) = c + u \qquad \frac{df}{du} = 1$$

$$1 * -2.61 = -2.61$$

Chain Rule



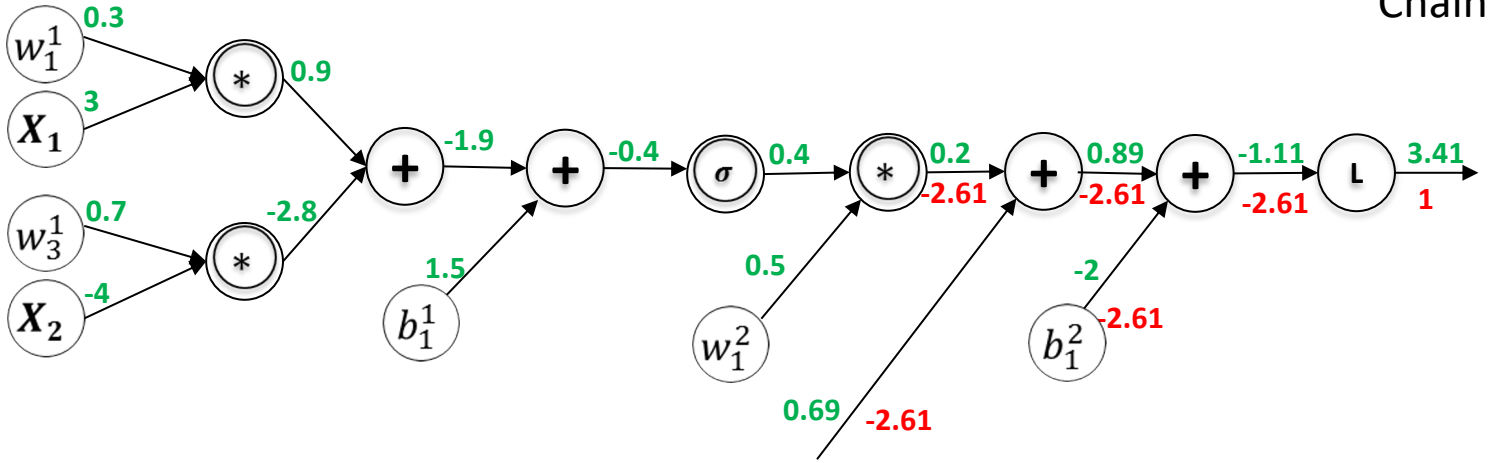
Example Backward Pass



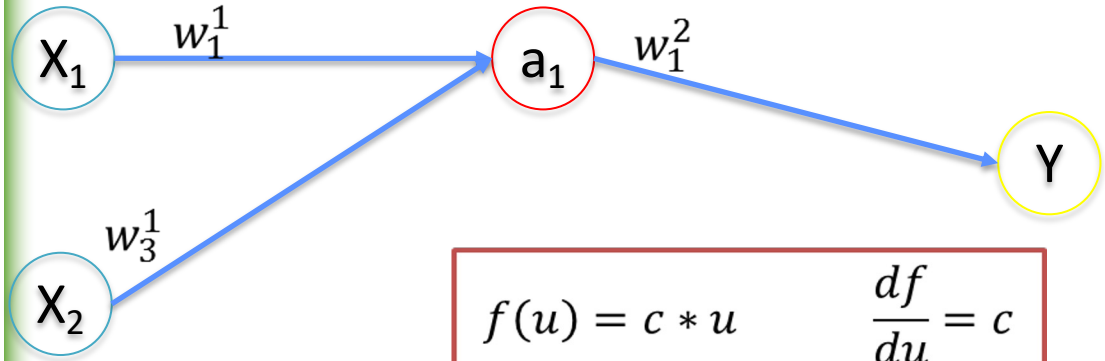
$$f(u) = c + u \qquad \frac{df}{du} = 1$$

