



Neural Networks

CSCI 601-471/671 (NLP: Self-Supervised Models)

<https://self-supervised.cs.jhu.edu/sp2025/>

How was HW1

- With a show of hand, which one best applies to you:
 1. It was smooth sailing through things I knew; my hamster nearly finished it.
 2. it was familiar stuff but I had to learn or refresh a few things.
 3. It was like shoveling snow in the middle of a blizzard, it just kept getting worse
 4. It was so challenging, it felt like climbing Mount Everest with slippers on.

“Can I use external libraries?” No, unless specified!

- Use the basic Python functions (no external libraries), unless explicitly specified.
- In almost all places, you’re not expected to write more than 3-4 lines of code.

```
[ ] # a function that returns the top `k` most similar words to `input_word` .
def my_most_similar(input_word, k):
    words = embeddings.vocab.keys() # list of words covered by this word embedding
    input_word_emd = embeddings[input_word]

    ### START CODE HERE ###
    ### END CODE HERE ###

    return top_k_most_similar_words

my_most_similar('cat', 10)
```

“I can’t install”

- It may be frustrating to deal with software dependencies...
- But it's [going to be] part of your job's reality ...
- Embrace it! 😊

HW2 is released

- Have you seen it?
- Due next Tuesday noon.
 - Feels like a long time away?
 - **it's due in 120 hours!**

Recap: Language Modeling

- **Language Modeling:** estimating distributions over language.
- **One approach** we previously saw: counting word co-occurrences.
 - **Pro:** **easy** — just count!
 - **Con:** **difficult** to scale to longer context due to **the sparsity challenge**.
- Another approach:
 - Using a **learnable function** that can estimate word transition probabilities.
 - **Now:** What are these learnable functions and how can we train them.

Neural Networks: Chapter Plan

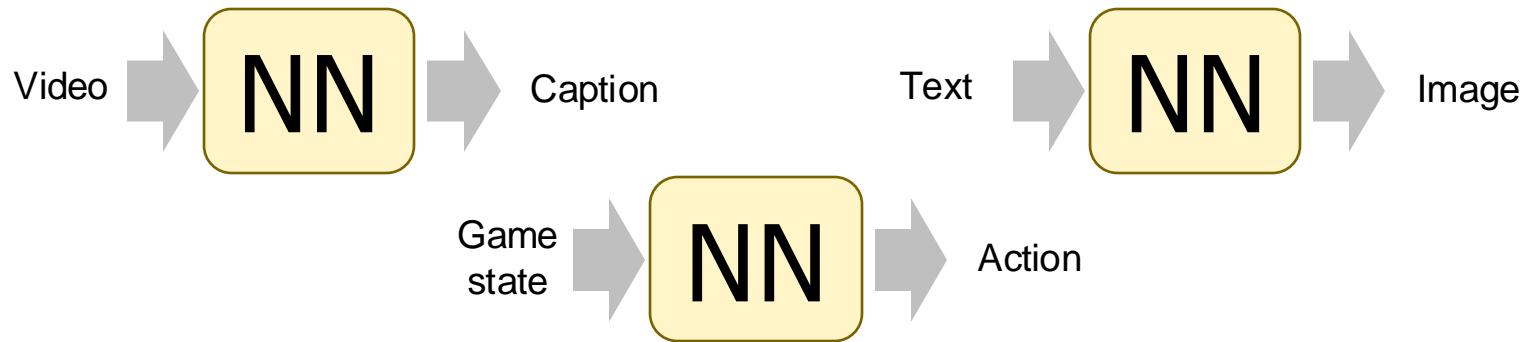
1. Defining neural networks (feedforward nets)
2. Neural nets: brief history
3. Algebra background for training neural nets
4. Training neural networks: analytical backpropagation
5. Backprop in practice

Chapter goal: Get comfortable with thinking, designing and building neural networks
— very powerful modeling tools.

Feedforward Neural Nets

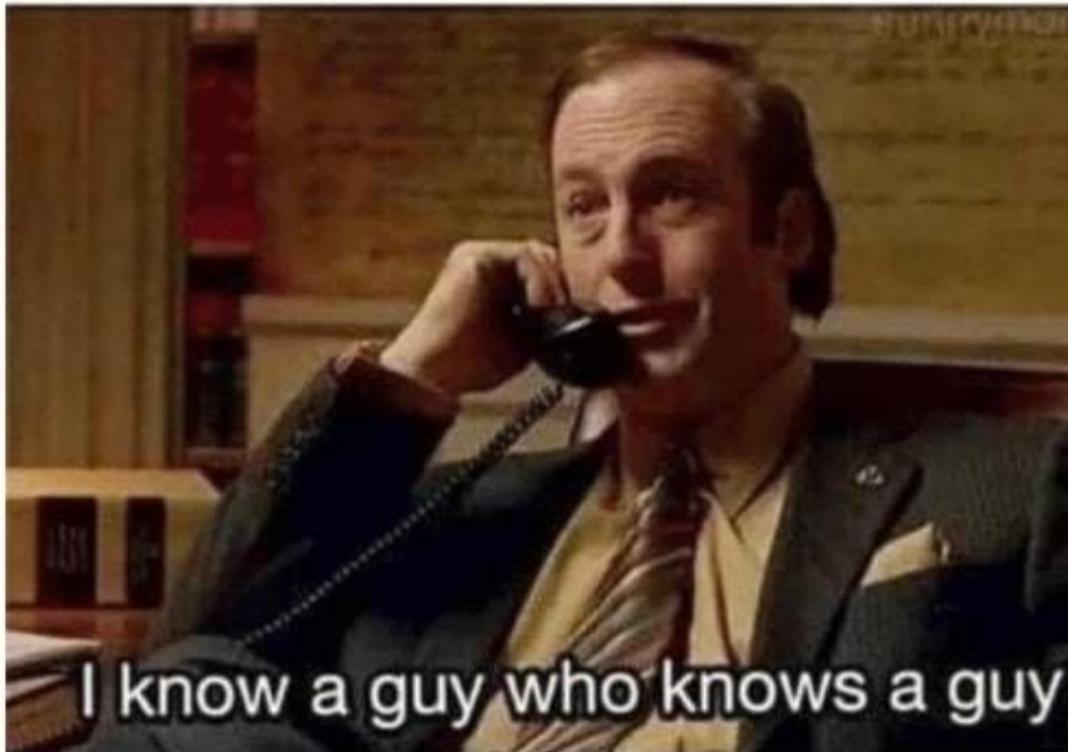
Neural Networks

- What are neural networks?
 - Functions that take an input and produce an output.



- What is inside this box?

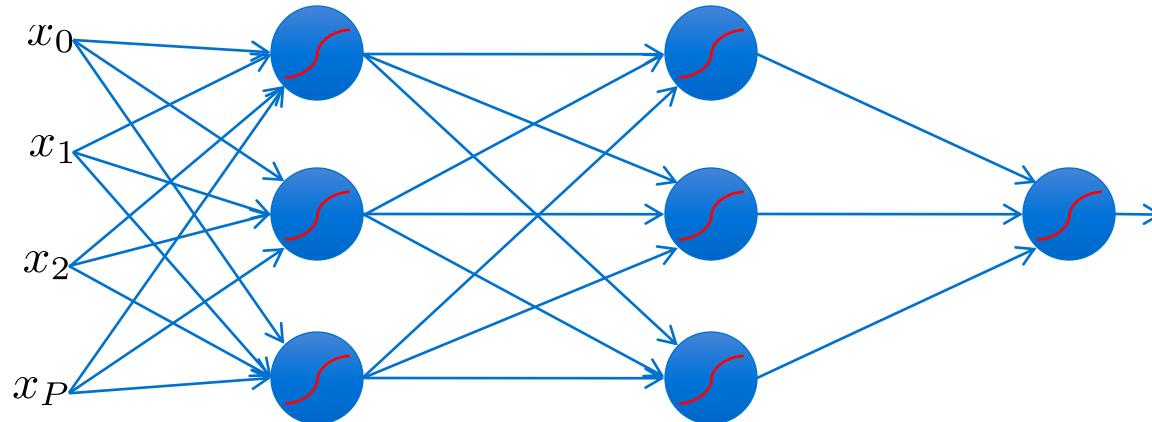
What's Inside Neural Nets?



I know a guy who knows a guy

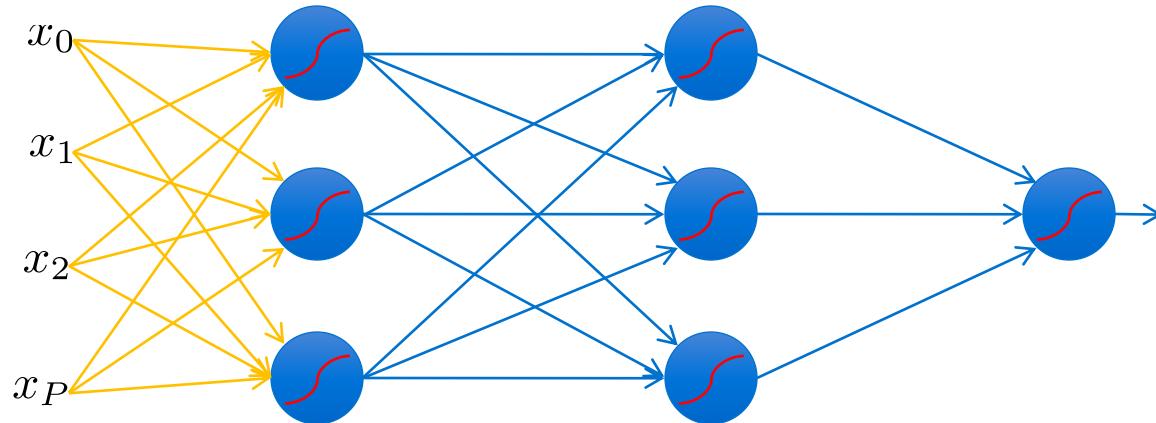
Feedforward networks

- This is a particular class called “feedforward” networks.
 - Cascade neurons together



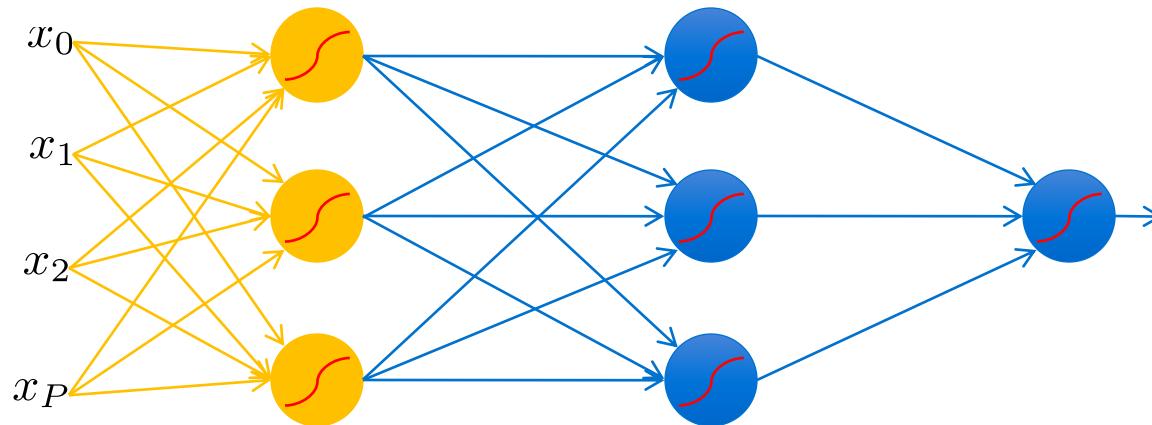
Feedforward networks

- Inputs multiplied by initial set of weights



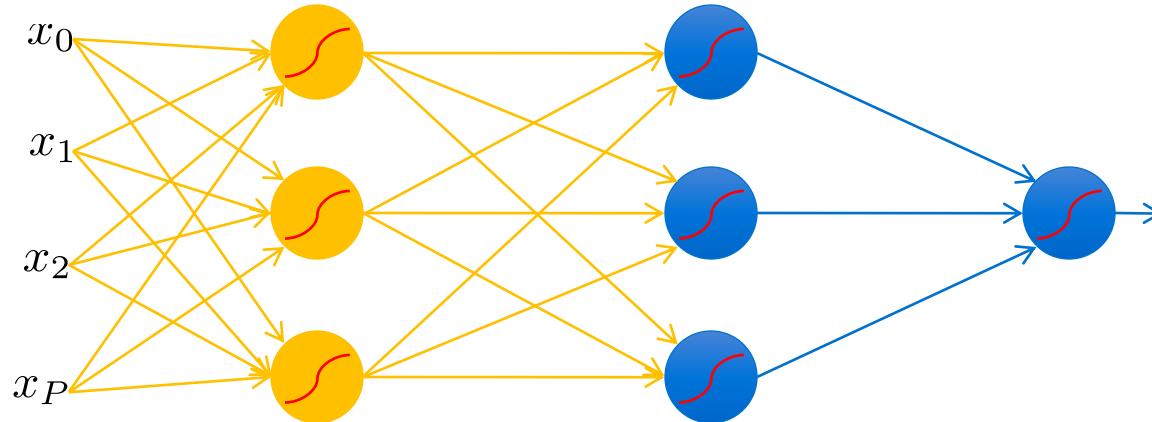
Feedforward networks

- Intermediate “predictions” computed at first hidden layer



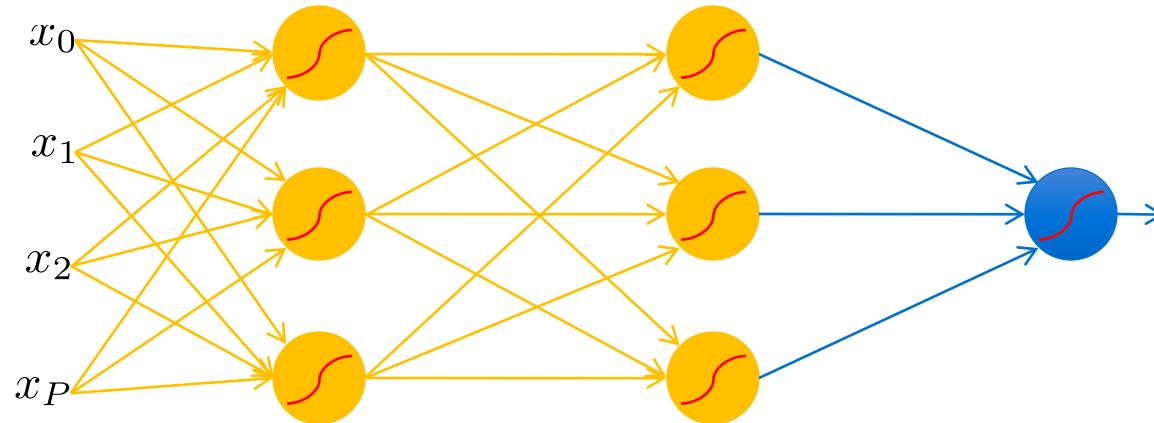
Feedforward networks

- Intermediate predictions multiplied by second layer of weights
- Predictions are **fed forward** through the network



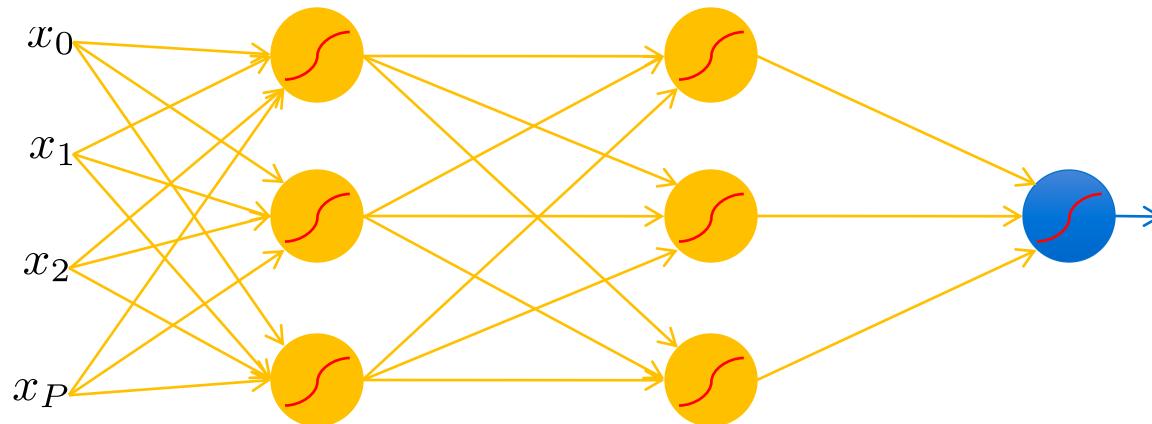
Feedforward networks

- Compute second set of intermediate predictions



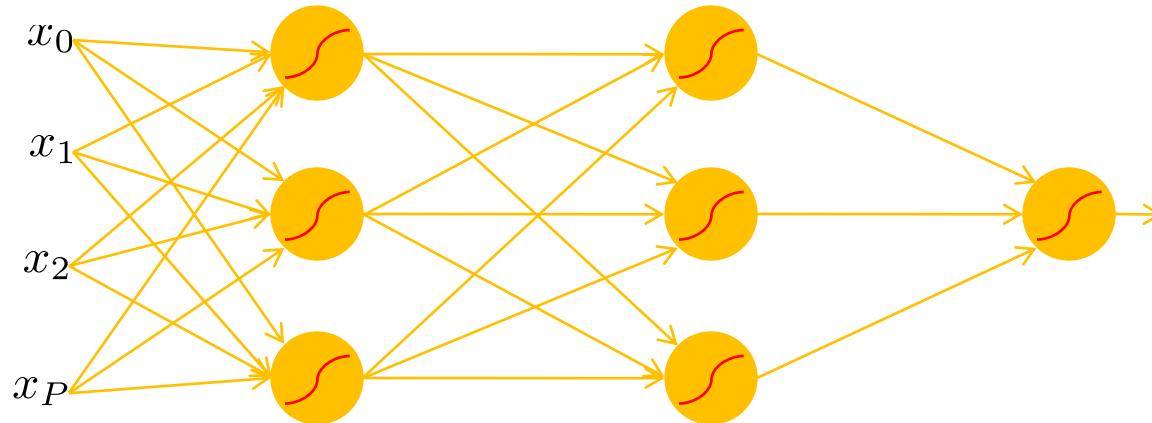
Feedforward networks

- Multiply by final set of weights



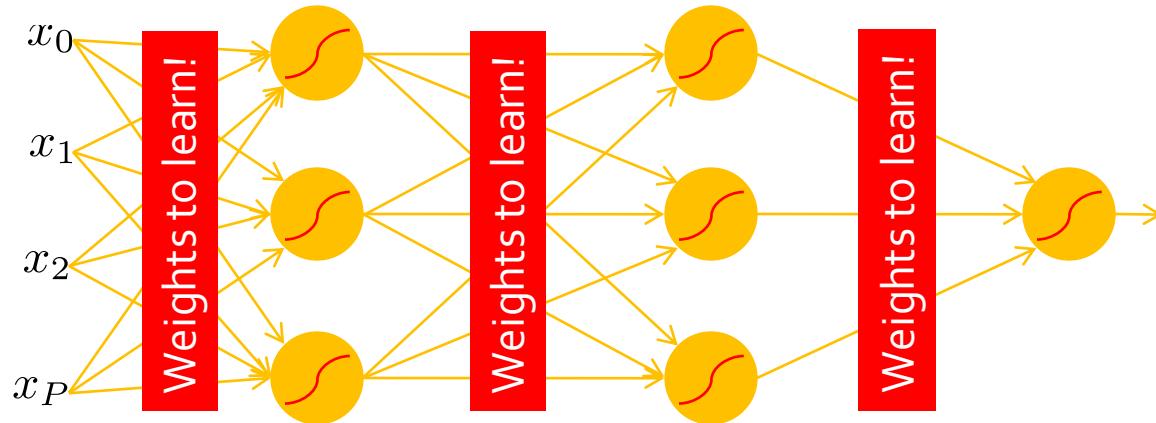
Feedforward networks

- Aggregate all the computations in the output
 - e.g. probability of a particular class



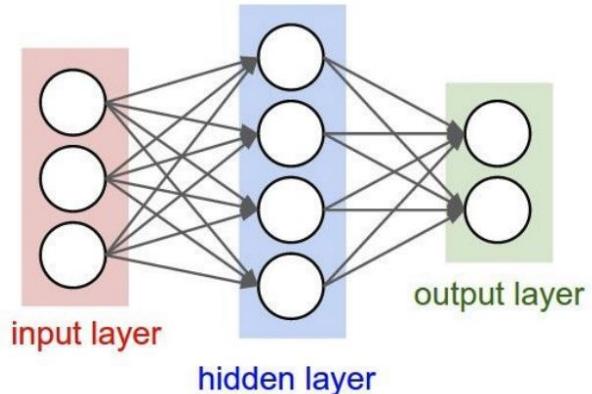
Feedforward networks

- All the intermediate parameters are ought to be learned.



Feedforward Neural Network

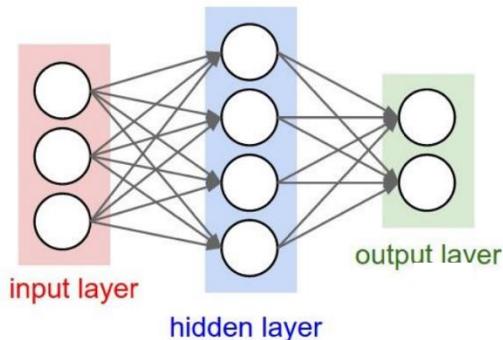
- Neural Networks are functions!
 - Function class for approximating real-valued, discrete-valued and vector valued target functions.
 - NN: $X \rightarrow Y$ where $X = [0,1]^n$, or \mathbb{R}^n and $Y = [0,1]^d, \{0,1\}^d$
- Example: A **2-layer** neural network
 - The input, hidden and output variables are represented by nodes
 - The links are the weight parameters
 - Arrows denote direction of information flow through the network



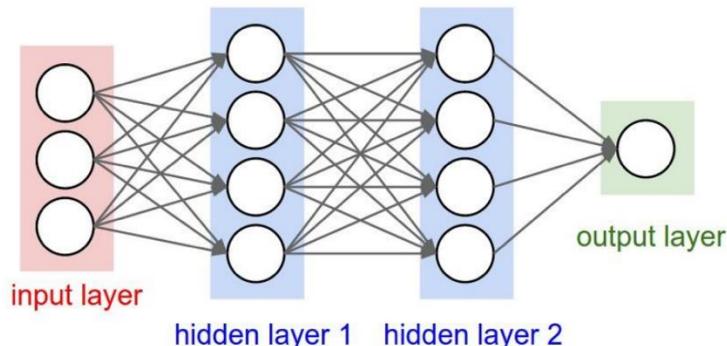
Neural Network: Making it bigger

Add more layers, or wider layers!

A **2-layer** neural network



A **3-layer** neural network

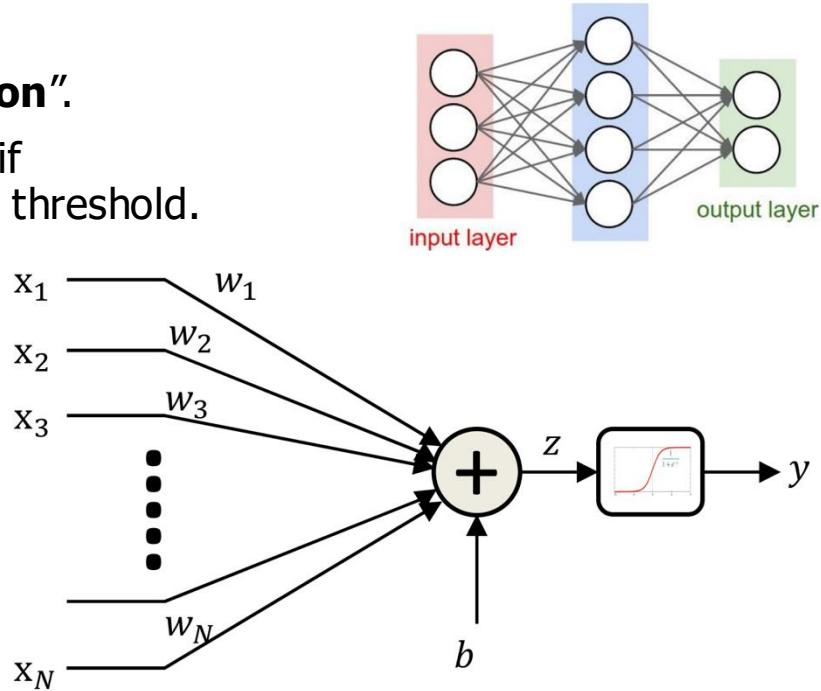


Feedforward Neural Network: The Neurons

- A mathematical model of neuron is “**perceptron**”.
- It consists of a non-linear function that “fires” if the affine (linear) function of inputs is above a threshold.

$$y = \sigma \left(b + \sum_{i=1}^N w_i x_i \right)$$

$$\sigma(z) = \frac{1}{1+e^{-x}} \text{ (sigmoid function)}$$

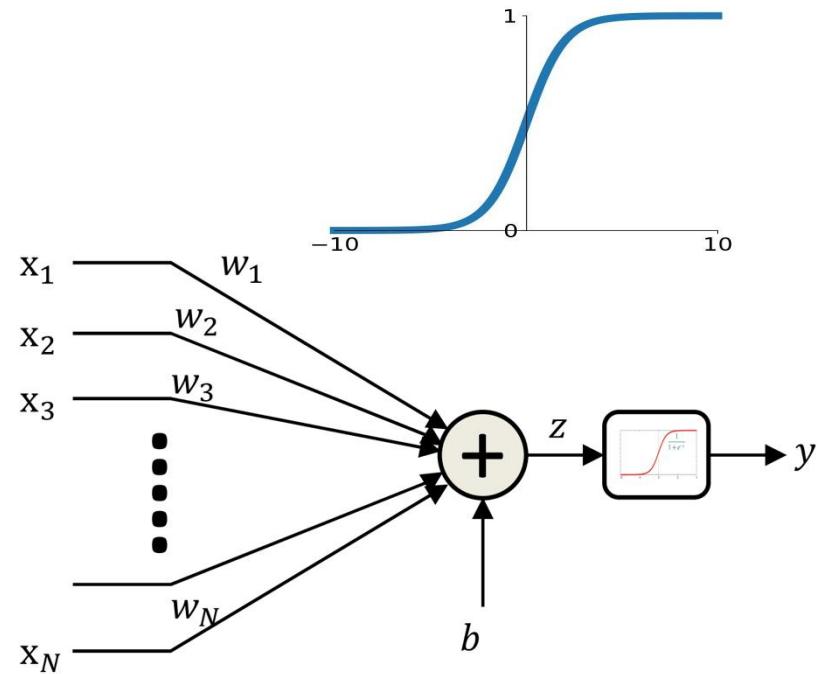


Feedforward Neural Network: The Neurons

- Sigmoid is a “squashing” function.
 - It maps small inputs to zero.
 - It maps large inputs to one.

$$y = \sigma \left(b + \sum_{i=1}^N w_i x_i \right)$$

$$\sigma(z) = \frac{1}{1+e^{-x}} \text{ (sigmoid function)}$$

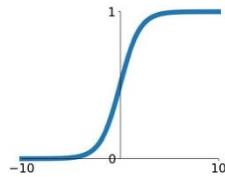


Other Activation Functions

Does not always have to be a squashing function

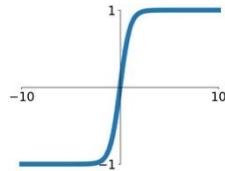
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



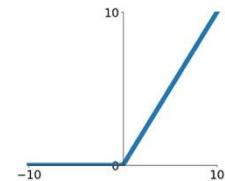
tanh

$$\tanh(x)$$



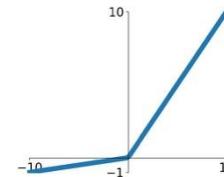
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

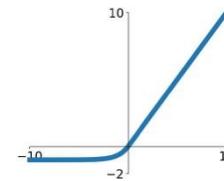


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

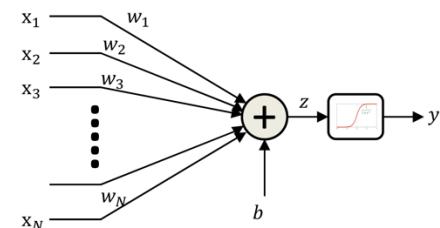
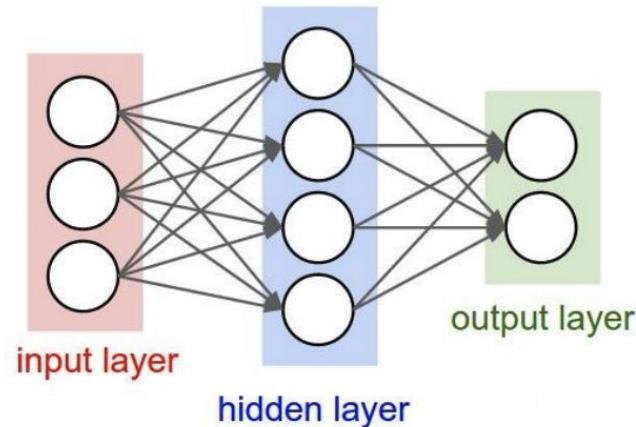
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



We will talk about their pro/cons later!

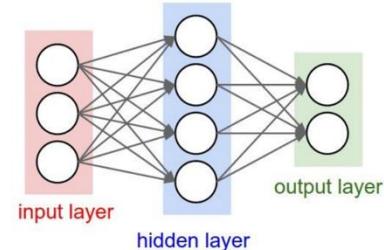
Terminology: Multi-Layer Perceptron (MLP)

- Multi-layer Perceptron (MLP):
 - A feedforward network with perceptron as its nodes.
- A feedforward network does **not** have to be an MLP.
 - But people sometimes use the names interchangeably! 
- The original MLP [McCulloch–Pitts] was based on “threshold” activation.



Formally Defining a 2 Layer Network

- Example: A **2-layer** MLP network
 - The input, hidden and output **variables** are represented by **nodes**
 - The links are the **weight parameters**
 - Arrows denote **direction of information flow** through the network



$$f(\mathbf{x}) = W_2 g(W_1 \mathbf{x}) \quad \mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^d$$

$$g(\mathbf{z}) = [\sigma(z_1), \dots, \sigma(z_h)] \text{ (nonlinearity)} \quad \sigma(z_i) = \frac{1}{1+e^{-x}} \text{ (sigmoid function)}$$

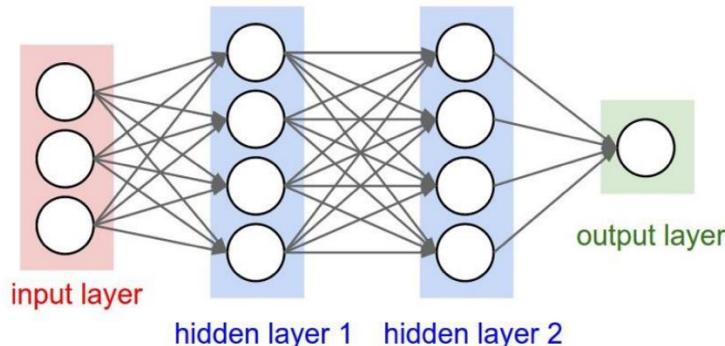
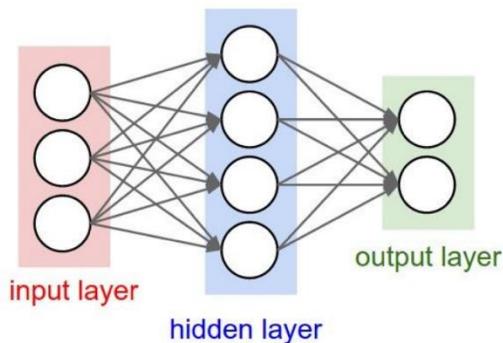
- $W_1 \in \mathbb{R}^{h \times n}$ and $W_2 \in \mathbb{R}^{d \times h}$ are the **parameters** that need to be learned.

Quiz Time (1)

- What is needed to fully specify a neural network?
 1. Architecture (which input goes through what function etc.)
 2. Parameters of the function (the weights)
 3. Both

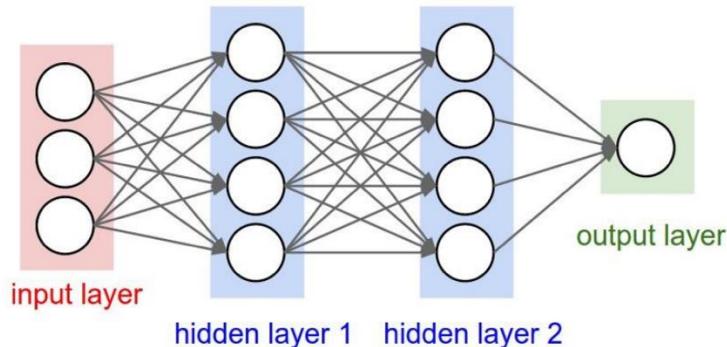
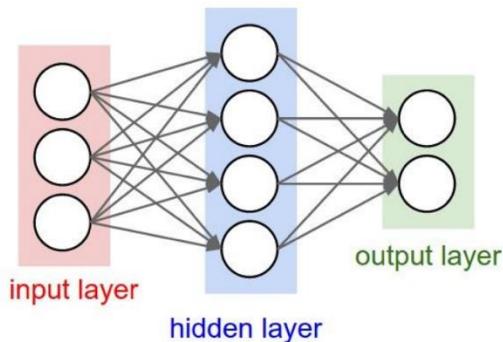
Quiz Time (2)

- Which of the followings has more parameters?

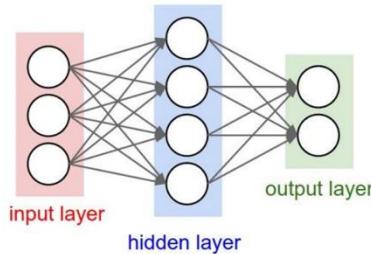


Quiz Time (3)

- Given an input to these models, which of them take longer to compute an output?



