CSE 417T
Introduction to Machine Learning

Lecture 20

Instructor: Chien-Ju (CJ) Ho

Logistics

- Homework 5 is due December 2 (Friday)

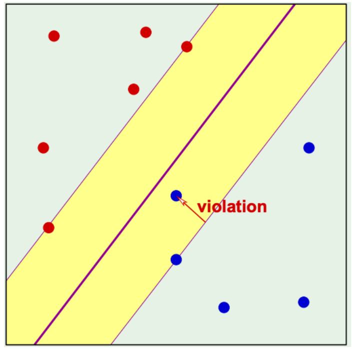
- Exam 2 will be on December 8 (Thursday)
  - Will focus on the topics in the second half of the semester
  - Format / logistics will be similar to Exam 1
  - More details to come

Recap

Support Vector Machines

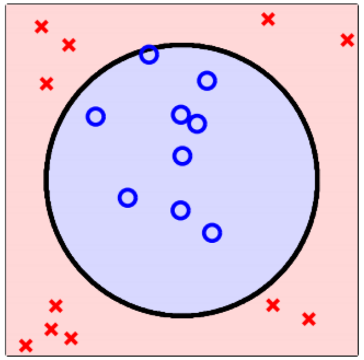
- Soft-margin SVM (approximates hard-margin SVM with $C \to \infty$)

$$\begin{array}{ll} \text{minimize}_{\vec{w},b,\vec{\xi}} & \frac{1}{2}\vec{w}^T\vec{w} + C\sum_{n=1}^{N}\xi_n \\ \text{subject to} & y_n(\vec{w}^T\vec{x}_n + b) \geq 1 - \xi_n, \forall n \\ & \xi_n \geq 0, \forall n \end{array}$$



- Kernel version of the soft-margin SVM (with Kernel K_Φ)

$$\begin{array}{ll} \text{maximize}_{\vec{\alpha}} & \sum_{n=1}^{N}\alpha_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N}\alpha_n\alpha_m y_n y_m K_\Phi(\vec{x}_n,\vec{x}_m) \\ \text{subject to} & \sum_{n=1}^{N}\alpha_n y_n = 0 \\ & 0 \leq \alpha_n \leq C, \forall n \end{array}$$



  - Solve for $\vec{\alpha}^*$ in the kernel SVM using QP

$$g(\vec{x}) = sign(\vec{w}^{*T}\Phi(\vec{x}) + b^*)$$
$$= \boldsymbol{sign(\textstyle\sum_{\alpha_n^*>0}\alpha_n^* y_n K_\Phi(\vec{x}_n,\vec{x}) + b^*)},$$

where $b^* = y_m - \sum_{\alpha_n^*>0}\alpha_n^* y_n K_\Phi(\vec{x}_n,\vec{x}_m)$ for some $\alpha_m^* > 0$

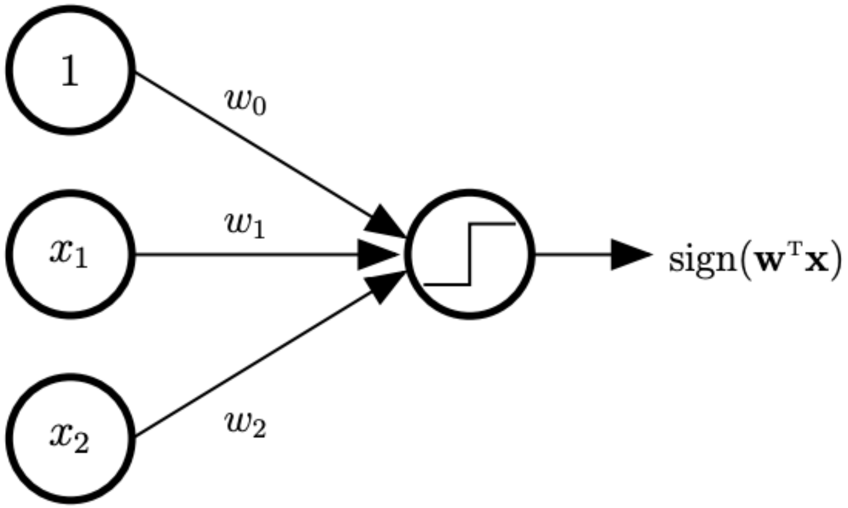
Neural Networks

Perceptron

- A hypothesis in Perceptron

$$h(\vec{x}) = sign(\vec{w}^T \vec{x})$$

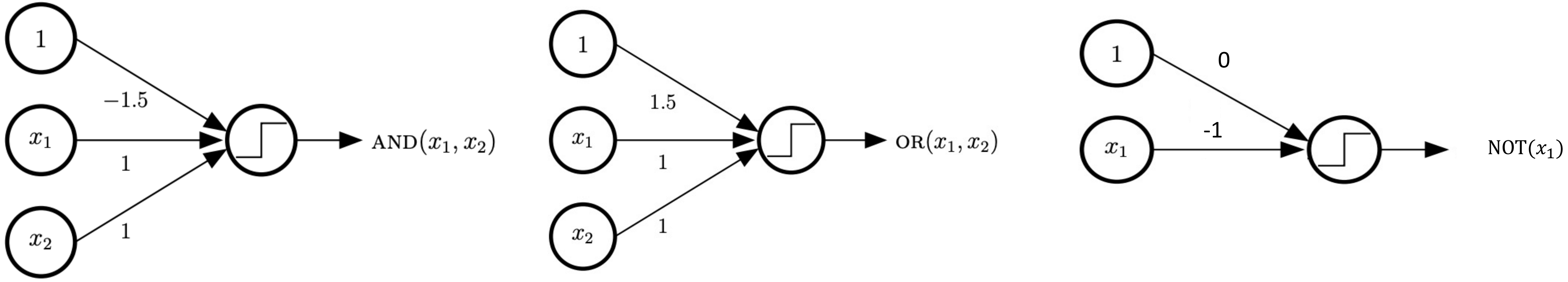
- Graphical representation of Perceptron



- Notations
  - $\vec{x} = (x_0, x_1, \dots, x_d)$
  - $\vec{w} = (w_0, w_1, \dots, w_d)$
  - Linear separator
    $h(\vec{x}) = sign(\vec{w}^T \vec{x})$

Inspired by neurons:
The output signal is triggered when the weighted combination of the inputs is larger than some threshold

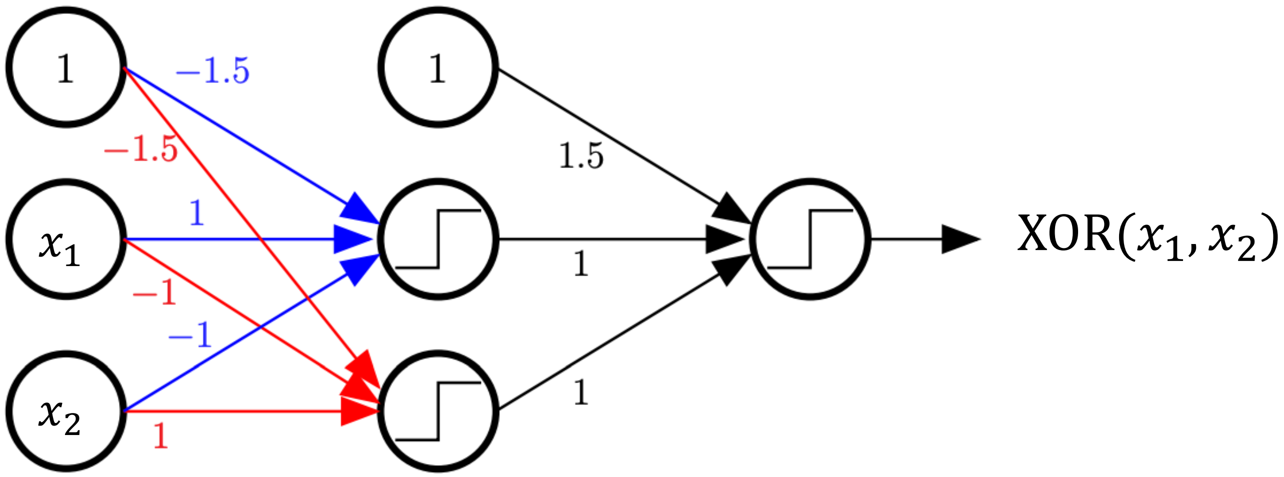
Implementing Logic Gates with Perceptron



Impossible to implement XOR using a single perceptron

Multi-Layer Perceptron

- $\text{XOR}(x_1, x_2) \rightarrow x_1\bar{x}_2 + \bar{x}_1 x_2$



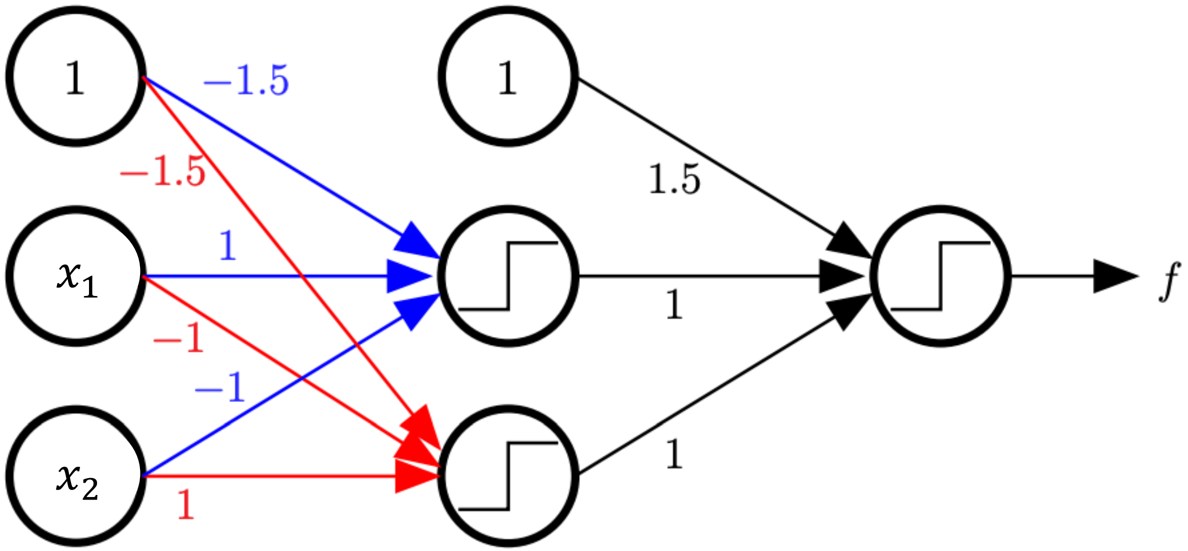
- Note: you are asked to create a neural network with one hidden layer that implements $\text{XOR}(\text{AND}(x_1, x_2), x_3)$ in HW5
  - Hint: Try to operate the Boolean algebra first
  - Using sign as the activation function would make sense

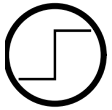
Universal Approximation Theorem

- A feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of $\mathbb{R}^n$, under mild assumptions on the activation function.
- Single-hidden-layer MLP can approximate ANY continuous target function!
- What about overfitting?
  - We'll discuss regularization methods later

Learn MLP From Data?