$$1 * -2.61 = -2.61$$

Chain Rule



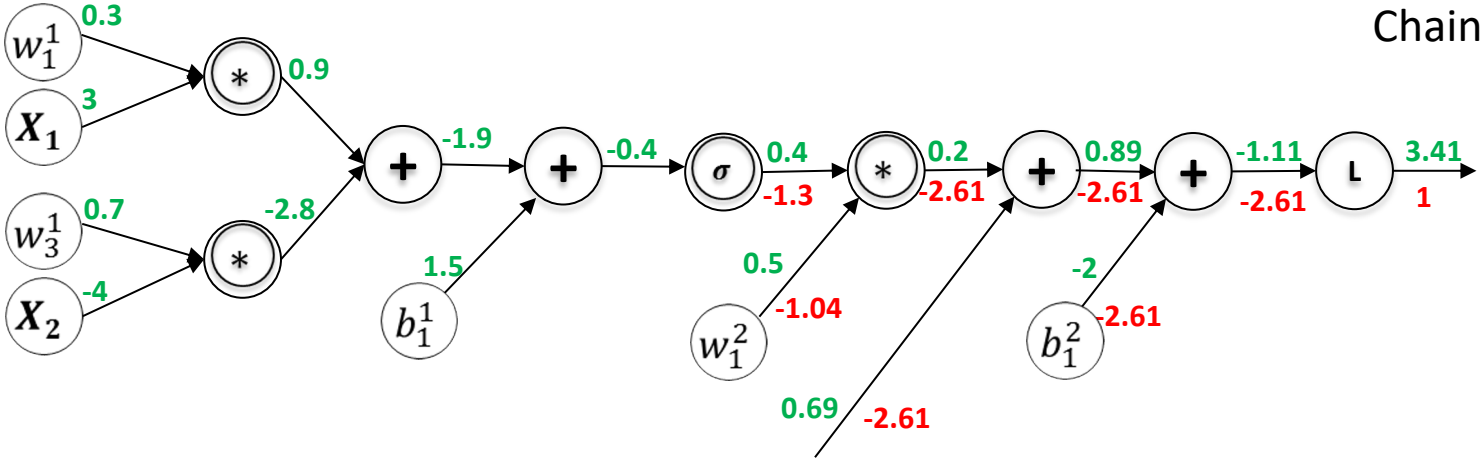
Example Backward Pass



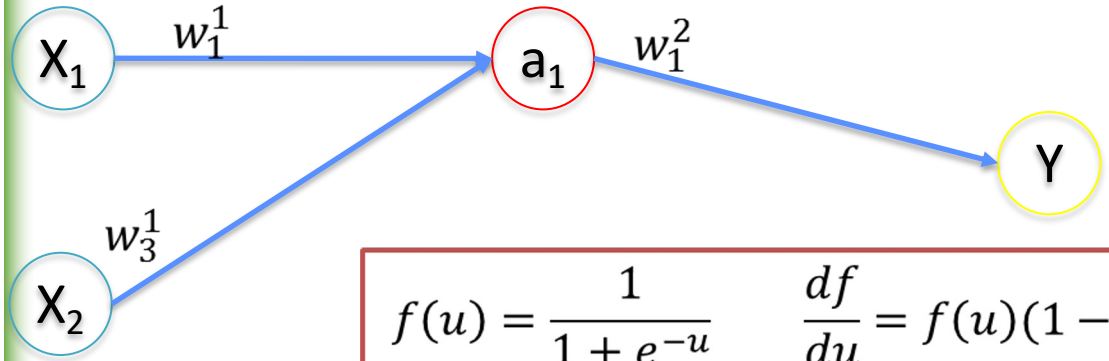
$$f(u) = c * u \qquad \frac{df}{du} = c$$

