Announcements

Class is 176.

Matlab Grader homework,
1 and 2 (of less than 9) homeworks Due 22 April **tonight,** Binary graded.
For HW1, please get word count <100
Homework 3 (not released yet) due ~29 April

Jupiter "GPU" home work released Wednesday. Due 10 May

Projects: 19 Groups formed. Look at Piazza for help

Wed : Data resources
Projects

Today:
Stanford CNN 8
Linear models for classification, Backpropagation

Wednesday
Stanford CNN 9, Kernel methods (Bishop 6),
Play with Tensorflow playground before class http://playground.tensorflow.org

CPU vs GPU

	# Cores	Clock Speed	Memory	Price
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.4 GHz	Shared with system	$339
CPU (Intel Core i7-6950X)	10 (20 threads with hyperthreading)	3.5 GHz	Shared with system	$1723
GPU (NVIDIA Titan Xp)	3840	1.6 GHz	12 GB GDDR5X	$1200
GPU (NVIDIA GTX 1070)	1920	1.68 GHz	8 GB GDDR5	$399

CPU: Fewer cores, but each core is much faster and much more capable; great at sequential tasks

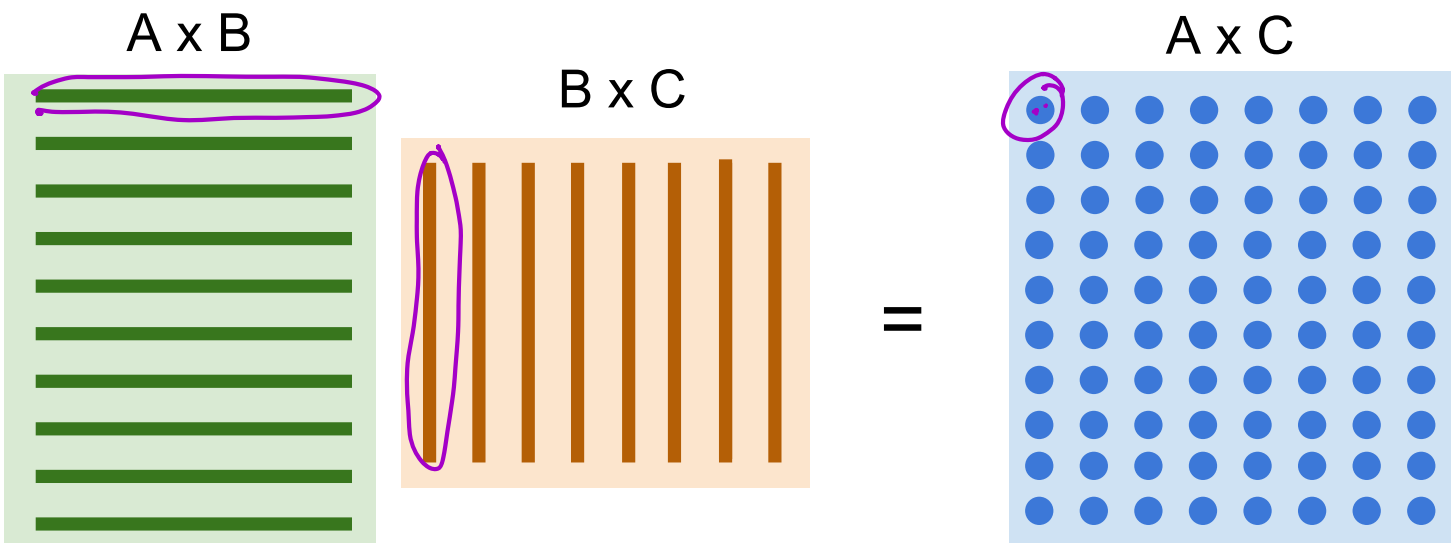
GPU: More cores, but each core is much slower and "dumber"; great for parallel tasks

Programming GPUs

- CUDA (NVIDIA only)
  - Write C-like code that runs directly on the GPU
  - Higher-level APIs: cuBLAS, cuFFT, cuDNN, etc
- OpenCL
  - Similar to CUDA, but runs on anything
  - Usually slower :(
- Udacity: Intro to Parallel Programming https://www.udacity.com/course/cs344
  - For deep learning just use existing libraries

GPU is efficient with matrices

Example: Matrix Multiplication



Main packages 2017

Today

A bit about these

Caffe
(UC Berkeley)

Caffe2
(Facebook)

Torch
(NYU / Facebook)

PyTorch
(Facebook)

Theano
(U Montreal)

TensorFlow
(Google)

Mostly these

Paddle
(Baidu)

CNTK
(Microsoft)

MXNet
(Amazon)

Developed by U Washington, CMU, MIT, Hong Kong U, etc but main framework of choice at AWS

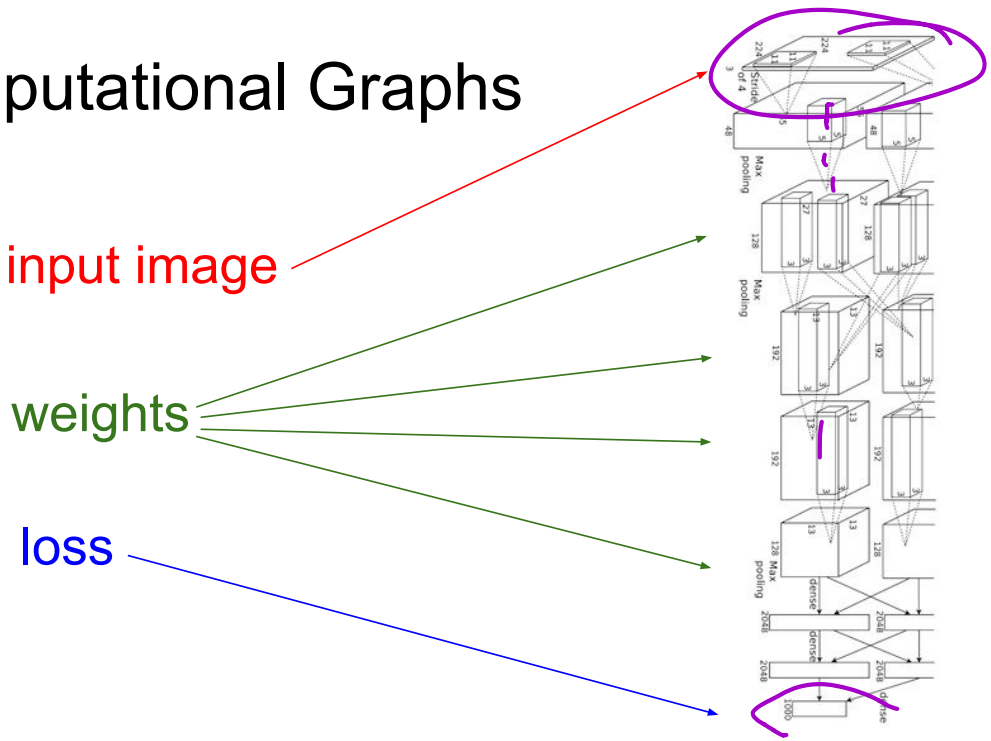
And others...

