CSC413 Neural Networks and Deep Learning

Lecture 2: Multi-layer Feedforward NN and Backpropagation

January 16 / 18, 2024

Table of Contents

Lecture Plan

Last week:

- Review of linear models
  - linear regression
  - linear classification (logistic regression)
- Gradient descent to train these models

This week:

- Why we need nonlinearities and multi-layer feedforward neural networks (multilayer Perceptron)
- How to train a multi-layer neural network using **backpropagation**

Section 1

Limits of Linear Models for Binary Classification

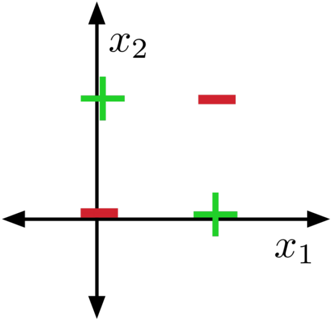
XOR example

Recall that a linear classifier has the following form:

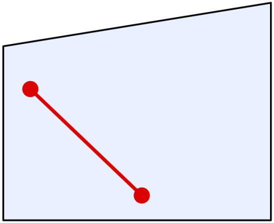
$$y = \sigma(w^\top x + b),$$

with w being the weights, b being the bias, x being the input, and $\sigma(\cdot)$ is the activation function (for example, a sigmoid).

- A linear classifier is very limited in expressive power.
- XOR is an example of a function that is not linearly separable.



Convex Sets



A set S is convex if any line segment connecting points in S lies in S.

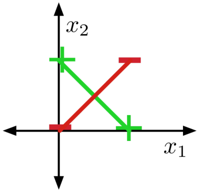
$\mathbf{x_1}, \mathbf{x_2} \in S \rightarrow \lambda \mathbf{x_1} + (1 - \lambda)\mathbf{x_2} \in S$ for $0 \leq \lambda \leq 1$

A simple inductive argument shows that for $\mathbf{x_1}, \ldots, \mathbf{x_N} \in S$, the **weighted average** or **convex combination** lies in the set:

$\lambda_1 \mathbf{x_1} + \cdots + \lambda_N \mathbf{x_N} \in S$ for $\lambda_1 + \cdots + \lambda_N = 1$

XOR not linearly separable

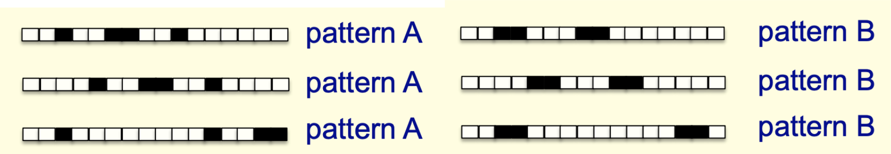
- Half-spaces are convex
- Suppose there were some feasible hypothesis. If the positive examples are in the positive half-space, because of convexity of a half-space, the green line segment must be in that half-space as well.
- Similarly, red line segment must lie within the negative half-space.



- But the intersection of these two line segments can't lie in both positive and negative half-spaces, as a point is either positive or negative, but not both. This is a contradiction!

A more troubling example

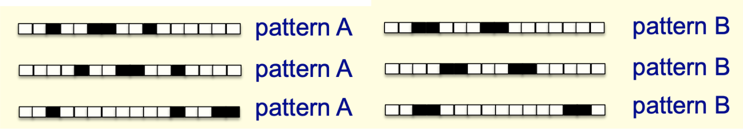
These images represent 16-dimensional vectors. Want to distinguish patterns A and B in all possible translations (with wrap-around).



Q: What is the difference between A and B?

We can show that a linear model cannot classify all translations of patterns A and B correctly.