Why Add Non-linearity?



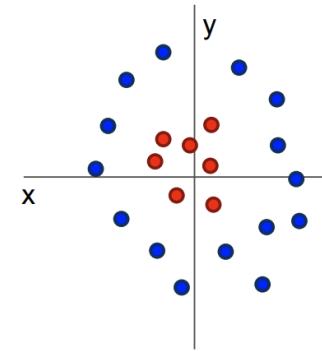
- Without non-linearity, the overall model amounts to a linear model.

$$f(\mathbf{x}) = W_2 g(W_1 \mathbf{x}) \rightarrow \tilde{f}(\mathbf{x}) = W_2 W_1 \mathbf{x} = W_3 \mathbf{x} \text{ (a linear function)}$$

drop g

- A linear function cannot approximate complex tasks.

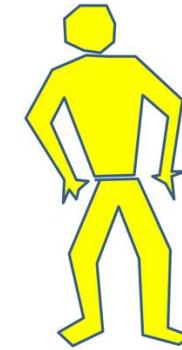
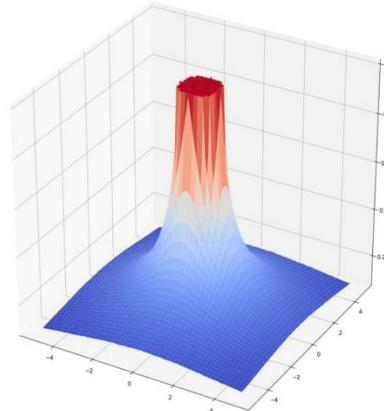
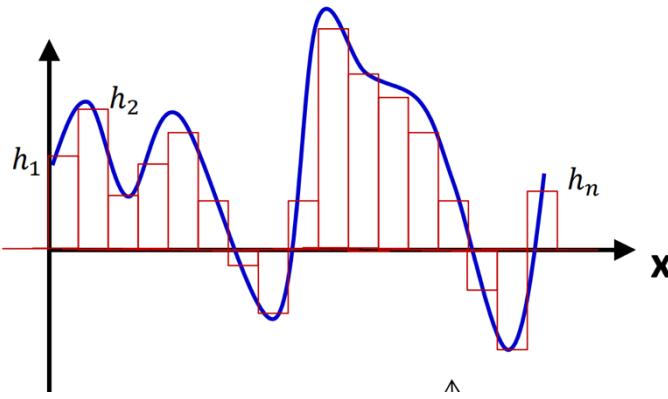
- Non-linearity **adds capacity** to the model to approximate **any** continuous function to **arbitrary** accuracy given **sufficiently many** hidden units.
 - See "universal approximation theorem"



Cannot separate red
and blue points with
linear classifier

Universal Approximation

- An MLP **can** represent **any function**, with **enough** expressivity.



Quiz Time

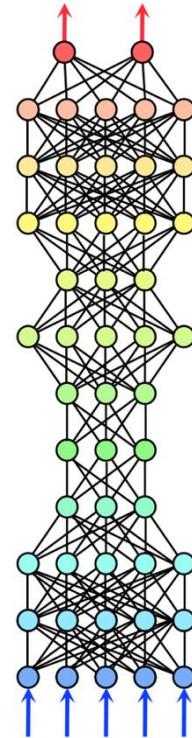
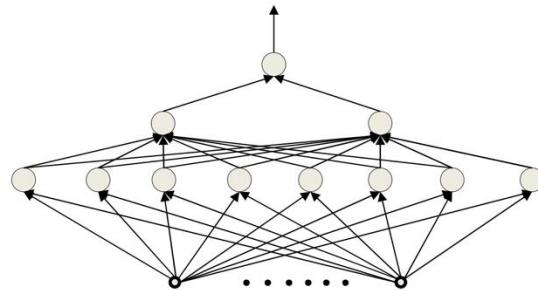
- What makes neural networks expressive functions?
 1. Activations (non-linearities)
 2. Depth (number of hidden layers)
 3. Width (number of variables in each hidden layer)
 4. All the above

Demo time!

- Link: <https://playground.tensorflow.org/>

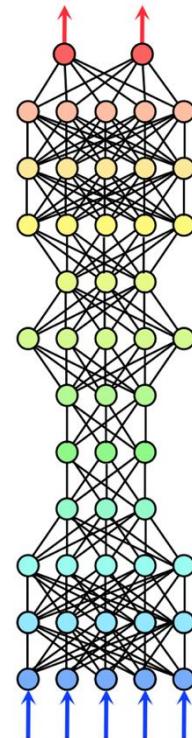
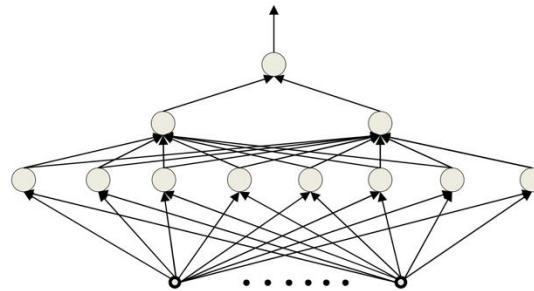
What is a good architecture? Depth vs. Width

- Architectural parameters of a neural network affect its capacity to learn.
 - Deep vs. wide



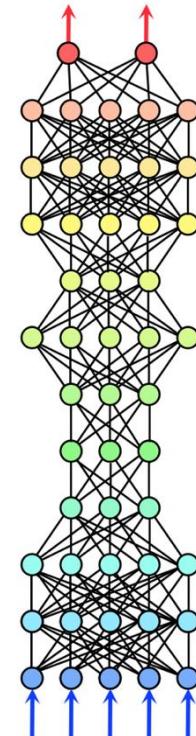
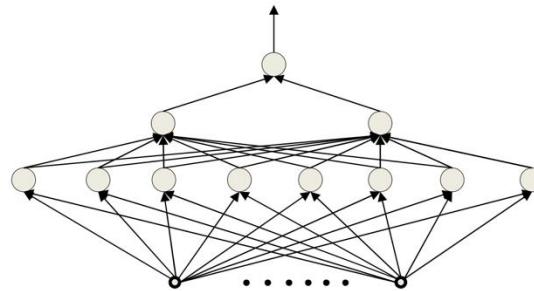
Depth vs Width on Boolean functions

- An MLP is a universal **Boolean** function.
- A **shallow** (single hidden layer) is a universal Boolean machine
 - But it may require an **exponentially large** number of units.
- **Deeper** networks may require far **fewer** neurons than shallower networks to express the same function



Depth vs Width on Boolean functions

- **Theorem:** There are certain class of functions with n inputs that can be represented with **deep** neural network with $O(n)$ units, whereas it would require $O(2^{\sqrt{n}})$ units for a **shallow** network.



Hastad, Almost optimal lower bounds for small depth circuits, 1986.
Delalleau & Bengio. Shallow vs. deep sum-product networks, 2011.

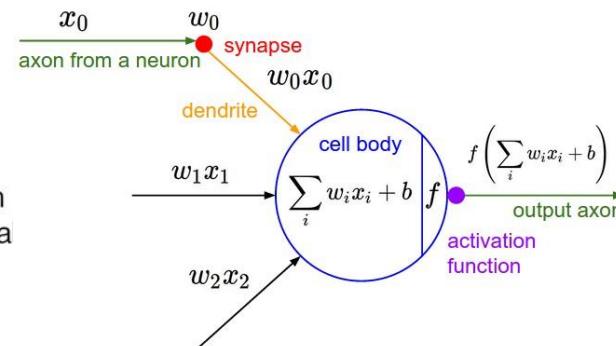
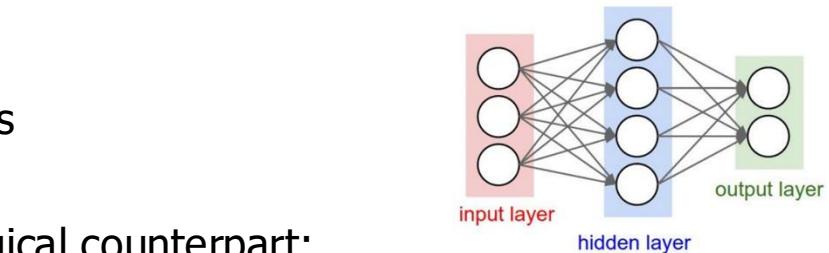
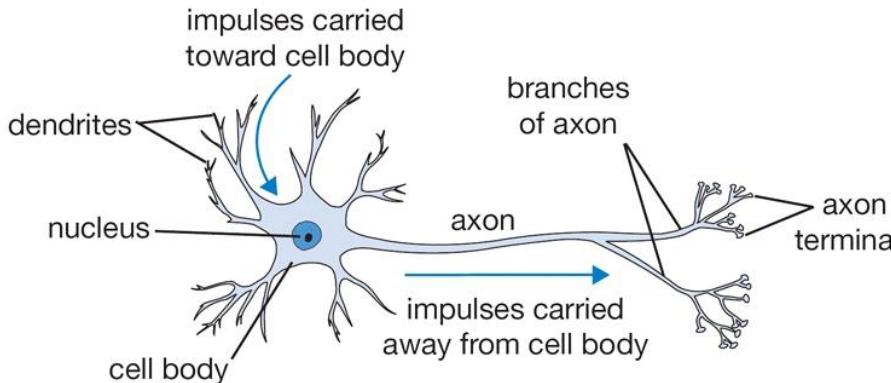
Summary

- An MLP is a universal function
- But can represent a given function only if
 - It is sufficiently wide
 - It is sufficiently deep
 - Depth can be traded off for (sometimes) exponential growth of the width of the network
- Optimal width and depth depend on the complexity of the problem.
- **Next:** A bit of history.

Neural Nets: Origin and History

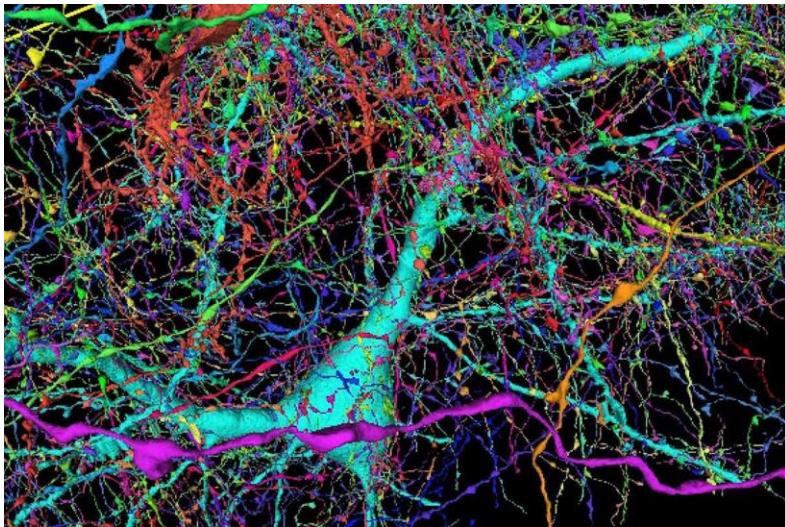
Artificial Neurons: An Inspiration from Nature

- A single node in your neural network
 - Accept information from multiple inputs
 - Transmit information to other neurons
- A neuron's function is inspired by its biological counterpart:
 - Apply some function on inputs signals
 - If output of function over threshold, neuron "fires"



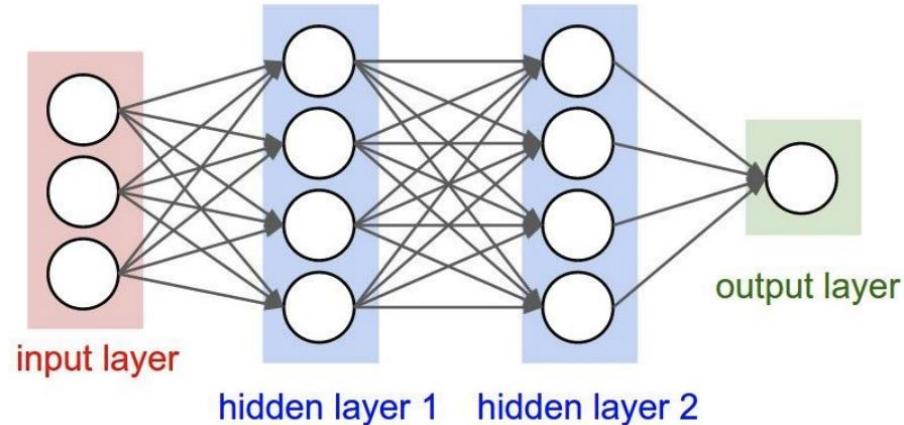
Artificial Neurons: Not Quite Analogous to Nature

Biological neurons:
complex connectivity



Source: Google Brain Map

Neurons in an artificial neural network:
organized based on a highly **regular structure** for computational efficiency



Very Brief History of Neural Networks

1. Single-layer neural networks (1943-1969)
2. Symbolic AI & knowledge engineering (1970-1985)
3. Multi-layer NNs and symbolic learning (1985-1995)
4. Shallow statistical learning/probabilistic models (1995-2010)
5. Deep networks and self-supervised learning (2010-?)

Very Brief History of Neural Networks

1. Single-layer neural networks (1943-1969)
2. Symbolic AI & knowledge engineering (1970-1985)
3. Multi-layer NNs and symbolic learning (1985-1995)
4. Shallow statistical learning/probabilistic models (1995-2010)
5. Deep networks and self-supervised learning (2010-?)

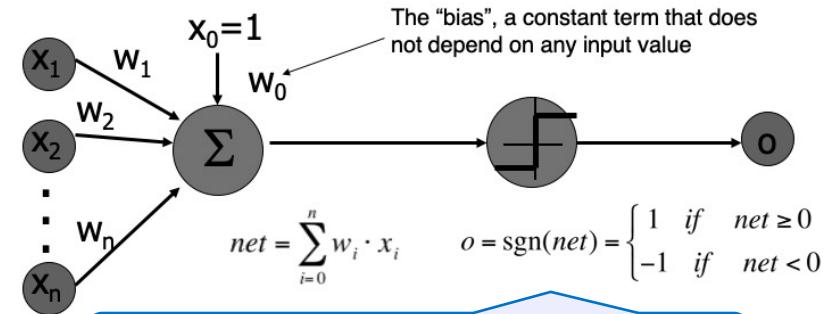
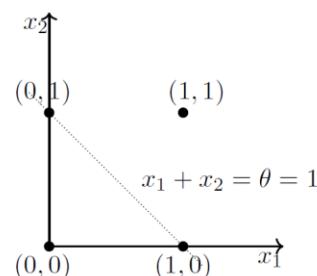
A Neuron as a Mathematical Model of Computation

- McCulloch and Pitts (1943) showed how linear threshold units can be used to compute logical functions



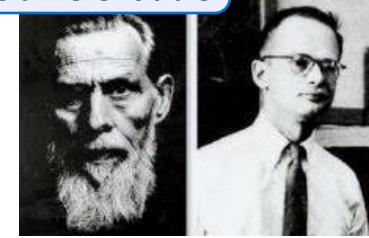
OR function

$$x_1 + x_2 = \sum_{i=1}^2 x_i \geq 1$$



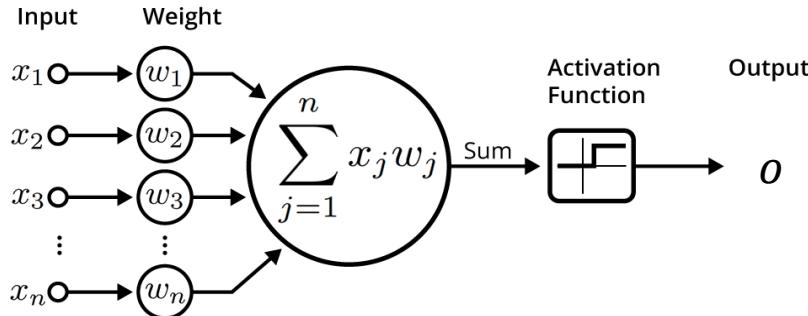
Notice the step function (threshold)!
Early models didn't need to be differentiable.

- An alternative model of computation (comparable to "Turing Machine")



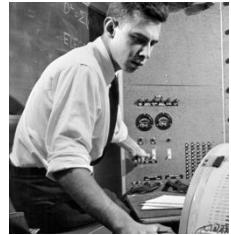
Perceptron Learning Rule – Imitating Nature's Learning Process

- Rosenblatt (1959) developed the **Perceptron** algorithm —
 - An iterative algorithm for learning the weights of a **linear threshold unit**.



- A single neuron with **a fixed input**, it can **incrementally change weights** and learn to produce **a fixed output** using the **Perceptron learning rule**.
- Update each weights by: $w_i = w_i + \eta(t - o)x_i$

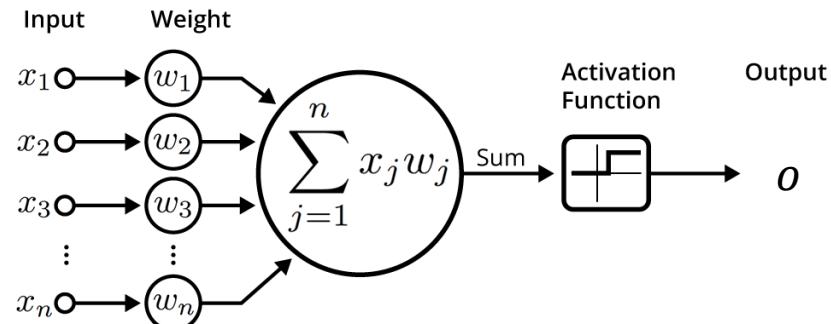
t : the target value



Quiz (1): Understanding Perceptron Update Rule

- Suppose the inputs $x_i \in \{0, 1\}$ and $\eta = 1$. If LTU's output o exactly matches the target value t , How would the update rule change the weights?
 - Would increase them
 - Would decrease them
 - Would not change them

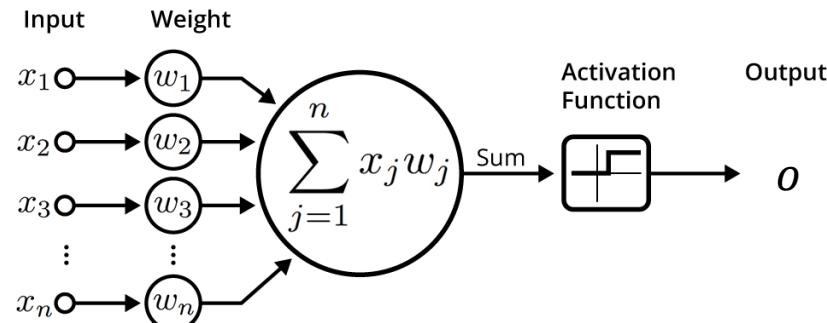
$$w_i = w_i + \eta(t - o)x_i$$



Quiz (2): Understanding Perceptron Update Rule

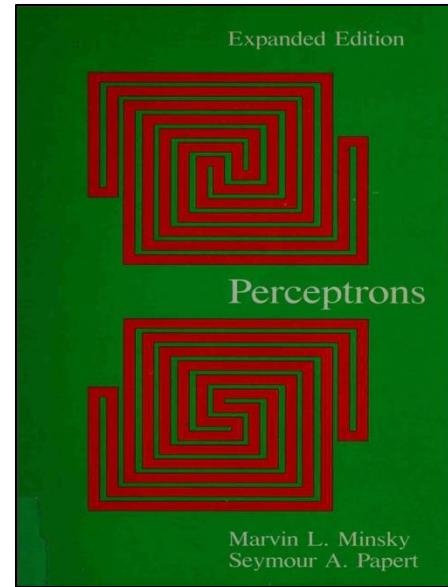
- Suppose the inputs $x_i \in \{0, 1\}$ and $\eta = 1$. If LTU's output o is **smaller** than the target value t , how would the update rule change the weights?
 1. Would increase them
 2. Would increase the weights for active inputs
 3. Would decrease them
 4. Would not change them
- After this update, the new output o would be:
 1. Larger
 2. Smaller
 3. Unchanged

$$w_i = w_i + \eta(t - o)x_i$$



Perceptron: Demise

- “Perceptrons” (1969) by Minsky and Papert illuminated few **limitations** of the perceptron.
- It showed that:
 - Shallow (2-layer) networks are **unable to learn or represent** many classification functions (e.g. XOR)
 - Only the **linearly separable** functions are learnable.
- Also, there was an understanding that deeper networks were infeasible to train.
- Result: research on NNs dissipated during the 70's and early 80's!



Very Brief History of Neural Networks

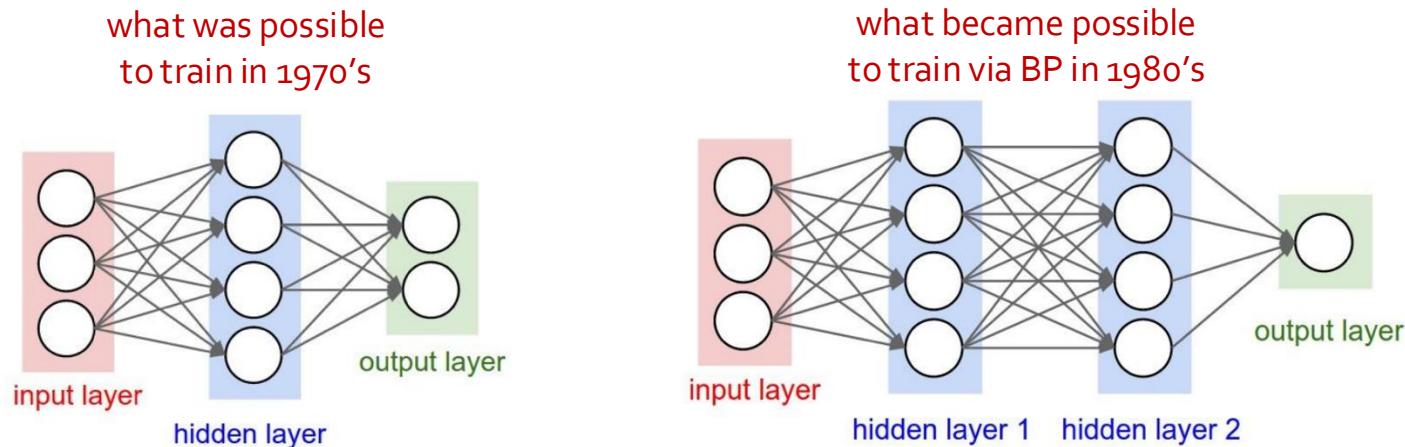
1. Single-layer neural networks (1943-1969)
2. Symbolic AI & knowledge engineering (1970-1985)
3. Multi-layer NNs and symbolic learning (1985-1995)
4. Shallow statistical learning/probabilistic models (1995-2010)
5. Deep networks and self-supervised learning (2010-?)

Very Brief History of Neural Networks

1. Single-layer neural networks (1943-1969)
2. Symbolic AI & knowledge engineering (1970-1985)
- 3. Multi-layer NNs and symbolic learning (1985-1995)**
4. Shallow statistical learning/probabilistic models (1995-2010)
5. Deep networks and self-supervised learning (2010-?)

Neural Networks Resurgence (1986)

- Interest in NNs revived in the mid 1980's due to the rise of "connectionism."
- Backpropagation algorithm** was [re-]introduced for training three-layer NN's.
 - Generalized the iterative "hill climbing" method to approximate networks with multiple layers, but no convergence guarantees.



Second NN Demise (1995-2010)

- Generic backpropagation did **not** generalize that well to training **deeper** networks.
 - Overfitting / underfitting remained an issue.
 - Computers were still quite slow
- Little theoretical justification for underlying methods.
- Machine learning research moved to graphical/probabilistic models and kernel methods.

Very Brief History of Neural Networks

1. Single-layer neural networks (1943-1969)
2. Symbolic AI & knowledge engineering (1970-1985)
3. Multi-layer NNs and symbolic learning (1985-1995)
4. Shallow statistical learning/probabilistic models (1995-2010)
5. Deep networks and self-supervised learning (2010-?)

Very Brief History of Neural Networks

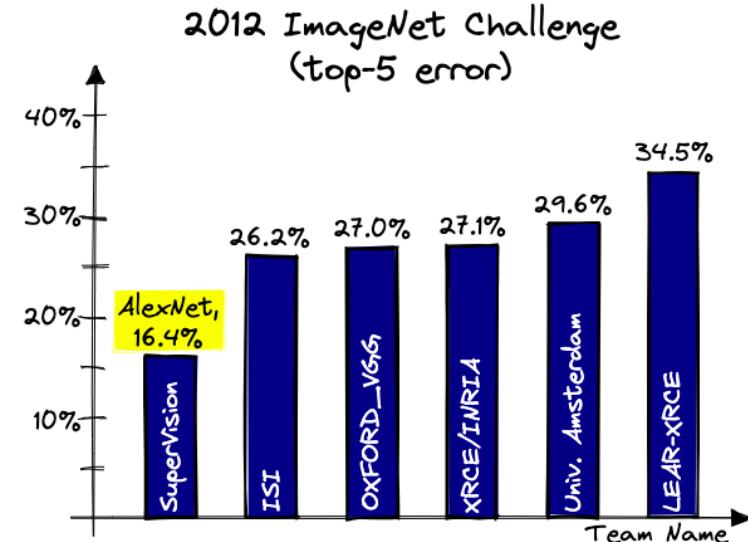
1. Single-layer neural networks (1943-1969)
2. Symbolic AI & knowledge engineering (1970-1985)
3. Multi-layer NNs and symbolic learning (1985-1995)
4. Shallow statistical learning/probabilistic models (1995-2010)
5. Deep networks and self-supervised learning (2010-?)

Very Brief History of Neural Networks

1. Single-layer neural networks (1943-1969)
2. Symbolic AI & knowledge engineering (1970-1985)
3. Multi-layer NNs and symbolic learning (1985-1995)
4. Shallow statistical learning/probabilistic models (1995-2010)
5. **Deep networks and self-supervised learning (2010-?)**

Deep Learning Revolution (2010...)

- Various successes with training deep neural works.
 - Convolutional neural nets (CNNs) for vision — 2012 AlexNet showed 16% error reduction on ImageNet benchmark.
 - Rise of deep reinforcement learning for games—AlphaGo beat human players.



Deep Learning Revolution (2010...)

- Various successes with training deep neural works.
 - Convolutional neural nets (CNNs) for vision — 2012 AlexNet showed 16% error reduction on ImageNet benchmark.
 - Rise of deep reinforcement learning for games—AlphaGo beat human players.

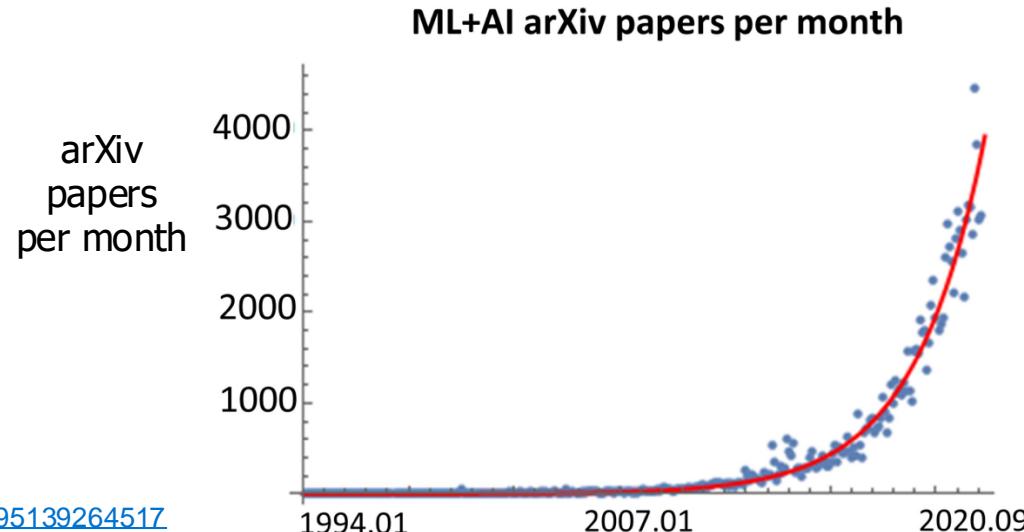
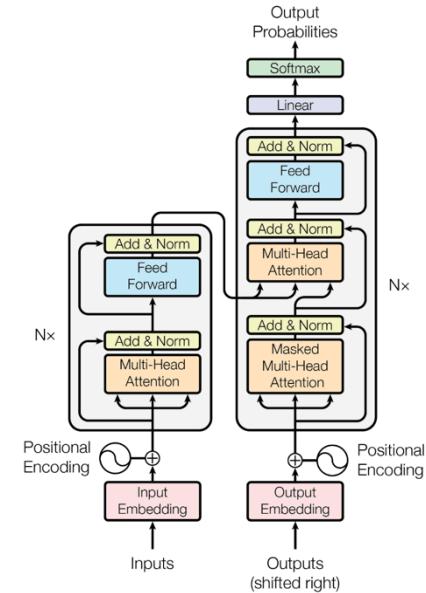
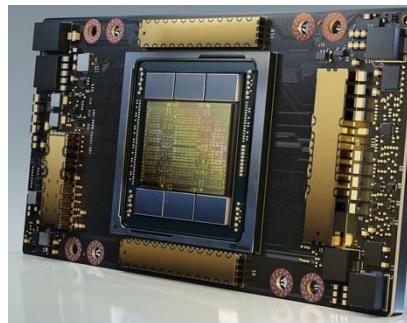
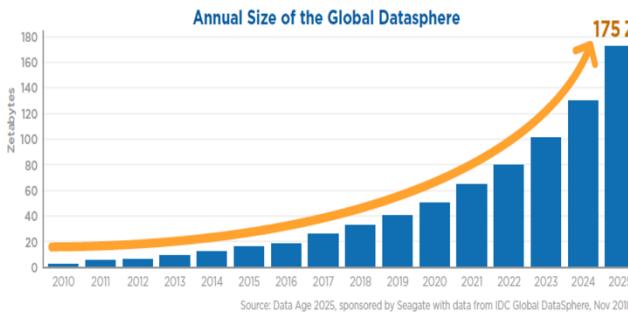


Figure credit:

<https://twitter.com/MarioKrenn6240/status/1314622995139264517>

Deep Learning Revolution (2010...)

- The success continued enabled by 3 forces:
 - Availability of massive [unlabeled] data — the data on Internet.
 - Faster computing technologies — specialized hardware (e.g., GPUs)
 - Algorithmic innovations — architectures, optimization, etc.



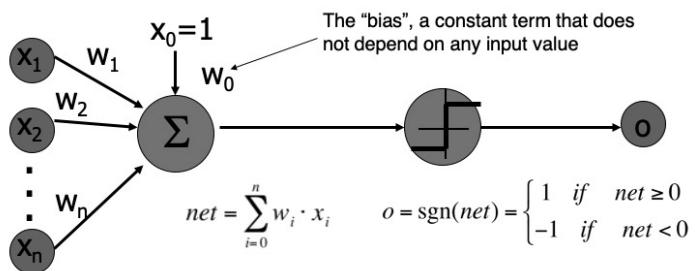
Very Brief History of Neural Networks

1. Single-layer neural networks (1943-1969)
2. Symbolic AI & knowledge engineering (1970-1985)
3. Multi-layer NNs and symbolic learning (1985-1995)
4. Shallow statistical learning/probabilistic models (1995-2010)
5. **Deep networks and self-supervised learning (2010-?)**

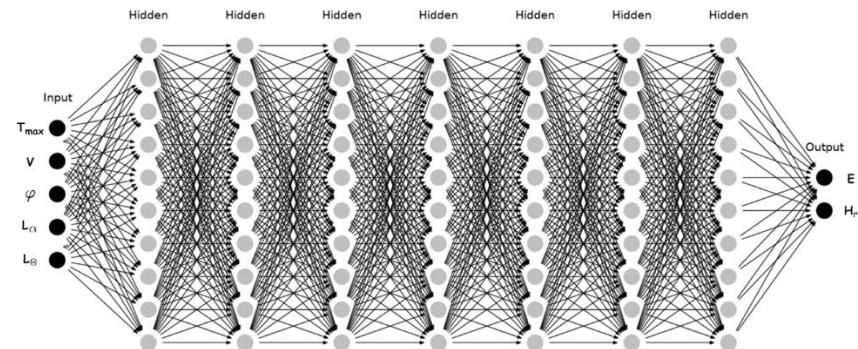


https://en.wikipedia.org/wiki/Big_Bounce

How it started



How it's going



Summary

- Neural networks have been long in the making since 1950s.
- It's a remarkable journey of science with many ups and downs.
- **Next:** How do you train NNs? We will start with some algebra refreshers.

Background for Training NNs

The Refreshers



Machine Learning Problems

- **Training data:** Given a set of inputs and output labels:
 - Inputs: $X = (x_1, \dots, x_n)$
 - Outputs: $Y = (y_1, \dots, y_n)$
- **Goal:** Find a function $f(x; \theta)$ with parameters θ that maps inputs in X to output to Y
- **Empirical risk:** measure the quality of the predictions with a loss function:

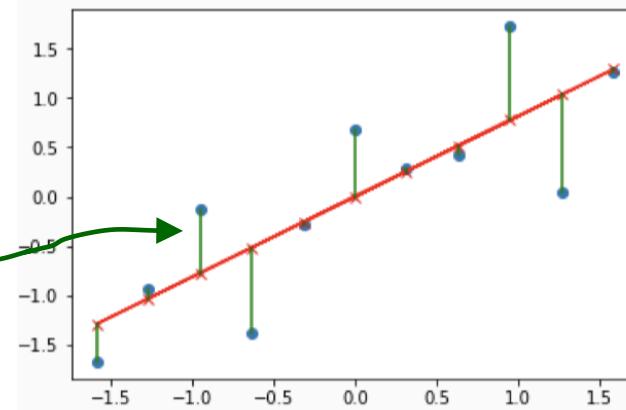
$$J(\theta) = \frac{1}{n} \sum_{i=1}^n \ell(f(x_i; \theta), y_i)$$

A Special Case: Linear Regression

- **Training data:** Given a set of inputs and output labels:
 - Inputs: $X = (x_1, \dots, x_n)$
 - Outputs: $Y = (y_1, \dots, y_n)$
- **Goal:** Find a linear function $f(x; \theta) = \theta \cdot x$ that is best predictive of observations
- **Empirical risk:** measure the quality of the predictions with a loss function:

$$J(\theta) = \frac{1}{n} \sum_{i=1}^n \ell(\theta \cdot x_i, y_i)$$

What are good choices
for loss function?