+Keras

DL frame work gives:

(1) Easily build big computational graphs
(2) Easily compute gradients in computational graphs
(3) Run it all efficiently on GPU (wrap cuDNN, cuBLAS, etc)

Computational Graphs



DL frame work gives:

(1) Easily build big computational graphs
(2) Easily compute gradients in computational graphs
(3) Run it all efficiently on GPU (wrap cuDNN, cuBLAS, etc)

Computational Graphs

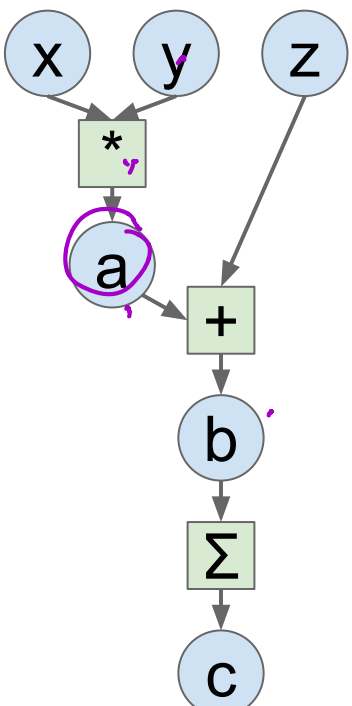
Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)
```



$\frac{\partial c}{\partial x}$ $\frac{\partial c}{\partial y}$ $\frac{\partial c}{\partial z}$

Computational Graphs

Numpy

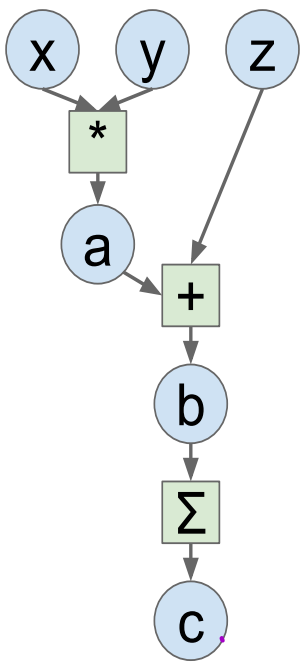
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



TensorFlow

```
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

My Advice:

TensorFlow is a safe bet for most projects. Not perfect but has huge community, wide usage. Maybe pair with high-level wrapper (Keras, Sonnet, etc)

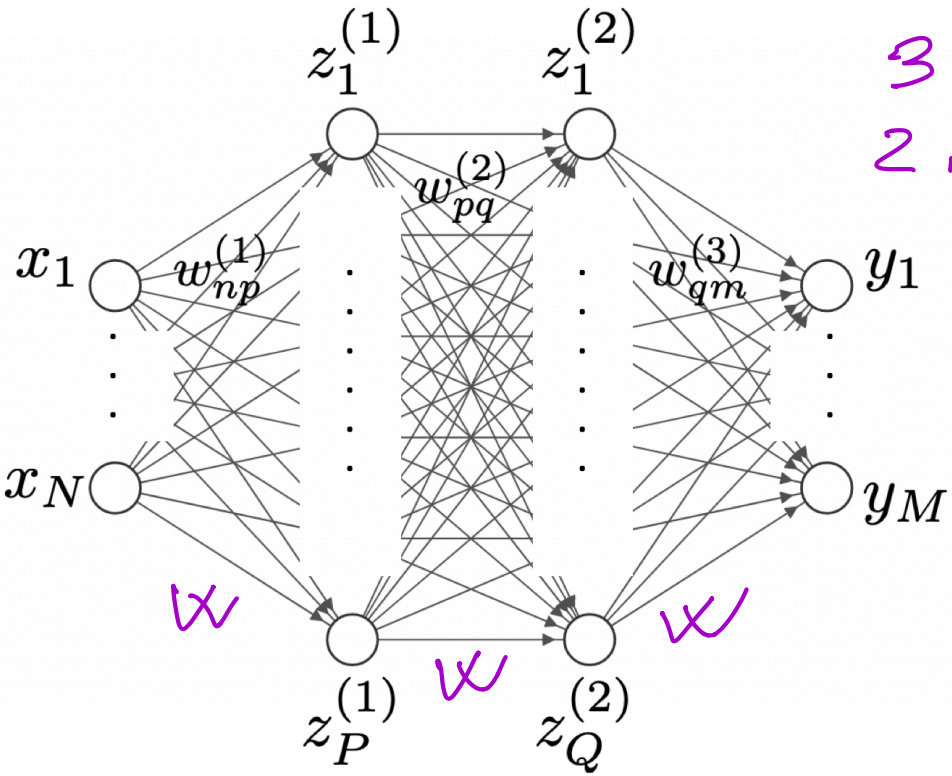
I think **PyTorch** is best for research. However still new, there can be rough patches.

Use **TensorFlow** for one graph over many machines

Consider **Caffe**, **Caffe2,** or **TensorFlow** for production deployment

Consider **TensorFlow** or **Caffe2** for mobile

What NN is this?



FC

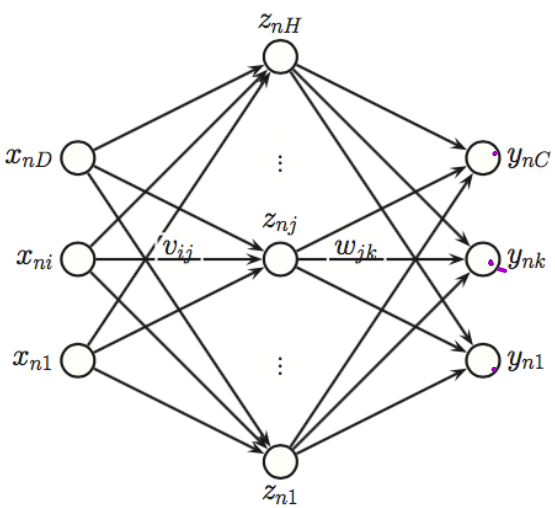
3 layer

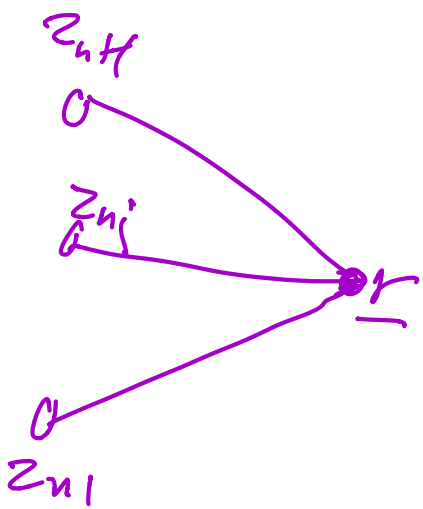
2 Hidden layers

FNN

Classification vs regression

Range 1 to range N





What is “linear” classification?

Classification is intrinsically non-linear

It puts non-identical things in the same class, so a difference in input vector sometimes causes zero change in the answer

“Linear classification” means that **the part that adapts is linear**

The adaptive part is followed by a fixed non-linearity.

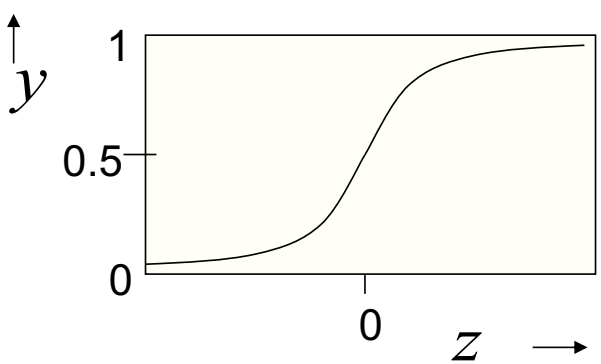
It may be preceded by a fixed non-linearity (e.g. nonlinear basis functions).

$$y(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0,$$

adaptive linear function

$$Decision = f(y(\mathbf{x}))$$

fixed non-linear function



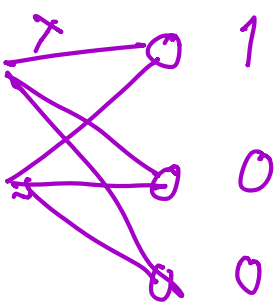
Representing the target values for classification

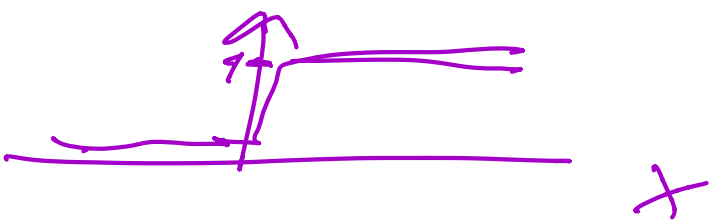
For two classes, we use an output with target values **1** for the "positive" class and **0** (or **-1**) for the other class

For probabilistic class labels the target value is **P(t=1)** and the model output can also represent **P(y=1).**

For **N classes** we often use a vector of N target values containing an **1** at the correct class and **0** elsewhere.

