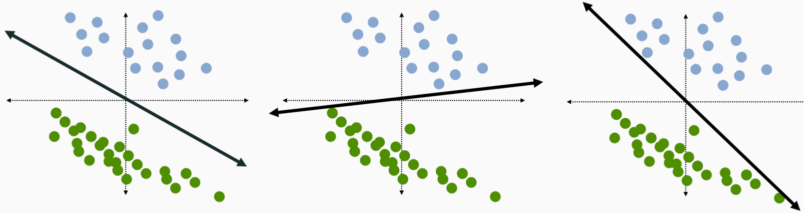
CS-GY 6923: Lecture 10
Finish SVMs, Neural Nets Introduction, Back propagation

NYU Tandon School of Engineering, Prof. Christopher Musco

SUPPORT VECTOR MACHINES

Goal: Find a separating hyperplane for linearly separable classification problem.



Ideally, choose the hyperplane that maximizes margin.

OPTIMIZATION FORMULATION

Original problem: $\arg\max_{\boldsymbol{\beta}} \left[\min_{i\in 1,\ldots,n} \frac{y_i \cdot \langle \mathbf{x}_i, \boldsymbol{\beta}\rangle}{\|\boldsymbol{\beta}\|_2}\right]$.

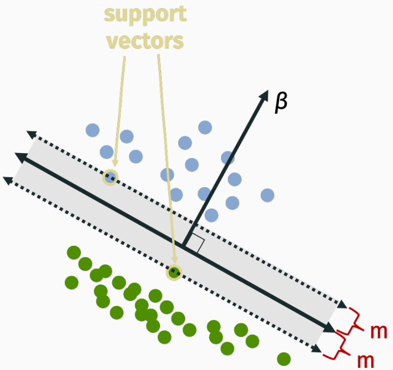
Equivalent formulation:

$$\min_{\boldsymbol{\beta}} \|\boldsymbol{\beta}\|_2^2 \qquad \text{subject to} \qquad y_i \cdot \langle \mathbf{x}_i, \boldsymbol{\beta}\rangle \geq 1 \text{ for all } i.$$

Under this formulation $m = \frac{1}{\|\boldsymbol{\beta}\|_2}$.

- Can be solved using a constrained optimization method.
- Can be combined with any non-linear kernel.
- Classification only requires computing kernel similarity with the support vectors.

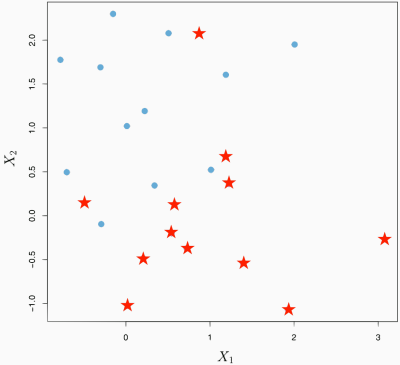
CLASSIFICATION



- When using a kernel like $k(\mathbf{x}_i, \mathbf{x}_j) = e^{-\|\mathbf{x}_i - \mathbf{x}_j\|_2^2}$, classification for a new points $\mathbf{x}_{new}$ only requires computing kernel similarity with the support vectors. Logistic regression requires similarity with all training points.

HARD-MARGIN SVM

Hard-margin SVMs have a few other critical issues in practice:

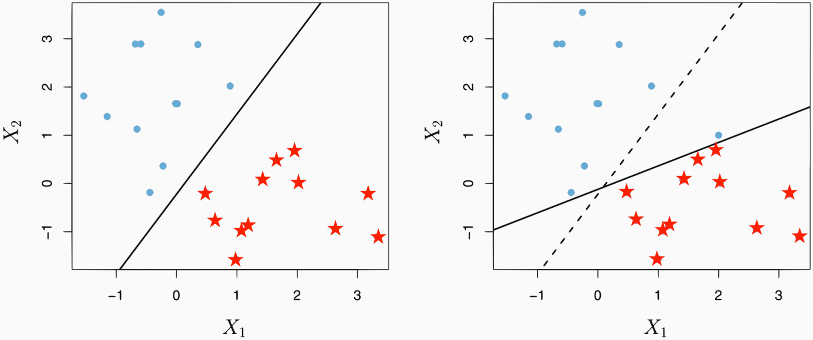


Data might not be linearly separable, in-which case the maximum margin classifier is not even defined.

Less likely to be an issue when using a non-linear kernel. If $\mathbf{K}$ is full rank then perfect separation is always possible. And typically it is, e.g. for an RBF kernel or moderate degree polynomial kernel.

HARD-MARGIN SVM

Another critical issue in practice:



Hard-margin SVM classifiers are not robust.

SOFT-MARGIN SVM

Solution: Allow the classifier to make some "mistakes"! A mistake can either be a misclassification, or simply a point allowed to be "inside" the margin.

Hard margin objective:

$$\min_{\boldsymbol{\beta}} \|\boldsymbol{\beta}\|_2^2 \qquad \text{subject to} \qquad y_i \cdot \langle \mathbf{x}_i, \boldsymbol{\beta} \rangle \geq 1 \text{ for all } i.$$

Soft margin objective:

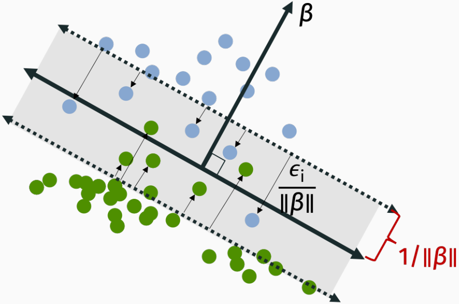
$$\min_{\boldsymbol{\beta}} \|\boldsymbol{\beta}\|_2^2 + C \sum_{i=1}^{n} \epsilon_i \quad \text{subject to} \quad y_i \cdot \langle \mathbf{x}_i, \boldsymbol{\beta} \rangle \geq 1 - \epsilon_i \text{ for all } i.$$

where $\epsilon_i \geq 0$ is a non-negative "slack variable".

$\epsilon_i / \|\boldsymbol{\beta}\|_2$ is is the magnitude of the "error" (distance past the margin) we allow $\mathbf{x}_i$ to travel. Recalling that margin is $1/\|\boldsymbol{\beta}\|_2$, $\epsilon_i \geq 1$ corresponds to a misclassification.

SOFT-MARGIN SVM

Recall that $\Delta_i = \frac{y_i \cdot \langle \mathbf{x}_i, \boldsymbol{\beta} \rangle}{\|\boldsymbol{\beta}\|_2}$.

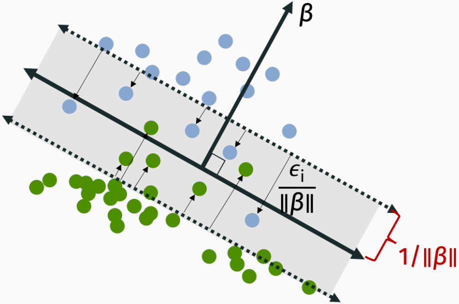


Soft margin objective:

$$\min_{\boldsymbol{\beta}} \|\boldsymbol{\beta}\|_2^2 + C \sum_{i=1}^{n} \epsilon_i \quad \text{subject to} \quad y_i \cdot \langle \mathbf{x}_i, \boldsymbol{\beta} \rangle \geq 1 - \epsilon_i \text{ for all } i.$$

SOFT-MARGIN SVM

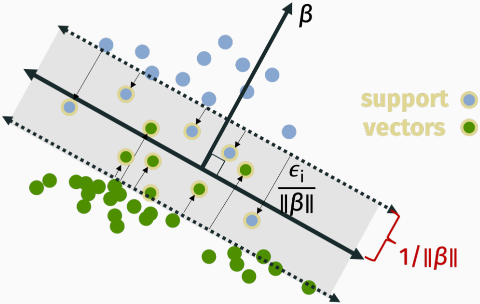
Recall that $\Delta_i = \frac{y_i \cdot \langle \mathbf{x}_i, \boldsymbol{\beta} \rangle}{\|\boldsymbol{\beta}\|_2}$.



