CSC321 Neural Networks and Machine Learning

Lecture 4

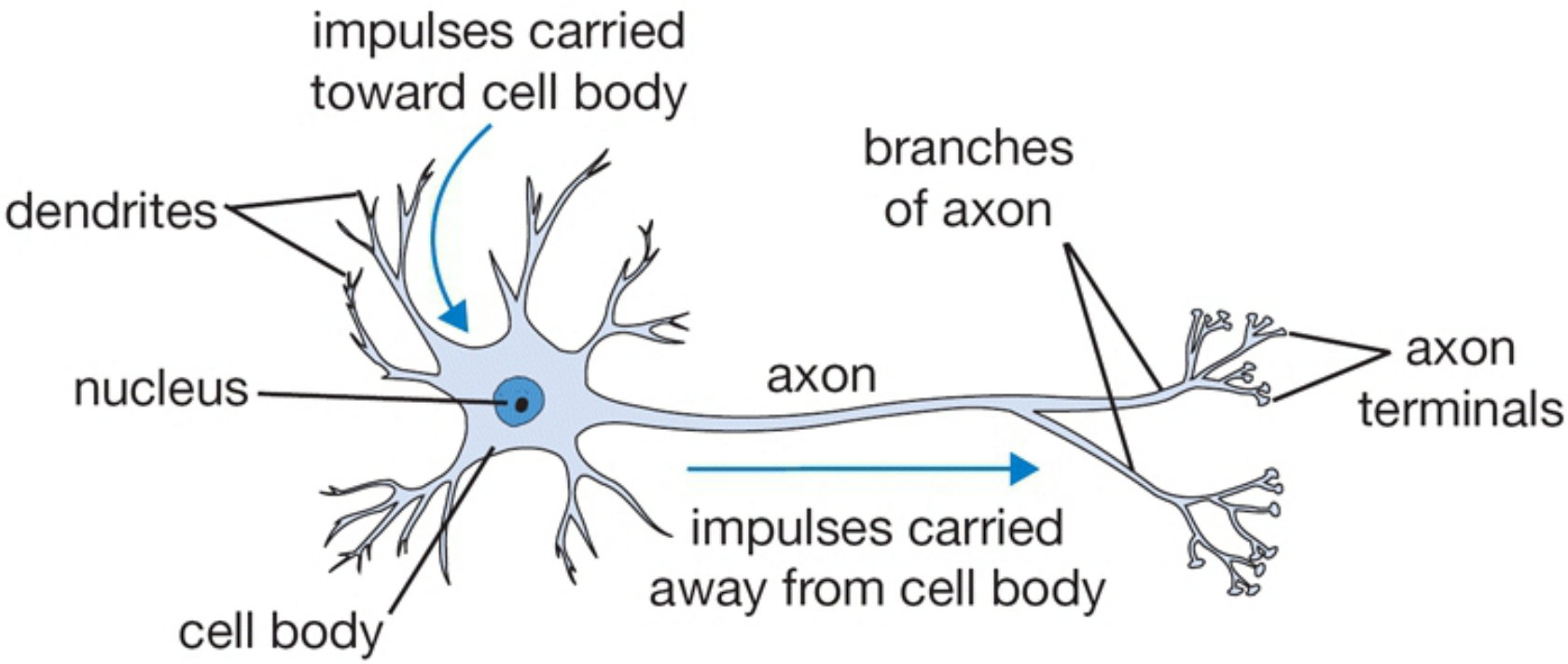
January 29, 2020

Agenda

- Biological and Artificial Neurons
- Neural Network
- Multi-Layer Perceptron (Fully-connected layers)
- Backpropagation

Biological and Artificial Neurons

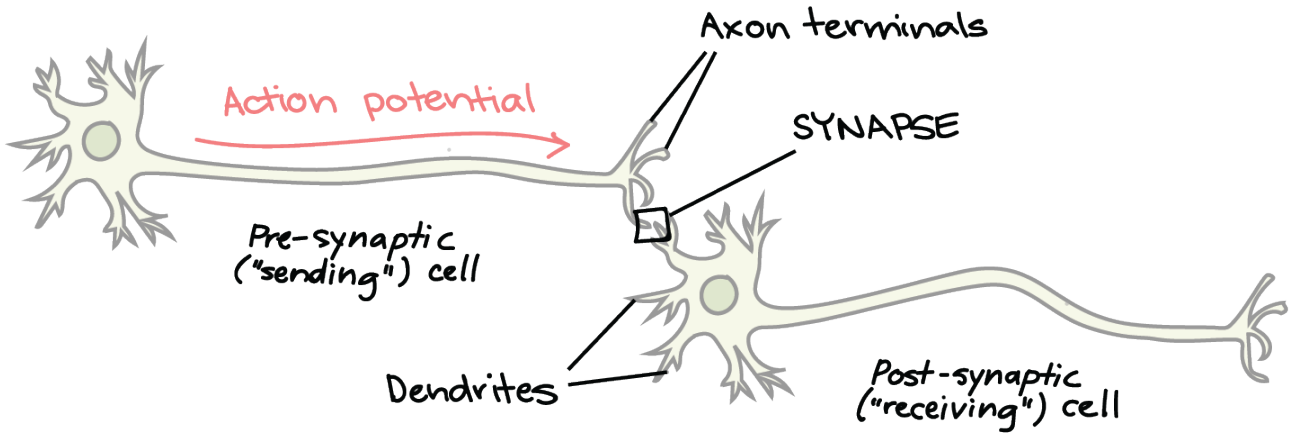
Neuron


impulses carried
toward cell body
dendrites
branches
of axon
nucleus
axon
axon
terminals
impulses carried
away from cell body
cell body

Neuron Anatomy

- The **dendrites**, which are connected to other cells that provides information.
- The **cell body**, which consolidates information from the dendrites.
- The **axon**, which is an extension from the cell body that passes information to other cells.
- The **synapse**, which is the area where the axon of one neuron and the dendrite of another connect.

What does a neuron do?



- Consolidates "information" (voltage difference) from its dendrites
- If the total activity in a neuron's dendrite lowers the voltage difference enough, the entire cell *depolarizes* and the neuron *fires*
- The voltage signal spread along the axon and to the synapse, then to the next nuerons
- Neuron sends information to the next cell

What makes a neuron fires?

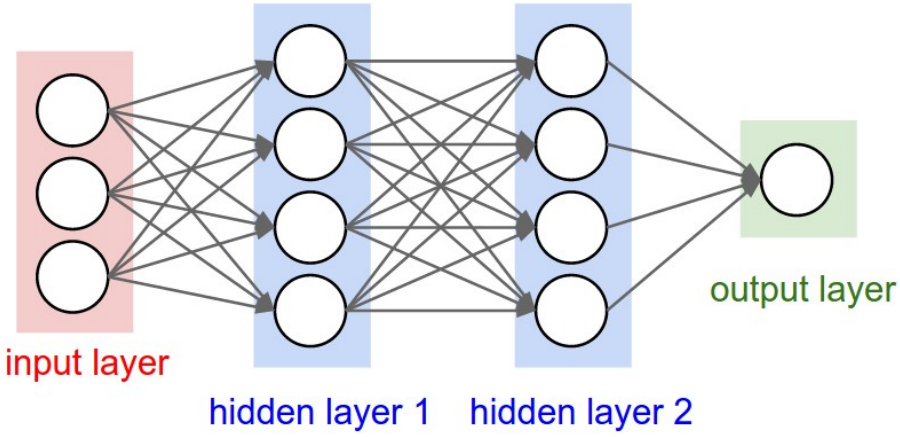
Neuron can fire in response to...