For probabilistic labels we can then use a vector of class probabilities as the target vector.





Three approaches to classification

Use **discriminant functions** directly without probabilities:

Convert input vector into real values. A simple operation (like thresholding) can get the class.

Choose real values to maximize the useable information about the class label that is in the real value.

Infer **conditional class probabilities**: $p(class = C_k \mid \mathbf{x})$

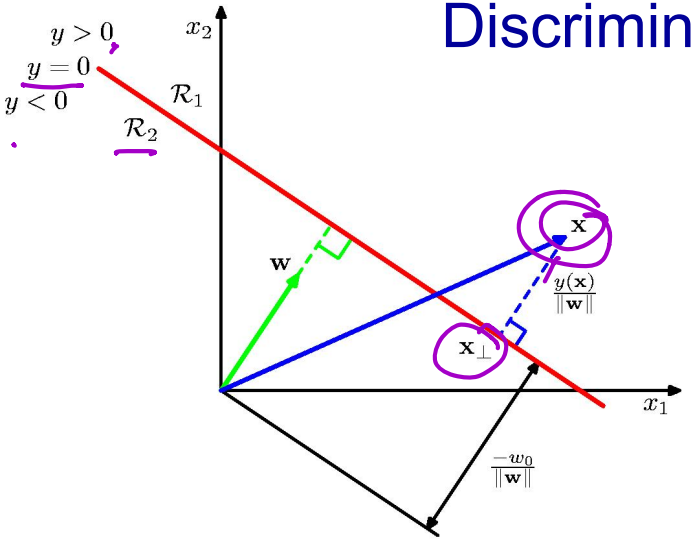
Compute the conditional probability of each class.

Then make a decision that minimizes some loss function

Compare the probability of the input under separate, class-specific, generative models.

E.g. fit a multivariate **Gaussian** to the input vectors of each class and see which Gaussian makes a test data vector most probable. (Is this the best bet?)

Discriminant functions



The planar decision surface in data-space for the simple linear discriminant function:

$$\mathbf{w}^T\mathbf{x} + w_0 \geq 0$$

$y_\perp = w^T x_\perp + w_0 = 0$

$\|w\|_2^2 = w^T w$

$\Rightarrow x = x_\perp + r\frac{w}{\|w\|_2}$

X on plane => y=0 =>

$w^T x = w^T x_\perp + r\frac{w^T w}{\|w\|_2}$

$= -w_0 + r\|w\|_2$

Distance from plane

$r = \frac{y}{\|w\|_2}$

Discriminant functions for N>2 classes

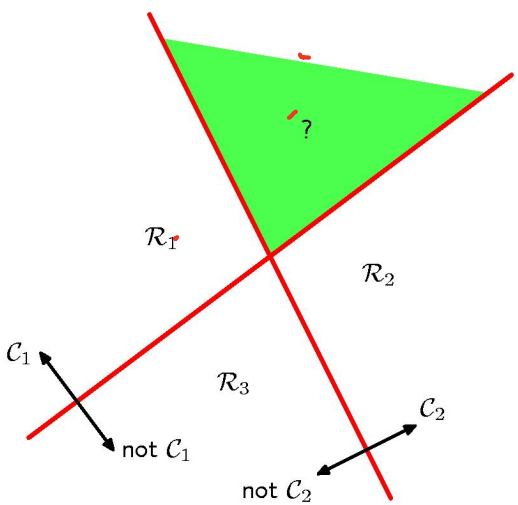
One possibility is using N two-way discriminant functions.

Each function discriminates one class from the rest.

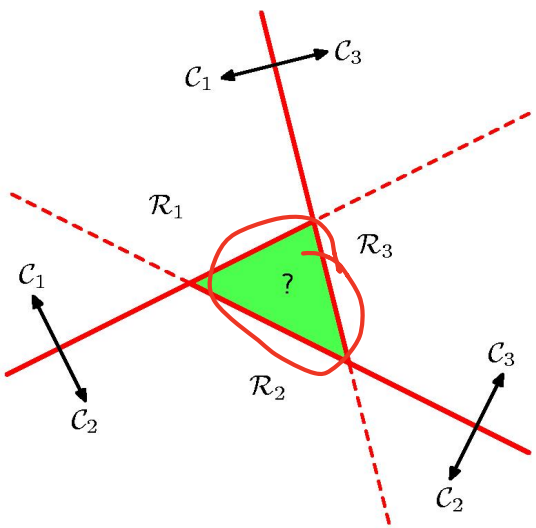
Another is using N(N-1)/2 two-way discriminant functions

Each function discriminates between two particular classes.

Both methods have problems



More than one good answer

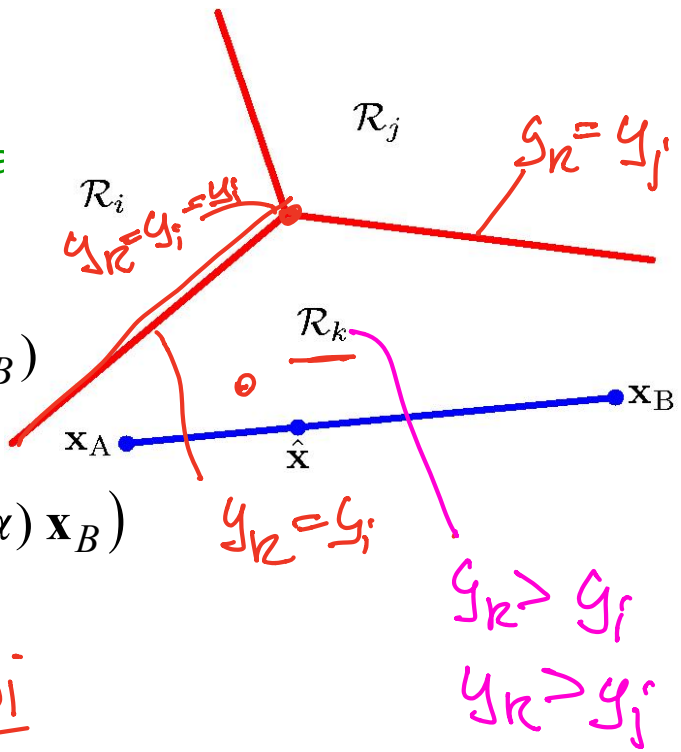


Two-way preferences need not be transitive!

A simple solution (4.1.2)

Use N discriminant functions, and pick the max. $y_i, y_j, y_k \cdots$

This is guaranteed to give consistent a convex decision regions if y is linear.



