CS-UY 4563: Lecture 16
Neural Networks cont.

NYU Tandon School of Engineering, Prof. Christopher Musco

LAB 6

Lab `lab_mnist_partial.ipynb` due **next Thursday, 4/9.**

- Covers kernel logistic regression and SVMs, which should be useful in many projects.
- Requires `Tensorflow` (easiest way to load MNIST data).

Key Concept

Approach in prior classes:

- Choose good features or a good kernel.
- Use optimization to find best model given those features.

Neural network approach:

- Learn good features and a good model simultaneously.

Input: $\vec{x} = x_1, \ldots, x_{N_I}$

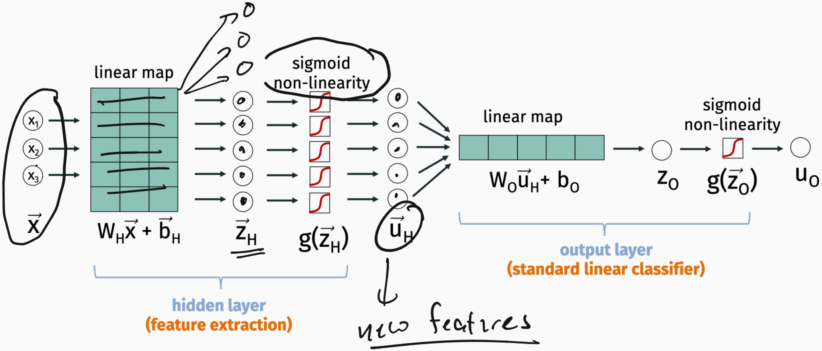
Model: $f(\vec{x}, \Theta)$:

- $\vec{z}_H \in \mathbb{R}^{N_H} = \mathsf{W}_H \vec{x} + \vec{b}_h$.
- $\vec{u}_H = [\vec{z}_H > 0]$
- $z_O \in \mathbb{R} = \mathsf{W}_O \vec{u}_H + b_O$
- $u_O = [z_O > 0]$

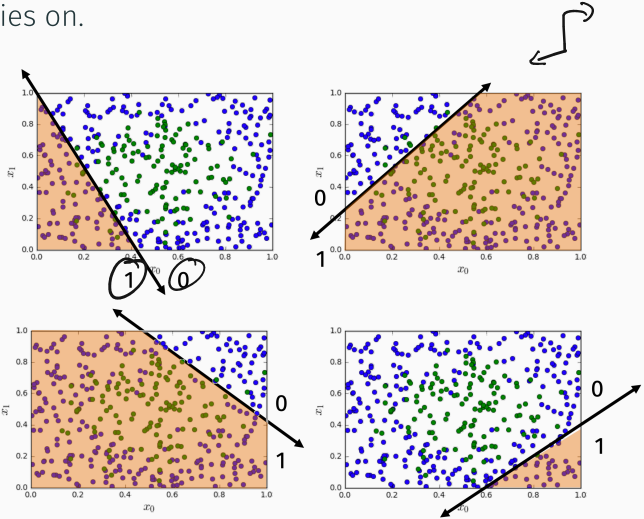
Parameters: $\Theta = [\mathsf{W}_H \in \mathbb{R}^{N_H \times N_I}, \vec{b}_H \in \mathbb{R}^{N_H}, \mathsf{W}_O \in \mathbb{R}^{1 \times N_H}, b_O \in \mathbb{R}]$.

W_H, W_O are <u>weight matrices</u> and $\vec{b}_H$, b_O are <u>bias</u> terms that account for the intercepts of our linear functions.

Function f which maps $\vec{x}$ to a class label u_O.

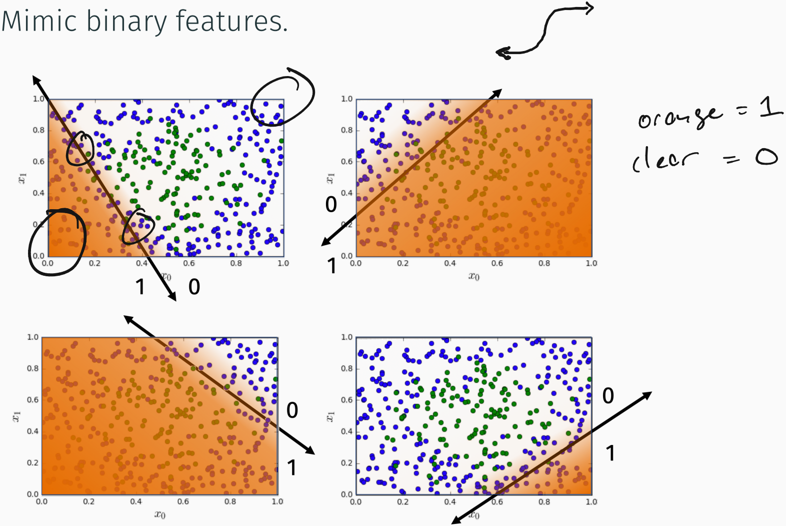


Features learned using step-function activation are binary, depending on which side of a set of learned hyperplanes each point lies on.