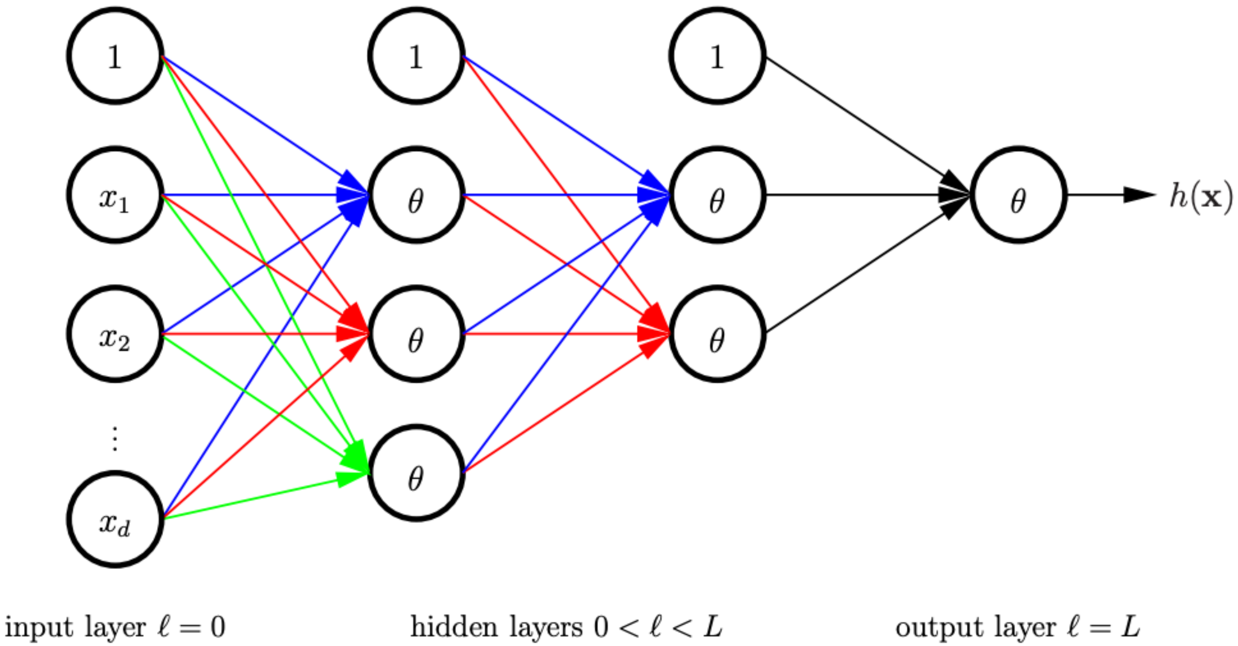
- Given D and the network structure, how to learn the "weights" (i.e., the weight vectors of every Perceptron)?



- Computationally challenging due to the "sign" function 

Neural Networks

- A softened version of multi-layer Perceptron (MLP)



θ: activation function

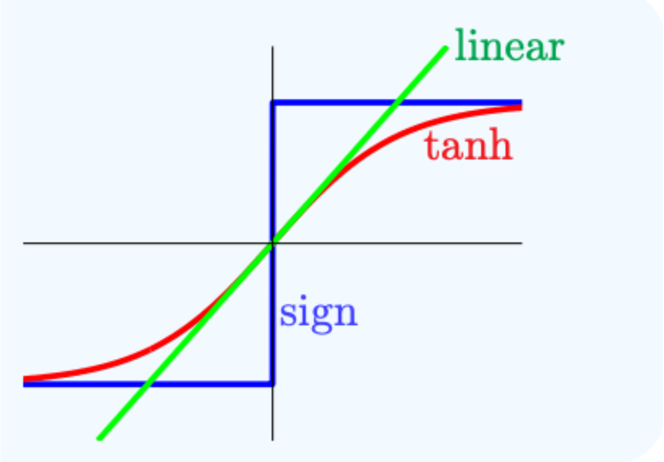
(Specify the "activation" of the neuron)

(The activation function in the output layer is often separately considered)

Activation Function

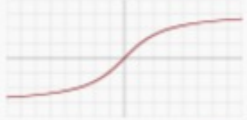
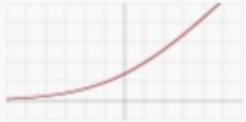
- Activation functions in Neural Networks
  - sign function: hard to optimize
  - linear function: the entire neural network is linear
  - tanh: a softened version of sign
    - $\tanh(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}}$

- Examine $\tanh(s)$
  - $\tanh(s) = \begin{cases} 1 \text{ when } s \to \infty \\ 0 \text{ when } s = 0 \\ -1 \text{ when s} \to \infty \end{cases}$
  - For $\theta(s) = \tanh(s), \theta'(s) = 1 - \theta(s)^2$



Activation Function

- There are other activation functions with different benefits. However, it doesn't impact our discussions, and we'll focus on tanh() as the activation function
- A few more examples

ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU)[2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU)[3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

Today’s Lecture

The notes are not intended to be comprehensive. They should be accompanied by lectures and/or textbook.

Let me know if you spot errors.

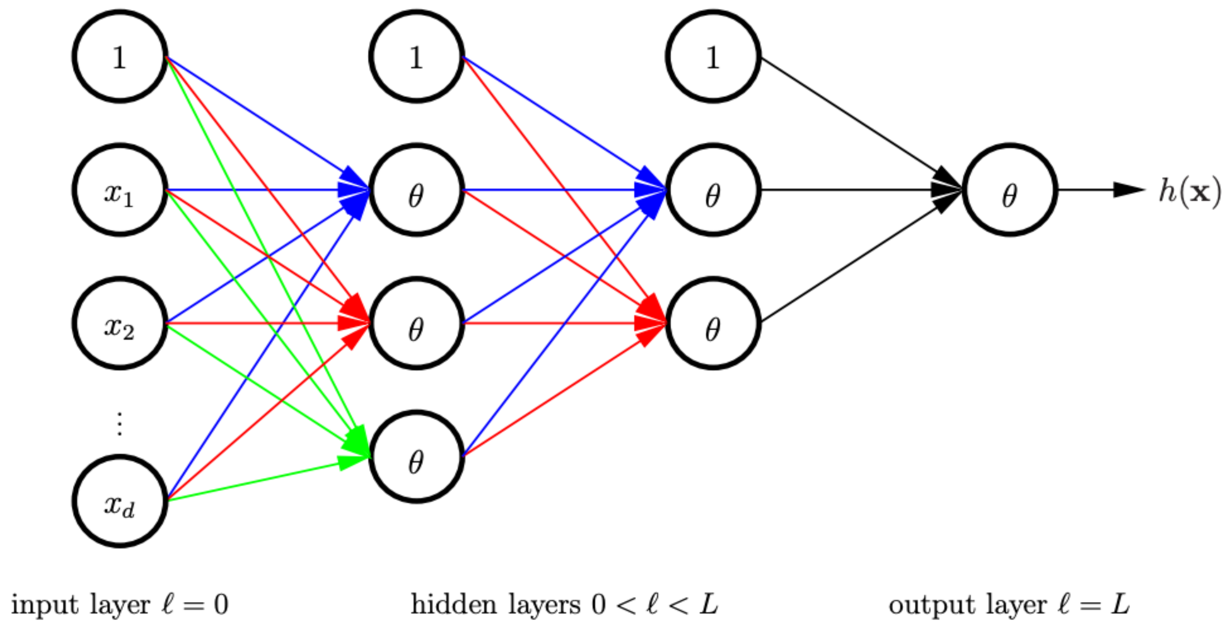
Goal of Today

- Formally characterize Neural Networks (introduce notations)
- Given a Neural Network hypothesis h, how do we make prediction $h(\vec{x})$
- Given D, how do we learn a Neural Network hypothesis

Notations of Neural Networks (NN)