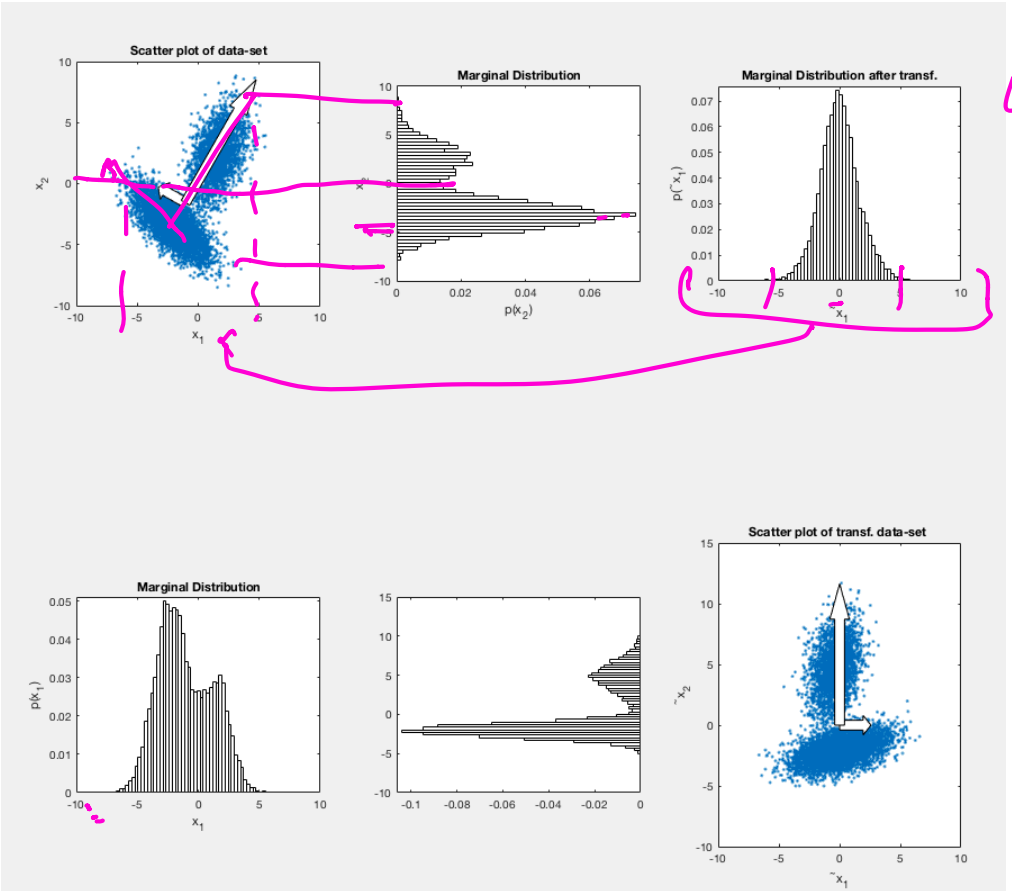
$y_k(\mathbf{x}_A) > y_j(\mathbf{x}_A)$ *and* $y_k(\mathbf{x}_B) > y_j(\mathbf{x}_B)$

implies (for positive α*) that*

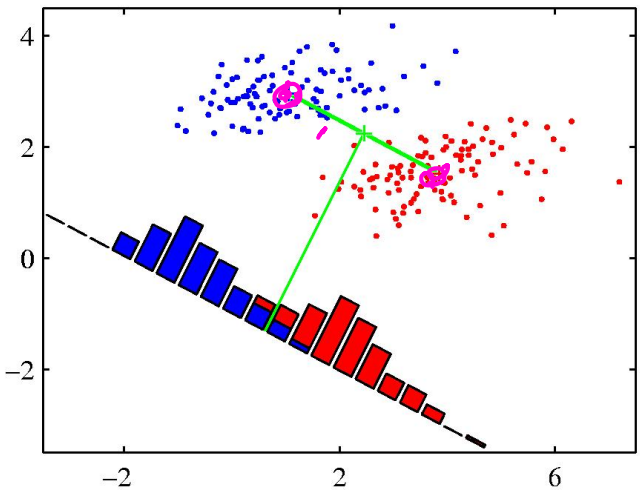
$y_k(\alpha \mathbf{x}_A + (1-\alpha)\mathbf{x}_B) > y_j(\alpha \mathbf{x}_A + (1-\alpha)\mathbf{x}_B)$

Decision boundary? $y_k = y_i$

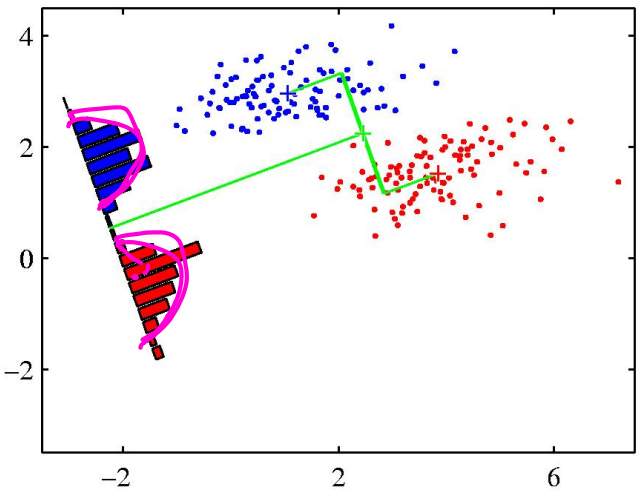
PCA don't work well



picture showing the advantage of Fisher's linear discriminant



When projected onto the line joining the class means, the classes are not well separated.



Fisher chooses a direction that makes the projected classes much tighter, even though their projected means are less far apart.

Math of Fisher's linear discriminants

What linear transformation is best for discrimination?

$$y = \mathbf{w}^T \mathbf{x}$$

The projection onto the vector separating the class means seems sensible:

$$\mathbf{w} \propto \mathbf{m}_2 - \mathbf{m}_1$$

But we also want small variance within each class:

$$s_1^2 = \sum_{n \varepsilon C_1} (y_n - m_1)$$

$$s_2^2 = \sum_{n \varepsilon C_2} (y_n - m_2)$$

Fisher's objective function is:

$$J(\mathbf{w}) = \frac{(m_2 - m_1)^2}{s_1^2 + s_2^2}$$ ← between ← within

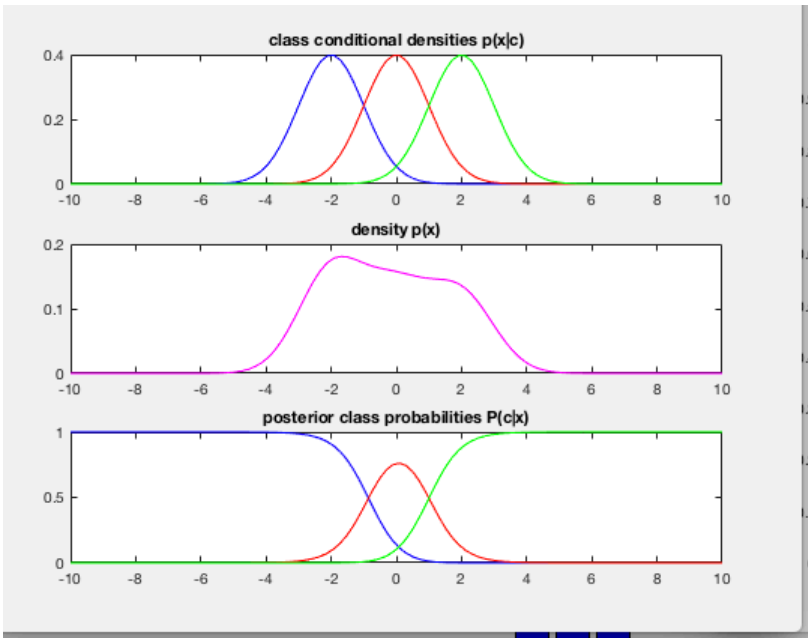
$$J(\mathbf{w}) = \frac{(m_2 - m_1)^2}{s_1^2 + s_2^2} = \frac{\mathbf{w}^T \mathbf{S}_B \mathbf{w}}{\mathbf{w}^T \mathbf{S}_W \mathbf{w}}$$

