

Deep learning

Function approximation for large state spaces

Cathy Wu

1.041/1.200 Transportation: Foundations and Methods

Readings

1. Michael Nielsen. **Neural Networks and Deep Learning**. 2019.
[[URL](#)]
 - Chapter 2: How the backpropagation algorithm works

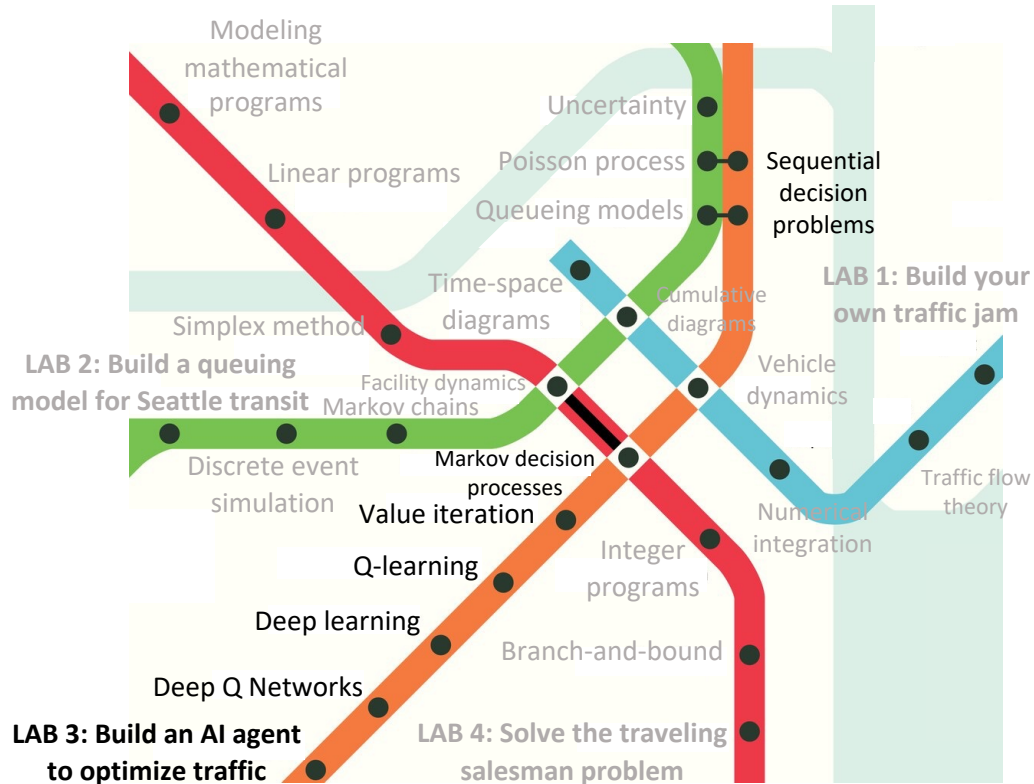
Unit 3: Machine learning for traffic control



Unit 3

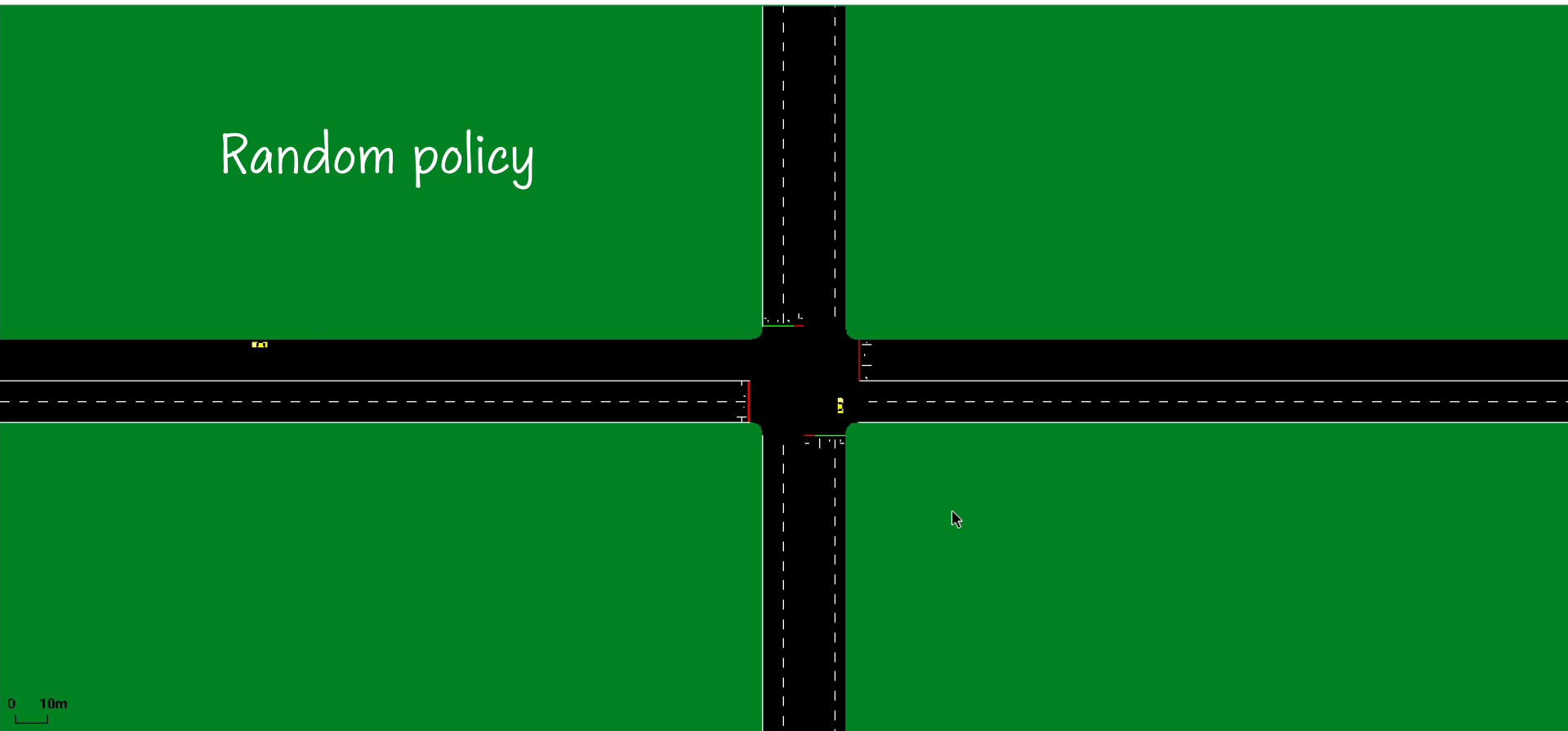
Optimizing

Multi-stage

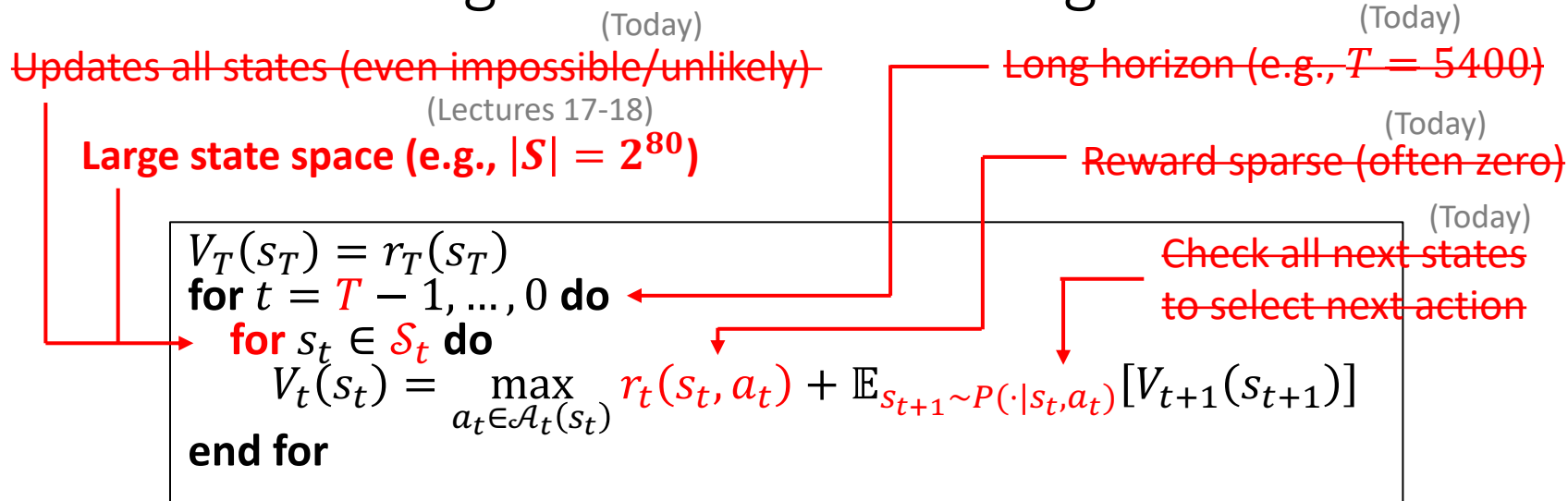


CL3: Build an AI agent to optimize traffic

Random policy



DP for traffic signal control: challenges



- Dynamic programming: $O(|S|^2|A|T) = 2^{80 \times 2} \times 4 \times 5400$
 - Not so efficient ☹️
- Parts that are (surprisingly) OK
 - DP recursion
 - Action space usually small

Outline

1. Motivation: function approximation for large state spaces
2. Gradient descent (GD)
3. Deep neural networks (DNN)
4. Training a DNN: SGD + Backpropagation

Outline

1. **Motivation: function approximation for large state spaces**
2. Gradient descent (GD)
3. Deep neural networks (DNN)
4. Training a DNN: SGD + Backpropagation

Challenge: large state spaces

- Issue: Too many states to visit & update for many problems of interest.
 - E.g. traffic signal control, Go, continuous state.

“function approximation setting”

Approximate Q function

$Q_{\theta}(s, a)$

Function approximation
e.g. linear, deep neural networks

Desired: θ such that $Q_{\theta}(\text{green}, \uparrow) \approx 8.7$

AND reasonable values for unseen states
and states not visited enough

Critical assumption: Similar states have similar action-values

Q-value table

“tabular setting”

2.5	1.4	3.2	5.4
1.0	3.2	5.1	6.3
5.2	4.2	5.5	7.2
8.7	3.4	2.0	8.0
4.8	2.5	3.5	4.2
1.0	3.0	3.3	1.2
-180	-172	-99.7	-150
4.2	2.1	3.2	3.7
2.1	2.0	3.7	3.1
3.0	1.2	3.2	2.7
0.1	1.5	0.1	1.0

$Q(s, a)$

s

a

↑ ↓ ← →

Solution approach: leverage deep learning

- Desired: a correct and consistent estimate of long-term value

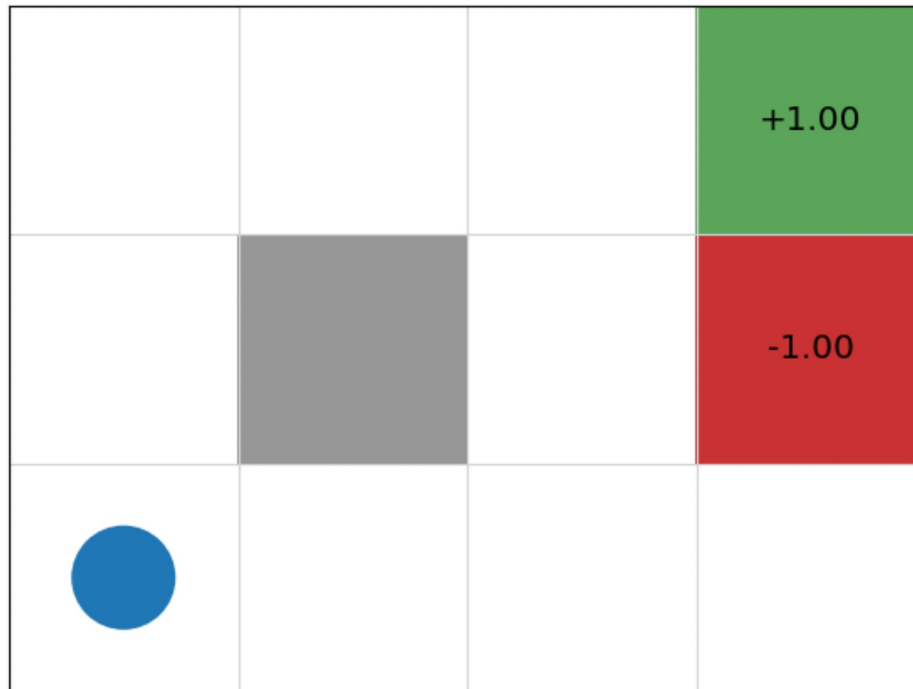
$$Q(s, a) \approx \max_{a' \in A} r(s, a) + \gamma \mathbb{E}_{s' \sim P(\cdot | s, a)} [Q(s', a')]$$

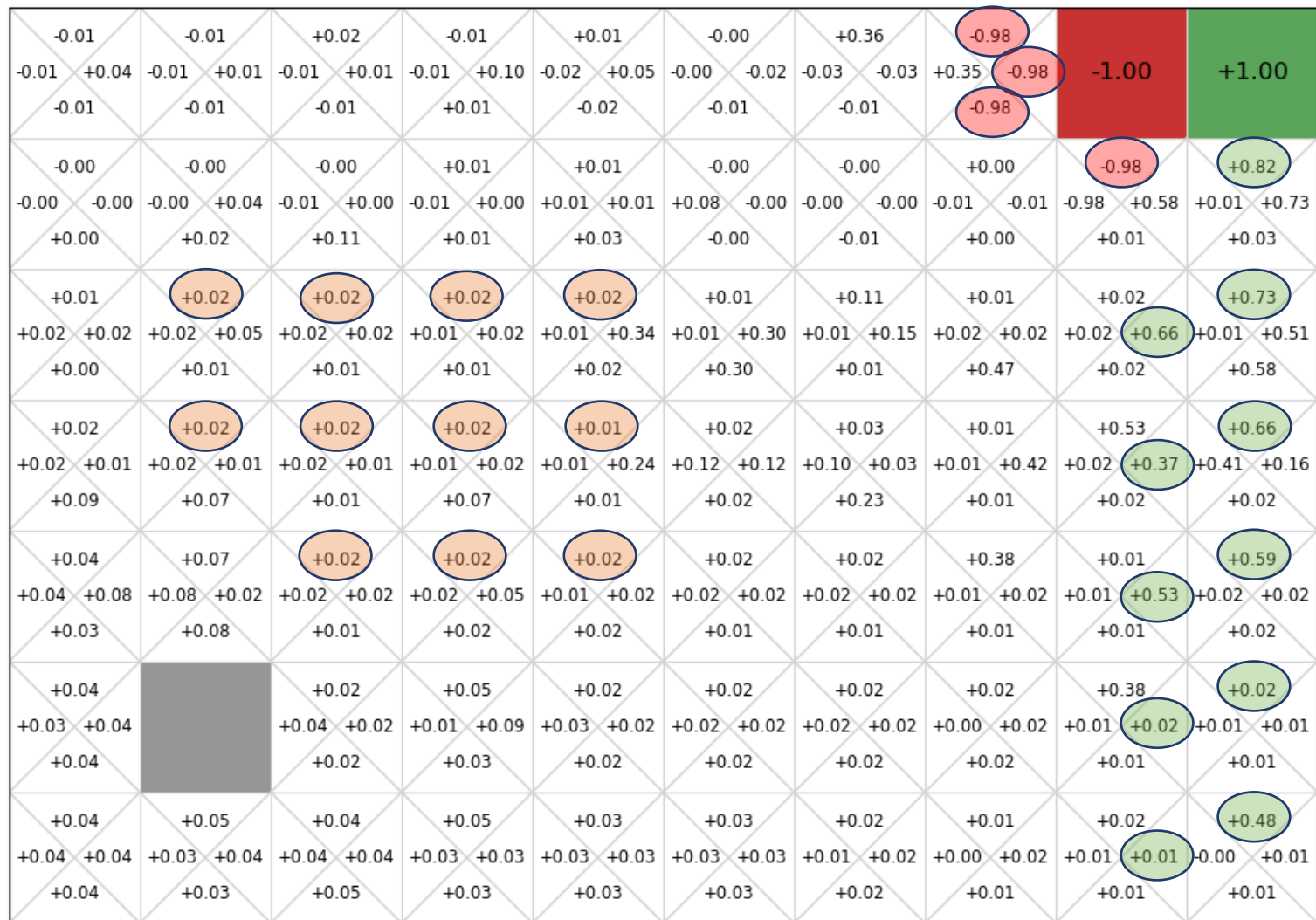
- Deep learning approach (rough idea)
 1. Approximate $Q(s, a)$ with a function of parameters θ , i.e., $Q_\theta(s, a)$
 - → Deep neural networks
 2. Collect a batch of data D , e.g. tuples $(s, a, r) \in D$
 3. Use $Q_\theta(s, a)$ to predict $Q(s, a)$ for each tuple in D
 4. If close, great! If not, update θ to reduce the error $|Q_\theta(s, a) - Q(s, a)|$
 - → Stochastic gradient descent + Backpropagation algorithms
 5. Collect another batch of data and repeat from step 3.

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.