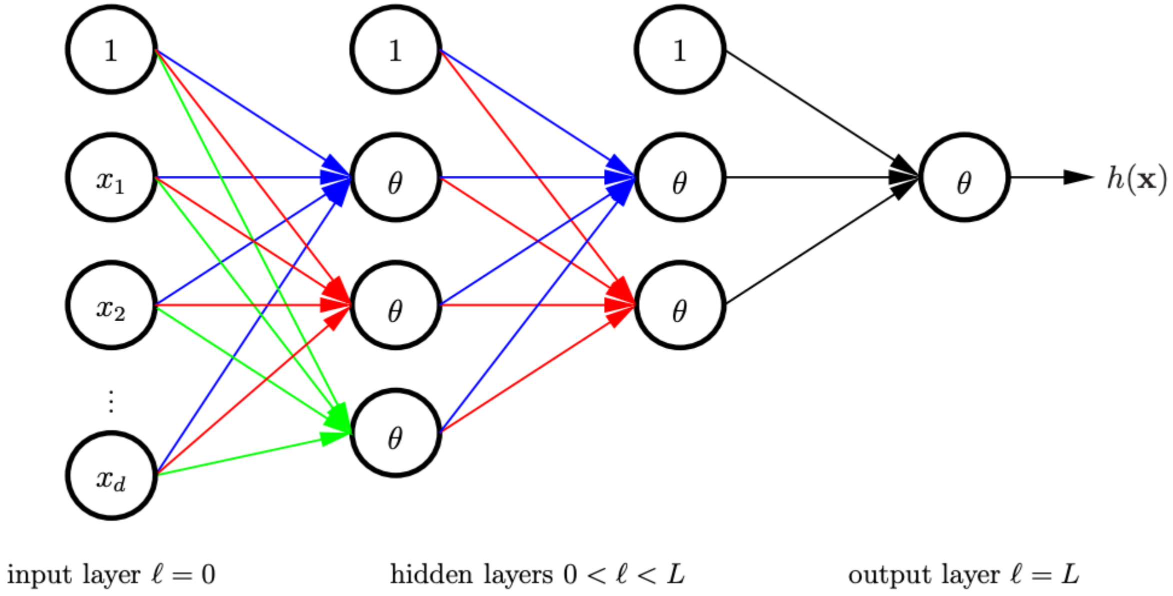
Notations of Neural Networks (NN)

- Layers $\ell = 0$ to L
  - Layer 0: input layer
  - Layer 1 to $L-1$: hidden layers
  - Layer L: output layer
- $d^{(\ell)}$: dimension of layer ℓ
  - # nodes (excluding 1s) in the layer
- $\vec{x}^{(\ell)}$: the nodes in layer ℓ
  - $\vec{x}^{(0)}$ is the input feature $\vec{x}$
  - $x_i^{(\ell)}$ is the i-th node in layer ℓ



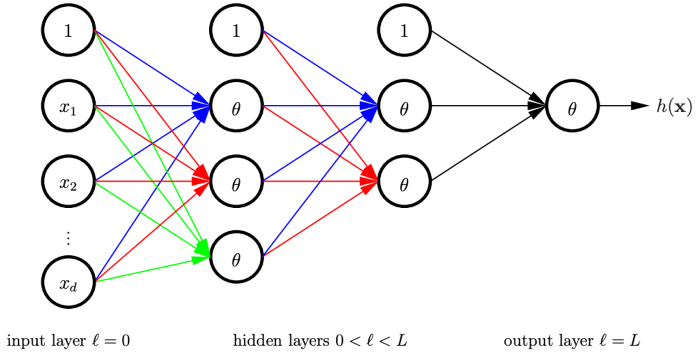
Notations of Neural Networks (NN)

- A hypothesis in linear model is specified by the weights $\{w_i\}$
- Similarly, a hypothesis in NN is characterized by the weights $\{w_{i,j}^{(\ell)}\}$
  - $1 \leq \ell \leq L$ layers
  - $0 \leq i \leq d^{(\ell-1)}$ inputs
  - $1 \leq j \leq d^{(\ell)}$ outputs



Notations of Neural Networks (NN)

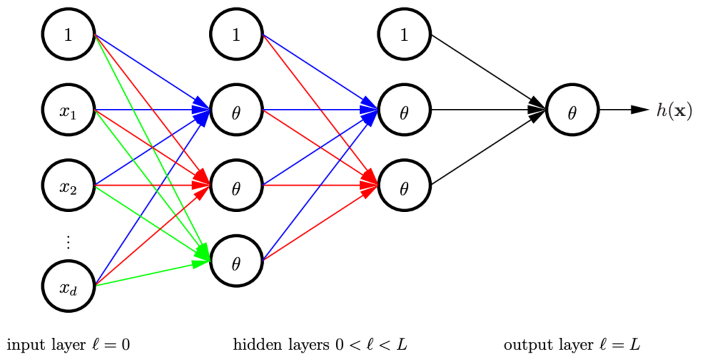
- Notations so far:
  - $d^{(\ell)}$: dimension of layer ℓ
  - $\vec{x}^{(\ell)}$: the nodes in layer ℓ
  - $w_{i,j}^{(\ell)}$: weights; characterize hypothesis in NN



- Lastly, linear signal $s_j^{(\ell)} = \sum_{i=0}^{d^{(\ell-1)}} w_{i,j}^{(\ell)} x_i^{(\ell-1)}$
  - By definition: $x_j^{(\ell)} = \theta\left(s_j^{\ell}\right)$

$$\mathbf{s}^{(\ell)} \xrightarrow{\theta} \mathbf{x}^{(\ell)}$$

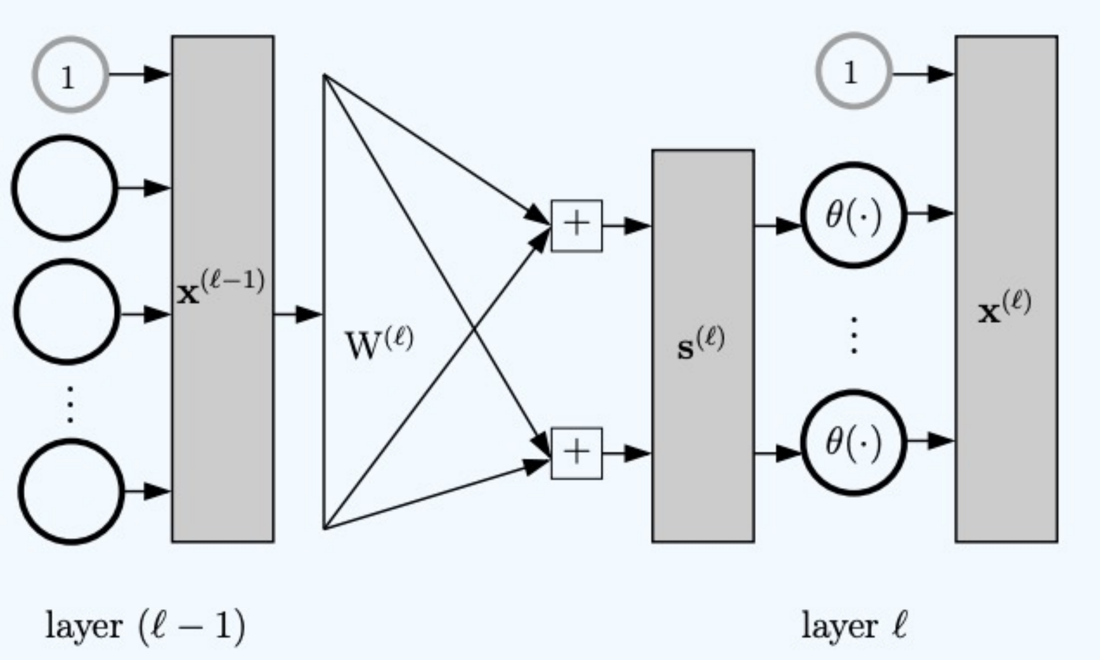
Notations of Neural Networks (NN)



- Notations so far:
  - $d^{(\ell)}$: dimension of layer ℓ
  - $\vec{x}^{(\ell)}$: the nodes in layer ℓ
  - $w_{i,j}^{(\ell)}$: weights; characterize hypothesis in NN

- Lastly, linear signal $s_j^{(\ell)} = \sum_{i=0}^{d^{(\ell-1)}} w_{i,j}^{(\ell)} x_i^{(\ell-1)}$
  - By definition: $x_j^{(\ell)} = \theta\left(s_j^{\ell}\right)$

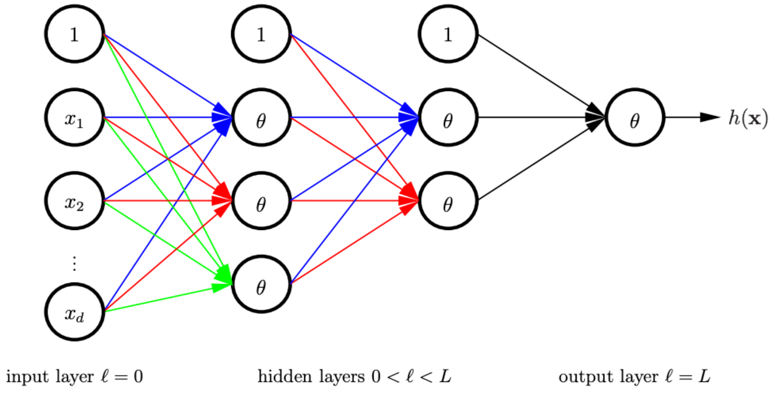
$$\mathbf{s}^{(\ell)} \xrightarrow{\theta} \mathbf{x}^{(\ell)}$$



Short Break and Q&A

Practice:

For a neural network with $L = 2$, $d^{(0)} = 3$, $d^{(1)} = 2$, $d^{(2)} = 1$, what is the total # weights?



