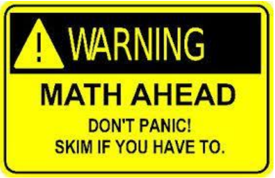



# CSC413/2516 Lecture 2: Multilayer Perceptrons & Backpropagation

Bo Wang

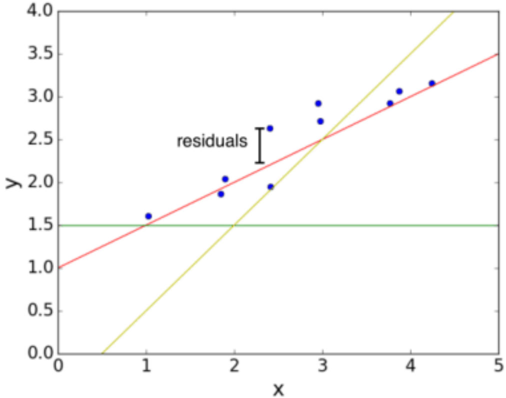
# Course information

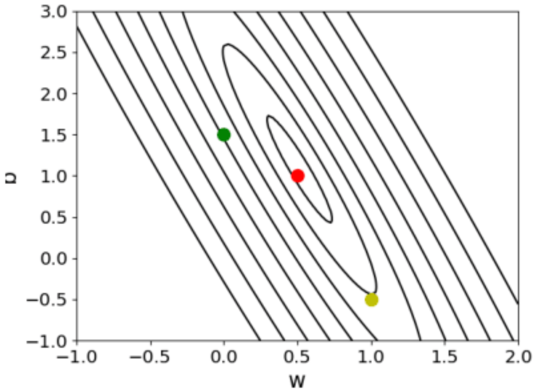
- Expectations and marking
  - Written homeworks (30% of total mark)
    - first homework is out, due 1/28
    - 2-3 short conceptual questions
    - Use material covered up through Tuesday of the preceding week
  - 4 programming assignments (40% of total mark)
    - Python, PyTorch
    - 10-15 lines of code
    - may also involve some mathematical derivations
    - give you a chance to experiment with the algorithms
  - Exams
    - midterm (10%), Feb, 11 , covering the first 4 lectures
    - final project (20%)
- See Course Information handout for detailed policies

# Course information

- Final Projects **(undergrad and grad students)**
  - Form a group: 2-3 persons
    - Undergrads can collaborate with grad students
    - Contributions have to be stated in the final report
    - Students from different backgrounds are encouraged to form a group
  - Proposal
    - One-page summary of the main topics
    - Deadline: TBD
  - Final report
    - tutorial (How to Write a Good Course Project Report , Feb 08)
    - 4 pages (excluding references)
    - Open review format
    - Deadline: TBD

# Recap: Linear Classification and Gradient Descent

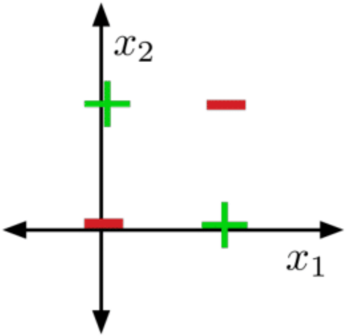



- Advantages: Easy to understand and implement; Widely-adopted;

# Limits of Linear Classification

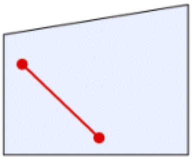
- Single neurons (linear classifiers) are very limited in expressive power.
- **XOR** is a classic example of a function that's not linearly separable.




- There's an elegant proof using convexity.

# Limits of Linear Classification

**Convex Sets**



- A set $\mathcal{S}$ is convex if any line segment connecting points in $\mathcal{S}$ lies entirely within $\mathcal{S}$. Mathematically,

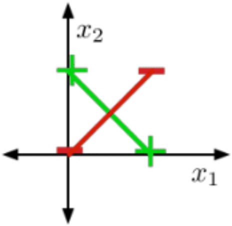
$$\mathbf{x}_1, \mathbf{x}_2 \in \mathcal{S} \implies \lambda \mathbf{x}_1 + (1 - \lambda)\mathbf{x}_2 \in \mathcal{S} \quad \text{for } 0 \leq \lambda \leq 1.$$

- A simple inductive argument shows that for $\mathbf{x}_1, \ldots, \mathbf{x}_N \in \mathcal{S}$, weighted averages, or convex combinations, lie within the set:

$$\lambda_1 \mathbf{x}_1 + \cdots + \lambda_N \mathbf{x}_N \in \mathcal{S} \quad \text{for } \lambda_i > 0,\ \lambda_1 + \cdots \lambda_N = 1.$$

**Showing that XOR is not linearly separable**

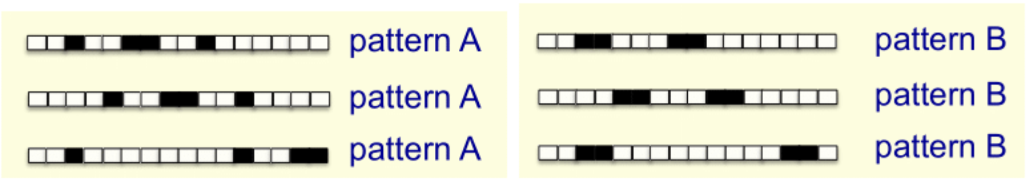
- Half-spaces are obviously convex.
- Suppose there were some feasible hypothesis. If the positive examples are in the positive half-space, then the green line segment must be as well.
- Similarly, the red line segment must line within the negative half-space.




- But the intersection can't lie in both half-spaces. Contradiction!

# Limits of Linear Classification

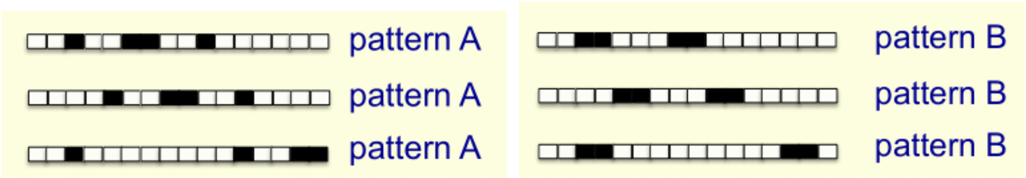
## A more troubling example




- These images represent 16-dimensional vectors. White = 0, black = 1.
- Want to distinguish patterns A and B in all possible translations (with wrap-around)
- Translation invariance is commonly desired in vision!

# Limits of Linear Classification

## A more troubling example




- These images represent 16-dimensional vectors. White $=0$, black $=1$.
- Want to distinguish patterns A and B in all possible translations (with wrap-around)
- Translation invariance is commonly desired in vision!
- Suppose there's a feasible solution. The average of all translations of A is the vector $(0.25, 0.25, \ldots, 0.25)$. Therefore, this point must be classified as A.
- Similarly, the average of all translations of B is also $(0.25, 0.25, \ldots, 0.25)$. Therefore, it must be classified as B. Contradiction!

Credit: Geoffrey Hinton

# Limits of Linear Classification

- Sometimes we can overcome this limitation using feature maps, just like for linear regression. E.g., for **XOR**:

$$\psi(\mathbf{x}) = \begin{pmatrix} x_1 \\ x_2 \\ x_1 x_2 \end{pmatrix}$$

| $x_1$ | $x_2$ | $\phi_1(\mathbf{x})$ | $\phi_2(\mathbf{x})$ | $\phi_3(\mathbf{x})$ | $t$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 |

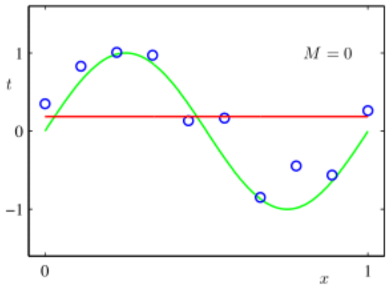
- This is linearly separable. (Try it!)
- Not a general solution: it can be hard to pick good basis functions. Instead, we'll use neural nets to learn nonlinear hypotheses directly.

# Feature maps

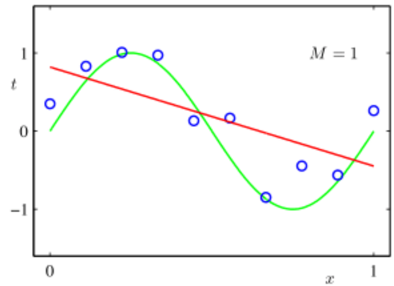
- We can convert linear models into nonlinear models using feature maps.
$$y = \mathbf{w}^\top \phi(\mathbf{x})$$
- E.g., if $\psi(x) = (1,\ x,\ \cdots, x^D)^\top$, then $y$ is a polynomial in $x$. This model is known as polynomial regression:
$$y = w_0 + w_1 x + \cdots + w_D x^D$$
- This doesn't require changing the algorithm — just pretend $\psi(x)$ is the input vector.
- We don't need an expicit bias term, since it can be absorbed into $\psi$.
- Feature maps let us fit nonlinear models, but it can be hard to choose good features.
  - Before deep learning, most of the effort in building a practical machine learning system was feature engineering.

