CS-GY 6923: Lecture 9
Support Vector Machines, Neural Nets Introduction

NYU Tandon School of Engineering, Prof. Christopher Musco

How to use **non-linear kernels** with logistic regression.

- Often leads to better classification than basic linear logistic regression.
- Equivalent to feature transformation, but often computationally faster.

EXAMPLES OF NON-LINEAR KERNELS

Commonly used positive semidefinite (PSD) kernel functions:

- Linear (inner-product) kernel: $k(\mathbf{x}, \mathbf{y}) = \langle \mathbf{x}, \mathbf{y} \rangle$
- Gaussian RBF Kernel: $k(\mathbf{x}, \mathbf{y}) = e^{-\|\mathbf{x}-\mathbf{y}\|_2^2/\sigma^2}$
- Laplace Kernel: $k(\mathbf{x}, \mathbf{y}) = e^{-\|\mathbf{x}-\mathbf{y}\|_2/\sigma}$
- Polynomial Kernel: $k(\mathbf{x}, \mathbf{y}) = (\langle \mathbf{x}, \mathbf{y} \rangle + 1)^q$.

Recall: Every PSD kernel has a corresponding feature transformation $\phi : \mathbb{R}^d \to \mathbb{R}^m$.

$$k(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x})^T \phi(\mathbf{y}))$$

Sometimes $\phi(\mathbf{x})$ is simple and explicit. **More often, it is not.**

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad \phi(\mathbf{x}) = \begin{bmatrix} 1 \\ \sqrt{2}x_1 \\ \sqrt{2}x_2 \\ \sqrt{2}x_3 \\ x_1^2 \\ x_2^2 \\ x_3^2 \\ \sqrt{2}x_1x_2 \\ \sqrt{2}x_1x_3 \\ \sqrt{2}x_2x_3 \end{bmatrix}$$

Degree 2 polynomial kernel, $k(\mathbf{x}, \mathbf{w}) = (\mathbf{x}^T\mathbf{w} + 1)^2$.

Typically doesn't matter because we only need to compute the kernel Gram matrix **K** to retrofit algorithms like logistic or linear regression to use non-linear kernels.

$$\phi(\mathbf{X})\,\phi(\mathbf{X})^T = \begin{bmatrix} \phi(\vec{x}_1) \\ \phi(\vec{x}_2) \\ \vdots \\ \phi(\vec{x}_n) \end{bmatrix} \begin{bmatrix} \phi(\vec{x}_1) & \phi(\vec{x}_2) & \cdots & \phi(\vec{x}_n) \end{bmatrix} = \mathbf{K}, \quad \mathbf{K}_{ij} = k(\vec{x}_i, \vec{x}_j)$$

Support Vector Machines (SVMs): Another algorithm for finding linear classifiers which is as popular as logistic regression.

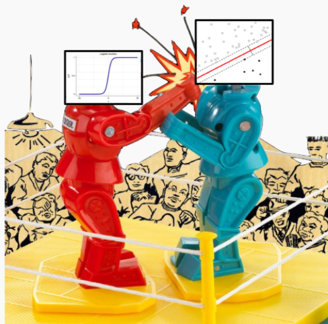
- Can also be combined with kernels.
- Developed from a pretty different perspective.
- But final algorithm is not that different.

- Invented in 1963 by Alexey Chervonenkis and Vladimir Vapnik. Also founders of VC-theory.
- First combined with non-linear kernels in 1993.

For some reason, SVMs are more commonly associated with non-linear kernels. For example, `sklearn`'s SVM classifier (called SVC) has support for non-linear kernels built in by default. Its logistic regression classifier does not.

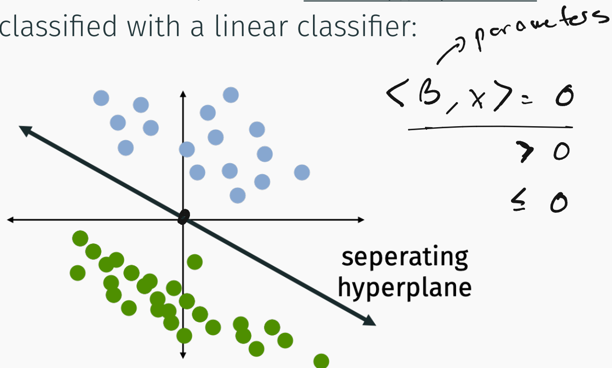
- I believe this is <u>mostly</u> for historical reasons and connections to theoretical machine learning.
- In the early 2000s SVMs where a "hot topic" in machine learning and their popularity persists.
- It is not clear to me if they are better than logistic regression, but honestly I'm not sure...

Next lab: `lab4.ipynb`.



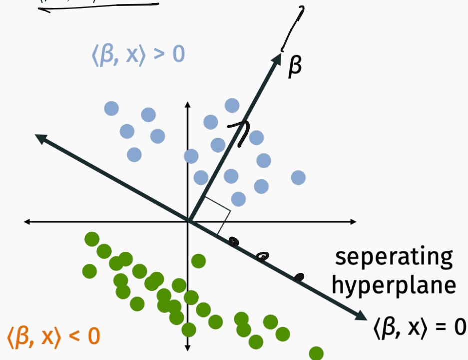
Machina-a-machina comparison of SVMs vs. logistic regression for a MNIST digit classification problem. Which provides better accuracy? Which is faster to train?