Notations so far:

- $d^{(\ell)}$: dimension of layer ℓ
- $\vec{x}^{(\ell)}$: the nodes in layer ℓ
- $w_{i,j}^{(\ell)}$: weights; characterize hypothesis in NN
- $s_j^{(\ell)} = \sum_{i=0}^{d^{(\ell-1)}} w_{i,j}^{(\ell)} x_i^{(\ell-1)}$: linear signal

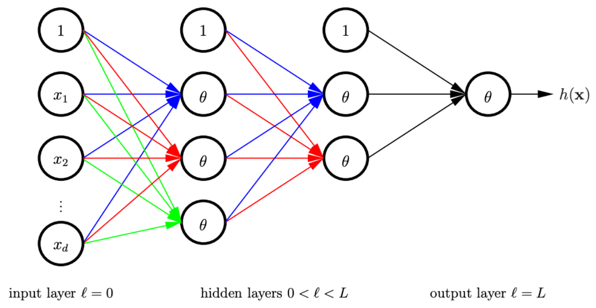
Forward Propagation

Given a NN hypothesis and a point $\vec{x}$, how do we make predictions

Backpropagation

Learn a Neural Network hypothesis from data

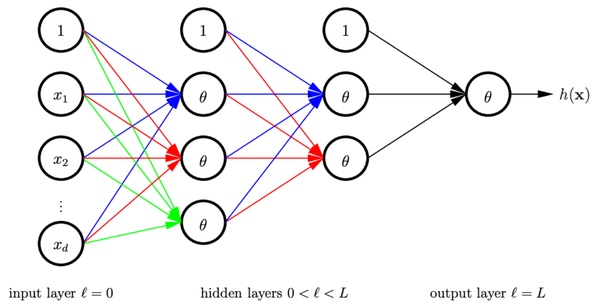
Forward Propagation



- A Neural network hypothesis h is characterized by $\left\{w_{i,j}^{(\ell)}\right\}$
- How to evaluate $h(\vec{x})$?

$$\mathbf{x} = \mathbf{x}^{(0)} \xrightarrow{\mathrm{W}^{(1)}} \mathbf{s}^{(1)} \xrightarrow{\theta} \mathbf{x}^{(1)} \xrightarrow{\mathrm{W}^{(2)}} \mathbf{s}^{(2)} \xrightarrow{\theta} \mathbf{x}^{(2)} \cdots \xrightarrow{\mathrm{W}^{(L)}} \mathbf{s}^{(L)} \xrightarrow{\theta} \mathbf{x}^{(L)} = h(\mathbf{x}).$$

Forward Propagation



- A Neural network hypothesis h is characterized by $\left\{w_{i,j}^{(\ell)}\right\}$
- How to evaluate $h(\vec{x})$?

$$\mathbf{x} = \mathbf{x}^{(0)} \xrightarrow{\mathrm{W}^{(1)}} \mathbf{s}^{(1)} \xrightarrow{\theta} \mathbf{x}^{(1)} \xrightarrow{\mathrm{W}^{(2)}} \mathbf{s}^{(2)} \xrightarrow{\theta} \mathbf{x}^{(2)} \cdots \xrightarrow{\mathrm{W}^{(L)}} \mathbf{s}^{(L)} \xrightarrow{\theta} \mathbf{x}^{(L)} = h(\mathbf{x}).$$

Forward propagation to compute $h(\mathbf{x})$:

$\mathbf{x}^{(0)} \leftarrow \mathbf{x}$ **[Initialization]**
for $\ell = 1$ to L **do** **[Forward Propagation]**
  $\mathbf{s}^{(\ell)} \leftarrow (\mathrm{W}^{(\ell)})^{\mathrm{T}}\mathbf{x}^{(\ell-1)}$
  $\mathbf{x}^{(\ell)} \leftarrow \begin{bmatrix} 1 \\ \theta(\mathbf{s}^{(\ell)}) \end{bmatrix}$
end for
$h(\mathbf{x}) = \mathbf{x}^{(L)}$ **[Output]**

Given weights $w_{i,j}^{(\ell)}$ and $\vec{x}^{(0)} = \vec{x}$, we can calculate all $\vec{x}^{(\ell)}$ and $\vec{s}^{(\ell)}$ through forward propagation.

How to Learn NN From Data?

- Given D, how to learn the weights $W = \left\{w_{i,j}^{(\ell)}\right\}$?

- Intuition: Minimize $E_{in}(W) = \frac{1}{N}\sum_{n=1}^{N} e_n(W)$
- How?
  - Gradient descent: $W(t+1) \leftarrow W(t) - \eta \nabla_W E_{in}(W)$
  - Stochastic gradient descent $W(t+1) \leftarrow W(t) - \eta \nabla_W e_n(W)$

- Key step: we need to be able to evaluate the gradient…
  - Not trivial to do given the network structure
  - Backpropagation is an algorithmic procedure to calculate the gradient

Backpropagation

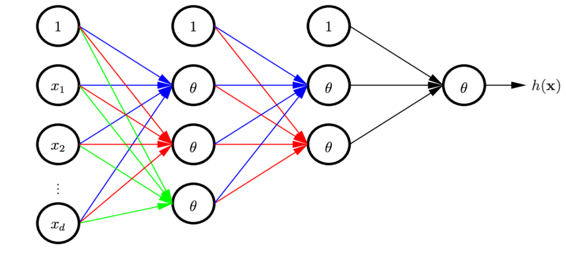
Use dynamic programming to evaluate the gradient

Quick Reminders on Dynamic Programming

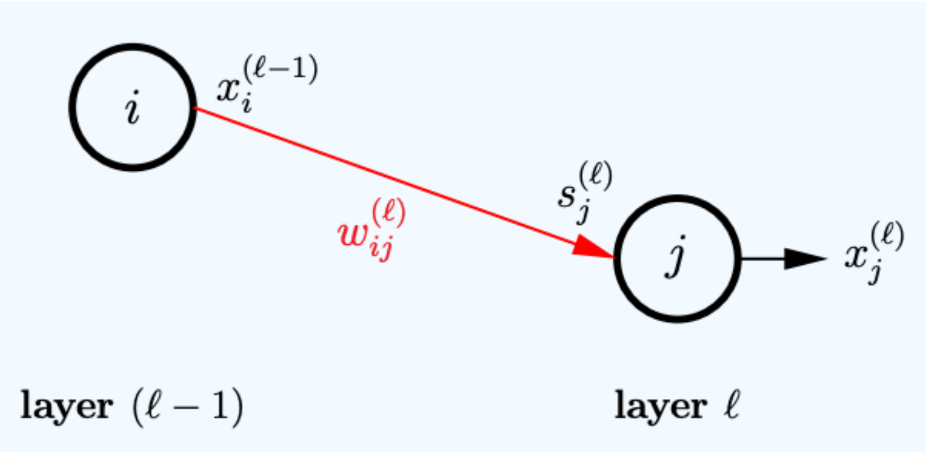
- Example: Fibonacci number
  - $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$
  - $F_0 = 0, F_1 = 1$
  - To evaluate F_N
    - Recursively apply the definition
      - Wasted computation
    - Dynamic programming: evaluate and store $F_0, F_1, \dots, F_N$
      - Use space to exchange for time

- Key step in backpropagation
  - Find a recursive definition of some key quantities
  - Solve the boundary conditions
  - Adopt dynamic programming

Compute the Gradient $\nabla_W e_n(W)$



- To evaluate $\nabla_W e_n(W)$, we need to calculate $\frac{\partial e_n(W)}{\partial w_{i,j}^{(\ell)}}$ for all (i, j, ℓ)
- Zoom in on the region around $w_{i,j}^{(\ell)}$



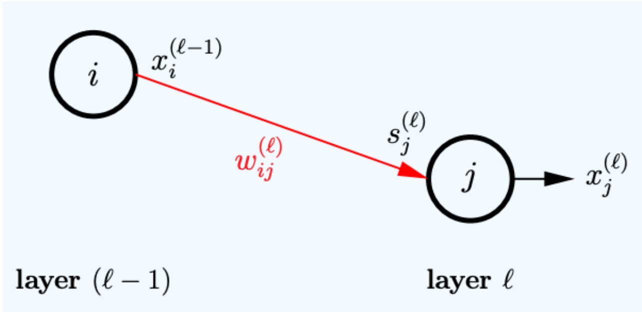
- Apply chain rule

$$\frac{\partial e_n(W)}{\partial w_{i,j}^{(\ell)}} = \frac{\partial e_n(W)}{\partial s_j^{(\ell)}} \frac{\partial s_j^{(\ell)}}{\partial w_{i,j}^{(\ell)}}$$

Compute the Gradient $\nabla_W e_n(W)$

- Apply chain rule

$$\frac{\partial e_n(W)}{\partial w_{i,j}^{(\ell)}} = \frac{\partial e_n(W)}{\partial s_j^{(\ell)}} \frac{\partial s_j^{(\ell)}}{\partial w_{i,j}^{(\ell)}}$$



- Let's look at the second term first
  - Remember $s_j^{(\ell)} = \sum_{i=0}^{d^{(\ell-1)}} w_{i,j}^{(\ell)} x_i^{(\ell-1)}$
  - Therefore, $\frac{\partial s_j^{(\ell)}}{\partial w_{i,j}^{(\ell)}} = x_i^{(\ell-1)}$

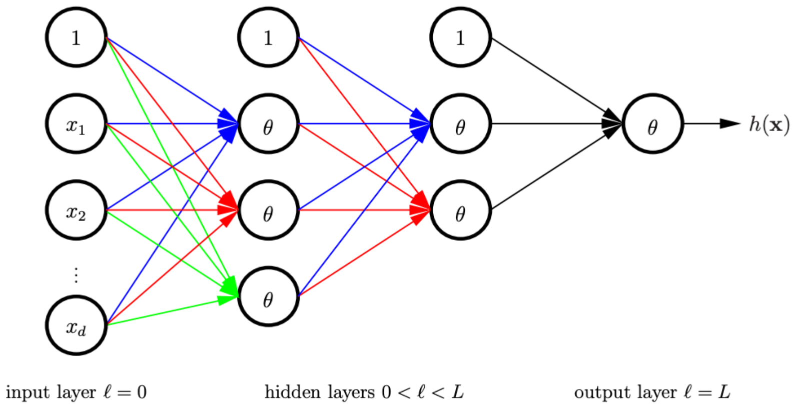
- What about the first term?
  - Let's define $\delta_j^{(\ell)} = \frac{\partial e_n(W)}{\partial s_j^{(\ell)}}$
  - We'll apply dynamic programming style algorithm to deal with this term