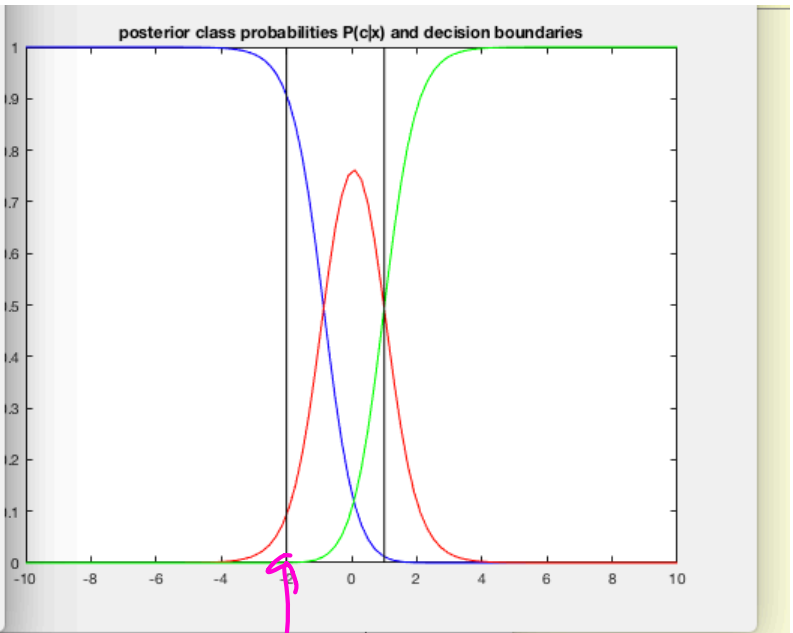
$$\mathbf{S}_B = (\mathbf{m}_2 - \mathbf{m}_1)(\mathbf{m}_2 - \mathbf{m}_1)^T$$

$$\mathbf{S}_W = \sum_{n \in C_1} (\mathbf{x}_n - \mathbf{m}_1)(\mathbf{x}_n - \mathbf{m}_1)^T + \sum_{n \in C_2} (\mathbf{x}_n - \mathbf{m}_2)(\mathbf{x}_n - \mathbf{m}_2)^T$$

$$\textit{Optimal solution}: \quad \mathbf{w} \propto \mathbf{S}_W^{-1} (\mathbf{m}_2 - \mathbf{m}_1)$$

We have done probabilistic classification!





Probabilistic Models for Discrimination (Bishop p 196)

Use a generative model of the input vectors for each class, see which model makes a input vector most probable.

The posterior probability of class 1 is:

$$p(C_1 \mid \mathbf{x}) = \frac{p(C_1)p(\mathbf{x} \mid C_1)}{p(C_1)p(\mathbf{x} \mid C_1) + p(C_0)p(\mathbf{x} \mid C_0)} = \frac{1}{1+e^{-z}}$$

$$\textit{where} \quad z = \ln \frac{p(C_1)p(\mathbf{x} \mid C_1)}{p(C_0)p(\mathbf{x} \mid C_0)} = \ln \frac{p(C_1 \mid \mathbf{x})}{1 - p(C_1 \mid \mathbf{x})}$$

z is called the **logit** and is given by the **log odds**

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

softmax $p(C_n \mid x) = \frac{e^{-z_n}}{\sum e^{-z_n}}$

$z_n = w_n^T x + b_n$

An example for continuous inputs

Assume input vectors for each class are Gaussian, all classes have the same covariance matrix.

normalizing constant

inverse covariance matrix

$$p(\mathbf{x} \mid C_k) = a \exp\left\{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}_k)\right\}$$

For two classes, C_1 and C_0, the posterior is a logistic:

$$p(C_1 \mid \mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + w_0)$$

$$\mathbf{w} = \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)$$

$$w_0 = -\frac{1}{2}\boldsymbol{\mu}_1^T\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_1 + \frac{1}{2}\boldsymbol{\mu}_0^T\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_0 + \ln\frac{p(C_1)}{p(C_0)}$$

See lecture 2

The role of the inverse covariance matrix

If the Gaussian is spherical no need to worry about the covariance matrix.

So, start by transforming the data space to make the Gaussian spherical

- This is called "**whitening"** the data.
- It pre-multiplies by the matrix **square root of the inverse covariance matrix**.

In transformed space, the weight vector is the difference between transformed means.

$$\mathbf{w} = \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)$$

gives the same value

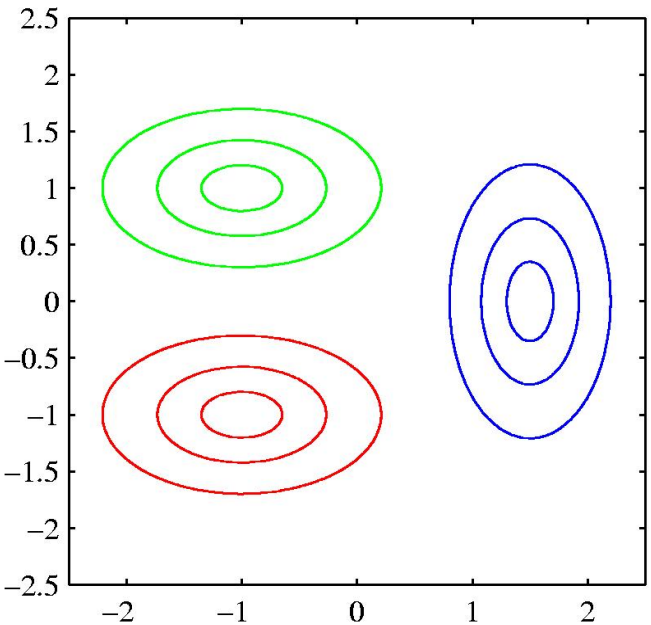
for $\mathbf{w}^T\mathbf{x}$ *as*:

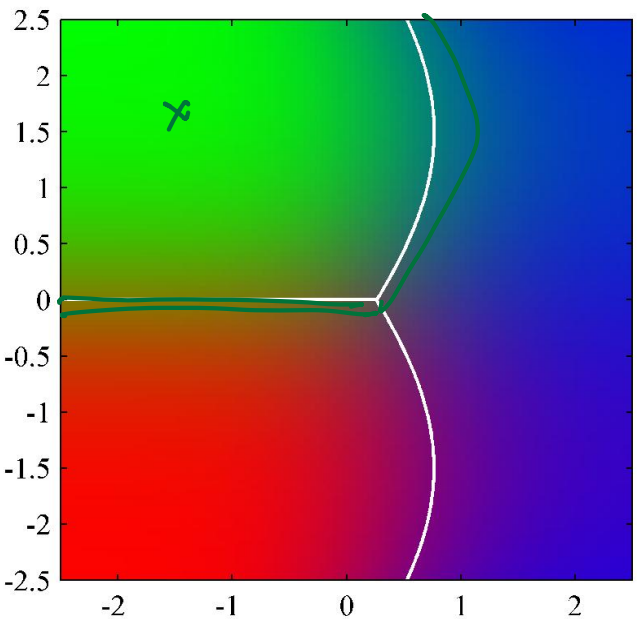
$$\mathbf{w}_{aff} = \boldsymbol{\Sigma}^{-\frac{1}{2}}\boldsymbol{\mu}_1 - \boldsymbol{\Sigma}^{-\frac{1}{2}}\boldsymbol{\mu}_0$$

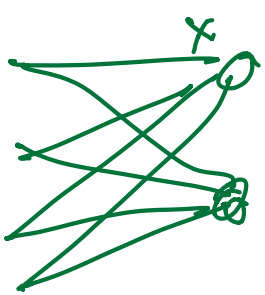
and $\mathbf{x}_{aff} = \boldsymbol{\Sigma}^{-\frac{1}{2}}\mathbf{x}$

gives for $\mathbf{w}_{aff}^T\mathbf{x}_{aff}$

Posterior when covariance matrices are different for each class (Bishop Fig)



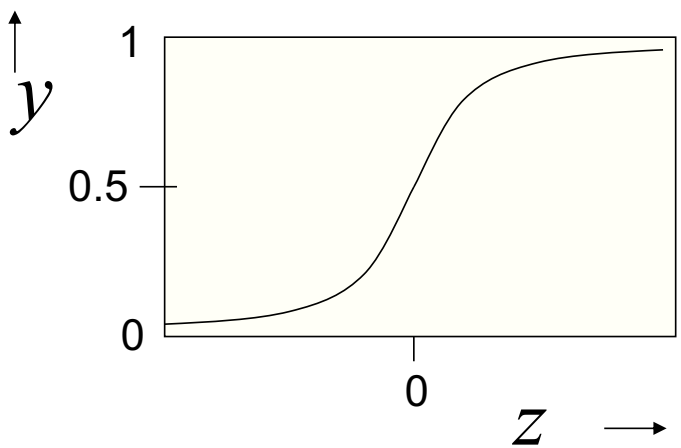




The decision surface is planar when the covariance matrices are the same and quadratic when not.