- retinal cells
- certain edges, lines, angles, movements
- hands and faces (in primates)
- specific people like Jennifer Aniston (in humans)
  - Existence of "Grandmother cells" is still contested

An Artificial Neural Network (Multi-Layer Perceptron)

Idea:

- Use a simplified (mathematical) model of a neuron as building blocks
- Connect the neurons together in the following way:

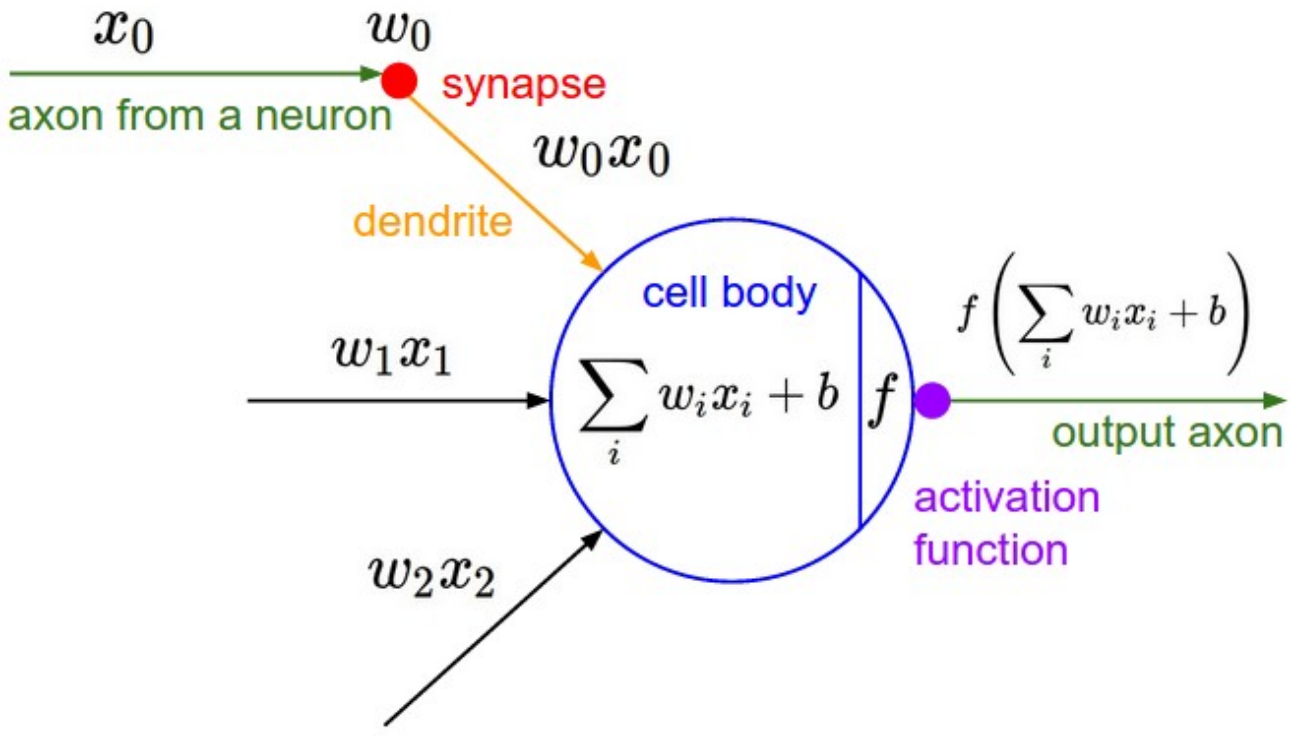


- An **input layer**: feed in input features (e.g. like retinal cells in your eyes)
- A number of **hidden layers**: don't have specific meaning
- An **output layer**: interpret output like a "grandmother cell"

But what do the neurons mean?

- Use x_i to encode the input
  - e.g. pixels in an image
  - like the neurons that are connected to the receptors in the eye
- Use y to encode the output (of a binary classification problem)
  - e.g. cancer vs. not cancer

Modeling Individual Neurons



- $x_1, x_2, \ldots =$ inputs to the neuron
- $w_1, w_2, \ldots =$ the neuron's **weights**
- $b =$ the neuron's **bias**
- $f =$ an **activation function**
- $f(\sum_i x_i w_i + b) =$ the neuron's **activation** (output)

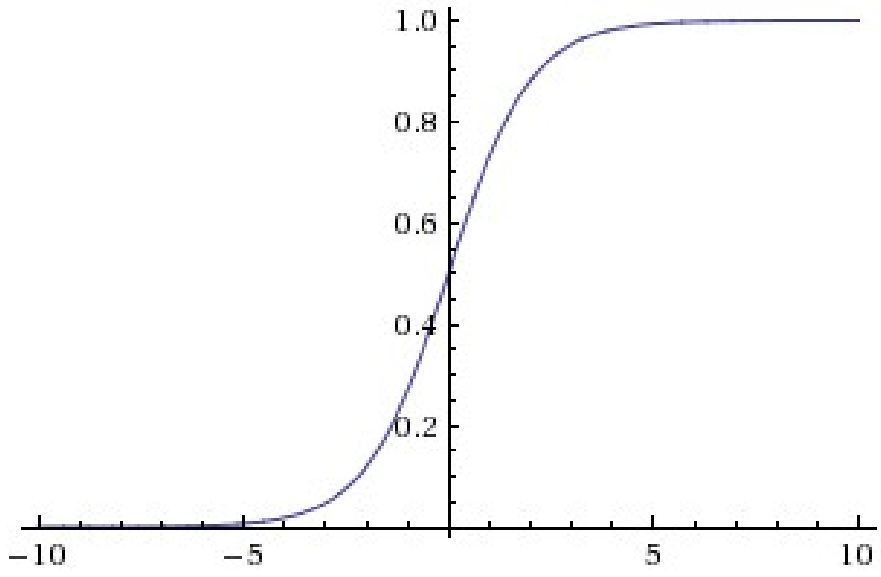
Activation Functions: common choices

Common Choices:

- Sigmoid activation
- Tanh activation
- ReLU activation

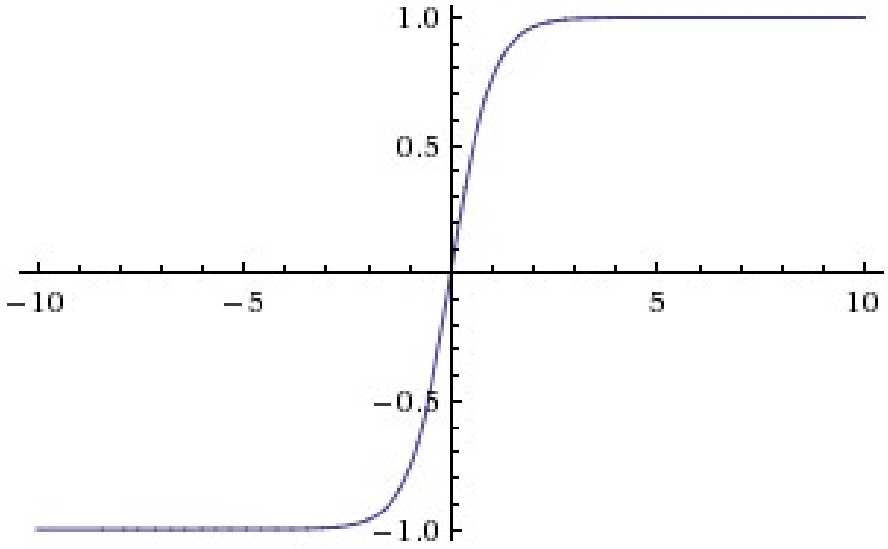
Rule of thumb: Start with ReLU activation. If necessary, try tanh.