- To sum up

$$\boxed{\frac{\partial e_n(W)}{\partial w_{i,j}^{(\ell)}} = \delta_j^{(\ell)} x_i^{(\ell-1)}}$$

Compute $\delta_j^{(\ell)} = \dfrac{\partial e_n(W)}{\partial s_j^{(\ell)}}$

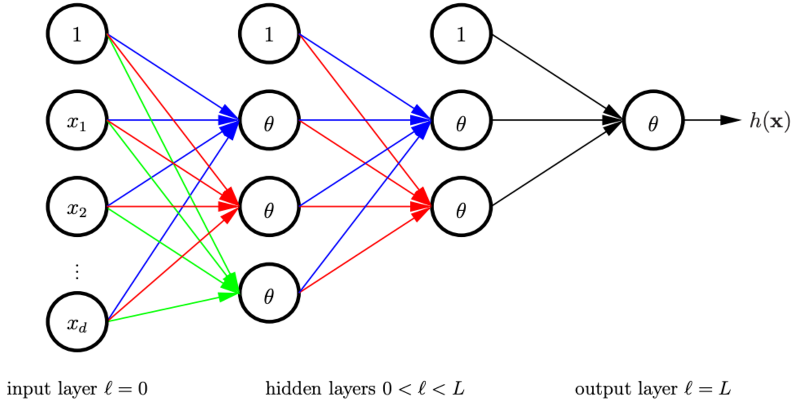
- Using dynamic programming style approach
  - Check boundary case (what is the boundary case?)
  - Write the recursive formulation



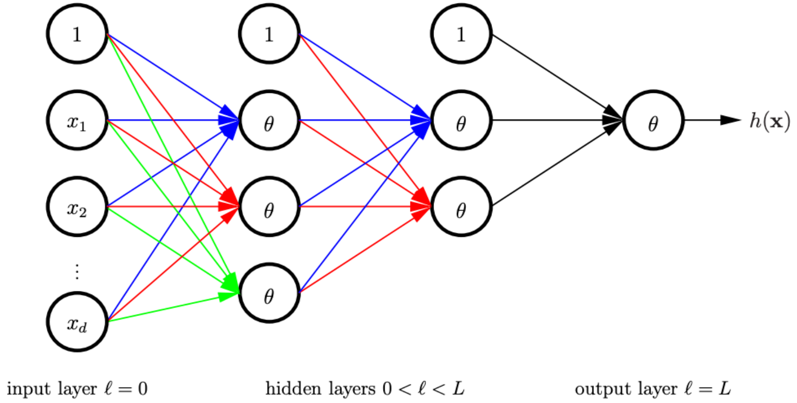
- Check boundary case (when $\ell = L$)
  - Output layer
  - For simplicity, assume we are doing regression and the error is squared error
    - $e_n(W) = \left(s_1^{(L)} - y_n\right)^2$ (Usually only one node in the output layer)
  - $\delta_1^{(L)} = 2(s_1^{(L)} - y_n)$ (similar discussion applies for other differentiable error function)
  - So the boundary condition at L is checked.
  - Next we will derive the backward recursive formulation (hence, backpropagation)

Compute $\delta_j^{(\ell)} = \dfrac{\partial e_n(W)}{\partial s_j^{(\ell)}}$

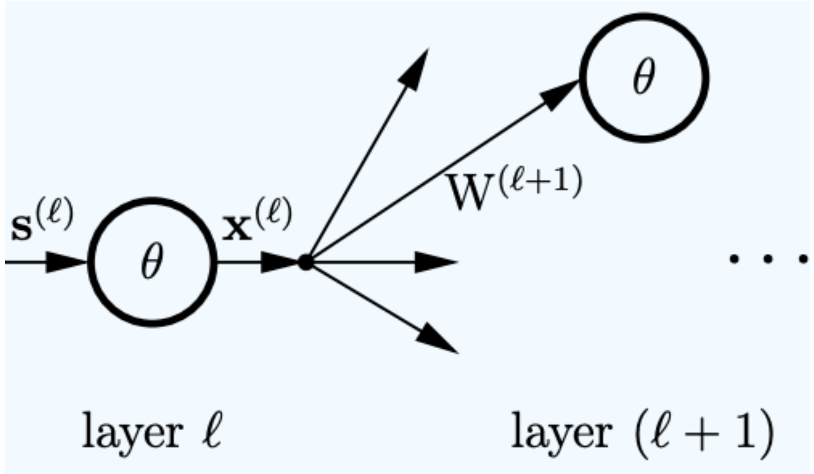
- Zoom in to see the chain of dependencies



Compute $\delta_j^{(\ell)} = \dfrac{\partial e_n(W)}{\partial s_j^{(\ell)}}$

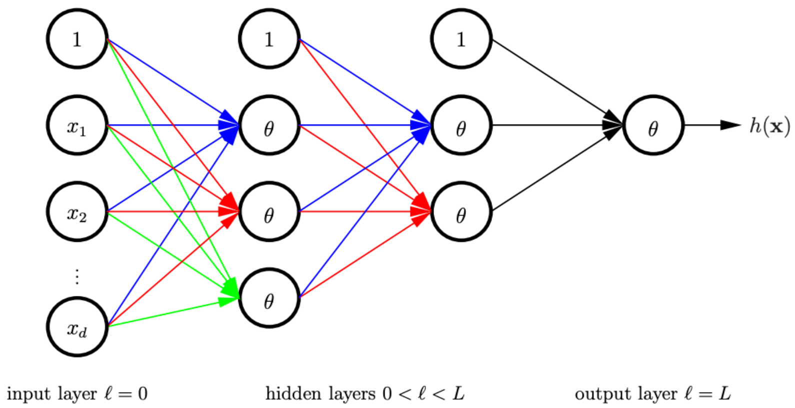


- Zoom in to see the chain of dependencies



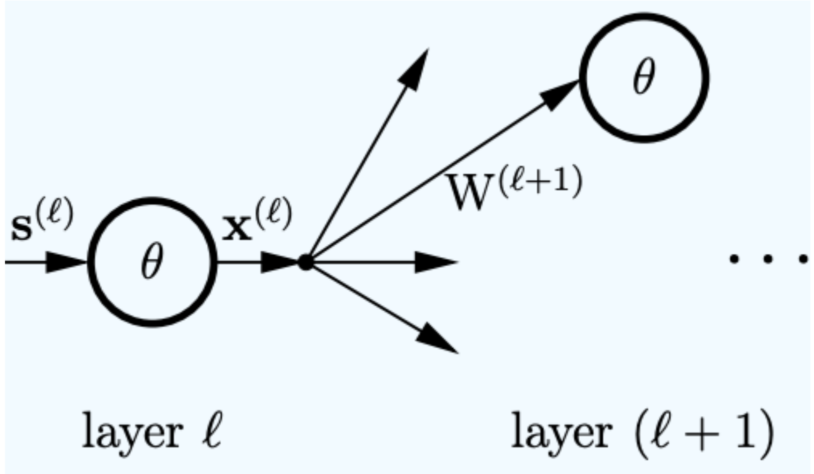
$$\mathbf{s}^{(\ell)} \longrightarrow \mathbf{x}^{(\ell)} \longrightarrow \mathbf{s}^{(\ell+1)}$$

Compute $\delta_j^{(\ell)} = \dfrac{\partial e_n(W)}{\partial s_j^{(\ell)}}$



- Zoom in to see the chain of dependencies

For $\theta(s) = \tanh(s)$,
$\theta'(s) = 1 - \theta(s)^2$



$$\mathbf{s}^{(\ell)} \longrightarrow \mathbf{x}^{(\ell)} \longrightarrow \mathbf{s}^{(\ell+1)}$$

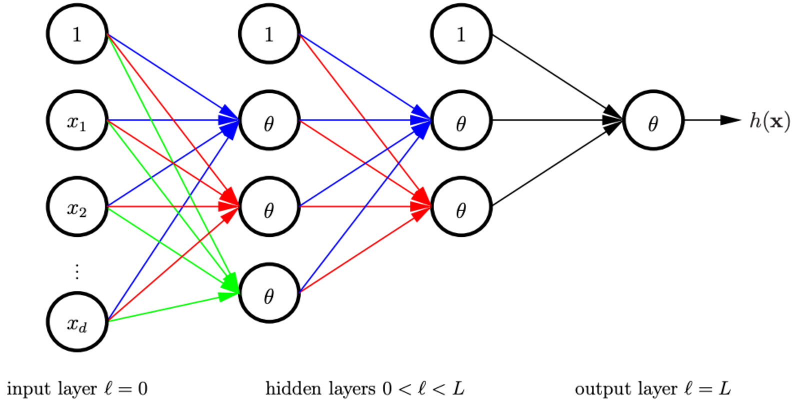
$$\delta_j^{(\ell)} = \frac{\partial e_n(W)}{\partial s_j^{(\ell)}}$$

$$= \sum_{k=1}^{d^{(\ell+1)}} \frac{\partial e_n(W)}{\partial s_k^{(\ell+1)}} \frac{\partial s_k^{(\ell+1)}}{\partial x_j^{(\ell)}} \frac{\partial x_j^{(\ell)}}{\partial s_j^{(\ell)}}$$

$$= \sum_{k=1}^{d^{(\ell+1)}} \delta_k^{(\ell+1)} w_{j,k}^{(\ell+1)} \theta'\left(s_j^{(\ell)}\right)$$

We have the backward recurve definition!