The logistic function

The output is a smooth function of the inputs and the weights.



$$z = \mathbf{w}^T \mathbf{x} + w_0$$

$$y = \sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\frac{\partial z}{\partial w_i} = x_i \qquad \frac{\partial z}{\partial x_i} = w_i$$

$$\frac{dy}{dz} = y\,(1 - y)$$

Its odd to express it in terms of y.

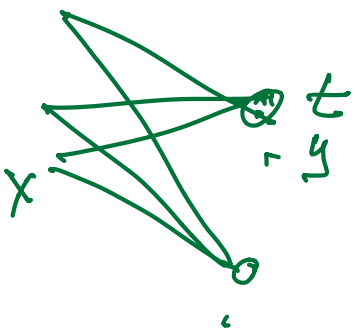
The natural error function for the logistic

Fitting logistic model using maximum likelihood, requires minimizing the negative log probability of the correct answer summed over the training set.

$$E = -\sum_{n=1}^{N} \ln p(t_n \mid y_n)$$

$$= -\sum_{n=1}^{N} t_n \ln y_n + (1 - t_n) \ln (1 - y_n)$$

if t =1 if t =0



error derivative on training case n

$$\frac{\partial E_n}{\partial y_n} = -\frac{t_n}{y_n} + \frac{1 - t_n}{1 - y_n}$$

$$= \frac{y_n - t_n}{y_n (1 - y_n)}$$

$$\text{Ber}(x \mid \mu) = \mu^x (1 - \mu)^{1-x}$$

Logistic regression (Bishop 205)

$p(C_1|\boldsymbol{x}) = \sigma(\boldsymbol{w}^T\boldsymbol{x})$

Observations $\{x_n, t_n\}$ $t_n \in [0, 1]$

Likelihood

$y = \sigma(\boldsymbol{w}^T\boldsymbol{x})$

$p(y|\boldsymbol{x}, \boldsymbol{w}) =$ $Bern(y, \mu) = \mu^y(1-\mu)^{(1-\mu)}$

$p(T|\boldsymbol{x}, \boldsymbol{w}) =$ $\prod_n^N y_n^{t_n}(1-y_n)^{1-t_n}$

Log-likelihood

$E_{\boldsymbol{w}} = -ln(p(T|\boldsymbol{x}, \boldsymbol{w}))=$ $-\sum_n^N (t_n \ln y_n + (1-t_n)\ln(1-y_n))$

Minimize –log like

Derivative

$\nabla_w E_w = -\sum_n^N \left[t_n \frac{1}{y_n} + \frac{1-t_n}{1-y_n}\cdot(-1)\right] y_n(1-y_n)\, x_n$

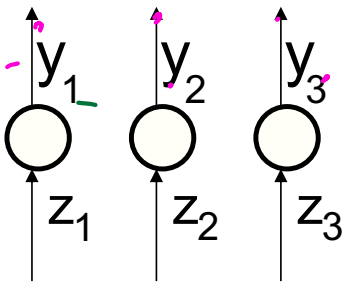
$= \sum_n^N (t_n - y_n)\, x_n$

$w_{i+1} = w_i - \eta \nabla E_w$

Cross-entropy or "**softmax**" function for multi-class classification

The output units use a non-local non-linearity:

$$y_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$



output units

$$\frac{\partial y_i}{\partial z_i} = y_i\,(1 - y_i)$$

The natural cost function is the negative log prob of the right answer

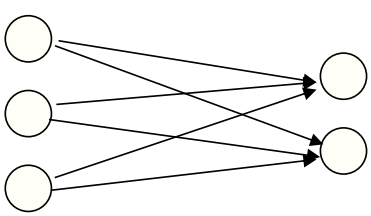
target value

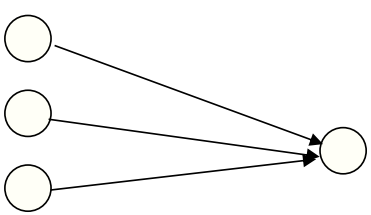
$$E = -\sum_j t_j \ln y_j$$

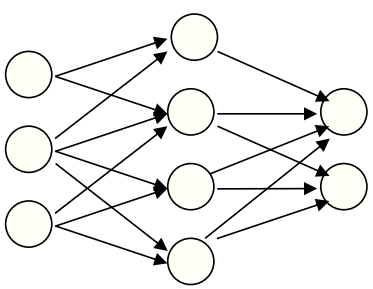
$$\frac{\partial E}{\partial z_i} = \sum_j \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_i} = y_i - t_i$$

$$p(T|W) \prod_n \prod_k p(C_k|z_n)^{t_{nk}} \left[1 - p(C_k|z_n)\right]^{1-t_{nk}}$$

Lecture 8: Backpropagation







Number of parameters

$\boldsymbol{t} = \boldsymbol{w}^T \boldsymbol{x}$, N measurement, M parameters

How large a **w** can we determine?

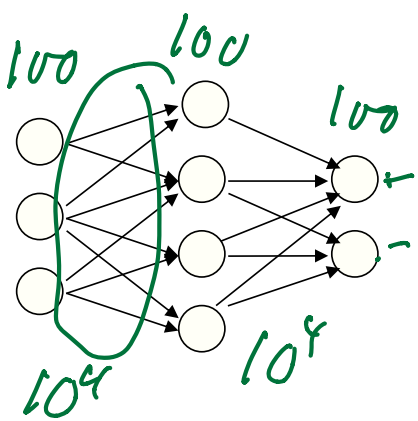
$\boldsymbol{t} = \varphi(\boldsymbol{w}, \boldsymbol{x})$

How large a **w** can we determine?

Consider a neural network, with one hidden layer, each layer having N=M=100 nodes

How large is **W**?

How many observations is needed to estimate **W**?



Why we need backpropagation

Networks without hidden units are very limited in the input-output mappings they can model.

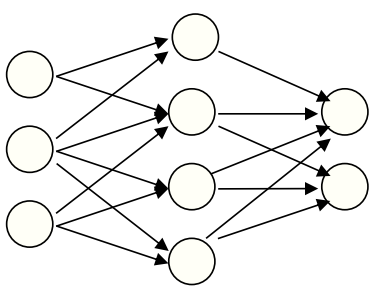
More layers of linear units do not help. Its still linear.

Fixed output non-linearities are not enough

We need multiple layers of adaptive non-linear hidden units, giving a universal approximator. But how to train such nets?

We need an efficient way of adapting all the weights, not just the last layer. Learning the weights going into hidden units is equivalent to learning features.

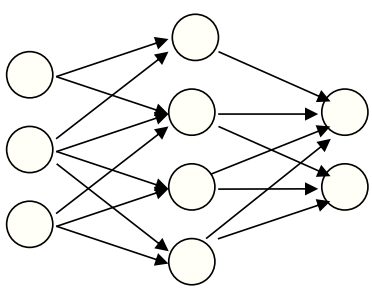
Nobody is telling us directly what hidden units should do.



The idea behind backpropagation

Don't know what the hidden units should be, but we can compute how fast the error changes as we change a hidden activity.