A more troubling example



- Suppose there's a feasible solution. Focus on Pattern A:
  - If $\mathbf{x}_1$ and $\mathbf{x}_2$ are two translations of pattern A and they are correctly classified as pattern A, because of convexity of half-spaces induced by a linear model, their convex combination is classified as pattern A too.
  - We can extend this argument for all possible translations of pattern A.
  - The average of all translations of A, which is a convex combination of them, is the vector $(0.25, 0.25, \cdots, 0.25)$. This point is also classified as pattern A.
- Now focus on Pattern B. With a similar argument, the average of all translations of B is also $(0.25, 0.25, \cdots, 0.25)$. This point must also be classified as pattern B.
- The same point is classified as pattern A and B. Contradiction!

(Nonlinear) Feature Maps

Sometimes, we can overcome this limitation with **nonlinear feature maps**

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

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!)

... but generally, it can be hard to pick good basis functions.

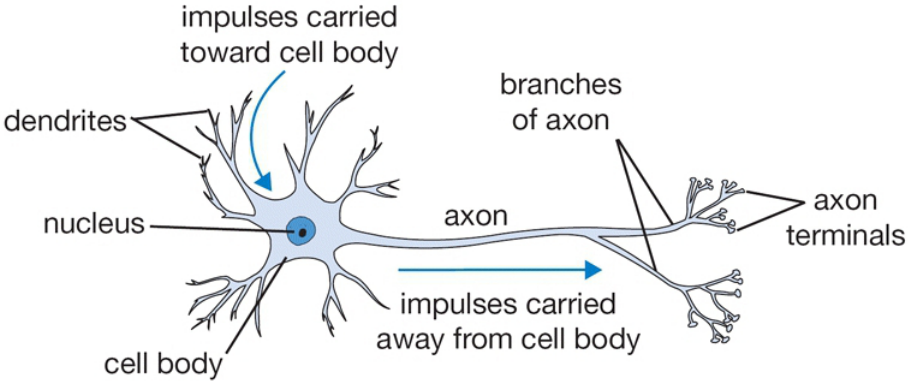
We'll use neural nets to learn nonlinear hypotheses directly.

Section 2

From Brain to Artificial Neural Networks

Neuron

Our brain has $\sim 10^{11}$ neurons, each of which communicates (is connected) to $\sim 10^4$ other neurons



Neuron Anatomy

- The **dendrites**, which are connected to other cells that provide 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.

Inspiration: The Brain

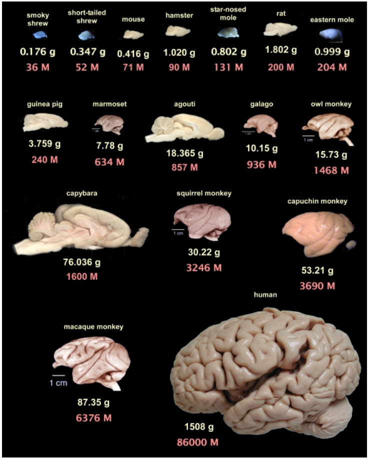
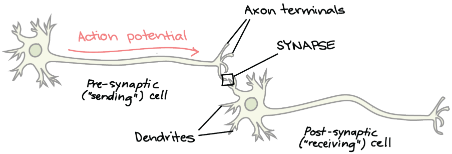


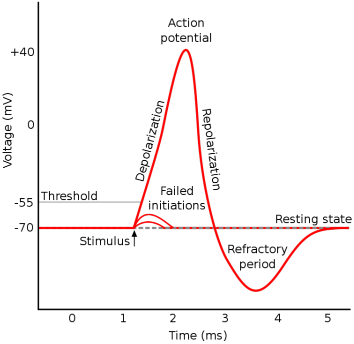
Figure 1: Brain mass and total number of neurons for the mammalian species.

Image credit: Suzana Herculano-Houzel, The Human Brain in Numbers: A Linearly Scaled-up Primate Brain, 2009.

What does a neuron do?

A neuron receives input signals from other neurons and accumulate voltage. If the accumulated voltage passes a threshold, it fires spiking responses. This spreads along the axon to the synapse, then to the next neurons.