Soft margin objective:

$$\min_{\boldsymbol{\beta}} \|\boldsymbol{\beta}\|_2^2 + C \sum_{i=1}^{n} \epsilon_i \quad \text{subject to} \quad \frac{y_i \cdot \langle \mathbf{x}_i, \boldsymbol{\beta} \rangle}{\|\boldsymbol{\beta}\|_2} \geq \frac{1}{\|\boldsymbol{\beta}\|_2} - \frac{\epsilon_i}{\|\boldsymbol{\beta}\|_2} \text{ for all } i.$$

SOFT-MARGIN SVM



Any $\mathbf{x}_i$ with a non-zero ϵ_i is a <u>support vector</u>. As before, only support vectors are needed for classification in the kernel setting. Good exercise to prove yourself.

EFFECT OF C

Soft margin objective:

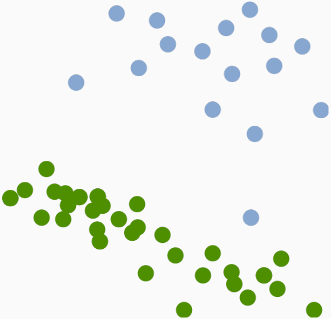
$$\min_{\boldsymbol{\beta}} \|\boldsymbol{\beta}\|_2^2 + C\sum_{i=1}^{n} \epsilon_i.$$

- Large C means penalties are punished more in objective $\implies$ smaller margin, less support vectors.
- Small C means penalties are punished less in objective $\implies$ larger margin, more support vectors.

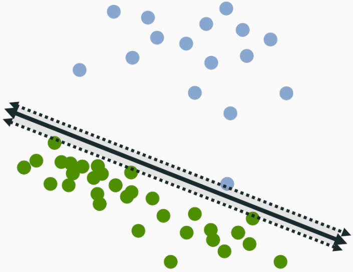
When data is linearly separable, as $C \to \infty$ we will always get a separating hyperplane. A smaller value of C might lead to a more robust solution.

EFFECT OF C

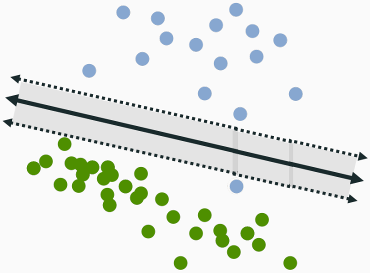
Example dataset:



EFFECT OF C



large C



smaller C

The classifier on the right is intuitively more robust. So for this data, a smaller choice for C might make sense.

Typically the smaller C is, the more support vectors (above image isn't a great example).

COMPARISON TO LOGISTIC REGRESSION

Some basic transformations of the soft-margin objective:

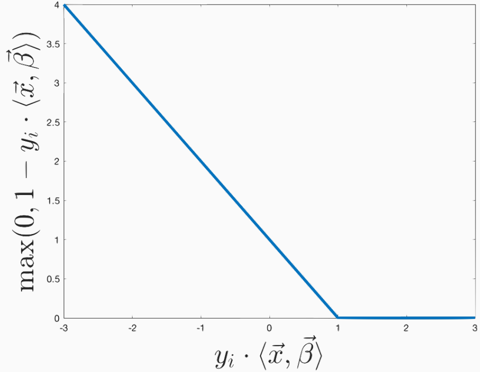
$$\min_{\boldsymbol{\beta}} \|\boldsymbol{\beta}\|_2^2 + C\sum_{i=1}^{n} \epsilon_i \quad \text{subject to} \quad y_i \cdot \langle \mathbf{x}_i, \boldsymbol{\beta}\rangle \geq 1 - \epsilon_i \text{ for all } i.$$

$$\min_{\boldsymbol{\beta}} \|\boldsymbol{\beta}\|_2^2 + C\sum_{i=1}^{n} \max(0, 1 - y_i \cdot \langle \mathbf{x}_i, \boldsymbol{\beta}\rangle).$$

$$\min_{\boldsymbol{\beta}} \lambda\|\boldsymbol{\beta}\|_2^2 + \sum_{i=1}^{n} \max(0, 1 - y_i \cdot \langle \mathbf{x}_i, \boldsymbol{\beta}\rangle).$$

These are all equivalent. $\lambda = 1/C$ is just another scaling parameter. Moved from a constrained problem to a much easier unconstrained optimization problem.

Hinge-loss: $\max(0, 1 - y_i \cdot \langle \mathbf{x}_i, \boldsymbol{\beta} \rangle)$. Recall that $y_i \in \{-1, 1\}$.



Soft-margin SVM:

$$\min_{\boldsymbol{\beta}} \left[\sum_{i=1}^{n} \max(0, 1 - y_i \cdot \langle \mathbf{x}_i, \boldsymbol{\beta} \rangle) + \lambda \|\boldsymbol{\beta}\|_2^2 \right]. \tag{1}$$

LOGISTIC LOSS

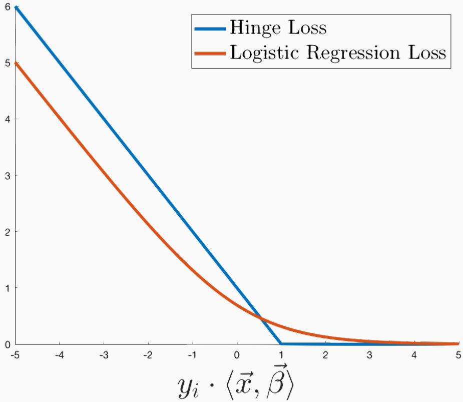
Recall the logistic loss for $y_i \in \{0, 1\}$:

$$\begin{aligned} L(\boldsymbol{\beta}) &= -\sum_{i=1}^{n} y_i \log(h(\langle \mathbf{x}_i, \boldsymbol{\beta} \rangle)) + (1 - y_i) \log(1 - h(\langle \mathbf{x}_i, \boldsymbol{\beta} \rangle)) \\ &= -\sum_{i=1}^{n} y_i \log\left(\frac{1}{1 + e^{-\langle \mathbf{x}_i, \boldsymbol{\beta} \rangle}}\right) + (1 - y_i) \log\left(\frac{e^{-\langle \mathbf{x}_i, \boldsymbol{\beta} \rangle}}{1 + e^{-\langle \mathbf{x}_i, \boldsymbol{\beta} \rangle}}\right) \\ &= -\sum_{i=1}^{n} y_i \log\left(\frac{1}{1 + e^{-\langle \mathbf{x}_i, \boldsymbol{\beta} \rangle}}\right) + (1 - y_i) \log\left(\frac{1}{1 + e^{\langle \mathbf{x}_i, \boldsymbol{\beta} \rangle}}\right) \end{aligned}$$

comparison of SVM to logistic regression

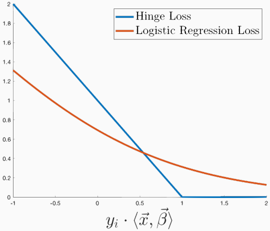
Compare this to the logistic regression loss reformulated for $y_i \in \{-1, 1\}$):

$$\sum_{i=1}^{n} -\log\left(\frac{1}{1 - e^{-y_i \cdot \langle \mathbf{x}_i, \boldsymbol{\beta} \rangle}}\right)$$



COMPARISON TO LOGISTIC REGRESSION

So, in the end, the function minimized when finding $\boldsymbol{\beta}$ for the standard **soft-margin SVM** is very similar to the objective function minimized when finding $\boldsymbol{\beta}$ using **logistic regression with ℓ_2 regularization.**



Both functions can be optimized using first-order methods like gradient descent. This is now a common choice for large problems. Will explore more on Lab 5.

NEURAL NETWORKS

NEURAL NETWORKS

Key Concept

Approach until now:

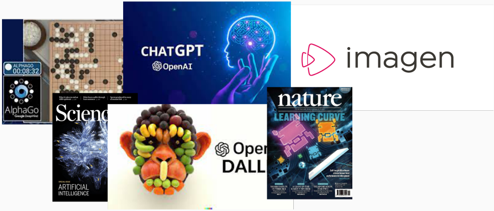
- Choose good features or a good kernel.
- Use optimization to find best model given those features.

Neural network approach:

- Learn good features and a good model simultaneously.

NEURAL NETWORKS

The leading method in machine learning right now.