Activation Function: Sigmoid



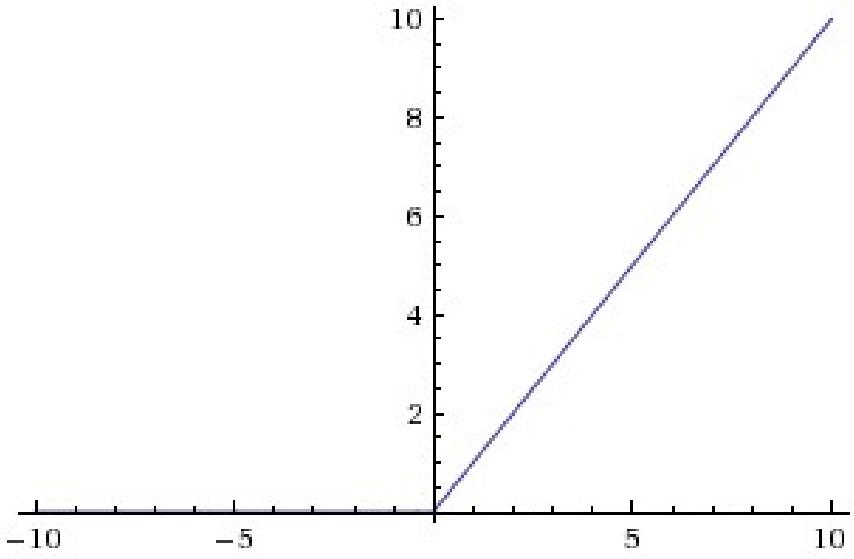
- somewhat problematic due to gradient signal
- all activations are positive

Activation Function: Tanh



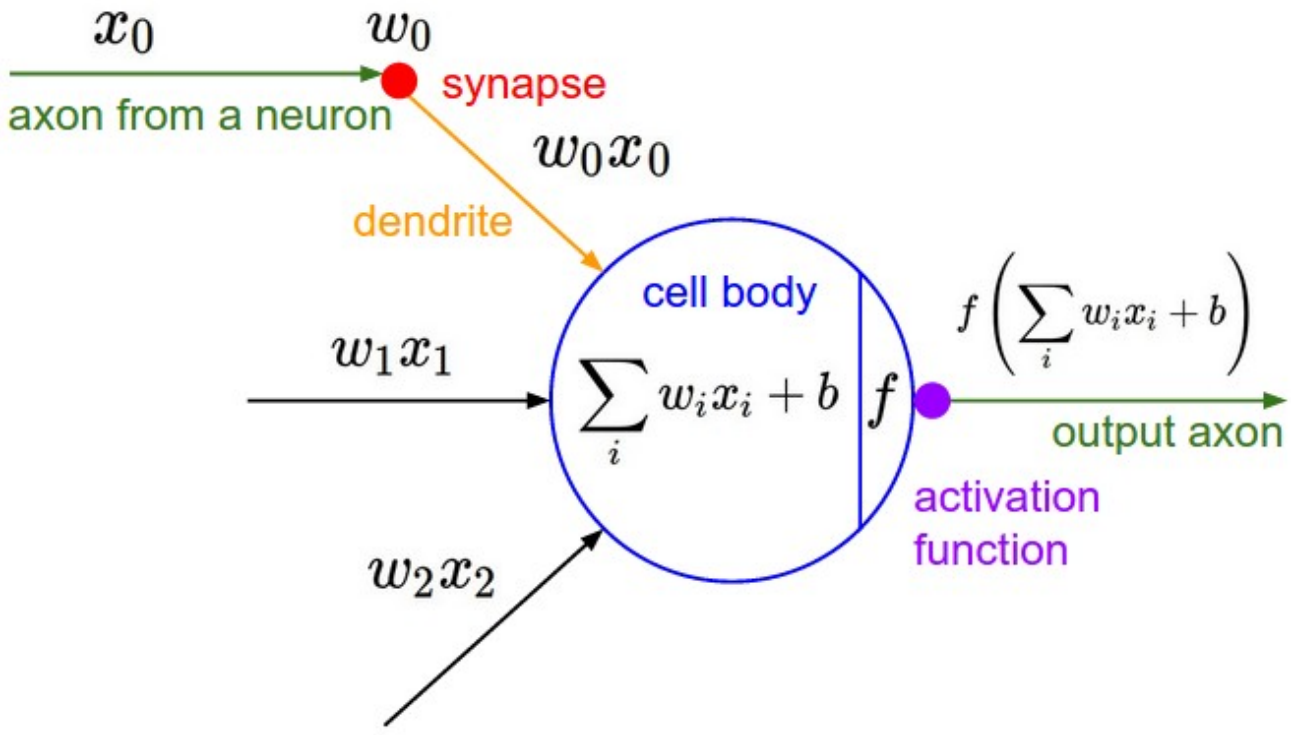
- scaled version of the sigmoid activation
- also somewhat problematic due to gradient signal
- activations can be positive or negative