Function approximation for large state spaces

- Main idea: Usually, we do not need see all $|S||A|$ state-action pairs to accurately represent a good Q function.
 - Ex (grid world): $|S||A| = 7 \times 10 \times 4 = 280$
- Tabular methods: The Q-table can be seen as a function expressing Q with as many parameters as there are state-action pairs
 - Recall: $Q: S \times A \rightarrow \mathbb{R}$ and 1_α is the indicator function where α is true.
 - Ex. $\tilde{Q}((x, y), a) = 1_{\{(x,y)=(0,0),a=\text{up}\}} 0.04 + 1_{\{(x,y)=(0,0),a=\text{down}\}} 0.04 + \dots + 1_{\{(x,y)=(1,0),a=\text{up}\}} 0.05$

Function approximation for large state spaces

- Function approximation setting: $\tilde{Q}_\theta: S \times A \rightarrow \mathbb{R}$
 - where typically $|\theta| \ll |S||A|$
 - Versus tabular function: $|\theta| = |S||A|$
 - Ex. (linear function). $\tilde{Q}_\theta((x, y), a) = a_0 + b_1x + b_2y + b_3a$
 - $\theta = (a_0, b_1, b_2, b_3)$, $|\theta| = 4$
 - Ex (polynomial function, 3rd degree). $\tilde{Q}_\theta((x, y), a) = a_0 + b_1x + b_2y + b_3a + c_1xy + c_2xa + c_3ya + c_4x^2 + c_5y^2 + c_6a^2 + d_1xya + d_2x^2y + d_3xy^2 + d_4xa^2 + d_5ya^2 + d_6x^2a + d_7y^2a + d_8x^3 + d_9y^3 + d_{10}a^3$
 - $\theta = (a_0, b_1, b_2, b_3, c_1, c_2, c_3, c_4, c_5, c_6, d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8, d_9, d_{10})$, $|\theta| = 20$

→	→	↑	→	→	↑	↑	←	-1.00	+1.00
↓	→	↓	↑	↓	←	→	↓	→	↑
→	→	↑	↑	→	→	→	↓	→	↑
↓	↓	←	↓	→	→	↓	→	↑	↑
→	↓	→	→	↑	↑	→	↑	→	↑
↑		←	→	←	↑	↑	↑	↑	↑
→	↑	↓	↑	↓	→	↑	→	↑	↑

Function Approximators

Common function approximators:

- Polynomial

- $Q(s, a) \approx p_n(f_1, f_2, \dots, f_m)$ where
 - p_n is an n-th degree polynomial
 - Features f_m are functions of the state and action
- For example, $p_2(f_1, f_2) = \theta_1 + \theta_2 f_1 + \theta_3 f_2 + \theta_4 f_1 f_2 + \theta_5 f_1^2 + \theta_6 f_2^2$
- Universal: any continuous function can be represented by a polynomial
- An n-th degree polynomial with m features is exponential in n number of parameters...

- Linear

- More powerful than they seem
- Can have linear functions with nonlinear features
- $Q(s, a) \approx p_1(f_1, f_2, \dots, f_m)$

- Neural networks

- Highly popular now
- Seems to “select” features automatically
- **What we'll discuss next**

Outline

1. Motivation: function approximation for large state spaces
2. **Gradient descent (GD)**
 - a. Stochastic gradient descent (SGD)
3. Deep neural networks (DNN)
4. Training a DNN: SGD + Backpropagation

Gradient descent

Main idea: iteratively minimize an (error) function.

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

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

Gradient descent algorithm (sketch)

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

Preview: Recall (L16)

- Q-learning update:

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

- We will use $R_k := r(s, a) + \gamma \max_{a'} Q_k(s', a')$

Recall: single-variate (1D) calculus

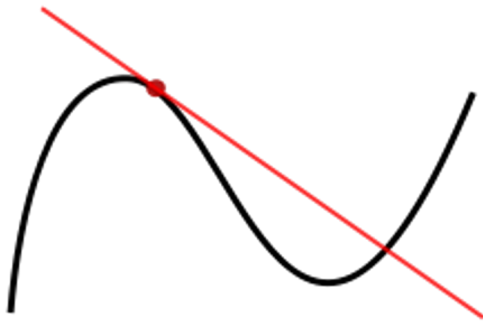
- i.e. $|\theta| = 1$

- Want: $\frac{d}{d\theta} f_{\theta}(x)$

[x is the data, constants]

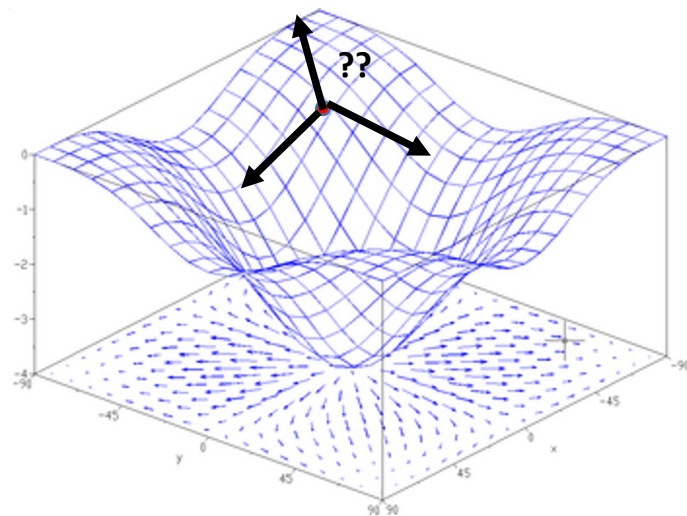
- Rate of change in $+\theta$ direction

- NOT: $\frac{df}{dx}$



- What about multiple directions to move in?

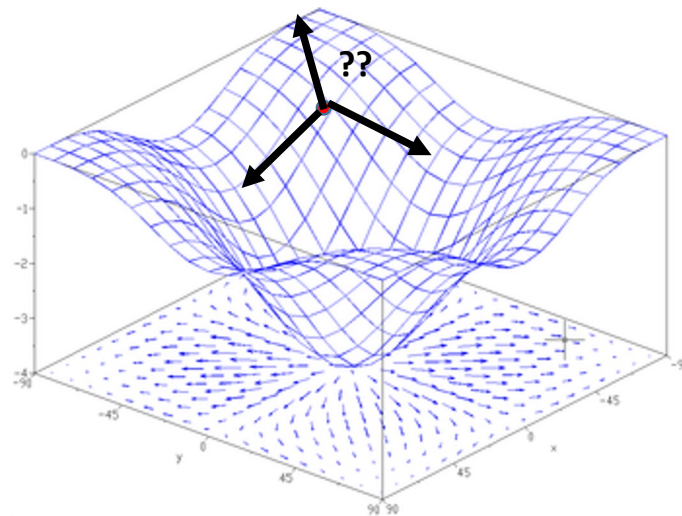
$|\theta| > 1$



Multi-variate gradients

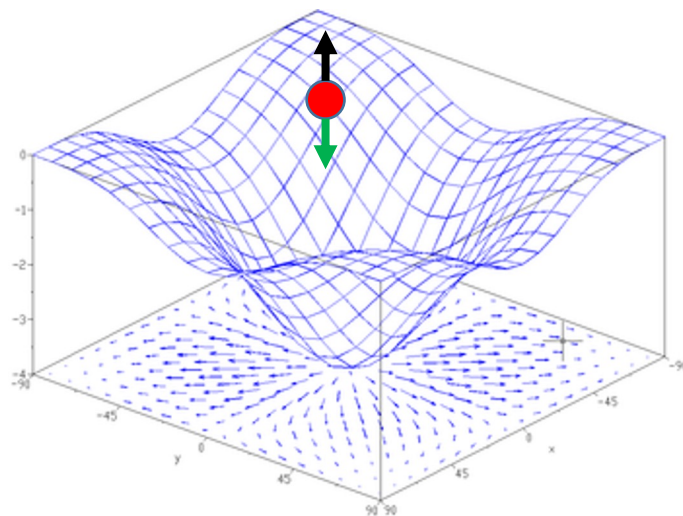
- Define: $\nabla_{\theta} f_{\theta}(x) = \begin{bmatrix} \frac{df}{d\theta_1} \\ \frac{df}{d\theta_2} \\ \vdots \\ \frac{df}{d\theta_{|\theta|}} \end{bmatrix}$

- 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

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

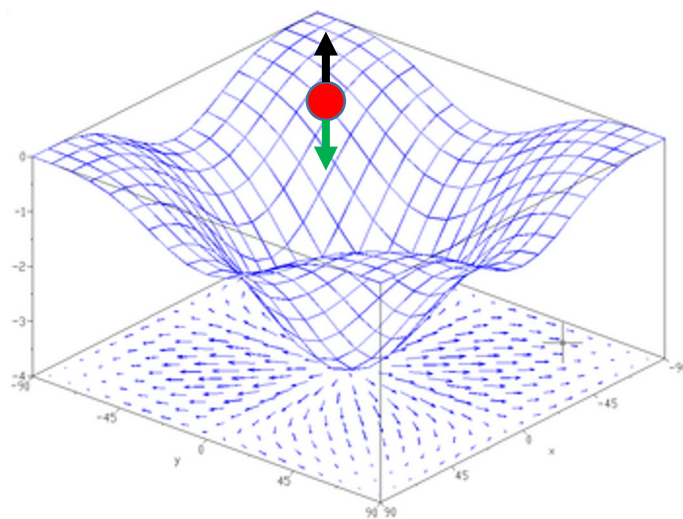
$$g(\theta_1, \theta_2) = -2\theta_1\theta_2$$

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

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

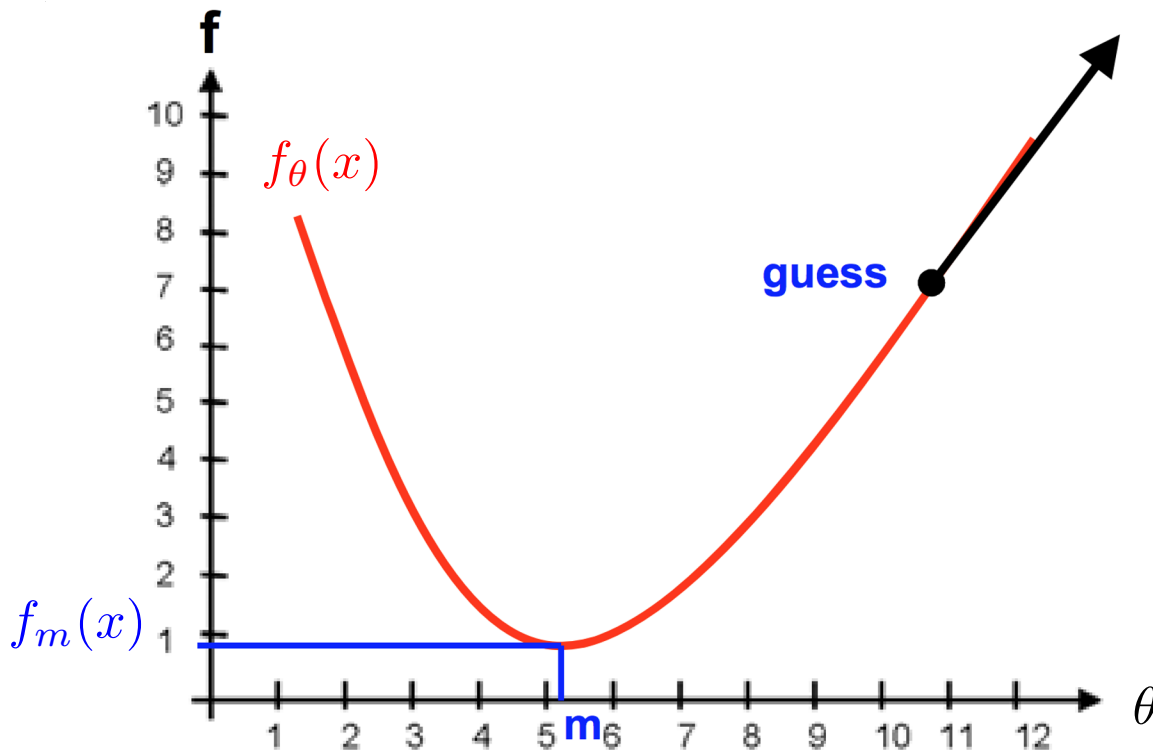
Gradient descent

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



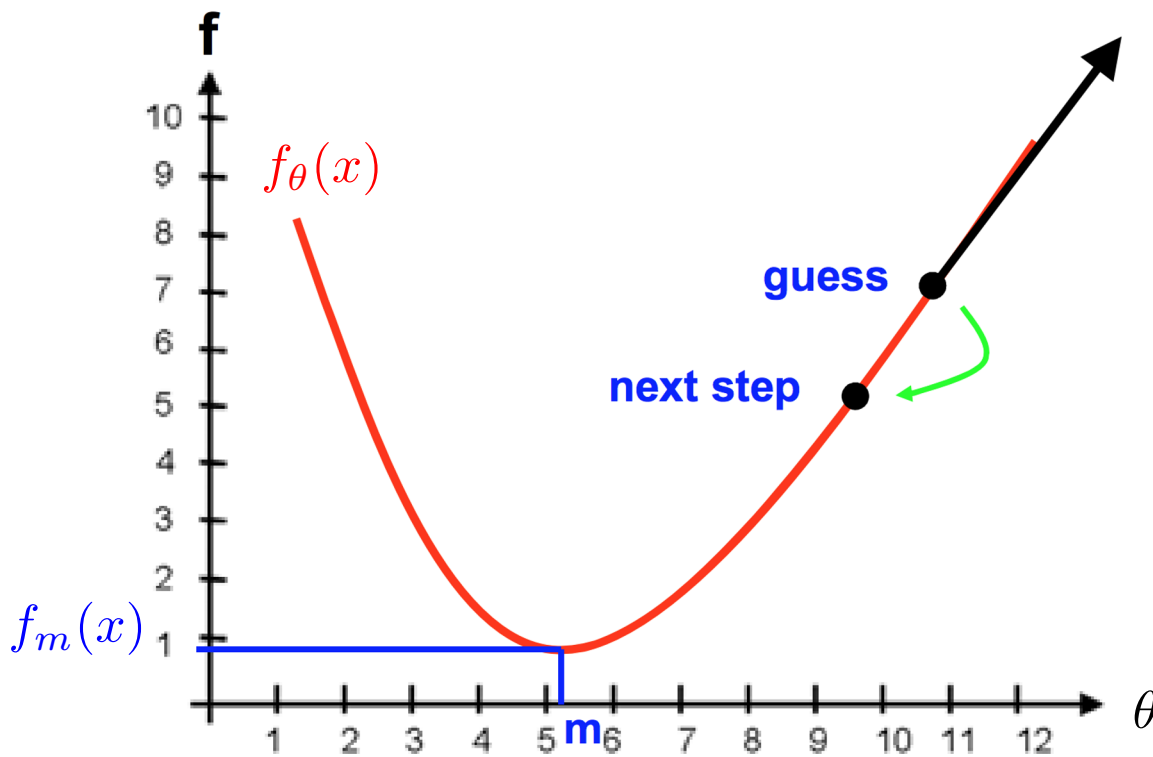
Gradient Descent

Find a minimum by following the slope:



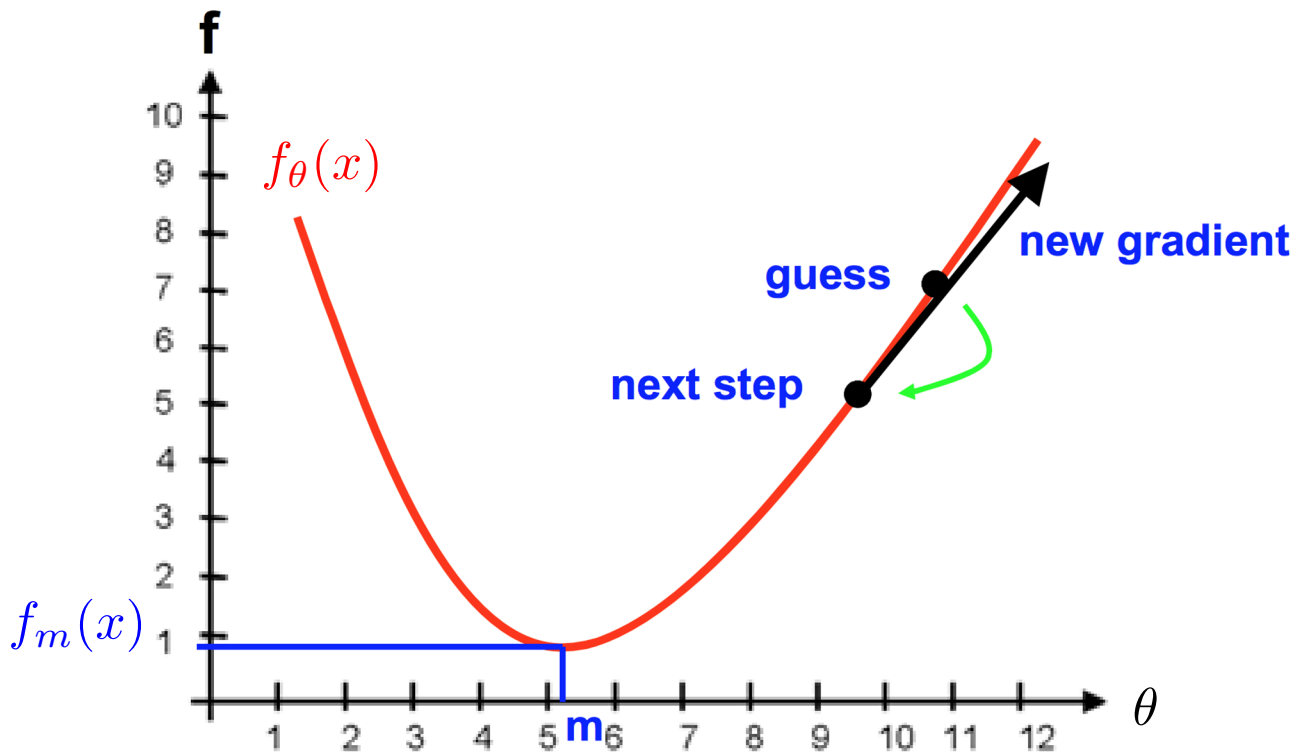
Gradient Descent

Find a minimum by following the slope:



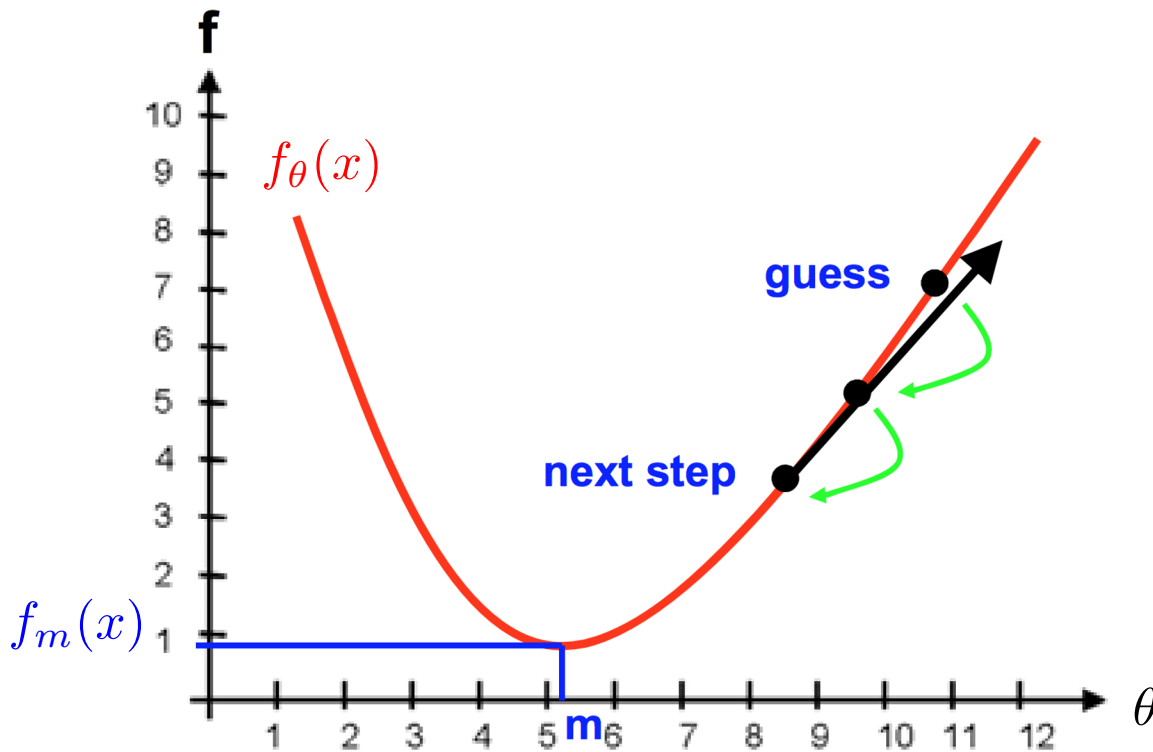
Gradient Descent

Find a minimum by following the slope:



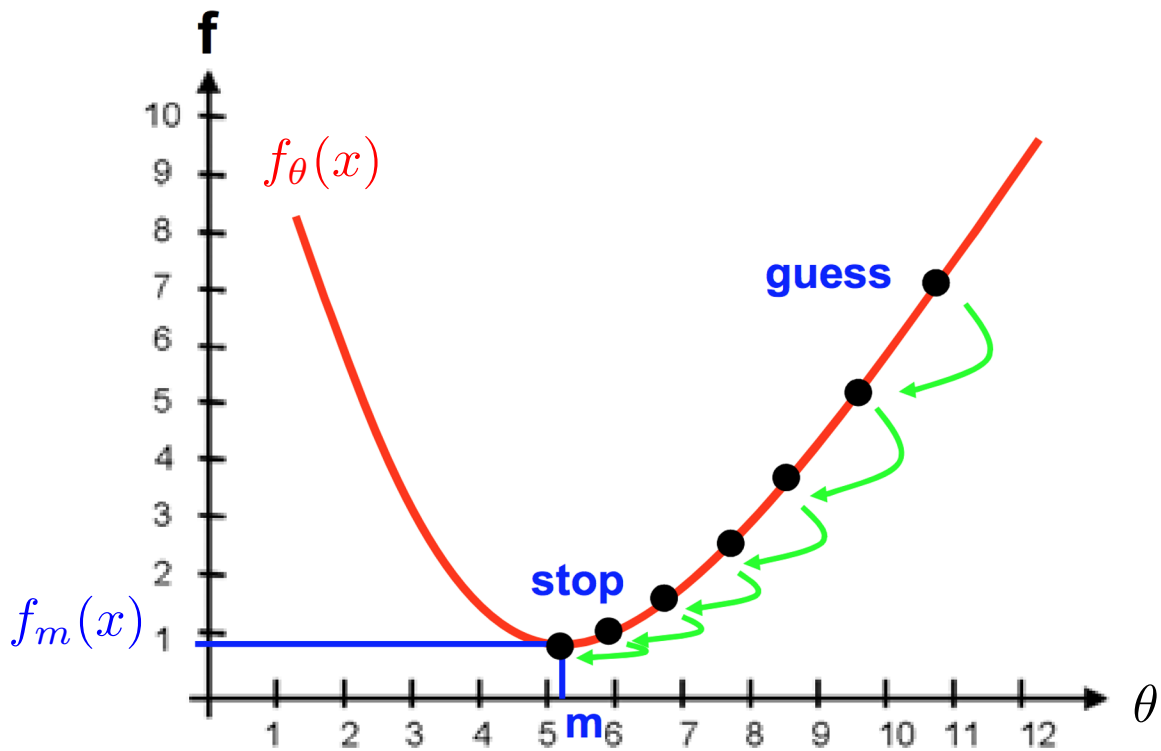
Gradient Descent

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.

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

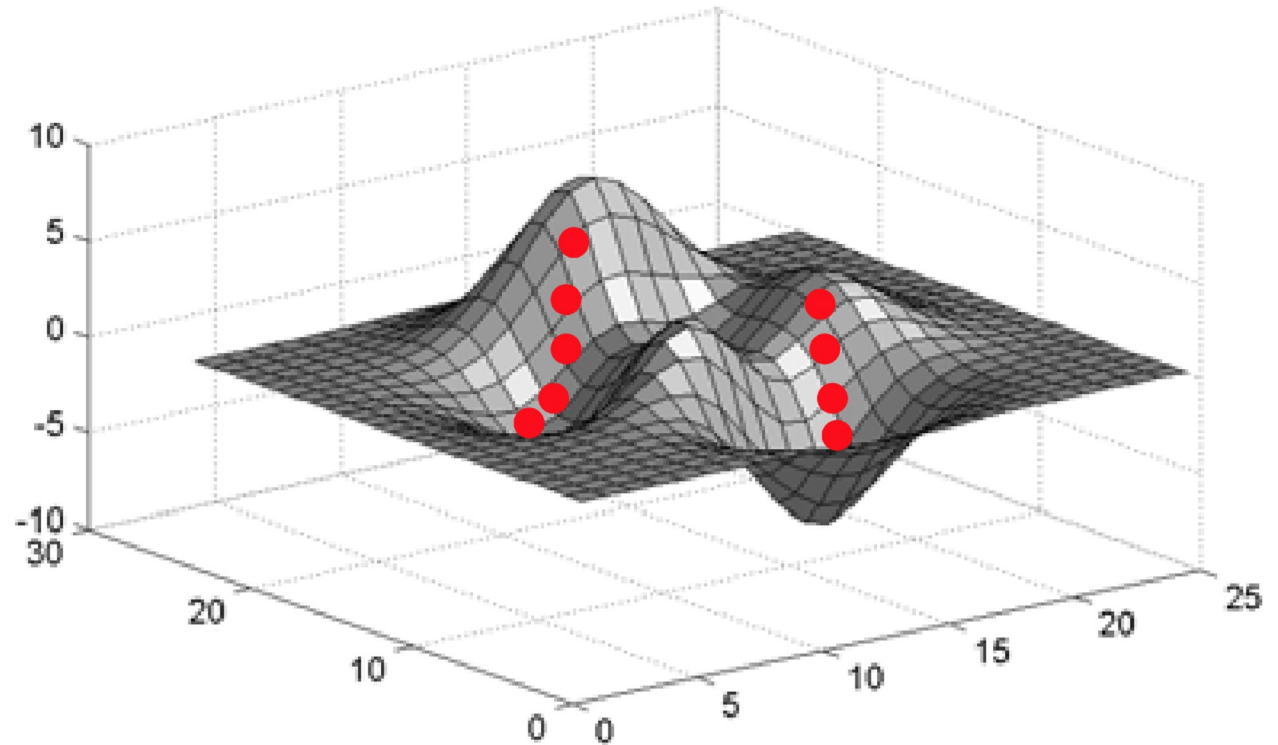
where $x = (s_k, a_k, R_k)_{k \in [N]}$

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 (gradient close to zero)

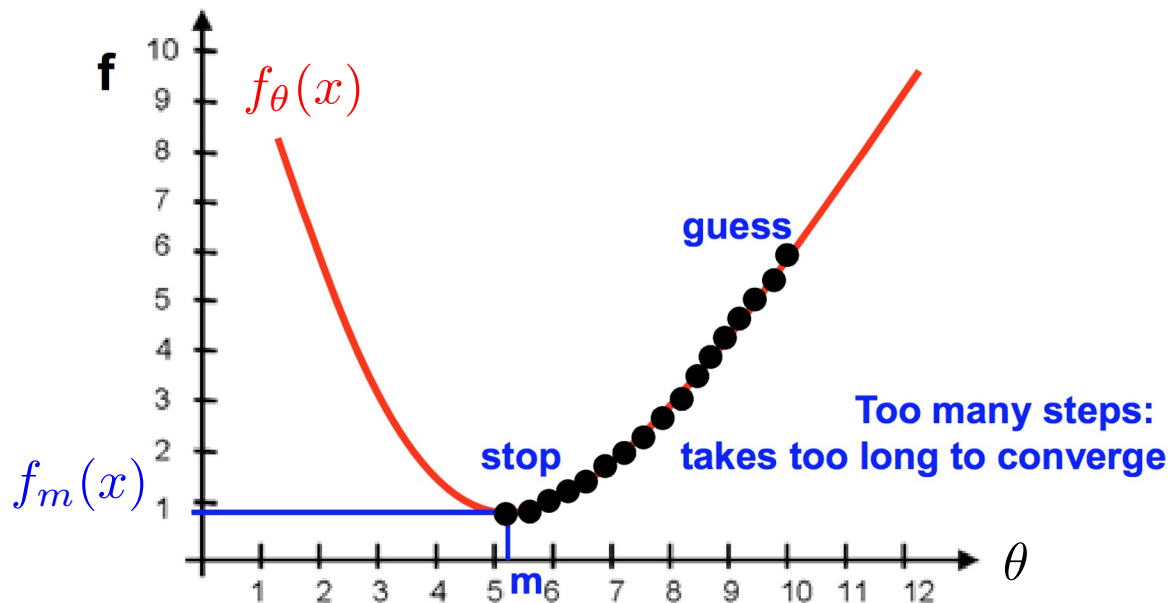
Gradient Descent

- Works just as well in higher dimensions



Picking a step

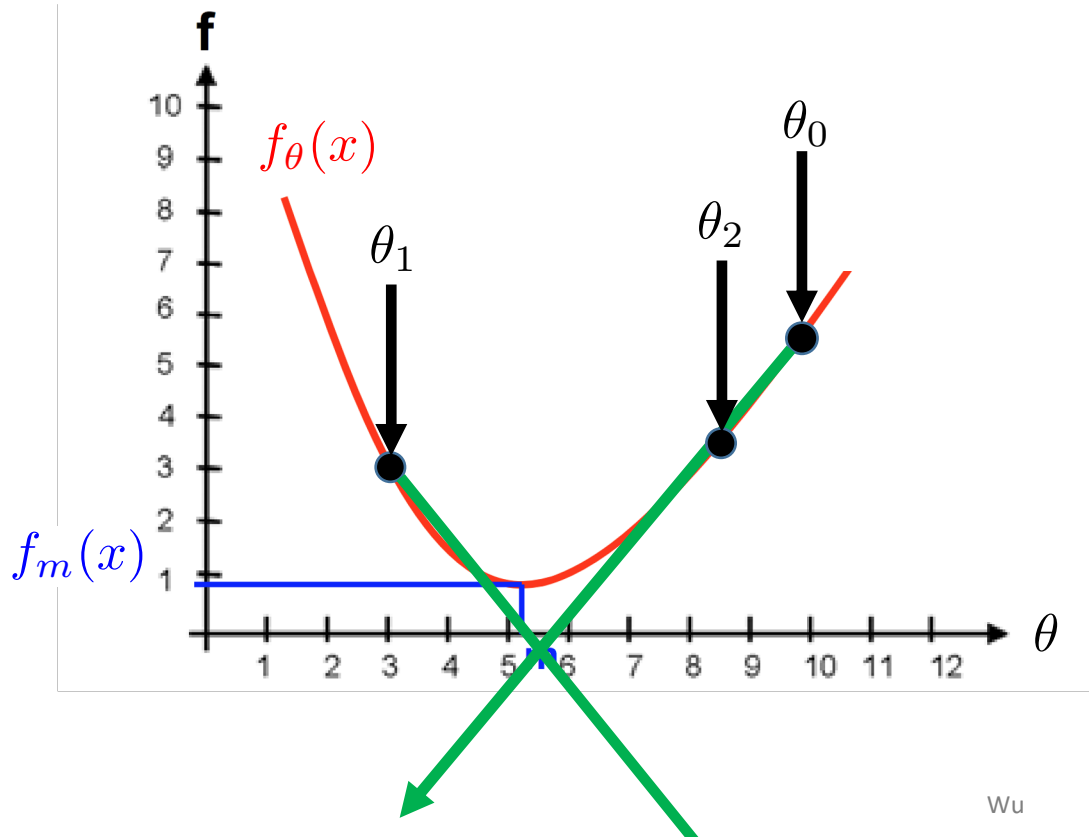
- Small steps: too slow!



Picking a step

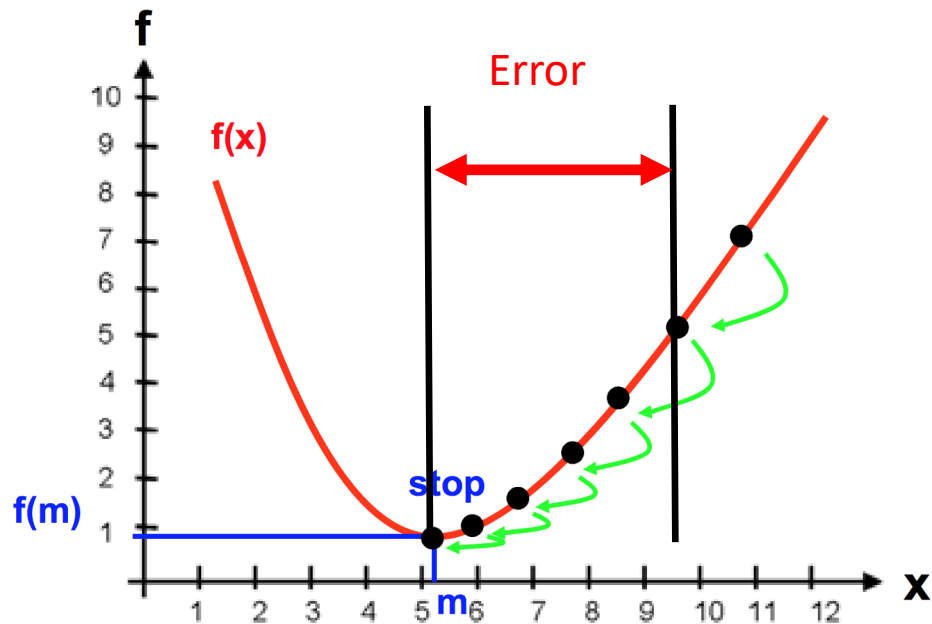
- Large steps: overshoots!

- Takes a while
- Possibly unstable!
Goes to infinity



Picking a step

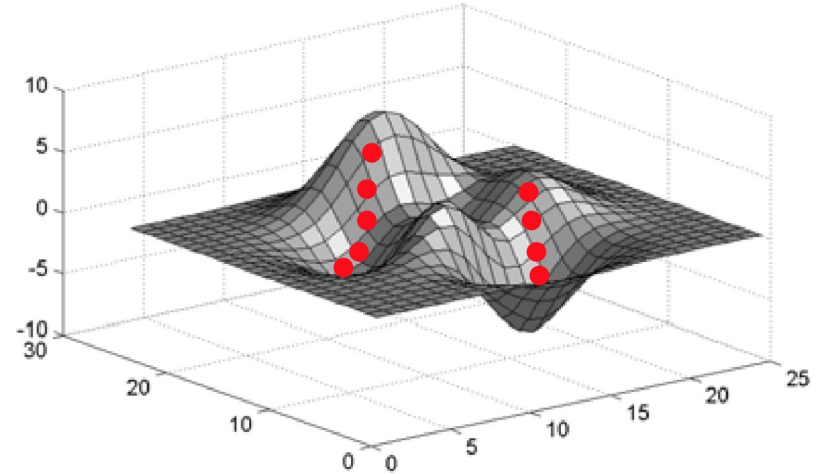
- In practice:
 - Many algorithms & theory to help with this
 - “Just start with .01”
 - Decrease it if too large (unstable updates)
 - Increase it if too small (slow improvement)



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

Stochastic Gradient Descent (SGD)

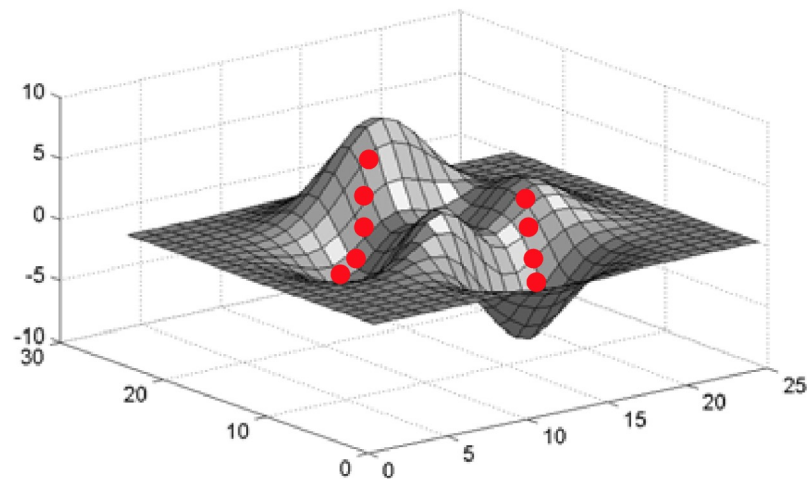
- Goal: Fit linear model: $f_{\theta}(x) = \sum_{i=0}^n \theta_i x_i$
- Loss: $L(f(x; \theta), y) = \frac{1}{D} \sum_{j=1}^D (f_{\theta}(x^j) - y_j)^2$
- Gradient: $\nabla L = \frac{2}{D} \sum_{j=1}^D (f_{\theta}(x^j) - y_j) \frac{\partial f_{\theta}(x^j)}{\partial \theta}$
- Each sample “contributes” 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



Outline


1. Motivation: function approximation for large state spaces
2. Gradient descent (GD)
3. **Deep neural networks (DNN)**
 - a. Activation functions
 - b. Backpropagation algorithm
 - c. Vanishing (exploding) gradients
4. Training a DNN: SGD + Backpropagation

Gradient descent with deep neural networks

Main idea: iteratively minimize an (error) function.