# Feature maps

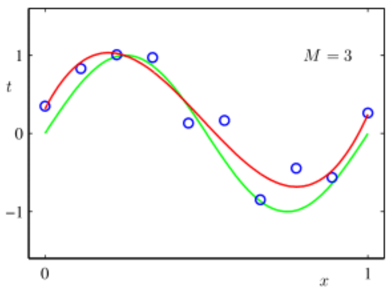
$y = w_0$



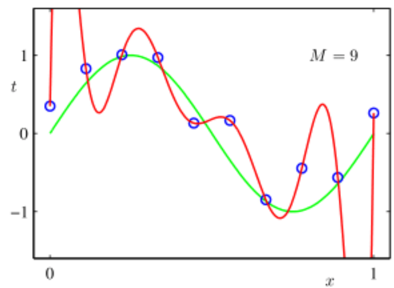

$y = w_0 + w_1 x$




$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3$



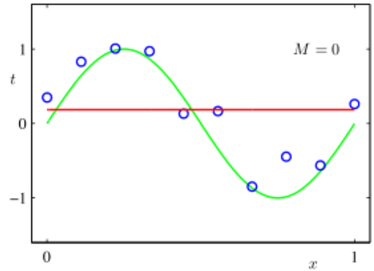

$y = w_0 + w_1 x + \cdots + w_9 x^9$



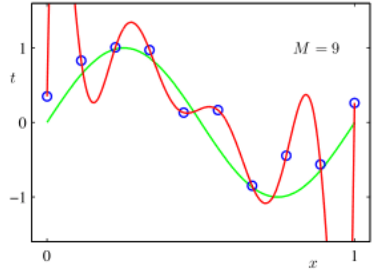

-Pattern Recognition and Machine Learning, Christopher Bishop.

# Generalization

Underfitting : The model is too simple - does not fit the data.

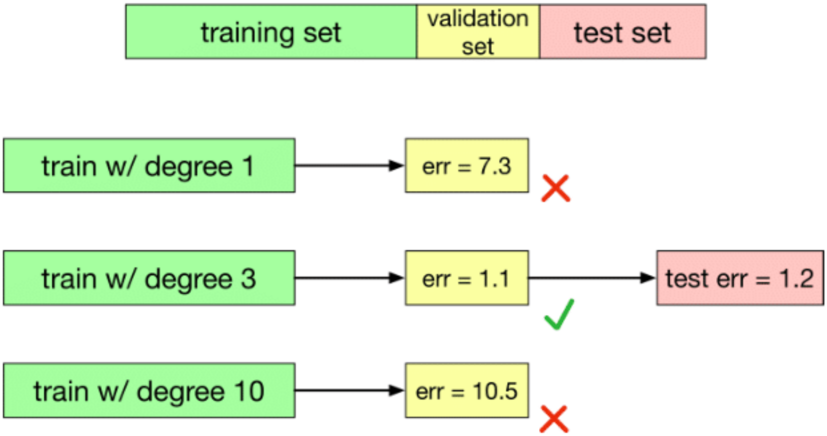



Overfitting : The model is too complex - fits perfectly, does not generalize.

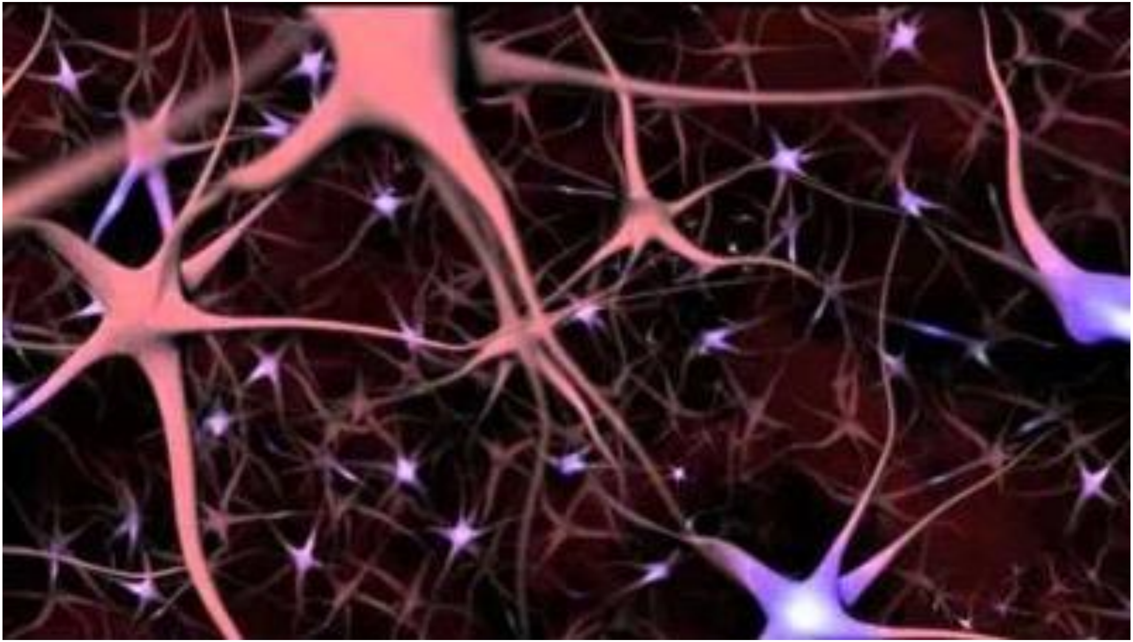


# Generalization

- We would like our models to generalize to data they haven't seen before
- The degree of the polynomial is an example of a hyperparameter, something we can't include in the training procedure itself
- We can tune hyperparameters using a validation set:

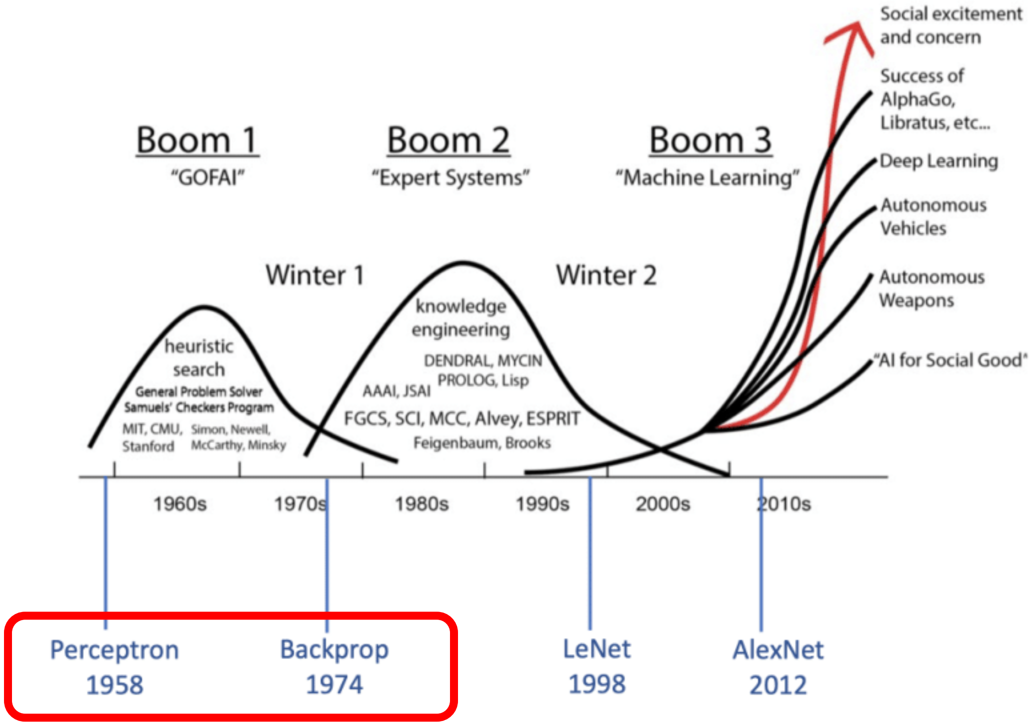


# Multi-Layer Perceptrons



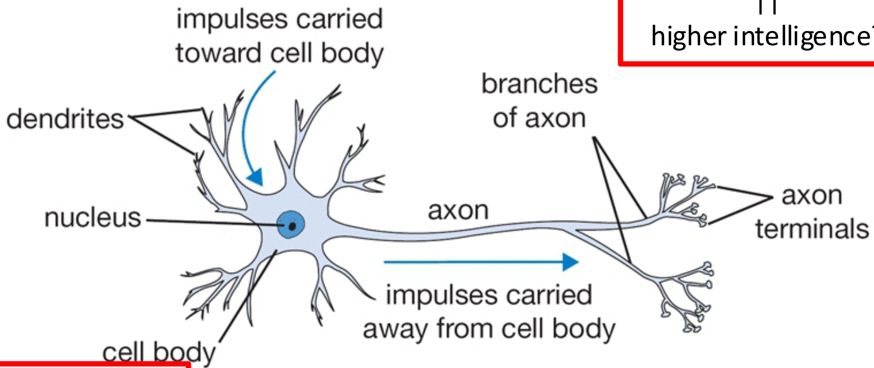
Source: https://www.youtube.com/watch?v=vyNkAuX29OU

# A brief history



# Multilayer Perceptrons

more neurons
||
higher intelligence?