Compute $\delta_j^{(\ell)} = \frac{\partial e_n(W)}{\partial s_j^{(\ell)}}$



- We can calculate $\delta_j^{(\ell)}$ in a dynamic programming manner:

- Boundary condition: $\delta_1^{(L)} = 2(s_1^{(L)} - y_n)$
- Recursive formulation: $\delta_j^{(\ell)} = \sum_{k=1}^{d^{(\ell+1)}} \delta_k^{(\ell+1)} w_{j,k}^{(\ell+1)} \theta'\left(s_j^{(\ell)}\right)$

- Calculate $\delta_j^{(\ell)}$ for $\ell < L$ in a backward manner

Backpropagation Algorithm

- Recall that $\frac{\partial e_n(W)}{\partial w_{i,j}^{(\ell)}} = \delta_j^{(\ell)} x_i^{(\ell-1)}$
- Backpropagation Algorithm
  - Initialize $w_{i,j}^{(\ell)}$ randomly [You will discuss the impacts of initialization in HW5]
  - For $t = 1$ to T
    - Randomly pick a point from D (for stochastic gradient descent)
    - Forward propagation: Calculate all $x_i^{(\ell)}$ and $s_i^{(\ell)}$
    - Backward propagation: Calculate all $\delta_j^{(\ell)}$
    - Update the weights $w_{i,j}^{(\ell)} \leftarrow w_{i,j}^{(\ell)} - \eta \delta_j^{(\ell)} x_i^{(\ell-1)}$
  - Return the weights

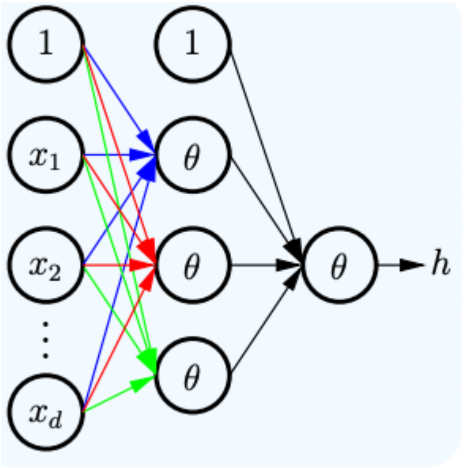
Discussion

- Backpropagation is gradient descent with efficient gradient computation
- Note that the E_{in} is not convex in weights
- Gradient descent doesn't guarantee to converge to global optimal

- Common approaches:
  - Run it many times
  - Each with a different initialization (the choice of initialization matters)
    - Initialization matters (more discussion next lecture)
    - Initializing at 0 is not a good choice (Q6b of HW5)
    - Initializing at larger weights is not a good idea for tanh as activation function (Q6a of HW5)

Single Hidden-Layer Neural Network

- How do we write a hypothesis in single-hidden layer mathematically?
  - $h(\vec{x}) = \theta\left(w_{0,1}^{(2)} + \sum_{j=1}^{d^{(1)}} w_{j,1}^{(2)} x_j^{(1)}\right)$
    $= \theta\left(w_{0,1}^{(2)} + \sum_{j=1}^{d^{(1)}} w_{j,1}^{(2)} \theta\left(\sum_{i=0}^{d^{(0)}} w_{i,j}^{(1)} x_i\right)\right)$



- How do we write a Kernel SVM hypothesis (linear model with nonlinear transformation)
  - $g(\vec{x}) = \theta\left(b^* + \sum_{\alpha_n^*>0} \alpha_n^* y_n K(\vec{x}_n, \vec{x})\right)$

- Interpretation:
  - The hidden layer is like "feature transform"
  - Shallow learning vs. deep learning
  - More discussion on neural networks and deep learning next lecture

Neural Network is Expressive

- Universal approximation theorem:
  - A feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of $\mathbb{R}^n$, under mild assumptions on the activation function.
  - A single-hidden-layer NN can approximate ANY continuous target function!
- We also seem to only discuss how to minimize E_{in}

What about overfitting?

Regularization in Neural Networks

Weight-Based Regularization

- Weight decay

$$E_{aug}(W) = E_{in}(W) + \frac{\lambda}{N}\sum_{i,j,\ell}\left(w_{i,j}^{(\ell)}\right)^2$$

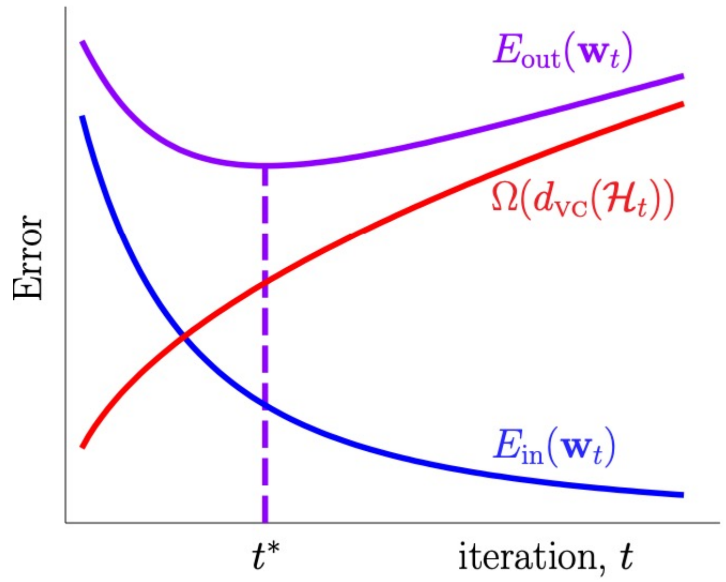
- Weight elimination

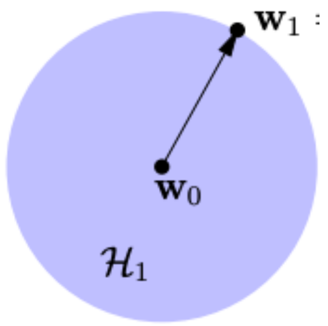
$$E_{aug}(W) = E_{in}(W) + \frac{\lambda}{N}\sum_{i,j,\ell}\frac{\left(w_{i,j}^{(\ell)}\right)^2}{1+\left(w_{i,j}^{(\ell)}\right)^2}$$

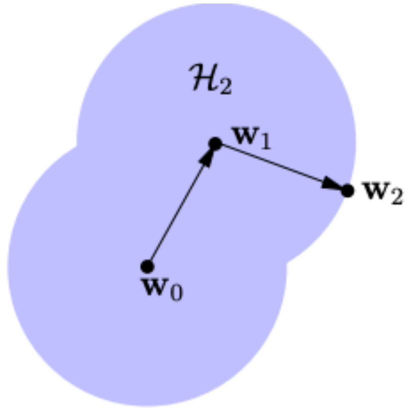
  - When $w_{i,j}^{(\ell)}$ is small, approximates weight decay
  - When $w_{i,j}^{(\ell)}$ is large, approximates adding a constant (no impacts to gradient)
  - “Decaying” more on smaller weights (i.e., eliminating small weights)

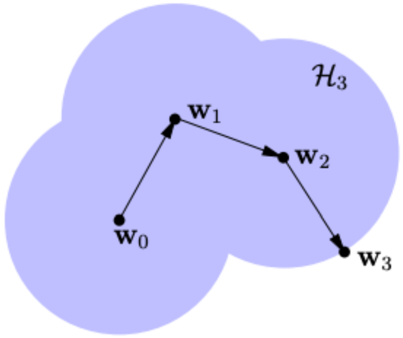
Early Stopping

- Consider gradient descent (GD)
  - H_1: the set of hypothesis GD can reach at $t = 1$
  - H_2: the set of hypothesis GD can reach at $t = 2$
  - …
  - $H_1 \subseteq H_2 \subseteq H_3 \subseteq \cdots$





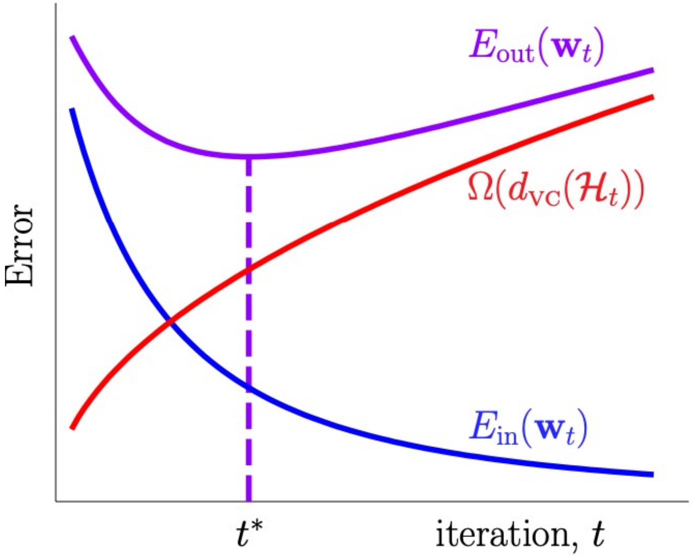


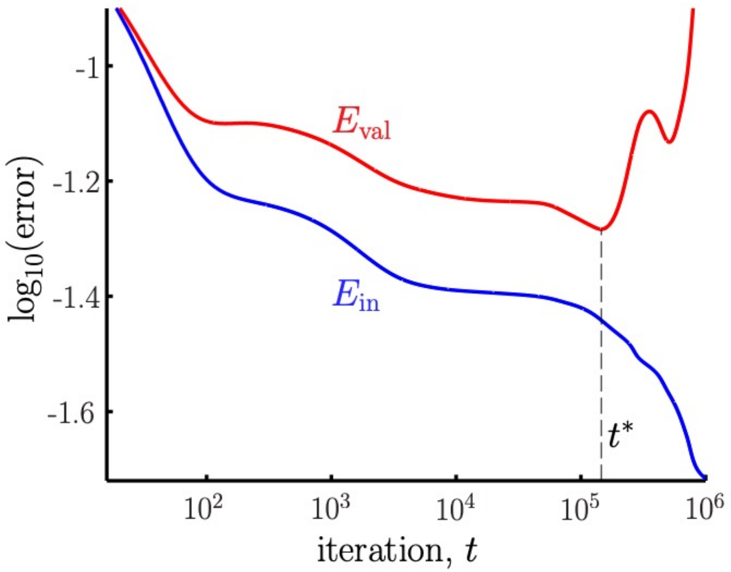


Early Stopping

- Stopping gradient descent early is a regularization method
    - Constrain the hypothesis set