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

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



Gradient descent algorithm:

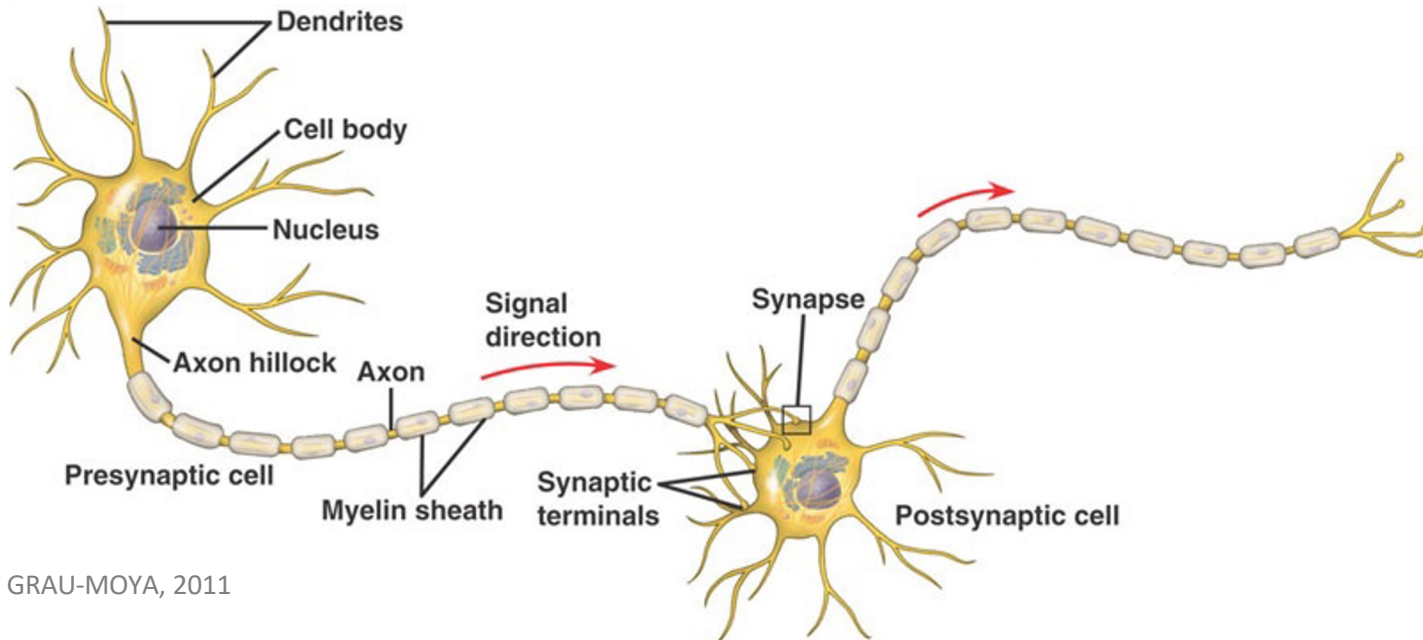
1. Pick a starting θ_0
2. Repeat:
 1. Compute descent direction: $-\nabla_{\theta} f_{\theta}(x)$
 2. Step in the direction: $\theta_{t+1} = \theta_t - \eta \nabla_{\theta} f_{\theta}(x)$
 3. Check if we should stop

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

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

Biological Inspiration

Neuron

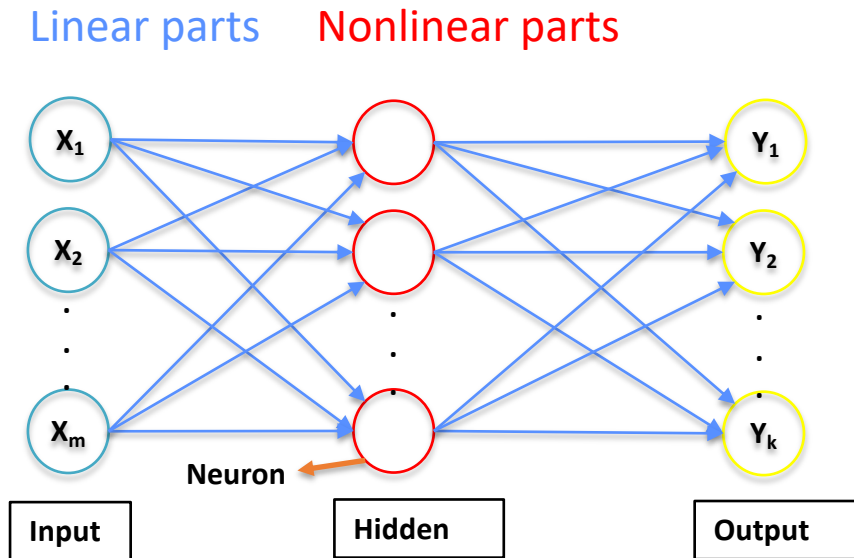


Source: GRAU-MOYA, 2011

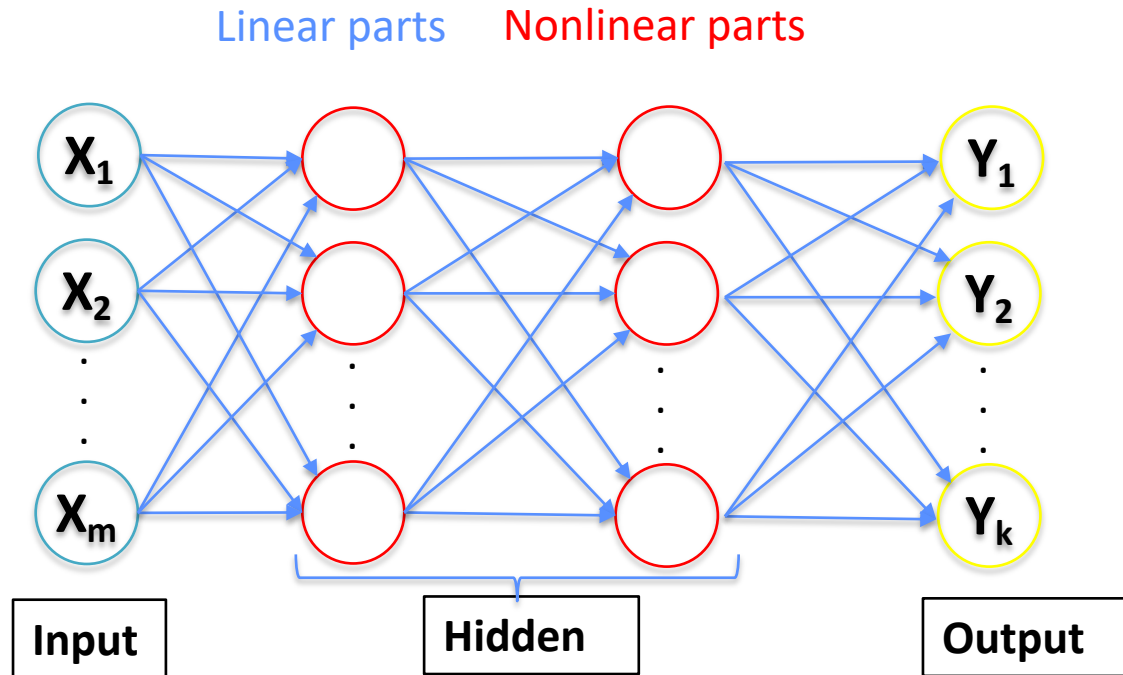
- Neural networks are mostly named so in analogy rather than exactness of function

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

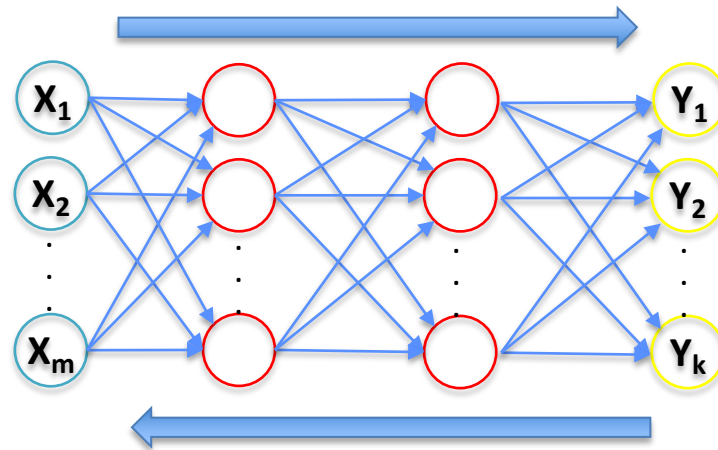


How to estimate the weights and biases of our NN?

Training a NN

0- Initialize the weights and biases

1- Forward pass



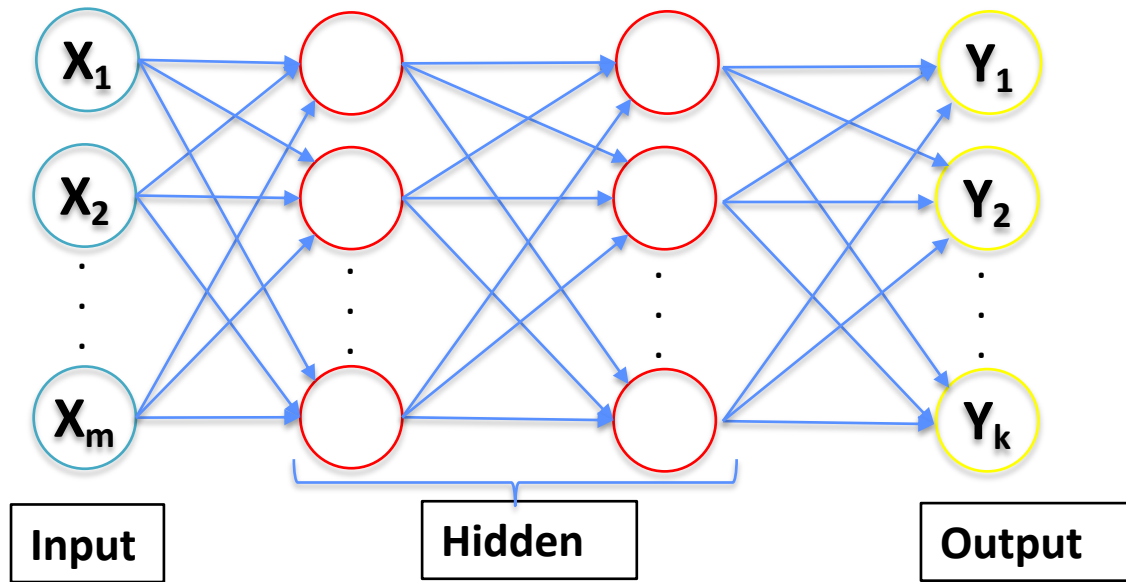
2- Compute the error

4- Update the weights

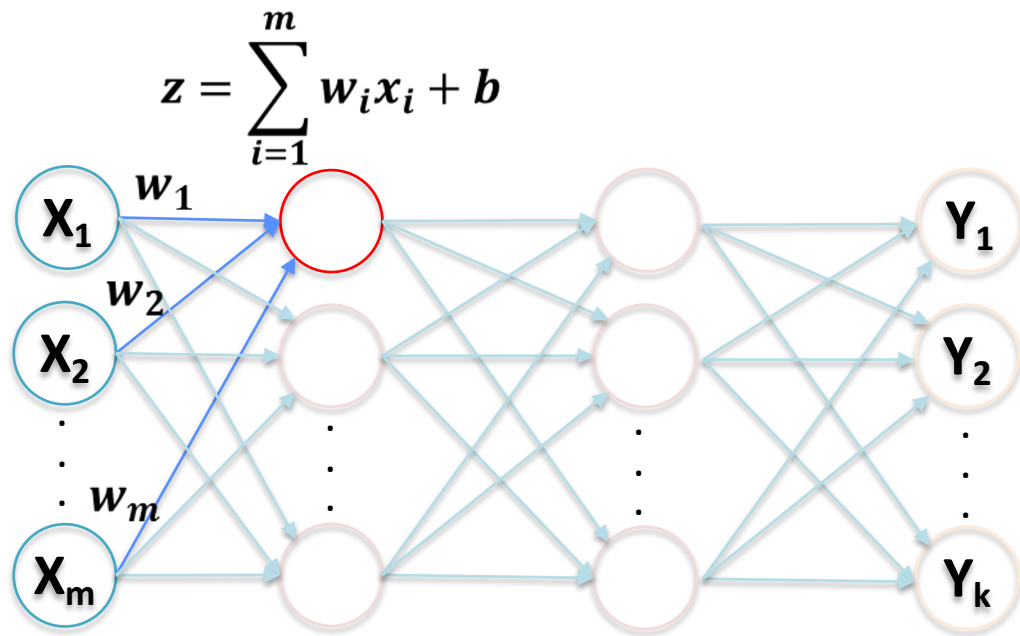
3- Backward pass, compute the gradients

Forward Pass

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

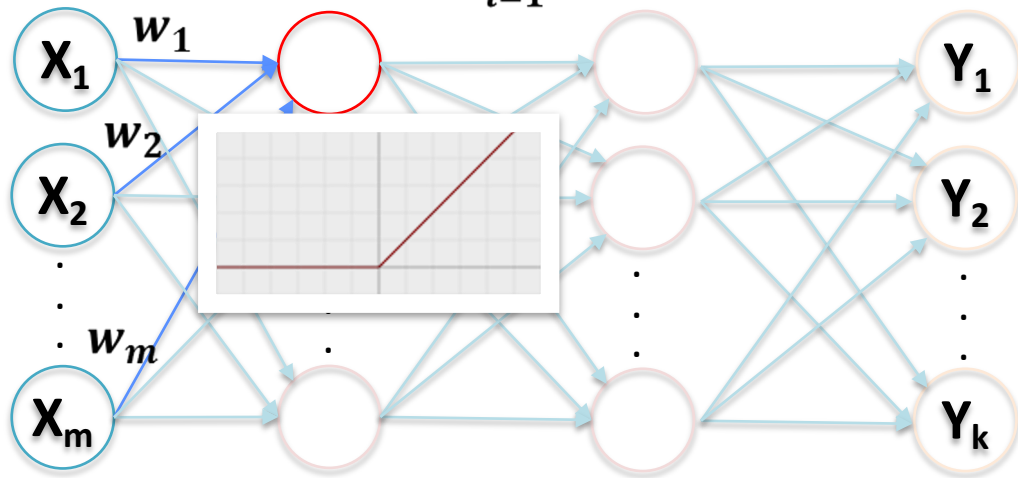


Forward Pass



Forward Pass

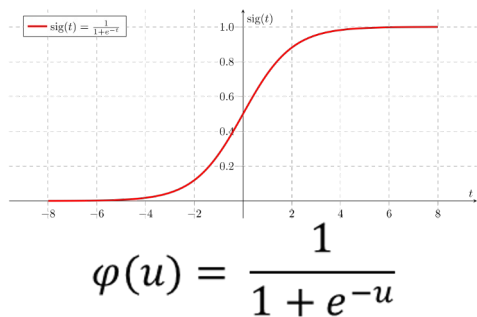
$$a = \varphi(z) = \varphi\left(\sum_{i=1}^m w_i x_i + b\right)$$