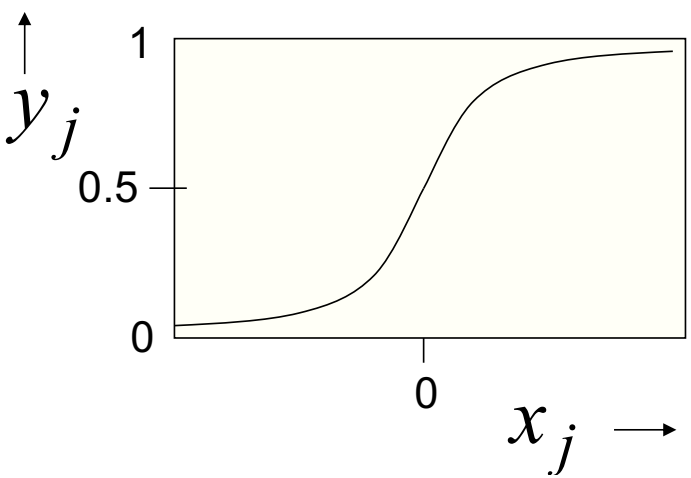
- Instead of using desired activities to train the hidden units, use error derivatives w.r.t. hidden activities.
- Each hidden activity affect many output units and have many separate effects on the error.
- Error derivatives for all the hidden units is computed efficiently.
- Once we have the error derivatives for the hidden activities, its easy to get the error derivatives for the weights going into a hidden unit.



Non-linear neurons with smooth derivatives

For backpropagation, we need neurons that have well-behaved derivatives.

- Typically they use the logistic function
- The output is a smooth function of inputs and weights.



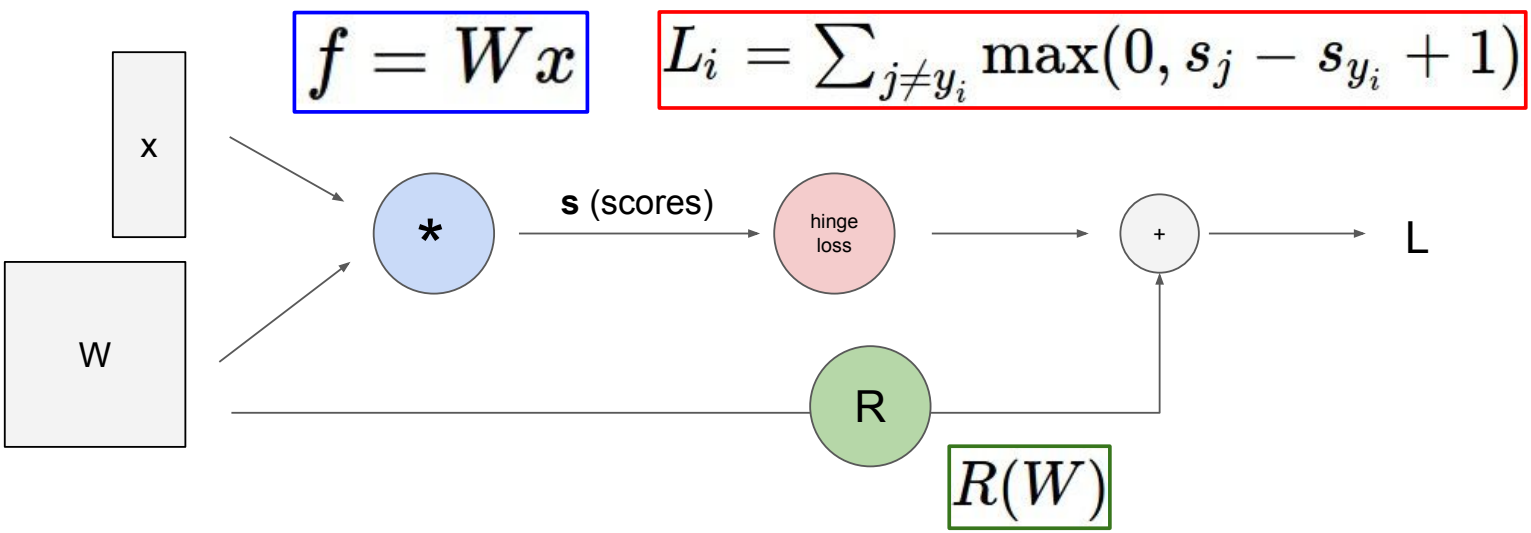
$$x_j = b_j + \sum_i y_i w_{ij}$$

$$y_j = \frac{1}{1 + e^{-x_j}}$$

$$\frac{\partial x_j}{\partial w_{ij}} = y_i \qquad \frac{\partial x_j}{\partial y_i} = w_{ij}$$

$$\frac{dy_j}{dx_j} = y_j\,(1 - y_j)$$

Computational graphs



CNN lecture 4 explain Backpropagation simple

Backpropagation: a simple example

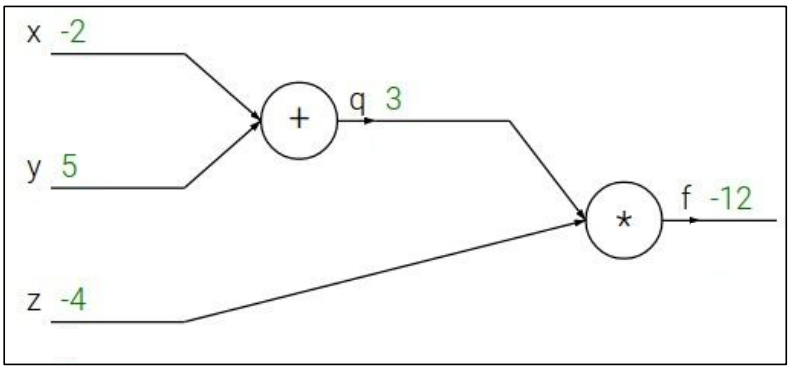
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backpropagation: a simple example

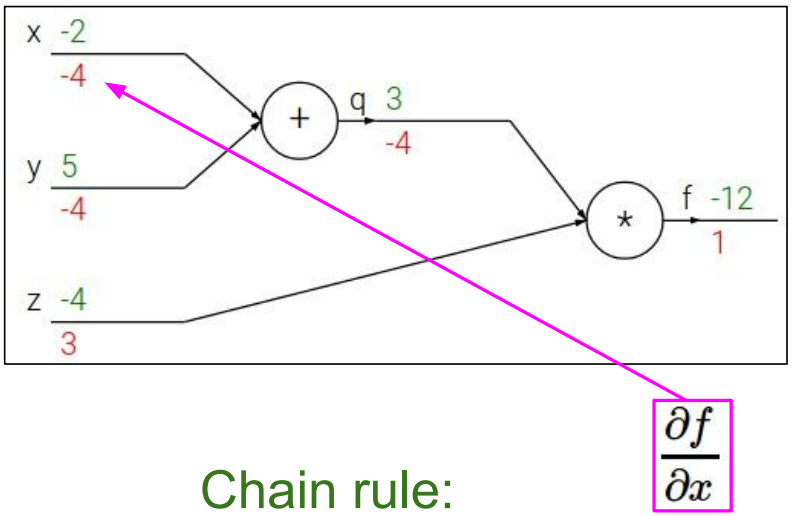
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



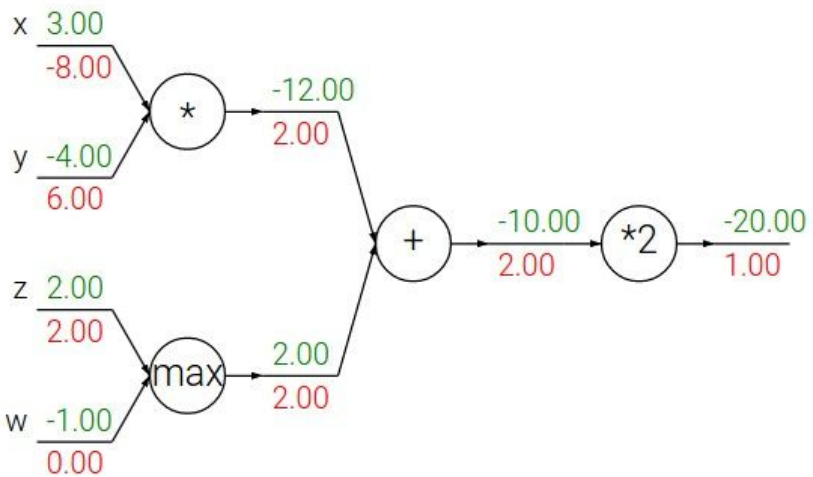
Chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$$

Patterns in backward flow

add gate: gradient distributor

Q: What is a **max** gate?