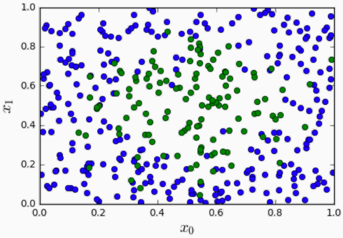


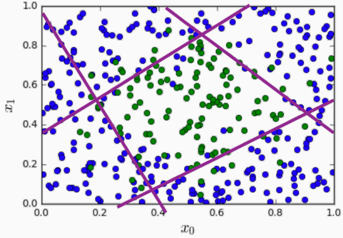
Focus of investment at universities, government research labs, funding agencies, and large tech companies.

Studied since the 1940s/50s. **Why the recent attention?** More on history of neural networks shortly.

SIMPLE MOTIVATING EXAMPLE

Classification when data is not linearly separable:



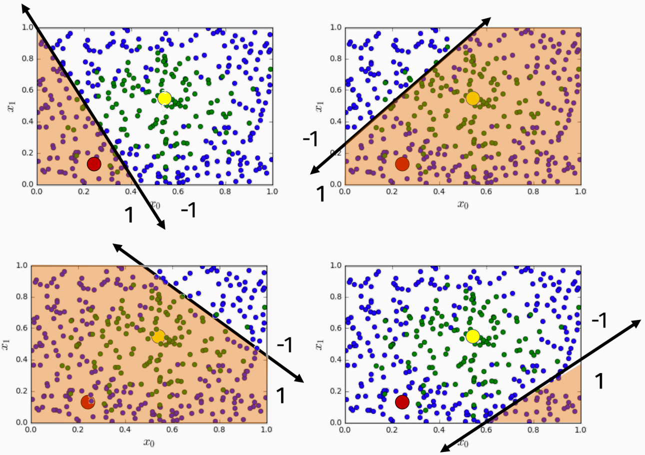


Could use feature transformations or a non-linear kernel.

Alternative approach: Divide the space up into regions using multiple linear classifiers.

SIMPLE MOTIVATING EXAMPLE

For each linear classifier β, add a new $-1, 1$ feature for every example $\mathbf{x} = [x_0, x_1]$ depending on the sign of $\langle \mathbf{x}, \beta \rangle$.



🔴 $\Longrightarrow$ 🟡 $\Longrightarrow$

SIMPLE MOTIVATING EXAMPLE

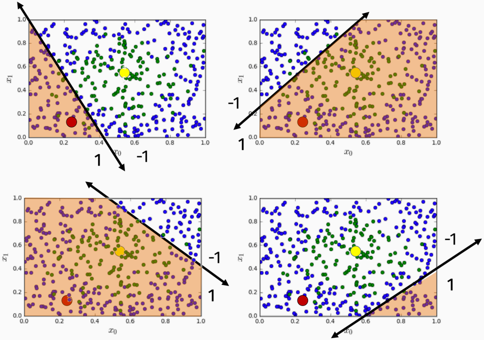
$$\begin{bmatrix} .2, .8, \\ .5, .5 \\ \vdots \\ .5, 1 \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_n \end{bmatrix} \implies \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \vdots \\ \mathbf{u}_n \end{bmatrix} = \begin{bmatrix} -1, -1, +1, -1 \\ -1, +1, +1, -1 \\ \vdots \\ -1, -1, -1, -1 \end{bmatrix}$$

Question: After data transformation, how should we map each new vector $\mathbf{u}_i$ to a class label?

$$\begin{bmatrix} -1, -1, +1, -1 \\ -1, +1, +1, -1 \\ \vdots \\ -1, -1, -1, -1 \end{bmatrix} \xrightarrow{?} \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

Our machine learning algorithms needs to **learn two things**:

- The original linear functions which divide our data set into regions (their slopes + intercepts).



- Another linear function which maps our new features to an output class probability.

POSSIBLE MODEL

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

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

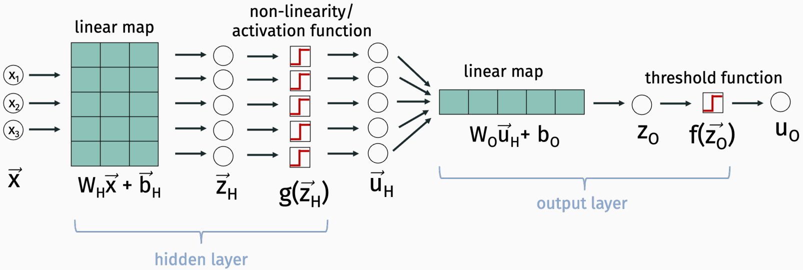
- $\mathrm{z}_H \in \mathbb{R}^{N_H} = \mathrm{W}_H \mathrm{x} + \boldsymbol{\beta}_h$.
- $\mathrm{u}_H = \mathrm{sign}(\mathrm{z}_H)$
- $z_O \in \mathbb{R} = \mathrm{W}_O \mathrm{u}_H + \boldsymbol{\beta}_O$
- $u_O = \mathbb{1}[z_O > \lambda]$

Parameters: $\Theta = [\mathrm{W}_H \in \mathbb{R}^{N_H \times N_I}, \boldsymbol{\beta}_H \in \mathbb{R}^{N_H}, \mathrm{W}_O \in \mathbb{R}^{1 \times N_H}, \beta_O \in \mathbb{R}]$.

W_H, W_O are weight matrices and $\boldsymbol{\beta}_H$, $\boldsymbol{\beta}_O$ are bias terms that account for the intercepts of our linear functions.

POSSIBLE MODEL

Our model is function f which makes $\mathbf{x}$ to a class label u_O.[1]



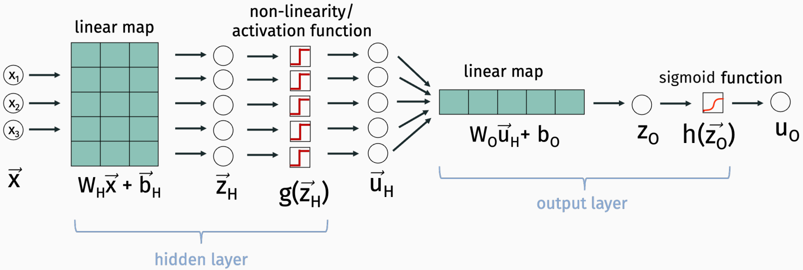
This is called a "multilayer perceptron": one of the oldest types of neural nets. Dates back to Frank Rosenblatt from 1958

- Number of input variables $N_I =$
- Number of hidden variables $N_H =$
- Number of output variables $N_O =$

[1] For regression, would cut off at z_O to get continuous output.

POSSIBLE MODEL

Our model is function f which maps $\mathbf{x}$ to a class label u_O.

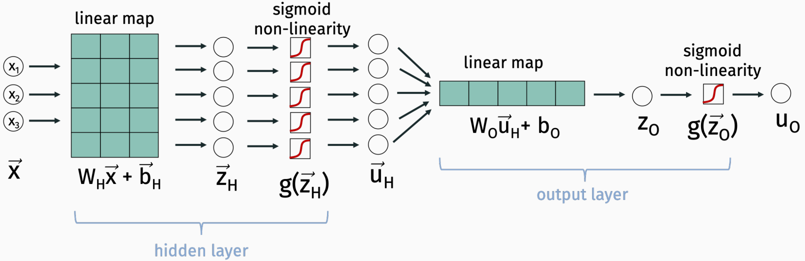


Training the model:

- Choose a loss function $L(f(\mathbf{x}, \Theta), y)$.
- Find optimal parameters: $\Theta^* = \arg\min_\Theta \sum_{i=1}^{n} L(f(\mathbf{x}_i, \Theta), y_i)$ using gradient descent.

FINAL MODEL

A more typical model uses smoother activation functions, aka non-linearities, which are more amenable to computing gradients. E.g. we might use the **sigmoid function** $g(x) = \frac{1}{1+e^{-x}}$.