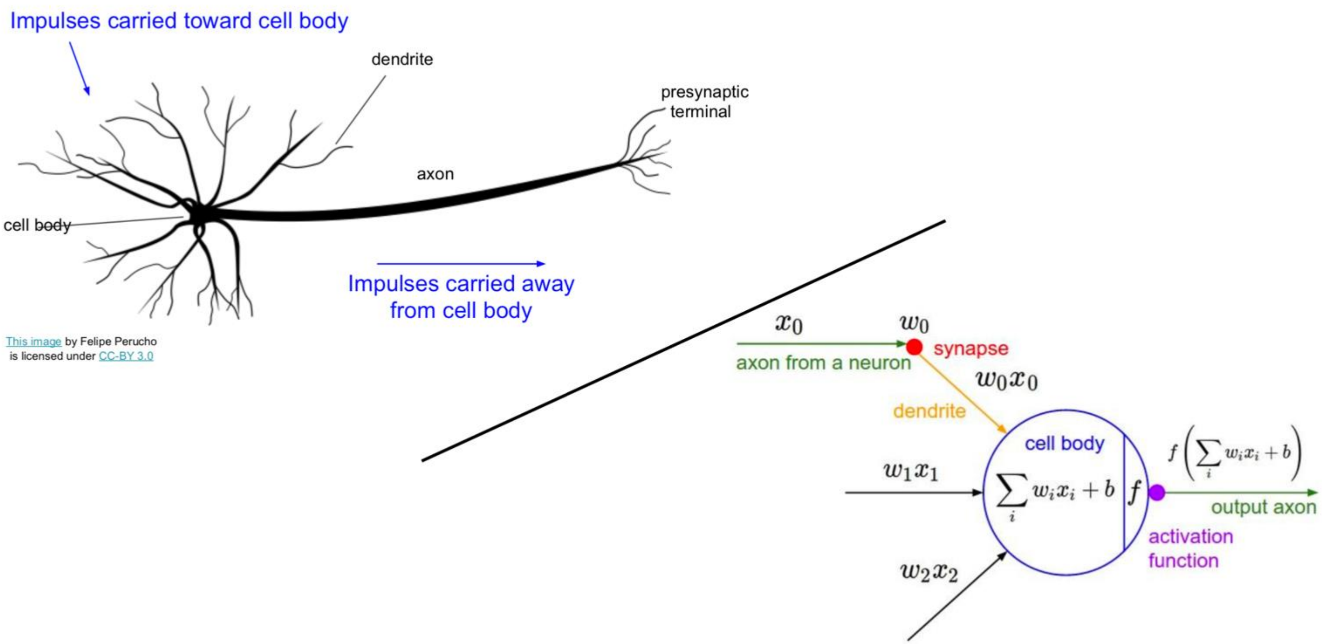

Some fun facts :

1 million x

100 billion neurons

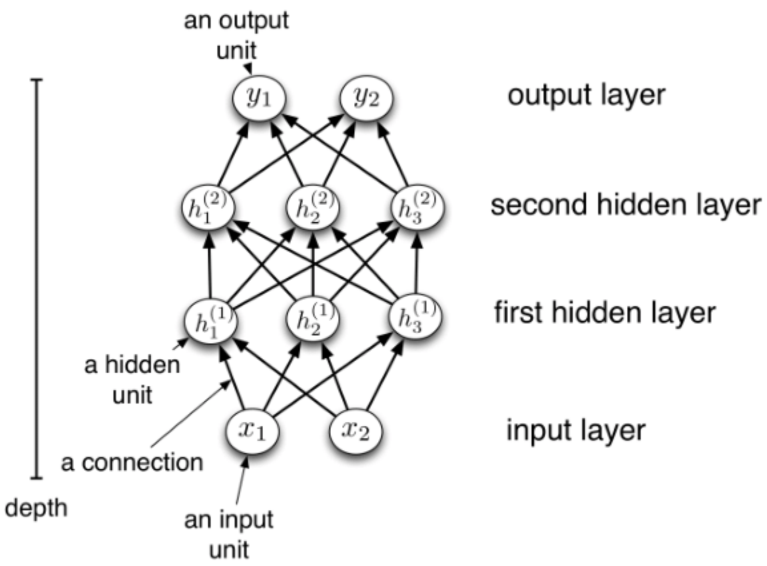
100,000 neurons

# Multilayer Perceptrons



# Multilayer Perceptrons

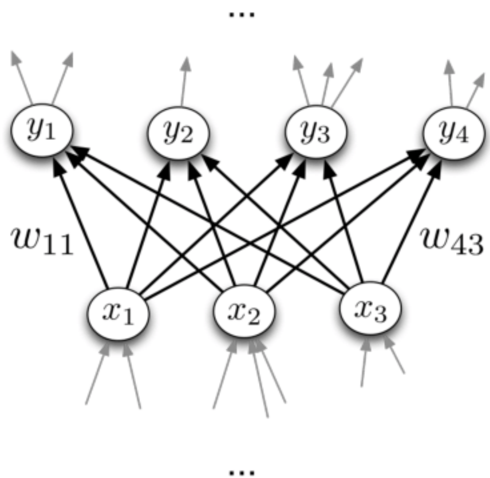
- We can connect lots of units together into a directed acyclic graph.
- This gives a feed-forward neural network. That's in contrast to recurrent neural networks, which can have cycles. (We'll talk about those later.)
- Typically, units are grouped together into layers.



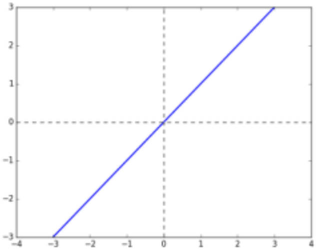
# Multilayer Perceptrons

- Each layer connects $N$ input units to $M$ output units.
- In the simplest case, all input units are connected to all output units. We call this a fully connected layer. We'll consider other layer types later.
- Note: the inputs and outputs for a layer are distinct from the inputs and outputs to the network.

- Recall from softmax regression: this means we need an $M \times N$ weight matrix.
- The output units are a function of the input units:
$$\mathbf{y} = f(\mathbf{x}) = \phi\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right)$$
- A multilayer network consisting of fully connected layers is called a multilayer perceptron. Despite the name, it has nothing to do with perceptrons!

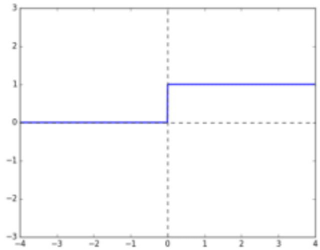


**Some activation functions:**



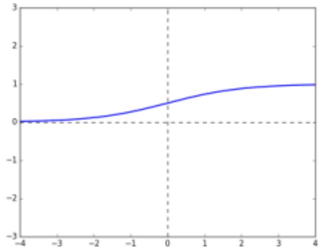
**Linear**

$$y = z$$



**Hard Threshold**

$$y = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$

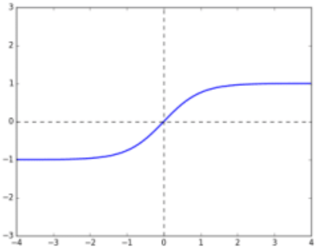


**Logistic**

$$y = \frac{1}{1 + e^{-z}}$$

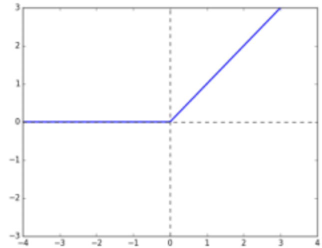
# Multilayer Perceptrons

**Some activation functions:**



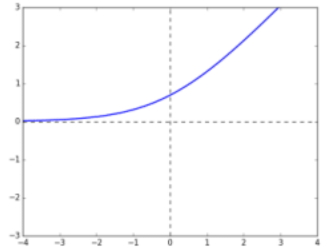
**Hyperbolic Tangent (tanh)**

$$y = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



**Rectified Linear Unit (ReLU)**

$$y = \max(0, z)$$



**Soft ReLU**

$$y = \log 1 + e^z$$

# Multilayer Perceptrons

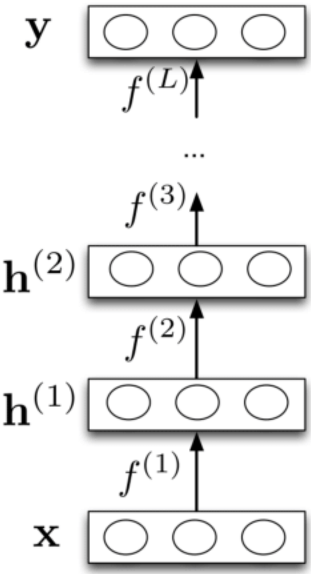
- Each layer computes a function, so the network computes a composition of functions:

$$\mathbf{h}^{(1)} = f^{(1)}(\mathbf{x})$$
$$\mathbf{h}^{(2)} = f^{(2)}(\mathbf{h}^{(1)})$$
$$\vdots$$
$$\mathbf{y} = f^{(L)}(\mathbf{h}^{(L-1)})$$

- Or more simply:

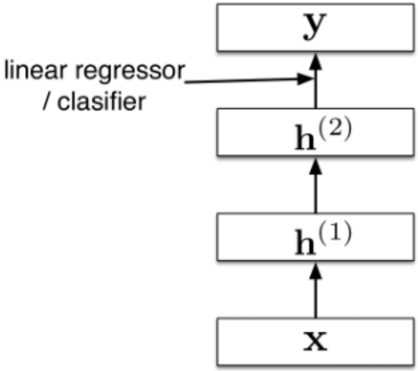
$$\mathbf{y} = f^{(L)} \circ \cdots \circ f^{(1)}(\mathbf{x}).$$

- Neural nets provide modularity: we can implement each layer's computations as a black box.



# Feature Learning

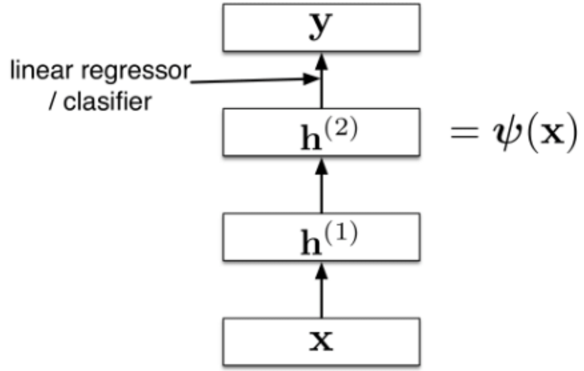
- Neural nets can be viewed as a way of learning features:



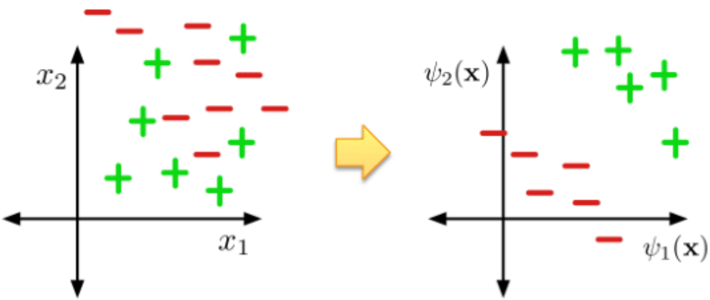

$= \psi(\mathbf{x})$

# Feature Learning

- Neural nets can be viewed as a way of learning features:




- The goal:



# Expressive Power

- We've seen that there are some functions that linear classifiers can't represent. Are deep networks any better?
- Any sequence of *linear* layers can be equivalently represented with a single linear layer.

$$\mathbf{y} = \underbrace{\mathbf{W}^{(3)}\mathbf{W}^{(2)}\mathbf{W}^{(1)}}_{\triangleq \mathbf{W}'}\mathbf{x}$$

  - Deep linear networks are no more expressive than linear regression!
  - Linear layers do have their uses — stay tuned!

https://arxiv.org/pdf/1610.00291.pdf

# Expressive Power

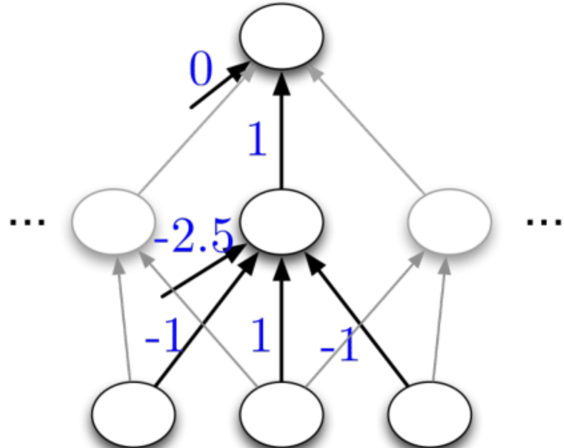
- Multilayer feed-forward neural nets with *nonlinear* activation functions are universal approximators: they can approximate any function arbitrarily well.
- This has been shown for various activation functions (thresholds, logistic, ReLU, etc.)
  - Even though ReLU is "almost" linear, it's nonlinear enough!

## Expressive Power

**Universality for binary inputs and targets:**

- Hard threshold hidden units, linear output
- Strategy: $2^D$ hidden units, each of which responds to one particular input configuration

| $x_1$ | $x_2$ | $x_3$ | $t$ |
|---|---|---|---|
| | $\vdots$ | | $\vdots$ |
| -1 | -1 | 1 | -1 |
| -1 | 1 | -1 | 1 |
| -1 | 1 | 1 | 1 |
| | $\vdots$ | | $\vdots$ |

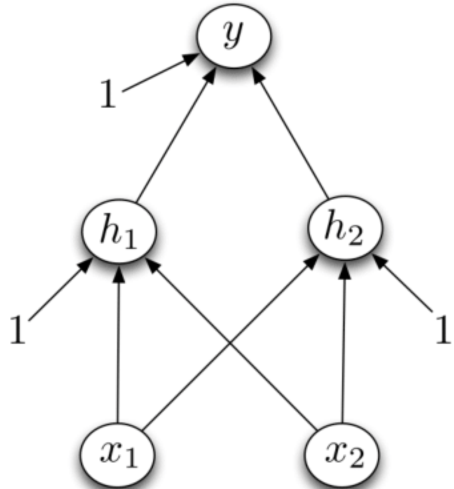



- Only requires one hidden layer, though it needs to be extremely wide!

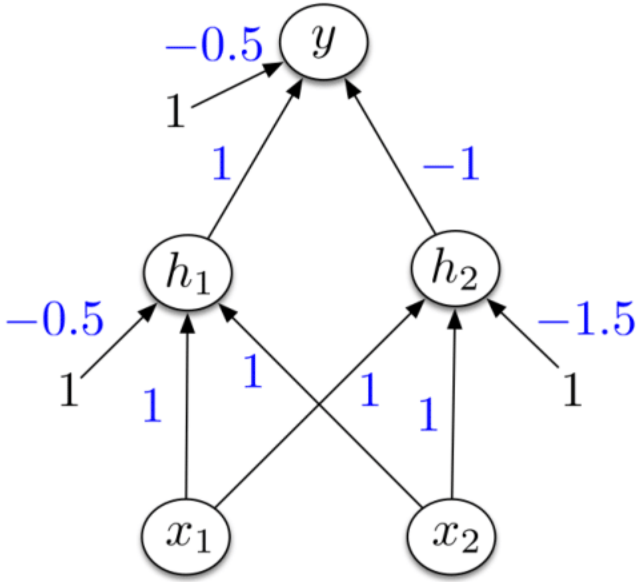
# Multilayer Perceptrons

**Designing a network to compute XOR:**

Assume hard threshold activation function



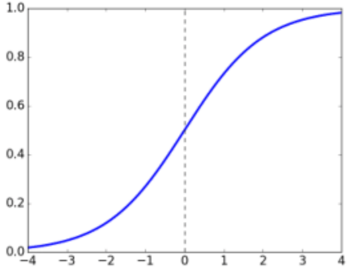
# Multilayer Perceptrons



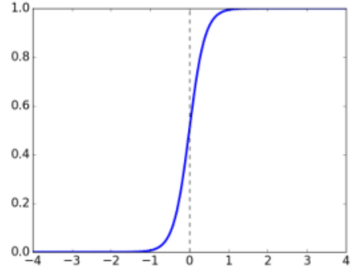

**Exercise**: Could you come up with another set of weights to compute XOR?

# Expressive Power

- What about the logistic activation function?
- You can approximate a hard threshold by scaling up the weights and biases:




$y = \sigma(x)$




$y = \sigma(5x)$

- This is good: logistic units are differentiable, so we can tune them with gradient descent. (Stay tuned!)