FEATURE EXTRACTION

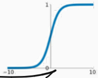
Features learned using sigmoid activation are real valued in $[0, 1]$. Mimic binary features.



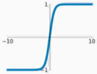
Things we can change in this basic classification network:

- More or less hidden variables.
- Different non-linearity/activation function.
- Different loss function (more on that next class).
- More hidden layers (allows for learning hierarchical features).

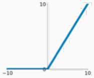
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$



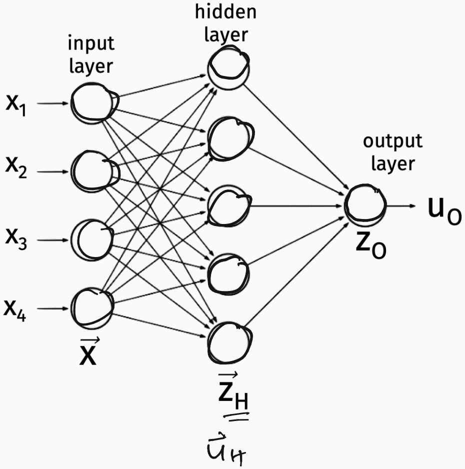
tanh
$\tanh(x)$



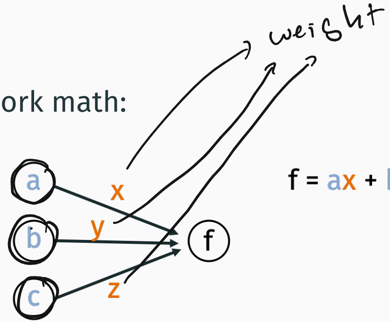
ReLU
$\max(0, x)$



Another common diagram for a 2-layered network:

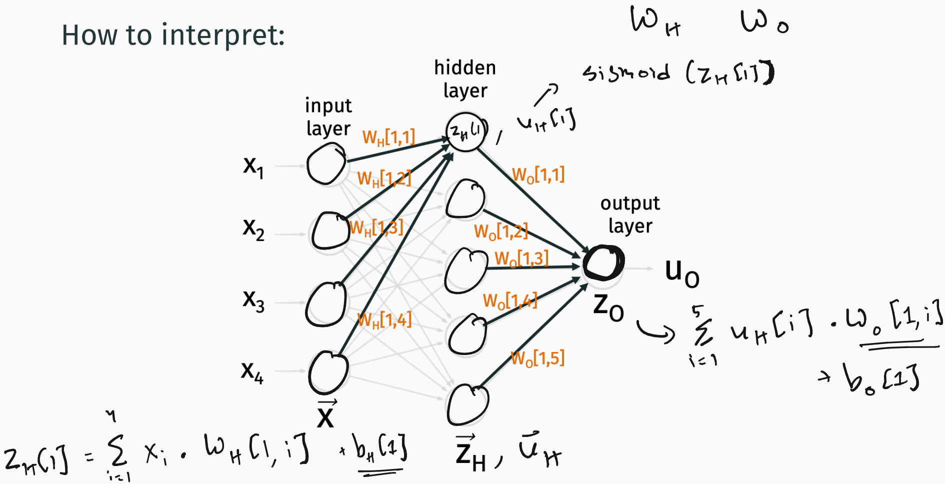


Neural network math:



f = ax + by + cz

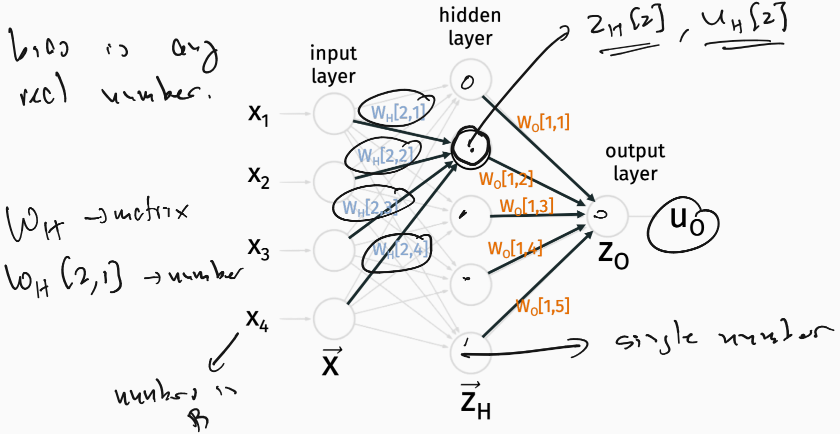
How to interpret:



W_H and W_O are our weight matrices from before.