We call a dataset with binary labels linearly separable if it can be perfectly classified with a linear classifier:



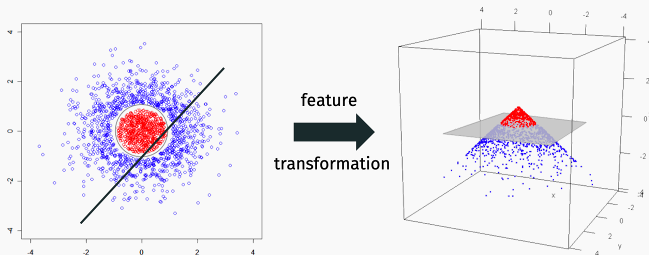
This the realizable setting we discussed in the learning theory lecture.

Formally, there exists a parameter β such that $\langle \beta, \mathbf{x} \rangle > 0$ for all $\mathbf{x}$ in class 1 and $\langle \beta, \mathbf{x} \rangle < 0$ for all $\mathbf{x}$ in class 0.



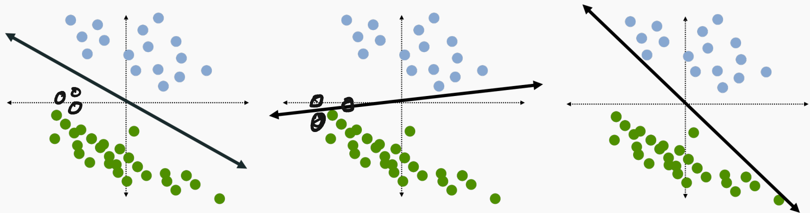
Note that if we multiply β by any constant c, $c\beta$ gives the same separating hyperplane because $\langle c\beta, \mathbf{x} \rangle = c\langle \beta, \mathbf{x} \rangle$.

A data set might be linearly separable when using a non-kernel/feature transformation even if it is not separable in the original space.



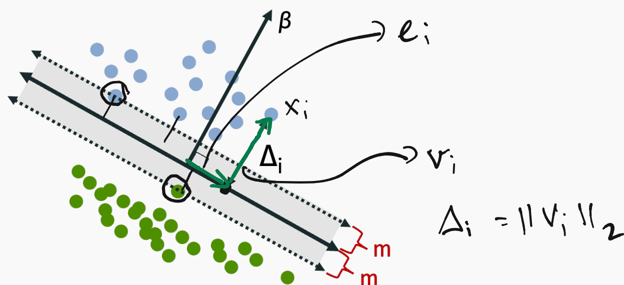
This data is separable when using a degree-2 polynomial kernel. If suffices for $\phi(\mathbf{x})$ to contain x_1^2 and x_2^2.

When data is linearly separable, there are typically multiple valid separating hyperplanes.



Which hyperplane/classification rule is best?

The margin m of a separating hyperplane is the minimum ℓ_2 (Euclidean) distance between a point in the dataset and the hyperplane.



$$m = \min_i \Delta_i \qquad \text{where} \qquad \Delta_i = \frac{|\langle \mathbf{x}_i, \boldsymbol{\beta} \rangle|}{\|\boldsymbol{\beta}\|_2}$$

$$v_i = \left(\frac{\|v_i\|_2}{\|\beta\|_2}\right) \cdot \beta$$

We have that $\mathbf{x}_i = \mathbf{v}_i + \mathbf{e}_i$ where $\mathbf{v}_i$ is parallel to $\boldsymbol{\beta}$ and $\mathbf{e}_i$ is perpendicular.

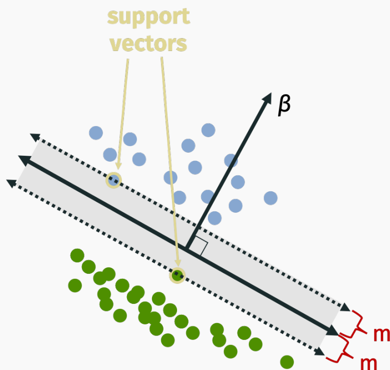
$$\Delta_i = \|\mathbf{v}_i\|_2 = \frac{1}{\|\mathbf{v}_i\|_2} \cdot \langle \mathbf{v}_i, \mathbf{v}_i \rangle = \frac{1}{\|\mathbf{v}_i\|_2} \cdot \frac{\|\mathbf{v}_i\|_2}{\|\boldsymbol{\beta}\|_2} \cdot |\langle \mathbf{v}_i, \boldsymbol{\beta} \rangle| = \frac{|\langle \mathbf{v}_i, \boldsymbol{\beta} \rangle|}{\|\boldsymbol{\beta}\|_2}.$$

Finally, we have that $\langle \mathbf{x}_i, \boldsymbol{\beta} \rangle = \langle \mathbf{v}_i, \boldsymbol{\beta} \rangle$ because $\langle \mathbf{e}_i, \boldsymbol{\beta} \rangle = 0$.