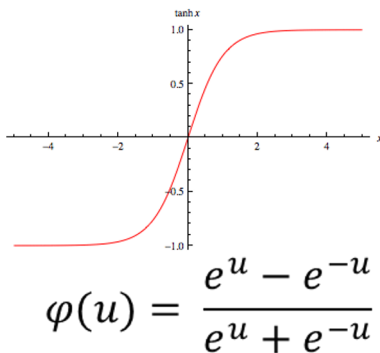
Activation Functions

- Introduce non-linear properties to hidden and output layers

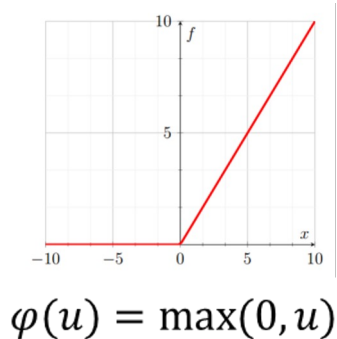
Sigmoid



tanh

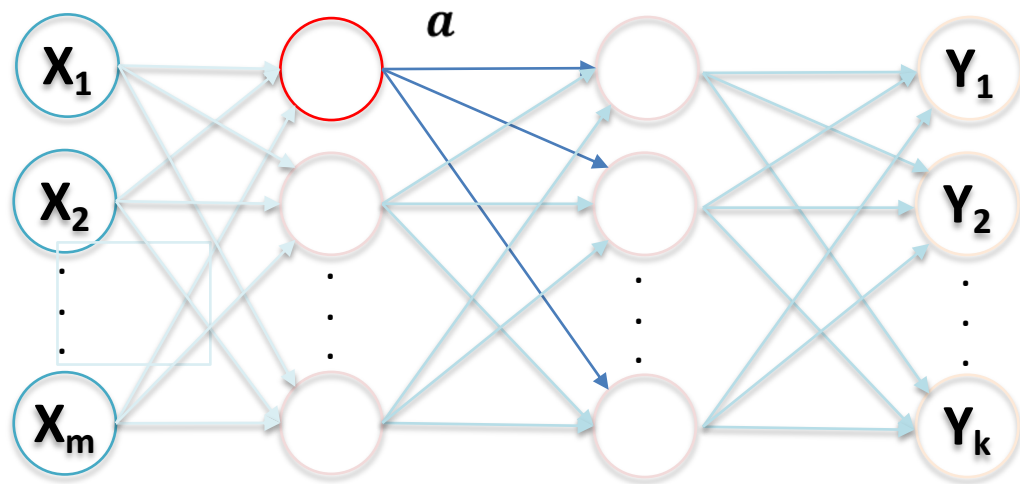


ReLU

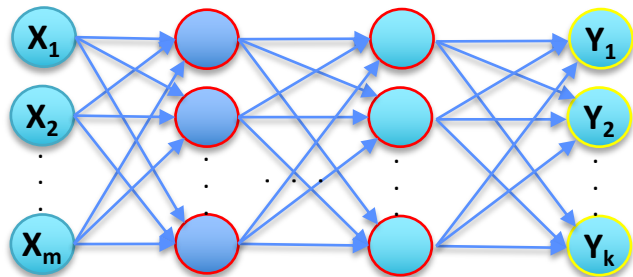


- Historically popular since they have some nice properties (i.e. smooth)
- Saturated neurons kill the gradients
- Does not saturate
- Computationally efficient

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:

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

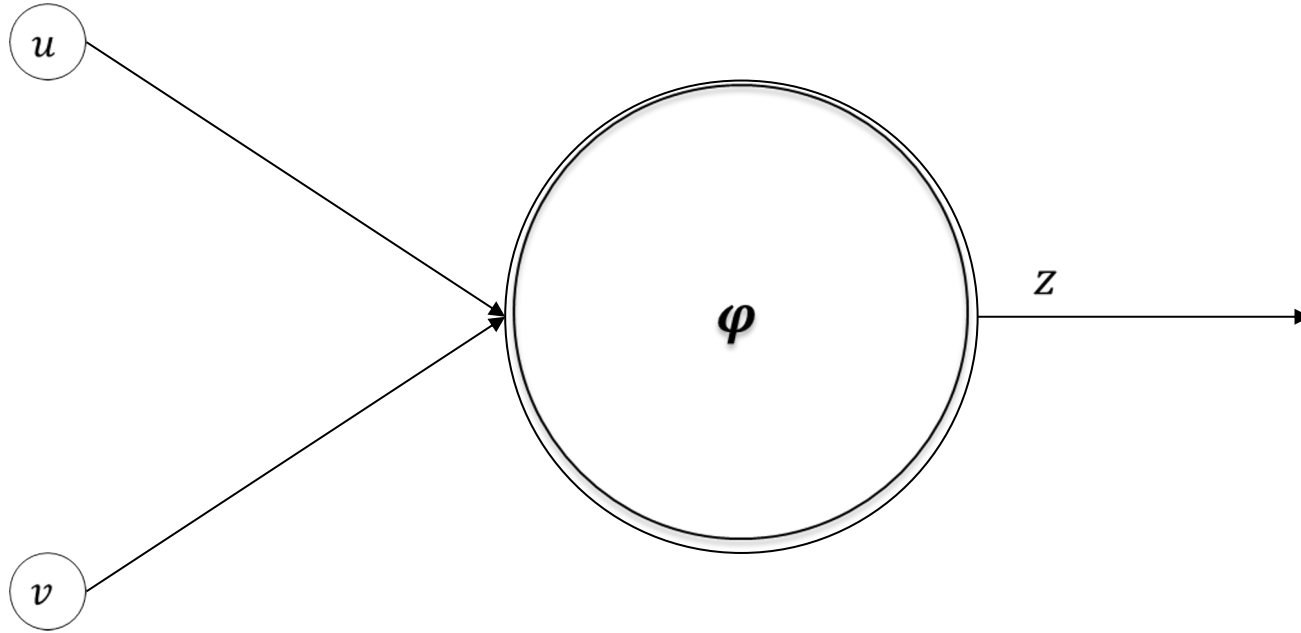
For more on cross-entropy and its relation to maximum likelihood estimation:

<https://www.quora.com/Whats-an-intuitive-way-to-think-of-cross-entropy>

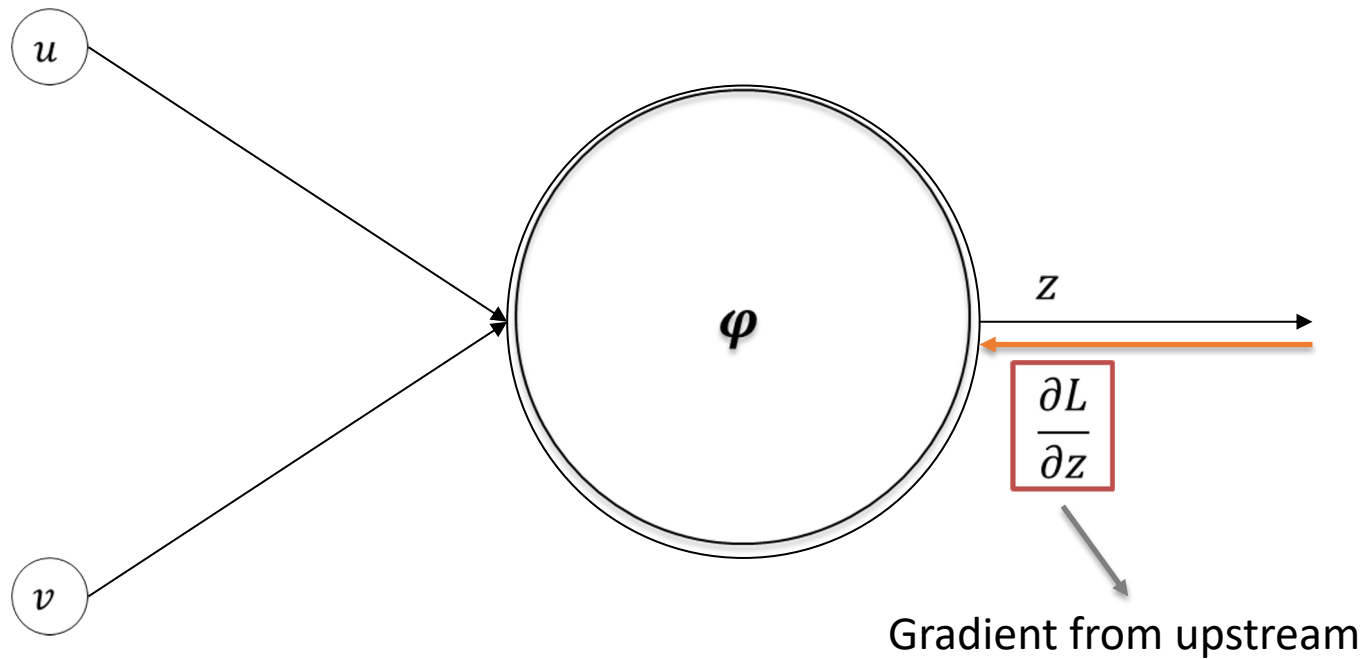
Backpropagation

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

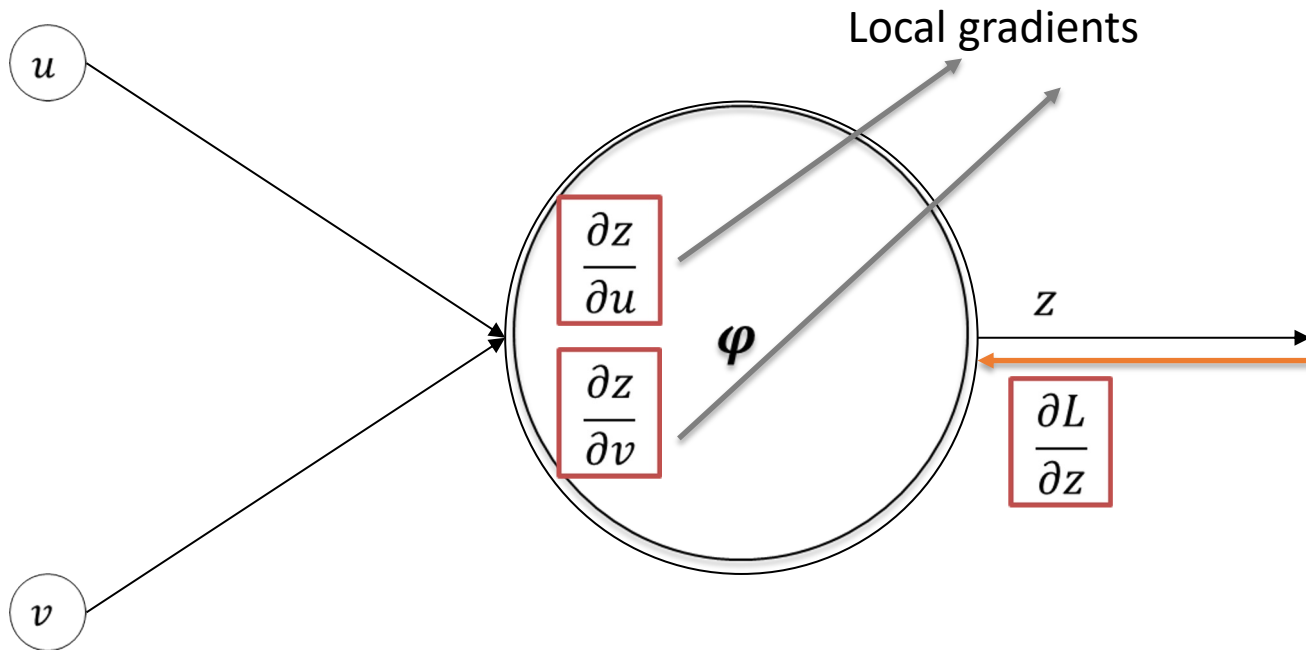
Gradients



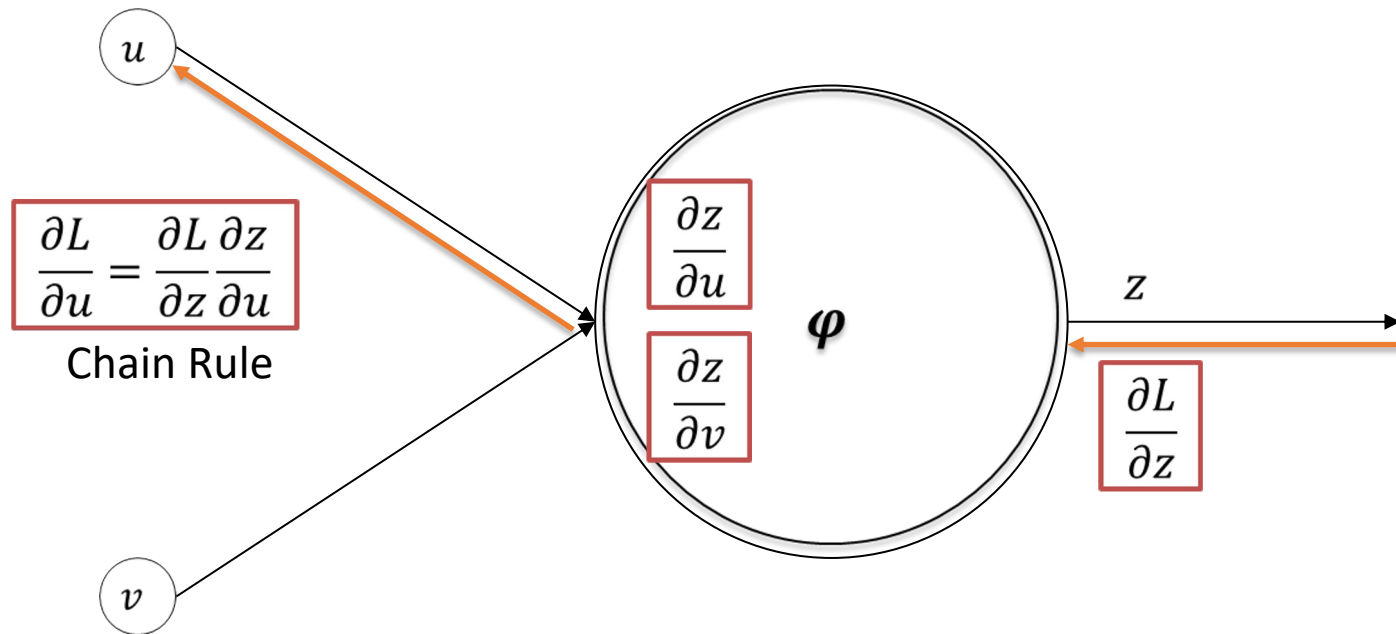
Gradients



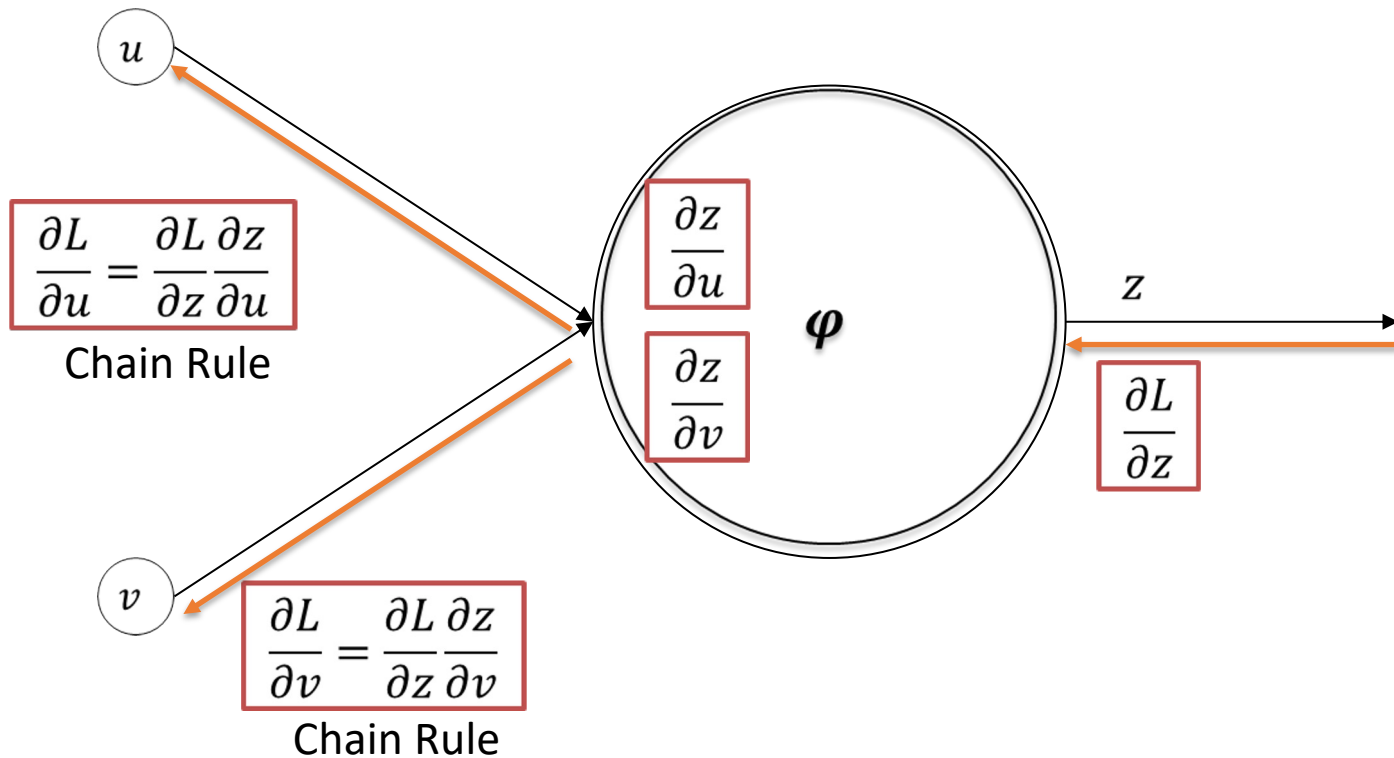
Gradients



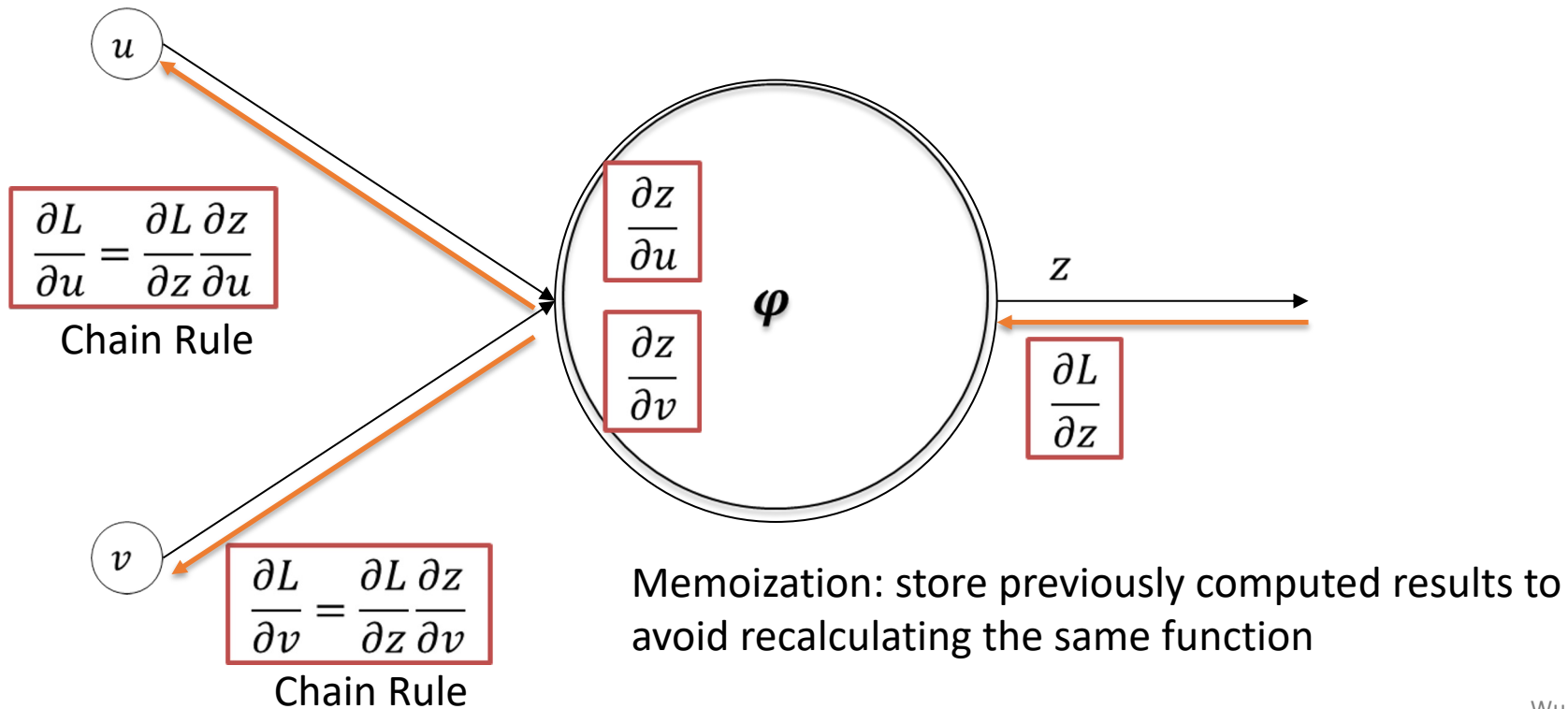
Gradients



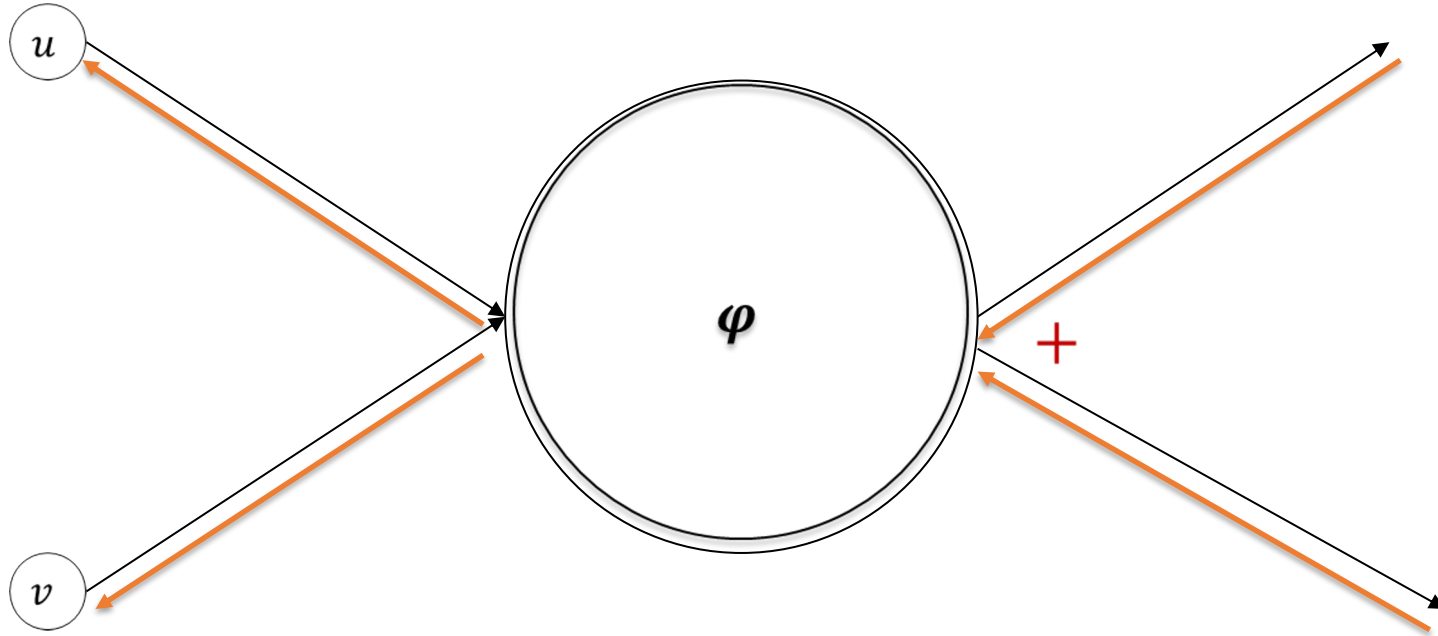
Gradients



Gradients

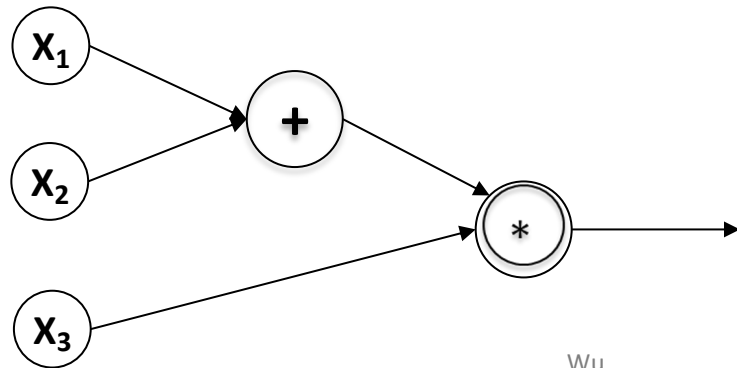


Gradients Accumulate



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

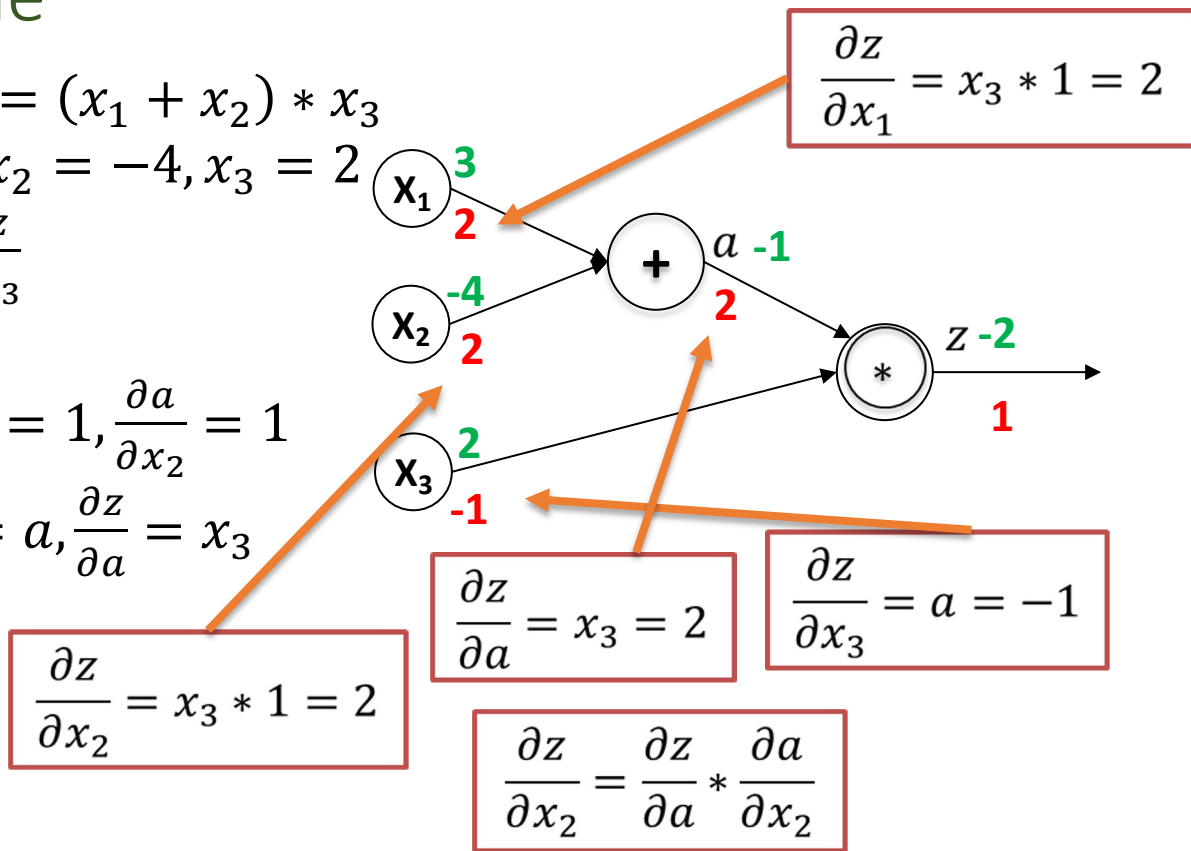


Simple Example

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

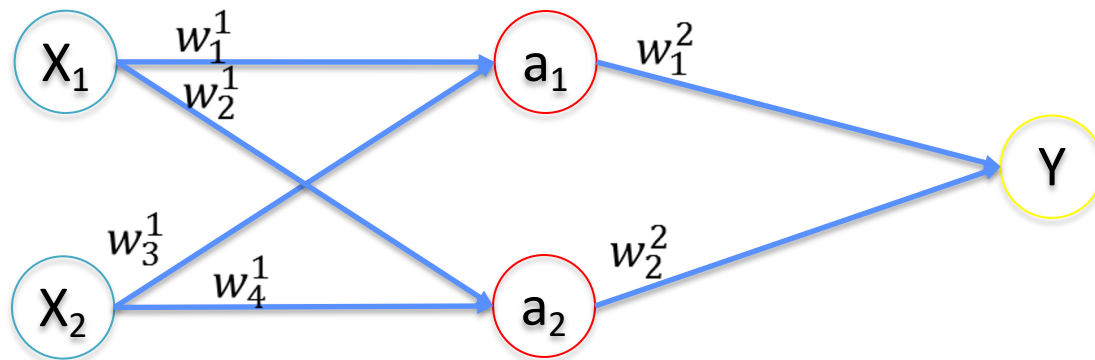
- $a = x_1 + x_2, \frac{\partial a}{\partial x_1} = 1, \frac{\partial a}{\partial x_2} = 1$

- $z = a * x_3, \frac{\partial z}{\partial x_3} = a, \frac{\partial z}{\partial a} = x_3$



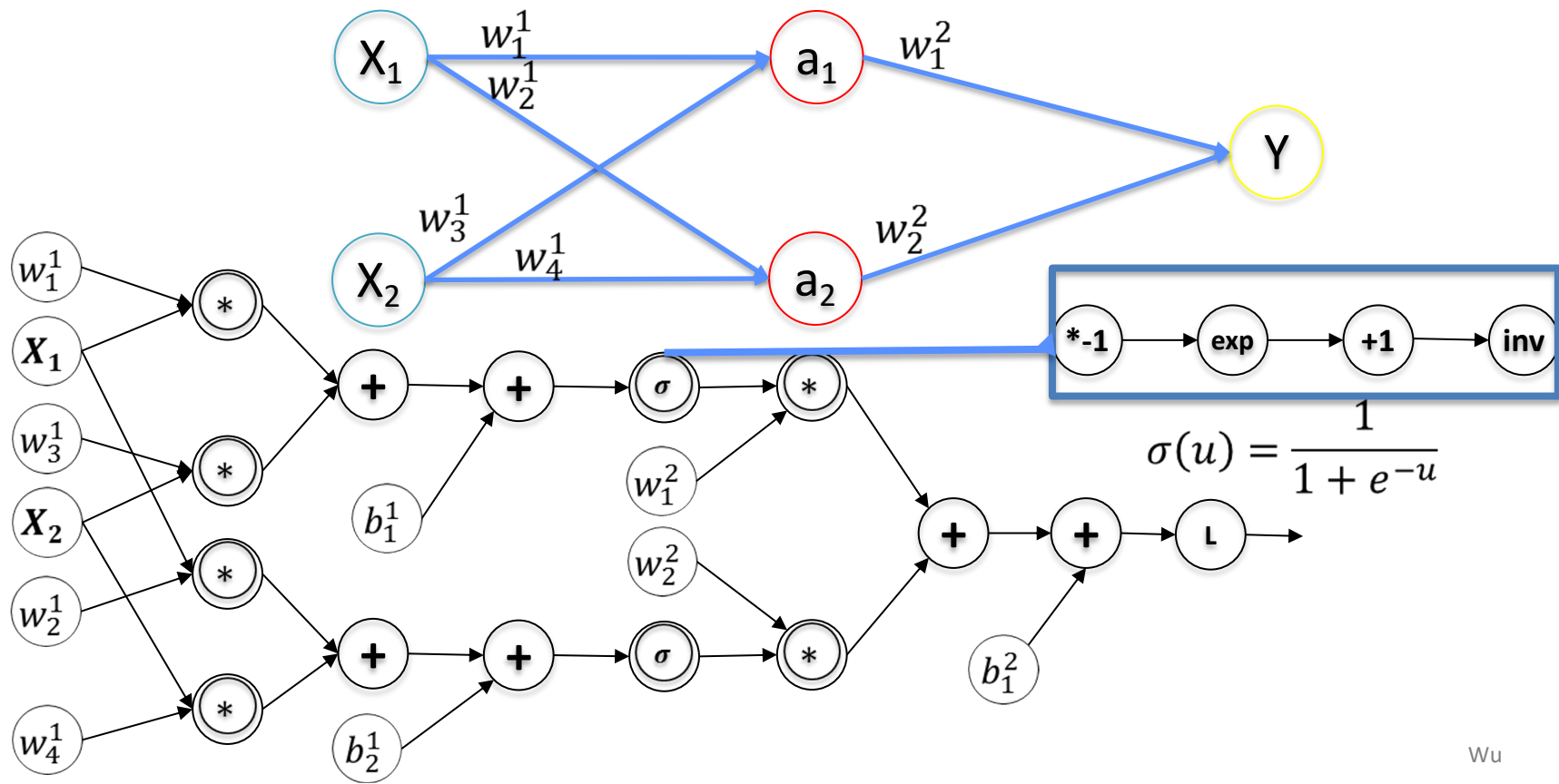
Chain Rule

Example: neural network value function w/ 1 data point

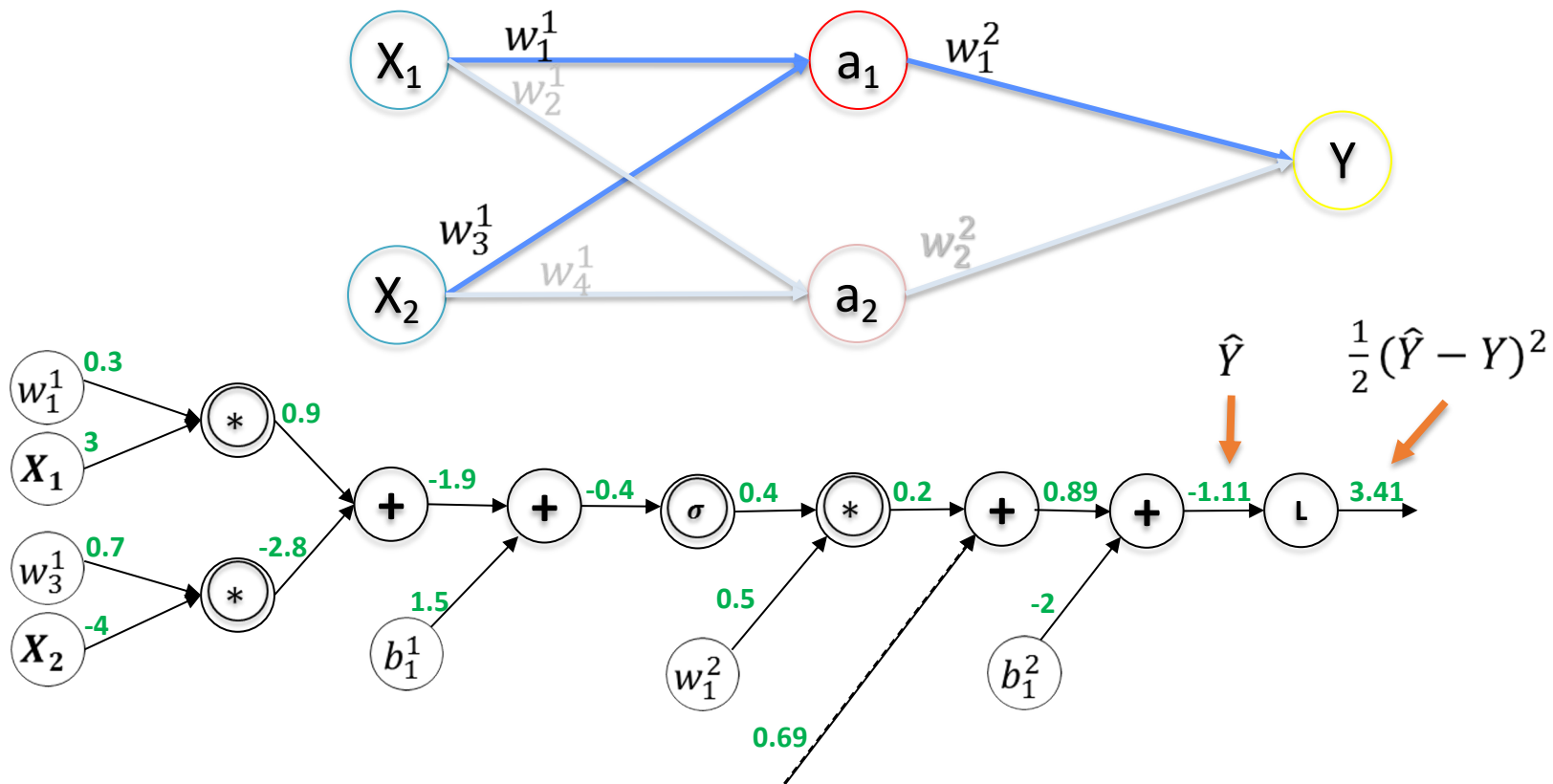


- Assume: $X = \begin{bmatrix} 3 \\ -4 \end{bmatrix}$, $Y = 1.5$, $w^1 = \begin{bmatrix} 0.3 \\ 0.9 \\ 0.7 \\ 0.6 \end{bmatrix}$, $w^2 = \begin{bmatrix} 0.5 \\ 0.8 \end{bmatrix}$, $b^1 = \begin{bmatrix} 1.5 \\ 1.5 \end{bmatrix}$, $b^2 = [-2]$, Activation = Sigmoid
- $z_1 = w_1^1 \cdot X_1 + w_3^1 \cdot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