Note: This diagram does not explicitly show the bias terms or the non-linear activation functions.

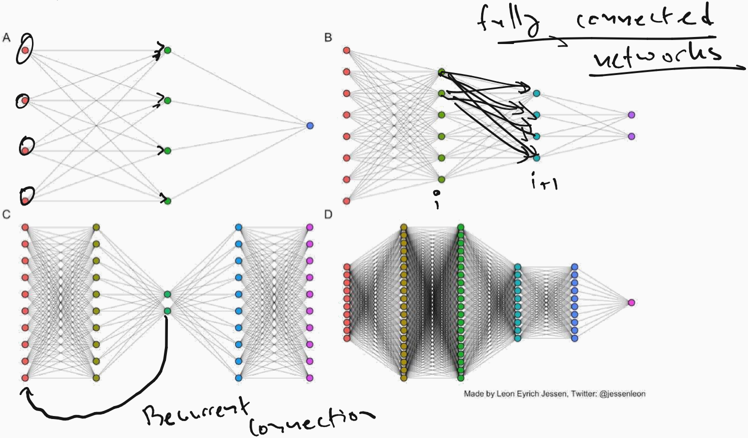
How to interpret:



W_H and W_O are our weight matrices from before.

Note: This diagram depicts a network with **"fully-connected"** layers. Every variable in layer i is connected to every variable in layer $i + 1$.

Effective way of visualize “architecture” of a neural network:

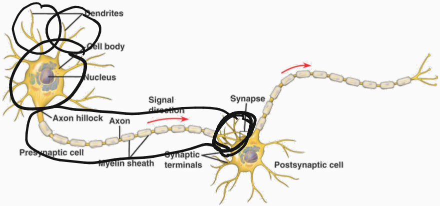


Visualize number of variables, types of connections, number of layers and their relative sizes.

These are all **feedforward** neural networks. No backwards (**recurrent**) connections.

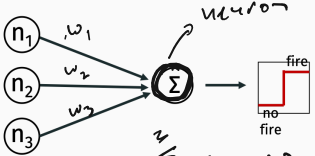
SOME HISTORY AND MOTIVATION

Simplified model of the brain:



Dendrites: Input electrical current from other neurons.
Axon: Output electrical current to other neurons.
Synapse: Where these two connect.

A neuron "fires" (outputs non-zero electric charge) if it receives enough cumulative electrical input from all neurons connected to it.



$\sum_i n_i \cdot w_i \geq \text{threshold}$

Output charge can be positive or negative (excitatory vs. inhibitory).

Inspired early work on neural networks:

- 1940s Donald Hebb proposed a Hebbian learning rule for how brains neurons change over time to allow learning.
- 1950s Frank Rosenblatt's Perceptron is one of the first "artifical" neural networks.
- Continued work throughout the 1960s.

Main issue with neural network methods: They are hard to train. Generally require a lot of computation power. Also pretty finicky: user needs to be careful with initialization, regularization, etc. when training. Often requires a lot of experimentation to get right.

EARLY NEURAL NETWORK EXPLOSION

Around 1985 several groups (re)-discovered the **backpropagation algorithm** which allows for efficient training of neural nets via **gradient descent**. Along with increased computational power this lead to a resurgence of interest in neural network models.

Backpropagation Applied to Handwritten Zip Code Recognition

Y. LeCun
B. Boser
J. S. Denker
D. Henderson
R. E. Howard
W. Hubbard
L. D. Jackel
AT&T Bell Laboratories Holmdel, NJ 07733 USA

The ability of learning networks to generalize can be greatly enhanced by providing constraints from the task domain. This paper demonstrates how such constraints can be integrated into a backpropagation network through the architecture of the network. This approach has been successfully applied to the recognition of handwritten zip code digits provided by the U.S. Postal Service. A single network learns the entire recognition operation, going from the normalized image of the character to the final classification.

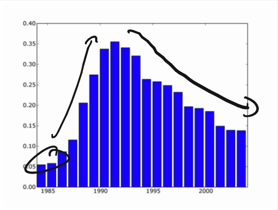
Very good performance on problems like digit recognition.

In the 1990s and early 2000s, kernel methods, SVMs, and probabilistic methods began to dominate the literature in machine learning:

- Work well “out of the box”.
- Relatively easy to understand theoretically.
- Not too computationally expensive for moderately sized datasets.

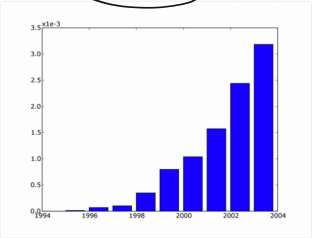
Fun blog post to check out from 2005:
`http://yaroslavvb.blogspot.com/2005/12/trends-in-machine-learning-according.html`

Finding trends in machine learning by search papers in Google Scholar that match a certain keyword:



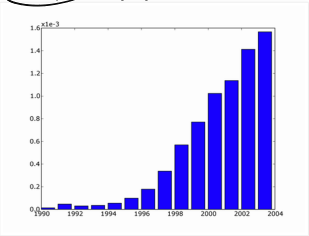
You can see a major upward trend starting around 1985 (that's when Yann LeCun and several others independently rediscovered backpropagation algorithm), peaking in 1992, and going downwards from then.