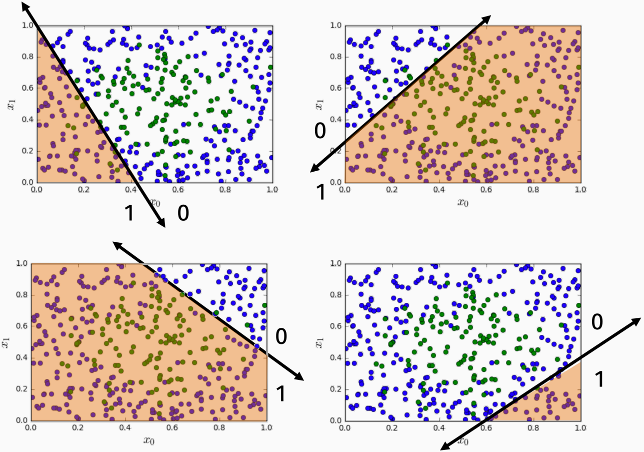
- Tune parameters by minimizing cross-entropy loss:

$$\sum_{i=1}^{n} L(f(\mathrm{x}_i, \Theta), y_i) = \sum_{i=1}^{n} -y_i \log(f(\mathrm{x}_i, \Theta)) - (1 - y_i) \log(1 - f(\mathrm{x}_i, \Theta))$$

- We will discuss soon how to compute gradients.

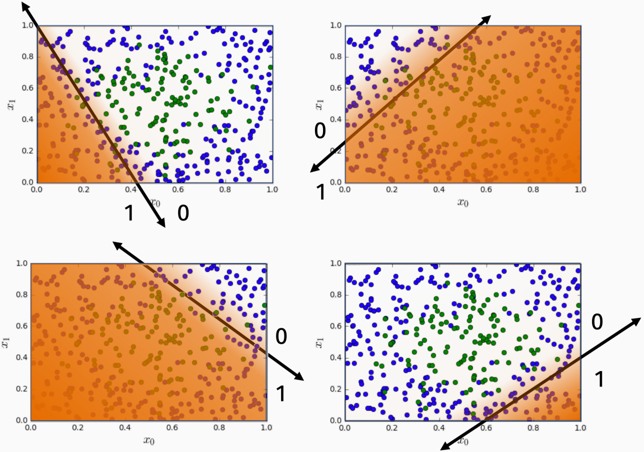
FEATURE EXTRACTION

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



FEATURE EXTRACTION

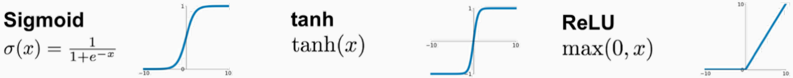
Features learned using sigmoid activation are real valued in $[0, 1]$. Mimic binary features.



HYPERPARAMETERS

Things we can change in this basic classification network:

- More or less hidden variables.
- We could add more layers.
- Different non-linearity/activation function.
- Different loss function.



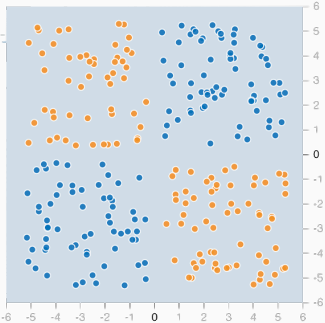
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$

tanh
$\tanh(x)$

ReLU
$\max(0, x)$

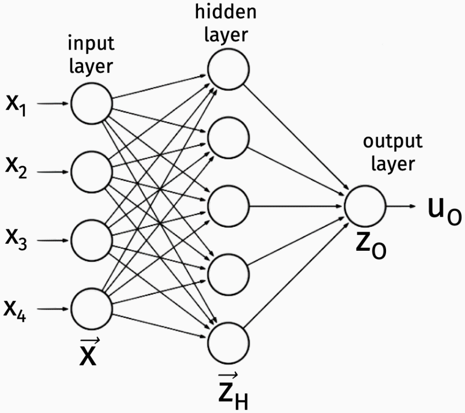
TEST YOUR INTUITION

How many hidden variables (e.g. splitting hyperplanes) would be needed to classify this dataset correctly?



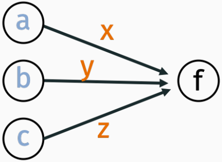
`https://playground.tensorflow.org/`

Another common diagram for a 2-layered network:



NOTATION

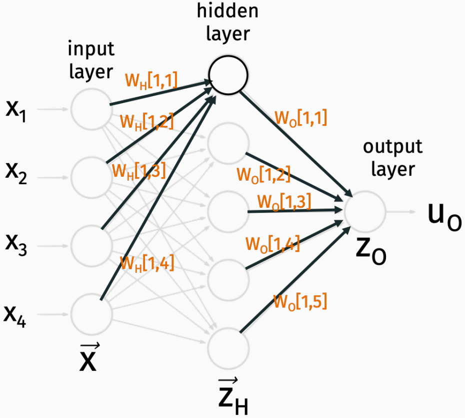
Neural network math:



$f = ax + by + cz$

NOTATION

How to interpret:

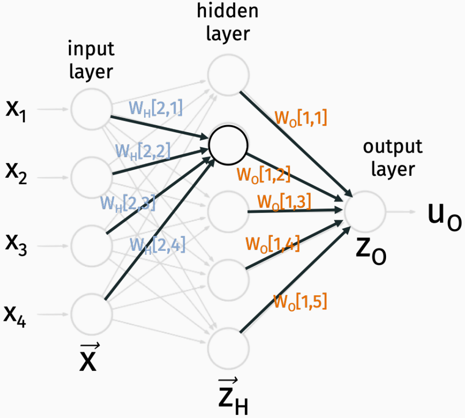


W_H and W_O are our weight matrices from before.

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

NOTATION

How to interpret:

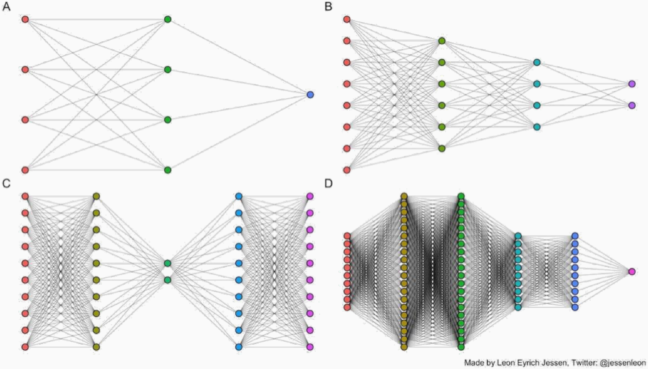


W_H and W_O are our weight matrices from before.

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

ARCHITECTURE VISUALIZATION

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



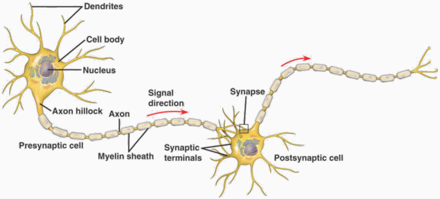
Made by Leon Eyrich Jessen, Twitter: @jessenleon

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

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

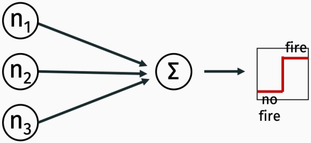
SOME HISTORY AND MOTIVATION

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.



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

CONNECTION TO BIOLOGY

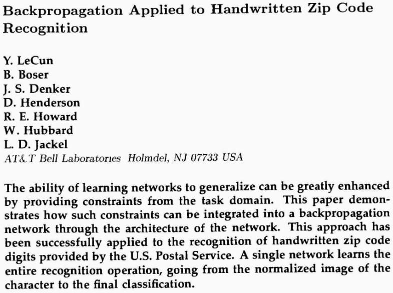
Inspired early work on neural networks:

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

Main issue with neural network methods: They are hard to train. Gradient descent converges very slowly. Also pretty finicky: user needs to be careful with initialization, regularization, etc. when training. We have gotten a lot better at resolving these issues though!

EARLY NEURAL NETWORK EXPLOSION

Around 1985 several groups (re)-discovered the **backpropagation algorithm** which allows for efficient training of neural nets via **(stochastic) 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.