Quiz: Loss functions

- Remember the objective function of our learning problem:

$$J(\theta) = \frac{1}{n} \sum_{i=1}^n \ell(f(x_i; \theta), y_i)$$

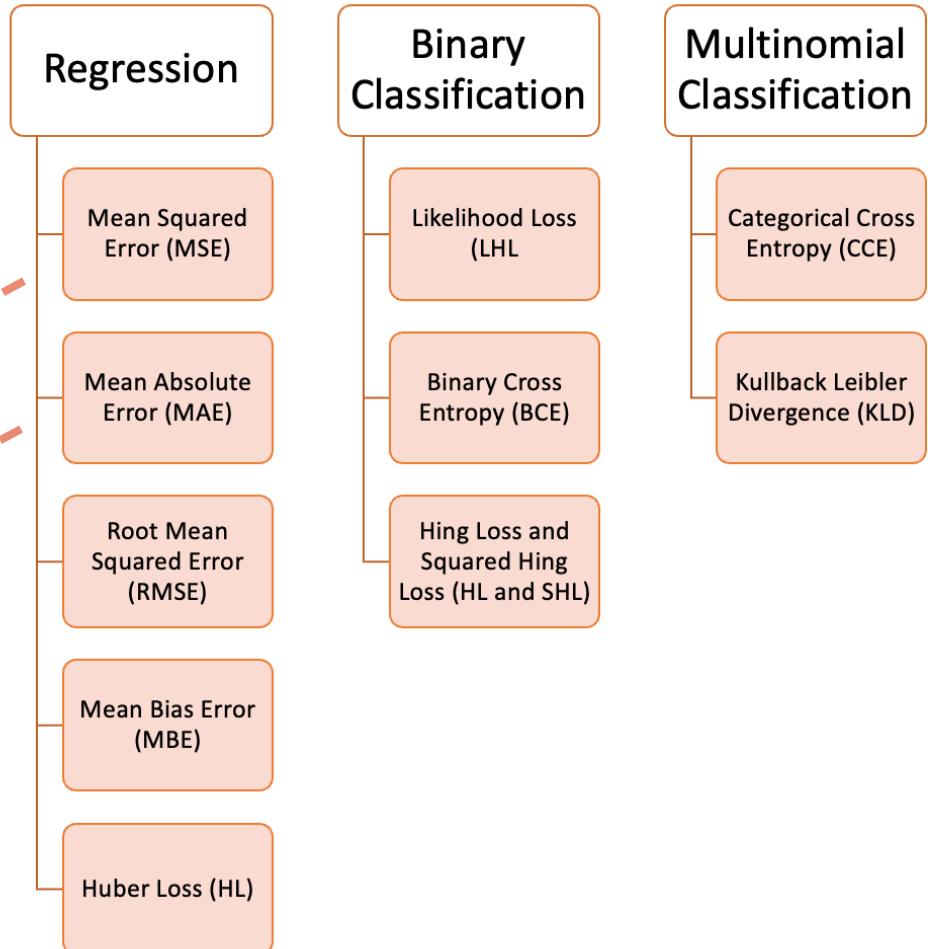
- Which of the followings is a more reasonable loss function $\ell(z, w)$?
 - If z and w are **far apart**, the loss value should be **higher**
 - If z and w are **far apart**, the loss value should be **lower**
 - Neither

Loss Functions

- The choice of loss function depends on the problem

$$\ell(y, \hat{y}) = (y - \hat{y})^2$$

$$\ell(y, \hat{y}) = |y - \hat{y}|$$

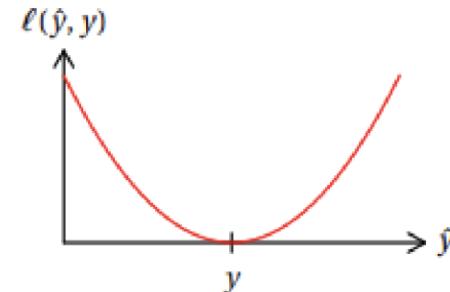
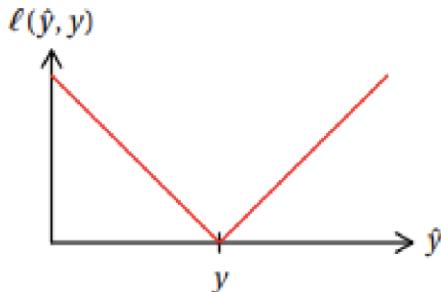


Quiz: MSE vs. MAE loss

- Remember MSE and MAE loss:

$$\text{MSE: } \ell(y, \hat{y}) = (y - \hat{y})^2$$
$$\text{MAE: } \ell(y, \hat{y}) = |y - \hat{y}|$$

- Which visualization corresponds to which loss?



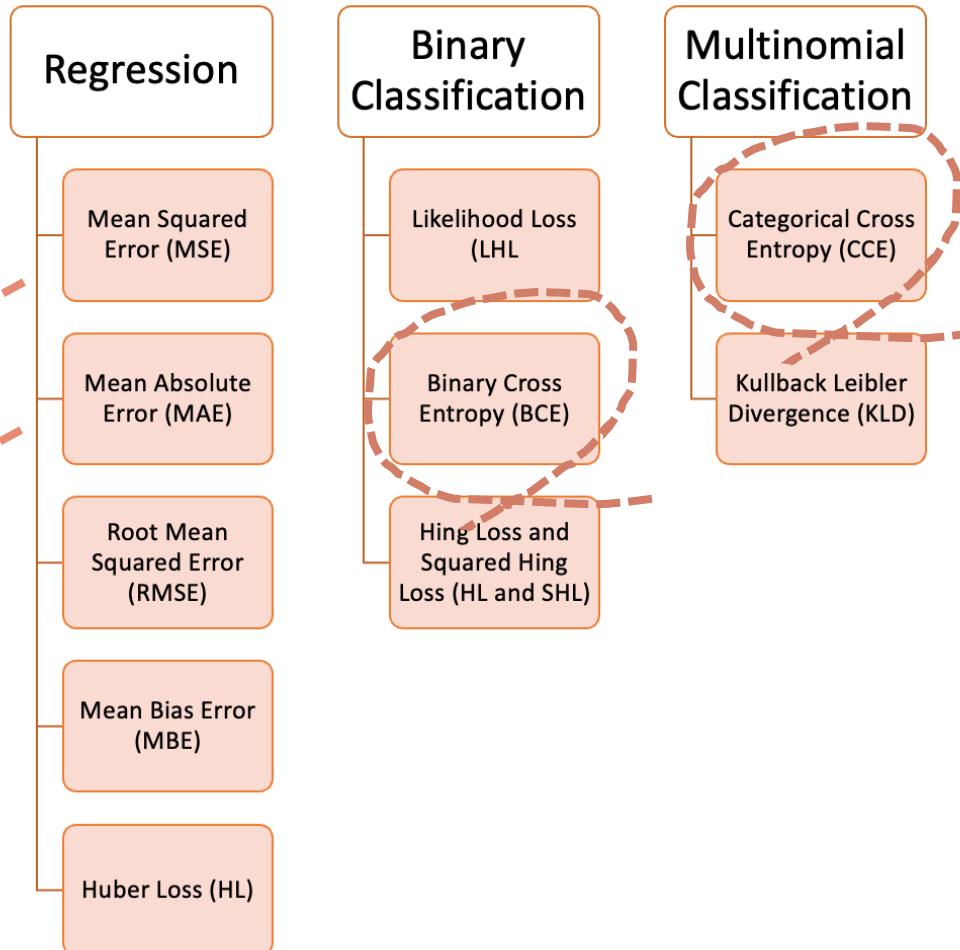
- Which loss is more sensitive to outlier data (noisy outputs)?
- Which loss is more difficult to compute gradients for?

Loss Functions

- The choice of loss function depends on the problem

$$\ell(y, \hat{y}) = (y - \hat{y})^2$$

$$\ell(y, \hat{y}) = |y - \hat{y}|$$



Loss Functions: Cross-Entropy

- A binary classification example: Without loss of generality:
 - Gold labels: $y = [1, 0]$ (i.e., first class is correct)
 - Predictions: $\hat{y} = [p, 1 - p]$
- CE loss: $\ell(y, \hat{y}) = -1 \times \log p - 0 \times \log(1 - p) = -\log p$

$$\ell(\mathbf{y}, \hat{\mathbf{y}}) = - \sum_j^n y_j \log(\hat{y}_j)$$

Summation over the dimensions of \mathbf{y}

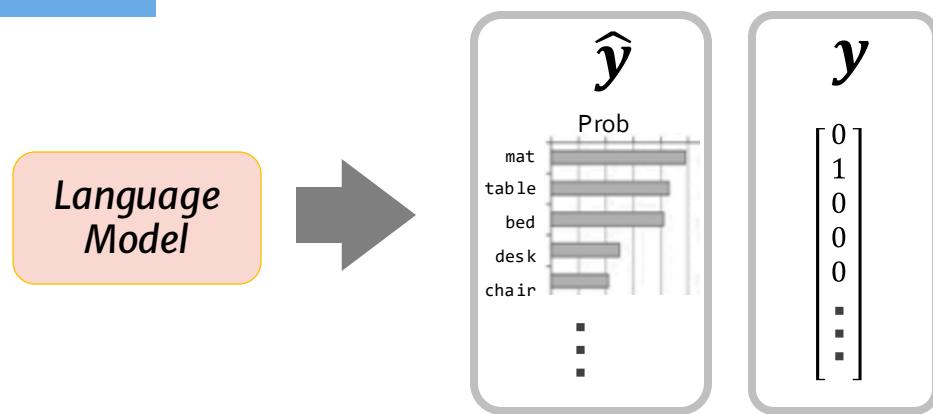
Quiz: CE for Binary Classification

- A binary classification example: Without loss of generality:
 - Gold labels: $y = [1, 0]$ (i.e., first class is correct)
 - Predictions: $\hat{y} = [p, 1 - p]$
- CE loss: $\ell(y, \hat{y}) = -1 \times \log p - 0 \times \log(1 - p) = -\log p$
- **Question 1:** Why the negative sign?
- **Question 2:** If the model prediction is miraculously accurate ($p = 1$), what is CE loss?
- **Question 3:** If the model prediction is embarrassingly off ($p = 0$), what is the CE loss?

$$\ell(y, \hat{y}) = -\sum_j^n y_j \log(\hat{y}_j)$$

Summation over the dimensions of y

Cross-Entropy Over Next-Word Distribution



$$\ell(\mathbf{y}, \hat{\mathbf{y}}) = - \sum_j^n y_j \log(\hat{y}_j)$$

Summation over the dimensions of \mathbf{y}

- Suppose for a fixed j , $y_j = 1$.
- **Question:** Then CE is the probability is?
 - It is the probability that our language model assigns to the j th term.

Machine Learning Problems

- **Training data:** Given a set of inputs and output labels:
 - Inputs: $X = (x_1, \dots, x_n)$
 - Outputs: $Y = (y_1, \dots, y_n)$
- **Goal:** Find a function $f(x; \theta)$ with parameters θ that maps inputs in X to output to Y
- **Empirical risk:** measure the quality of the predictions with a loss function:

$$J(\theta) = \frac{1}{n} \sum_{i=1}^n \ell(f(x_i; \theta), y_i)$$

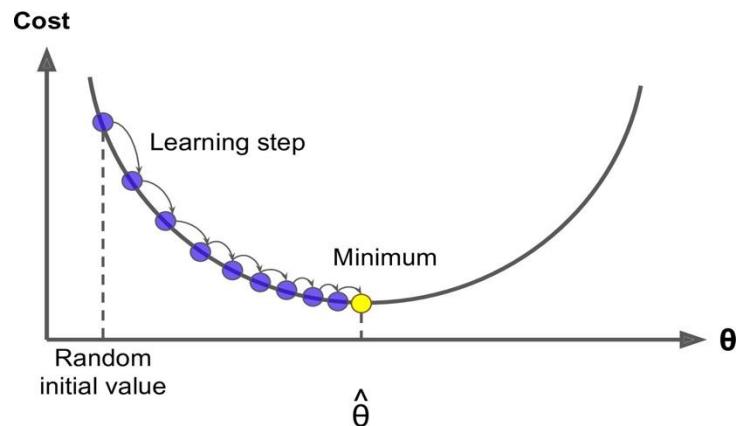
- Machine learning as optimization:

$$\operatorname{argmin}_{\theta} J(\theta)$$

How do you solve this optimization?

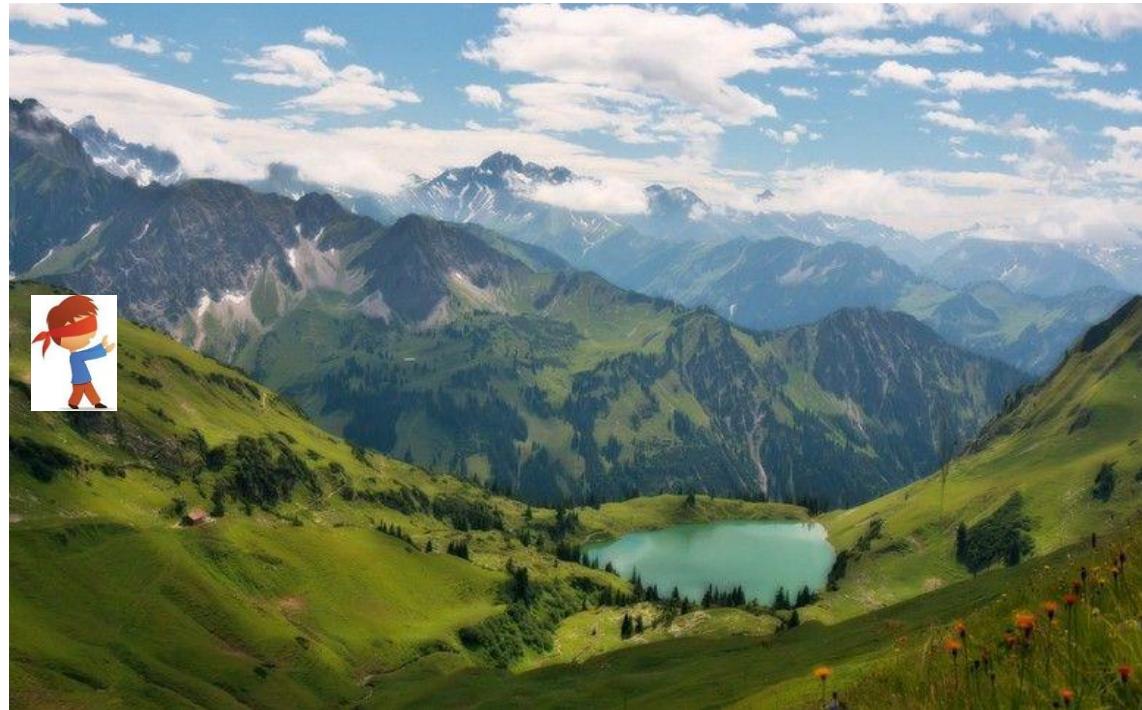
Gradient Descent

- We have a cost function $J(\theta)$ we want to minimize
 - We can use Gradient Descent algorithm!
- **Idea:** for current value of θ , calculate gradient of $J(\theta)$, then take **small step** in direction of negative gradient. Repeat.
- Note: Our objectives may not be convex like this. But life turns out to be okay!



Gradient Descent (1): Intuition

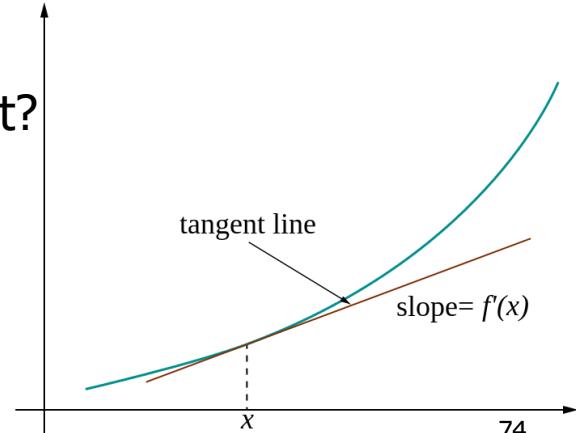
- Imagine you're blindfolded
- Need to walk down a hill
- You can use your hands to find the directions that may be downhill



[slide: Andrej Karpathy]

Gradient Descent (2): Intuition

- In 1-dimension, the **derivative** of a function: $\frac{\partial J}{\partial \theta_j} = \lim_{h \rightarrow 0} \frac{J(\theta_j + h) - J(\theta_j)}{h}$
- Gradient** is the vector of derivatives: $\nabla_{\theta} J = \left[\frac{\partial J}{\partial \theta_1}, \dots, \frac{\partial J}{\partial \theta_n} \right]$
- Gradient descent involves iterative steps in direction of **negative gradient**.
- Question:** Why step in direction of negative gradient?
 - Gradient quantifies how rapidly the function $L(\theta)$ varies when we change the argument θ_j by a tiny amount.



Gradient Descent (3)

- Iteratively subtract the gradient with respect to the model parameters (θ)

$$\theta^{new} \leftarrow \theta^{old} - \alpha \nabla_{\theta} J(\theta^{old})$$

α = Step size or learning rate

- We're moving in a direction opposite to the gradient of the loss $J(\theta)$
- I.e., we're moving towards smaller loss $J(\theta)$

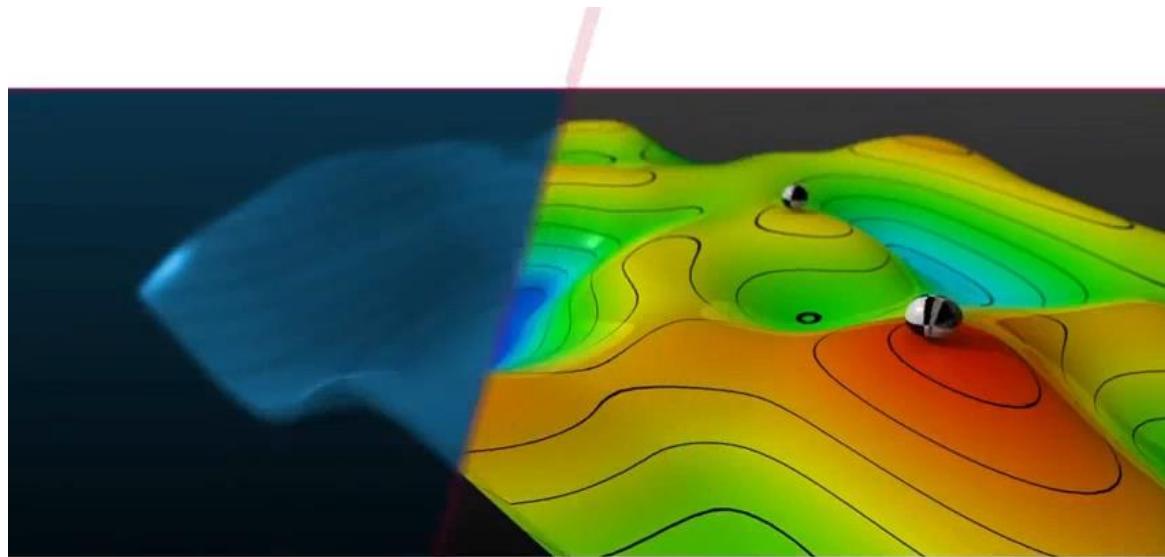
- In PyTorch:

```
while True:  
    theta_grad = evaluate_gradient(J,corpus,theta)  
    theta = theta - alpha * theta_grad
```

Gradient Descent (4)

- Update equation (in matrix notation):

$$\theta^{new} = \theta^{old} - \alpha \nabla_{\theta} J(\theta)$$



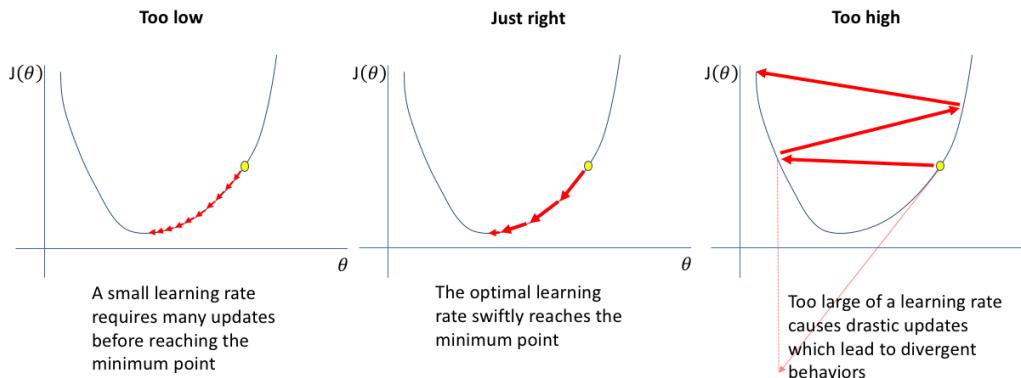
clideo.com

[demo credit: ICMS YouTube channel]

Gradient Descent: Setting the Step Size

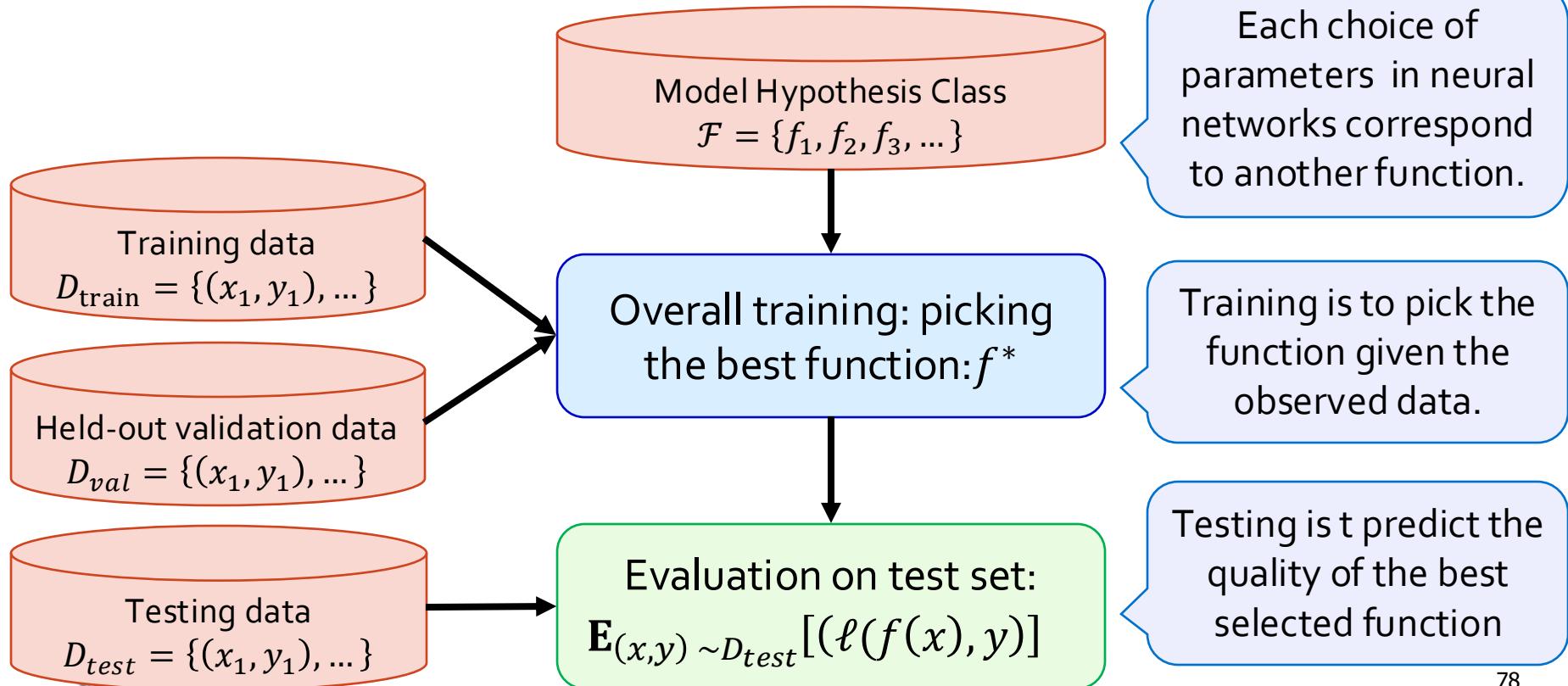
- What is a good value for step size α ?

- What is a good value for step size α ?
 - If α = too small, it may be too slow
 - If α = too large, it may oscillate
- It may take trial-and-errors to find the sweet spot.
 - Another trick is to define a “schedule” for gradually reducing the learning rate starting from a large number.



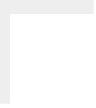
- It may take trial-and-errors to find the sweet spot.
- Another trick is to define a “schedule” for gradually reducing the learning rate starting from a large number.

A Typical Learning and Evaluation Protocol



Summary Thus Far

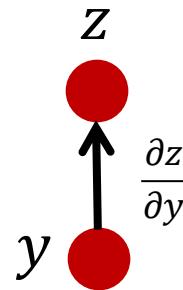
- A statistical learning problem can be formulated as an **optimization** problem.
- The objective of this optimization consists of:
 - Learning data (input/outputs)
 - Predictive model architecture (encoding how an input gets mapped to an output)
 - Loss function (quantifying quality of predictions)
- Soon, we will see how to use Neural Nets as the predictive model.



Algebra Refresher (Needed for Backpropagation)

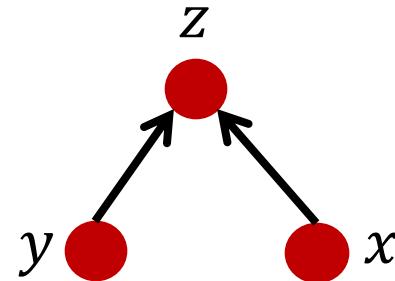
Derivatives

- First let's get the notation right:
- The **arrow** shows **functional dependence** of z on y , i.e. given y , we can calculate z .
 - For example: $z(y) = 2y^2$
- The derivative of z , with respect to y : $\frac{\partial z}{\partial y}$



Quiz time!

- If $z(x, y) = y^4x^5$ what is the following derivative $\frac{\partial z}{\partial y}$?
 - $\frac{\partial z}{\partial y} = 4y^3x^5$
 - $\frac{\partial z}{\partial y} = 5y^4x^4$
 - $\frac{\partial z}{\partial y} = 20y^3x^4$
 - None of the above



Gradient

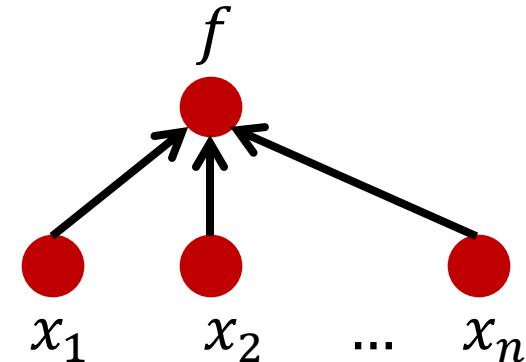
- Given a function with 1 output and n inputs

$$f(\mathbf{x}) = f(x_1, x_2, \dots, x_n) \in \mathbb{R}$$

- Its gradient is a vector of partial derivatives with respect to each input

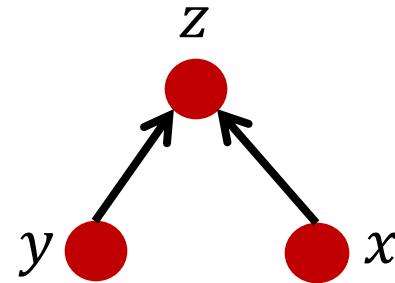
$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} \in \mathbb{R}^n$$

(always assume vectors are
column vectors, i.e., they're in $\mathbb{R}^{n \times 1}$)



Quiz time!

- If $z(x, y) = y^4x^5$ what is the following gradient ∇z ?
 - $\nabla z(x, y) = 4y^3x^5$
 - $\nabla z(x, y) = (5y^4x^4, 20y^3x^4)$
 - $\nabla z(x, y) = (5y^4x^4, 4y^3x^5)$
 - None of the above



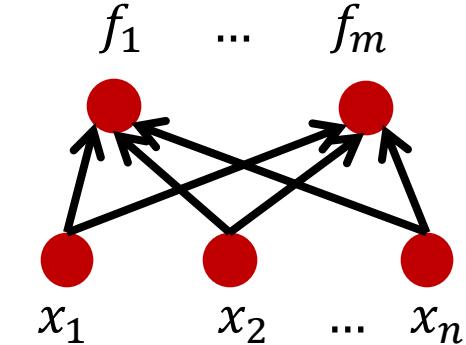
Jacobian Matrix: Generalization of the Gradient

- Given a function with **m outputs** and **n inputs**

$$\mathbf{f}(\mathbf{x}) = [f_1(x_1, x_2, \dots, x_n), \dots, f_m(x_1, x_2, \dots, x_n)] \in \mathbb{R}^m$$

- It's Jacobian is an **$m \times n$ matrix** of partial derivatives: $(\mathbf{J}_f(\mathbf{x}))_{ij} = \frac{\partial f_i}{\partial x_j}$

$$\mathbf{J}_f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{m \times n}$$



Quiz: Jacobian's special case (1)

- Remember Jacobians:

$$\mathbf{f}(\mathbf{x}) = [f_1(x_1, x_2, \dots, x_n), \dots, f_m(x_1, x_2, \dots, x_n)] \in \mathbb{R}^m$$

$$\mathbf{J}_{\mathbf{f}}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{m \times n} \quad \text{or} \quad (\mathbf{J}_{\mathbf{f}}(\mathbf{x}))_{ij} = \frac{\partial f_i}{\partial x_j}$$

- When $m=1$ (scalar-valued function), Jacobian reduces to ...?

$$\nabla^T \mathbf{f}(\mathbf{x}) \text{ (gradient transpose)}$$

Quiz: Jacobian's special case (2)

- Remember Jacobians:

$$\mathbf{f}(\mathbf{x}) = [f_1(x_1, x_2, \dots, x_n), \dots, f_m(x_1, x_2, \dots, x_n)] \in \mathbb{R}^m$$

$$\mathbf{J}_{\mathbf{f}}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{m \times n} \quad \text{or} \quad (\mathbf{J}_{\mathbf{f}}(\mathbf{x}))_{ij} = \frac{\partial f_i}{\partial x_j}$$

- When $m=n=1$ (single-variable function), Jacobian reduces to ...?

the derivative of \mathbf{f}

Jacobian for Matrix Inputs

- Given a function with m **outputs** and $n \times p$ **inputs**

$$\mathbf{f}(\mathbf{X}) = [f_1(\mathbf{X}), \dots, f_m(\mathbf{X})] \in \mathbb{R}^m, \text{ where } \mathbf{X} = \begin{bmatrix} x_{11} & \cdots & x_{1p} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{np} \end{bmatrix} \in \mathbb{R}^{n \times p}$$

- Jacobian is a $m \times n \times p$ **tensor** (i.e., matrix of matrices) of partial derivatives:

$$(\mathbf{J}_{\mathbf{f}}(\mathbf{X}))_{ijk} = \frac{\partial f_i}{\partial x_{jk}}$$

- The dimension of the Jacobian:= output dim x input dims.
- The Jacobian math holds if you keep adding **more dimensions** to the input or output.

Why Use Matrix/Tensor Form?

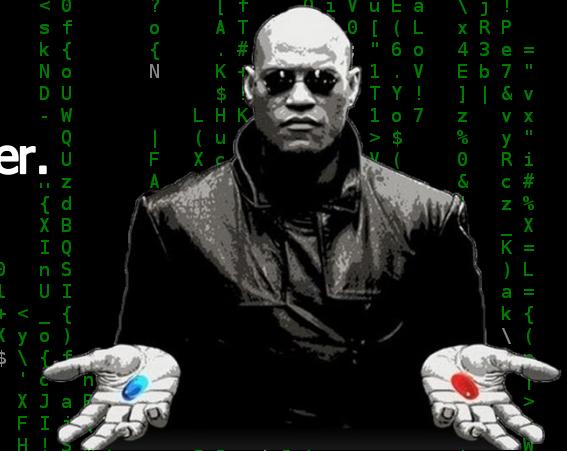
In essence, matrix form (multi-variate calculus) is just an extension of single-variable calculus.

Two reasons:

- Compact derivations: with matrix form calculations we can compute a concise statements.

○ Implementing algorithms in matrix form is much faster.

- GPUs are optimized for VERY FAST matrix/tensor operations.

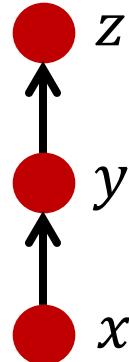


Chain Rule

- Function composition:

$$z \circ y(x) = z(y(x)) = z(x)$$

If z is a function of y , and
 y is a function of x , then
 z is a function of x , as well.