On other hand, search for "support vector machine" shows no sign of slowing down



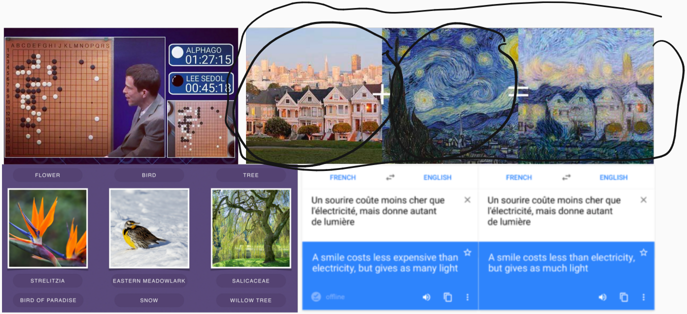
(1995 is when Vapnik and Cortez proposed the algorithm)

Also, "Naive Bayes" seems to be growing without bound



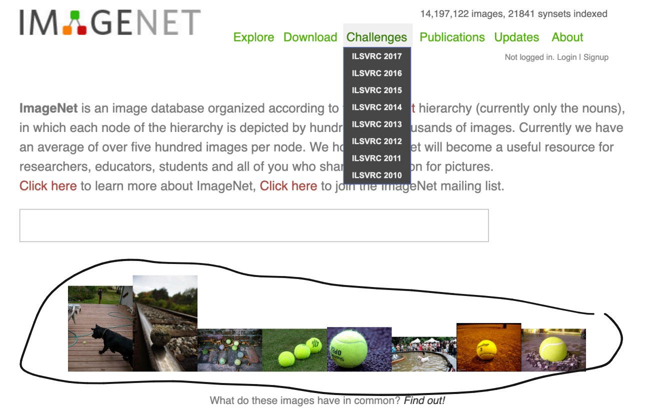
If I were to trust this, I would say that Naive Bayes research the hottest machine learning area right now

In recent years this trend completely turned around:



Recent state-of-the-art results in game playing, image recognition, content generation, natural language processing, machine translation, many other areas.

All changed with the introduction of AlexNet and the 2012 ImageNet Challenge...



Very general image classification task.

All changed with AlexNet and the 2012 ImageNet Challenge...

team name	team members	filename	flat cost	hie cost	description
NEC-UIUC	NEC: Yuanqing Lin, Fengjun Lv, Shenghuo Zhu, Ming Yang, Timothee Cour, Kai Yu UIUC: LiangLiang Cao, Zhen Li, Min-Hsuan Tsai, Xi Zhou, Thomas Huang Rutgers: Tong Zhang	flat_opt.txt	0.28191	2.1144	using sift and lbp feature with two non-linear coding representations and stochastic SVM optimized for top-5 hit rate

2010 Results

Team name	Filename	Error (5 guesses)	Description
SuperVision	test-preds-141-146.2009-131-137-145-146.2011-145f.	0.15315	Using extra training data from ImageNet Fall 2011 release
SuperVision	test-preds-131-137-145-135-145f.txt	0.16422	Using only supplied training data
ISI	pred_FVs_wLACs_weighted.txt	0.26172	Weighted sum of scores from each classifier with SIFT+FV, LBP+FV, GIST+FV, and CSIFT+FV, respectively.

support vector machine

2012 Results

2019 TURING AWARD WINNERS

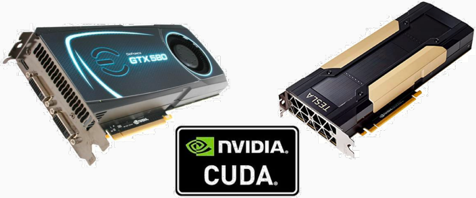
"For conceptual and engineering breakthroughs that have made deep neural networks a critical component of computing."

Yann LeCun   Geoff Hinton   Yoshua Bengio

What were these breakthroughs? What made training large neural networks computationally feasible?

Hardware innovation: Widely available, inexpensive GPUs allowing for cheap, highly parallel linear algebra operations.

- 2007: Nvidia released CUDA platform, which allows GPUs to be easily programmed for general purposed computation.



AlexNet architecture used 60 million parameters. Could not have been trained using CPUs alone (except maybe on a government super computer).