NEURAL NETWORK DECLINE

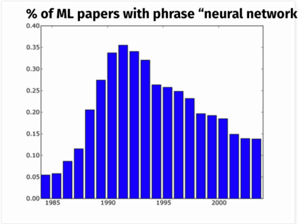
From 1990s - 2010, kernel methods, SVMs, and probabilistic methods began to dominate the literature in machine learning:

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

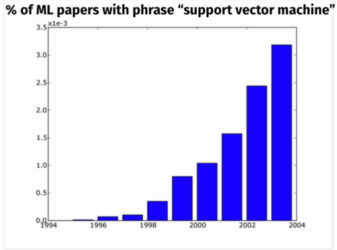
Fun blog post to check out from 2005: `http://yaroslavvb.blogspot.com/2005/12/trends-in-machine-learning-according.html`

NEURAL NETWORK DECLINE

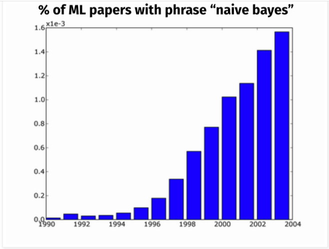
Finding trends in machine learning by search papers in Google Scholar that match a certain keyword:


% of ML papers with phrase “neural network”

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.


% of ML papers with phrase “support vector machine”

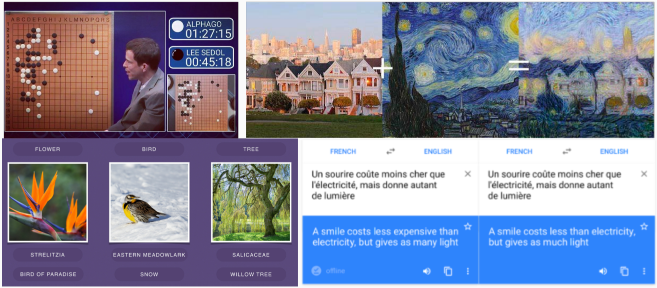
(1995 is when Vapnik and Cortez proposed the algorithm)


% of ML papers with phrase “naive bayes”

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

MODERN NEURAL NETWORK RESURGENCE

In recent years this trend completely turned around:



Un sourire coûte moins cher que l'électricité, mais donne autant de lumière

A smile costs less expensive than electricity, but gives as many light

Un sourire coûte moins cher que l'électricité, mais donne autant de lumière

A smile costs less than electricity, but gives as much light

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

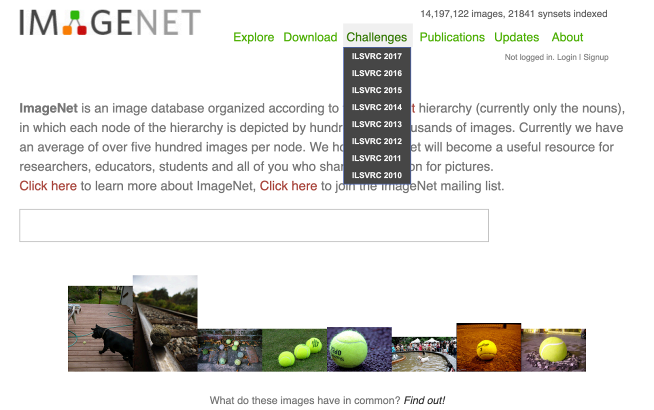
2019 TURING AWARD WINNERS

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

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

ALEXNET

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



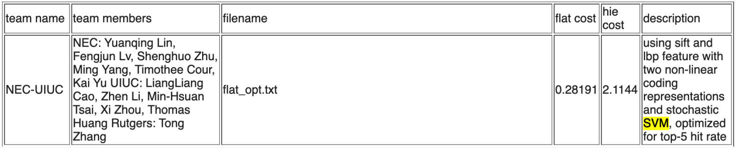
ImageNet is an image database organized according to t hierarchy (currently only the nouns), in which each node of the hierarchy is depicted by hundr usands of images. Currently we have an average of over five hundred images per node. We ho et will become a useful resource for researchers, educators, students and all of you who shar on for pictures.
Click here to learn more about ImageNet, Click here to jo ageNet mailing list.

What do these images have in common? *Find out!*

Very general image classification task.

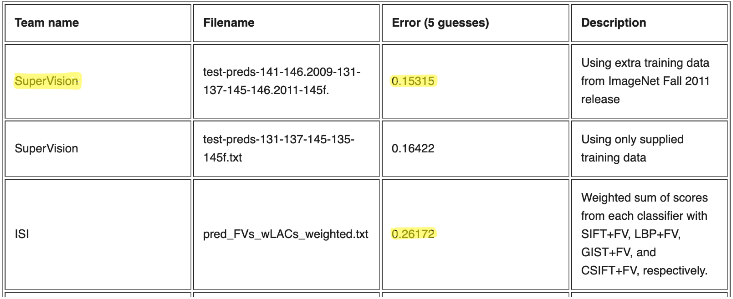
ALEXNET

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

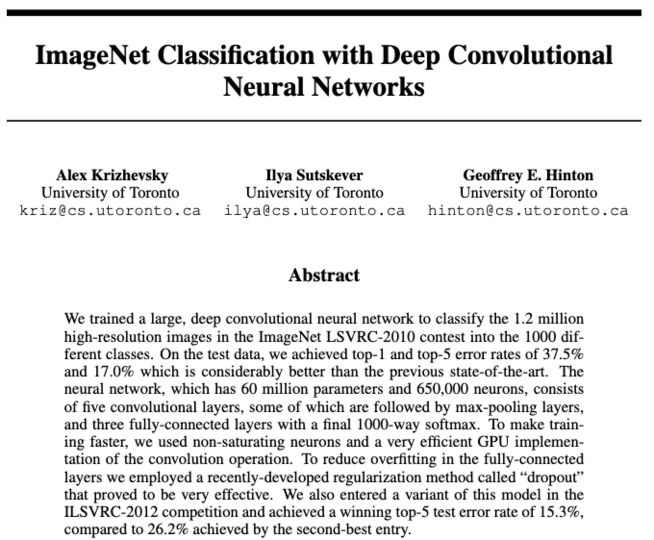
2010 Results



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.

2012 Results

ALEXNET



ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky
University of Toronto
kriz@cs.utoronto.ca

Ilya Sutskever
University of Toronto
ilya@cs.utoronto.ca

Geoffrey E. Hinton
University of Toronto
hinton@cs.utoronto.ca

Abstract

We trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes. On the test data, we achieved top-1 and top-5 error rates of 37.5% and 17.0% which is considerably better than the previous state-of-the-art. The neural network, which has 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax. To make training faster, we used non-saturating neurons and a very efficient GPU implementation of the convolution operation. To reduce overfitting in the fully-connected layers we employed a recently-developed regularization method called "dropout" that proved to be very effective. We also entered a variant of this model in the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%, compared to 26.2% achieved by the second-best entry.

MODERN NEURAL NETWORKS

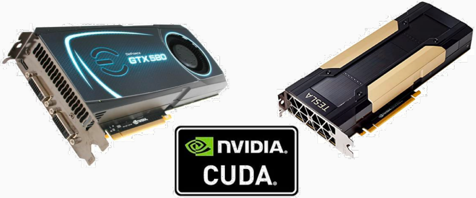
Why 2012?

- Clever ideas in changing neural network architecture and training. E.g. ReLU non-linearities, dropout regularization, batch normalization, data augmentation.
- Wide-spread access to GPU computing power.

GRAPHICS PROCESSING UNIT

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).

TRAINING NEURAL NETWORKS

Two main algorithmic tools for training neural network models:

1. Stochastic gradient descent.
2. Backpropogation.

TRAINING NEURAL NETWORKS

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

$$g_\ell\left(\mathrm{W}_\ell\left(\ldots \mathrm{W}_3 \cdot g_2\left(\mathrm{W}_2 \cdot g_1\left(\mathrm{W}_1\mathrm{x} + \boldsymbol{\beta}_1\right) + \boldsymbol{\beta}_2\right) + \boldsymbol{\beta}_3 \ldots\right) + \beta_\ell\right).$$

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

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

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

where L is, e.g., binary cross-entropy (logistic) loss:

$$L\left(y_i, f(\boldsymbol{\theta}, \mathrm{x}_i)\right) = -y_i \log(f(\boldsymbol{\theta}, \mathrm{x}_i)) - (1 - y_i)\log(1 - f(\boldsymbol{\theta}, \mathrm{x}_i)).$$

GRADIENT OF THE LOSS

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 $\mathbf{W}_0, \boldsymbol{\beta}_0, \ldots, \mathbf{W}_\ell, \boldsymbol{\beta}_\ell$.