$$0.5 * -2.61 = -1.30$$
$$0.4 * -2.61 = -1.04$$

Chain Rule



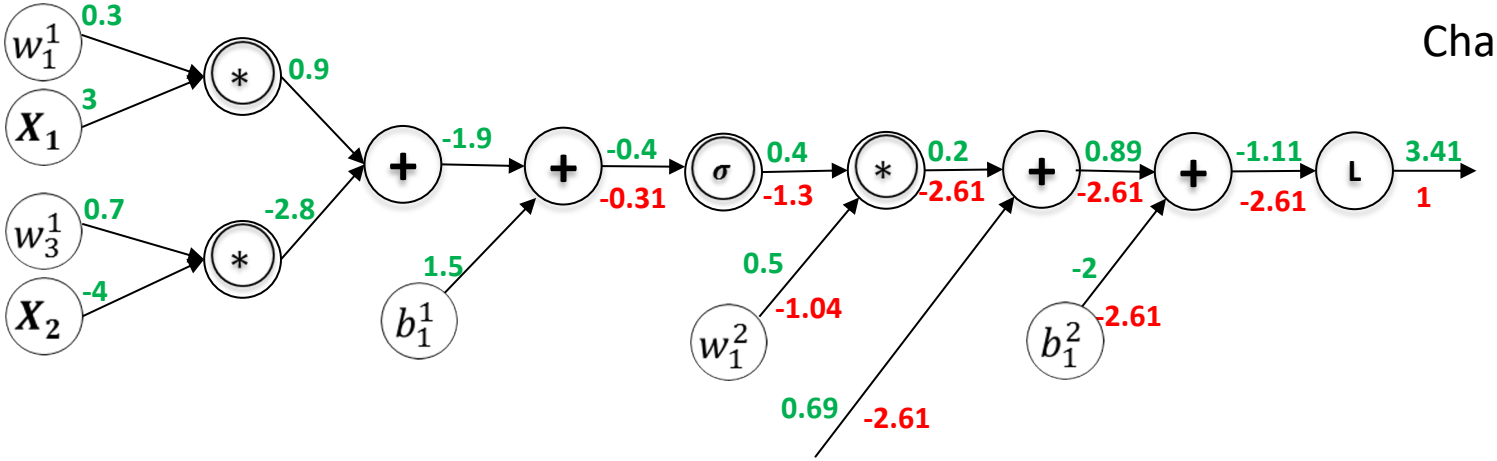
Example Backward Pass



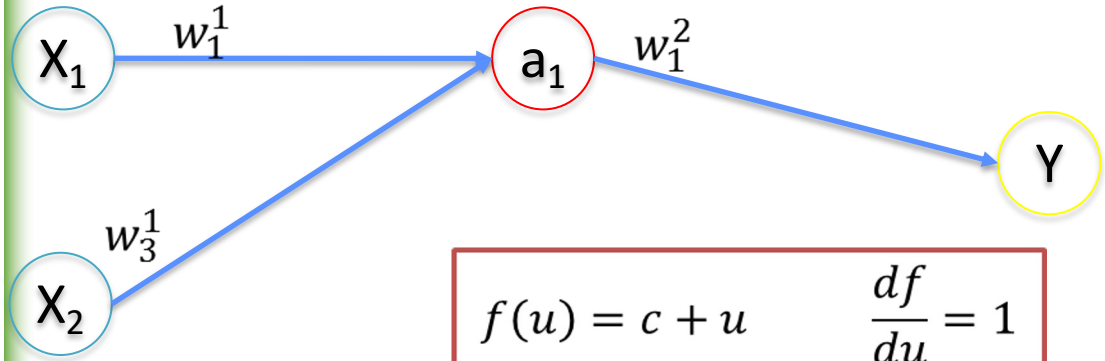
$$f(u) = \frac{1}{1+e^{-u}} \qquad \frac{df}{du} = f(u)(1-f(u))$$