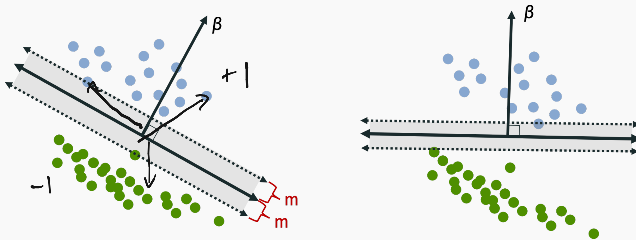
$$\Delta_i = \frac{|\langle v_i, \beta \rangle|}{\|\beta\|_2}$$

$$\langle x_i, \beta \rangle = \langle v_i + e_i, \beta \rangle = \langle v_i, \beta \rangle + \langle e_i, \beta \rangle, \quad \langle e_i, \beta \rangle = 0$$

A **support vector** is any data point $\mathbf{x}_i$ such that $\frac{|\langle \mathbf{x}_i, \boldsymbol{\beta} \rangle|}{\|\boldsymbol{\beta}\|_2} = m$.



A hard-margin support vector machine (SVM) classifier finds the **maximum margin (MM) linear classifier**.



I.e. the separating hyperplane which maximizes the margin m.

{0, 1}

Denote the maximum margin by m^*.

$\|\beta\|_2 = 1$

$$m^* = \max_{\boldsymbol{\beta}} \left[\min_{i \in 1,\dots,n} \frac{|\langle \mathbf{x}_i, \beta \rangle|}{\|\boldsymbol{\beta}\|_2} \right]$$
$$= \max_{\boldsymbol{\beta}} \left[\min_{i \in 1,\dots,n} \frac{y_i \cdot \langle \mathbf{x}_i, \boldsymbol{\beta} \rangle}{\|\boldsymbol{\beta}\|_2} \right]$$

where $y_i = -1, 1$ depending on what class $\mathbf{x}_i$.[1]

[1]Note that this is a different convention than the 0, 1 class labels we typically use.

Equivalent formulation:

$$m^* = \max_{\mathbf{v}:\|\mathbf{v}\|_2=1} \left[\min_{i \in 1,\ldots,n} y_i \cdot \langle \mathbf{x}_i, \mathbf{v} \rangle \right]$$

$$\frac{1}{m^*} = \min_{\substack{\mathbf{v}:\|\mathbf{v}\|_2=1 \\ c}} c \qquad \text{subject to} \quad y_i \cdot \langle \mathbf{x}_i, \mathbf{v} \rangle \geq 1 \text{ for all } i.$$

$$= \min_{\substack{\mathbf{v}:\|\mathbf{v}\|_2=1 \\ c}} \|c \cdot \mathbf{v}\|_2 \qquad \text{subject to} \quad y_i \cdot \langle \mathbf{x}_i, c \cdot \mathbf{v} \rangle \geq 1 \text{ for all } i.$$

Equivalent formulation:

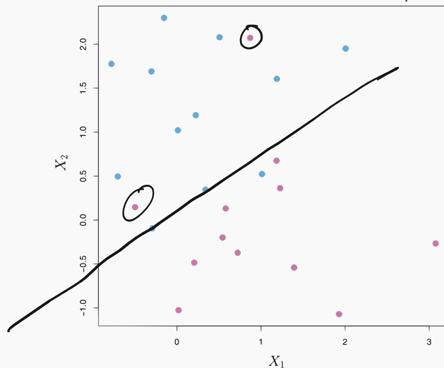
$$\min_{\boldsymbol{\beta}} \|\boldsymbol{\beta}\|_2^2 \quad \text{subject to} \quad y_i \cdot \langle \mathbf{x}_i, \boldsymbol{\beta} \rangle \geq 1 \text{ for all } i.$$

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

This is a **constrained optimization problem.** In particular, a linearly constrained quadratic program, which is a type of problem we have efficient optimization algorithms for.

HARD-MARGIN SVM

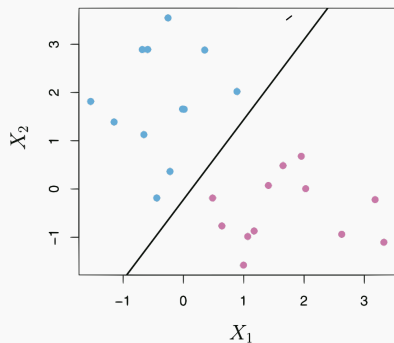
Hard-margin SVMs have a few critical issues in practice:

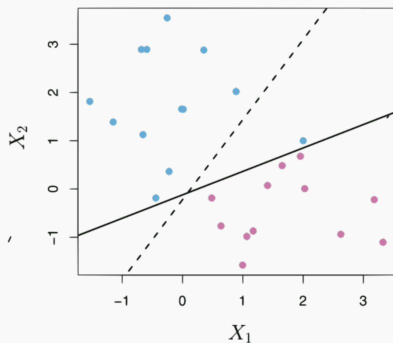


Data might not be linearly separable, in-which case the maximum margin classifier is not even defined.

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.