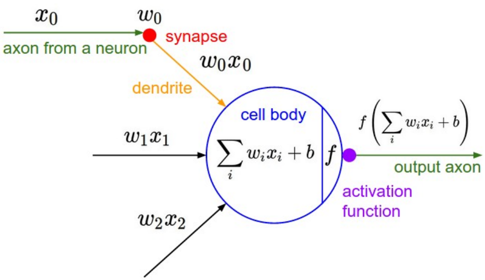
Right image credit: https://en.wikipedia.org/wiki/Action_potential

What makes a neuron fire?

Neurons can fire in response to. . .

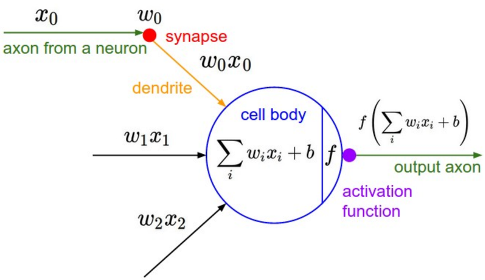
- retinal cells
- certain edges, lines, angles, movements
- hands and faces (in primates)
- specific people (in humans)
  - The existence of these “grandmother cells” (or “Jennifer Aniston” cell) is contested.

Modeling Individual Neurons



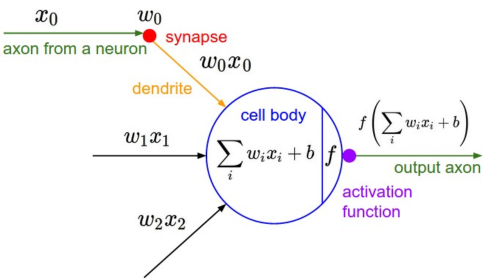
- $x_1, x_2, \ldots =$ inputs to the neuron
- $w_1, w_2, \ldots =$ the neuron's **weights**
- $b =$ the neuron's **bias**
- $f =$ an **activation function**
- $f(\sum_i x_i w_i + b) =$ the neuron's **activation** (output)

Linear Models 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
- $y = \sum_i x_i w_i + b = \mathbf{w}^T \mathbf{x} + b$

Logistic Regression Model (for Binary Classification) as a Single Neuron



- $x_1, x_2, \ldots$: inputs
- $w_1, w_2, \ldots$: components of the **weight vector w**
- b : the **bias**
- $f = \sigma$
- $y = \sigma(\sum_i x_i w_i + b) = \sigma(\mathbf{w}^T \mathbf{x} + b)$
- If we use the cross-entropy loss function to train this neuron, this becomes the same as the logistic regression model.

Logistic Regression Models (for Multi-Class Classification) as a Neural Network

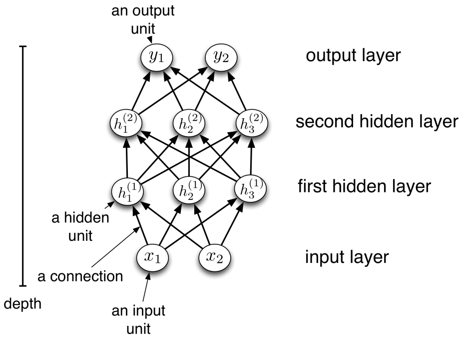
We use K neurons (one for each class):

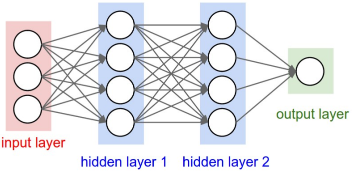
- $x_1, x_2, \ldots$: inputs
- $w_{1,1}, w_{1,2}, \ldots$: components of the **weight matrix** W
- $b_1, b_2, \ldots$: components of the **bias vector** $\mathbf{b}$
- $f = \text{softmax}$: applied to the entire vector of values
- $\mathbf{y} = \text{softmax}(W\mathbf{x} + \mathbf{b})$: outputs of K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

Section 3

Multilayer Perceptrons (Feedforward Fully Connected Neural Networks)