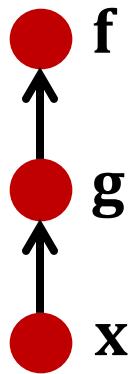
Then:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$$

Chain Rule for Multivariable Functions

- Let $\mathbf{x} \in \mathbb{R}^d$, $\mathbf{g}: \mathbb{R}^d \rightarrow \mathbb{R}^n$, $\mathbf{f}: \mathbb{R}^n \rightarrow \mathbb{R}^m$
- Composing them: $\mathbf{f} \circ \mathbf{g}(\mathbf{x}) = \mathbf{f}(\mathbf{g}(\mathbf{x})): \mathbb{R}^d \rightarrow \mathbb{R}^m$

The result looks similar to the single-variable setup:



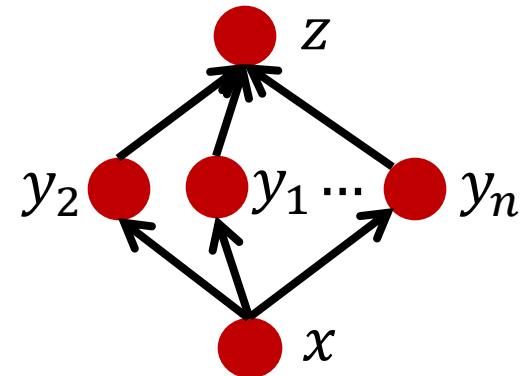
$$\mathbf{J}_{\mathbf{f} \circ \mathbf{g}}(\mathbf{x}) = \mathbf{J}_{\mathbf{f}}(\mathbf{g}(\mathbf{x})) \mathbf{J}_{\mathbf{g}}(\mathbf{x})$$

Note, the above statement is a **matrix** multiplication!

Function $\mathbf{f} \circ \mathbf{g}$ has m outputs and d inputs \rightarrow Jacobian's dims: m by d

Quiz Time!

Let $x \in \mathbb{R}$, $\mathbf{y}: \mathbb{R} \rightarrow \mathbb{R}^n$, $\mathbf{z}: \mathbb{R}^n \rightarrow \mathbb{R}$



What is the Jacobean of $z \circ \mathbf{y}(x) = z(y_1(x), \dots, y_n(x))$?

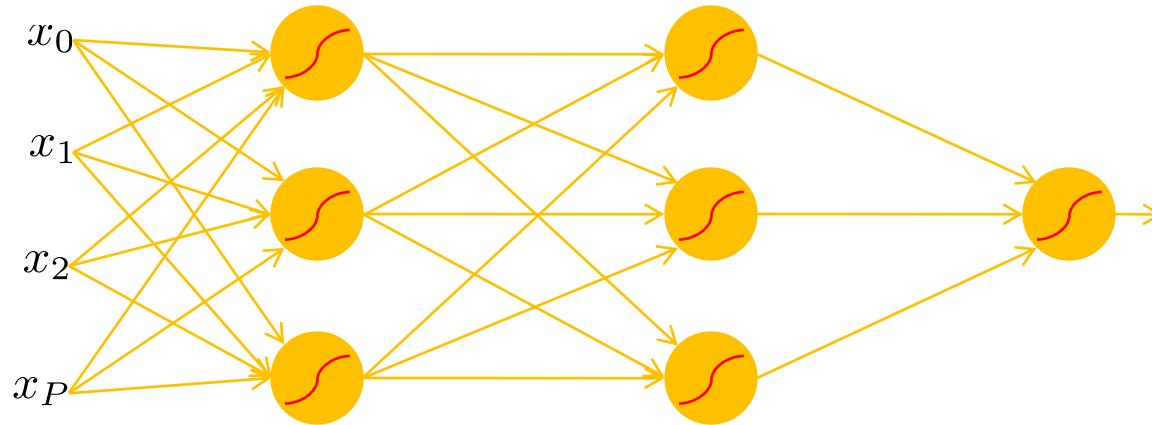
1. $J_{z \circ \mathbf{y}}(x) = J_z(\mathbf{y}(x)) J_y(x)$
2. $J_{z \circ \mathbf{y}}(x) = \left[\frac{\partial z}{\partial y_1}, \dots, \frac{\partial z}{\partial y_n} \right] \left[\frac{\partial y_1}{\partial x}, \dots, \frac{\partial y_n}{\partial x} \right]^T$
3. $J_{z \circ \mathbf{y}}(x) = \sum_{i=1}^n \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$
4. All the above!

Summary

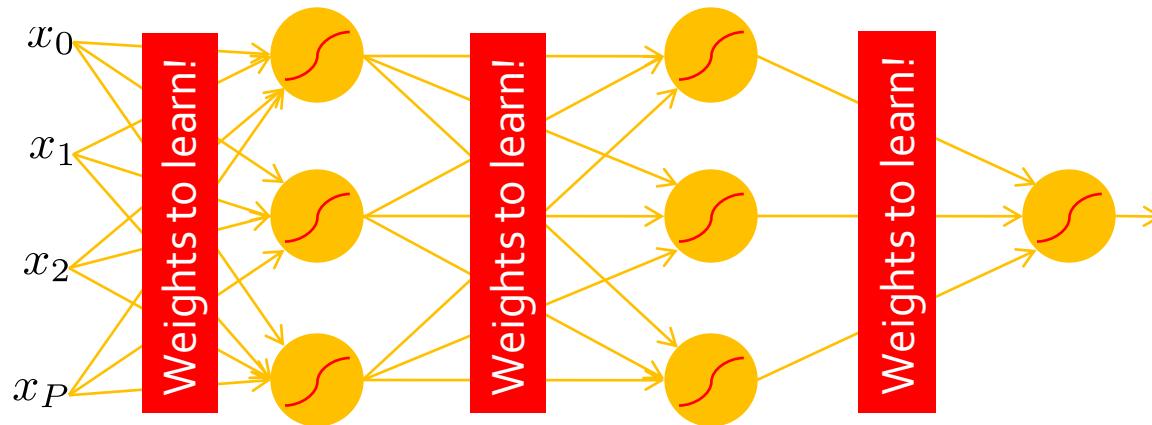
- We reviewed lots of background about neural networks!
 - Linear algebra foundation
 - Gradient descent
 - Extending gradients to tensor form: Jacobians
- **Next:** training a neural net!

Training Neural Networks: Analytical Backprop

Recap: Multi-Layer Perceptron

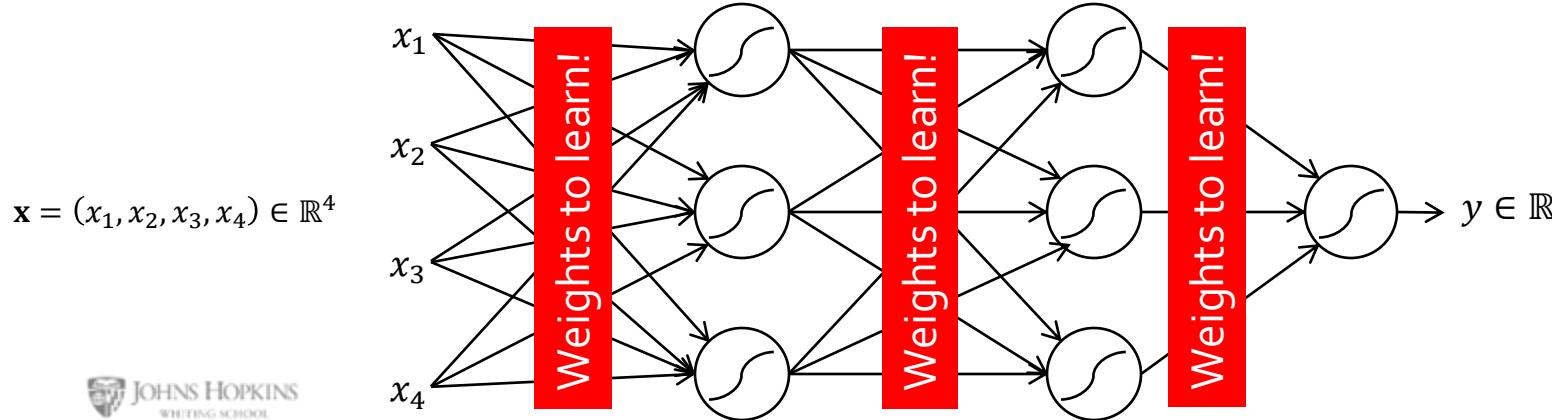


Recap: Multi-Layer Perceptron



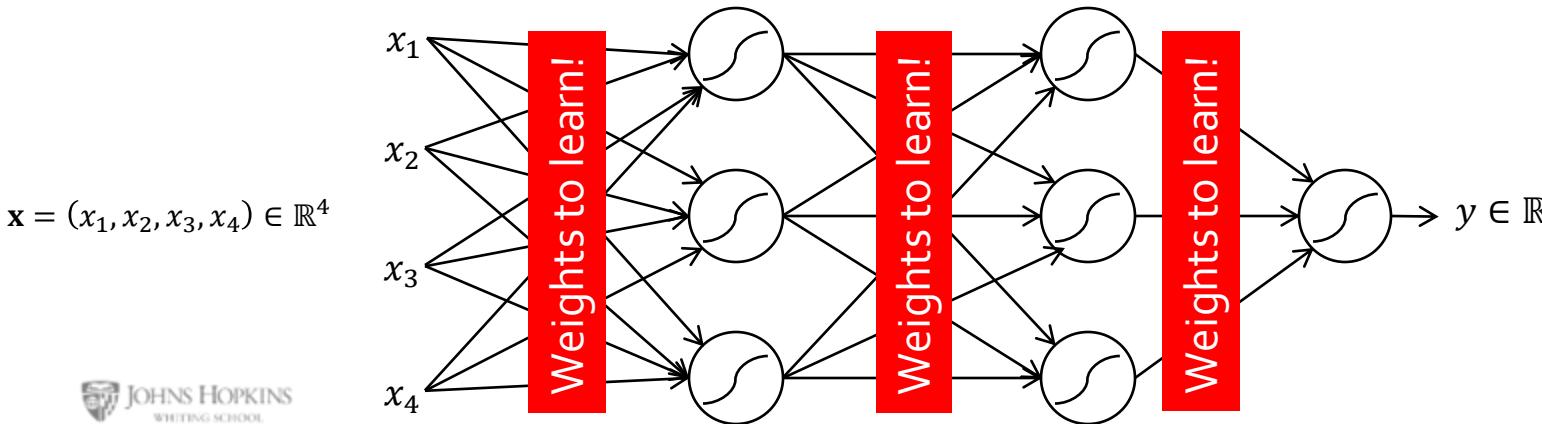
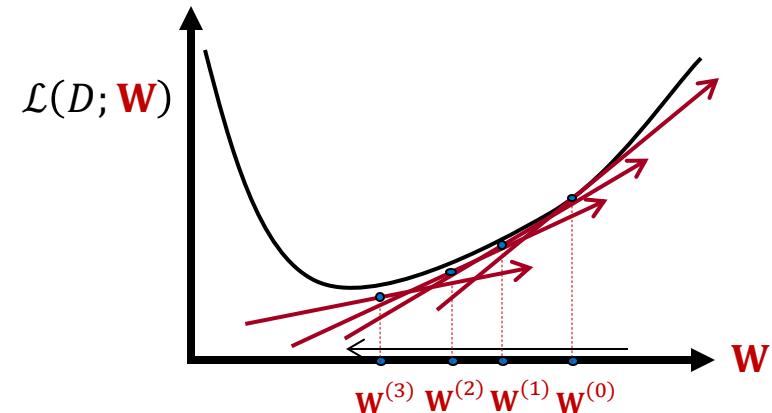
Training Neural Networks: Setup

- We are given an architecture though its weights \mathbf{W} .
- We are given a training data $D = \{(\mathbf{x}_i, y_i^*)\}$
- We are given a loss function $\ell: \mathbb{R} \times \mathbb{R} \rightarrow (0, 1)$
 - $\ell(y^*, y)$ quantifies distance between an answer y^* and prediction $y = \text{NN}(\mathbf{x}; \mathbf{W})$ — lower is better.
- Overall objective to optimize: $\mathcal{L}(D; \mathbf{W}) = \sum_{(\mathbf{x}_i, y_i^*) \in D} \ell(\text{NN}(\mathbf{x}_i; \mathbf{W}), y_i^*)$



Training Neural Networks ~ Optimizing Parameters

- We can use **gradient descent** to minimizes the loss.
- At each step, the **weight vector** is modified in the **direction that produces the steepest descent** along the error surface.

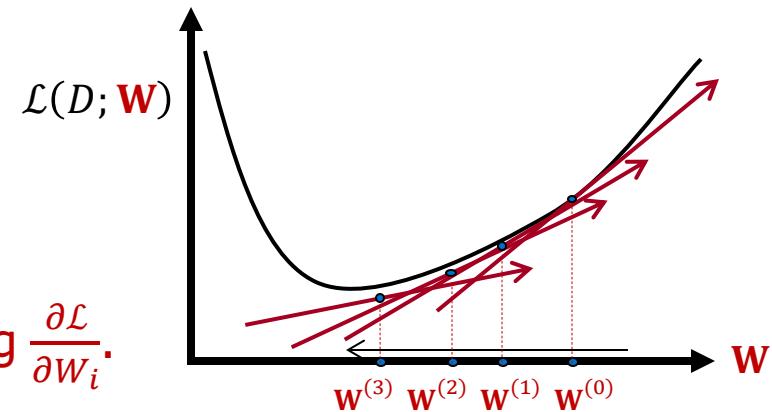


Training Neural Networks ~ Optimizing Parameters

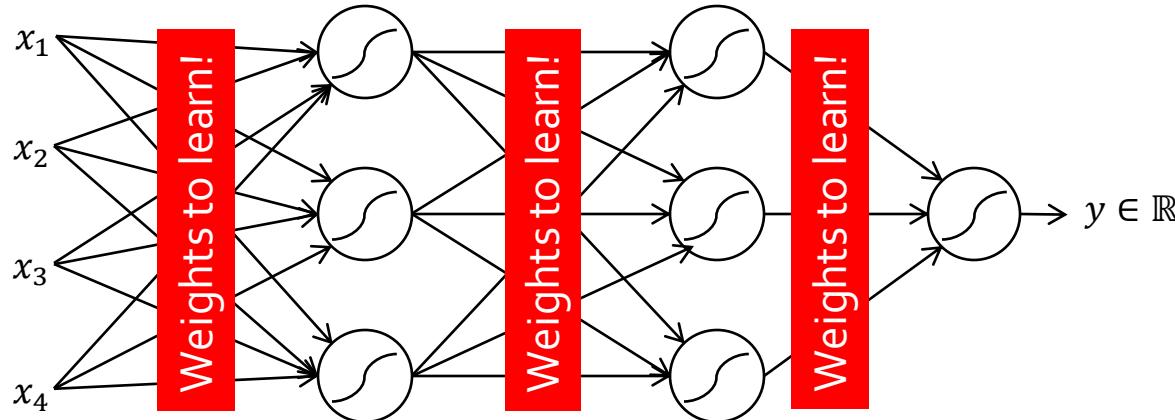
For each sub-parameter $W_i \in \mathbf{W}$:

$$W_i^{(t+1)} = W_i^{(t)} - \alpha \frac{\partial \mathcal{L}}{\partial W_i}$$

It all comes down to **efficiently** computing $\frac{\partial \mathcal{L}}{\partial W_i}$.

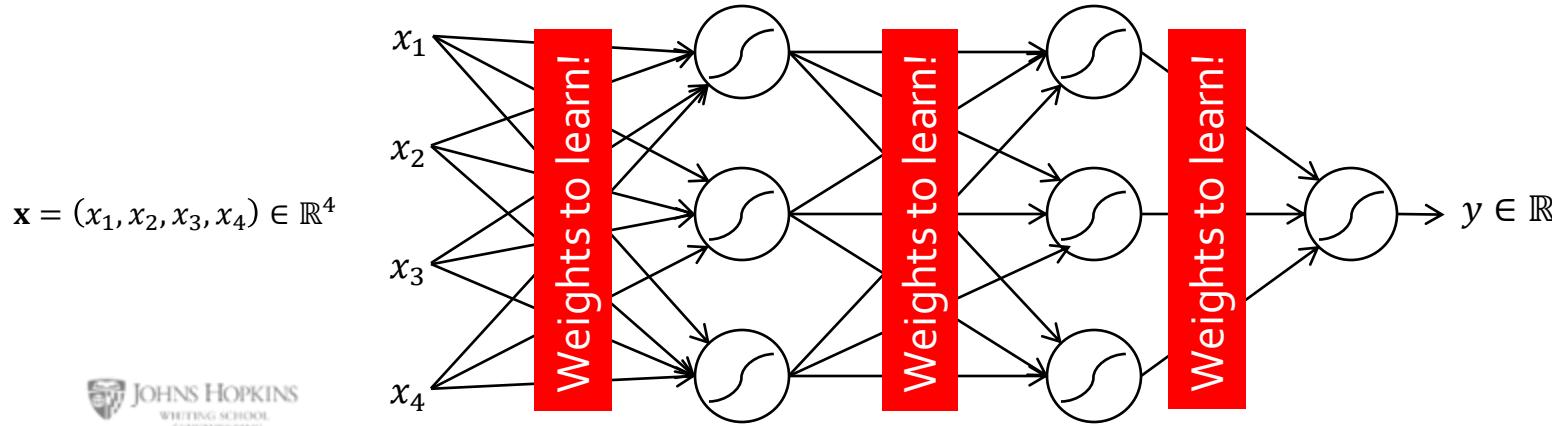


$$\mathbf{x} = (x_1, x_2, x_3, x_4) \in \mathbb{R}^4$$



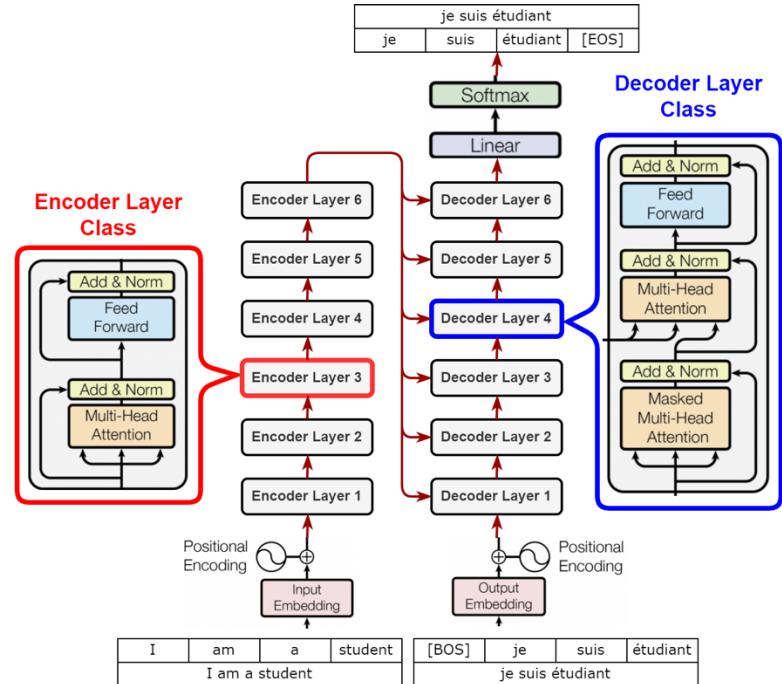
Training Neural Networks ~ Computing the Gradients

- How do you **efficiently** compute $\frac{\partial \mathcal{L}}{\partial W_i}$ for all parameters?
- It's easy to learn the final layer – it's just a linear unit.
- How about the weights in the earlier layers (i.e., before the final layer)?

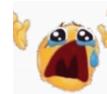


Necessity of a Principled Algorithm for Gradient Computation

- **Depth** gives more representational capacity to neural networks.
- However, computing gradients for deeper layers is **not trivial and tedious**.
- Even if we have analytical formula for gradient, if they're architecture-specific, they **must be repeated for each new architecture**.
- The solution is “Backpropagation” algorithm!



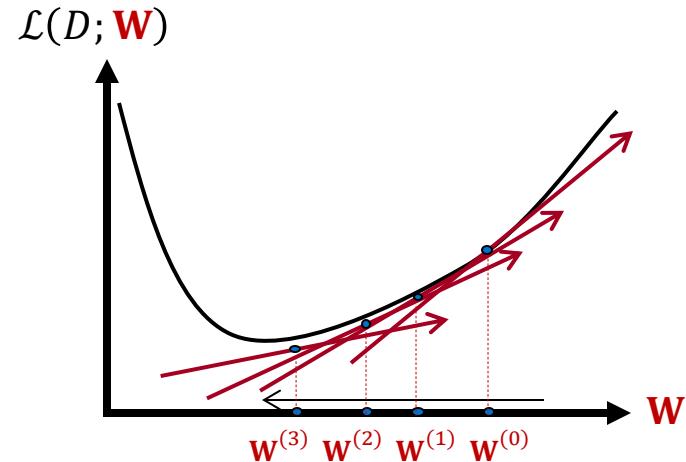
Architecture of the BERT model with over 24 layers and millions of parameters — we will study get to this model in a few weeks!



BP: Required Intuitions

1. Gradient Descent

- Change the weights \mathbf{W} in the direction of gradient to minimize the error function.

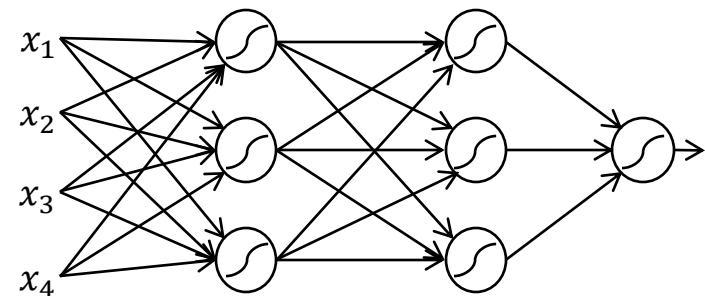


2. Chain Rule

- Use the chain rule to calculate the weights of the intermediate weights

3. Dynamic Programming (Memoization)

- Memoize the weight updates to make the updates faster.



A Generic Multi-Layer Perceptron

- Given the following definition:

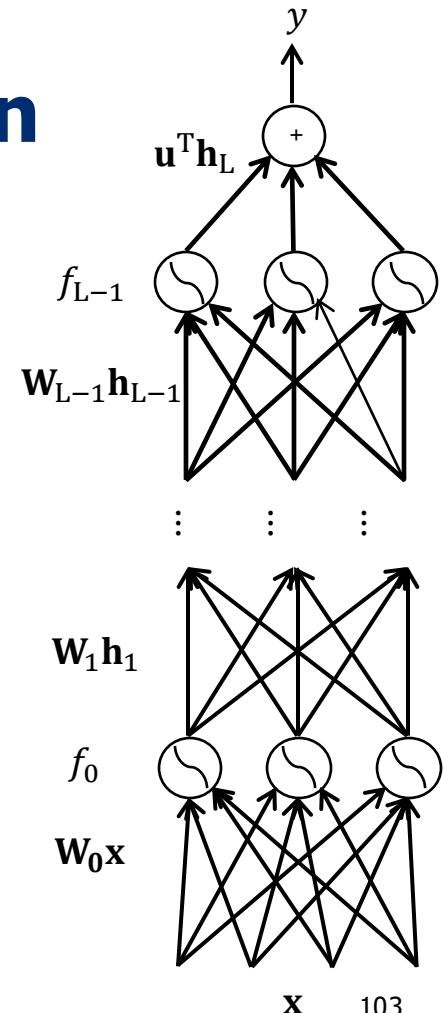
$$\mathbf{x} = \mathbf{h}_0 \in \mathbb{R}^{d_0} \text{ (input)}$$

$$\mathbf{h}_{i+1} = f_i(\mathbf{W}_i \mathbf{h}_i) \in \mathbb{R}^{d_i} \text{ (hidden layer } i, 0 \leq i \leq L - 1)$$

$$y = \mathbf{u}^T \mathbf{h}_L \in \mathbb{R} \text{ (output)}$$

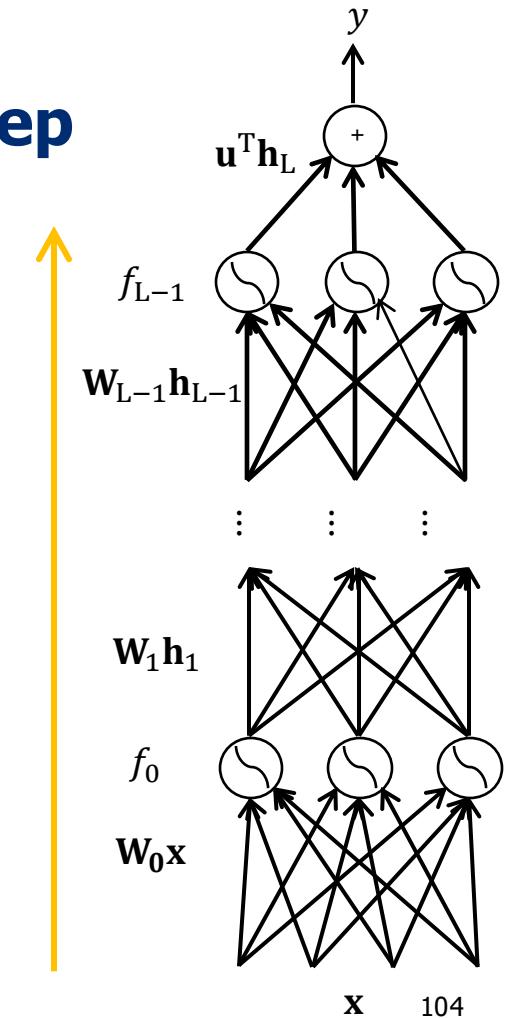
$$\mathcal{L} = \ell(y, y^*) \in \mathbb{R} \text{ (loss)}$$

- Trainable parameters: $\mathbf{W} = \{\mathbf{W}_0, \mathbf{W}_1, \dots, \mathbf{W}_L, \mathbf{u}\}$



A Generic Neural Network: Forward Step

- Given some [initial] values for the parameters, we can compute **the forward pass**, layer by layer.
- Forward pass is basically L matrix multiplications, each followed by an activation function.
- Matrix multiplication can be done efficiently with GPUs.
 - Therefore, **forward pass is somewhat fast**.
- Complexity of forward pass is **linear of depth $O(L)$** .



A Generic Neural Network: Direct Gradients

$$\mathbf{x} = \mathbf{h}_0 \in \mathbb{R}^{d_0} \text{ (input)}$$

$$\mathbf{h}_{i+1} = f_i(\mathbf{W}_i \mathbf{h}_i) \in \mathbb{R}^{d_i}$$

$$(0 \leq i \leq L - 1)$$

$$y = \mathbf{u}^T \mathbf{h}_L \in \mathbb{R} \text{ (output)}$$

$$\mathcal{L} = \ell(y, y^*) \in \mathbb{R} \text{ (loss)}$$

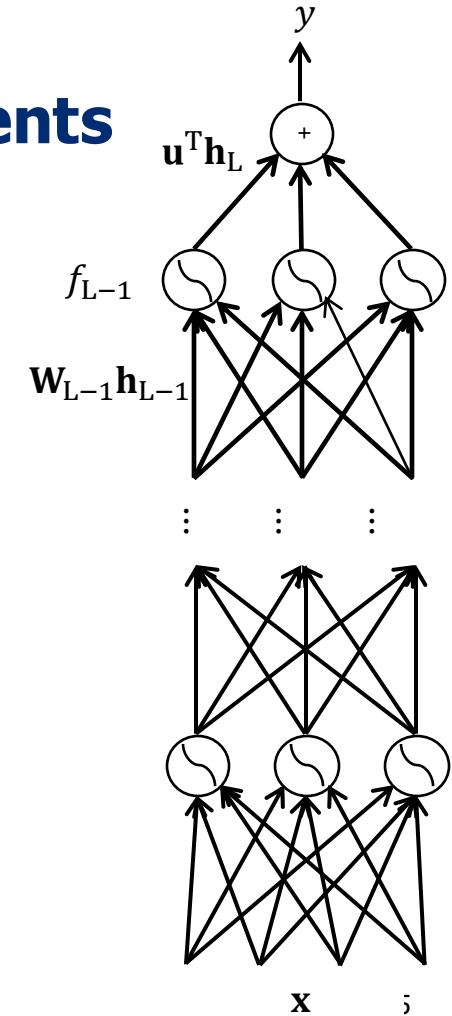
$$\mathbf{W} = \{\mathbf{W}_0, \mathbf{W}_1, \dots, \mathbf{W}_L, \mathbf{u}\}$$

We want the gradients of \mathcal{L} with respect to model parameters.

Use the chain rule to simplify the following term:

$$\nabla_{\mathbf{W}_{L-1}} \mathcal{L} = (\mathbf{J}_{\mathcal{L}}(\mathbf{W}_{L-1}))^T =$$

$$(\mathbf{J}_{\mathcal{L}}(y) \mathbf{J}_y(\mathbf{h}_L) \mathbf{J}_{\mathbf{h}_L}(\mathbf{W}_{L-1}))^T$$



A Generic Neural Network: Direct Gradients

$$\mathbf{x} = \mathbf{h}_0 \in \mathbb{R}^{d_0} \text{ (input)}$$

$$\mathbf{h}_{i+1} = f_i(\mathbf{W}_i \mathbf{h}_i) \in \mathbb{R}^{d_i}$$

$$(0 \leq i \leq L - 1)$$

$$y = \mathbf{u}^T \mathbf{h}_L \in \mathbb{R} \text{ (output)}$$

$$\mathcal{L} = \ell(y, y^*) \in \mathbb{R} \text{ (loss)}$$

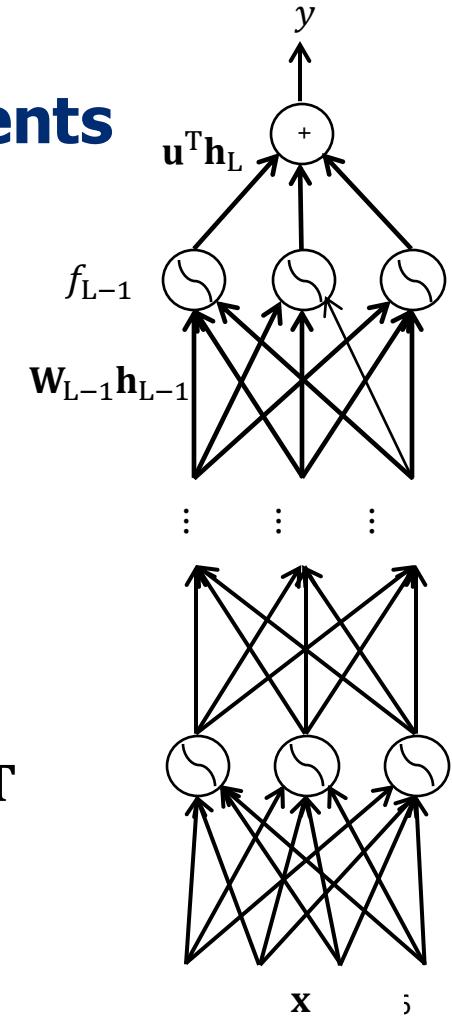
$$\mathbf{W} = \{\mathbf{W}_0, \mathbf{W}_1, \dots, \mathbf{W}_L, \mathbf{u}\}$$

We want the gradients of \mathcal{L} with respect to model parameters.

Use the chain rule to simplify the following term:

$$\nabla_{\mathbf{W}_{L-2}} \mathcal{L} = (\mathbf{J}_{\mathcal{L}}(\mathbf{W}_{L-2}))^T =$$

$$(\mathbf{J}_{\mathcal{L}}(y) \mathbf{J}_y(\mathbf{h}_L) \mathbf{J}_{\mathbf{h}_L}(\mathbf{h}_{L-1}) \mathbf{J}_{\mathbf{h}_{L-1}}(\mathbf{W}_{L-2}))^T$$



A Generic Neural Network: Direct Gradients

$$\mathbf{x} = \mathbf{h}_0 \in \mathbb{R}^{d_0} \text{ (input)}$$

$$\mathbf{h}_{i+1} = f_i(\mathbf{W}_i \mathbf{h}_i) \in \mathbb{R}^{d_i}$$

$$(0 \leq i \leq L - 1)$$

$$y = \mathbf{u}^T \mathbf{h}_L \in \mathbb{R} \text{ (output)}$$

$$\mathcal{L} = \ell(y, y^*) \in \mathbb{R} \text{ (loss)}$$

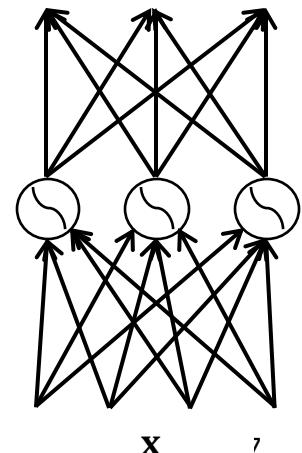
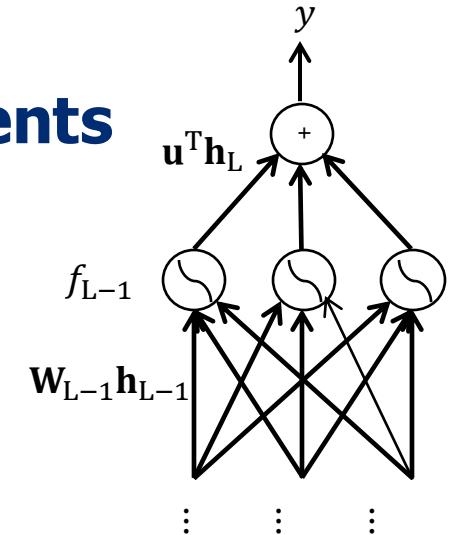
$$\mathbf{W} = \{\mathbf{W}_0, \mathbf{W}_1, \dots, \mathbf{W}_L, \mathbf{u}\}$$

We want the gradients of \mathcal{L} with respect to model parameters.