While important in theory, hard-margin SVMs have a few critical issues in practice:





Hard-margin SVM classifiers are not robust.

Solution: Allow the classifier to make some mistakes!

Hard margin objective:

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

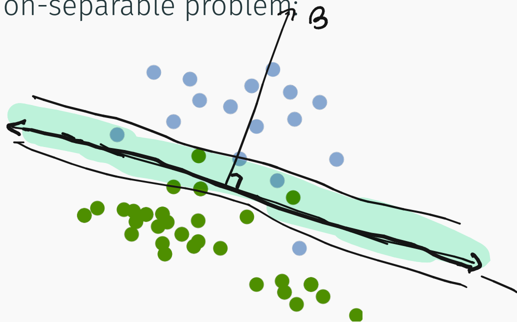
Soft margin objective:

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

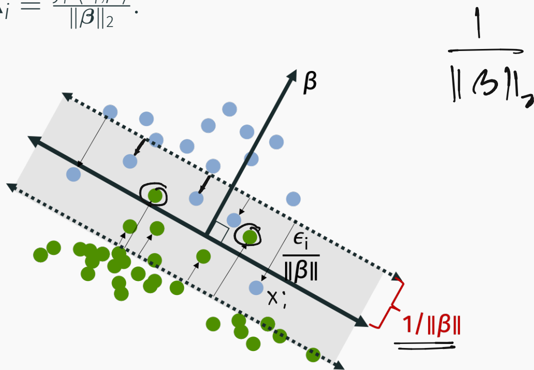
where $\epsilon_i \geq 0$ is a non-negative "slack variable". This is the magnitude of the error made on example $\mathbf{x}_i$. $\epsilon_1, \ldots, \epsilon_n$

$C \geq 0$ is a non-negative tuning parameter.

Example of a non-separable problem:



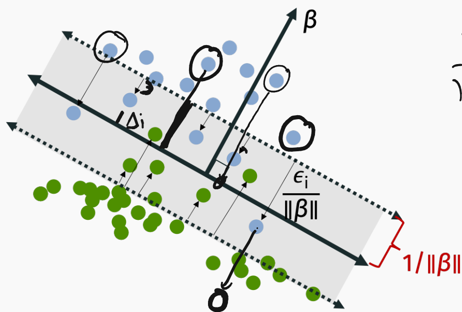
Recall that $\Delta_i = \frac{y_i \cdot \langle \mathbf{x}_i, \boldsymbol{\beta} \rangle}{\|\boldsymbol{\beta}\|_2}$.



Soft margin objective:

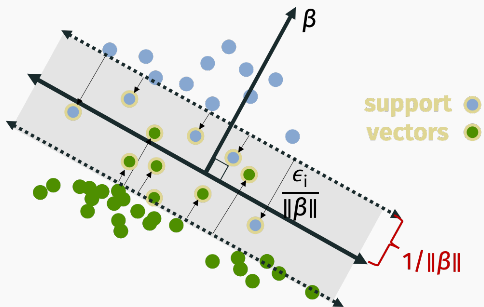
$$\min_{\boldsymbol{\beta}} \|\boldsymbol{\beta}\|_2^2 + C \sum_{i=1}^{n} \epsilon_i \quad \text{subject to} \quad y_i \cdot \langle \mathbf{x}_i, \boldsymbol{\beta} \rangle \geq 1 - \epsilon_i \text{ for all } i.$$

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



Soft margin objective:

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



Any $\mathbf{x}_i$ with a non-zero ϵ_i is a support vector.

Soft margin objective:

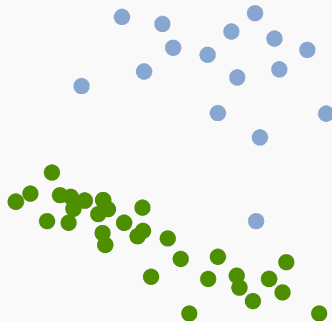
$C \geq 0$

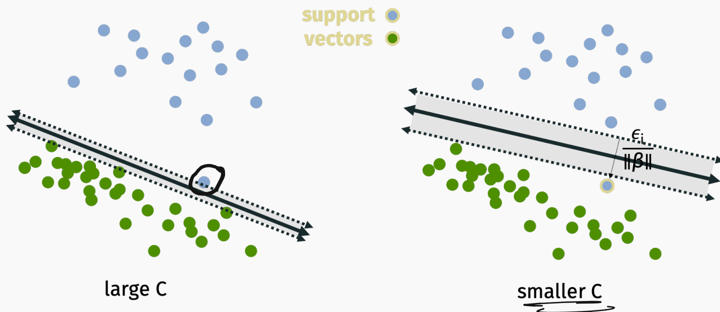
$$\min_{\boldsymbol{\beta}} \|\boldsymbol{\beta}\|_2^2 + C \sum_{i=1}^{n} \epsilon_i.$$

- 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.