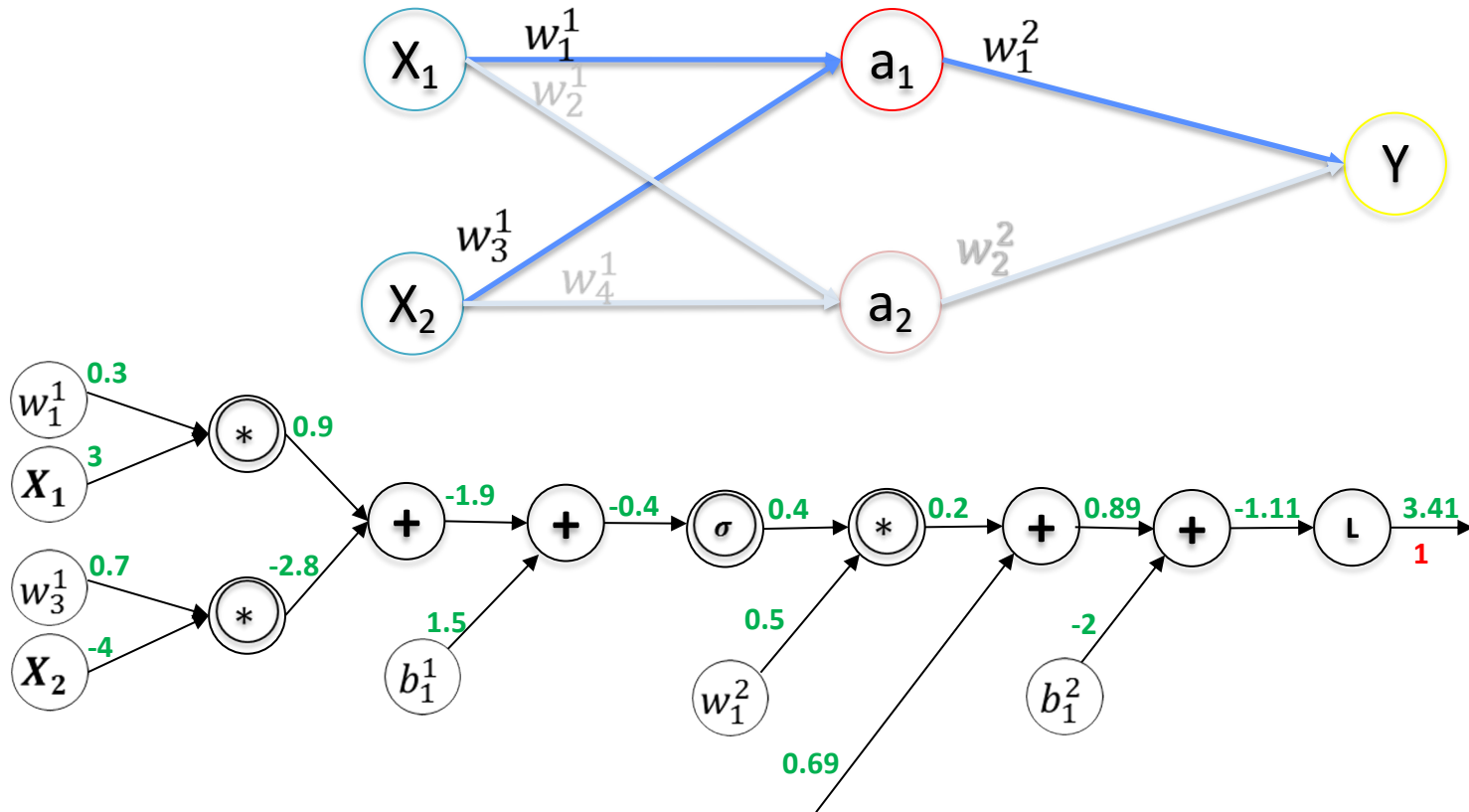
Example



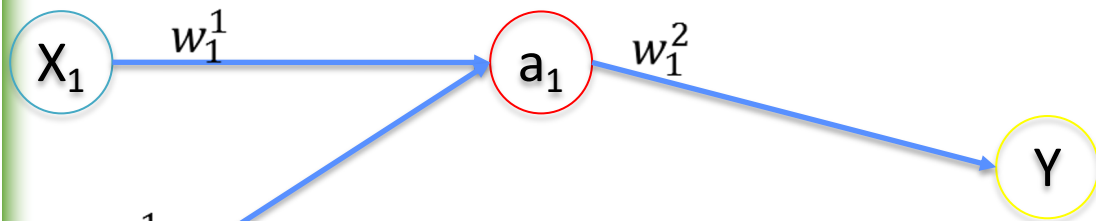
Example Forward Pass



Example Backward Pass



Example Backward Pass



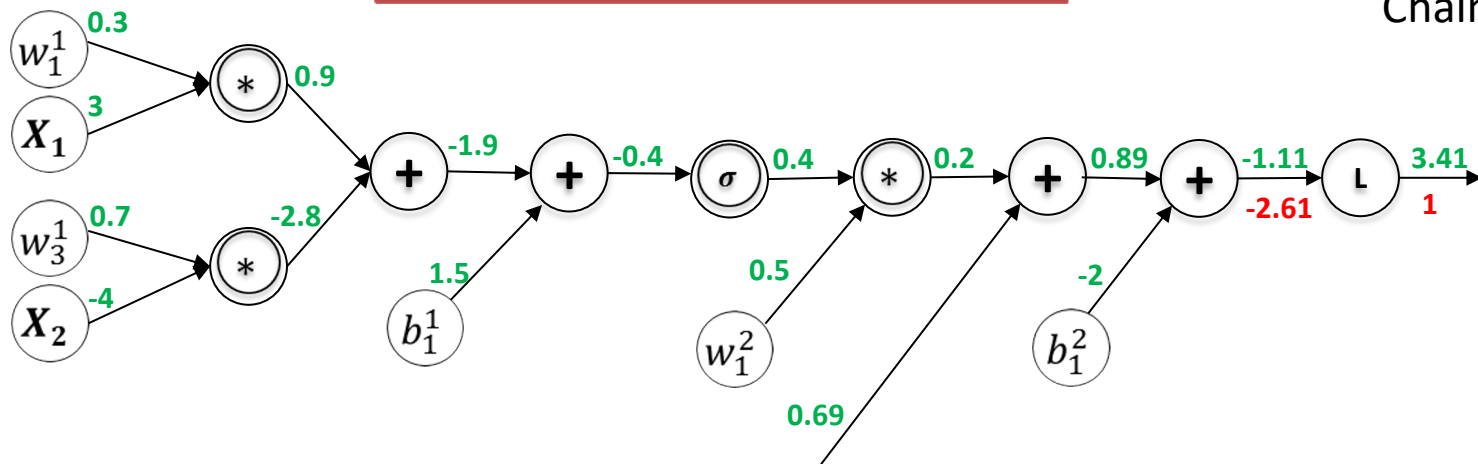
$$f(u) = \frac{1}{2}(u - c)^2 \quad \frac{df}{du} = u - c$$

$Y=1.5$

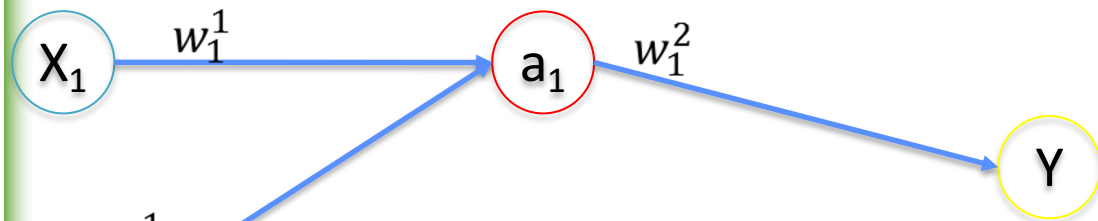


$$(-1.11 - 1.5) * 1 = -2.61$$

Chain Rule



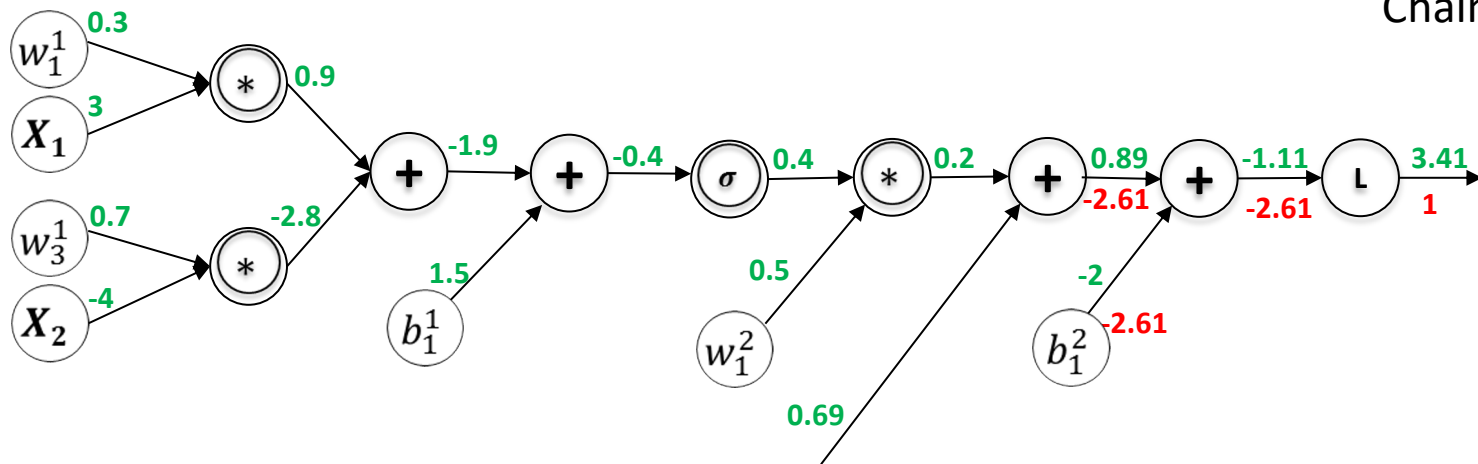
Example Backward Pass



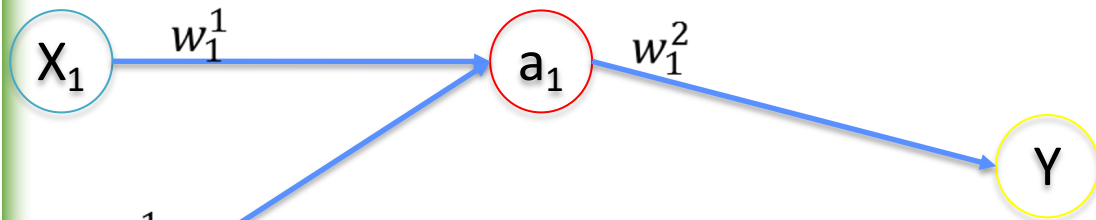
$$f(u) = c + u \quad \frac{df}{du} = 1$$

$$1 * -2.61 = -2.61$$

Chain Rule



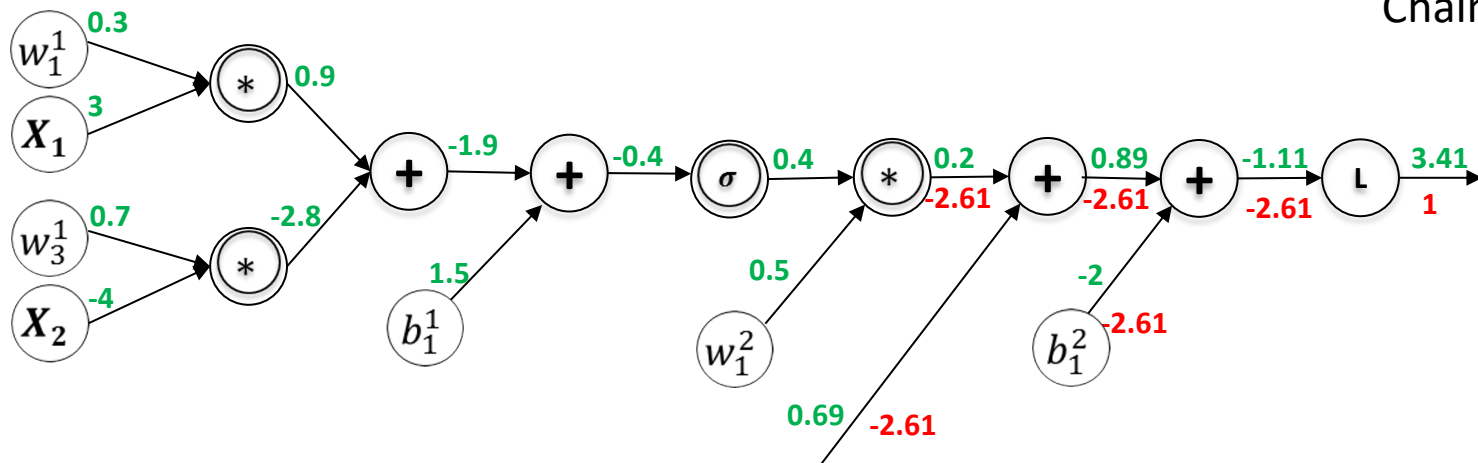
Example Backward Pass



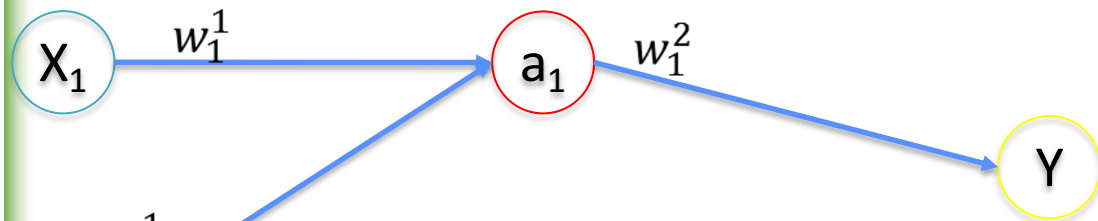
$$f(u) = c + u \quad \frac{df}{du} = 1$$

$$1 * -2.61 = -2.61$$

Chain Rule



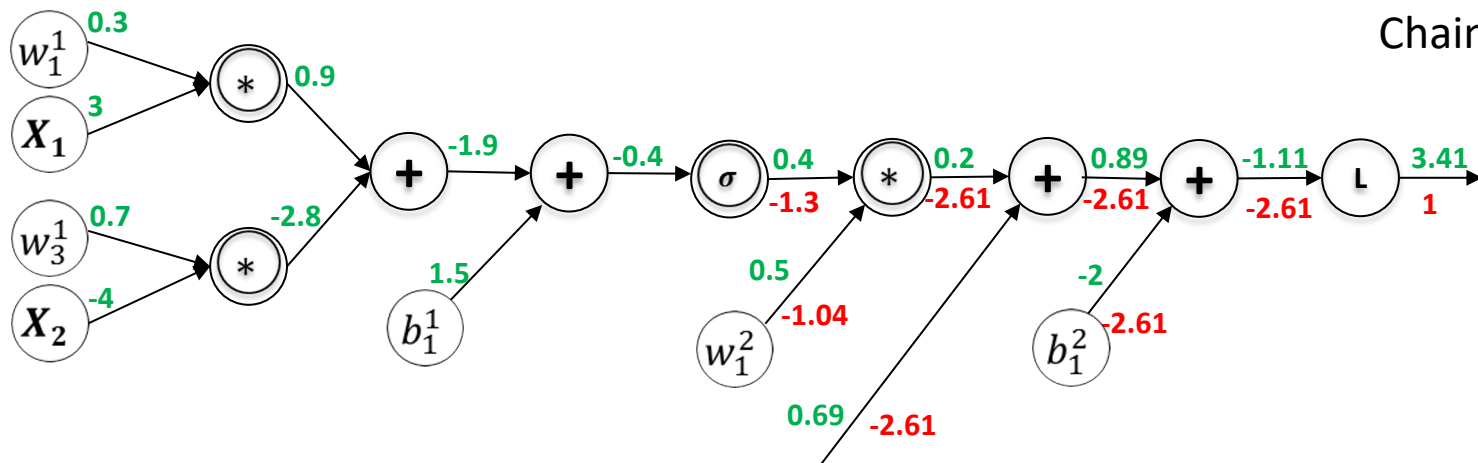
Example Backward Pass



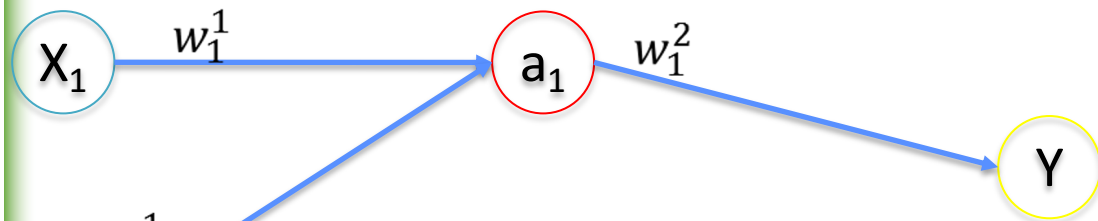
$$f(u) = c * u \quad \frac{df}{du} = c$$