Two main algorithmic tools for training neural network models:

1. Stochastic gradient descent.
2. Backpropogation.

Let $f(\vec{\theta}, \vec{x})$ be our neural network. A typical ℓ-layer feed forward model has the form:

$$g_\ell \left(\mathbf{W}_\ell \left(\ldots \mathbf{W}_3 \cdot g_2 \left(\mathbf{W}_2 \cdot g_1 \left(\mathbf{W}_1 \vec{x} + \vec{b}_1 \right) + \vec{b}_2 \right) + \vec{b}_3 \ldots \right) + b_\ell \right).$$

$\mathbf{W}_i$ and $\vec{b}_i$ are the weight matrix and bias vector for layer i and g_i is the non-linearity (e.g. sigmoid). $\vec{\theta} = [\mathbf{W}_0, \vec{b}_0, \ldots, \mathbf{W}_\ell, \vec{b}_\ell]$ is a vector of all entries in these matrices.

Goal: Given training data $(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n)$ minimize the loss

$$\mathcal{L}(\vec{\theta}) = \sum_{i=1}^{n} L\left(y_i, f(\vec{\theta}, \vec{x}_i) \right)$$

Example: We might use the binary cross-entropy loss for binary classification:

$$L\left(y_i, f(\vec{\theta}, \vec{x}_i) \right) = y_i \log(f(\vec{\theta}, \vec{x}_i)) + (1 - y_i) \log(1 - f(\vec{\theta}, \vec{x}_i))$$

GRADIENT OF THE LOSS

Most common approach: minimize the loss by using gradient descent. Which requires us to compute the gradient of the loss function, $\nabla \mathcal{L}$. Note that this gradient has an entry for every value in $\mathbf{W}_0, \vec{b}_0, \ldots, \mathbf{W}_\ell, \vec{b}_\ell$.

As usual, our loss function has finite sum structure, so:

$$\nabla \mathcal{L}(\vec{\theta}) = \sum_{i=1}^{n} \nabla L\left(y_i, f(\vec{\theta}, \vec{x}_i)\right)$$

$\vec{x}_i, y_i$

cross entropy for example.

So we can focus on computing:

$$\nabla L\left(y, f(\vec{\theta}, \vec{x})\right)$$

$\vec{x}, y$

for a single training example $(\vec{x}, y)$.

Applying chain rule to loss:

$$\nabla_{\vec\theta} L\left(y, f(\vec{\theta}, \vec{x})\right) = \frac{\partial L}{\partial f(\vec{\theta}, \vec{x})} \cdot \nabla_{\vec\theta} f(\vec{\theta}, \vec{x})$$

scalar

Binary cross-entropy example:

$$L\left(y, f(\vec{\theta}, \vec{x})\right) = y \log(f(\vec{\theta}, \vec{x})) + (1-y)\log(1 - f(\vec{\theta}, \vec{x}))$$

$$\left[y \frac{1}{f(\vec\theta, x)} - (1-y)\frac{1}{1 - f(\vec\theta, \vec x)} \right] \cdot \nabla_{\vec\theta} f(\vec\theta, \vec x)$$

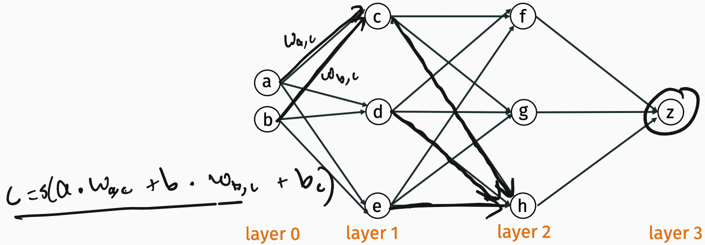
vector

compute by evaluating $f(\vec\theta, \vec x)$

We have reduced our goal to computing $\nabla f(\vec{\theta}, \vec{x})$, where the gradient is with respect to the parameters $\vec{\theta}$.

Back-propagation is a natural and efficient way to compute $\nabla f(\vec{\theta}, \vec{x})$. It derives its name because we compute gradient from back to front: starting with the parameters closest to the output of the neural net.

Let's understand how backprop works with a simple example.

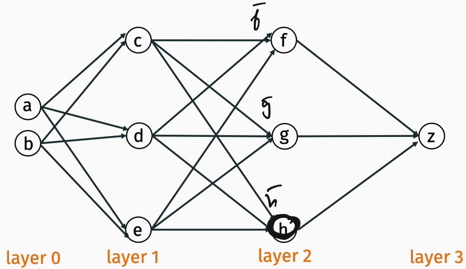


Notation for next few slides:

- $\vec{x} = [a, b]$. $f(\vec{\theta}, \vec{x}) = z$.
- $W_{i,j}$ is the weight of edge from node i to node j.
- $s(\cdot) : \mathbb{R} \rightarrow \mathbb{R}$ is the non-linear activation function.
- b_i is the bias for node i.