Use the chain rule to simplify the following term:

$$\nabla_{\mathbf{W}_{L-i}} \mathcal{L} = (\mathbf{J}_{\mathcal{L}}(\mathbf{W}_{L-i}))^T =$$

$$(\mathbf{J}_{\mathcal{L}}(y) \mathbf{J}_y(\mathbf{h}_L) \mathbf{J}_{\mathbf{h}_L}(\mathbf{h}_{L-1}) \dots \mathbf{J}_{\mathbf{h}_{L-i+1}}(\mathbf{W}_{L-i}))^T$$



A Generic Neural Network: Direct Gradients

$$\mathbf{x} = \mathbf{h}_0 \in \mathbb{R}^{d_0} \text{ (input)}$$

$$\mathbf{h}_{i+1} = f_i(\mathbf{W}_i \mathbf{h}_i) \in \mathbb{R}^{d_i}$$

$$(0 \leq i \leq L - 1)$$

$$y = \mathbf{u}^T \mathbf{h}_L \in \mathbb{R} \text{ (output)}$$

$$\mathcal{L} = \ell(y, y^*) \in \mathbb{R} \text{ (loss)}$$

$$\mathbf{W} = \{\mathbf{W}_0, \mathbf{W}_1, \dots, \mathbf{W}_L, \mathbf{u}\}$$

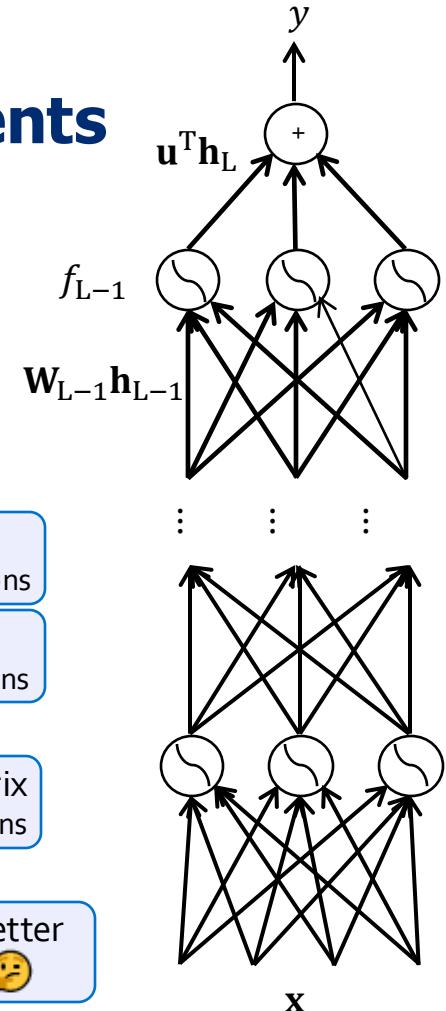
We want the gradients of \mathcal{L} with respect to model parameters.

- $\nabla_{\mathbf{W}_{L-1}} \mathcal{L} = (\mathbf{J}_{\mathcal{L}}(\mathbf{W}_{L-1}))^T = (\mathbf{J}_{\mathcal{L}}(y) \mathbf{J}_y(\mathbf{h}_L) \mathbf{J}_{\mathbf{h}_L}(\mathbf{W}_{L-1}))^T$
- $\nabla_{\mathbf{W}_{L-2}} \mathcal{L} = (\mathbf{J}_{\mathcal{L}}(\mathbf{W}_{L-2}))^T = (\mathbf{J}_{\mathcal{L}}(y) \mathbf{J}_y(\mathbf{h}_L) \mathbf{J}_{\mathbf{h}_L}(\mathbf{h}_{L-1}) \mathbf{J}_{\mathbf{h}_{L-1}}(\mathbf{W}_{L-2}))^T$
- ...
- $\nabla_{\mathbf{W}_0} \mathcal{L} = (\mathbf{J}_{\mathcal{L}}(\mathbf{W}_{L-3}))^T = (\mathbf{J}_{\mathcal{L}}(y) \mathbf{J}_y(\mathbf{h}_L) \mathbf{J}_{\mathbf{h}_L}(\mathbf{h}_{L-1}) \dots \mathbf{J}_{\mathbf{h}_1}(\mathbf{W}_0))^T$

In total, how many matrix multiplications are done here?

- (A) $O(L)$ (B) $O(L^2)$ (C) $O(L^3)$ (D) $O(\exp(L))$

Can we do better than this? 😐



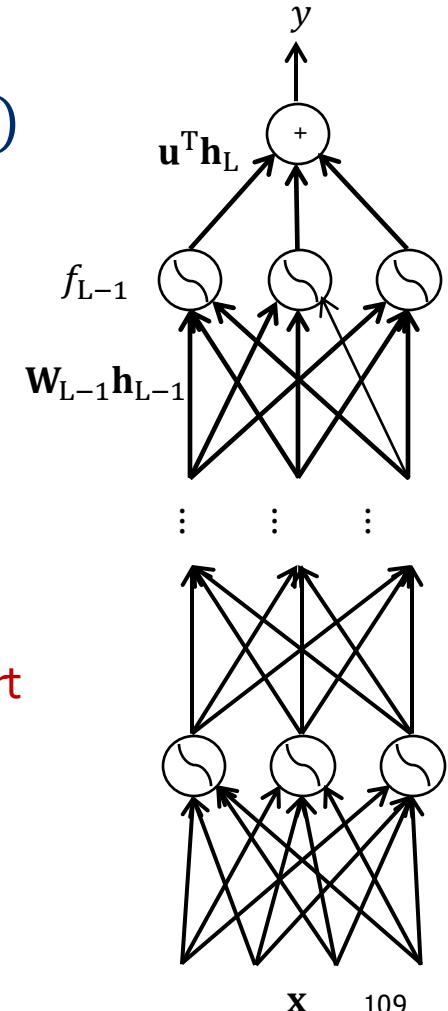
Caching Gradients to Turn $O(L^2)$ into $O(L)$

- Suppose we're computing:

$$\nabla_{\mathbf{W}_{L-1}} \mathcal{L} = \left(\mathbf{J}_L(y) \mathbf{J}_y(\mathbf{h}_L) \mathbf{J}_{\mathbf{h}_L}(\mathbf{W}_{L-1}) \right)^T$$

$$\nabla_{\mathbf{W}_{L-2}} \mathcal{L} = \left(\mathbf{J}_L(y) \mathbf{J}_y(\mathbf{h}_L) \mathbf{J}_{\mathbf{h}_L}(\mathbf{h}_{L-1}) \mathbf{J}_{\mathbf{h}_{L-1}}(\mathbf{W}_{L-2}) \right)^T$$

- Parameter gradients **depend on the gradients of the earlier layers!**
- So, when computing gradients at each layer, **we don't need to start from scratch!**
- I can **reuse gradients** computed for higher layers for lower layers (i.e., memoization).



Caching Gradients to Turn $O(L^2)$ into $O(L)$

- First, precompute these cache values:

$O(L)$ matrix multiplications

First layer: $\delta_L = \mathbf{J}_L(y) \mathbf{J}_y(\mathbf{h}_L)$

Subsequent layers: $\delta_i = \delta_{i+1} \mathbf{J}_{\mathbf{h}_i}(\mathbf{h}_{i-1}), \forall i: 0 \leq i \leq L - 1$

- Now, let's go back to the gradients:

$$\nabla_{\mathbf{W}_{L-1}} \mathcal{L} = (\mathbf{J}_L(y) \mathbf{J}_y(\mathbf{h}_L) \mathbf{J}_{\mathbf{h}_L}(\mathbf{W}_{L-1}))^\top = (\delta_L \mathbf{J}_{\mathbf{h}_L}(\mathbf{W}_{L-1}))^\top$$

One matrix multiplications

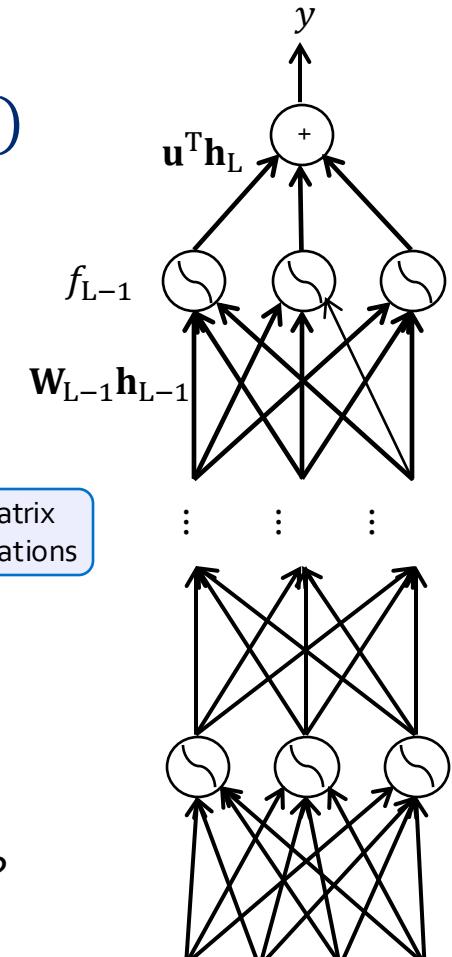
$$\nabla_{\mathbf{W}_{L-2}} \mathcal{L} = (\mathbf{J}_L(y) \mathbf{J}_y(\mathbf{h}_L) \mathbf{J}_{\mathbf{h}_L}(\mathbf{h}_{L-1}) \mathbf{J}_{\mathbf{h}_{L-1}}(\mathbf{W}_{L-2}))^\top = (\delta_{L-1} \mathbf{J}_{\mathbf{h}_{L-1}}(\mathbf{W}_{L-2}))^\top$$

One matrix multiplications

...

$$\nabla_{\mathbf{W}_0} \mathcal{L} = (\mathbf{J}_L(y) \mathbf{J}_y(\mathbf{h}_L) \mathbf{J}_{\mathbf{h}_L}(\mathbf{h}_{L-1}) \dots \mathbf{J}_{\mathbf{h}_1}(\mathbf{W}_0))^\top = (\delta_1 \mathbf{J}_{\mathbf{h}_1}(\mathbf{W}_0))^\top$$

One matrix multiplications



In total, how many matrix multiplications are done here when using caching?

- (A) $O(L)$ (B) $O(L^2)$ (C) $O(L^3)$ (D) $O(\exp(L))$

Gradient: Local Grad + Upstream Grad

- Gradients at each layer computed by

$$\nabla_{\mathbf{W}_{L-i}} \mathcal{L} = \left(\delta_{L-i+1} \mathbf{J}_{\mathbf{h}_{L-i+1}}(\mathbf{W}_{L-i}) \right)^T$$

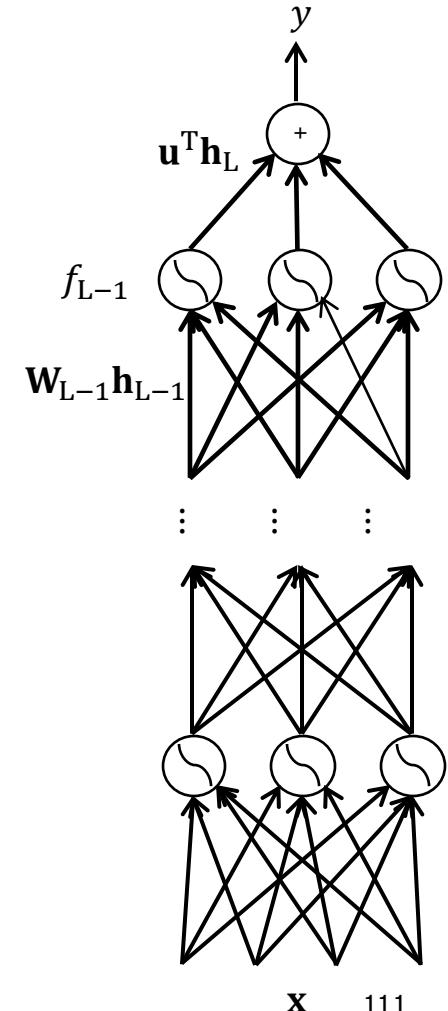
Upstream gradient ~ We lookup from the layer above.

Local Gradient

Let δ_i denote Jacobian at the output of layer i :

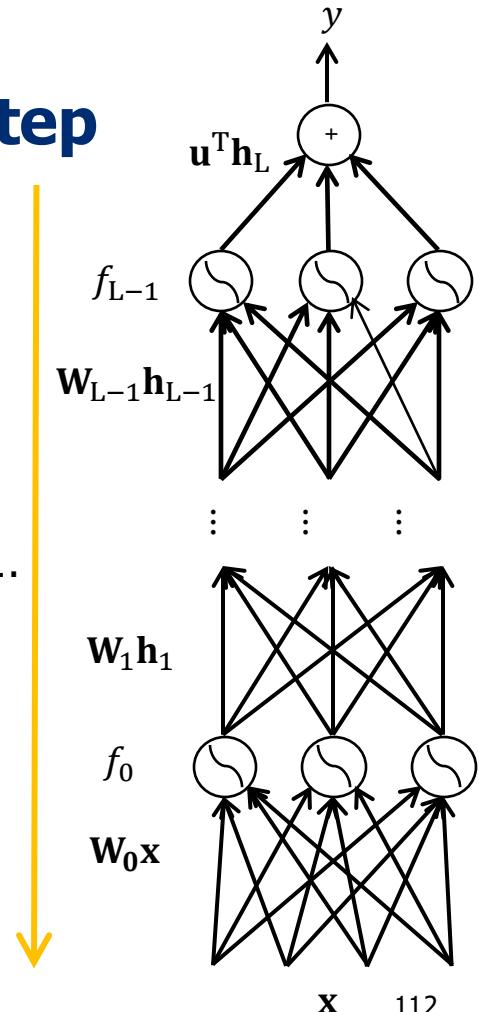
$$\delta_L = \mathbf{J}_{\mathcal{L}}(\mathbf{y}) \mathbf{J}_y(\mathbf{h}_L)$$

$$\delta_i = \delta_{i+1} \mathbf{J}_{\mathbf{h}_i}(\mathbf{h}_{i-1})$$



A Generic Neural Network: Backward Step

- Backward step computes the gradients starting from the end to the beginning, layer by layer.
- Start by computing **local gradients**: $J_{h_{L-i+1}}(W_{L-i})$
- Use then to compute **upstream gradients** δ_L , then δ_{L-1} , then δ_{L-2} ,
- Use these to compute **global gradients**: $\nabla_{W_i} \mathcal{L}$
- Computational cost as a function of depth:
 - With memoization, gradient computation is a **linear** function of depth L
 - (same cost as the forward process!!)
 - Without memorization, gradients computation would grow **quadratic** with L



A Generic Neural Network: Back Propagation

Initialize network parameters with random values

Loop until convergence

 Loop over training instances

i. Forward step:

 Start from the input and compute all the layers till the end (loss \mathcal{L})

In practice, this step is done over **batches** of instances!

ii. Backward step:

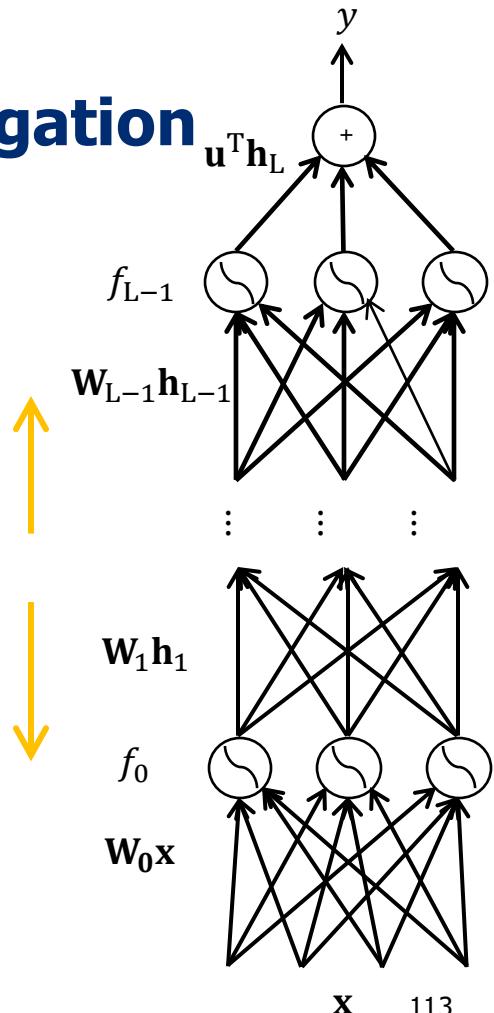
 Compute **local gradients**, starting from the last layer

 Compute **upstream gradients** δ_i values, starting from the last layer

 Use δ_i values to compute global gradients $\nabla_{\mathcal{L}}(\mathbf{W}_i)$ at each layer

iii. Gradient update:

 Update each parameter: $\mathbf{W}_i^{(t+1)} \leftarrow \mathbf{W}_i^{(t)} - \alpha \nabla_{\mathbf{W}_i} \mathcal{L}$



Summary

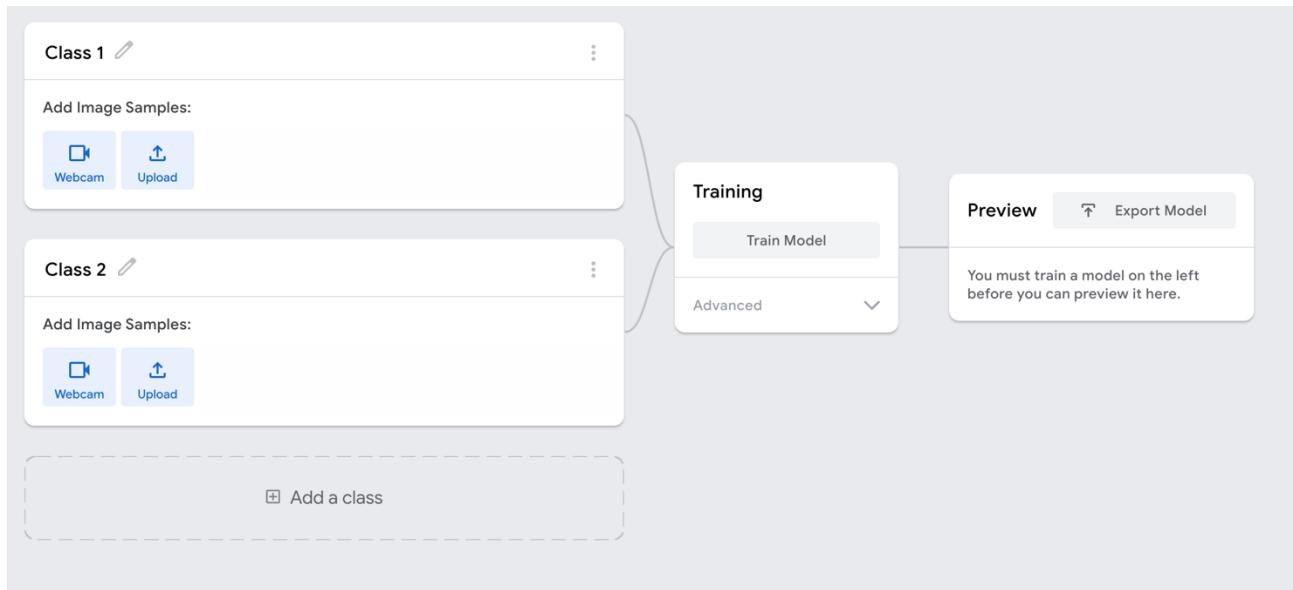
- Backpropagation: an algorithm for training neural networks.
- Using Dynamic Programming for efficient computation of gradients.
- **Next:** Backprop in real practice.

Quick Pulse Check

- How is **Backprop** different from/related to **Gradient Descent**?
 - **Backpropagation** is an algorithm used to compute the gradient of the loss function with respect to each weight in the network.
 - **Gradient descent**, on the other hand, is an optimization algorithm that uses these computed gradients to update the network's weights.
 - *Basically, backpropagation provides the necessary gradients for gradient descent to perform the weight updates.*

Demo Time!

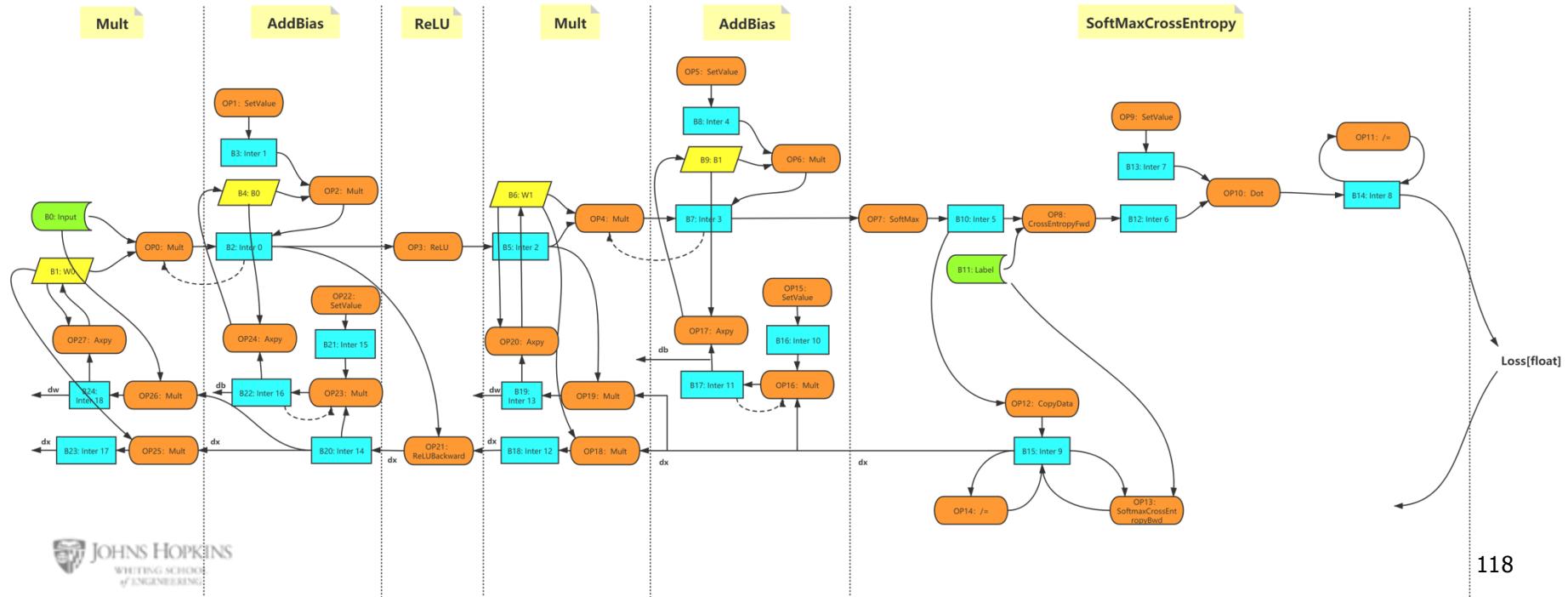
- <https://teachablemachine.withgoogle.com/>



Backprop in Real Practice: Computation Graph

Computation Graph: Example

- In reality, neural networks are not as regular as the previous example ...

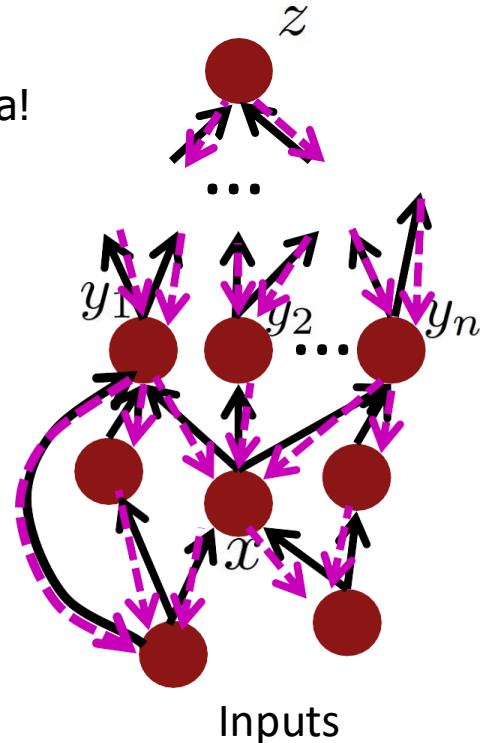
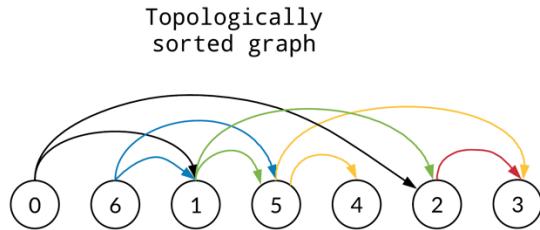
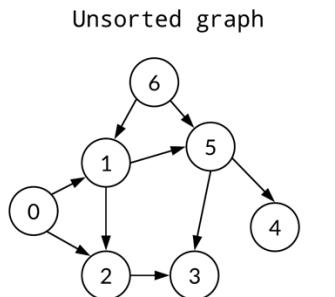


Backprop in General Computation Graph

Single scalar output

- What if the network does not have a regular structure? Same idea!

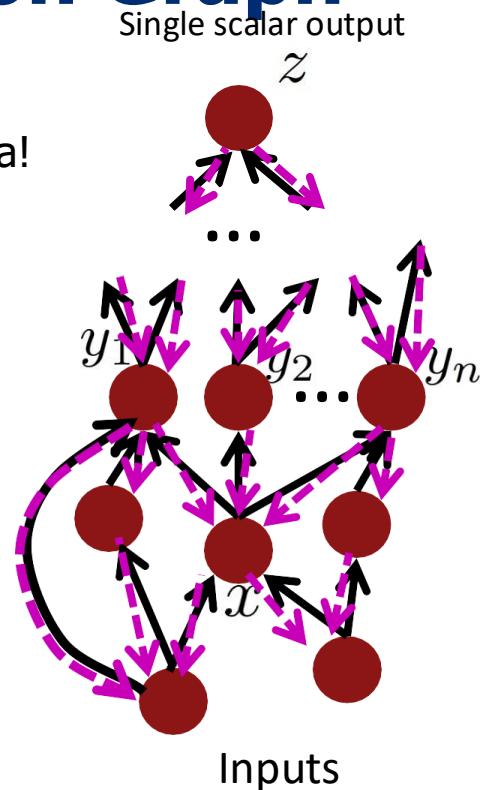
1. Sort the nodes in **topological order** (what depends on what)
 - **Cost:** Linear in the number of nodes/edges.



Backprop in General Computation Graph

- What if the network does not have a regular structure? Same idea!

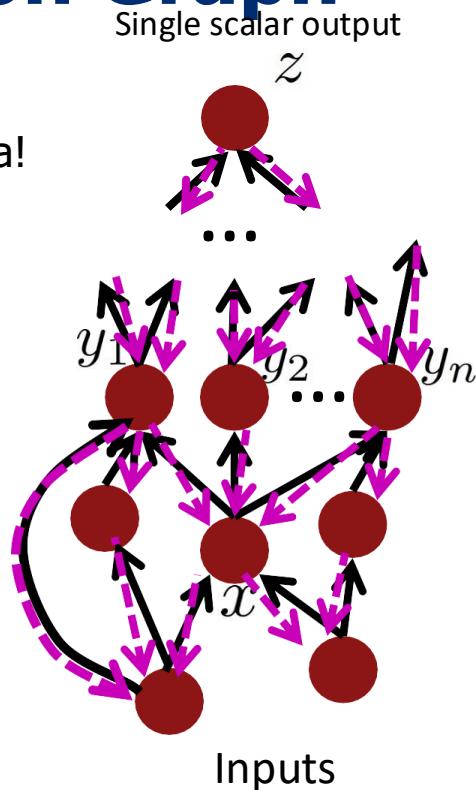
1. Sort the nodes in **topological order** (what depends on what)
2. Forward-Propagation:
 - Visit nodes in topological sort order and compute value of node given predecessors
 - **Cost:** Linear in the number of node/edges



Backprop in General Computation Graph

- What if the network does not have a regular structure? Same idea!

1. Sort the nodes in **topological order** (what depends on what)
2. Forward-Propagation:
 - Visit nodes in topological sort order and compute value of node given predecessors
3. Backward-Propagation:
 - Compute **local gradients**
 - Visit nodes in reverse order and compute **global gradients** using gradients of successors
 - **Cost:** Linear in the number of nodes/edges.