# Expressive Power

- Limits of universality
  - You may need to represent an exponentially large network.
  - If you can learn any function, you'll just overfit.
  - Really, we desire a *compact* representation!

# Expressive Power

- Limits of universality
  - You may need to represent an exponentially large network.
  - If you can learn any function, you'll just overfit.
  - Really, we desire a *compact* representation!
- We've derived units which compute the functions AND, OR, and NOT. Therefore, any Boolean circuit can be translated into a feed-forward neural net.
  - This suggests you might be able to learn *compact* representations of some complicated functions

# After the break

After the break: **Backpropagation**

## Back-Propagation

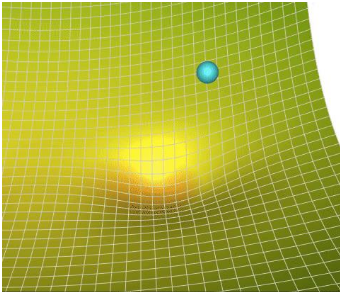

Source: https://www.youtube.com/watch?v=Suevq-kZdIw

# Overview

- We've seen that multilayer neural networks are powerful. But how can we actually learn them?
- Backpropagation is the central algorithm in this course.
  - It's is an algorithm for computing gradients.
  - Really it's an instance of **reverse mode automatic differentiation**, which is much more broadly applicable than just neural nets.
    - This is "just" a clever and efficient use of the Chain Rule for derivatives.
    - We'll see how to implement an automatic differentiation system next week.

# Recap: Gradient Descent

- **Recall:** gradient descent moves opposite the gradient (the direction of steepest descent)



- Weight space for a multilayer neural net: one coordinate for each weight or bias of the network, in *all* the layers
- Conceptually, not any different from what we've seen so far — just higher dimensional and harder to visualize!
- We want to compute the cost gradient $\mathrm{d}\mathcal{J}/\mathrm{d}\mathbf{w}$, which is the vector of partial derivatives.
  - This is the average of $\mathrm{d}\mathcal{L}/\mathrm{d}\mathbf{w}$ over all the training examples, so in this lecture we focus on computing $\mathrm{d}\mathcal{L}/\mathrm{d}\mathbf{w}$.

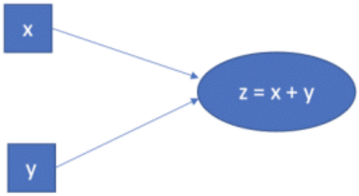
## Recap : Univariate Chain Rule

- We've already been using the univariate Chain Rule.
- Recall: if $f(x)$ and $x(t)$ are univariate functions, then

$$\frac{\mathrm{d}}{\mathrm{d}t} f(x(t)) = \frac{\mathrm{d}f}{\mathrm{d}x} \frac{\mathrm{d}x}{\mathrm{d}t}.$$

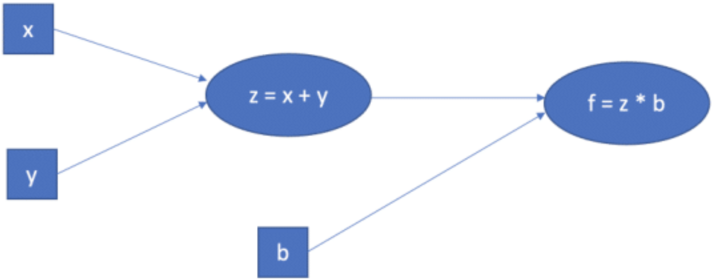
# Recap: Computation Graph

- A computational graph is a directed graph where the **nodes** correspond to **operations** or **variables**.
- Variables can feed their value into operations, and operations can feed their output into other operations. This way, every node in the graph defines a function of the variables.
- For example : we want to plot the operation $z = x + y$, then



# Recap: Computation Graph

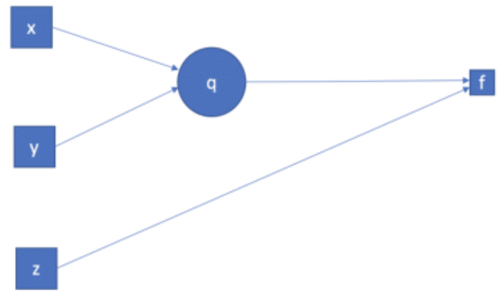
- A computational graph is a directed graph where the **nodes** correspond to **operations** or **variables**.
- Variables can feed their value into operations, and operations can feed their output into other operations. This way, every node in the graph defines a function of the variables.
- Another example : we want to plot the operation $f = (x + y) * b$, then



# A simple example

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

$$q = x + y; f = q * z$$



# A simple example : Forward Pass

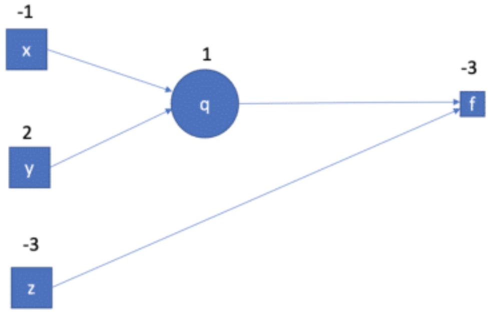
$$f(x, y, z) = (x + y) * z$$

$$q = x + y; f = q * z$$

$$e.g., x = -1, y = 2, z = 3$$

$$then, q = 1, f = -3$$

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



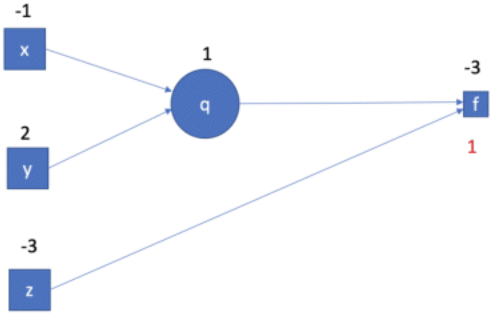
# A simple example : Backward Pass

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

$$q = x + y; f = q * z$$

$$e.g., x = -1, y = 2, z = 3$$

$$baseline : \frac{\partial f}{\partial f} = 1$$



# A simple example : Backward Pass

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

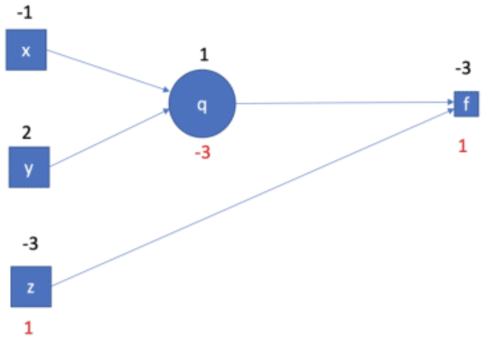
$$q = x + y; f = q * z$$

$$e.g., x = -1, y = 2, z = 3$$

$$baseline : \frac{\partial f}{\partial f} = 1$$

$$\frac{\partial f}{\partial z} = \frac{\partial f}{\partial f}\frac{\partial f}{\partial z} = q = 1$$

$$\frac{\partial f}{\partial q} = \frac{\partial f}{\partial f}\frac{\partial f}{\partial q} = z = -3$$