Activation Function: ReLU



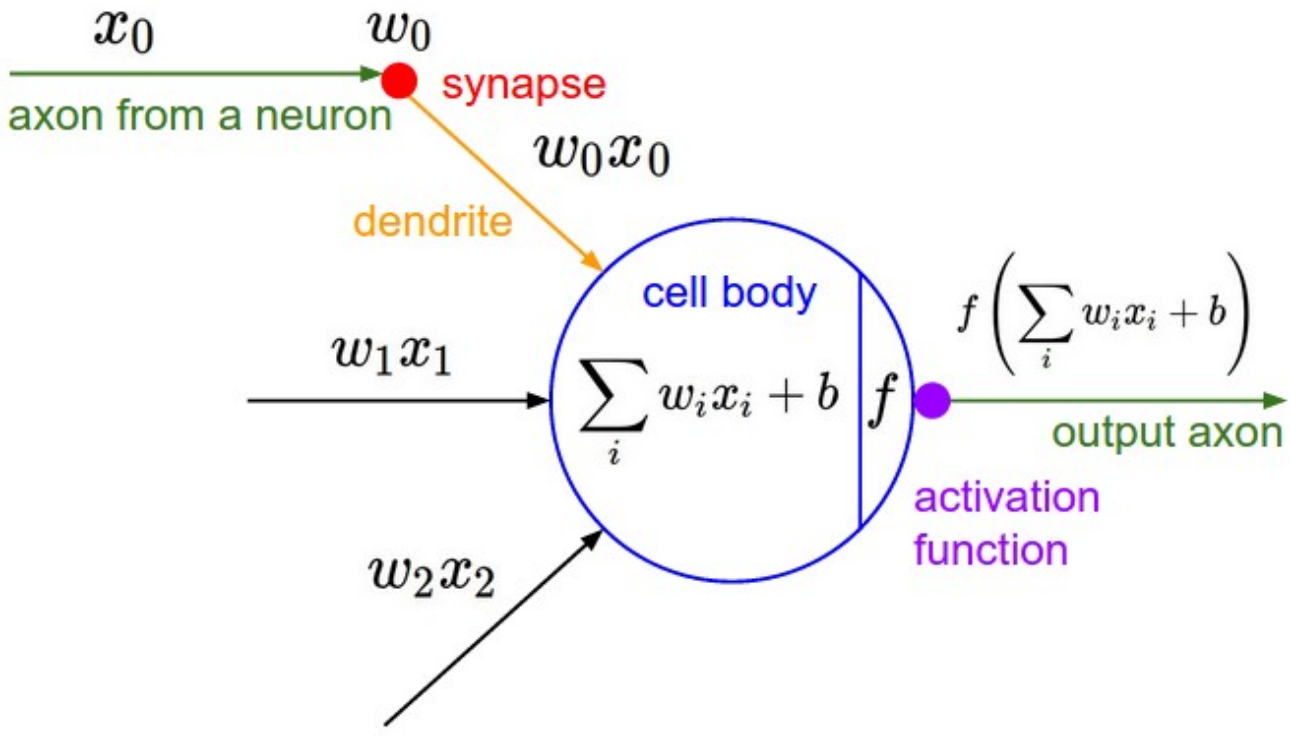
- most often used nowadays
- all activations are positive
- easy to compute gradients
- can be problematic if the bias is too large and negative, so the activations are always 0

Linear Regression as a Single Neuron



- $x_1, x_2, \ldots$: inputs
- $w_1, w_2, \ldots$: components of the **weight vector w**
- b : the **bias**
- f : identity function
- $y = \sum_i x_i w_i + b = \mathbf{w}^T \mathbf{x} + b$

Binary Classification (Logistic Regression) as a Single Neuron



- $x_1, x_2, \ldots$: inputs
- $w_1, w_2, \ldots$: components of the **weight vector w**
- b : the **bias**
- $f = \sigma$
- $y = \sigma(\sum_i x_i w_i + b) = \sigma(\mathbf{w}^T \mathbf{x} + b)$

Multi-Classification (Logistic Regression) as a Neural Network

We use K neurons (one for each class):

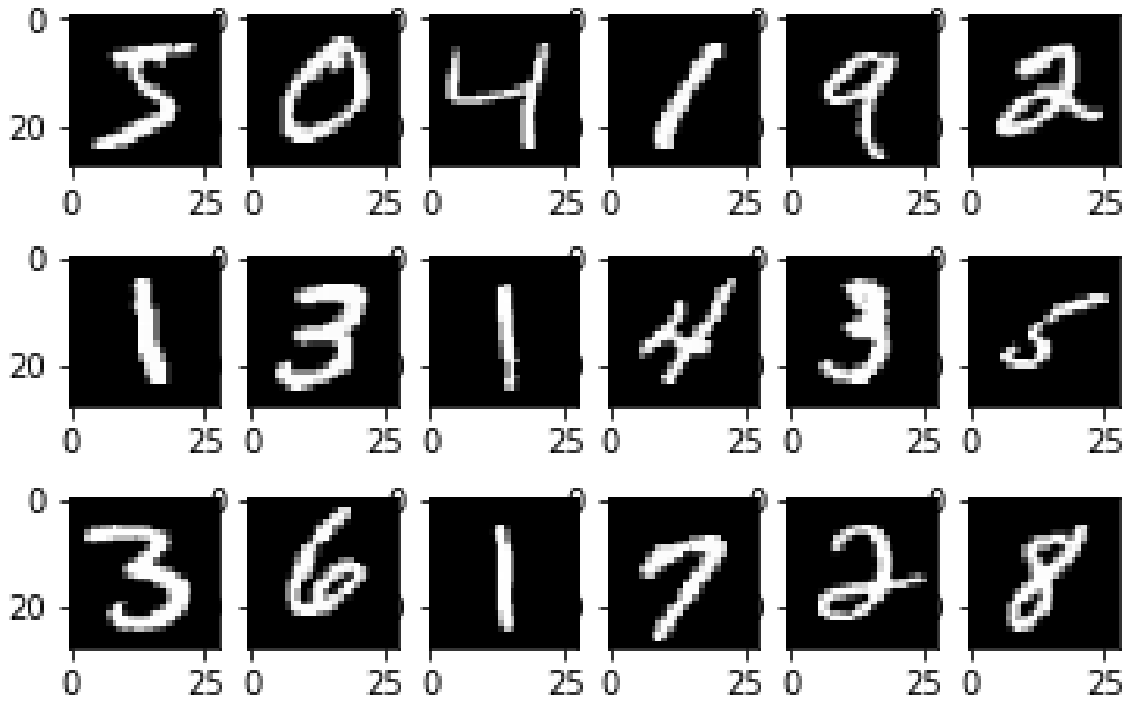
- $x_1, x_2, \ldots$: inputs
- $w_{1,1}, w_{1,2}, \ldots$: components of the **weight matrix** W
- $b_1, b_2, \ldots$: components of the **bias vector b**
- $f = \text{softmax}$: applied to the entire vector of values
- $\mathbf{y} = \text{softmax}(W\mathbf{x} + \mathbf{b})$: outputs of K neurons

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.
- From homework 2, this data set is not *linearly separable*:

x	t
-1	1
1	0
3	1

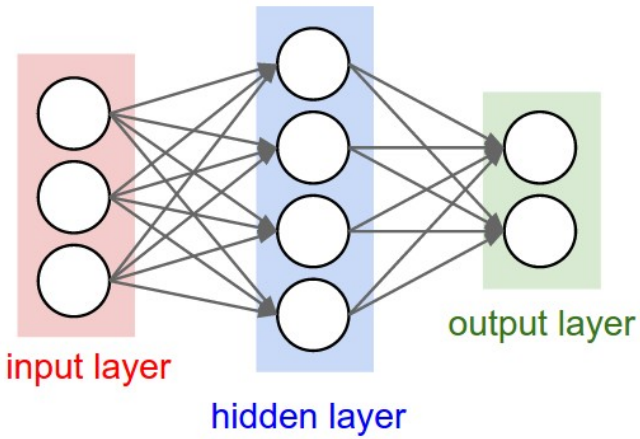
MNIST Digit Recognition (Tutorial 4)



- Input: An 28x28 pixel image
  - **x** is a vector of length 784
- Target: The digit represented in the image
  - **t** is a one-hot vector of length 10
- Model (from tutorial 4)
  - $\mathbf{y} = \text{softmax}(W\mathbf{x} + \mathbf{b})$