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

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

So we can focus on computing:

$$\nabla L\left(y_i, f(\boldsymbol{\theta}, \mathbf{x}_i)\right)$$

for a single training example $(\mathbf{x}_i, y_i)$.

CHAIN RULE REVIEW

For a scalar function $f(x)$, we write the derivative with respect to x as:

$$f'(x) = \frac{df}{dx} = \lim_{t \to 0} \frac{f(x+t) - f(x)}{t}$$

For a multivariate function $f(x, y, z)$ wr write the partial derivative with repect to x as:

$$\frac{df}{dx} = \lim_{t \to 0} \frac{f(x+t, y, z) - f(x, y, z)}{t}$$

CHAIN RULE REVIEW

Let $y(x)$ be a function of x and let $f(y)$ be a function of y. The chain rule says that:

$$\frac{df}{dx} = \frac{df}{dy}\frac{dy}{dx}$$

$$\begin{aligned}\frac{df}{dx} &= \lim_{t\to 0} \frac{f(y(x+t)) - f(y(x))}{t} \\ &= \lim_{t\to 0} \frac{f(y(x+t)) - f(y(x))}{y(x+t) - y(x)} \cdot \frac{y(x+t) - y(x)}{t} \\ &= \lim_{t\to 0} \frac{f(y(x)+c) - f(y(x))}{c} \cdot \frac{y(x+t) - y(x)}{t}\end{aligned}$$

where $c = y(x+t) - y(x)$.

As long as $\lim_{t\to 0} y(x+t) - y(x) = 0$ then the first term equals $\frac{df}{dy}$. The second term equals $\frac{dy}{dx}$.

MULTIVARIABLE CHAIN RULE

Let $y(x), z(x), w(x)$ be functions of x and let $f(y, z, w)$ be a function of y, z, w.

$$\frac{df}{dx} = \frac{df}{dy} \cdot \frac{dy}{dx} + \frac{df}{dz} \cdot \frac{dz}{dx} + \frac{df}{dw} \cdot \frac{dw}{dx}$$

Example: Let $y(x) = x^3$ and $z(x) = x^2$. Let $f(y, z) = y \cdot z$. Then:

$$\begin{aligned} \frac{df}{dx} &= \left(\frac{df}{dy} \cdot \frac{dy}{dx}\right) + \left(\frac{df}{dz} \cdot \frac{dz}{dx}\right) \\ &= \end{aligned}$$

GRADIENT OF THE LOSS

Applying chain rule each partial derivative of the loss:

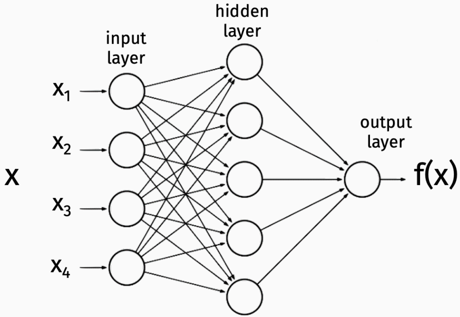
$$\nabla L\left(y, f(\boldsymbol{\theta}, \mathrm{x})\right) = \frac{\partial L}{\partial f(\boldsymbol{\theta}, \mathrm{x})} \cdot \nabla f(\boldsymbol{\theta}, \mathrm{x})$$

Binary cross-entropy example:

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

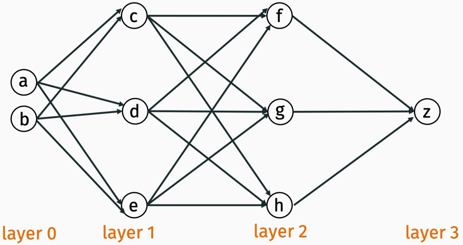
GRADIENT OF THE LOSS

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



Backpropagation is an efficient way to compute $\nabla f(\boldsymbol{\theta}, \mathrm{x})$. It derives its name because we compute gradient from back to front: starting with the parameters closest to the output of the neural net.

BACKPROP EXAMPLE



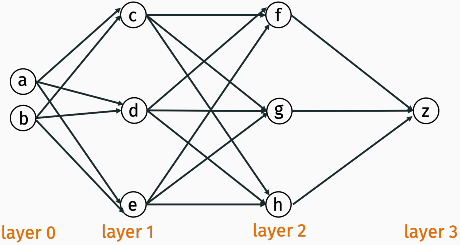
Notation for few slides:

- $a, b, \ldots, z$ are the node names, and denote values at the nodes after applying non-linearity.
- $\bar{a}, \bar{b}, \ldots, \bar{z}$ denote values before applying non-linearity.
- $W_{i,j}$ is the weight of edge from node i to node j.
- $s(\cdot) : \mathbb{R} \to \mathbb{R}$ is the non-linear activation function.
- β_j is the bias for node j.

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

BACKPROP EXAMPLE

For any node j, let $\bar{j}$ denote the value obtained before applying the non-linearity g.



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

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

BACKPROP EXAMPLE

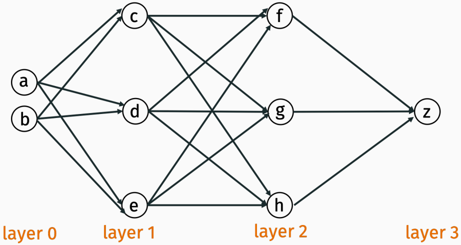
Goal: Compute the gradient $\nabla f(\boldsymbol{\theta}, \mathbf{x})$, which contains the partial derivatives with respect to every parameter:

- $\partial z / \partial \beta_z$
- $\partial z / \partial W_{f,z}$, $\partial z / \partial W_{g,z}$, $\partial z / \partial W_{h,z}$
- $\partial z / \partial \beta_f$, $\partial z / \partial \beta_g$, $\partial z / \partial \beta_h$
- $\partial z / \partial W_{c,f}$, $\partial z / \partial W_{c,g}$, $\partial z / \partial W_{c,h}$
- $\partial z / \partial W_{d,f}$, $\partial z / \partial W_{d,g}$, $\partial z / \partial W_{d,h}$
- $\vdots$
- $\partial z / \partial W_{a,c}$, $\partial z / \partial W_{a,d}$, $\partial z / \partial W_{a,e}$

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

BACKPROP EXAMPLE

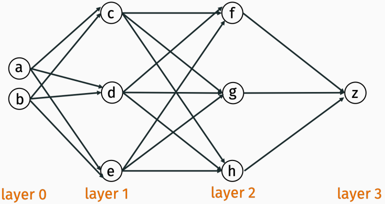
Step 1: Forward pass.



- 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.

$$\bar{c} = W_{a,c} \cdot a + W_{b,c} \cdot b + \beta_c \qquad c = s(\bar{c})$$

$$\bar{d} = W_{a,d} \cdot a + W_{b,d} \cdot b + \beta_d \qquad d = s(\bar{d})$$

$$\bar{e} = W_{a,e} \cdot a + W_{b,e} \cdot b + \beta_e \qquad e = s(\bar{e})$$

$$\bar{f} = W_{c,f} \cdot c + W_{d,f} \cdot d + W_{e,f} \cdot e + \beta_f \qquad f = s(\bar{f})$$

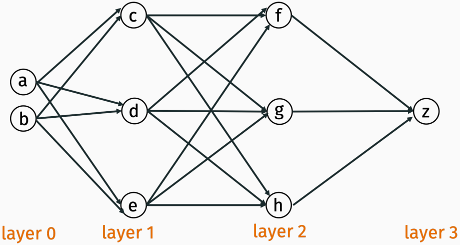
$$\vdots$$

$$\bar{z} = W_{f,z} \cdot f + W_{g,z} \cdot g + W_{h,z} \cdot f + \beta_z \qquad z = s(\bar{z})$$

Question: What is runtime in terms of # of parameters P?