# A simple example : Backward Pass

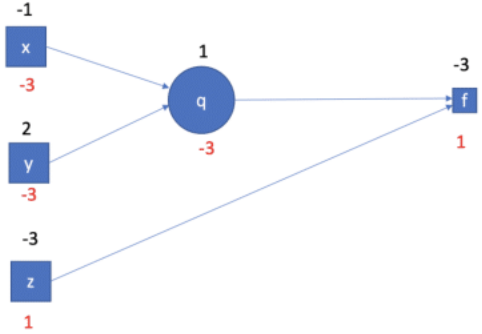
$$f(x, y, z) = (x + y) * z$$

$$q = x + y; f = q * z$$

$$e.g., x = -1, y = 2, z = 3$$

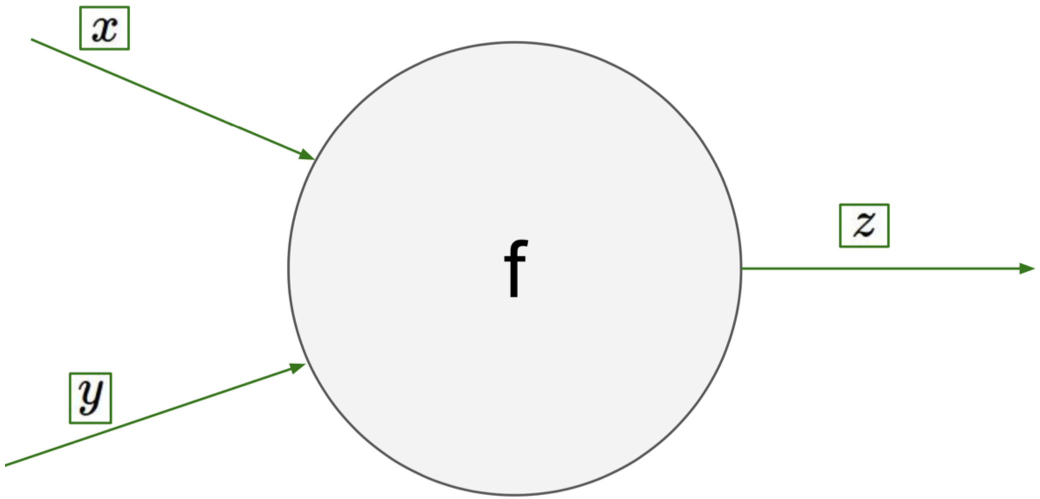
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = (-3) * (1) = -3$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y} = (-3) * (1) = -3$$



# A simple example : Backward Pass

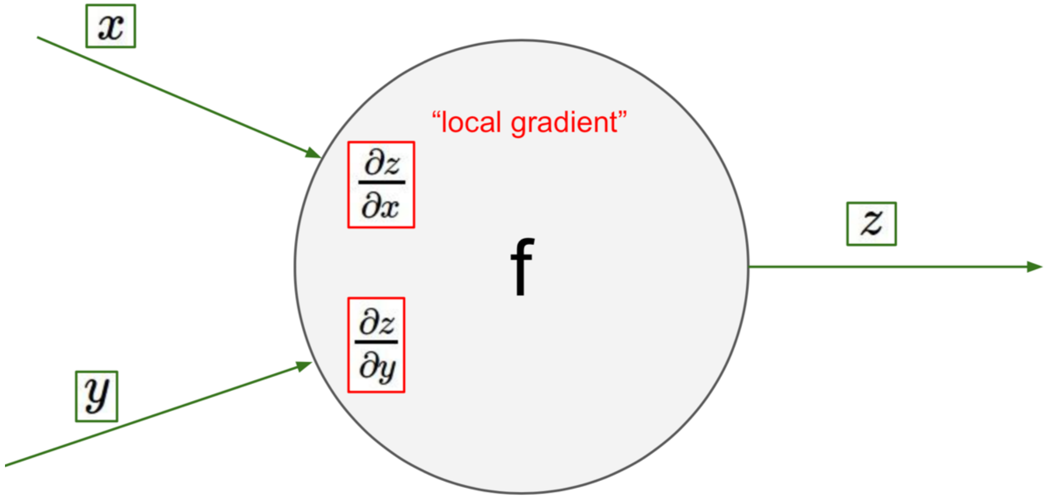
A quick summary:




Source: Fei-Fei Li & Justin Johnson & Serena Yeung, csc231N, Stanford University

# A simple example : Backward Pass

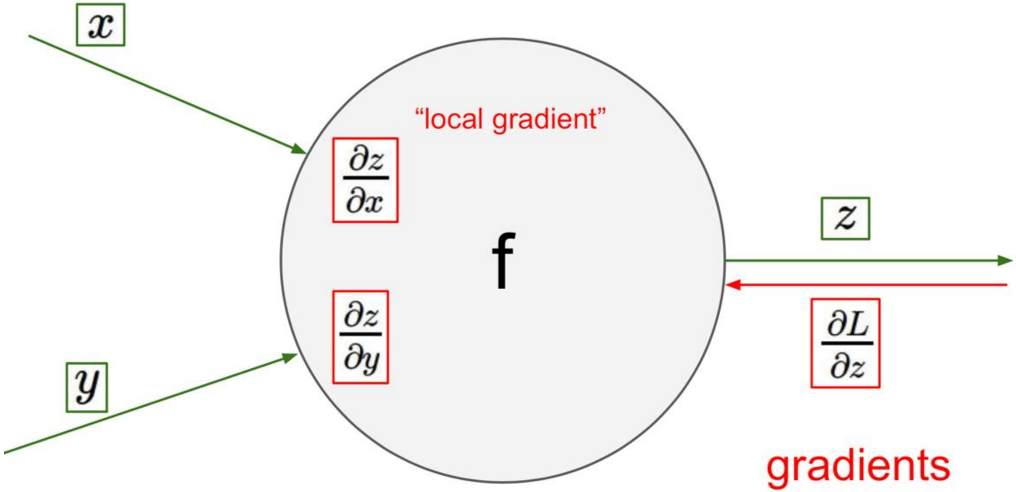
A quick summary:




Source: Fei-Fei Li & Justin Johnson & Serena Yeung, csc231N, Stanford University

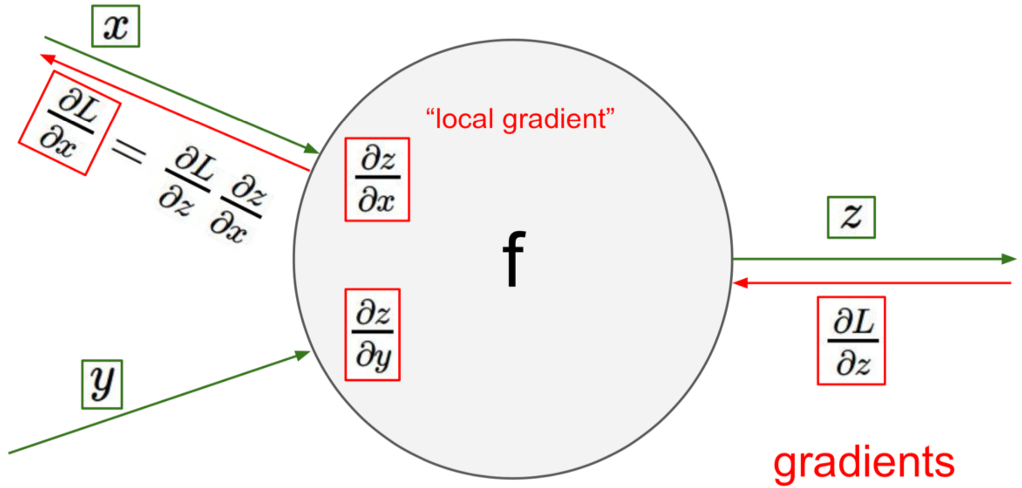
# A simple example : Backward Pass

A quick summary:




Source: Fei-Fei Li & Justin Johnson & Serena Yeung, csc231N, Stanford University

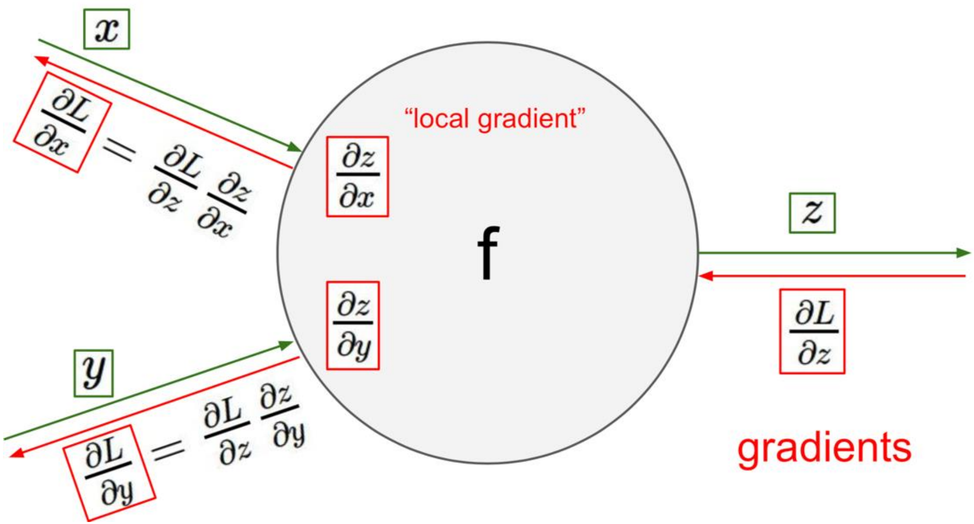
A quick summary:




Source: Fei-Fei Li & Justin Johnson & Serena Yeung, csc231N, Stanford University

# A simple example : Backward Pass

A quick summary:




Source: Fei-Fei Li & Justin Johnson & Serena Yeung, csc231N, Stanford University

# A more complex example: logistic least squares model

**Recall: Univariate logistic least squares model**

$$
\begin{aligned}
z &= wx + b \\
y &= \sigma(z) \\
\mathcal{L} &= \frac{1}{2}(y - t)^2
\end{aligned}
$$

Let's compute the loss derivatives.