Adding a Hidden Layer

Two layer neural network



- Input size: 784 (number of features)
- Hidden size: 50 (we choose this number)
- Output size: 10 (number of classes)

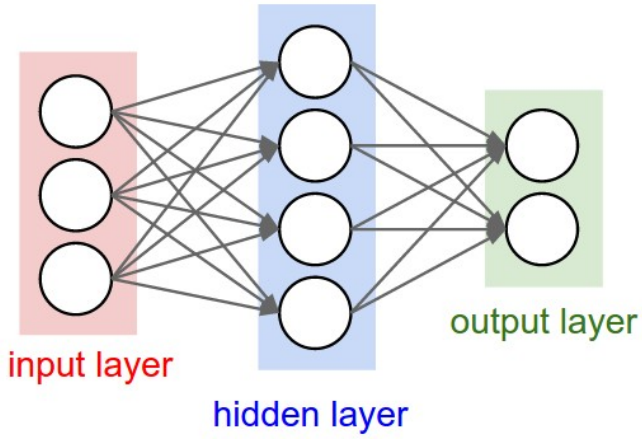
Side note about machine learning models

When discussing machine learning models, we usually

- first talk about **how to make predictions** assume the weights are trained
- *then* talk about how to traing the weights

Often the second step requires gradient descent or some other optimization method

Making Predictions: computing the hidden layer

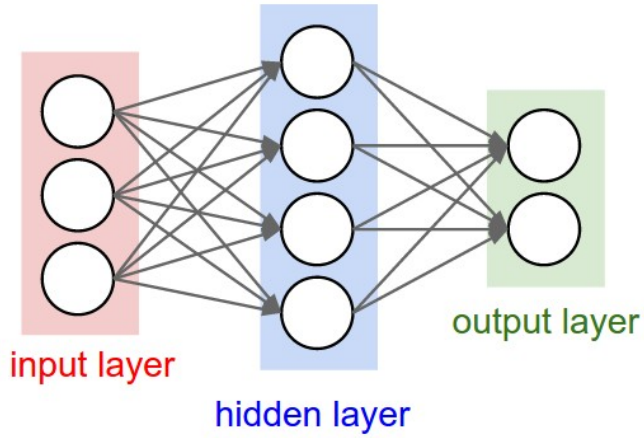


$$h_1 = f(\sum_{i=1}^{784} w_{1,i}^{(1)} x_i + b_1^{(1)})$$

$$h_2 = f(\sum_{i=1}^{784} w_{2,i}^{(1)} x_i + b_2^{(1)})$$

...

Making Predictions: computing the output (pre-activation)

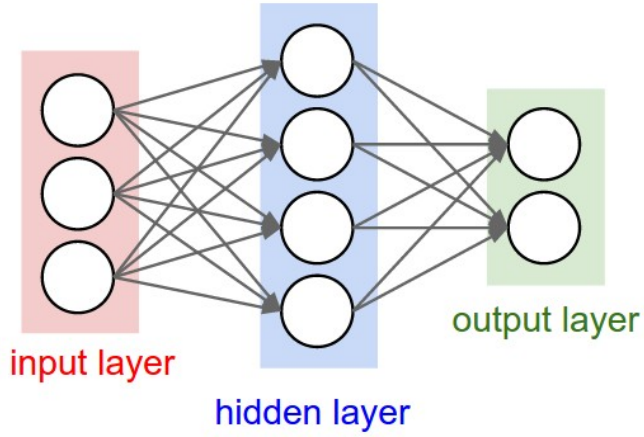


$$z_1 = \sum_{j=1}^{50} w_{1,j}^{(2)} h_j + b_1^{(2)}$$

$$z_2 = \sum_{j=1}^{50} w_{2,j}^{(2)} h_j + b_2^{(2)}$$

...

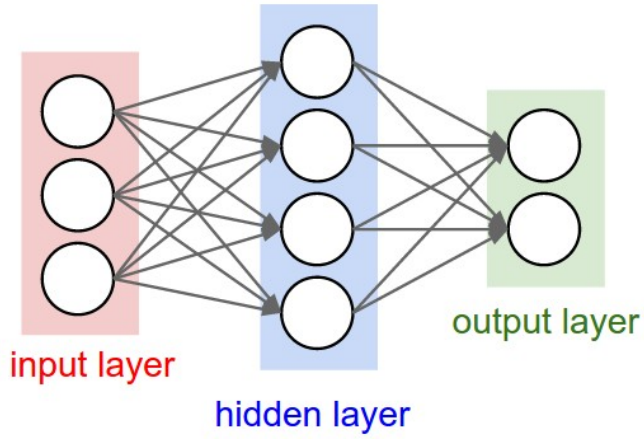
Making Predictions: applying the output activation



$$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ \cdots \\ z_{10} \end{bmatrix}$$

$$\mathbf{y} = \text{softmax}(\mathbf{z})$$

Making Predictions: Vectorized



$$\mathbf{h} = f(W^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$$

$$\mathbf{z} = f(W^{(2)}\mathbf{h} + \mathbf{b}^{(2)})$$

$$\mathbf{y} = \text{softmax}(\mathbf{z})$$

Example: Small Neural Network

From homework 2, this data set is not *linearly separable* (i.e. we can't correctly classify all 3 points using logistic regression, or using a single neuron)

x	t
-1	1
1	0
3	1

Can we come up with a neural network with two hidden units to solve this problem?

Use RELU activation.

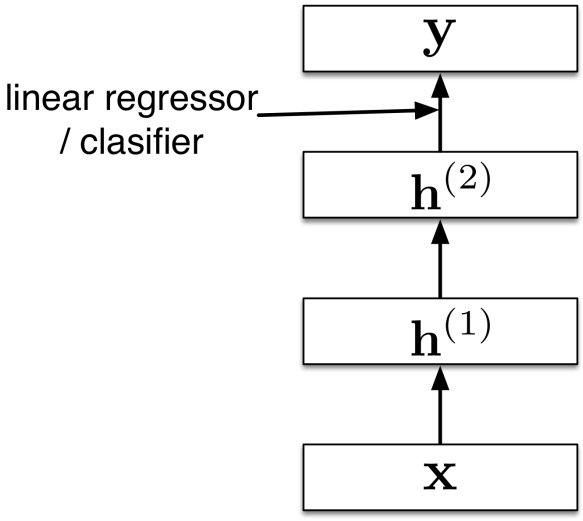
Example: Neural Network

$$h_1 = \text{relu}(-3x)$$
$$h_2 = \text{relu}(x)$$
$$y = \text{relu}(h_1 + h_2 - 2)$$

x	t	h_1	h_2	y
-1	1	3	0	1
1	0	0	1	0
3	1	0	3	1

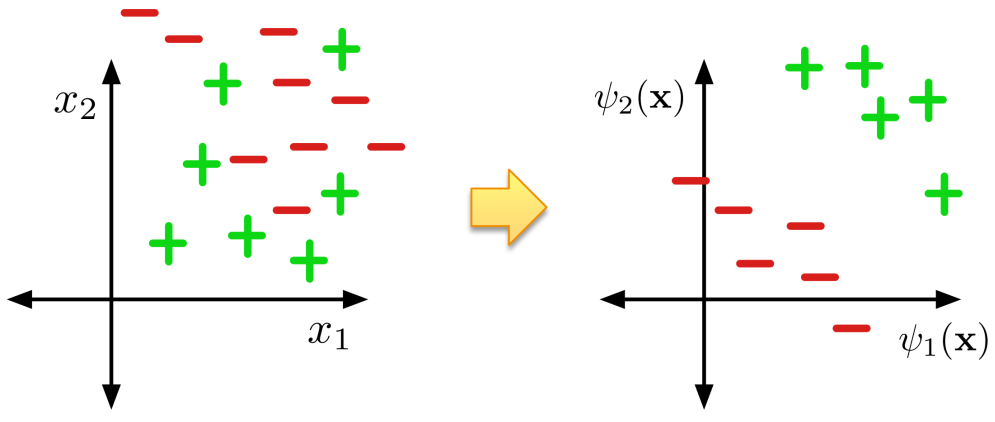
Feature Learning

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



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

The goal is for these features to become linearly separable:



Demo

Expressive Power: Linear Layers (No Activation Function)

- We've seen that there are some functions that linear classifiers can't represent. Are deep networks any better?
- Any sequence of *linear* layers (with no activation function) can be equivalently represented with a single linear layer.

$$\begin{aligned}\mathbf{y} &= \underbrace{W^{(3)}W^{(2)}W^{(1)}}\mathbf{x} \\ &= W'\mathbf{x}\end{aligned}$$

Deep linear networks are no more expressive than linear regression!

Expressive Power: MLP (nonlinear activation)

- 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!

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

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!

Backpropagation

Training Neural Networks

- How do we find good weights for the neural network?
- We can continue to use the loss functions:
  - cross-entropy loss for classification
  - square loss for regression
- The neural network operations we used (weights, etc) are continuous

We can use gradient descent!

Gradient Descent Recap

- Start with a set of parameters (initialize to some value)
- Compute the gradient $\frac{\partial \mathcal{E}}{\partial w}$ for each parameter (also $\frac{\partial \mathcal{E}}{\partial b}$)
  - This computation can often vectorized
- Update the parameters towards the negative direction of the gradient

Gradient Descent for Neural Networks

- Conceptually, the exact same idea!
- However, we have more parameters than before
  - Higher dimensional
  - Harder to visualize
  - More "steps"

Since $\frac{\partial \mathcal{E}}{\partial w}$, is the average of $\frac{\partial \mathcal{L}}{\partial w}$ across training examples, we'll focus on computing $\frac{\partial \mathcal{L}}{\partial w}$

Univariate Chain Rule

Recall: if $f(x)$ and $x(t)$ are univariate functions, then

$$\frac{d}{dt} f(x(t)) = \frac{df}{dx} \frac{dx}{dt}$$

Univariate Chain Rule for Logistic Least Square

Recall: Univariate logistic least squares model (from homework 2)

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

Let's compute the loss derivative

Univariate Chain Rule Computation (1)

How you would have done it in calculus class (and maybe homework 2):

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

Univariate Chain Rule Computation (2)

Similarly for $\frac{\partial \mathcal{L}}{\partial b}$

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

Univariate Chain Rule Computation (2)

Similarly for $\frac{\partial \mathcal{L}}{\partial b}$

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

Q: What are the disadvantages of this approach?

A More Structured Way to Compute the Derivatives

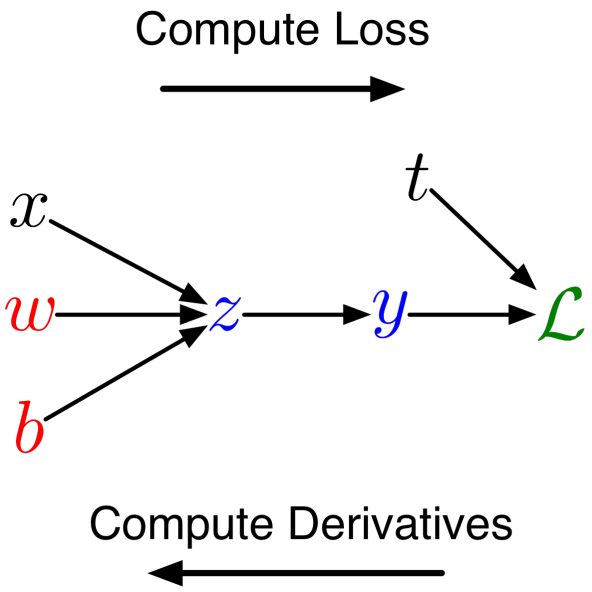
$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

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

Less repeated work; easier to write a program to efficiently compute derivatives

Computation Graph

We can diagram out the computations using a *computation graph*.



The *nodes* represent all the inputs and computed quantities

The *edges* represent which nodes are computed directly as a function of which other nodes.

Chain Rule (Error Signal) Notation

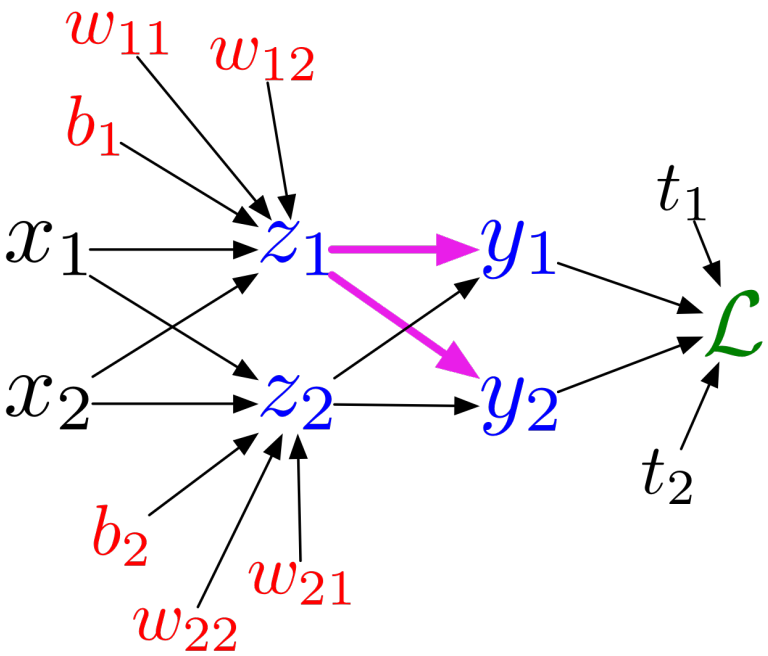
- Use $\overline{y}$ to denote the derivative $\frac{d\mathcal{L}}{dy}$
  - sometimes called the **error signal**
- This notation emphasizes that the error signals are just values our program is computing (rather than a mathematical operation).
- This is notation introduced by Prof. Roger Grosse, and not standard notation

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

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

Multiclass Logistic Regression Computation Graph

In general, the computation graph *fans out*:



$$z_l = \sum_j w_{lj} x_j + b_l$$

$$y_k = \frac{e^{z_k}}{\sum_l e^{z_l}}$$

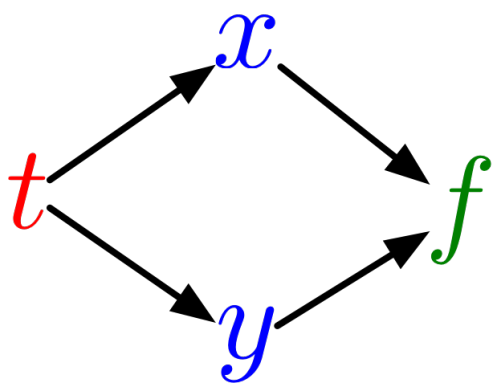
$$\mathcal{L} = -\sum_k t_k \log y_k$$

There are multiple paths for which a weight like w_{11} affects the loss L.