$$0.4(1-.4) * -1.3 = -0.31$$

Chain Rule



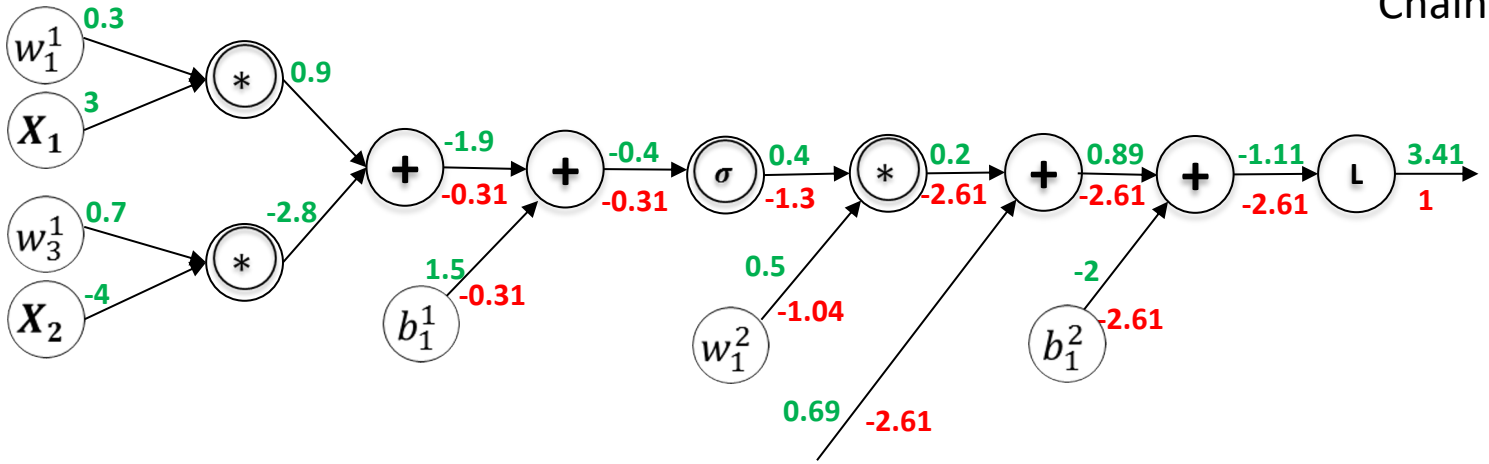
Example Backward Pass



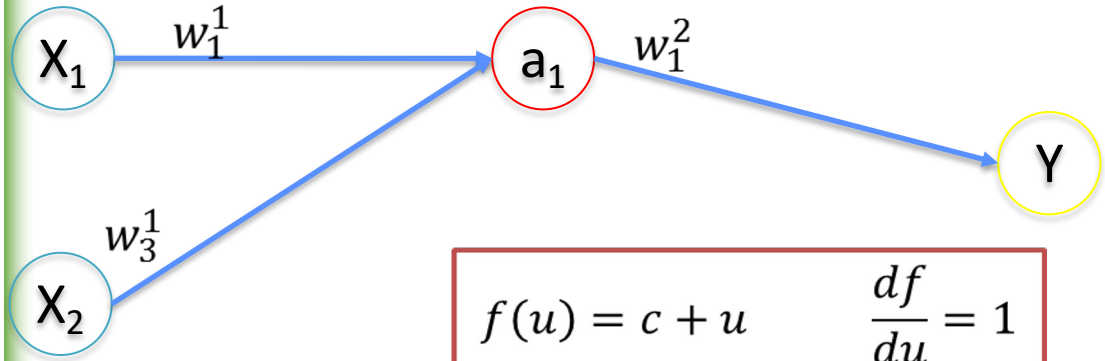
$$f(u) = c + u \qquad \frac{df}{du} = 1$$