Computation Graph: View inside each node

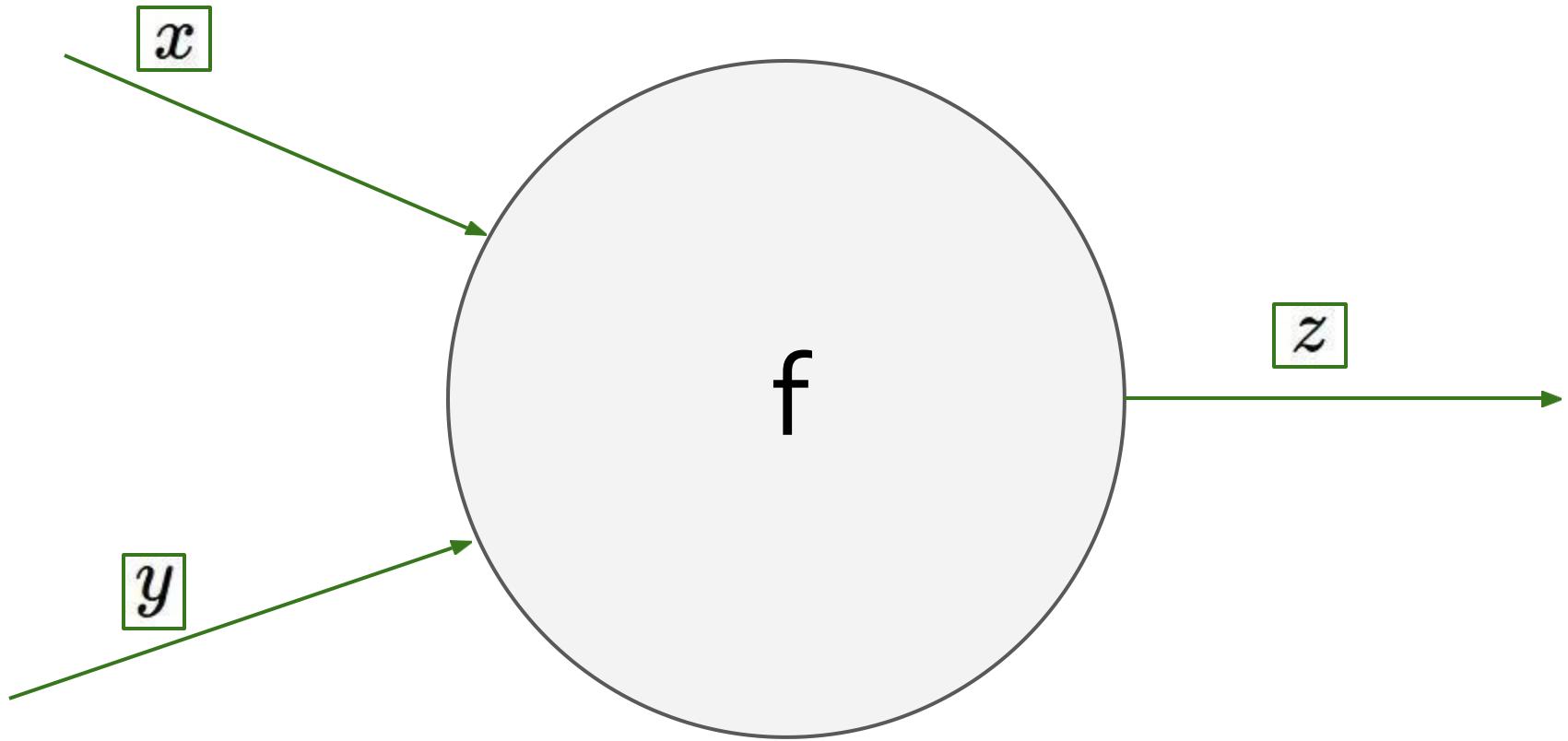


Figure from Andrej Karpathy

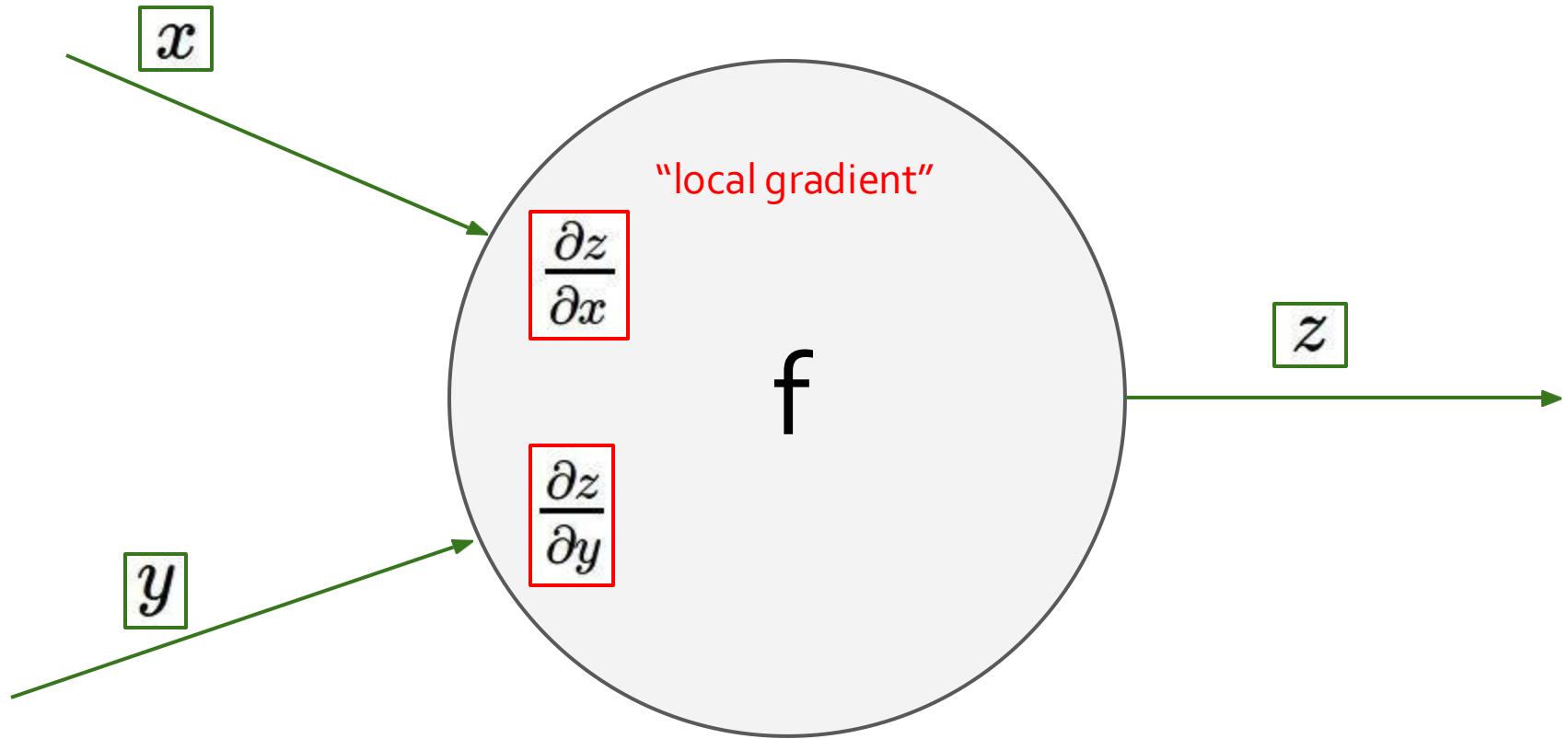
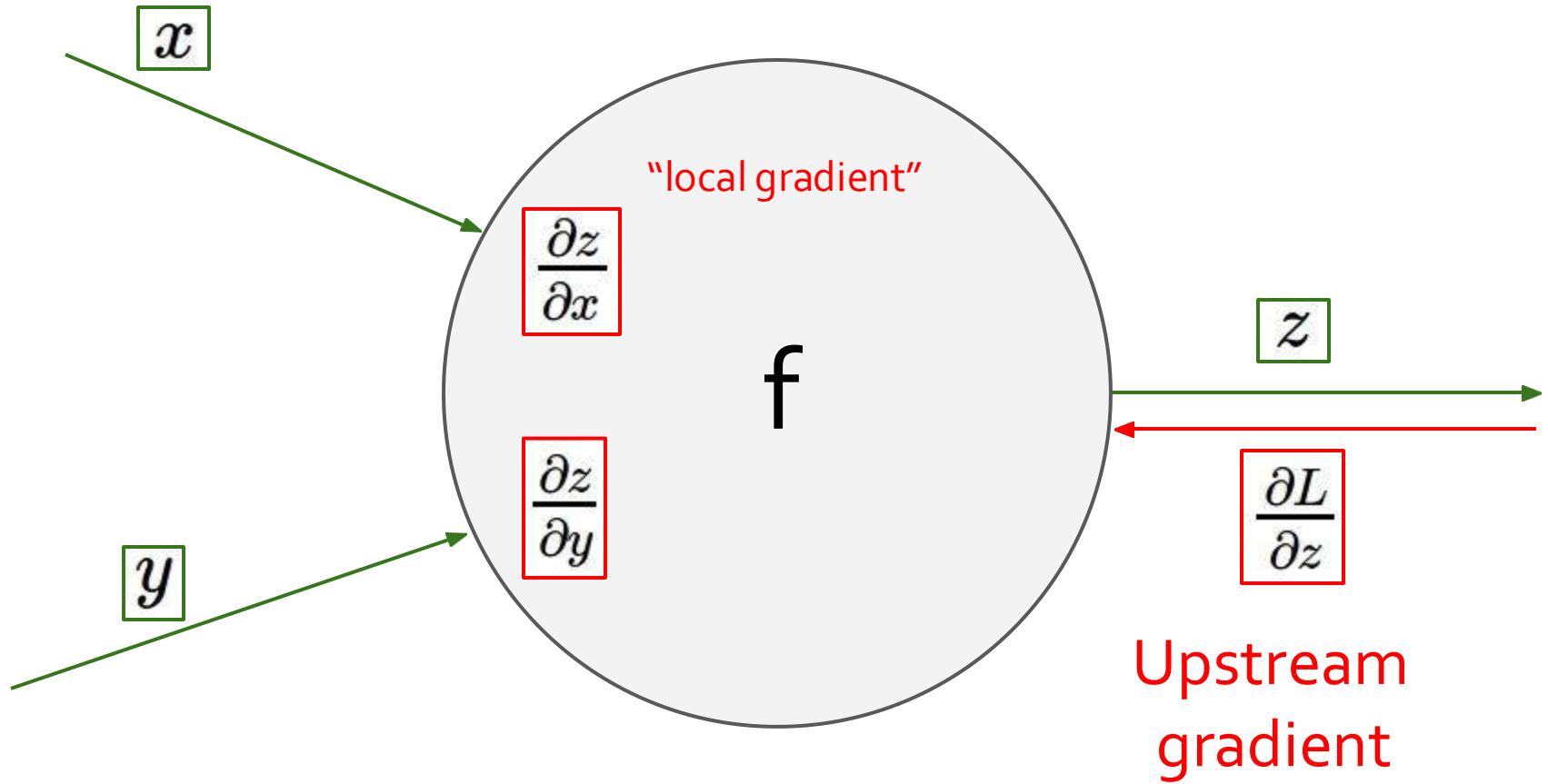
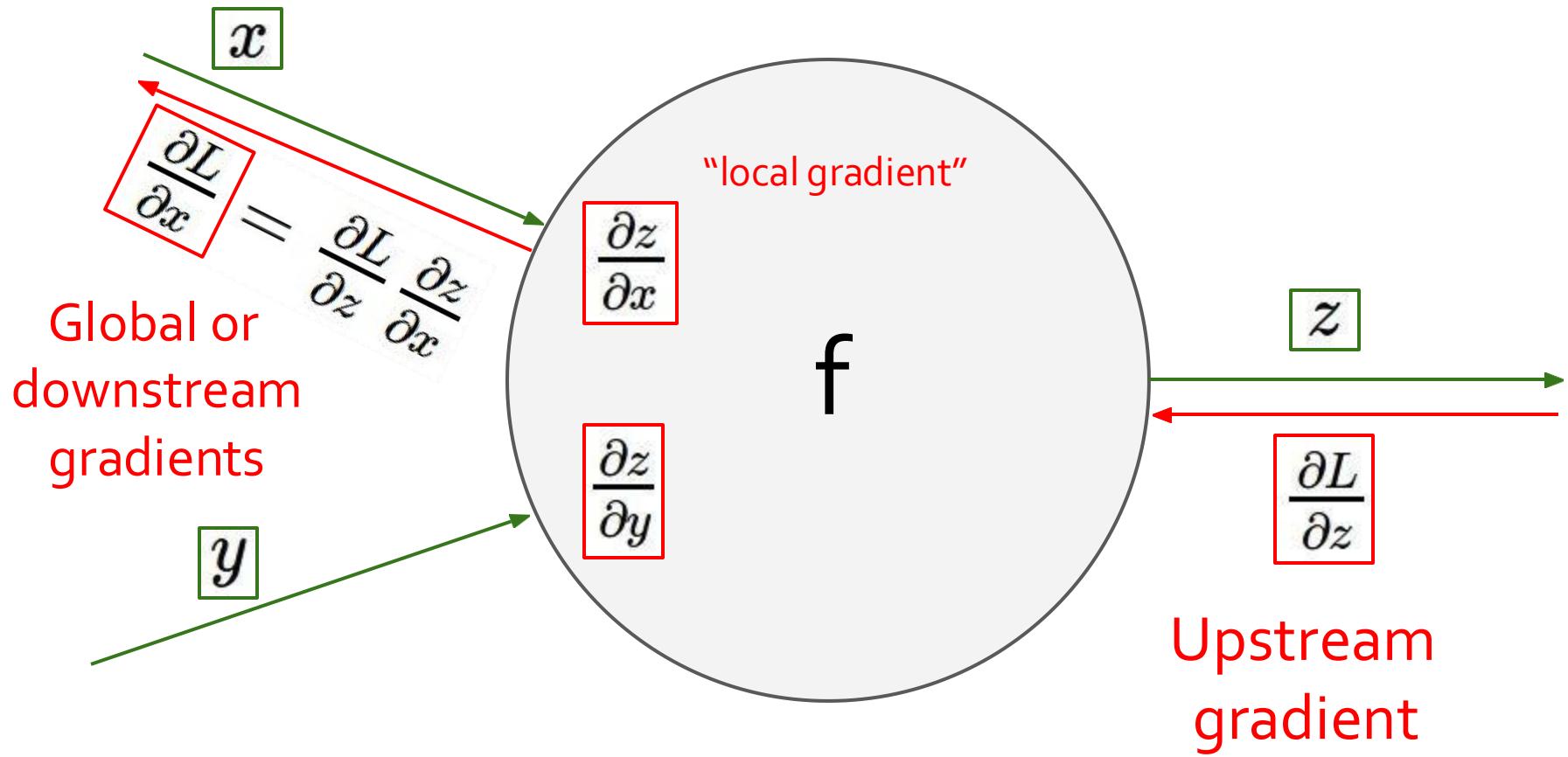
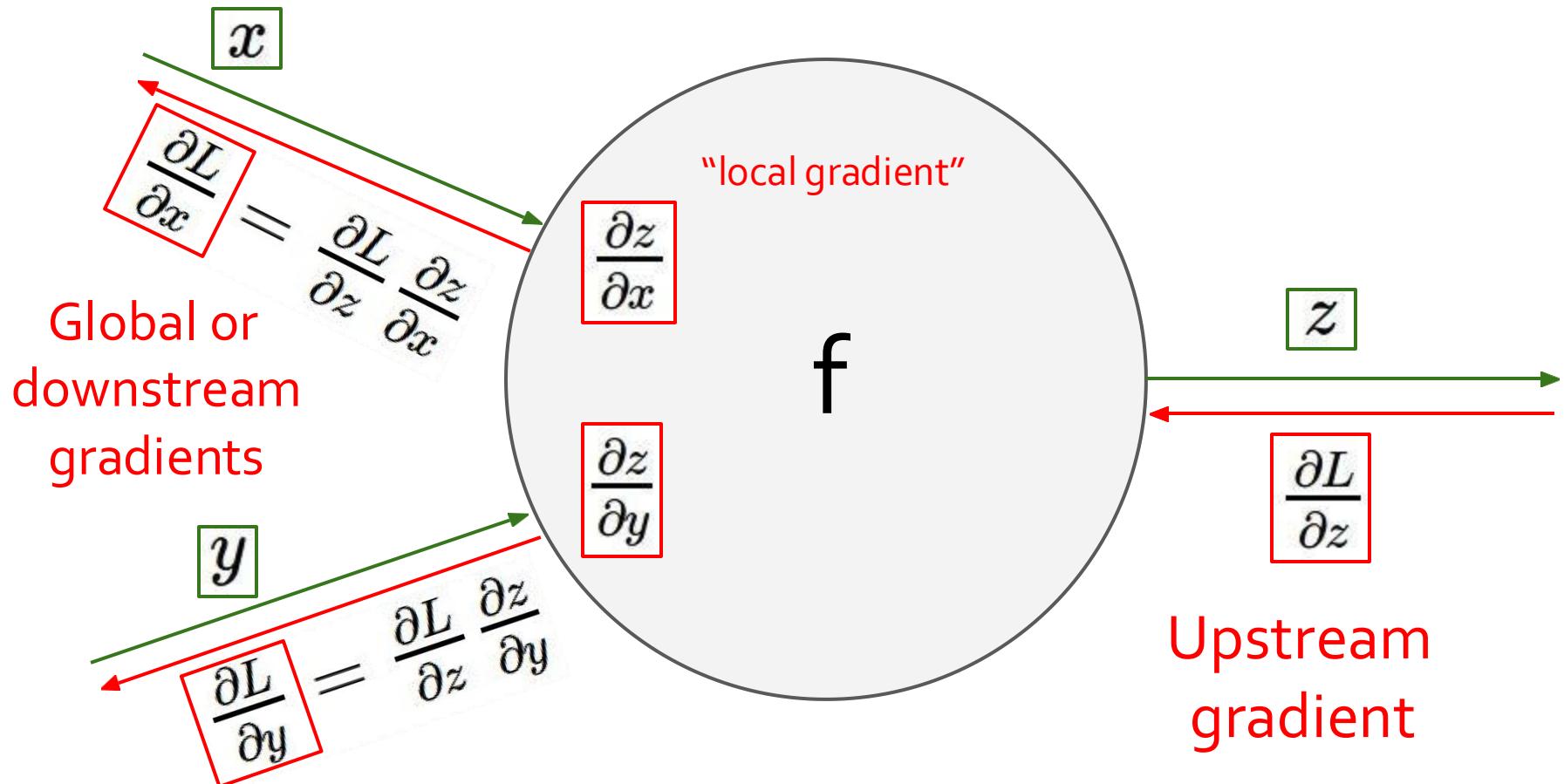


Figure from Andrej Karpathy







Global or
downstream
gradients

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x}$$

$$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial y}$$

“local gradient”

$$\frac{\partial z}{\partial x}$$

$$\frac{\partial z}{\partial y}$$

f

$$z$$

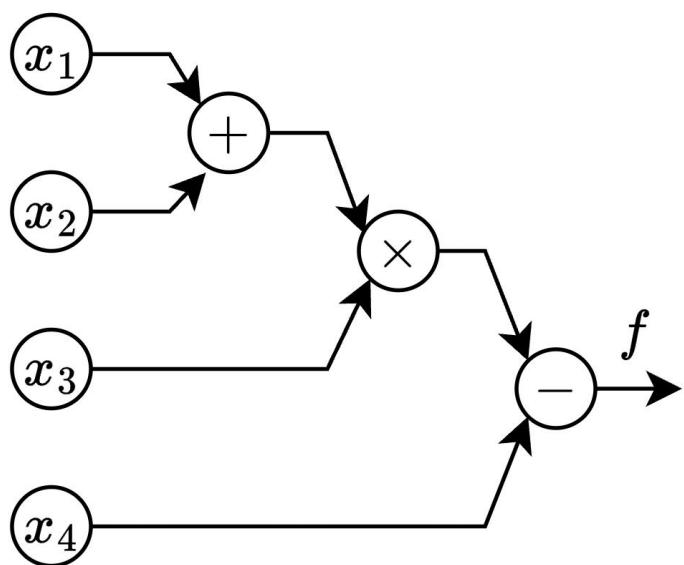
$$\frac{\partial L}{\partial z}$$

Upstream
gradient

Computation Graph: An Example

$$f(x_1, x_2, x_3, x_4, x_5) = (x_1 + x_2)x_3 - x_4$$

Evaluated at: $(x_1, x_2, x_3, x_4) = (5, 4, 3, 2)$



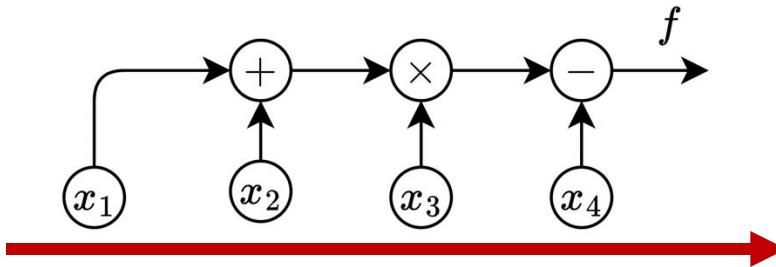
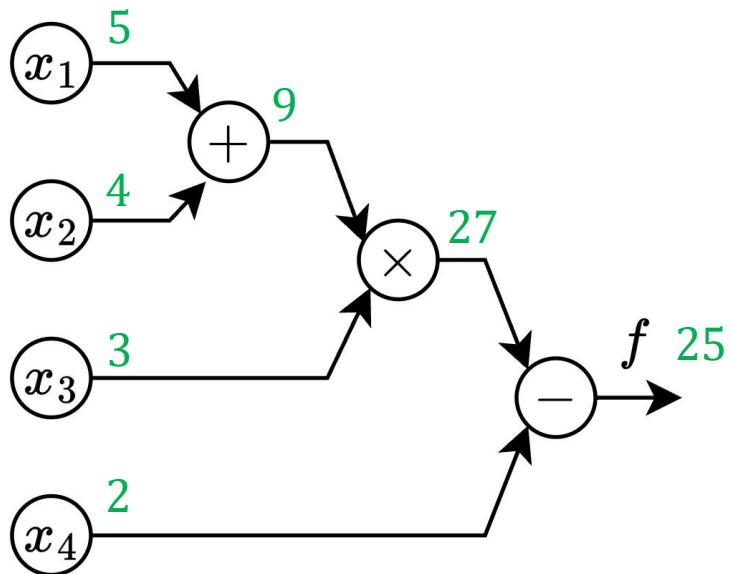
Want: $\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial x_4}$

In what order should
we process the
forward step?

Computation Graph: An Example

$$f(x_1, x_2, x_3, x_4, x_5) = (x_1 + x_2)x_3 - x_4$$

Evaluated at: $(x_1, x_2, x_3, x_4) = (5, 4, 3, 2)$

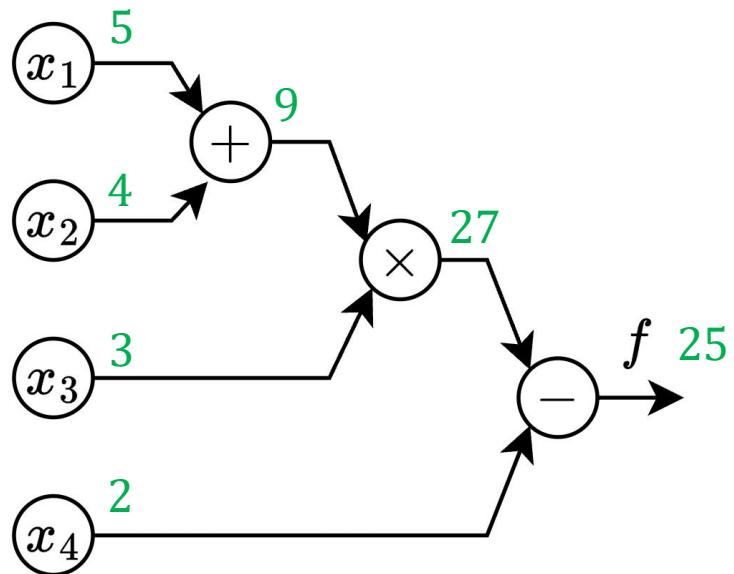


In what order should we process the **forward** step?

Computation Graph: An Example

$$f(x_1, x_2, x_3, x_4, x_5) = (x_1 + x_2)x_3 - x_4$$

Evaluated at: $(x_1, x_2, x_3, x_4) = (5, 4, 3, 2)$



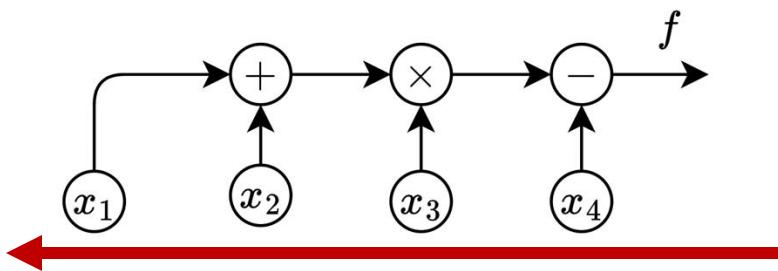
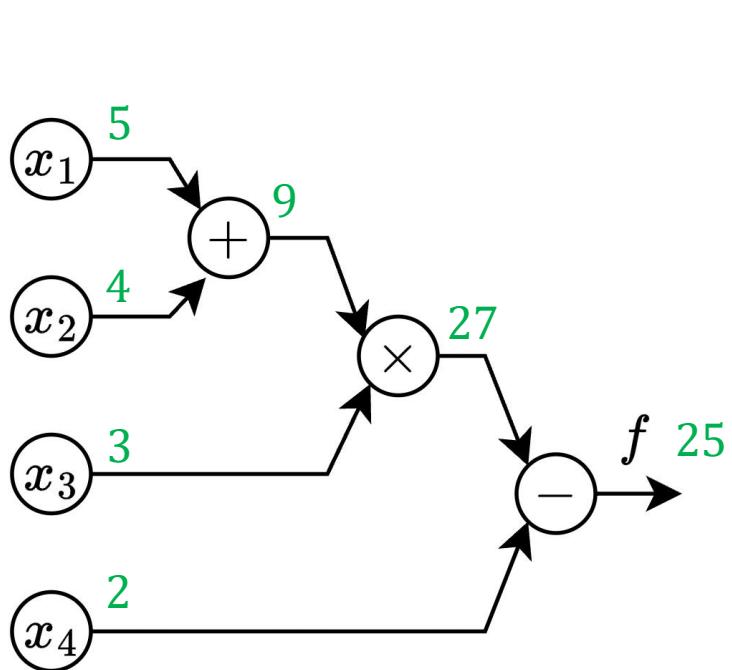
Want: $\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial x_4}$

In what order should
we process the
backward step?

Computation Graph: An Example

$$f(x_1, x_2, x_3, x_4, x_5) = (x_1 + x_2)x_3 - x_4$$

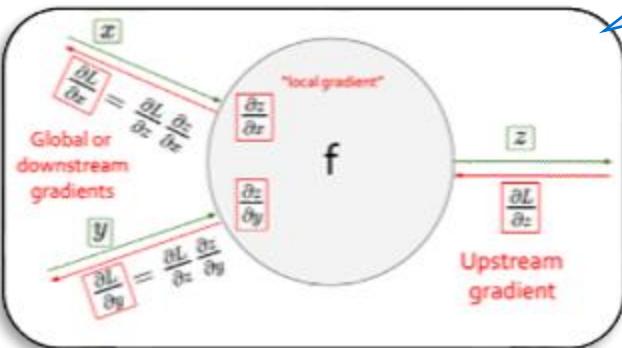
Evaluated at: $(x_1, x_2, x_3, x_4) = (5, 4, 3, 2)$



In what order should
we process the
backward step?

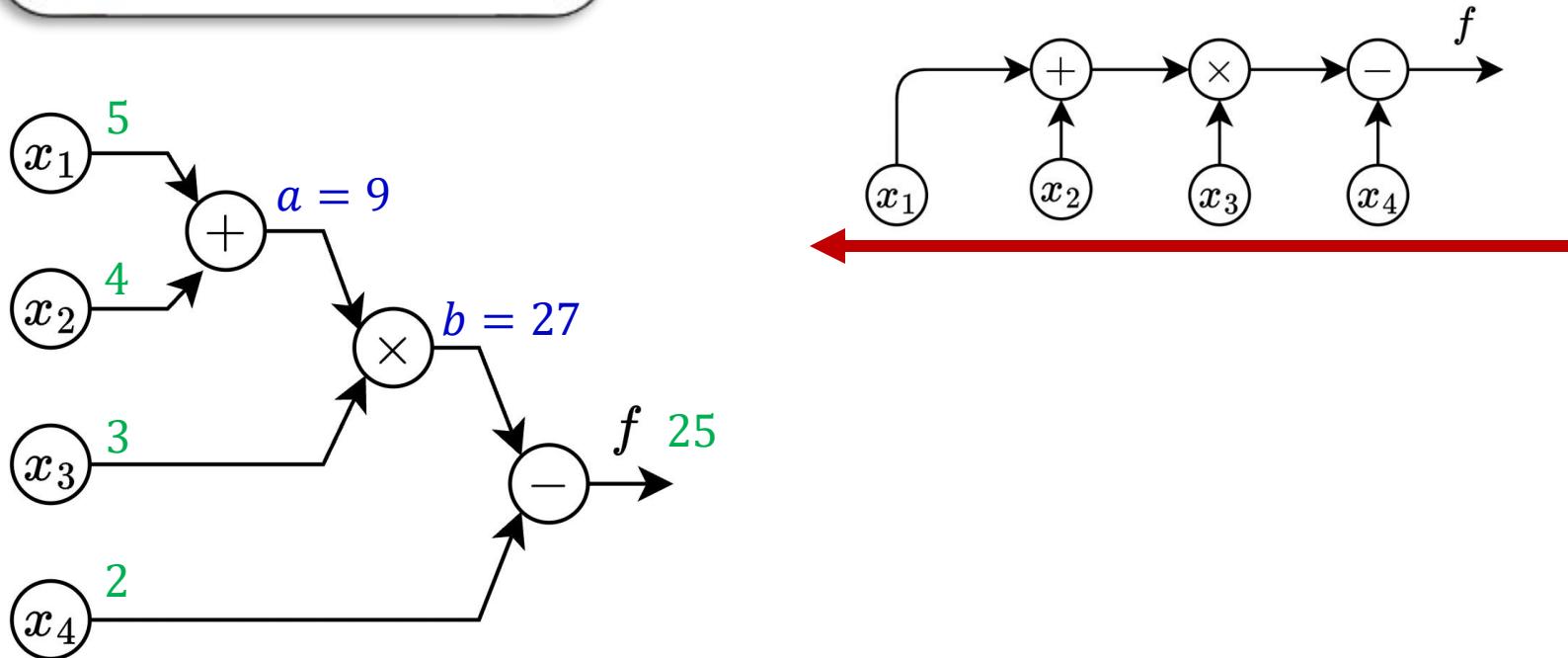
Remember this!

Graph: An Example



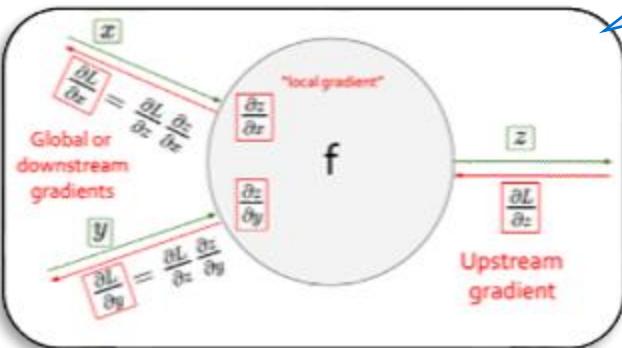
$$f(x_1, x_2, x_3, x_4, x_5) = (x_1 + x_2)x_3 - x_4$$

$$\text{Evaluated at: } (x_1, x_2, x_3, x_4) = (5, 4, 3, 2)$$



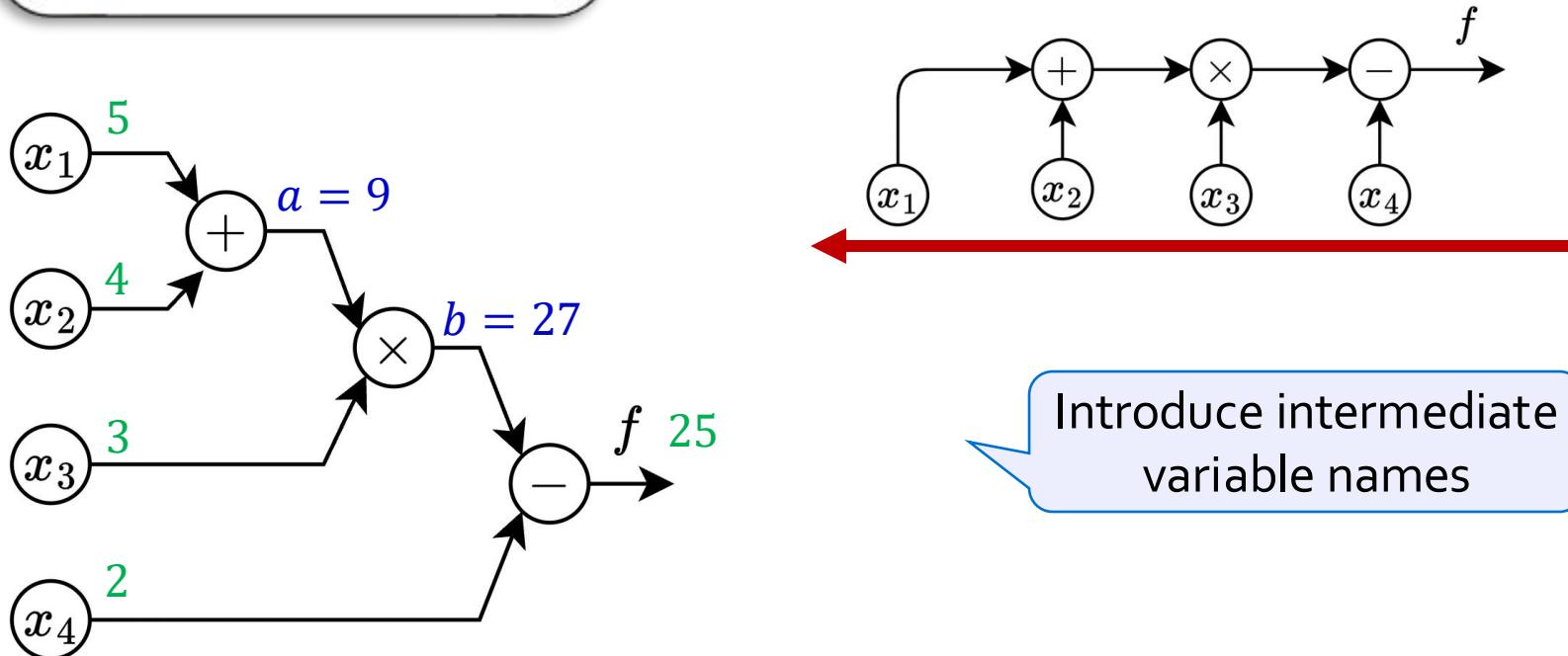
Remember this!

Graph: An Example



$$f(x_1, x_2, x_3, x_4, x_5) = (x_1 + x_2)x_3 - x_4$$

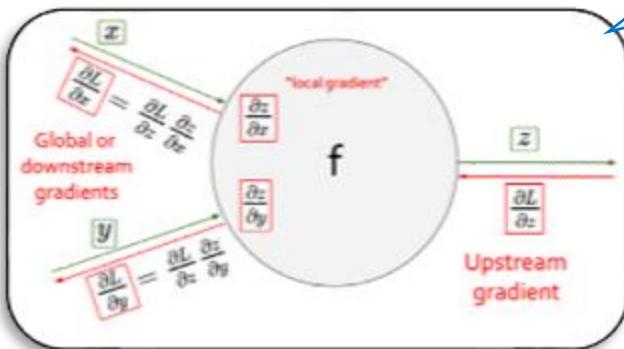
$$\text{Evaluated at: } (x_1, x_2, x_3, x_4) = (5, 4, 3, 2)$$



Remember this!

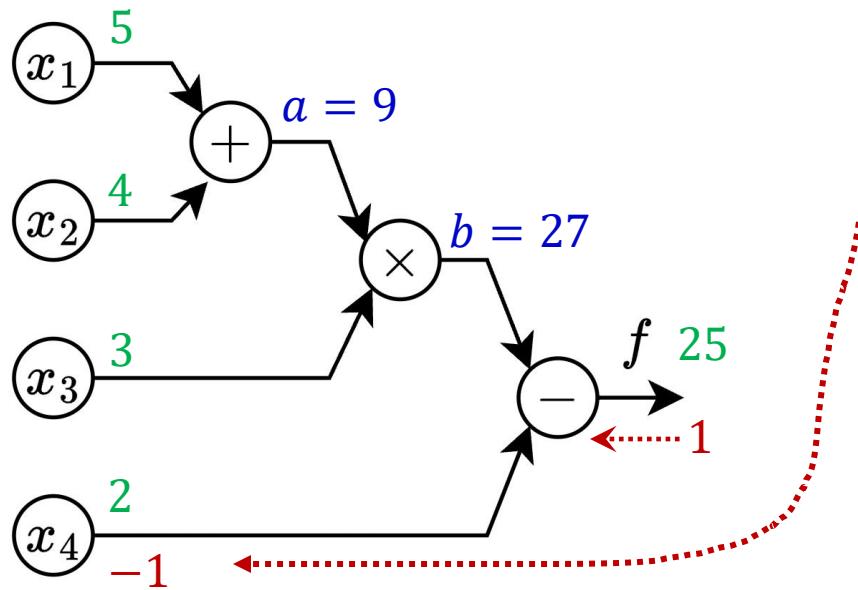
U: Upstream grad
L: Local grad

Graph: An Example



$$f(x_1, x_2, x_3, x_4, x_5) = (x_1 + x_2)x_3 - x_4$$

Evaluated at: $(x_1, x_2, x_3, x_4) = (5, 4, 3, 2)$



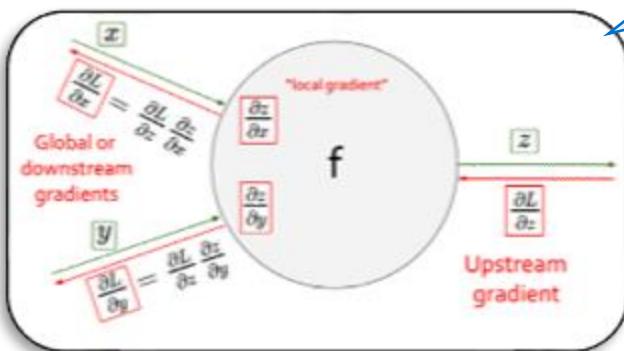
$$f = b - x_4$$

$$\frac{\partial f}{\partial x_4} = \frac{\text{L}}{\frac{\partial f}{\partial x_4}} \times \frac{\text{U}}{\frac{\partial f}{\partial f}} = (-1) \times 1 = -1$$

U: Upstream grad
L: Local grad

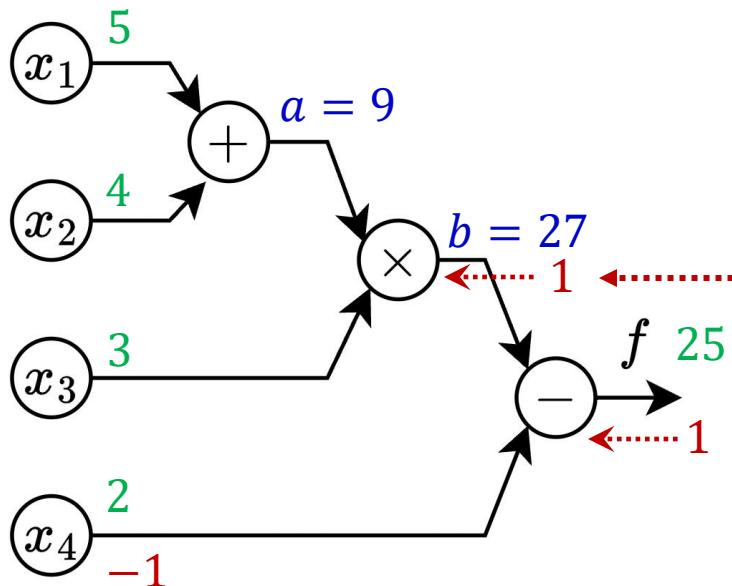
Remember this!