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

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

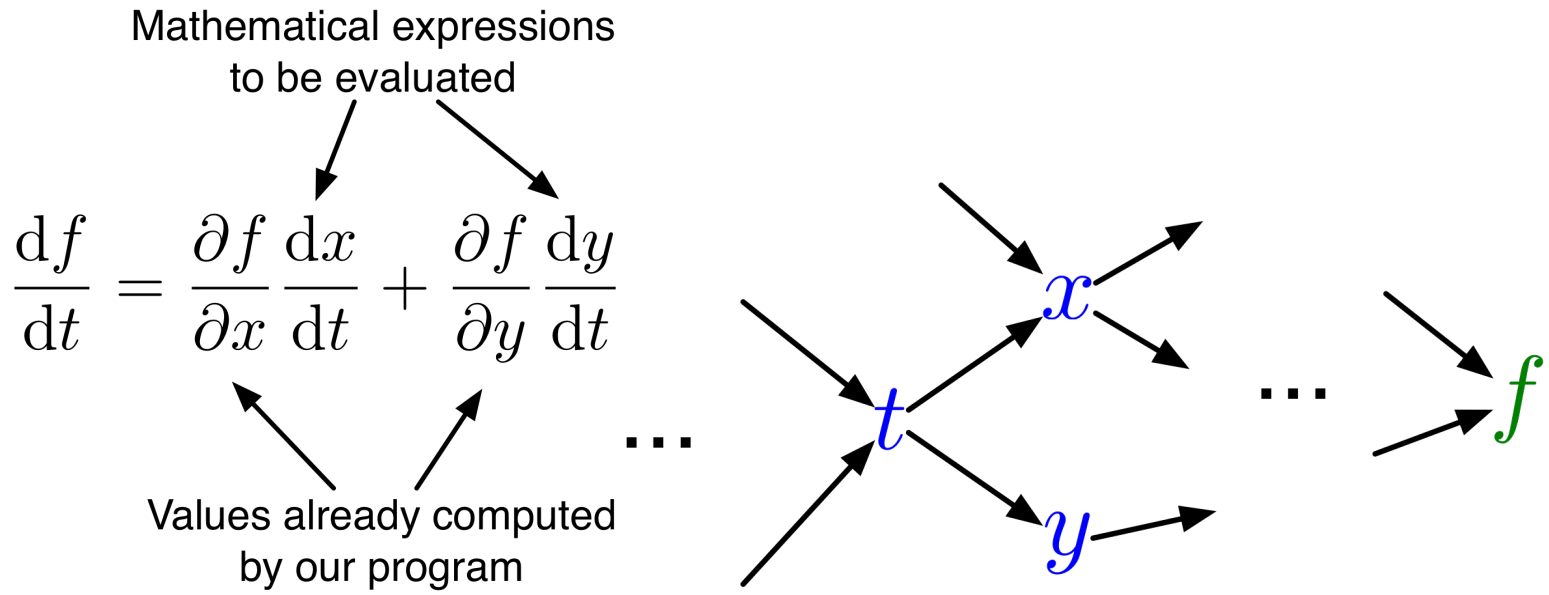


Multivariate Chain Rule Example

If $f(x, y) = y + e^{xy}$, $x(t) = \cos t$ and $y(t) = t^2 \ldots$

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

Multivariate Chain Rule Notation



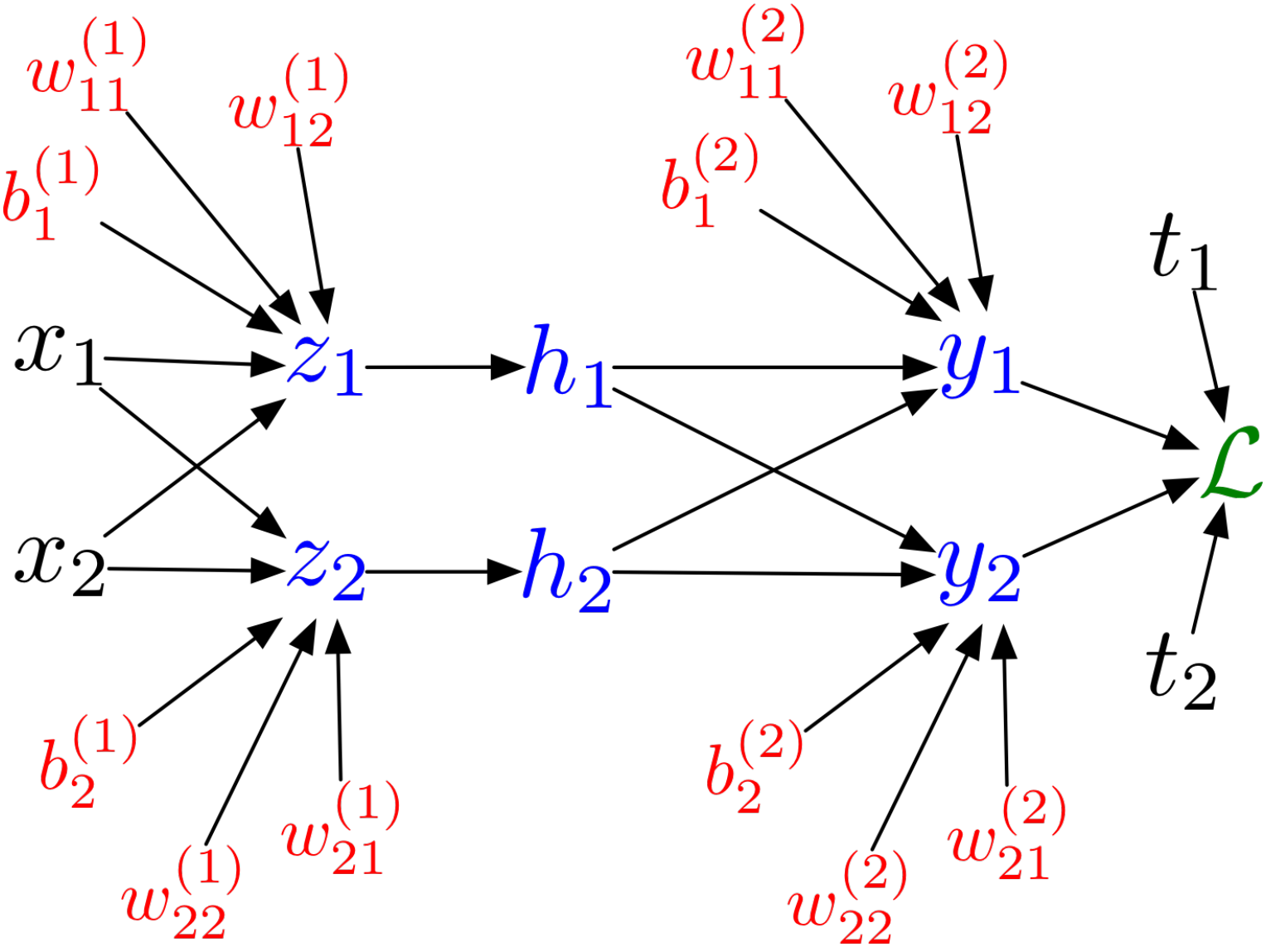
In our notation

$$\overline{t} = \overline{x}\frac{dx}{dt} + \overline{y}\frac{dy}{dt}$$

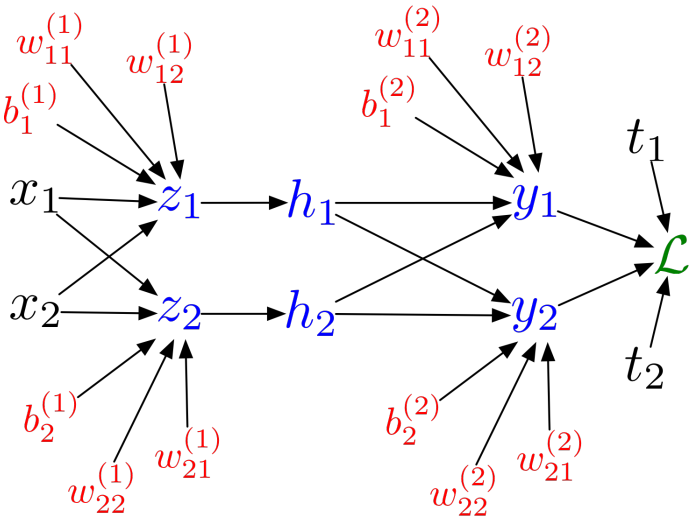
The Backpropagation Algorithm

- Backpropagation is an *algorithm* to compute gradients efficiency
  - Forward Pass: Compute predictions (and save intermediate values)
  - Backwards Pass: Compute gradients
- The idea behind backpropagation is very similar to *dynamic programming*
  - Use chain rule, and be careful about the order in which we compute the derivatives

Backpropagation Example (on the board)



Backpropagation for a MLP



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:

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

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

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

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

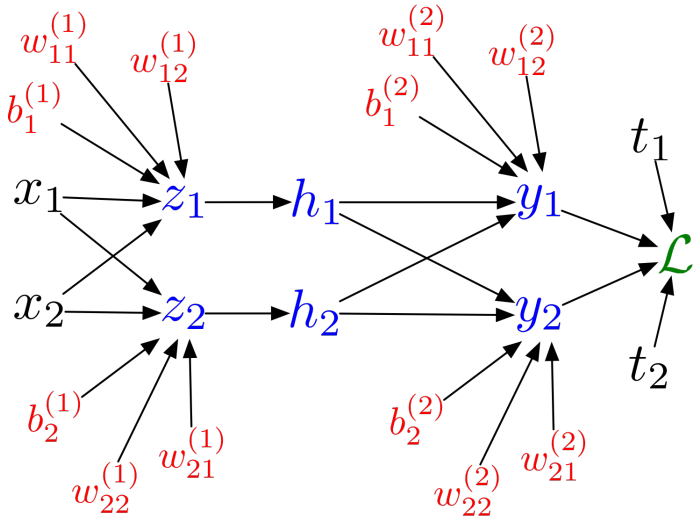
$$\overline{h_i} = \sum_k \overline{y_k} w_{ki}^{(2)}$$

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

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

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

Backpropagation for a MLP (Vectorized)



Forward pass:

$$\mathbf{z} = W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$

$$\mathbf{h} = \sigma(\mathbf{z})$$