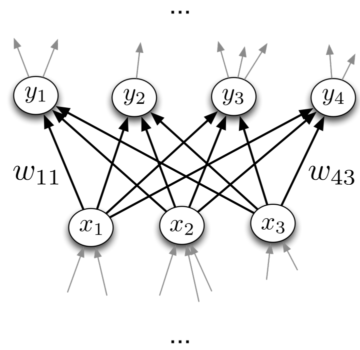
Multilayer Perceptrons (Feedforward Fully Connected Neural Networks)





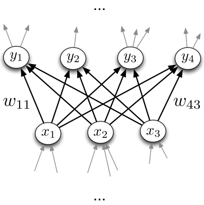
- We can connect lots of units together into a **directed acyclic graph**.
- Typically, units are grouped together into **layers**.
  - An **input layer**: feed in input features (e.g. like retinal cells in your eyes)
  - A number of **hidden layers**
  - An **output layer**: interpret output like a "grandmother cell"
- This gives a **feed-forward neural network**.

Multilayer Perceptrons (Feedforward FC Neural Networks)



- Each hidden layer i connects N_{i-1} input units to N_i output units.
- In the simplest case, all input units are connected to all output units. We call this a **fully connected layer**. We will consider other layer types later.
  - The inputs and outputs for a layer are distinct from the inputs and outputs to the network.

Multilayer Perceptrons (Feedforward FC Neural Networks)



- If we need to compute $M[= N_i]$ outputs from $N = [N_{i-1}]$ inputs, we can do so in parallel using matrix multiplication. This means we will be using a $M \times N$ weight matrix.
- The output units are a function of the input units:

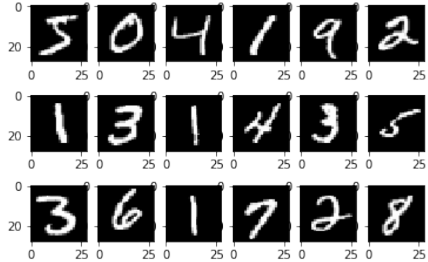
$$y = f(x) = \sigma(Wx + b)$$

- A multilayer network consisting of fully connected layers is called a **multilayer perceptron**. Despite the name, it has nothing to do with the Perceptron algorithm.

But what do these 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
  - like a "grandmother cell"
- Use $h_i^{(k)}$ to denote a unit in the hidden layer
  - difficult to interpret

MNIST Digit Recognition

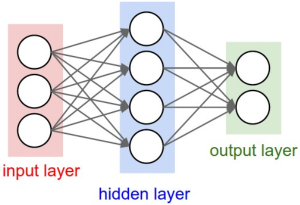


With a logistic regression model, we would have:

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

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)

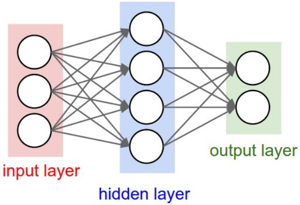
Side note about machine learning models

When discussing machine learning and deep learning models, we usually

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

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

Making Predictions: computing the hidden layer

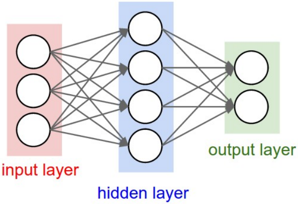


$$h_1 = f(\sum_{i=1}^{784} w_{1,i}^{(1)} x_i + b_1^{(1)})$$

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

$\ldots$

Making Predictions: computing the output (pre-activation)

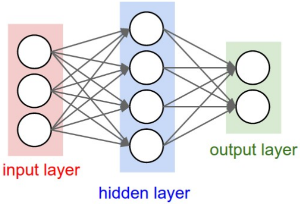


$$z_1 = \sum_{j=1}^{50} w_{1,j}^{(2)} h_j + b_1^{(2)}$$

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

$\dots$

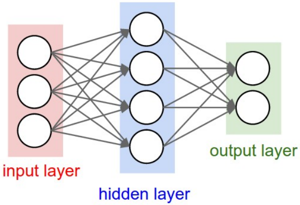
Making Predictions: applying the output activation