Graph: An Example



$$f(x_1, x_2, x_3, x_4, x_5) = (x_1 + x_2)x_3 - x_4$$

Evaluated at: $(x_1, x_2, x_3, x_4) = (5, 4, 3, 2)$



$$f = b - x_4$$

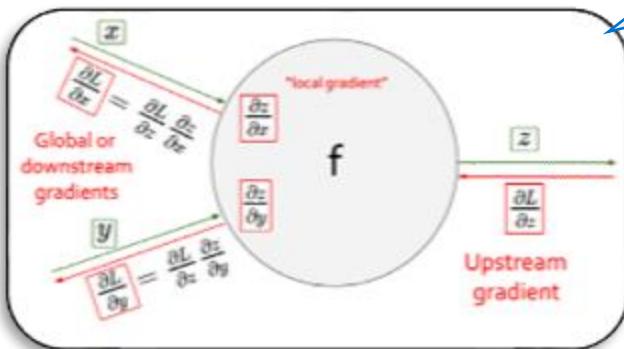
$$\frac{\partial f}{\partial x_4} = \frac{\text{L}}{\frac{\partial f}{\partial x_4}} \times \frac{\text{U}}{\frac{\partial f}{\partial f}} = (-1) \times 1 = -1$$

$$\frac{\partial f}{\partial b} = \frac{\text{L}}{\frac{\partial f}{\partial b}} \times \frac{\text{U}}{\frac{\partial f}{\partial f}} = 1 \times 1 = 1$$

Remember this!

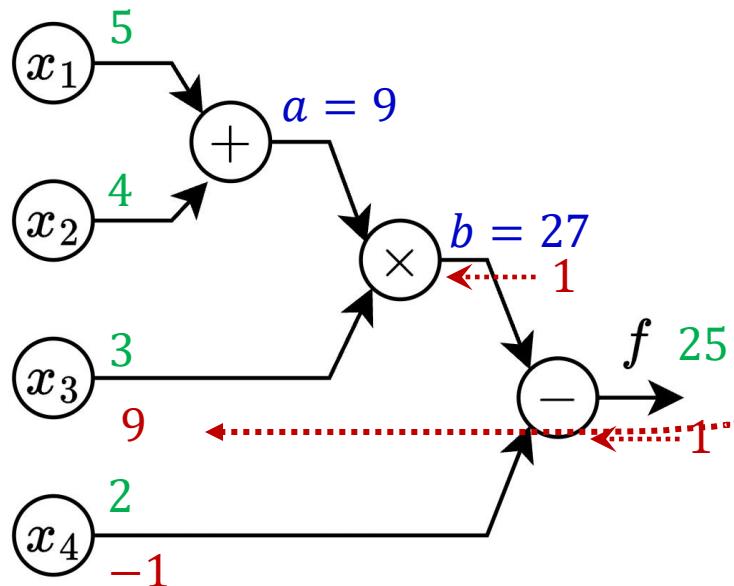
U: Upstream grad
L: Local grad

Graph: An Example



$$f(x_1, x_2, x_3, x_4, x_5) = (x_1 + x_2)x_3 - x_4$$

Evaluated at: $(x_1, x_2, x_3, x_4) = (5, 4, 3, 2)$



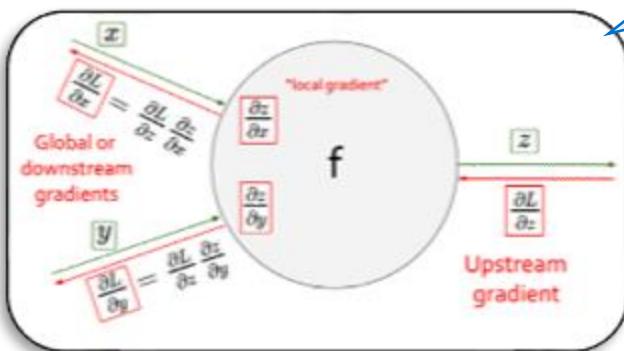
$$b = a \times x_3$$

$$\frac{\partial f}{\partial x_3} = \overbrace{\frac{\partial b}{\partial x_3}}^L \times \overbrace{\frac{\partial f}{\partial b}}^U = a \times 1 = 9$$

U: Upstream grad
L: Local grad

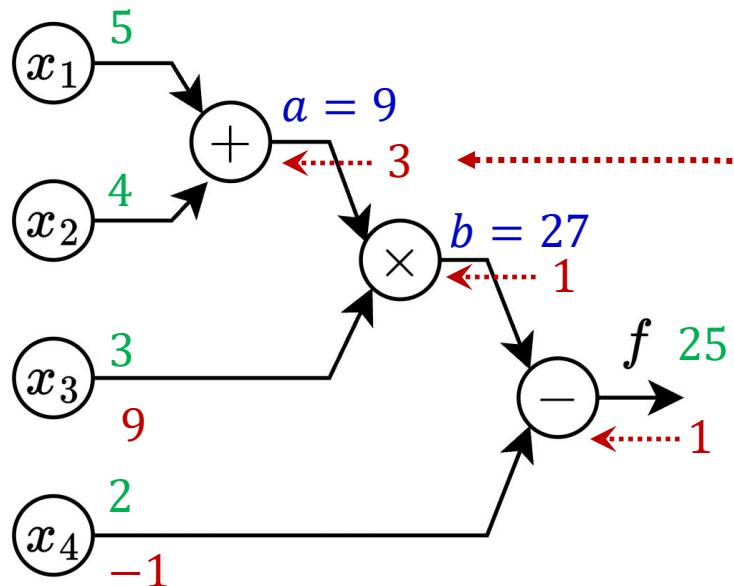
Remember this!

Graph: An Example



$$f(x_1, x_2, x_3, x_4, x_5) = (x_1 + x_2)x_3 - x_4$$

Evaluated at: $(x_1, x_2, x_3, x_4) = (5, 4, 3, 2)$



$$b = a \times x_3$$

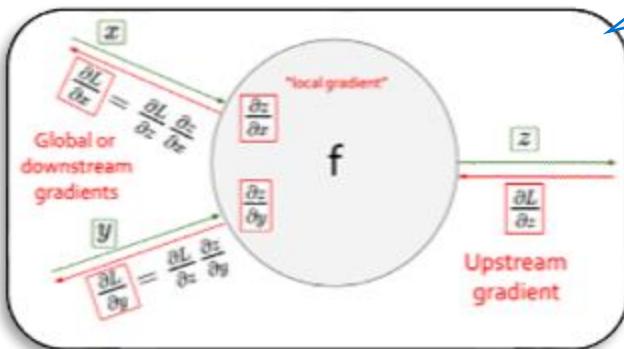
$$\frac{\partial f}{\partial x_3} = \frac{L}{\frac{\partial b}{\partial x_3}} \times \frac{U}{\frac{\partial f}{\partial b}} = a \times 1 = 9$$

$$\frac{\partial f}{\partial a} = \frac{L}{\frac{\partial b}{\partial a}} \times \frac{U}{\frac{\partial f}{\partial b}} = x_3 \times 1 = 3$$

Remember this!

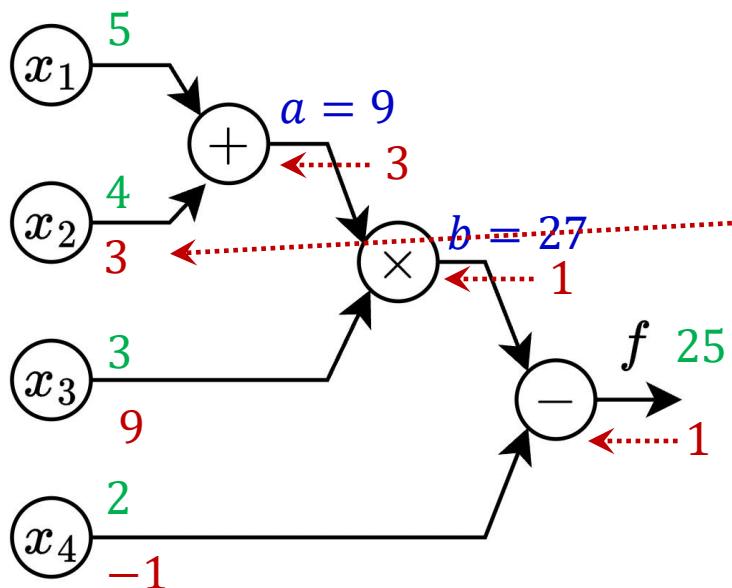
U: Upstream grad
L: Local grad

Graph: An Example



$$f(x_1, x_2, x_3, x_4, x_5) = (x_1 + x_2)x_3 - x_4$$

Evaluated at: $(x_1, x_2, x_3, x_4) = (5, 4, 3, 2)$



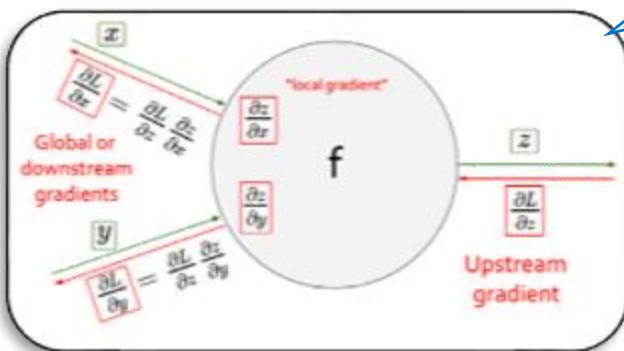
$$a = x_1 + x_2$$

$$\frac{\partial f}{\partial x_2} = \frac{\text{L}}{\frac{\partial a}{\partial x_2}} \times \frac{\text{U}}{\frac{\partial f}{\partial a}} = 1 \times 3 = 3$$

U: Upstream grad
L: Local grad

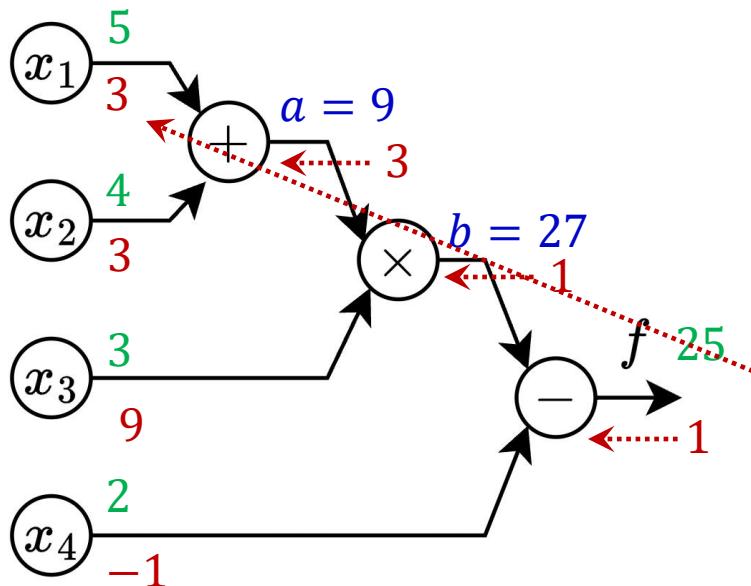
Remember this!

Graph: An Example



$$f(x_1, x_2, x_3, x_4, x_5) = (x_1 + x_2)x_3 - x_4$$

Evaluated at: $(x_1, x_2, x_3, x_4) = (5, 4, 3, 2)$



$$a = x_1 + x_2$$

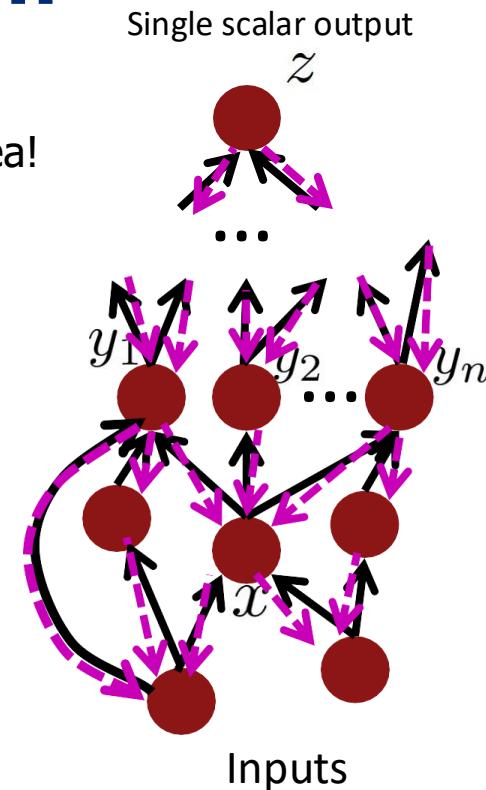
$$\frac{\partial f}{\partial x_2} = \overbrace{\frac{\partial a}{\partial x_2}}^L \times \overbrace{\frac{\partial f}{\partial a}}^U = 1 \times 3 = 3$$

$$\frac{\partial f}{\partial x_1} = \overbrace{\frac{\partial a}{\partial x_1}}^L \times \overbrace{\frac{\partial f}{\partial a}}^U = 1 \times 3 = 3$$

Backprop via Computation Graph

- What if the network does not have a regular structure? Same idea!

1. Sort the nodes in **topological order** (what depends on what)
2. Forward-Propagation:
 - Visit nodes in topological sort order and compute value of node given predecessors
3. Backward-Propagation:
 - Compute **local gradients**
 - Visit nodes in reverse order and compute **global gradients** using gradients of successors



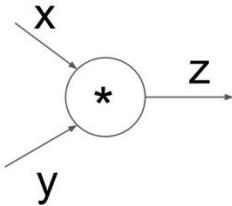
Summary

- **Computation graphs:** directed graph where the nodes correspond to mathematical operations.
 - A way of expressing mathematical operations.
- This allows general-purpose implementation of Backprop to any form of networks (not just multilayer perceptron).
 - This is why in practice you don't need to worry about implementing Backprop!! 
- **Next:** Implementing Backprop yourself + industrial software libraries.

Backprop via Automatic Differentiation

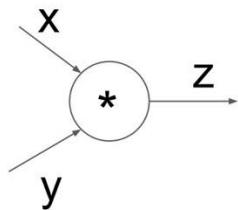
Backward propagation

- The computation graph makes it easy to backpropagate all the way
- Let's think about how this is implemented!



```
1  class Tensor:  
2      def __init__(self, value):  
3          self.value = value  
4  
5      def __add__(self, other):  
6          pass  
7  
8      def __mul__(self, other):  
9          pass  
10  
11  
12      # example  
13      a = Tensor(1)  
14      b = Tensor(2)  
15      c = a + b
```

```
1 ✓ class Tensor:
2   ✓     def __init__(self, value, children=(), _op=None, label=''):
3       self.value = value
4       self.grad = 0.0
5       self._prev = set(children)
6       self._op = _op
7       self.label = label
8
9   ✓     def __add__(self, other):
10      out = Tensor(self.value + other.value, children=(self, other), _op='+')
11      return out
12
13  ✓     def __mul__(self, other):
14      out = Tensor(self.value * other.value, children=(self, other), _op='*')
15      return out
```



```
class Tensor:
    def __init__(self, value, children=(), _op=None, label ""):
        self.value = value
        self.grad = 0.0
        self._backward = lambda: None
        self._prev = set(children)
        self._op = _op
        self.label = label

    def __mul__(self, other):
        out = Tensor(self.value * other.value, children=(self, other), _op="*")

        def _backward():
            self.grad += other.value * out.grad
            other.grad += self.value * out.grad
        out._backward = _backward
        return out
```

The computational graph
should be directed and
acyclic.

We start calling backward in
order

```
def backward(self):
    netowork = []
    visited = set()
    def build_netowork(node):
        if node not in visited:
            visited.add(node)
            for child in node._prev:
                build_netowork(child)
            netowork.append(node)
    build_netowork(self)
    self.grad = 1.0
    for node in reversed(netowork):
        node._backward()
```

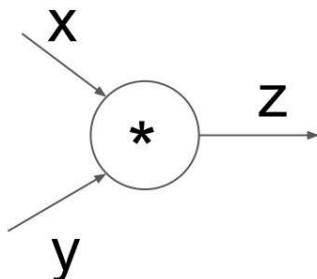


Auto-diff in PyTorch



PyTorch's Implementation: Forward/Backward API

- PyTorch has implementation of forward/backward operations for various operators.
- Example: multiplication operator



```
class Multiply(torch.autograd.Function):  
    @staticmethod  
    def forward(ctx, x, y):  
        ctx.save_for_backward(x, y) ← Need to cash some values for use in backward  
        z = x * y  
        return z  
    @staticmethod  
    def backward(ctx, grad_z): ← Upstream gradient  
        x, y = ctx.saved_tensors  
        grad_x = y * grad_z # dz/dx * dL/dz  
        grad_y = x * grad_z # dz/dy * dL/dz  
        return grad_x, grad_y ← Multiply upstream and local gradients
```

PyTorch Operators

- PyTorch's lower-level functions translate activities to graphics processor via libraries like OpenGL

| | | |
|---|---|---------------|
| mul_scalar_glsl | [vulkan] Add image format qualifier to glsl files (#69330) | last year |
| nchw_to_image.glsl | [vulkan] Enable 2D texture types (#86971) | 2 months ago |
| nchw_to_image2d.glsl | [vulkan] Enable 2D texture types (#86971) | 2 months ago |
| nchw_to_image_int32.glsl | [Vulkan] Enable copying QInt8 and QInt32 tensors from cpu to vulkan. (#...) | last month |
| nchw_to_image_int8.glsl | [Vulkan] Enable copying QInt8 and QInt32 tensors from cpu to vulkan. (#...) | last month |
| nchw_to_image_uint8.glsl | [Vulkan] Enable copying QInt8 and QInt32 tensors from cpu to vulkan. (#...) | last month |
| permute_4d.glsl | [vulkan] Add image format qualifier to glsl files (#69330) | last year |
| quantize_per_tensor_qint32.glsl | [Vulkan] Enable QInt8 and QInt32 quantization (#89788) | last month |
| quantize_per_tensor_qint8.glsl | [Vulkan] Enable QInt8 and QInt32 quantization (#89788) | last month |
| quantize_per_tensor_quint8.glsl | [Vulkan] Enable QIn8 and QInt32 quantization (#89788) | last month |
| quantized_add.glsl | [Vulkan][TCC] Fix quantized shaders (#89456) | last month |
| quantized_conv2d.glsl | [Vulkan][TCC] Fix quantized shaders (#89456) | last month |
| quantized_conv2d_dw.glsl | [Vulkan][TCC] Fix quantized shaders (#89456) | last month |
| quantized_conv2d_pw_2x2.glsl | [Vulkan][TCC] Fix quantized shaders (#89456) | last month |
| quantized_div.glsl | [Vulkan][TCC] Fix quantized shaders (#89456) | last month |
| quantized_mul.glsl | [Vulkan][TCC] Fix quantized shaders (#89456) | last month |
| quantized_sub.glsl | [Vulkan][TCC] Fix quantized shaders (#89456) | last month |
| quantized_upsample_nearest2d.glsl | [Vulkan][TCC] Fix quantized shaders (#89456) | last month |
| reflection_pad2d.glsl | [vulkan] Add image format qualifier to glsl files (#69330) | last year |
| replication_pad2d.glsl | [vulkan] replication_pad2d.glsl; use clamp() instead of min(max()) (#...) | 7 months ago |
| select_depth.glsl | [Vulkan] Implement select/int operator (#81771) | 5 months ago |
| sigmoid.glsl | [vulkan] Add image format qualifier to glsl files (#69330) | last year |
| sigmoid_glsl | [vulkan] Add image format qualifier to glsl files (#69330) | last year |
| slice_4d.glsl | [vulkan] Add image format qualifier to glsl files (#69330) | last year |
| softmax.glsl | [vulkan] Add image format qualifier to glsl files (#69330) | last year |
| stack_feature.glsl | [Vulkan] Implement Stack operator (#81064) | 5 months ago |
| sub.glsl | [Vulkan] Implement arithmetic ops where one of the arguments is a ten... | 5 months ago |
| sub_glsl | [Vulkan] Implement arithmetic ops where one of the arguments is a ten... | 5 months ago |
| tanh.glsl | [vulkan] Clamp tanh activation op input to preserve numerical stabili... | 10 months ago |
| tanh_glsl | [vulkan] Clamp tanh activation op input to preserve numerical stabili... | 10 months ago |
| threshold.glsl | [vulkan] fix some broken tests in vulkan_api_test (#80962) | 6 months ago |
| upsample_nearest2d.glsl | [vulkan] Add image format qualifier to glsl files (#69330) | last year |

Example Activation Functions

master · pytorch / aten / src / ATen / native / vulkan / glsl / tanh.glsl

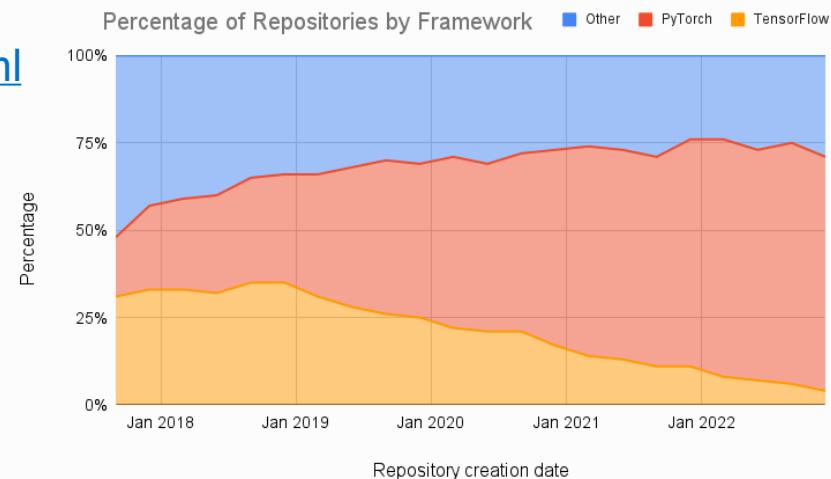
SS-JIA [vulkan] Clamp tanh activation op input to preserve numerical stabili... ...
By 2 contributors

27 lines (21 sloc) | 777 Bytes

```
1 #version 450 core
2 #define PRECISION $precision
3 #define FORMAT    $format
4
5 layout(std430) buffer;
6
7 /* Qualifiers: layout - storage - precision - memory */
8
9 layout(set = 0, binding = 0, FORMAT) uniform PRECISION restrict writeonly image3D  uOutput;
10 layout(set = 0, binding = 1)           uniform PRECISION                         sampler3D uInput;
11 layout(set = 0, binding = 2)           uniform PRECISION restrict          Block {
12     ivec4 size;
13 } uBlock;
14
15 layout(local_size_x_id = 0, local_size_y_id = 1, local_size_z_id = 2) in;
16
17 void main() {
18     const ivec3 pos = ivec3(gl_GlobalInvocationID);
19
20     if (all(lessThan(pos, uBlock.size.xyz))) {
21         const vec4 intex = texelFetch(uInput, pos, 0);
22         imageStore(
23             uOutput,
24             pos,
25             tanh(clamp(intex, -15.0, 15.0)));
26     }
27 }
```

Check out PyTorch Documentations

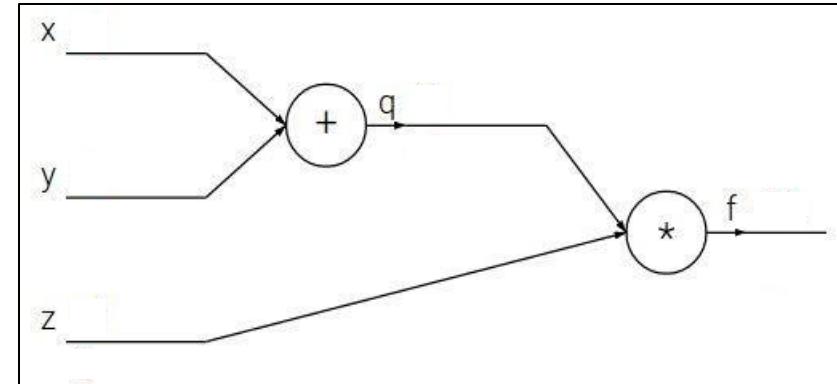
- This is the main library the vast majority of the community uses.
 - It contains hundreds of mathematical operations with “backward()” function to allow automatic gradient computation on computation graph.
-
- See: <https://pytorch.org/docs/stable/index.html>



Backprop in PyTorch

$$f(x, y, z) = (x + y)z$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



```
x = torch.tensor(-2.0, requires_grad=True)
y = torch.tensor(5.0, requires_grad=True)
z = torch.tensor(-4.0, requires_grad=True)

f = (x+y)*z # Define the computation graph

f.backward() # PyTorch's internal backward gradient computation

print('Gradients after backpropagation:', x.grad, y.grad, z.grad)
```

Why Learn All These Details About Backprop?

- Modern deep learning frameworks compute gradients for you!
- But why take a class on compilers or systems when they are implemented for you?
 - Understanding what is going on under the hood is useful!
- Backpropagation doesn't always work perfectly out of the box
 - Understanding why is crucial for debugging and improving models

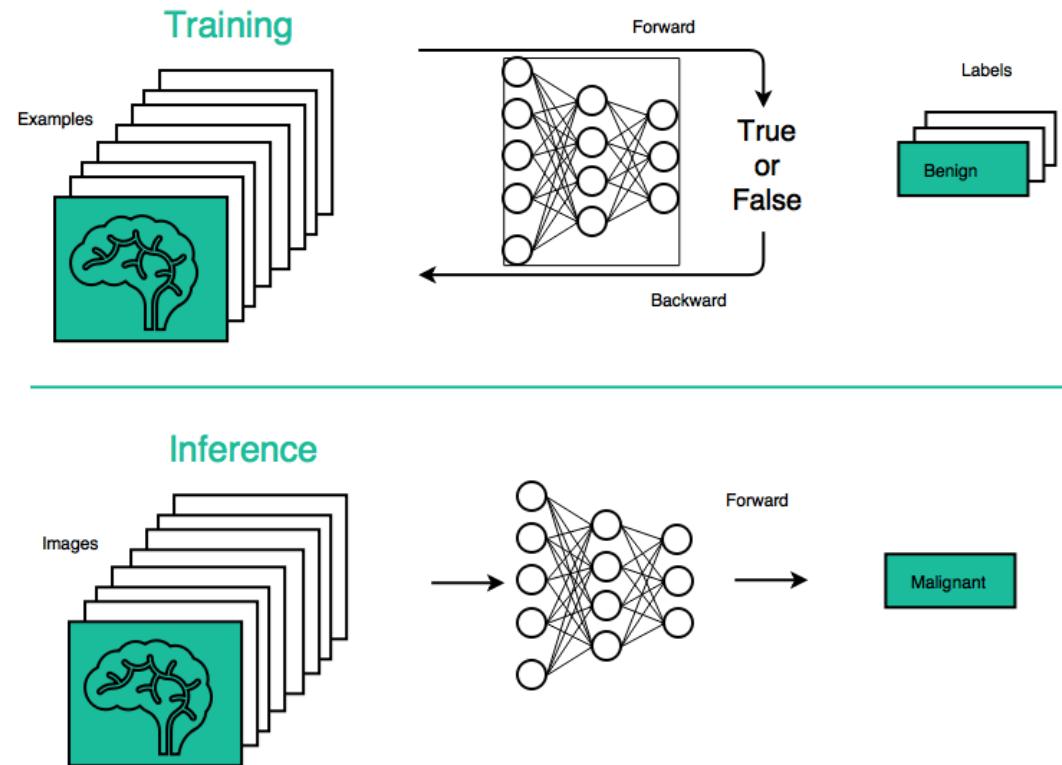
Summary

- Modern deep learning libraries such as PyTorch implement a vast library of operations to allow automatic and efficient Backprop.
- We will make extensive use of PyTorch in this class (yay!)
- Next: We will discuss a few practical considerations regarding training NNs.

Practical considerations for training neural nets

Batching

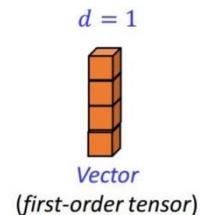
- GPUs are **fast** with **Tensor operations**
- Rather than visiting instances in sequentially , **batch them together** for **faster** training and inference.



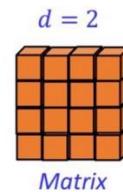
Batches of Data: Example

- The case of natural language:

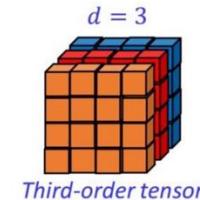
- Each word is mapped to a vector \mathbb{R}^d



- Then, each sentence of length ℓ is mapped to a matrix $\mathbb{R}^{\ell \times d}$



- A batch of sentences (size b) is mapped to a tensor $\mathbb{R}^{\ell \times d \times b}$



Batches of Data, In Practice

- PyTorch makes it easy to batch data.
 - All its functionalities are designed around batched process.
 - For example, you can create any tensor of **any** dimension.

TORCH.RAND

```
torch.rand(*size, *, generator=None, out=None, dtype=None, layout=torch.strided, device=None,  
          requires_grad=False, pin_memory=False) → Tensor
```



Returns a tensor filled with random numbers from a uniform distribution on the interval [0, 1)

The shape of the tensor is defined by the variable argument `size`.

Parameters

`size (int...)` – a sequence of integers defining the shape of the output tensor. Can be a variable number of arguments or a collection like a list or tuple.

Batches of Data, In Practice

- Avoid loops, use tensors.

```
import torch

def matmul(A, B):
    C = torch.zeros_like(A)
    for i in range(A.size(0)):
        for j in range(B.size(1)):
            for k in range(A.size(1)):
                C[i, j] += A[i, k] * B[k, j]
    return C

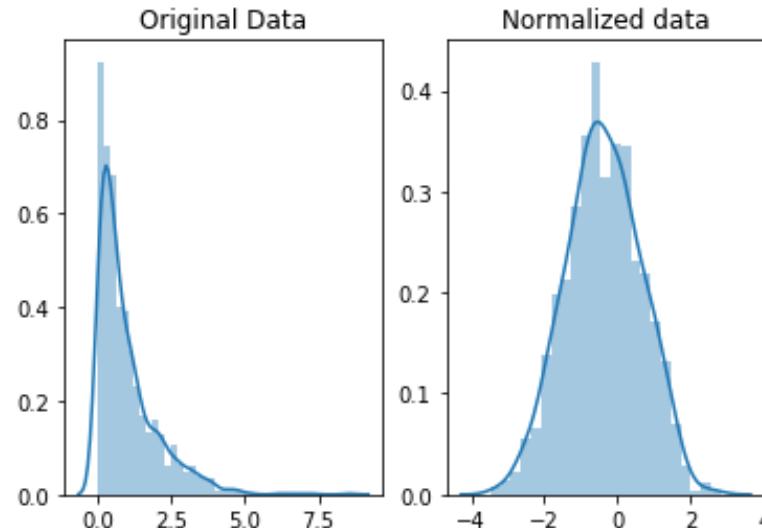
# Example usage:
A = torch.randn(10, 10)
B = torch.randn(10, 10)
C = matmul(A, B)
```

```
import torch

# Example usage:
A = torch.randn(10, 10)
B = torch.randn(10, 10)
C = torch.matmul(A, B)
```

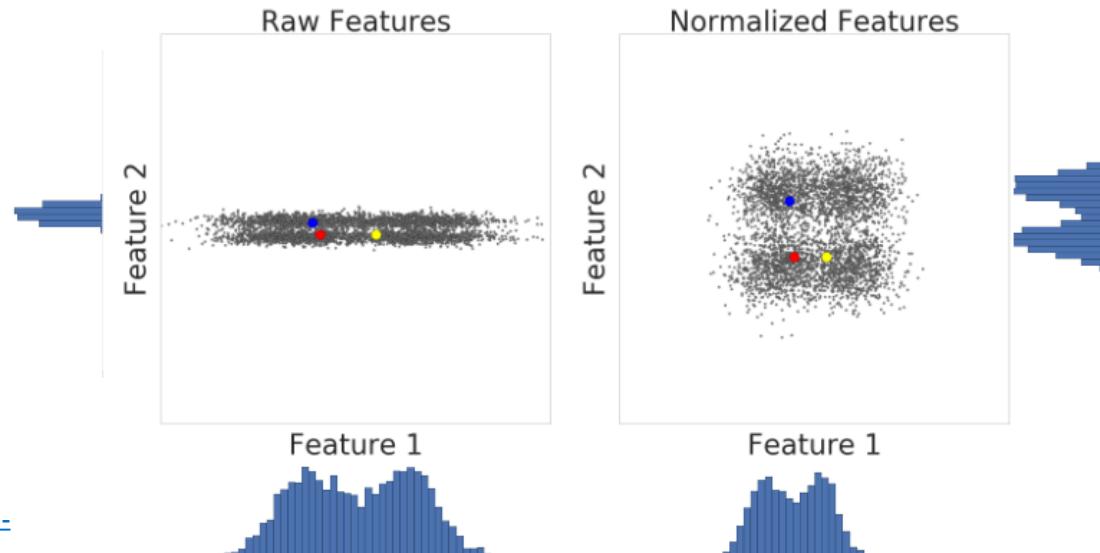
Normalize Your Data!

- We do not like very large numbers.
 - Large numbers lead to numerical problems (e.g., overflow) and lead to NaNs 😱
- We prefer if our data is distributed around zero.



Normalize Your Data!