# Univariate Chain Rule

**How you would have done it in calculus class**

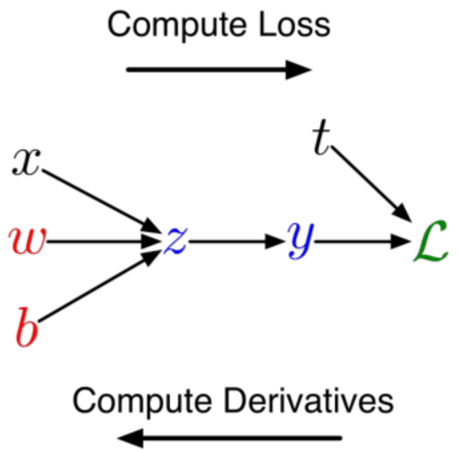
$$
\begin{aligned}
\mathcal{L} &= \frac{1}{2}(\sigma(wx+b)-t)^2 \\
\frac{\partial \mathcal{L}}{\partial w} &= \frac{\partial}{\partial w}\left[\frac{1}{2}(\sigma(wx+b)-t)^2\right] \\
&= \frac{1}{2}\frac{\partial}{\partial w}(\sigma(wx+b)-t)^2 \\
&= (\sigma(wx+b)-t)\frac{\partial}{\partial w}(\sigma(wx+b)-t) \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)\frac{\partial}{\partial w}(wx+b) \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)x
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial b} &= \frac{\partial}{\partial b}\left[\frac{1}{2}(\sigma(wx+b)-t)^2\right] \\
&= \frac{1}{2}\frac{\partial}{\partial b}(\sigma(wx+b)-t)^2 \\
&= (\sigma(wx+b)-t)\frac{\partial}{\partial b}(\sigma(wx+b)-t) \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)\frac{\partial}{\partial b}(wx+b) \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)
\end{aligned}
$$

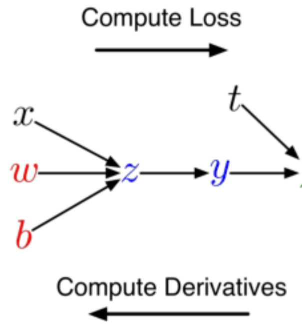
What are the disadvantages of this approach?

# Univariate Chain Rule

- We can diagram out the computations using a computation graph.
- The nodes represent all the inputs and computed quantities, and the edges represent which nodes are computed directly as a function of which other nodes.



# A more structured way to do it




**Computing the loss:**

$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

**Computing the derivatives:**

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y} = y - t$$
$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y}\,\sigma'(z)$$
$$\frac{\partial\mathcal{L}}{\partial w} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}\,x$$
$$\frac{\partial\mathcal{L}}{\partial b} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}$$

# Univariate Chain Rule

**A slightly more convenient notation:**

- Use $\overline{y}$ to denote the derivative $\mathrm{d}\mathcal{L}/\mathrm{d}y$, sometimes called the error signal.
- This emphasizes that the error signals are just values our program is computing (rather than a mathematical operation).
- This is not a standard notation, but I couldn't find another one that I liked.

**Computing the loss:**

$$\begin{aligned} z &= wx + b \\ y &= \sigma(z) \\ \mathcal{L} &= \frac{1}{2}(y - t)^2 \end{aligned}$$

**Computing the derivatives:**

$$\begin{aligned} \overline{y} &= y - t \\ \overline{z} &= \overline{y}\,\sigma'(z) \\ \overline{w} &= \overline{z}\,x \\ \overline{b} &= \overline{z} \end{aligned}$$

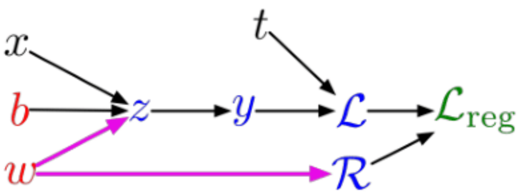
# After the break

After the break: **Back-propagation in Multivariate Forms**

# Multivariate Chain Rule

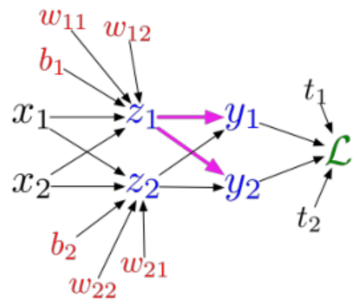
**Problem:** what if the computation graph has fan-out $> 1$?
This requires the multivariate Chain Rule!

### $L_2$-Regularized regression




$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$
$$\mathcal{R} = \frac{1}{2}w^2$$
$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda\mathcal{R}$$

### Multiclass logistic regression

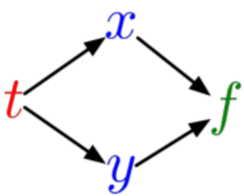



$$z_\ell = \sum_j w_{\ell j} x_j + b_\ell$$
$$y_k = \frac{e^{z_k}}{\sum_\ell e^{z_\ell}}$$
$$\mathcal{L} = -\sum_k t_k \log y_k$$

# Multivariate Chain Rule

- Suppose we have a function $f(x, y)$ and functions $x(t)$ and $y(t)$. (All the variables here are scalar-valued.) Then

$$\frac{\mathrm{d}}{\mathrm{d}t} f(x(t), y(t)) = \frac{\partial f}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t}$$




- Example:

$$\begin{aligned} f(x, y) &= y + e^{xy} \\ x(t) &= \cos t \\ y(t) &= t^2 \end{aligned}$$

- Plug in to Chain Rule:

$$\begin{aligned} \frac{\mathrm{d}f}{\mathrm{d}t} &= \frac{\partial f}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t} \\ &= (ye^{xy}) \cdot (-\sin t) + (1 + xe^{xy}) \cdot 2t \end{aligned}$$

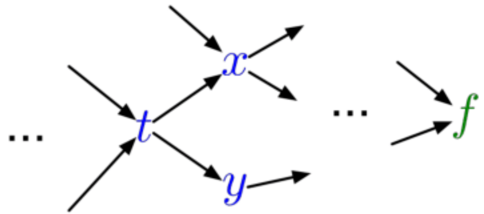
# Multivariable Chain Rule

- In the context of backpropagation:

Mathematical expressions to be evaluated

$$\frac{\mathrm{d}f}{\mathrm{d}t} = \frac{\partial f}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t}$$

Values already computed by our program




- In our notation:

$$\overline{t} = \overline{x}\,\frac{\mathrm{d}x}{\mathrm{d}t} + \overline{y}\,\frac{\mathrm{d}y}{\mathrm{d}t}$$

# Backpropagation

**Full backpropagation algorithm:**

Let $v_1, \ldots, v_N$ be a topological ordering of the computation graph (i.e. parents come before children.)

$v_N$ denotes the variable we're trying to compute derivatives of (e.g. loss).

forward pass

For $i = 1, \ldots, N$

Compute $v_i$ as a function of $\mathrm{Pa}(v_i)$

backward pass

$\overline{v_N} = 1$

For $i = N-1, \ldots, 1$

$$\overline{v_i} = \sum_{j \in \mathrm{Ch}(v_i)} \overline{v_j} \frac{\partial v_j}{\partial v_i}$$

# Backpropagation

**Full backpropagation algorithm:**

Let $v_1, \ldots, v_N$ be a topological ordering of the computation graph (i.e. parents come before children.)