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

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

Making Predictions: Vectorized



$$
\begin{aligned}
\mathbf{h} &= f(W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \\
\mathbf{z} &= W^{(2)}\mathbf{h} + \mathbf{b}^{(2)} \\
\mathbf{y} &= \mathrm{softmax}(\mathbf{z})
\end{aligned}
$$

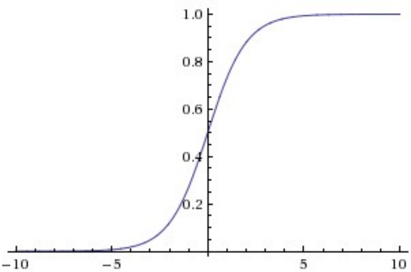
Activation Functions: common choices

Common Choices:

- Sigmoid activation
- Tanh activation
- ReLU activation

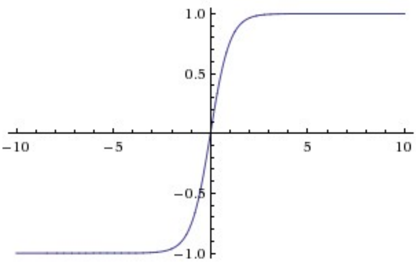
Rule of thumb: Start with ReLU activation. If necessary, try tanh.

Activation Function: Sigmoid



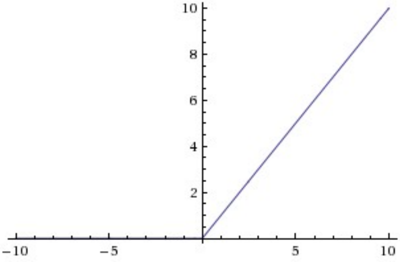
- somewhat problematic due to gradient signal
- all activations are positive

Activation Function: Tanh



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

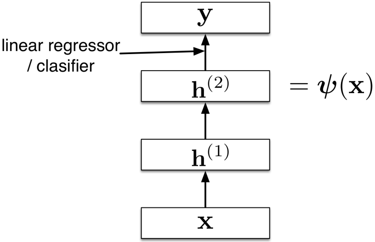
Activation Function: ReLU



- 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

Feature Learning

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



The goal is for these features to become linearly separable:

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.

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

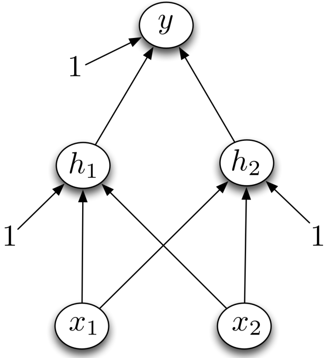
- Deep *linear* networks are no more expressive than linear models.
- But the dynamics of training can be different than a single layer linear model.
- We need to have nonlinearities to increase expressivity of NN.

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!

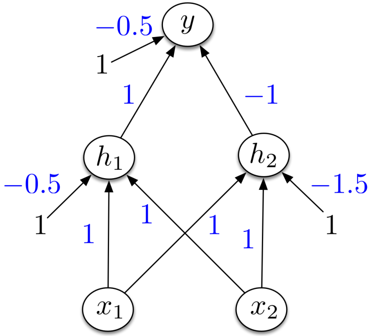
Designing a network to classify XOR

Assume hard threshold activation function



Note that x_1 XOR x_2 = [x_1 OR x_2] AND [NOT (x_1 AND x_2)]

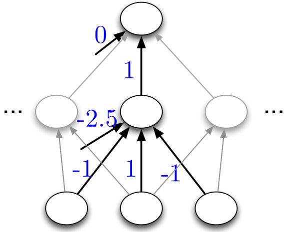
Designing a network to classify XOR



- h_1 computes $\mathbb{I}[x_1 + x_2 - 0.5 > 0]$
    - i.e. x_1 OR x_2
- h_2 computes $\mathbb{I}[x_1 + x_2 - 1.5 > 0]$
    - i.e. x_1 AND x_2
- y computes $\mathbb{I}[h_1 - h_2 - 0.5 > 0] \equiv \mathbb{I}[h_1 + (1 - h_2) - 1.5 > 0]$
    - i.e. h_1 AND (NOT h_2) $= x_1$ XOR x_2