BACKPROP EXAMPLE

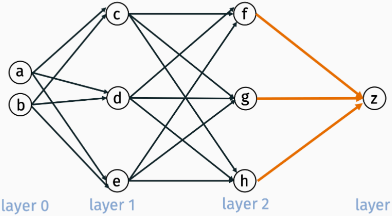
Step 2: Backward pass.



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

BACKPROP EXAMPLE

Step 2: Backward pass. Deepest layer.



$$\frac{\partial z}{\partial \beta_z} = \frac{\partial \bar{z}}{\partial \beta_z} \cdot \frac{\partial z}{\partial \bar{z}} = 1 \cdot s'(\bar{z})$$

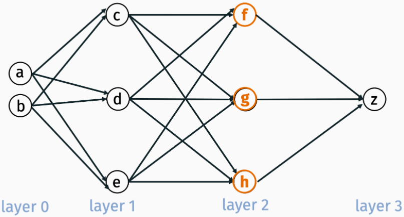
$$\frac{\partial z}{\partial W_{f,z}} = \frac{\partial \bar{z}}{\partial W_{f,z}} \cdot \frac{\partial z}{\partial \bar{z}} = f \cdot s'(\bar{z})$$

$$\frac{\partial z}{\partial W_{g,z}} = \frac{\partial \bar{z}}{\partial W_{g,z}} \cdot \frac{\partial z}{\partial \bar{z}} = g \cdot s'(\bar{z})$$

$$\frac{\partial z}{\partial W_{h,z}} = \frac{\partial \bar{z}}{\partial W_{h,z}} \cdot \frac{\partial z}{\partial \bar{z}} = h \cdot s'(\bar{z})$$

BACKPROP EXAMPLE

Step 2: Backward pass.



$$\frac{\partial z}{\partial f} = \frac{\partial \bar{z}}{\partial f} \cdot \frac{\partial z}{\partial \bar{z}} = W_{f,z} \cdot s'(\bar{z})$$

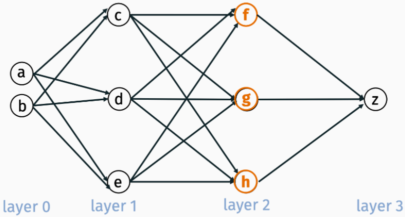
$$\frac{\partial z}{\partial g} = \frac{\partial \bar{z}}{\partial g} \cdot \frac{\partial z}{\partial \bar{z}} = W_{g,z} \cdot s'(\bar{z})$$

$$\frac{\partial z}{\partial h} = \frac{\partial \bar{z}}{\partial h} \cdot \frac{\partial z}{\partial \bar{z}} = W_{h,z} \cdot s'(\bar{z})$$

Compute partial derivatives with respect to nodes, even though these are not used in the gradient.

BACKPROP EXAMPLE

Step 2: Backward pass.



$$\frac{\partial z}{\partial \bar{f}} = \frac{\partial z}{\partial f} \cdot \frac{\partial f}{\partial \bar{f}} = \frac{\partial z}{\partial f} \cdot s'(\bar{f})$$

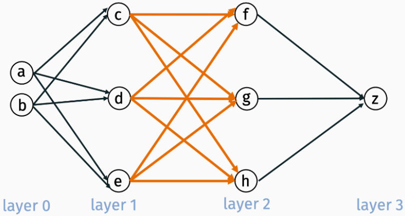
$$\frac{\partial z}{\partial \bar{g}} = \frac{\partial z}{\partial g} \cdot \frac{\partial g}{\partial \bar{g}} = \frac{\partial z}{\partial g} \cdot s'(\bar{g})$$

$$\frac{\partial z}{\partial \bar{h}} = \frac{\partial z}{\partial h} \cdot \frac{\partial h}{\partial \bar{h}} = \frac{\partial z}{\partial h} \cdot s'(\bar{h})$$

And for “pre-nonlinearity” nodes.

BACKPROP EXAMPLE

Step 2: Backward pass. Next layer.



$$\frac{\partial z}{\partial \beta_f} = \frac{\partial z}{\partial \bar{f}} \cdot \frac{\partial \bar{f}}{\partial \beta_f} = \frac{\partial z}{\partial \bar{f}} \cdot 1$$

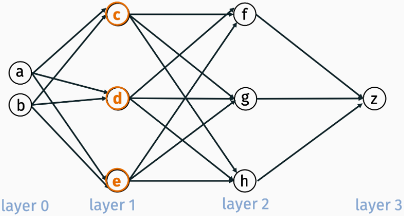
$$\frac{\partial z}{\partial W_{c,f}} = \frac{\partial z}{\partial \bar{f}} \cdot \frac{\partial \bar{f}}{\partial W_{c,f}} = \frac{\partial z}{\partial \bar{f}} \cdot c$$

$$\frac{\partial z}{\partial W_{d,f}} = \frac{\partial z}{\partial \bar{f}} \cdot \frac{\partial \bar{f}}{\partial W_{d,f}} = \frac{\partial z}{\partial \bar{f}} \cdot d$$

$$\frac{\partial z}{\partial W_{e,f}} = \frac{\partial z}{\partial \bar{f}} \cdot \frac{\partial \bar{f}}{\partial W_{e,f}} = \frac{\partial z}{\partial \bar{f}} \cdot e$$

BACKPROP EXAMPLE

Step 2: Backward pass. Next layer. Use multivariate chain rule.



$$\frac{\partial z}{\partial c} = \frac{\partial z}{\partial \bar{f}} \cdot \frac{\partial \bar{f}}{\partial c} + \frac{\partial z}{\partial \bar{g}} \cdot \frac{\partial \bar{g}}{\partial c} + \frac{\partial z}{\partial \bar{h}} \cdot \frac{\partial \bar{h}}{\partial c}$$

$$= \frac{\partial z}{\partial \bar{f}} \cdot W_{c,f} + \frac{\partial z}{\partial \bar{g}} \cdot W_{c,g} + \frac{\partial z}{\partial \bar{h}} \cdot W_{c,h}$$

$$\frac{\partial z}{\partial d} = \frac{\partial z}{\partial \bar{f}} \cdot W_{d,f} + \frac{\partial z}{\partial \bar{g}} \cdot W_{d,g} + \frac{\partial z}{\partial \bar{h}} \cdot W_{d,h}$$

$$\frac{\partial z}{\partial e} = \frac{\partial z}{\partial \bar{f}} \cdot W_{e,f} + \frac{\partial z}{\partial \bar{g}} \cdot W_{e,g} + \frac{\partial z}{\partial \bar{h}} \cdot W_{e,h}$$

BACKPROP LINEAR ALGEBRA

Linear algebraic view.

Let $\mathbf{v}_i$ be a vector containing the value of all nodes j in layer i.

$$\mathbf{v}_3 = \begin{bmatrix} z \end{bmatrix} \qquad \mathbf{v}_2 = \begin{bmatrix} f \\ g \\ h \end{bmatrix} \qquad \mathbf{v}_1 = \begin{bmatrix} c \\ d \\ e \end{bmatrix}$$

Let $\bar{\mathbf{v}}_i$ be a vector containing $\bar{j}$ for all nodes j in layer i.

$$\bar{\mathbf{v}}_3 = \begin{bmatrix} \bar{z} \end{bmatrix} \qquad \bar{\mathbf{v}}_2 = \begin{bmatrix} \bar{f} \\ \bar{g} \\ \bar{h} \end{bmatrix} \qquad \bar{\mathbf{v}}_1 = \begin{bmatrix} \bar{c} \\ \bar{d} \\ \bar{f} \end{bmatrix}$$

Note: $\mathbf{v}_i = s(\bar{\mathbf{v}}_i)$, where s is applied entrywise.

Linear algebraic view.