Example dataset:





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

Reformulation of soft-margin objective:

$$\max_{\boldsymbol{\alpha}} \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i,j} y_i y_j \alpha_i \alpha_i \langle \mathbf{x}_i, \mathbf{x}_j \rangle - \frac{1}{2C}\sum_{i=1}^{n} \alpha_i^2$$

$$\text{subject to } \alpha_i \geq 0, \quad \sum_{i=1}^{n} \alpha_i y_i = 0.$$

$K(x_i, x_j)$ $\beta \in \mathbb{R}^d$ $\alpha \in \mathbb{R}^n$

Obtained by taking the Lagrangian dual of the objective. Beyond the scope of this class, but important for a few reasons:

- Objective only depends on inner products $\langle \mathbf{x}_i, \mathbf{x}_j \rangle$, which makes it clear how to combine the soft-margin SVM with a kernel.
- Dual formulation can be solved faster in low-dimensions.
- Possible to prove that α_i is only non-zero for the support vectors. When classifying a new data point, only need to compute inner products (or the non-linear kernel inner product) with this subset of training vectors.

Some basic transformations of the soft-margin objective:

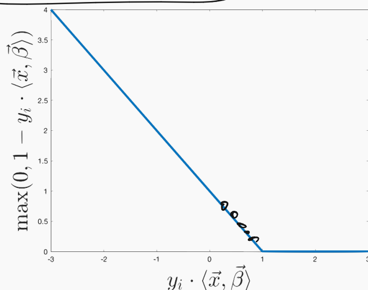
$$\min_{\boldsymbol{\beta}} \|\boldsymbol{\beta}\|_2^2 + C\sum_{i=1}^{n} \epsilon_i \quad \text{subject to} \quad y_i \cdot \langle \mathbf{x}_i, \boldsymbol{\beta}\rangle \geq 1 - \epsilon_i \text{ for all } i.$$

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

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

These are all equivalent. $\lambda = 1/C$ is just another scaling parameter.

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:

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

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

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

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

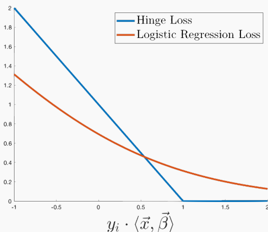
y_i = 1

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

Yi-Jen Chiang

Doris Aronow

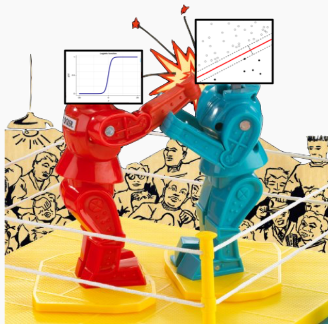
So, in the end, the function minimized when finding β for the standard **soft-margin SVM** is very similar to the objective function minimized when finding β using **logistic regression with ℓ_2 regularization.** Sort of...



Both functions can be optimized using first-order methods like gradient descent. This is now a common choice for large problems.

COMPARISON TO LOGISTIC REGRESSION

The jury is still out on how different these methods are...



- Work through `demo_mnist_svm.ipynb`.
- Then complete lab `lab4.ipynb`.

NEURAL NETWORKS

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:

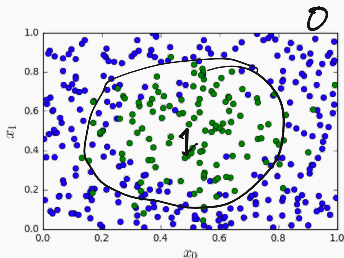
- Learn good features and a good model simultaneously.

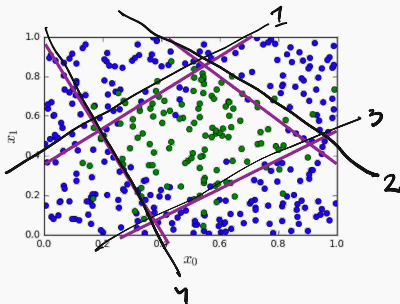
The hot-topic in machine learning right now.

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

Studied since the 1940s/50s. **Why the sudden attention?** More on history of neural networks at the end of lecture.

Classification when data is not linearly separable:

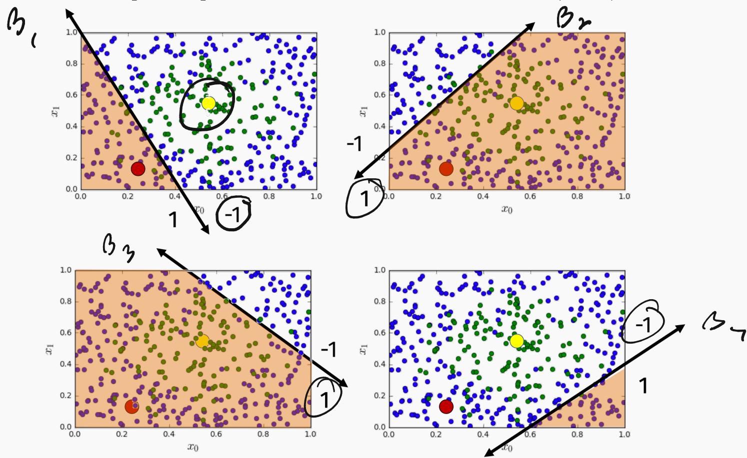




Could use feature transformations or a non-linear kernel.