Example: $h = s(c \cdot W_{c,h} + d \cdot W_{d,h} + e \cdot W_{e,h} + b_h)$

For any node j, let $\bar{j}$ denote the value obtained <u>before</u> applying the non-linearity [redacted]. s



So if $h = s(c \cdot W_{c,h} + d \cdot W_{d,h} + e \cdot W_{e,h} + b_h)$ then we use $\bar{h}$ to denote:

$$\bar{h} = c \cdot W_{c,h} + d \cdot W_{d,h} + e \cdot W_{e,h} + b_h$$

$$h = s(\bar{h})$$

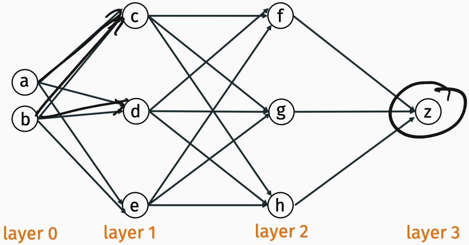
Goal: Compute the gradient $\nabla f(\vec{\theta}, \vec{x})$, which contains the partial derivatives with respect to every parameter:

- $\partial z/\partial b_z$
- $\partial z/\partial W_{f,z}$, $\partial z/\partial W_{g,z}$, $\partial z/\partial W_{h,z}$
- $\partial z/\partial b_f$, $\partial z/\partial b_g$, $\partial z/\partial b_h$
- $\partial z/\partial W_{c,f}$, $\partial z/\partial W_{c,g}$, $\partial z/\partial W_{c,h}$
- $\partial z/\partial W_{d,f}$, $\partial z/\partial W_{d,g}$, $\partial z/\partial W_{d,h}$
- $\vdots$
- $\partial z/\partial W_{a,c}$, $\partial z/\partial W_{a,d}$, $\partial z/\partial W_{a,e}$

z

Two steps: Forward pass to compute function value. Backwards pass to compute gradients.

Step 1: Forward pass.

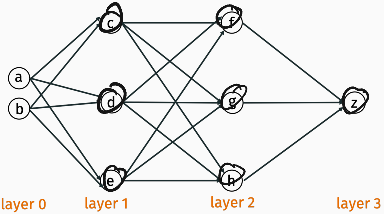


- Using current parameters, compute the output z by moving from left to right.
- Store all intermediate results:

$$\bar{c}, \bar{d}, \bar{e}, c, d, e, \bar{f}, \bar{g}, \bar{h}, f, g, h, \bar{z}, z.$$

BACKPROP EXAMPLE

Step 1: Forward pass.



$$\bar{c} = w_{a,c} \cdot a + w_{b,c} \cdot b + b_c$$

$$\bar{d} = w_{a,d} \cdot a + w_{b,d} \cdot b + b_d$$

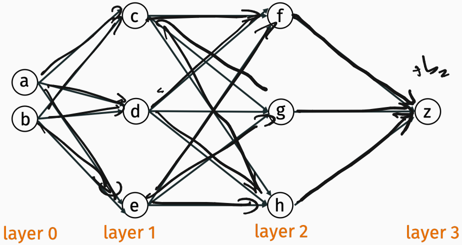
$$\bar{e} = w_{a,e} \cdot a + w_{b,e} \cdot b + b_e$$

$$c = s(\bar{c})$$

$$d = s(\bar{d})$$

$$e = s(\bar{e})$$

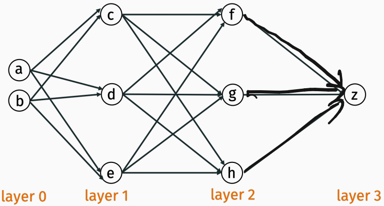
Step 2: Backward pass.



- Using current parameters and computed node values, compute the partial derivatives of all parameters by moving from right to left.

Step 2: Backward pass.

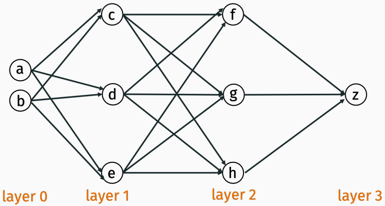
$z = s(\bar{z})$ $\quad \bar{z} = w_{f,z} \cdot f + w_{g,z} \cdot g + w_{h,z} \cdot h + b_z$



$$\frac{\partial z}{\partial w_{f,z}} = \underbrace{\frac{\partial \bar{z}}{\partial w_{f,z}}}_{f} \cdot \frac{\partial z}{\partial \bar{z}} \Big\} s'(z')$$

$$\frac{\partial(z)}{\partial w_{g,z}} = g \cdot s'(\bar{z}) \qquad \frac{\partial(z)}{\partial w_{H,z}} = h \cdot s'(\bar{z})$$

Step 2: Backward pass.



BACKPROP LINEAR ALGEBRA

Linear algebraic view.

Let $\vec{\delta}_i$ be a vector containing $\partial z / \partial j$ for all nodes j in layer i.