$$\begin{aligned} 0.5 * -2.61 &= -1.30 \\ 0.4 * -2.61 &= -1.04 \end{aligned}$$

Chain Rule



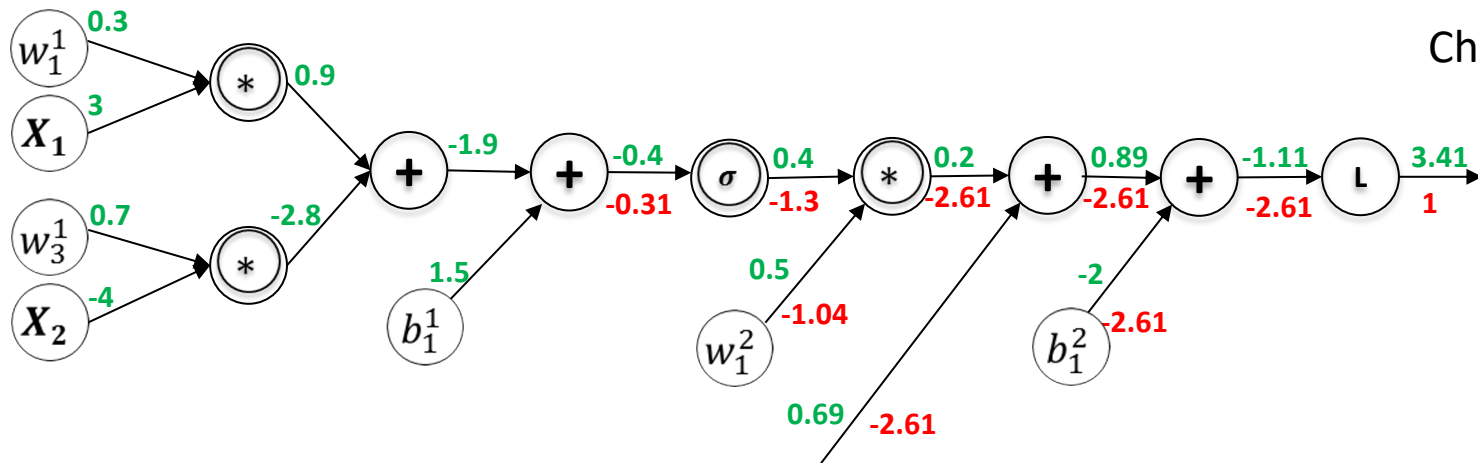
Example Backward Pass



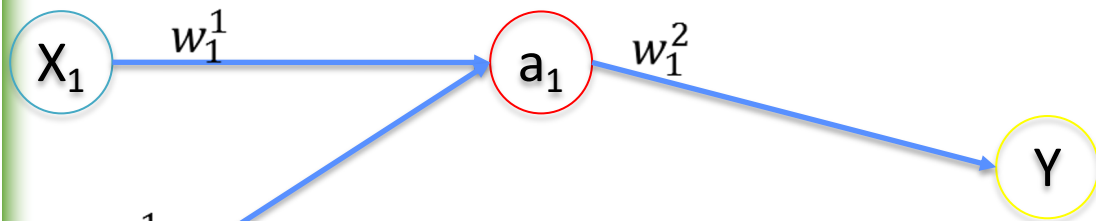
$$f(u) = \frac{1}{1 + e^{-u}} \quad \frac{df}{du} = f(u)(1 - f(u))$$

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

Chain Rule



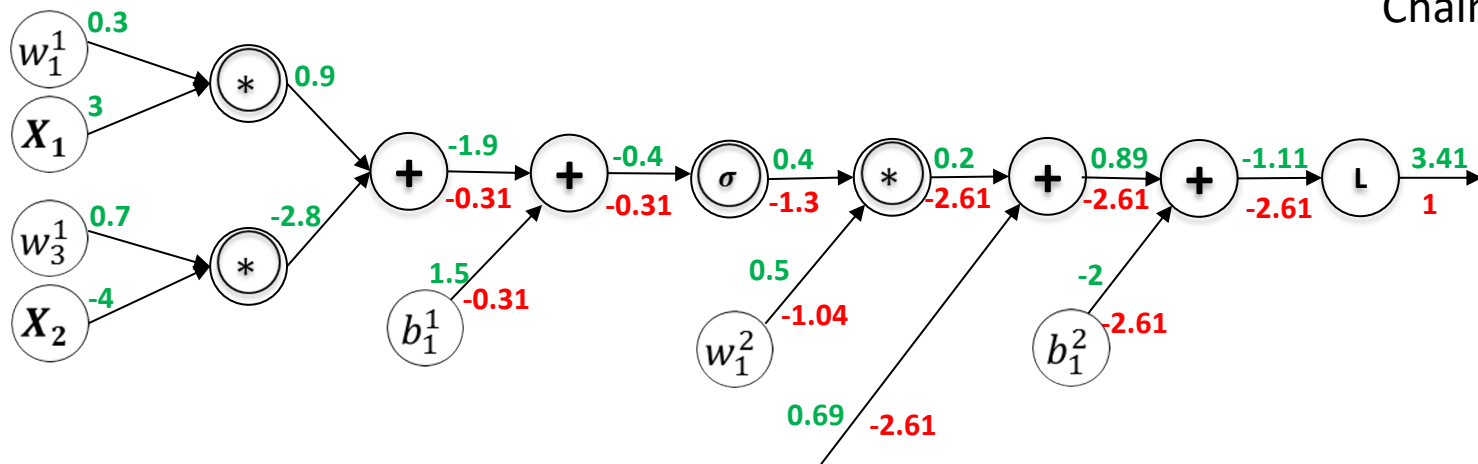
Example Backward Pass



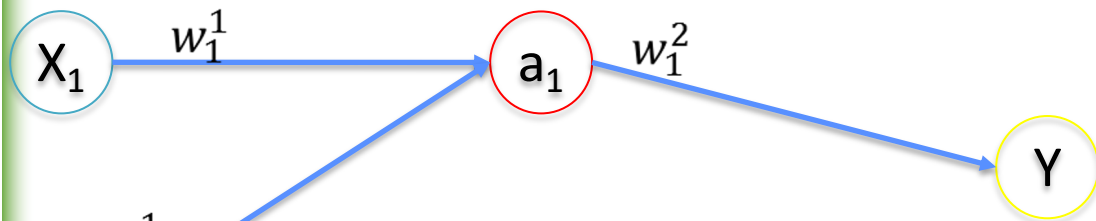
$$f(u) = c + u \quad \frac{df}{du} = 1$$

$$1 * -0.31 = -0.31$$

Chain Rule



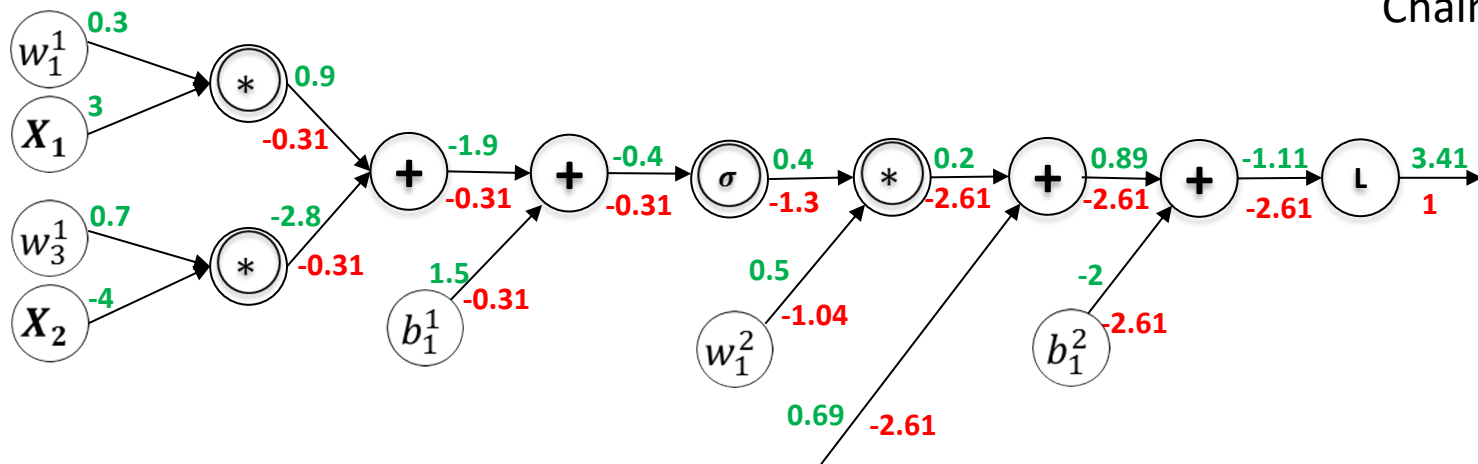
Example Backward Pass



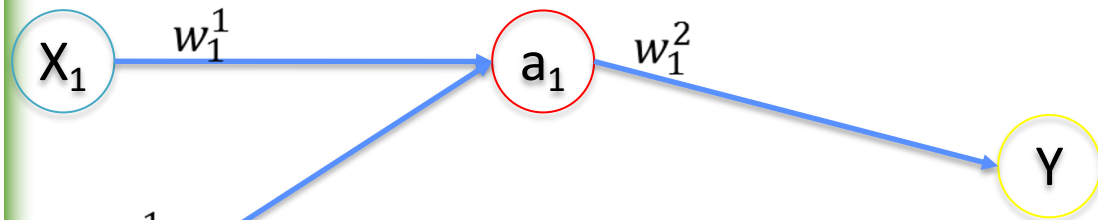
$$f(u) = c + u \quad \frac{df}{du} = 1$$

$$1 * -0.31 = -0.31$$

Chain Rule



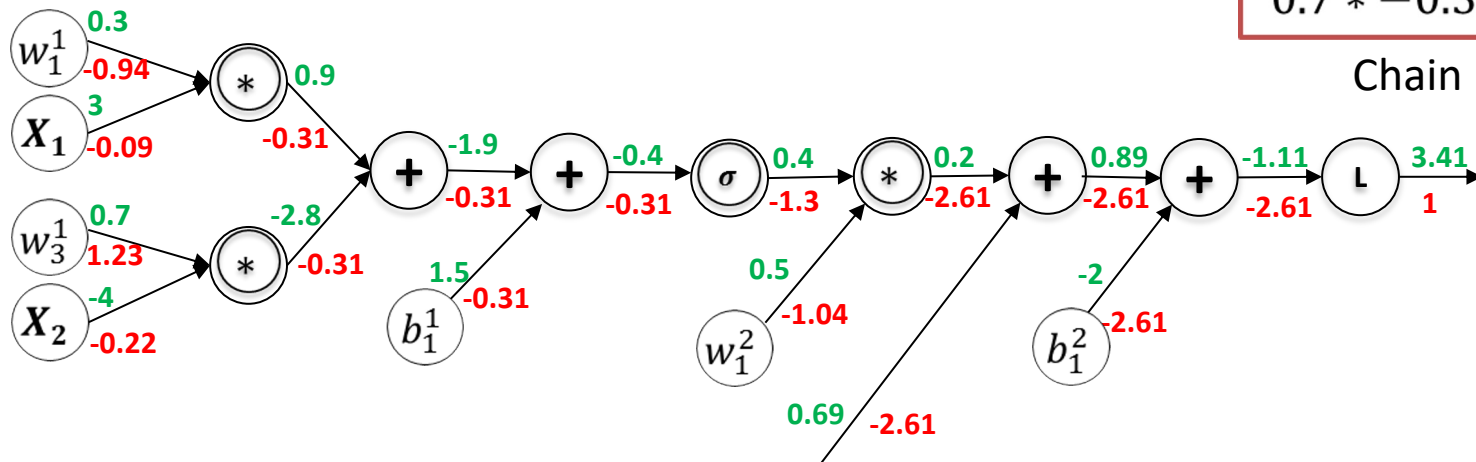
Example Backward Pass



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

$$\begin{aligned} 3 * -0.31 &= -0.94 \\ 0.3 * -0.31 &= -0.09 \\ -4 * -0.31 &= 1.23 \\ 0.7 * -0.31 &= -0.22 \end{aligned}$$

Chain Rule



Example: Update

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

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

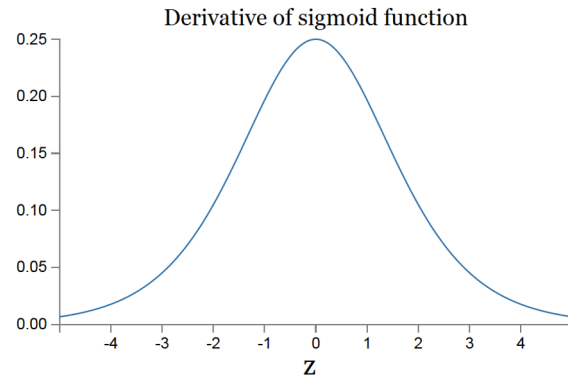
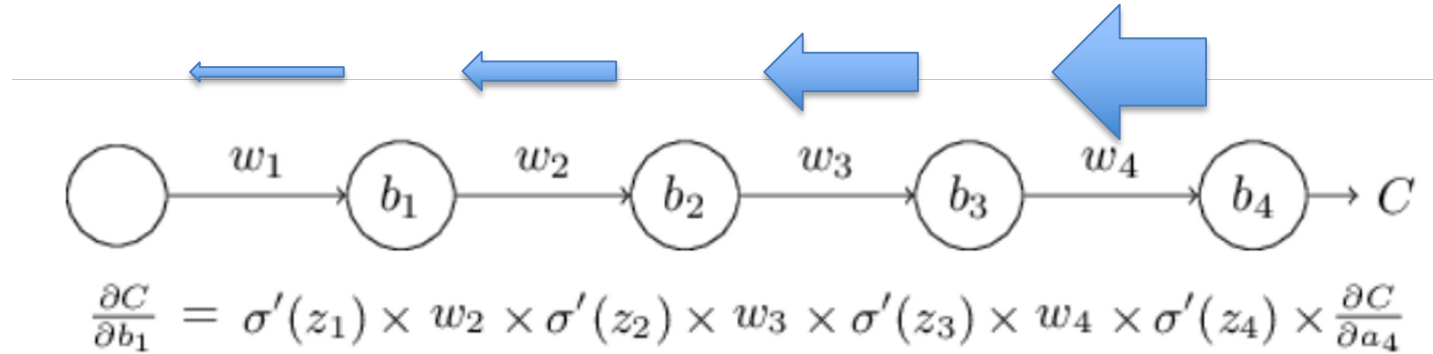
$$b^{r+1} = b^r - \gamma \frac{\partial L}{\partial b}$$

$$\gamma = 0.9$$

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

Vanishing Gradients Problem

■ Deep NN



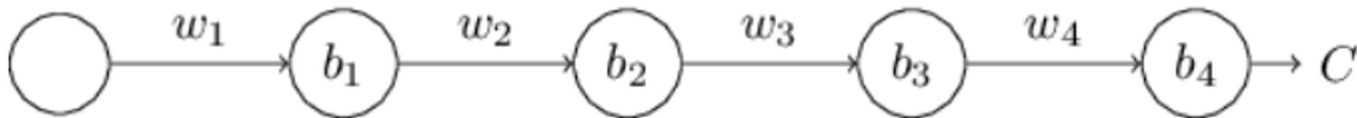
$$\begin{aligned} w &< 1 \\ \sigma' &\leq 0.25 \end{aligned} \longrightarrow w \times \sigma' < 0.25$$

In practice:

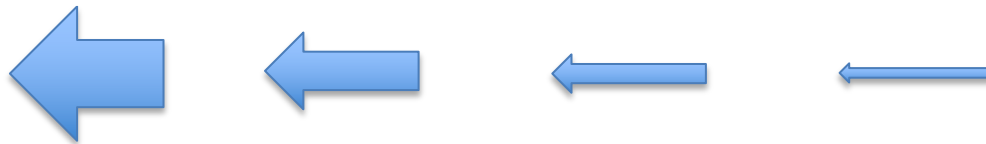
- Use ReLU
- Don't use sigmoid

Exploding Gradients Problem

Deep NN



$$w \times \sigma' > 1 \quad \frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$



Be careful with the weight initialization

Hyperparameters

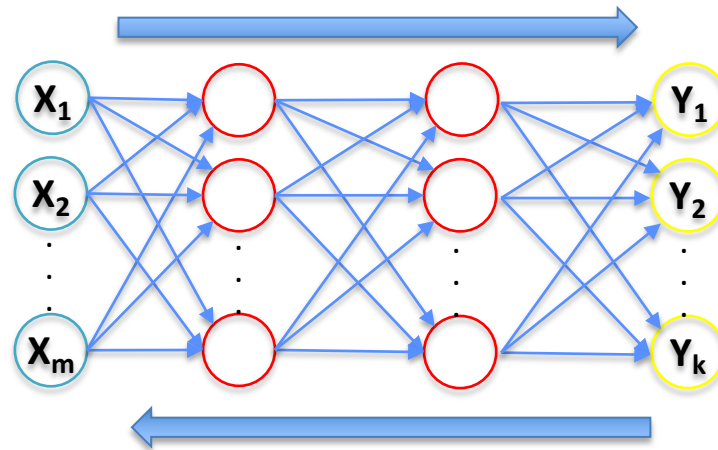
- Hyperparameters:
 - Learning rate
 - Network architecture
 - Regularization

- Use cross-validation for optimizing the hyperparameters

Recap (Backpropagation)

0- Initialize the weights and biases

1- Forward pass



2- Compute the error

4- Update the weights

3- Backward pass, compute the gradients

Outline

1. Motivation: function approximation for large state spaces
2. Gradient descent (GD)
3. Deep neural networks (DNN)
4. **Training a DNN: SGD + Backpropagation**

Training deep neural networks

Main idea: iteratively minimize an (error) function.

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

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

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

via Backprop



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

References

References:

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

With slides adapted from:

1. Eugene Vinitzky (Berkeley EE290O)