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

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



● $\Rightarrow [1 \; 1 \; 1 \; -1]$ ● $\Rightarrow [-1 \; 1 \; 1 \; -1]$

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

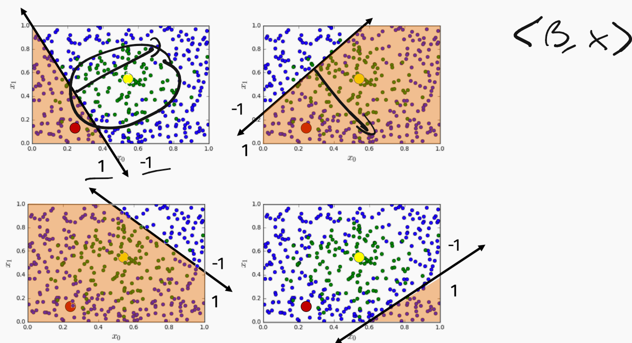
Question: After data transformation, how should we map a new vectors $\mathbf{u}$ to a class label?

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

SIMPLE MOTIVATING EXAMPLE

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 label (typically by thresholding).

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

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

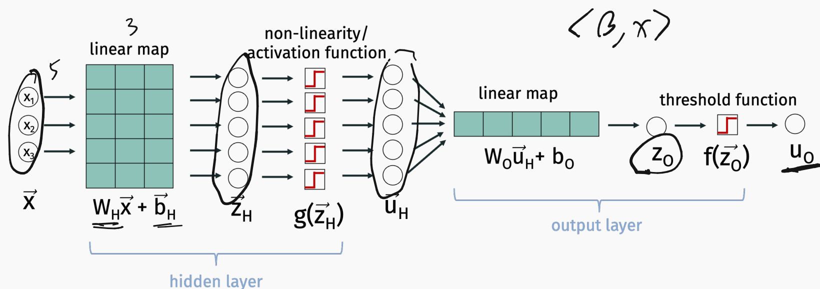
- $\mathbf{z}_H \in \mathbb{R}^{N_H} = \mathbf{W}_H \mathbf{x} + \mathbf{b}_h$.
- $\mathbf{u}_H = \text{sign}(\mathbf{z}_H)$
- $z_O \in \mathbb{R} = \mathbf{W}_O \mathbf{u}_H + \mathbf{b}_O$
- $u_O = \mathbb{1}[z_O > \lambda]$

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

$\mathbf{W}_H$, $\mathbf{W}_O$ are weight matrices and $\mathbf{b}_H$, $\mathbf{b}_O$ 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.[2]

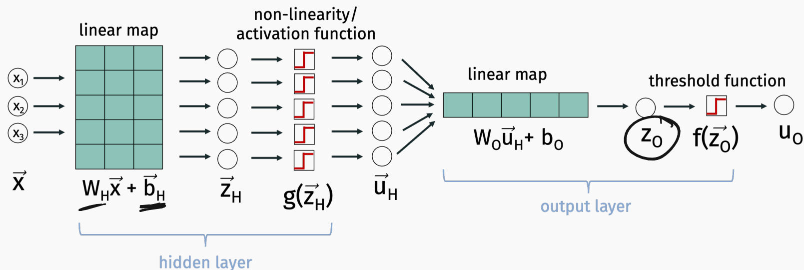


This is called a "multilayer perceptron": one of the oldest types of neural nets. Dates back to Frank Rosenblatt from 1958

- Number of input variables $N_I =$ 3
- Number of hidden variables $N_H =$ 5
- Number of output variables $N_O =$ 1

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

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

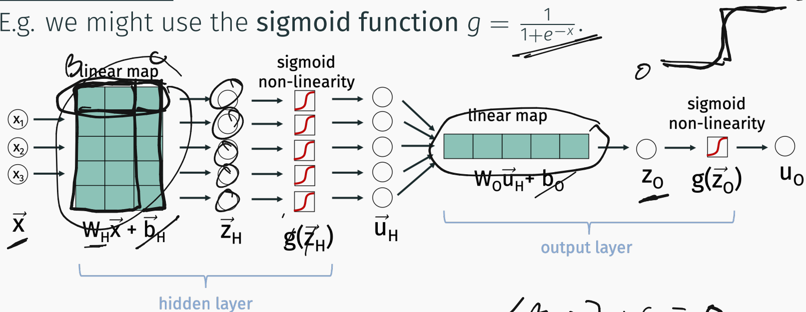


Training the model:

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

How to find optimal parameters?

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** $g = \frac{1}{1+e^{-x}}$.