$$1 * -0.31 = -0.31$$

Chain Rule



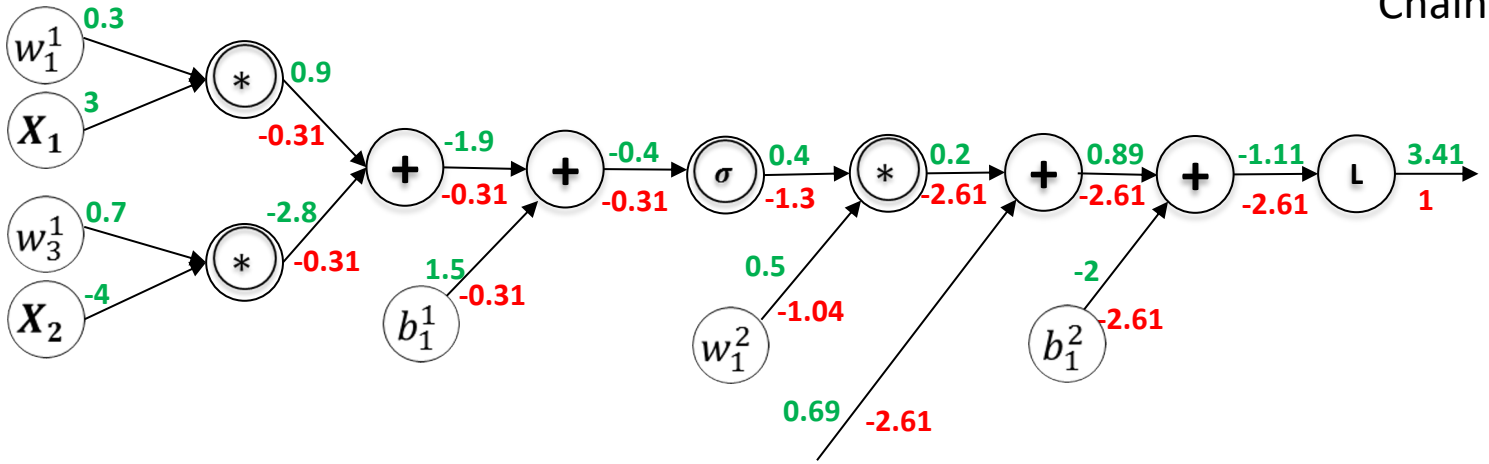
Example Backward Pass



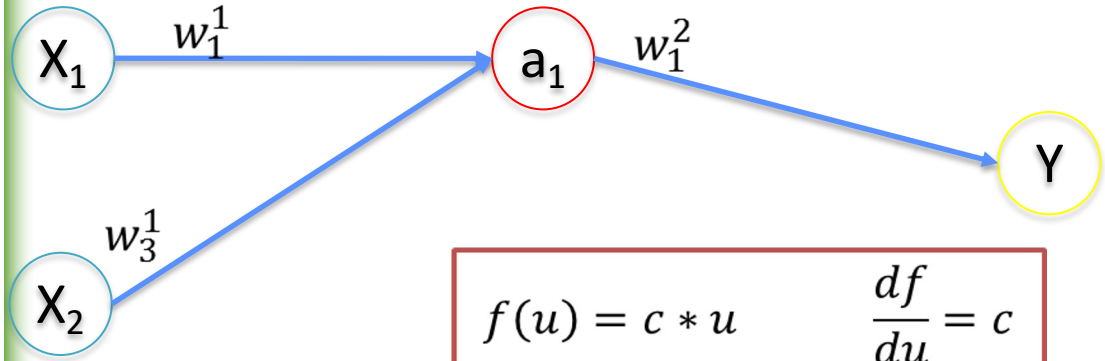
$$f(u) = c + u \qquad \frac{df}{du} = 1$$