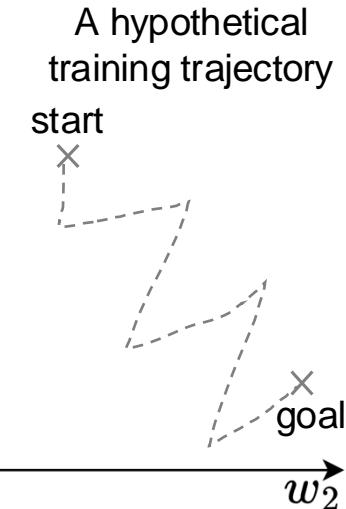
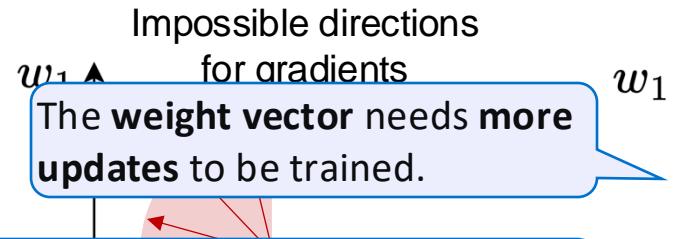
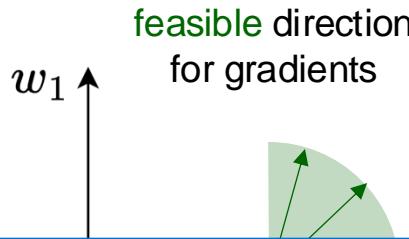
- We do not like very large numbers.
 - Large numbers lead to numerical problems (e.g., overflow) and lead to NaNs 😱
- We prefer if our data is distributed around zero.



Non-Zero-Centered Data

$$f = \mathbf{w}^\top \mathbf{x} + b \Rightarrow \frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial f} \frac{\partial f}{\partial w_i} = \text{upstream} \times x_i$$

- If data is always positive (i.e., $\forall i: x_i > 0$), all the dimensions of $\nabla_w \mathcal{L}$ would have the same sign (all positive or all negative, same sign as upstream).



Solution:

- Inter-leave **normalization operators** to normalize data around zero.
- Choose activation functions that are centered around zero.

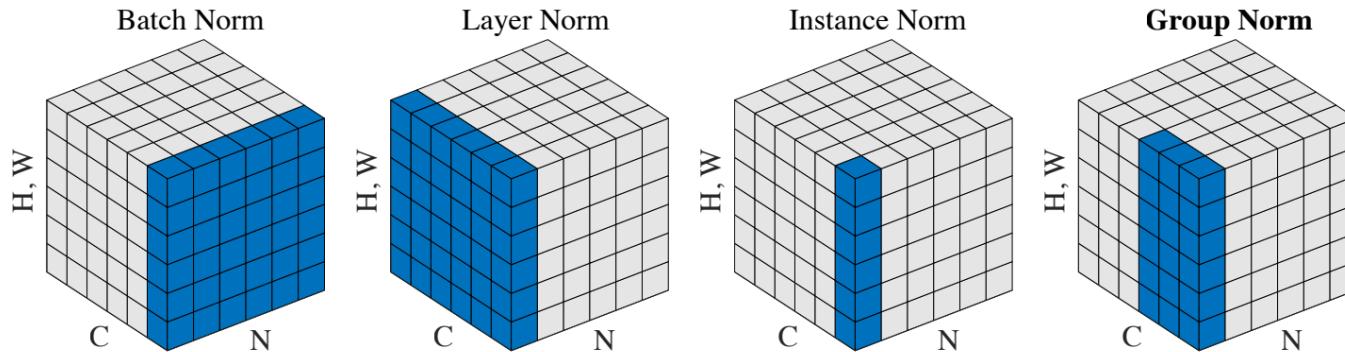


Normalization: Layer, Batch, ...

- Normalization of values standardizes the ranges of values
- Prevents value disparities
- Stabilizes and speeds up training

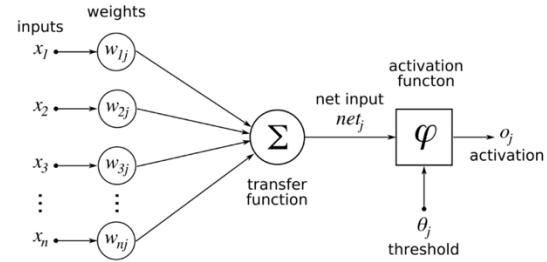
$$y = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} * \gamma + \beta$$

See PyTorch documentations: <https://pytorch.org/docs/stable/nn.html#normalization-layers>



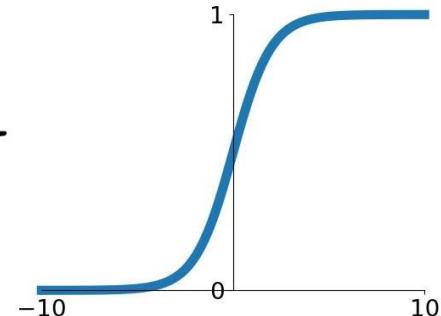
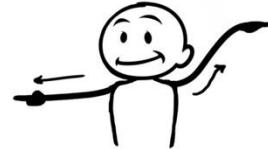
Activation Functions

- How do you choose what activation function to use?
- In general, it is problem-specific and might require trial-and-error.
- Here are some tips about popular action functions.



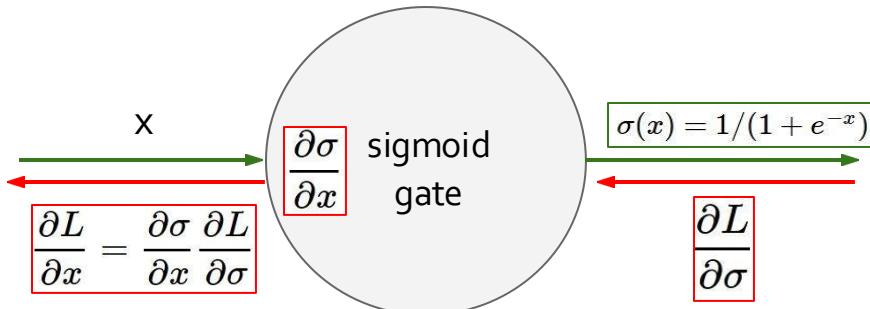
Activation Functions : Sigmoid

- Squashes numbers to range [0, 1]
- Historically popular, interpretation as “firing rate” of a neuron
- Key limitation:** Saturated neurons “kill” the gradients
- Whenever $|x| > 5$, the gradients are basically zero.

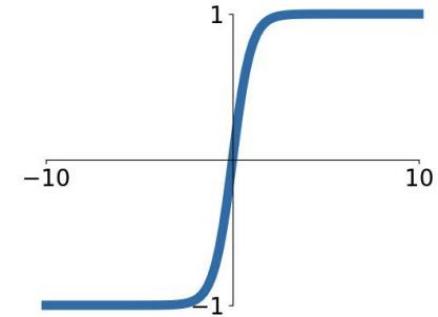


$$\sigma(x) = 1/(1 + e^{-x})$$

If all the gradients flowing back will be zero and weights will never change.



Activation Functions : Tanh



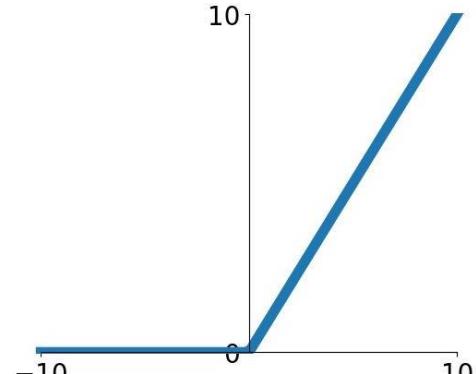
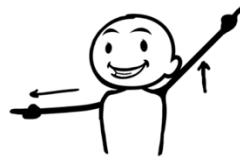
tanh(x)

[LeCun et al., 1991]

- Symmetric around $[-1, 1]$
- Still saturates $|x| > 3$ and “kill” the gradients
- Zero-centered — faster optimization (why?)

Activation Functions : ReLU

- Computationally efficient
- In practice, converges faster than sigmoid/tanh in practice
- Does not saturate (in +region) — will die less!



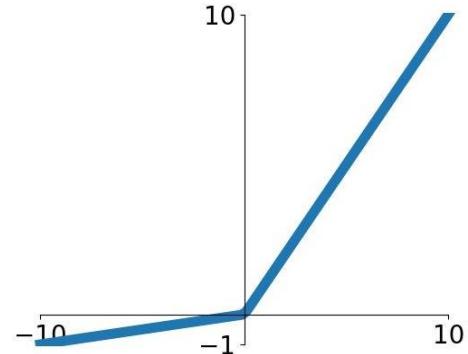
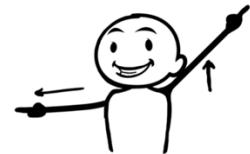
ReLU

(Rectified Linear Unit)

[Krizhevsky et al., 2012]

Activation Functions : Leaky ReLU

- Does not saturate — will not die.
- Computationally efficient
- In practice it converges faster than sigmoid/tanh in practice



$$f(x) = \max(0.01x, x)$$

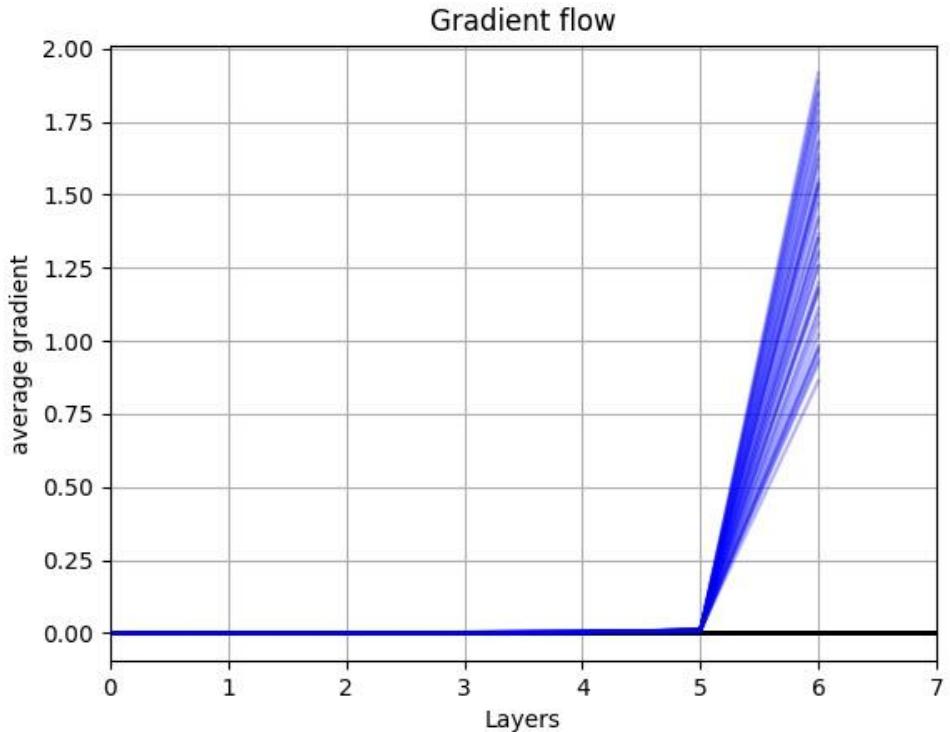
- Other parametrized variants:
 - Parametric Rectifier (PReLU): $f(x) = \max(\alpha x, x)$ [He et al., 2015]
 - Maxout: $\max(w_1^T x + b_1, w_2^T x + b_2)$ [Goodfellow et al., 2013]
- Provide more flexibility, though at the cost of more learnable parameters.
 - For example, Maxout doubles the number of parameters.

Choose Activations: In Practice

- In general, it is problem-specific and might require trial-and-error.
- A useful recipe:
 1. Generally, ReLU is a good activation to start with.
 2. Time/compute permitting, you can try other activations to squeeze out more performance.

Exploding/Vanishing Gradients

- If many numbers $|x| > 1$ get multip
- NaN gradients --> no learning!
- If many numbers $|x| < 1$ get multip
- Zero gradients -> no learning!



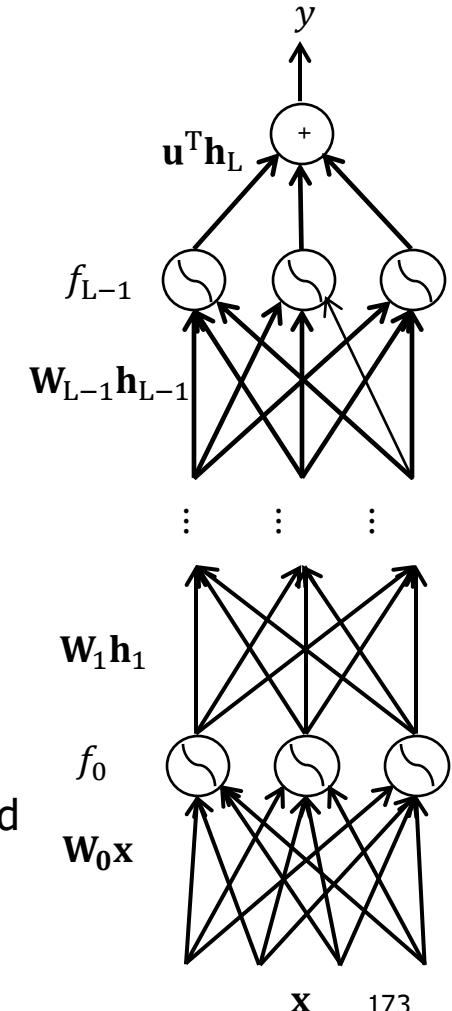
Exploding/Vanishing Gradients

- Remember gradient computation at layer $L - k$:

$$\nabla_{\mathcal{L}}(\mathbf{W}_{L-k}) = \left(\mathbf{J}_\ell(y) \mathbf{J}_y(\mathbf{h}_L) \mathbf{J}_{\mathbf{h}_L}(\mathbf{h}_{L-1}) \mathbf{J}_{\mathbf{h}_{L-1}}(\mathbf{h}_{L-2}) \dots \mathbf{J}_{\mathbf{h}_{L-k+1}}(\mathbf{h}_{L-k}) \right)^T$$

O(k)-many matrix multiplication

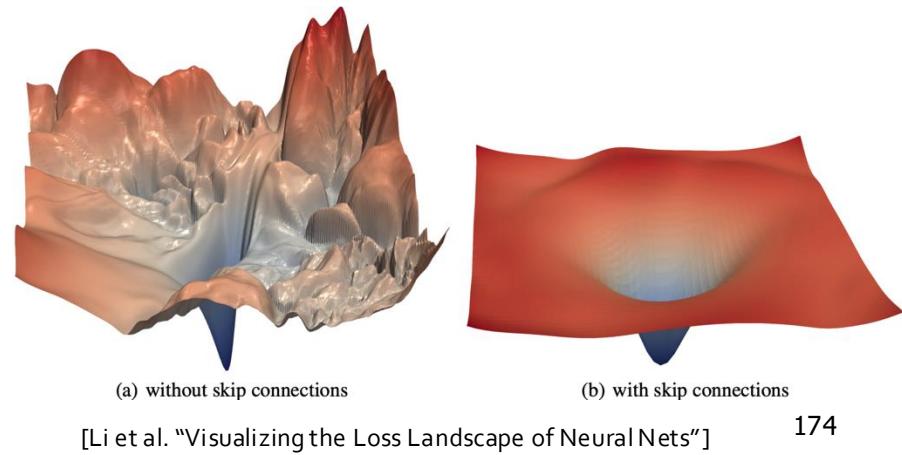
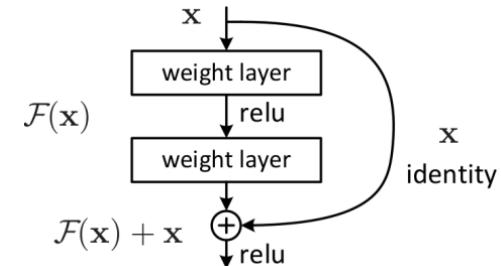
- This matrix multiplication could quickly approach
 - ∞ , if the matrix elements are large — exploding gradients.
 - 0, if the matrix elements are small — vanishing gradients.
 - $\infty/0$ gradients would kill learning (no flow of information).**
- For those interested, convergences of matrix powers is determined by its largest eigenvalue (HW, extra credit).



Residual Connections/Blocks

- Create direct “information highways” between layers.
- Shown to **diminish vanishing/exploding** gradients
- Early in the training, there are fewer layers to propagate through.
 - The network would restore the skipped layers, as it learns richer features.
 - It is also shown to make the optimization objective smoother.

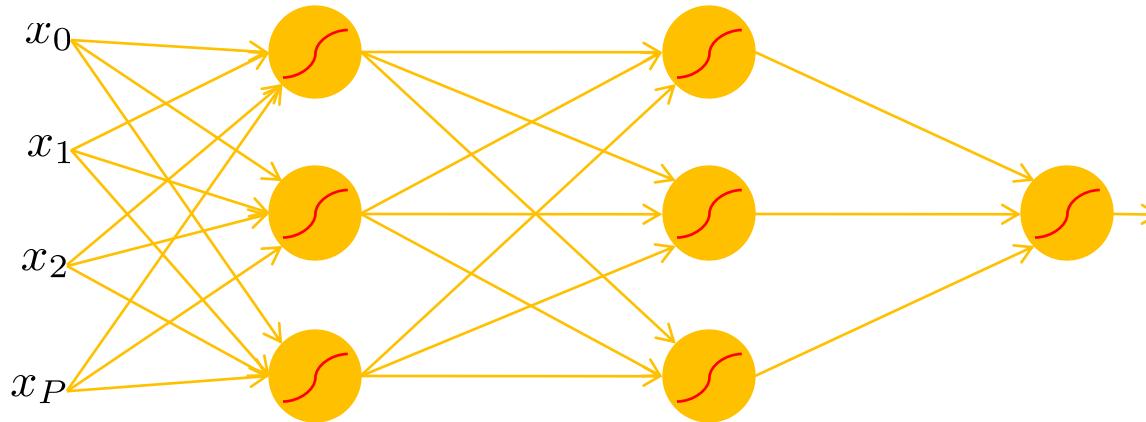
[Fun fact: [the paper](#) (He et al. 2015) introducing residual layers is the most cited paper of century!!]



[Li et al. “Visualizing the Loss Landscape of Neural Nets”]

Weight Initialization

- Initializing all weights with a **fixed constant** (e.g., 0's) is a very **bad idea!** (why?)



- If the neurons start with the same weights, then all the neurons will follow the same gradient, and will always end up doing the same thing as one another.
- Effective initialization is one that breaks such "symmetries" in the weight space.

Weight Initialization

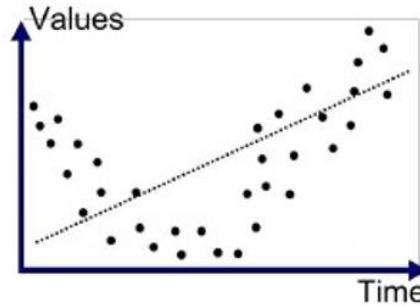
- Better idea: initialize weights with random Gaussian noise.

```
x = torch.tensor.empty(3, 5)  
nn.init.normal_(w)
```

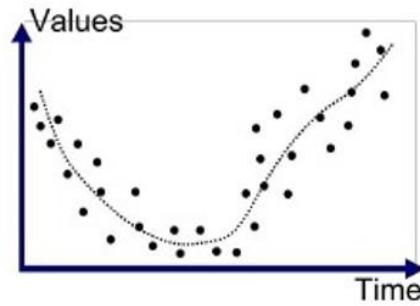
- There are fancier initializations (Xavier, Kaiming, etc.) that we won't get into.

Over-training/-fitting Prevention

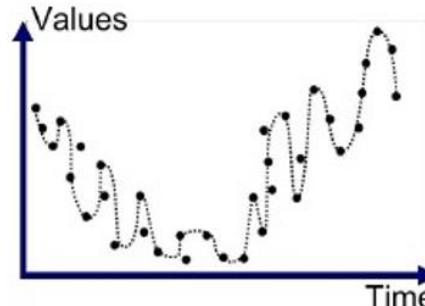
- Running too many epochs and/or a NN with many hidden layers may lead to an **overfit** network
- Keep a **held-out validation** set and evaluate accuracy after every epoch
- **Early stopping:** maintain weights for best performing network on the validation set and return it when performance decreases significantly beyond that.



Underfitted



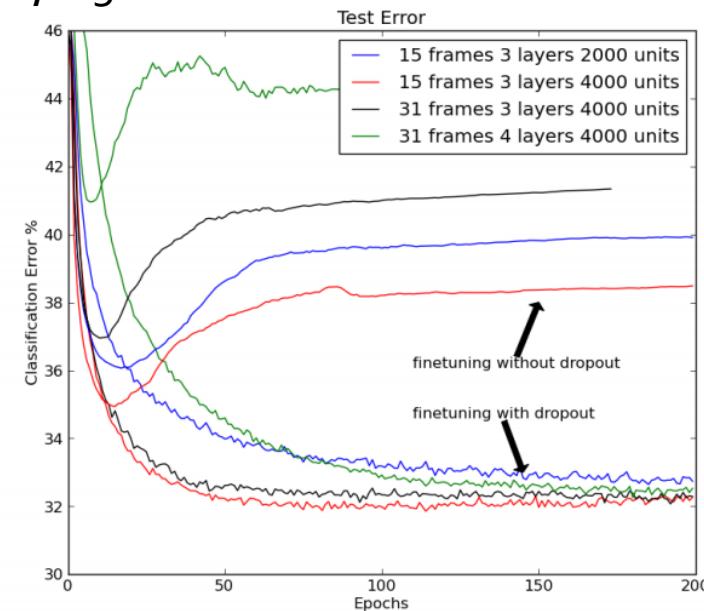
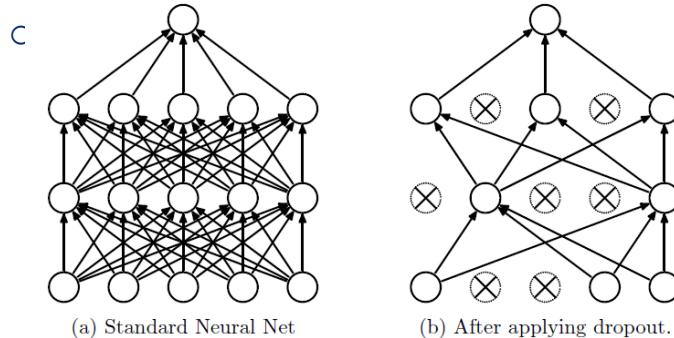
Good Fit/Robust



Overfitted

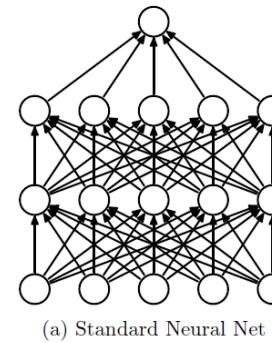
Dropout Training

- In each forward pass, randomly set some neurons to zero
- Probability of dropping is a **hyperparameter**; 0.5 is common
- Dropout is **implicitly an ensemble** (average) o
 - Each binary mask is one model
 - For example, a layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!

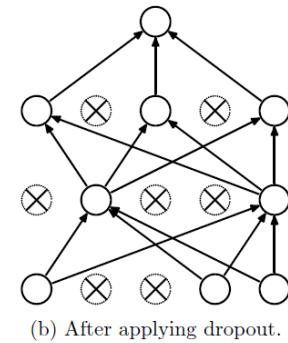


Dropout During Test Time

- The issue for the **test** time:
 - Dropout adds randomization. 😞
 - Each dropout mask would lead to a slightly different outcome.
- In ideal world, we would like to “average out” the outcome across all the possible random masks:
 - Not feasible.
 - Remember the example: a layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!
 - Only $\sim 10^{82}$ atoms in the universe ...



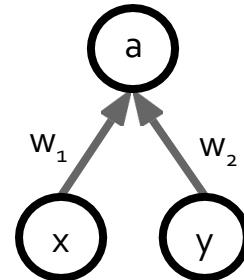
(a) Standard Neural Net



(b) After applying dropout.

Dropout During Test Time (2)

- The alternative is to **not apply dropout**.
- Without dropout, the input values to each neuron would be higher than what was seen during the training (**mismatch between train/test**).
- **Example:** imagine we apply dropout ($p=0.5$) to the following model:
 - Training time: $E[a] = \frac{1}{4}(w_1x_1 + w_2x_2) + \frac{1}{4}(0 + 0) + \frac{1}{4}(0 + w_2x_2) + \frac{1}{4}(w_1x_1 + 0) = \frac{1}{2}(w_1x_1 + w_2x_2)$
 - Test time: $E[a] = w_1x_1 + w_2x_2$
- **Solution:** **scale the values** proportional to dropout probability.
 - Can be applied in either testing (scaling down) or training (scaling up).
 - A very common interview question! ☺



Dropout in Practice

Just call the PyTorch function!

It automatically

- activates the dropout for **training**.
- deactivates it during **evaluations** and scales the values according to its parameter.

```
dropout = nn.Dropout(p=0.2)
x = torch.randn(20, 16)
y = dropout(x)
```

```
# training step
...
model.train()
...
```

```
# evaluate model:
...
model.eval()
...
```

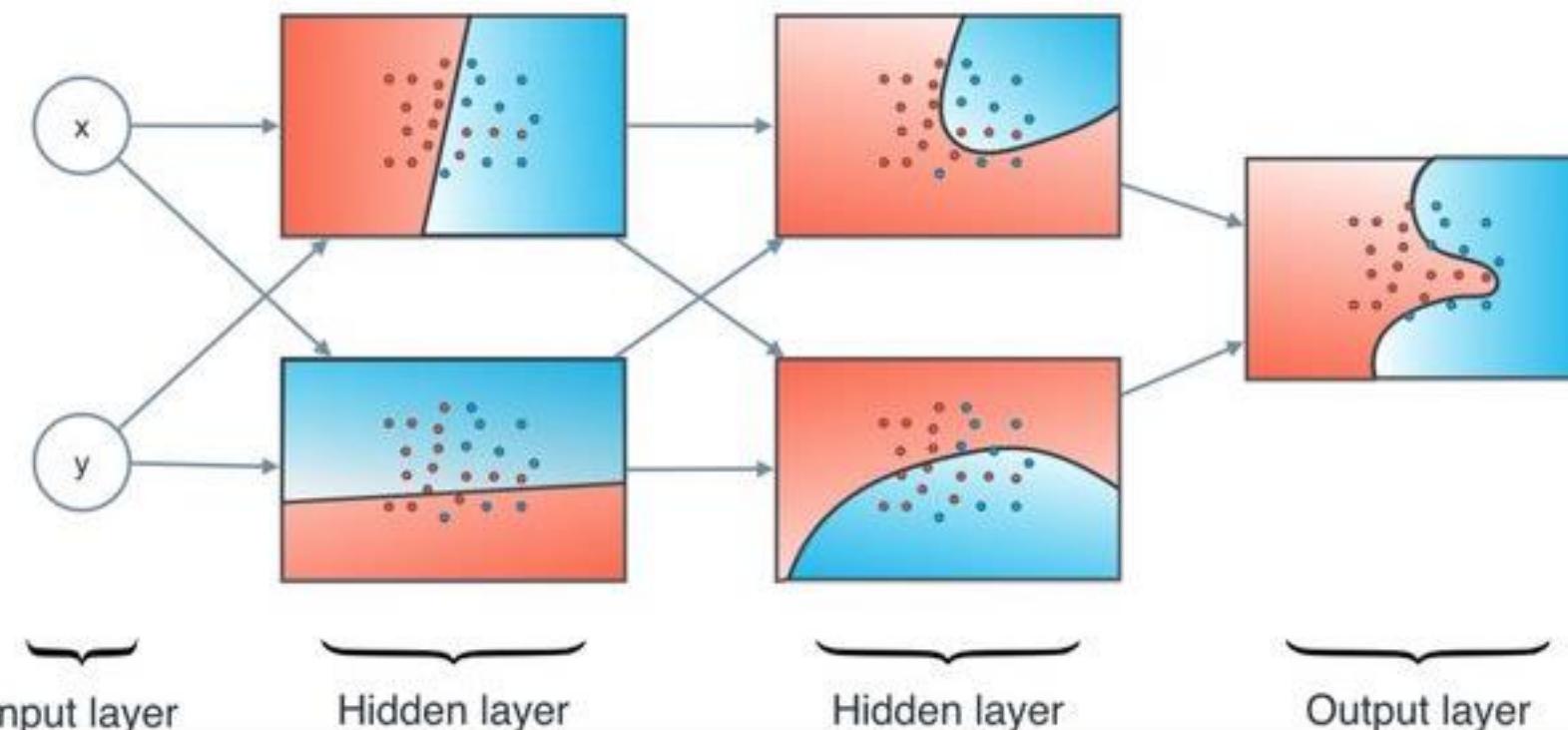
The Only Time You Want to Overfit: The First Tryout

- A model with buggy implementation (e.g., incorrect gradient calculations or updates) **cannot learn anything**.
- Therefore, a good and easy sanity check is to see if you can overfit few examples.
 - This is really the first test you should do, before any hyperparameter tuning.
- Try to train to 100% training accuracy/performance on a small sample (<30) of training data and monitor the **training** loss trends.
 - Does it down? If not, something must be wrong.
 - Try checking the **learning rate** or modifying the initialization.
 - If those don't help, check the gradients.
 - If they're **NaN** or **Inf**, might indicate **exploding gradients**.
 - If they're **zeros**, might indicate **vanishing gradients**.

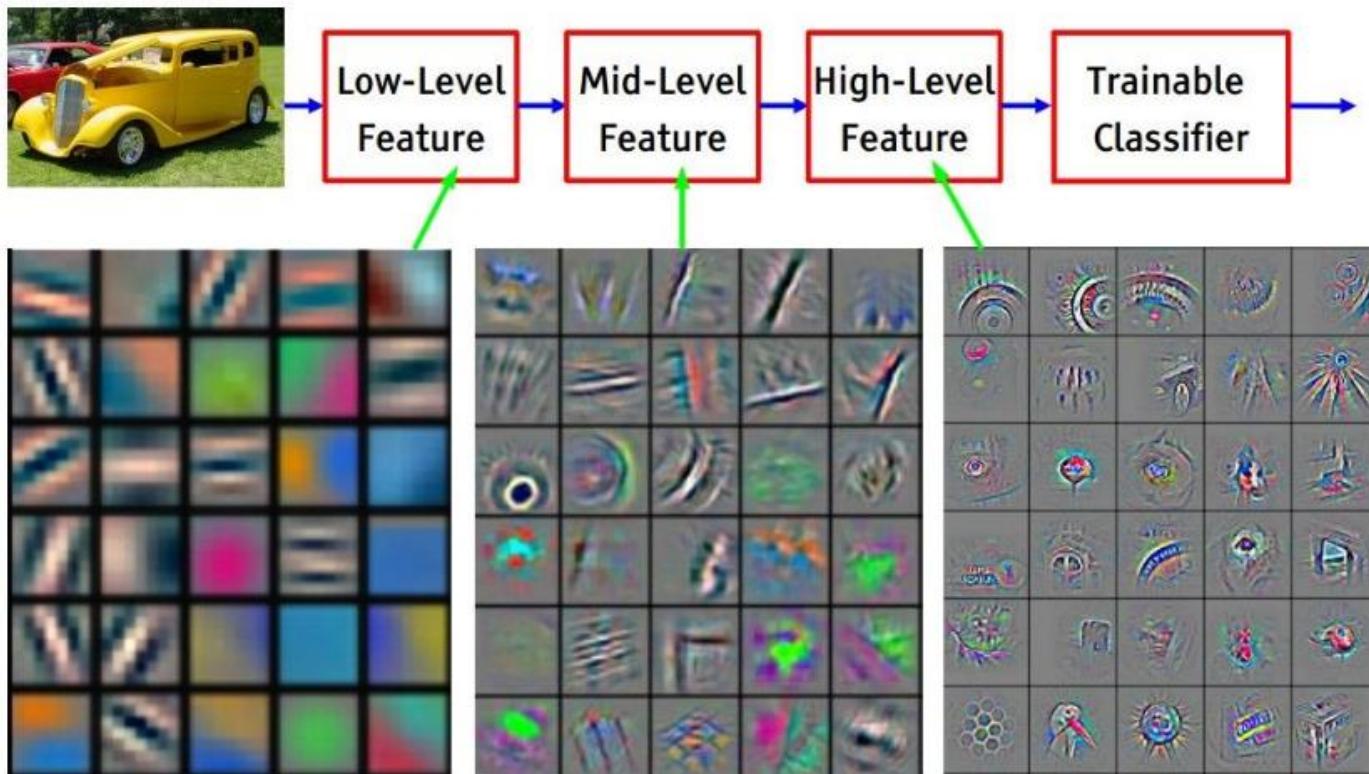
Additional Comments on Training

- No guarantee of convergence; neural networks form non-convex functions with multiple local minima
- In practice, many large networks can be trained on large data.
- Many steps (tens of thousands) may be needed for adequate training.
- May be tricky to set learning rate or number of hidden units/layers.
- To avoid local minima: several trials with different random initial weights with majority or voting techniques

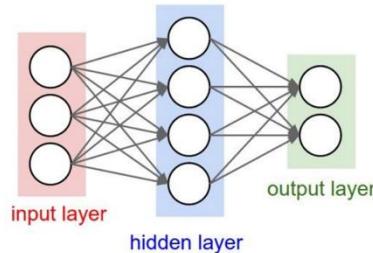
Intuition about Neural Net Representations



Intuition about Neural Net Representations



Summary



- Feed-forward network architecture
 - But many of the concepts here hold for any architecture.
- We learned Backprop, a general-purpose algorithm for efficient training of NNs.
 - Recursively (and hence efficiently) apply the chain-rule along computation graph.
 - The most important algorithm in neural networks! 🎉
- Lots of empirical tricks for training neural networks:
 - Things to be careful about: over-fitting, activations, exploding/vanishing gradients, ...