$v_N$ denotes the variable we're trying to compute derivatives of (e.g. loss).

forward pass

For $i = 1, \ldots, N$

Compute $v_i$ as a function of $\mathrm{Pa}(v_i)$

backward pass

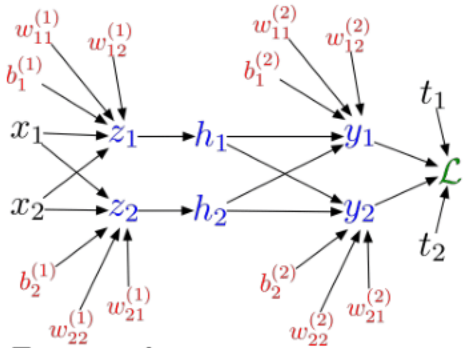
$\overline{v_N} = 1$

For $i = N-1, \ldots, 1$

$$\overline{v_i} = \sum_{j \in \mathrm{Ch}(v_i)} \overline{v_j} \frac{\partial v_j}{\partial v_i}$$

# Backpropagation

**Multilayer Perceptron** (multiple outputs):




**Forward pass:**

$$z_i = \sum_j w_{ij}^{(1)} x_j + b_i^{(1)}$$

$$h_i = \sigma(z_i)$$

$$y_k = \sum_i w_{ki}^{(2)} h_i + b_k^{(2)}$$

$$\mathcal{L} = \frac{1}{2} \sum_k (y_k - t_k)^2$$

**Backward pass:**

$$\overline{\mathcal{L}} = 1$$

$$\overline{y_k} = \overline{\mathcal{L}}\,(y_k - t_k)$$

$$\overline{w_{ki}^{(2)}} = \overline{y_k}\, h_i$$

$$\overline{b_k^{(2)}} = \overline{y_k}$$

$$\overline{h_i} = \sum_k \overline{y_k}\, w_{ki}^{(2)}$$

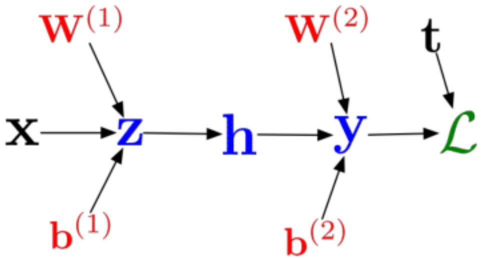
$$\overline{z_i} = \overline{h_i}\, \sigma'(z_i)$$

$$\overline{w_{ij}^{(1)}} = \overline{z_i}\, x_j$$

$$\overline{b_i^{(1)}} = \overline{z_i}$$

# Vector Form

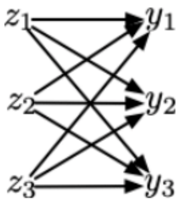
- Computation graphs showing individual units are cumbersome.
- As you might have guessed, we typically draw graphs over the vectorized variables.




- We pass messages back analogous to the ones for scalar-valued nodes.

# Vector Form

- Consider this computation graph:




$$\mathbf{z} \longrightarrow \mathbf{y}$$

- Backprop rules:

$$\mathbf{z} \in \mathcal{R}^N, \mathbf{y} \in \mathcal{R}^M \qquad \overline{z_j} = \sum_k \overline{y_k} \frac{\partial y_k}{\partial z_j} \qquad \overline{\mathbf{z}} = \frac{\partial \mathbf{y}}{\partial \mathbf{z}}^\top \overline{\mathbf{y}},$$

where $\partial \mathbf{y}/\partial \mathbf{z}$ is the Jacobian matrix (**note**: check the matrix shapes):

$$\left(\frac{\partial \mathbf{y}}{\partial \mathbf{z}}\right)_{M \times N} = \begin{pmatrix} \frac{\partial y_1}{\partial z_1} & \cdots & \frac{\partial y_1}{\partial z_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial z_1} & \cdots & \frac{\partial y_m}{\partial z_n} \end{pmatrix}$$

# Vector Form

Examples

- Matrix-vector product

$$\mathbf{z} = \mathbf{W}\mathbf{x} \qquad \frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \mathbf{W} \qquad \overline{\mathbf{x}} = \mathbf{W}^\top \overline{\mathbf{z}}$$

- Elementwise operations

$$\mathbf{y} = \exp(\mathbf{z}) \qquad \frac{\partial \mathbf{y}}{\partial \mathbf{z}} = \begin{pmatrix} \exp(z_1) & & 0 \\ & \ddots & \\ 0 & & \exp(z_D) \end{pmatrix} \qquad \overline{\mathbf{z}} = \exp(\mathbf{z}) \circ \overline{\mathbf{y}}$$

- Note: we never explicitly construct the Jacobian. It's usually simpler and more efficient to compute the Vector Jacobian Product (VJP) directly.

# Hessian: Higher-order Gradients

- Hessian

$$\mathbf{H} = \begin{pmatrix} \frac{\partial^2 \mathcal{L}}{\partial x_1^2} & \frac{\partial^2 \mathcal{L}}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 \mathcal{L}}{\partial x_1 \partial x_n} \\ \frac{\partial^2 \mathcal{L}}{\partial x_2 \partial x_1} & \frac{\partial^2 \mathcal{L}}{\partial x_2^2} & \cdots & \frac{\partial^2 \mathcal{L}}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 \mathcal{L}}{\partial x_n \partial x_1} & \frac{\partial^2 \mathcal{L}}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 \mathcal{L}}{\partial x_n^2} \end{pmatrix}$$

- Note: Again, we never explicitly construct the Hessian. It's usually simpler and more efficient to compute the Vector Hessian Product (VHP) directly.
- Note: You will need to practice this in HW1.

# Vector Form

**Full backpropagation algorithm (vector form):**

Let $\mathbf{v}_1, \ldots, \mathbf{v}_N$ be a topological ordering of the computation graph (i.e. parents come before children.)

$\mathbf{v}_N$ denotes the variable we're trying to compute derivatives of (e.g. loss). It's a scalar, which we can treat as a 1-D vector.

forward pass

$$\text{For } i = 1, \ldots, N$$

$$\quad \text{Compute } \mathbf{v}_i \text{ as a function of } \mathrm{Pa}(\mathbf{v}_i)$$

backward pass

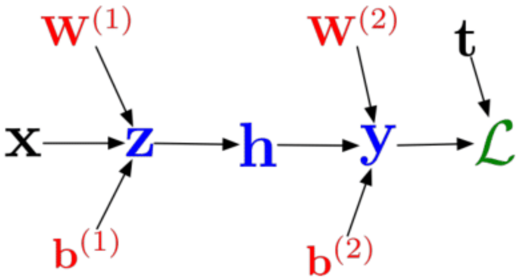
$$\overline{\mathbf{v}_N} = 1$$

$$\text{For } i = N-1, \ldots, 1$$

$$\quad \overline{\mathbf{v}_i} = \sum_{j \in \mathrm{Ch}(\mathbf{v}_i)} \frac{\partial \mathbf{v}_j}{\partial \mathbf{v}_i}^\top \overline{\mathbf{v}_j}$$

# Vector Form

**MLP example in vectorized form:**




**Forward pass:**

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$
$$\mathbf{h} = \sigma(\mathbf{z})$$
$$\mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$$
$$\mathcal{L} = \frac{1}{2}\|\mathbf{t} - \mathbf{y}\|^2$$

**Backward pass:**

$$\overline{\mathcal{L}} = 1$$
$$\overline{\mathbf{y}} = \overline{\mathcal{L}}\,(\mathbf{y} - \mathbf{t})$$
$$\overline{\mathbf{W}^{(2)}} = \overline{\mathbf{y}}\mathbf{h}^\top$$
$$\overline{\mathbf{b}^{(2)}} = \overline{\mathbf{y}}$$
$$\overline{\mathbf{h}} = \mathbf{W}^{(2)\top}\overline{\mathbf{y}}$$
$$\overline{\mathbf{z}} = \overline{\mathbf{h}} \circ \sigma'(\mathbf{z})$$
$$\overline{\mathbf{W}^{(1)}} = \overline{\mathbf{z}}\mathbf{x}^\top$$
$$\overline{\mathbf{b}^{(1)}} = \overline{\mathbf{z}}$$

# Computational Cost

- Computational cost of forward pass: one add-multiply operation per weight

$$z_i = \sum_j w_{ij}^{(1)} x_j + b_i^{(1)}$$

- Computational cost of backward pass: two add-multiply operations per weight

$$\begin{aligned}\overline{w_{ki}^{(2)}} &= \overline{y_k}\, h_i \\ \overline{h_i} &= \sum_k \overline{y_k} w_{ki}^{(2)}\end{aligned}$$

- Rule of thumb: the backward pass is about as expensive as two forward passes.
- For a multilayer perceptron, this means the cost is linear in the number of layers, quadratic in the number of units per layer.

# Closing Thoughts

- Backprop is used to train the overwhelming majority of neural nets today.
    - Even optimization algorithms much fancier than gradient descent (e.g. second-order methods) use backprop to compute the gradients.
- Despite its practical success, backprop is believed to be neurally implausible.
    - No evidence for biological signals analogous to error derivatives.
    - All the biologically plausible alternatives we know about learn much more slowly (on computers).
    - So how on earth does the brain learn?

## Closing Thoughts

The psychological profiling [of a programmer] is mostly the ability to shift levels of abstraction, from low level to high level. To see something in the small and to see something in the large.

– Don Knuth

- By now, we've seen three different ways of looking at gradients:
  - **Geometric:** visualization of gradient in weight space
  - **Algebraic:** mechanics of computing the derivatives
  - **Implementational:** efficient implementation on the computer
- When thinking about neural nets, it's important to be able to shift between these different perspectives!