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

$$\boldsymbol{\delta}_3 = \begin{bmatrix} 1 \end{bmatrix} \qquad \boldsymbol{\delta}_2 = \begin{bmatrix} \partial z/\partial f \\ \partial z/\partial g \\ \partial z/\partial h \end{bmatrix} \qquad \boldsymbol{\delta}_1 = \begin{bmatrix} \partial z/\partial c \\ \partial z/\partial d \\ \partial z/\partial e \end{bmatrix}$$

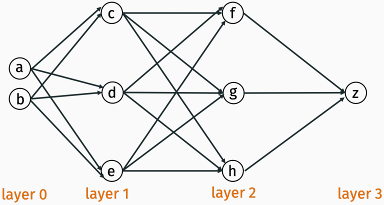
Let $\bar{\boldsymbol{\delta}}_i$ be a vector containing $\partial z/\partial \bar{j}$ for all nodes j in layer i.

$$\bar{\boldsymbol{\delta}}_3 = \begin{bmatrix} \partial z/\partial \bar{z} \end{bmatrix} \qquad \bar{\boldsymbol{\delta}}_2 = \begin{bmatrix} \partial z/\partial \bar{f} \\ \partial z/\partial \bar{g} \\ \partial z/\partial \bar{h} \end{bmatrix} \qquad \bar{\boldsymbol{\delta}}_1 = \begin{bmatrix} \partial z/\partial \bar{c} \\ \partial z/\partial \bar{d} \\ \partial z/\partial \bar{e} \end{bmatrix}$$

Note: $\bar{\boldsymbol{\delta}}_i = s'(\bar{\mathrm{v}}_i) \times \boldsymbol{\delta}_i$ where × denotes entrywise multiplication.

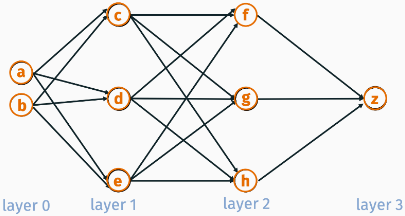
BACKPROP LINEAR ALGEBRA

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



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

BACKPROP LINEAR ALGEBRA



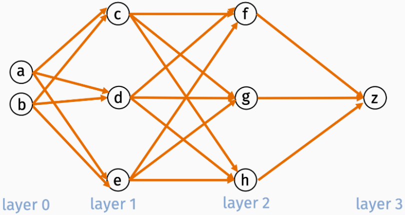
Claim 1: Node derivative computation is matrix multiplication.

$$\boldsymbol{\delta}_i = \mathbf{W}_i^T \bar{\boldsymbol{\delta}}_{i+1}$$

What is the computational complexity if $\mathbf{W}_i \in \mathbb{R}^{k \times m}$?

BACKPROP LINEAR ALGEBRA

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

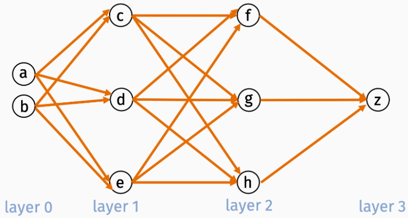


$$\mathbf{\Delta}_2 = \begin{bmatrix} \partial z/\partial W_{f,z} & \partial z/\partial W_{g,z} & \partial z/\partial W_{h,z} \end{bmatrix}$$

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

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

BACKPROP LINEAR ALGEBRA



Claim 2: Weight derivative computation is an outer-product between the $(i+1)^{\text{st}}$ derivative vector and the i^{th} value vector.

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

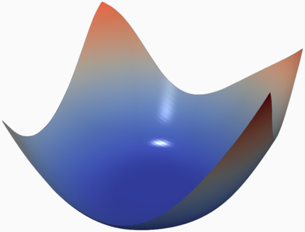
What is the computational complexity of computing the derivatives for a single weight matrix $\mathbf{W}_i \in \mathbb{R}^{k \times m}$?

Takeaways:

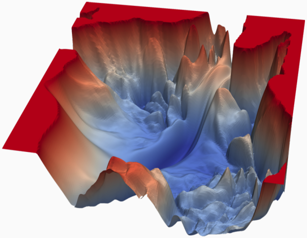
- Backpropagation can be used to compute derivatives for all weights and biases for any feedforward neural network.
- Total computation cost is linear in the number of parameters of the network to compute $f(\boldsymbol{\theta}, \mathbf{x})$ and thus $\nabla L(y, f(\boldsymbol{\theta}, \mathbf{x}))$ for a single training example $\mathbf{x}, \mathbf{y}$.
- SGD can be run in $O(P)$ time per iteration for a network with P parameters.
- Final computation boils down to linear algebra operations (matrix multiplication and vector operations) which can be performed quickly on a GPU.

CONVERGENCE

Least squares regression, logistic regression, SVMs, even all of these with kernels lead to convex losses.



convex loss



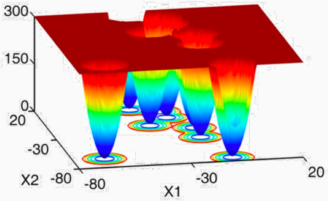
cross-entropy loss for neural net

Neural networks very much do not...

CONVERGENCE

But SGD still performs remarkably well in practice. Understanding this phenomenon is still an open research question in machine learning and optimization. Current hypotheses include:

- Initialization seems important (random uniform vs. random Gaussian vs. Xavier initialization vs. He initialization vs. etc.)
- Randomization helps in escaping local minima.
- Many local minima are global minima?
- SGD finds "good" local minima?



AUTODIFF

Issue: Backpropagation + SGD is fast, but tedious to implement.

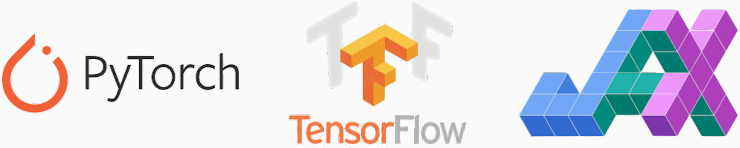
Typical to use automatic differentiation, which can compute the gradient of pretty much any function you can code up.

```
def loss(W, b):
    preds = predict(W, b, inputs)
    label_probs = preds * targets + (1 - preds)
    return -np.sum(jnp.log(label_probs))

from jax import grad
W_grad, b_grad = grad(loss, (0, 1))(W, b)
print(’W_grad’, W_grad)
print(’b_grad’, b_grad)
```

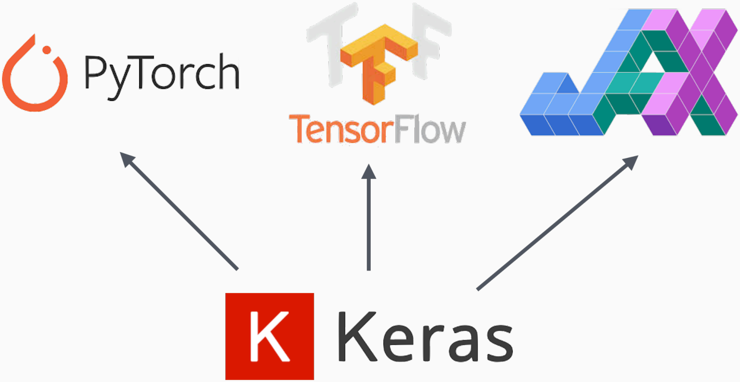
LIBRARIES

May mature low-level libraries that handle neural network representation, autodiff, have built in optimizers (SGD, ADAM, etc.), etc.



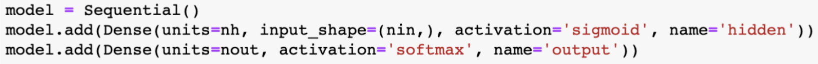
LIBRARIES

Higher-level libraries like Keras make it even easy to work with this software. Tools for easily defining and building neural networks with specific structure, tracking training, etc.



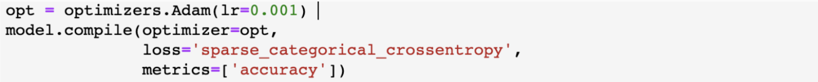
LIBRARIES

Define:



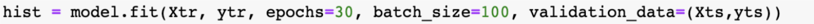
```
model = Sequential()
model.add(Dense(units=nh, input_shape=(nin,), activation='sigmoid', name='hidden'))
model.add(Dense(units=nout, activation='softmax', name='output'))
```

Compile:



```
opt = optimizers.Adam(lr=0.001)
model.compile(optimizer=opt,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Train:



```
hist = model.fit(Xtr, ytr, epochs=30, batch_size=100, validation_data=(Xts,yts))
```

We will release two demos on working with Keras: `keras_demo_synthetic.ipynb` and `keras_demo_mnist.ipynb`