Expressive Power: Universality for binary inputs and targets

x_1	x_2	x_3	t
	⋮		⋮
-1	-1	1	-1
-1	1	-1	1
-1	1	1	1
	⋮		⋮

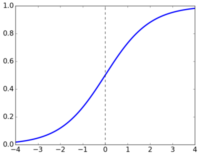


- Hard threshold hidden units, linear output
- Strategy: 2^D hidden units, each of which responds to one particular input configuration
- Only requires one hidden layer, though it needs to be extremely wide.

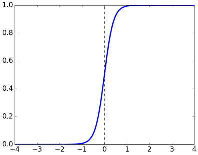
Expressive Power

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

$y = \sigma(x)$



$y = \sigma(5x)$

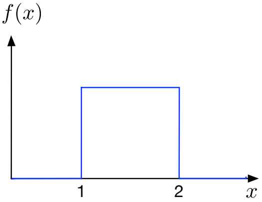


- This is good: logistic units are differentiable, so we can train them with gradient descent.

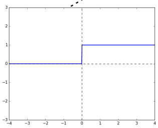
Expressive Power

Let us do some exercises . . .

- Q: How can we represent the function that takes value of $+1$ in $x \in [1, 2]$ and 0 elsewhere using a simple NN with *hard threshold* activation function?

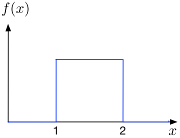


$$f(x) = w_1\phi(x - b_1) + w_2\phi(x - b_2)$$

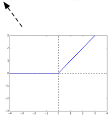


Let us do some exercises . . .

- Q: How can we *approximately* represent the function that takes value of $+1$ in $x \in [1, 2]$ and 0 elsewhere using a simple NN with *ReLU* activation function?



$$f(x) \approx w_1\phi(v_1(x - b_1)) + w_2\phi(v_2(x - b_2)) + \ldots$$



Limits of universality results

- You may need to represent an exponentially large network.
- How can you find the appropriate weights to represent a given function?
- If you can learn any function, you might just overfit.
- We desire a *compact* representation.

Computing XOR Demo

Demo: https://playground.tensorflow.org/

Section 4

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 Least Squares with a Logistic 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 derivative

Univariate Chain Rule Computation (1)

How you would have done it in calculus class

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

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

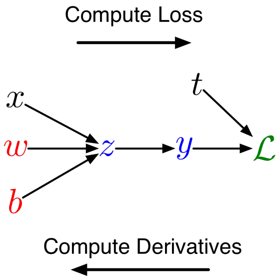
$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

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

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

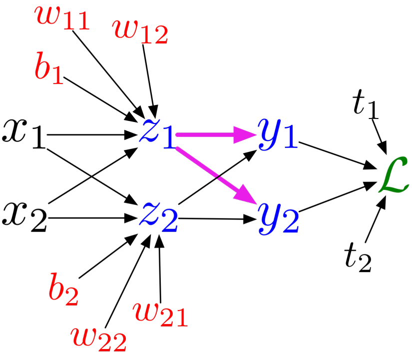
- Use $\overline{y}$ to denote the derivative $\frac{d\mathcal{L}}{dy}$
  - sometimes called the **error signal**
- This notation emphasizes that the error signals are just values our program is computing (rather than a mathematical operation).
- This is notation introduced by Prof. Roger Grosse, and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

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

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

Multiclass Logistic Regression Computation Graph

In general, the computation graph *fans out*:



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

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

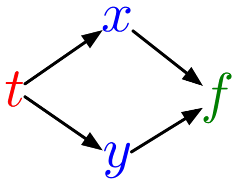
$$\mathcal{L} = -\sum_k t_k \log y_k$$

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 $y(t) = t^2 \ldots$

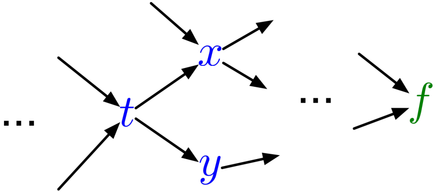
$$\begin{aligned}\frac{d}{dt} f(x(t), y(t)) &= \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt} \\ &= (ye^{xy}) \cdot (-\sin t) + (1 + xe^{xy}) \cdot 2t\end{aligned}$$