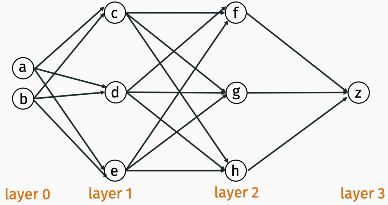
$$\delta_3 = \begin{bmatrix} 1 \end{bmatrix} \qquad \vec{\delta}_2 = \begin{bmatrix} \partial z/\partial f \\ \partial z/\partial g \\ \partial z/\partial h \end{bmatrix} \qquad \vec{\delta}_1 = \begin{bmatrix} \partial z/\partial c \\ \partial z/\partial d \\ \partial z/\partial e \end{bmatrix}$$

Let $\bar{\delta}_i$ be a vector containing $\partial z / \partial \bar{j}$ for all nodes j in layer i.

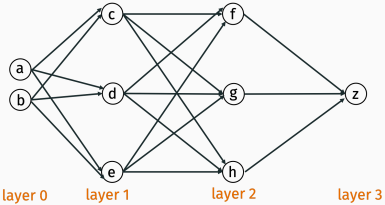
$$\bar{\delta}_3 = \begin{bmatrix} \partial z/\partial \bar{z} \end{bmatrix} \qquad \delta_2 = \begin{bmatrix} \partial z/\partial \bar{f} \\ \partial z/\partial \bar{g} \\ \partial z/\partial \bar{h} \end{bmatrix} \qquad \bar{\delta}_1 = \begin{bmatrix} \partial z/\partial \bar{c} \\ \partial z/\partial \bar{d} \\ \partial z/\partial \bar{e} \end{bmatrix}$$

Note: $\bar{\delta}_i = s'(\vec{\delta}_i) \times \vec{\delta}_i$ where s' is the derivative s' and this function, as well as the $\times$ are applied entrywise.

Let $\mathbf{W}_i$ be a matrix containing all the weights for edges between layer i and layer $i+1$.



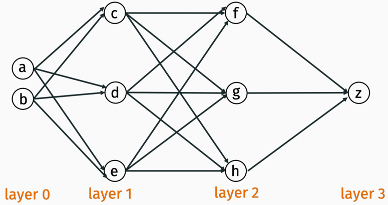
$$\mathbf{W}_2 = \begin{bmatrix} W_{f,z} & W_{g,z} & W_{h,z} \end{bmatrix} \quad \mathbf{W}_1 = \begin{bmatrix} W_{c,f} & W_{d,f} & W_{e,f} \\ W_{c,g} & W_{d,g} & W_{e,g} \\ W_{c,h} & W_{d,h} & W_{e,h} \end{bmatrix} \quad \mathbf{W}_0 = \begin{bmatrix} W_{a,c} & W_{b,c} \\ W_{a,d} & W_{b,d} \\ W_{a,e} & W_{b,e} \end{bmatrix}$$



Claim 1: Node derivative computation is matrix multiplication.

$$\vec{\delta}_i = W_i^T \bar{\delta}_{i+1}$$

Let $\mathbf{\Delta}_i$ be a matrix contain the derivatives for all weights for edges between layer i and layer $i+1$.

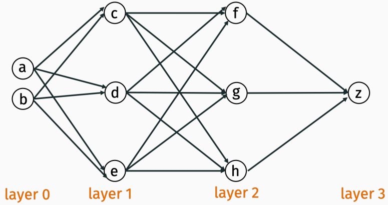


$$\mathbf{\Delta}_2 = \begin{bmatrix} \partial z/\partial W_{f,z} & \partial z/\partial W_{g,z} & \partial z/\partial W_{h,z} \end{bmatrix}$$

$$\mathbf{\Delta}_1 = \begin{bmatrix} \partial z/\partial W_{c,f} & \partial z/\partial W_{d,f} & \partial z/\partial W_{e,f} \\ \partial z/\partial W_{c,g} & \partial z/\partial W_{d,g} & \partial z/\partial W_{e,g} \\ \partial z/\partial W_{c,h} & \partial z/\partial W_{d,h} & \partial z/\partial W_{e,h} \end{bmatrix}$$

$$\mathbf{\Delta}_0 = \ldots$$

BACKPROP LINEAR ALGEBRA



Claim 2: Weight derivative computation is an outer-product.

$$\boldsymbol{\Delta}_i = \vec{v}_i \vec{\delta}_{i+1}^T$$

where $\vec{v}_i$ contains the values of all nodes in layer i. E.g. $\vec{v}_0 = \begin{bmatrix} a \\ b \end{bmatrix}$.

Takeaways:

- Backpropogation can be used to compute derivatives for all weights and biases for any feedforward neural network.
- Final computation boils down to linear algebra operations (matrix multiplication and vector operations) which can be performed quickly on a GPU.