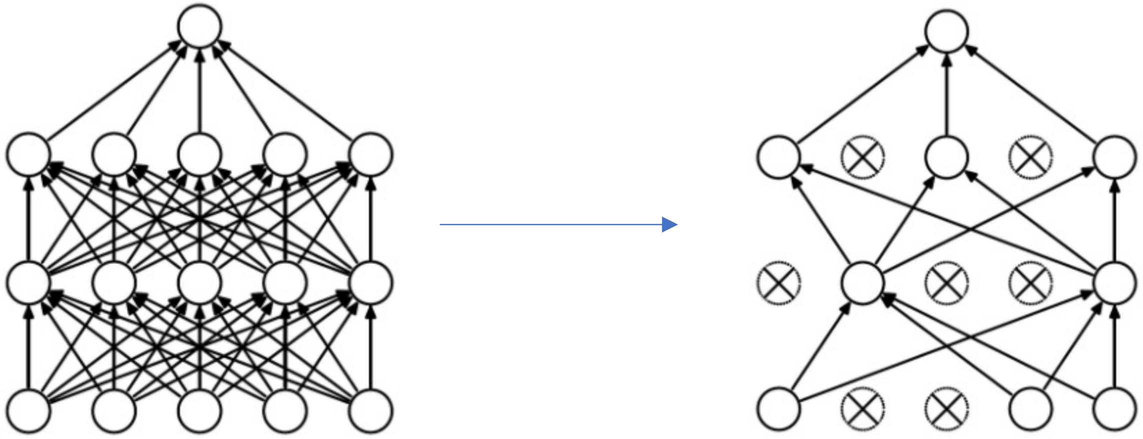
- How to find the optimal stopping point t^*?
    - Using validation is a common approach





Dropout

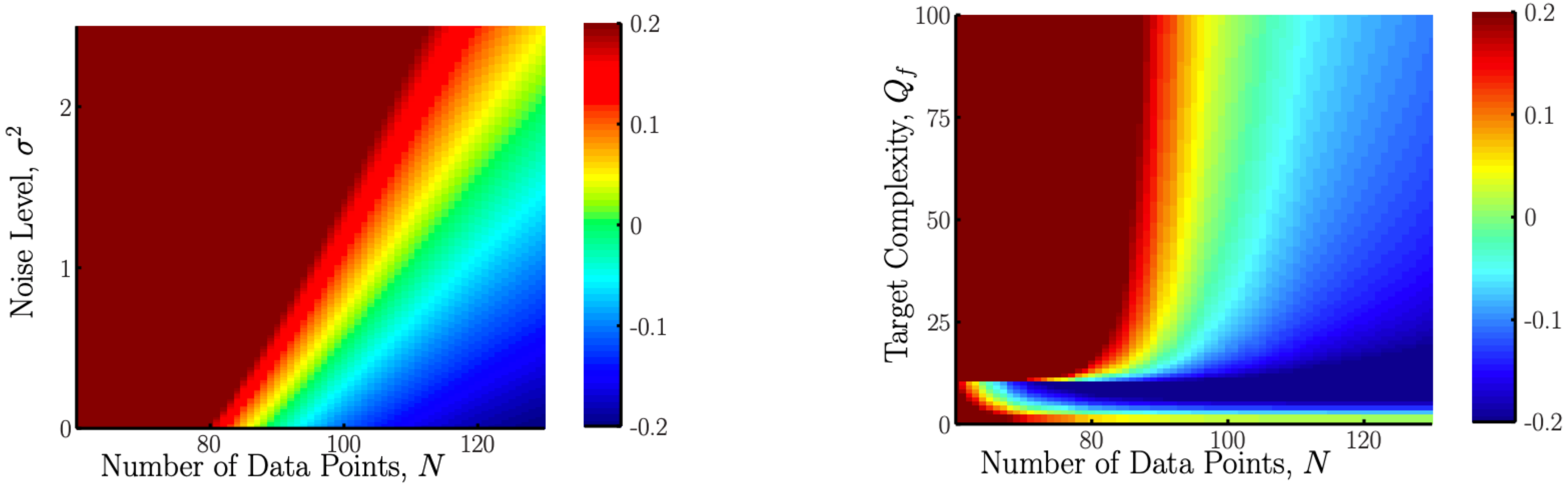
- Neural networks is very expressive (low bias, potentially high variance)
- Dropout
  - Randomly drop p portion of the weights during training



  - Learn many models with dropout
  - Average them during prediction (reduce weights by a ratio of p)

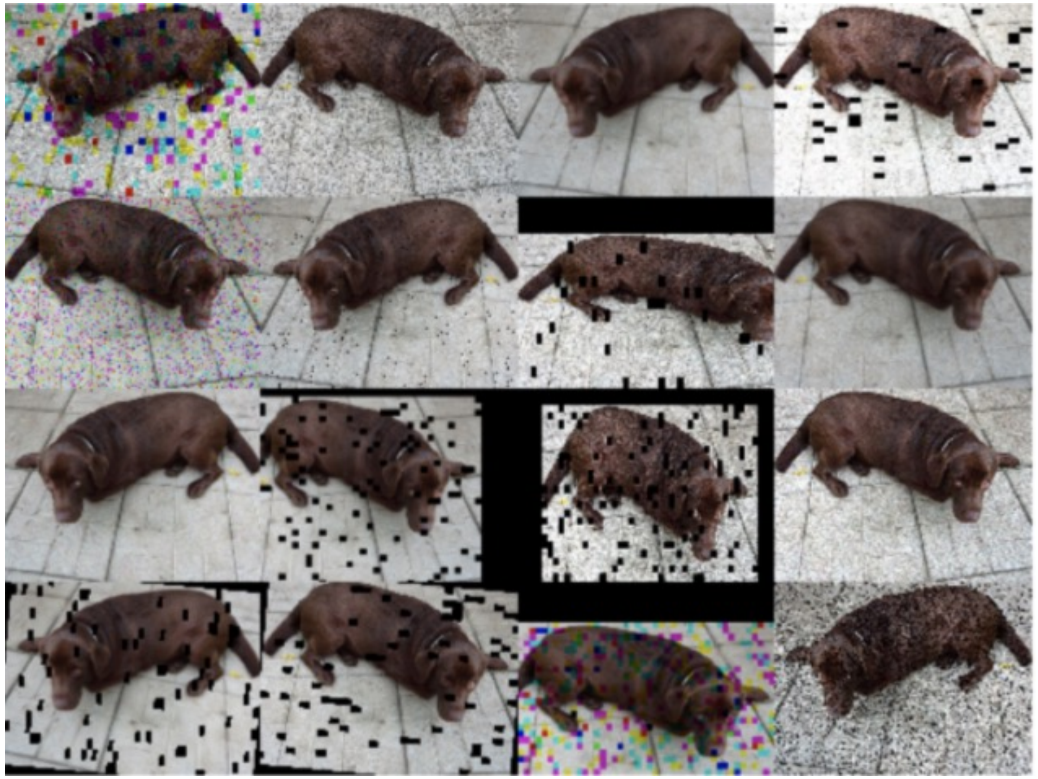
A Nontraditional Method to Avoid Overfitting

- What's the cause of overfitting?



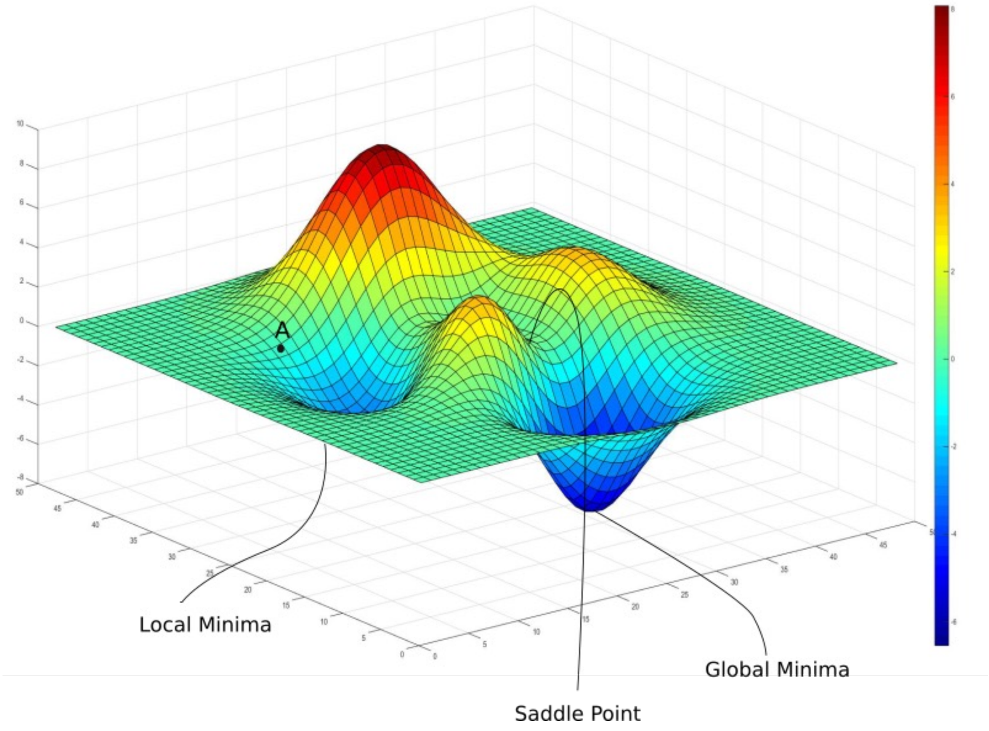
- Fitting the noise instead of the target
- Regularization: Constrain H so it's not that powerful to fit noise
- How about adding noises to data?