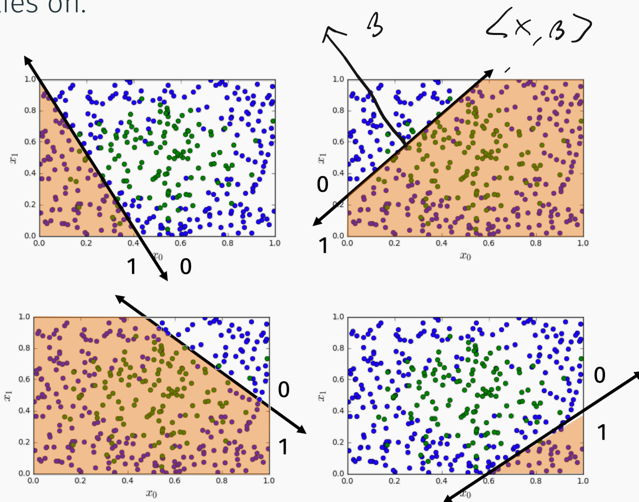


- Use cross-entropy loss:

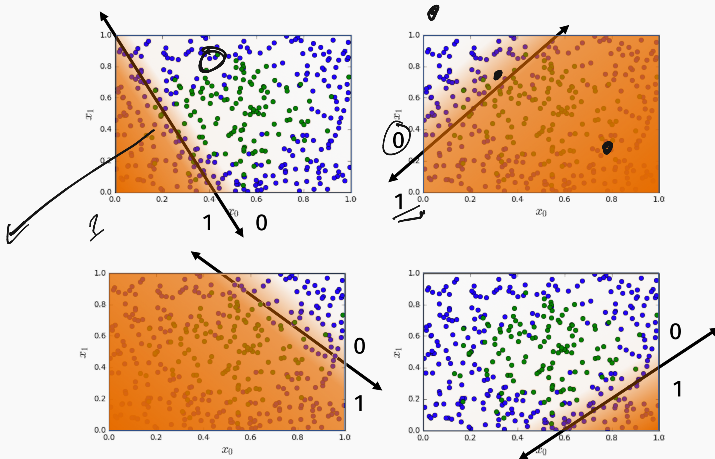
$$L(f(\mathbf{x}_i, \Theta), y_i) = -y \log(f(\mathbf{x}_i, \Theta)) - (1 - y_i) \log(1 - f(\mathbf{x}_i, \Theta))$$

- We will discuss soon how to compute gradients.

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



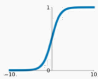
Features learned using sigmoid activation are real valued in $[0, 1]$. Mimic binary features.



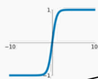
Things we can change in this basic classification network:

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

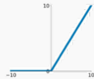
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$



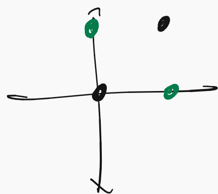
tanh
$\tanh(x)$

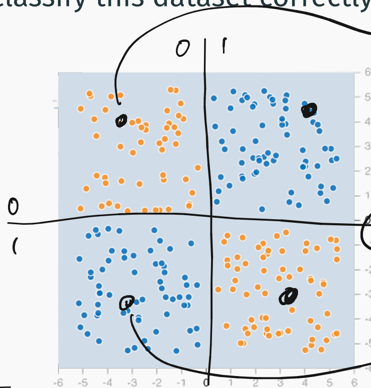


ReLU
$\max(0, x)$



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





https://playground.tensorflow.org/

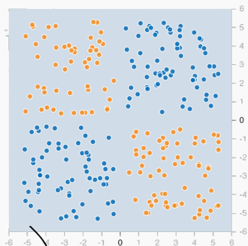
TEST YOUR INTUITION

$\log(0)$

$1 \cdot e^{-___}$

$z = \langle x, \beta \rangle$

$\left(\frac{1}{e^{100}}\right)$



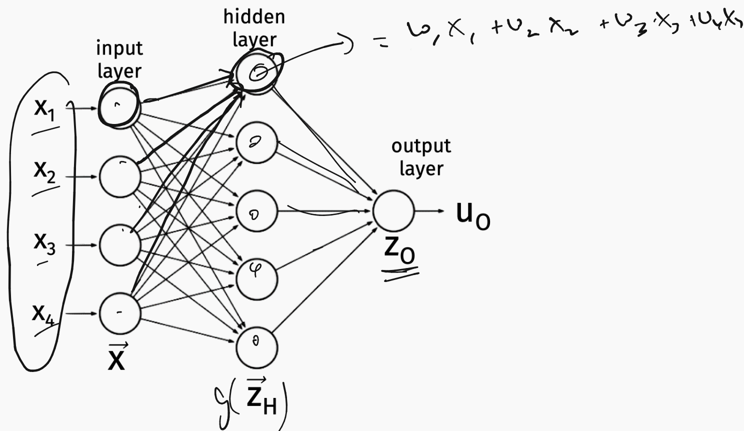
2

$\log\left(\frac{1}{1+e^{-z}}\right)$ → 0

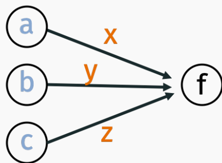
$\log\left(1-\left(\frac{1}{1+e^{-z}}\right)\right)$ → 1

1.00000000|001

Another common diagram for a 2-layered network:

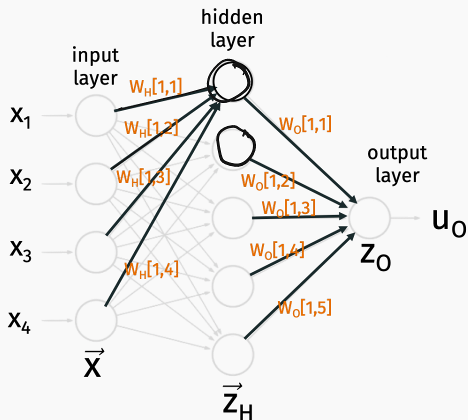


Neural network math:



f = ax + by + cz

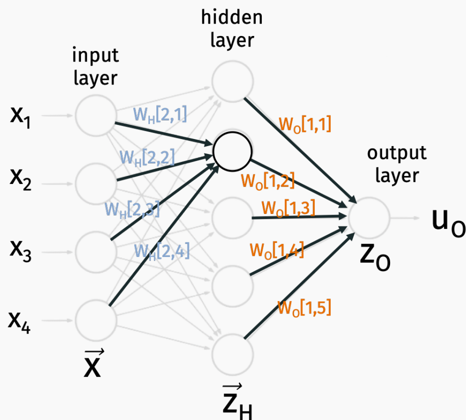
How to interpret:



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:

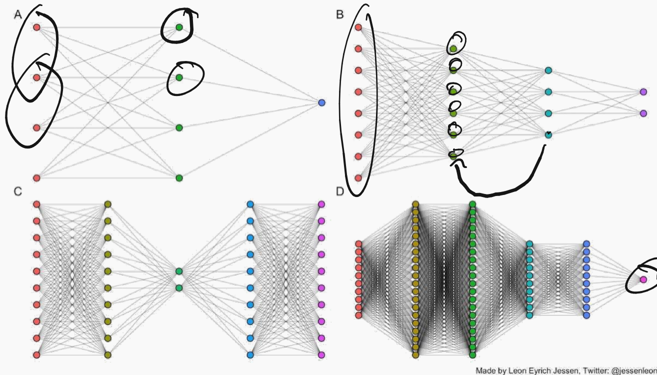


W_H and W_O are our weight matrices from before.

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

ARCHITECTURE VISUALIZATION

Effective way of visualize "architecture" of a neural network:

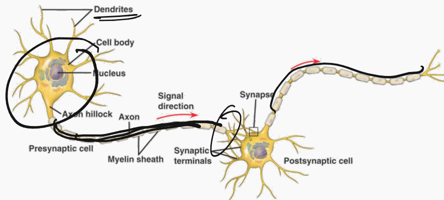


Visualize number of variables, types of connections, number of layers and their relative sizes.

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

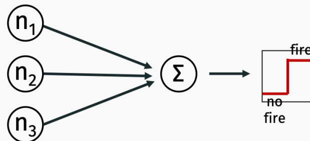
SOME HISTORY AND MOTIVATION

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.



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. We have gotten a lot better at resolving these issues though!

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.

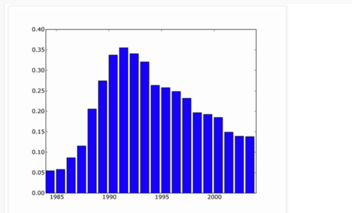
From 1990s - 2010, kernel methods, SVMs, and probabilistic methods began to dominate the literature in machine learning:

- 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`

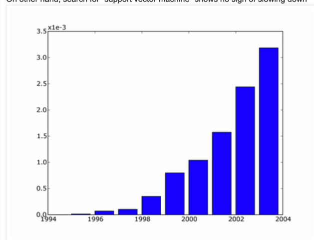
NEURAL NETWORK DECLINE

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



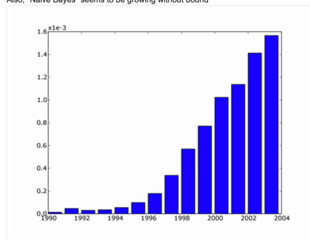
You can see a major upward trend starting around 1985 (that's when Yann LeCun and several others independently rediscovered backpropagation algorithm), peaking in 1992, and going downwards from then.

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



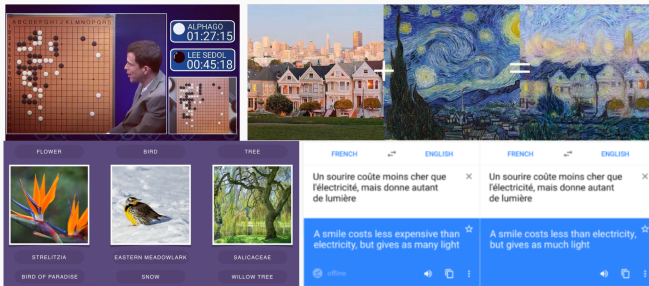
(1995 is when Vapnik and Cortez proposed the algorithm)

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.

2012 Results

Why 2012?

- Clever ideas in changing neural network architectures. E.g. convolutional units baked into the neural net.
- Wide-spread access to GPU computing power (CUDA and publicly available Nvidia GPU first released in 2007).

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?

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

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

W_i and b_i are the weight matrix and bias vector for layer i and g_i is the non-linearity (e.g. sigmoid). $\boldsymbol{\theta} = [\mathrm{W}_0, \mathrm{b}_0, \ldots, \mathrm{W}_\ell, \mathrm{b}_\ell]$ 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

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

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

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

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

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:

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

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

Applying chain rule to loss:

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

Binary cross-entropy example:

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

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

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