We computed $\nabla L\left(y_i, f(\vec{\theta}, \vec{x}_i)\right)$ for a single training example $(\vec{x}_i, y_i)$. Computing entire gradient requires computing:

$$\nabla \mathcal{L}(\vec{\theta}) = \sum_{i=1}^{n} \nabla L\left(y_i, f(\vec{\theta}, \vec{x}_i)\right)$$

Computing the entire sum would be very expensive.

Second tool: Stochastic Gradient Descent (SGD).

- Powerful randomized variant of GD used to train neural networks (or any other model where computing gradients is expensive.

Recall gradient descent update:

- For $t = 1, \ldots, T$:
  - $\vec{\theta}_{t+1} = \vec{\theta}_t - \eta \nabla \mathcal{L}(\vec{\theta}_t)$

where η is a learning rate parameter.

Let $L_j(\vec{\theta})$ denote $L\left(y_j, f(\vec{\theta}, \vec{x}_j)\right)$.

Claim: If $j \in 1, \ldots, n$ is chosen uniformly at random. Then:

$$n \cdot \mathbb{E}\left[\nabla L_j(\vec{\theta})\right] = \nabla \mathcal{L}(\vec{\theta}).$$

$\nabla L_j(\vec{\theta})$ is called a stochastic gradient.

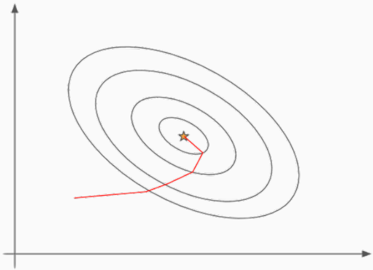
SGD iteration:

- Initialize $\vec{\theta}_0$ (typically randomly).
- For $t = 1, \ldots, T$:
    - Choose j uniformly at random.
    - Compute stochastic gradient $\vec{g} = \nabla L_j(\vec{\theta}_t)$.
        - For neural networks this is done using backprop with training example $(\vec{x}_j, y_j)$.
    - Update $\vec{\theta}_{t+1} = \vec{\theta}_t - \eta \vec{g}$

Move in direction of steepest descent in expectation.

Gradient descent: Fewer iterations to converge, higher cost per iteration.

Stochastic Gradient descent: More iterations to converge, lower cost per iteration.

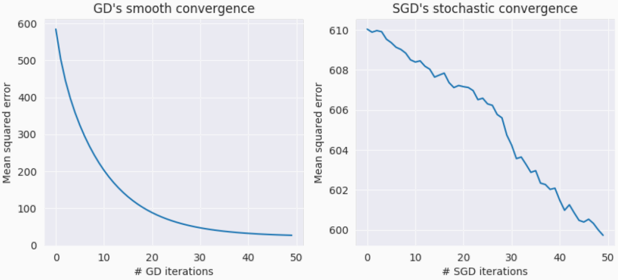


Gradient Descent

Stochastic Gradient Descent

Gradient descent: Fewer iterations to converge, higher cost per iteration.

Stochastic Gradient descent: More iterations to converge, lower cost per iteration.



Practical Modification 1: **Cyclic Gradient Descent.**

Assume order of data is relatively random. Instead of choosing j randomly at each iteration, choose $j = 1, j = 2, \ldots, j = n, j = 1, \ldots, j = n, \ldots$.

Question: Why might we want to do this?

- Relatively similar convergence behavior to standard SGD.
- **Import term:** one epoch denotes one pass over all training examples: $j = 1, \ldots, j = n$.
- Convergence rates for training neural networks are often discussed in terms of epochs instead of iterations.

Practical Modification 2: **Mini-batch Gradient Descent.**

Observe that for any batch size s,

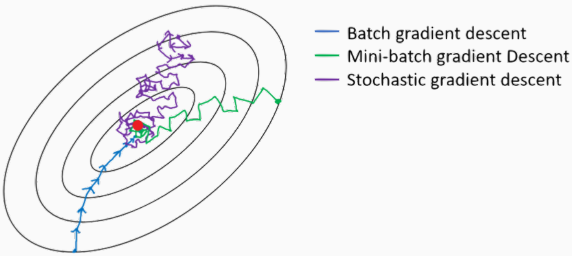
$$n \cdot \mathbb{E}\left[\frac{1}{s}\sum_{i=1}^{s} \nabla L_{j_i}(\vec{\theta})\right] = \nabla \mathcal{L}(\vec{\theta}).$$

if $j_1, \ldots, j_s$ are chosen independently and uniformly at random from $1, \ldots, n$.

Instead of computing a full stochastic gradient, compute the average gradient of a small random set (a mini-batch) of training data examples.

Question: Why might we want to do this?

MINI-BATCH GRADIENT DESCENT



- For small batch size s, mini-batch gradients are nearly as fast to compute as stochastic gradients (due to parallelism).
- Overall faster convergence (fewer iterations needed).