Multivariate Chain Rule Notation

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



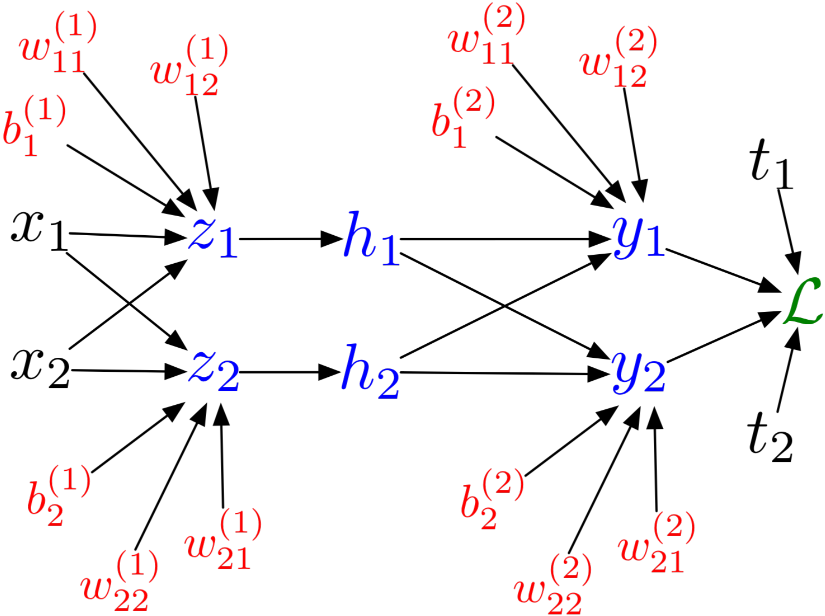
In our notation

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

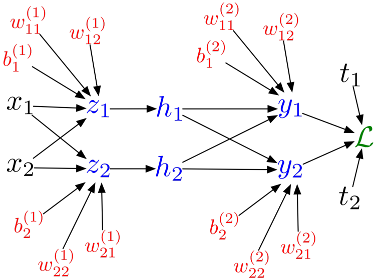
The Backpropagation Algorithm

- 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:

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

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

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

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

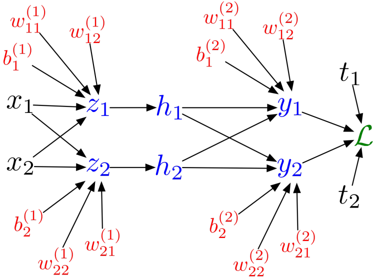
$$\overline{h_i} = \sum_k \overline{y_k} w_{ki}^{(2)}$$

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

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

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

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:

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

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

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

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

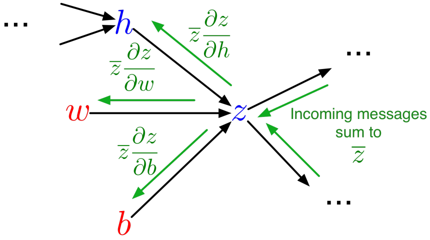
$$\overline{\mathbf{h}} = W^{(2)\,T}\overline{y}$$

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

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

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

Implementing Backpropagation



Forward pass: Each node...

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

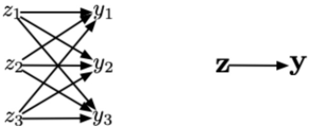
Backward pass: Each node...

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

This algorithm provides **modularity**!

Backpropagation in Vectorized Form

- Consider this computation graph:



- 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):

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

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?

Section 5

What to do this week?

What to do this week?

- Programming HW 1 is out.
- Math HW 1 is out too.
- Attend your tutorial session after the lecture!
- The HWs are due next Friday.