$$\mathbf{y} = W^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$$

$$\mathcal{L} = \frac{1}{2}||\mathbf{y} - \mathbf{t}||^2$$

Backward pass:

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

$$\overline{\mathbf{y}} = \overline{\mathcal{L}}(\mathbf{y} - \mathbf{t})$$

$$\overline{W^{(2)}} = \overline{\mathbf{y}}\mathbf{h}^T$$

$$\overline{\mathbf{b}^{(2)}} = \overline{\mathbf{y}}$$

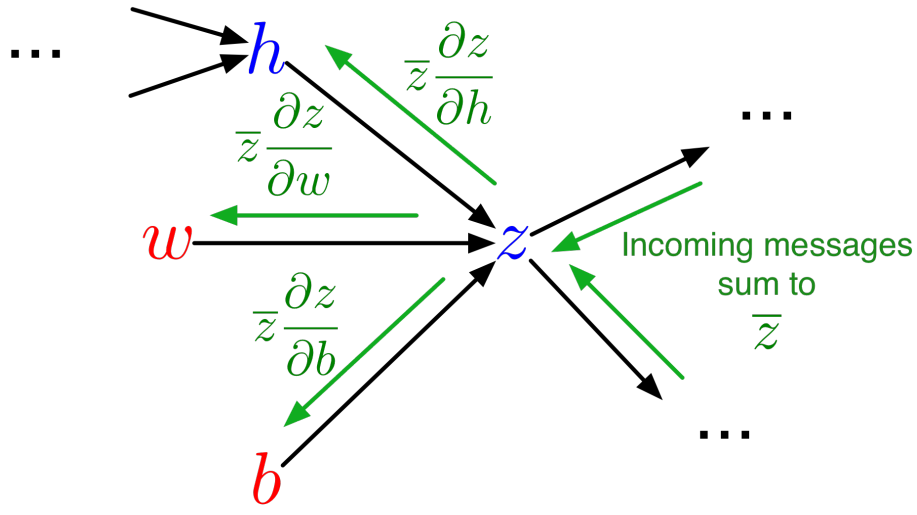
$$\overline{\mathbf{h}} = W^{(2)^T}\overline{y}$$

$$\overline{\mathbf{z}} = \overline{\mathbf{h}} \circ \sigma'(\mathbf{z})$$

$$\overline{W^{(1)}} = \overline{\mathbf{z}}\mathbf{x}^T$$

$$\overline{\mathbf{b}^{(1)}} = \overline{\mathbf{z}}$$

Implementing Backpropagation



Forward pass: Each node...

- receives messages (inputs) from its children
- uses these messages to compute its own values
- passes messages to its parents

Backward pass: Each node...

- receives messages (error signals) from its children
- uses these messages to compute error signal
- passes messages to its parents

This algorithm provides **modularity**!

In PyTorch (from tutorial 4)

```
model = nn.Linear(784, 10) # classification model
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.005)

zs = model(xs)              # forward pass
loss = criterion(zs, ts) # compute the loss (cost)
loss.backward()             # backwards pass (error signals)
optimizer.step()            # update the parameters
optimizer.zero_grad()       # a clean up step
```

Backpropagation in practice

- 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 (biologically) 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?

Reference

Most of the backpropagation slides are based on the works of:

- Roger Grosse
- Jimmy Ba