Adding Noises as Regularization



Initialization

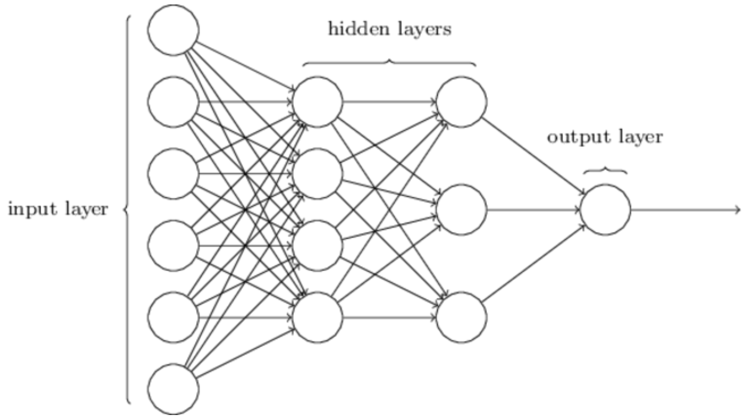
Error is Nonconvex in Neural Networks



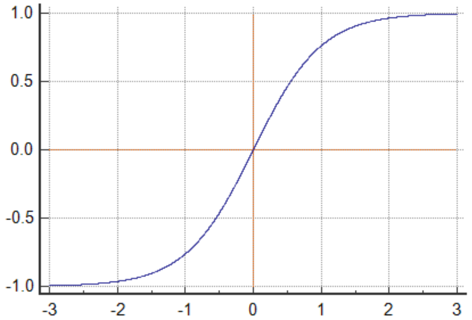
- We mostly adopt gradient-descent-style algorithms for optimization.
- No guarantee to converge to global optimal.
- Need to run it many times.
- Initialization matters!

Vanishing Gradient Problem

- Backpropagation
  - $\frac{\partial e_n(W)}{\partial w_{i,j}^{(\ell)}} = \delta_j^{(\ell)} x_i^{(\ell-1)}$
  - $\delta_j^{(\ell)} = \theta'\left(s_j^{(\ell)}\right) \sum_{k=1}^{d^{(\ell+1)}} \delta_k^{(\ell+1)} w_{j,k}^{(\ell+1)}$



- If we use activation function $\theta(s) = \tanh(s)$
  - $\theta'(s) = 1 - \theta(s)^2 < 1$



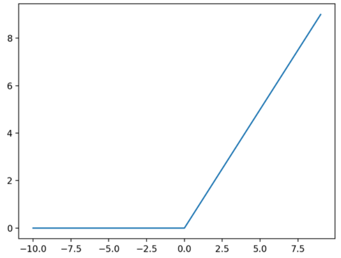
- In deep learning with a lot of layers,
  - the gradient might vanish
  - hard to update the early layers

Vanishing Gradient Problem

- $\delta_j^{(\ell)} = \theta'\left(s_j^{(\ell)}\right) \sum_{k=1}^{d^{(\ell+1)}} \delta_k^{(\ell+1)} w_{j,k}^{(\ell+1)}$

- There is also a corresponding "exploding gradient problem"

- What can we do
  - Choose more suitable activation functions
    - One common choice is Rectified Linear Unit (ReLU) and its variant
      - $\theta(s) = \max(0, s)$



  - Choose better initialization
    - Many approaches

Weight Initialization

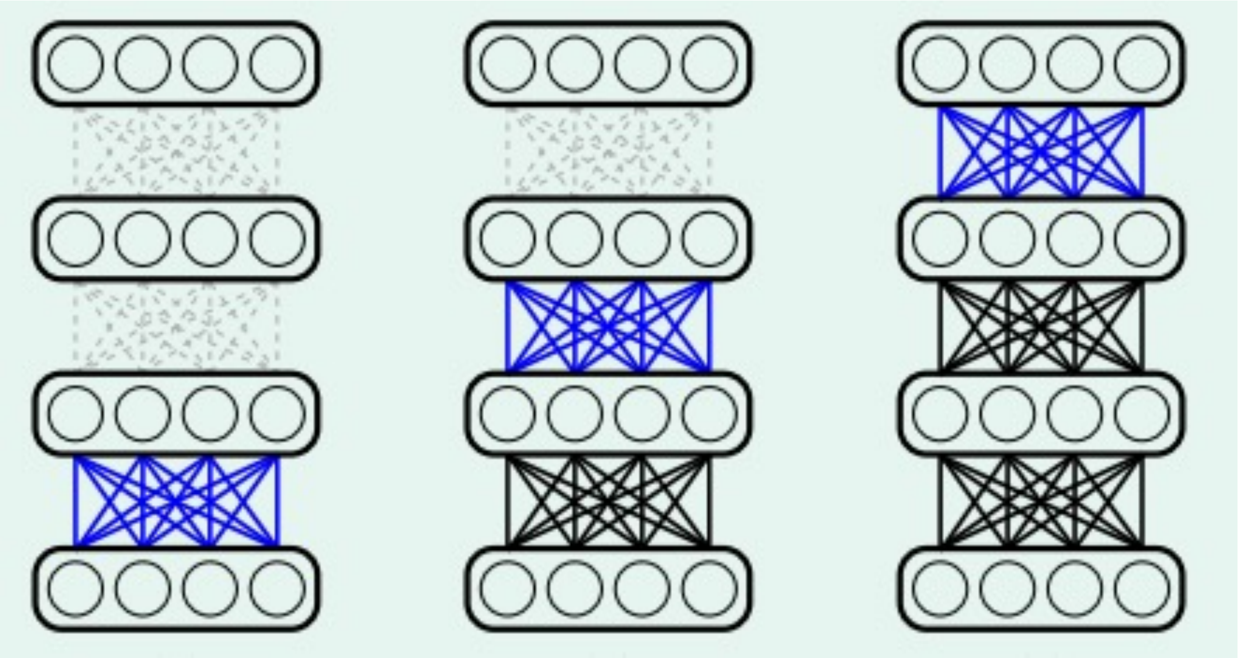
- Initializing weights to 0 is a bad idea
  - Q5 of HW1

- Randomly Initializing weights to regions so that vanishing/exploding gradients are less likely to happen
  - Activation-function dependent
    - e.g., Xavier initialization for tanh

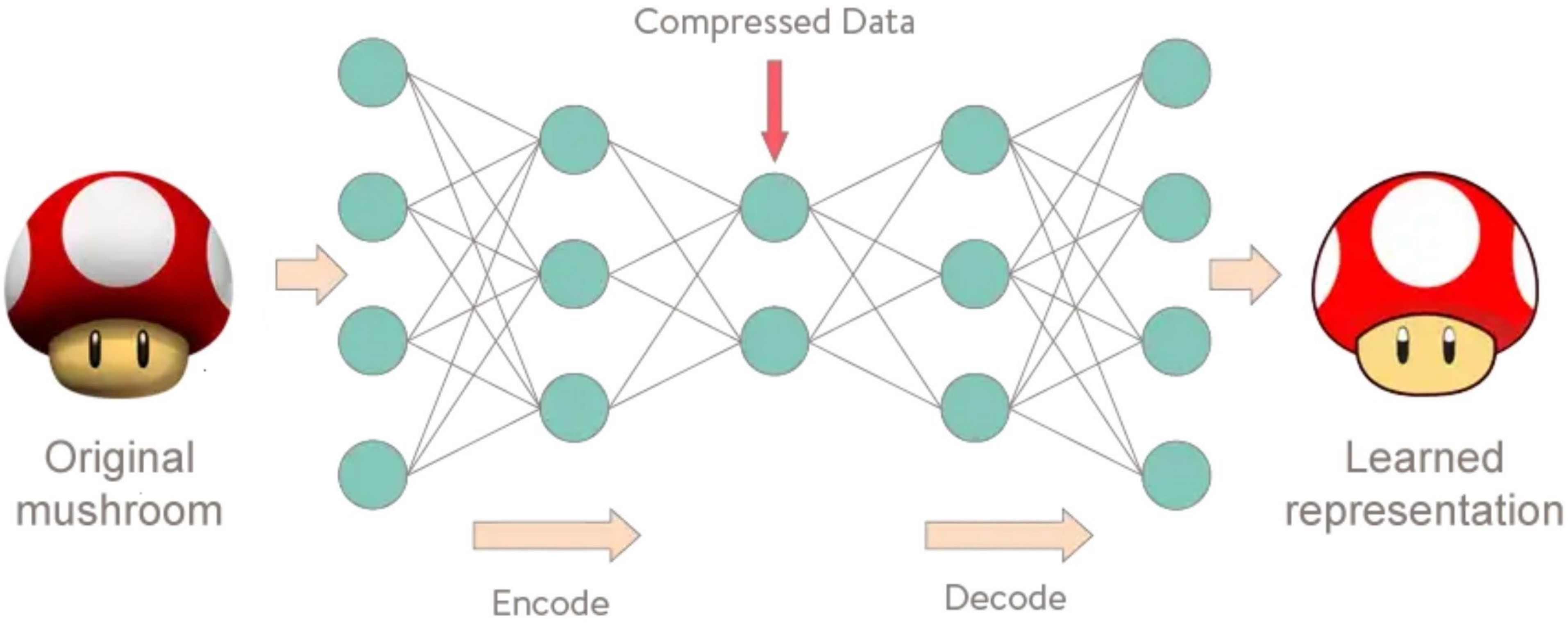
- Learning the initialization
  - E.g., autoencoder

Initialization

- Hard to initialize the entire network well.
- Intuition: Initialize the weights layer by layer such that each layer preserves the properties of the previous layer.



Autoencoder



Unsupervised learning!

https://www.meetup.com/Le-Mans-School-of-AI/photos/29159837/477498782/