$$1 * -0.31 = -0.31$$

Chain Rule



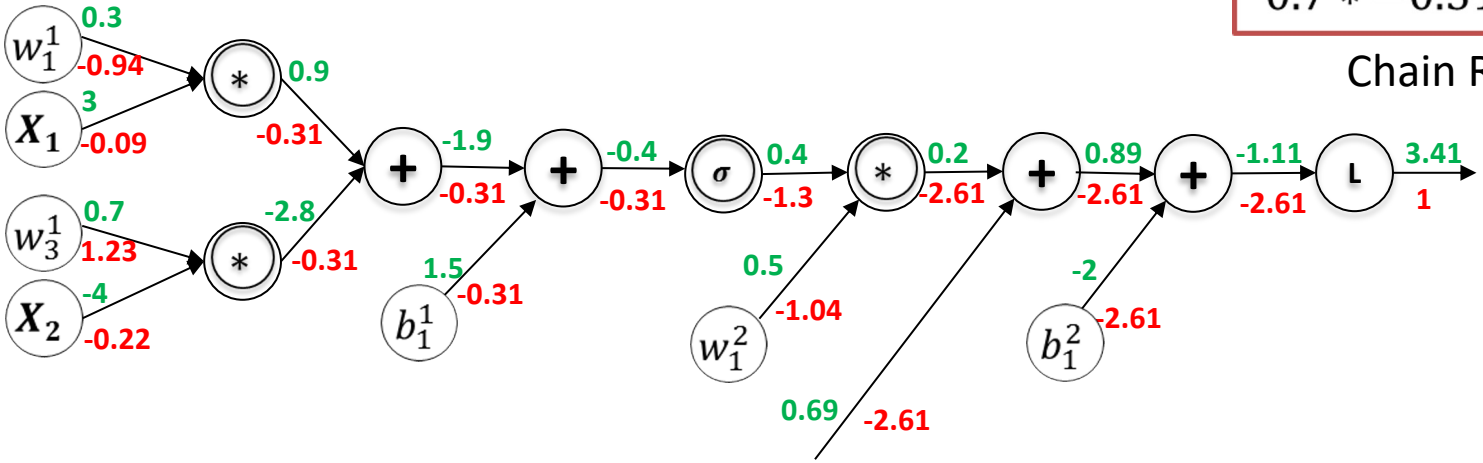
Example Backward Pass



$$f(u) = c * u \qquad \frac{df}{du} = c$$

$$3 * -0.31 = -0.94$$
$$0.3 * -0.31 = -0.09$$
$$-4 * -0.31 = 1.23$$
$$0.7 * -0.31 = -0.22$$

Chain Rule



Example: Update

- Gradient descent update at the $(r+1)$ iteration

$$w^{r+1} = w^r - \gamma \frac{\partial L}{\partial w}$$

$$b^{r+1} = b^r - \gamma \frac{\partial L}{\partial b}$$

$$\gamma = 0.9$$

- $w_1^1 = 0.3 - 0.9(-0.94) = 1.15$
- $w_3^1 = 0.7 - 0.9(1.23) = -0.41$
- $b_1^1 = 1.5 - 0.9(-0.31) = 1.78$
- …

Appendix: Policy gradient proof

Proof

- The objective is an expectation. Want to compute the gradient w.r.t. θ (simplify notation from: $V(\pi_\theta)$ to $V(\theta)$). First, bring the gradient to the inside.

$$\nabla_\theta V(\theta) = \nabla_\theta \mathbb{E}_\tau[R(\tau)] = \nabla_\theta \int \mathbb{P}(\tau|\pi_\theta, M) R(\tau) d\tau$$

Log trick

$$\nabla_\theta \log \mathbb{P}(\tau|\pi_\theta, M) = \frac{\nabla_\theta \mathbb{P}(\tau|\pi_\theta, M)}{\mathbb{P}(\tau|\pi_\theta, M)}$$

$$= \int \nabla_\theta \mathbb{P}(\tau|\pi_\theta, M) R(\tau) d\tau$$

$$= \int \mathbb{P}(\tau|\pi_\theta, M) \nabla_\theta \log \mathbb{P}(\tau|\pi_\theta, M) \, R(\tau) d\tau$$

$$= \mathbb{E}_\tau[R(\tau) \nabla_\theta \log \mathbb{P}(\tau|\pi_\theta, M)]$$

- Last expression is an unbiased gradient estimator
  Just sample $\tau_t \sim \mathbb{P}(\tau|\pi_\theta, M)$, and compute $\hat{g}_t = R(\tau_t) \nabla_\theta \log \mathbb{P}(\tau_t|\pi_\theta, M)$
- Issue: Need to be able to compute & differentiate the density $\mathbb{P}(\tau|\pi_\theta, M)$ w.r.t θ

Proof

Likelihood (with stochastic policies)

$$\mathbb{P}(\tau|\pi_\theta, M) = \mu(s_0) \prod_{t=0}^{T-1} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

$$\log \mathbb{P}(\tau|\pi_\theta, M) = \log \mu(s_0) + \sum_{t=0}^{T-1} \log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t)$$

$$\nabla_\theta \log \mathbb{P}(\tau|\pi_\theta, M) = \nabla_\theta \log \mu(s_0) + \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) + \nabla_\theta \log p(s_{t+1}|s_t, a_t)$$

0 0

→ model free

Alternative proof: likelihood rescaling

- Interested in policy gradient: $\nabla_\Delta V(\theta+\Delta)|_{\Delta=0}$
- Likelihood rescaling

$$V(\theta+\Delta) = \mathbb{E}_{\tau(\theta)}\left[R(\tau(\theta))\frac{\prod_t \pi_{\theta+\Delta}(a_t|s_t)}{\prod_t \pi_\theta(a_t|s_t)}\right]$$

- Apply chain rule to get

$$\nabla_\Delta V(\theta+\Delta)\Big|_{\Delta=0} = \mathbb{E}_{\tau(\theta)}\left[R(\tau(\theta))\sum_t \frac{\nabla \pi_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)}\right]$$
$$= \mathbb{E}_\tau[R(\tau)\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t)]$$