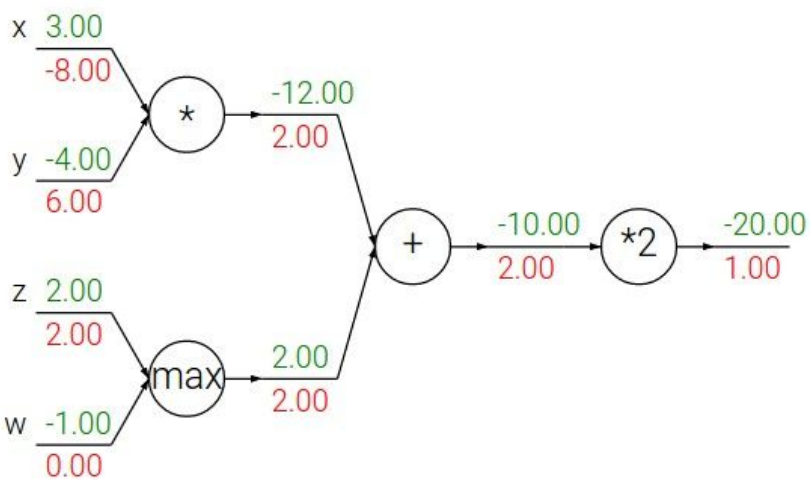


Patterns in backward flow

add gate: gradient distributor

max gate: gradient router

mul gate: gradient switcher



Bernoulli distribution

Random variable $x \in \{0,1\}$

Coin flipping: heads=1, tails=0

$$p(x=1|\mu) = \mu$$

Bernoulli Distribution

$$\begin{aligned} \mathrm{Bern}(x|\mu) &= \mu^x(1-\mu)^{1-x} \\ \mathbb{E}[x] &= \mu \\ \mathrm{var}[x] &= \mu(1-\mu) \end{aligned}$$

ML for Bernoulli

Given: $\mathcal{D} = \{x_1, \ldots, x_N\}$, m heads (1), $N-m$ tails (0)

$$p(\mathcal{D}|\mu) = \prod_{n=1}^{N} p(x_n|\mu) = \prod_{n=1}^{N} \mu^{x_n}(1-\mu)^{1-x_n}$$

$$\ln p(\mathcal{D}|\mu) = \sum_{n=1}^{N} \ln p(x_n|\mu) = \sum_{n=1}^{N} \{x_n \ln \mu + (1-x_n)\ln(1-\mu)\}$$

$$\mu_{\mathrm{ML}} = \frac{1}{N}\sum_{n=1}^{N} x_n = \frac{m}{N}$$

Maximum Likelihood and Least Squares (from lecture 3)

Computing the gradient and setting it to zero yields

$$\nabla_{\mathbf{w}} \ln p(\mathbf{t}|\mathbf{w}, \beta) = \beta \sum_{n=1}^{N} \left\{ t_n - \mathbf{w}^{\mathrm{T}} \boldsymbol{\phi}(\mathbf{x}_n) \right\} \boldsymbol{\phi}(\mathbf{x}_n)^{\mathrm{T}} = \mathbf{0}.$$

Solving for w,

$$\mathbf{w}_{\mathrm{ML}} = \left(\boldsymbol{\Phi}^{\mathrm{T}} \boldsymbol{\Phi} \right)^{-1} \boldsymbol{\Phi}^{\mathrm{T}} \mathbf{t}$$

The Moore-Penrose pseudo-inverse, $\boldsymbol{\Phi}^{\dagger}$.

where

$$\boldsymbol{\Phi} = \begin{pmatrix} \phi_0(\mathbf{x}_1) & \phi_1(\mathbf{x}_1) & \cdots & \phi_{M-1}(\mathbf{x}_1) \\ \phi_0(\mathbf{x}_2) & \phi_1(\mathbf{x}_2) & \cdots & \phi_{M-1}(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(\mathbf{x}_N) & \phi_1(\mathbf{x}_N) & \cdots & \phi_{M-1}(\mathbf{x}_N) \end{pmatrix}.$$

LSQ for classification

Each class $\mathcal{C}_k$ is described by its own linear model so that

$$y_k(\mathbf{x}) = \mathbf{w}_k^{\mathrm{T}}\mathbf{x} + w_{k0} \tag{4.13}$$

where $k = 1, \ldots, K$. We can conveniently group these together using vector notation so that

$$\mathbf{y}(\mathbf{x}) = \widetilde{\mathbf{W}}^{\mathrm{T}}\widetilde{\mathbf{x}} \tag{4.14}$$

Consider a training set $\{\boldsymbol{x}_n, \boldsymbol{t}_n\}, n = 1 \ldots N$
Define **X** and **T**

LSQ solution:

$$\widetilde{\mathbf{W}} = (\widetilde{\mathbf{X}}^{\mathrm{T}}\widetilde{\mathbf{X}})^{-1}\widetilde{\mathbf{X}}^{\mathrm{T}}\mathbf{T} = \widetilde{\mathbf{X}}^{\dagger}\mathbf{T} \tag{4.16}$$

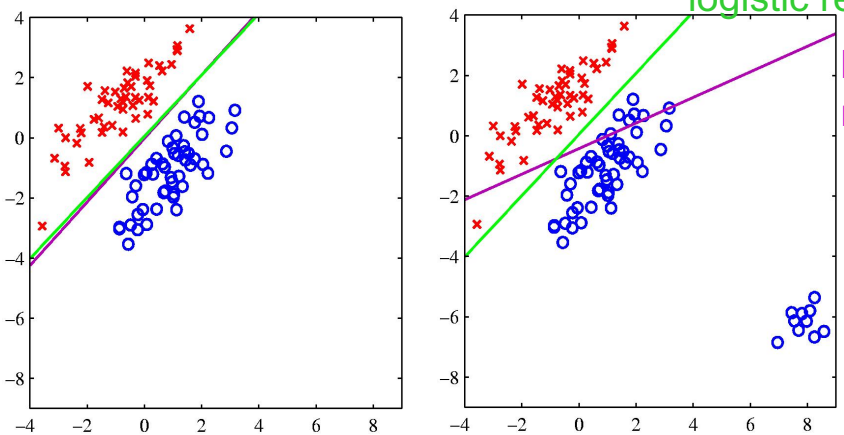
And prediction

$$\mathbf{y}(\mathbf{x}) = \widetilde{\mathbf{W}}^{\mathrm{T}}\widetilde{\mathbf{x}} = \mathbf{T}^{\mathrm{T}}\left(\widetilde{\mathbf{X}}^{\dagger}\right)^{\mathrm{T}}\widetilde{\mathbf{x}}. \tag{4.17}$$

Using “least squares” for classification

It does not work as well as better methods, but it is easy:

It reduces classification to least squares regression.



LSQ solution:

$$\widetilde{\mathbf{W}} = (\widetilde{\mathbf{X}}^{\mathrm{T}}\widetilde{\mathbf{X}})^{-1}\widetilde{\mathbf{X}}^{\mathrm{T}}\mathbf{T} = \widetilde{\mathbf{X}}^{\dagger}\mathbf{T} \qquad (4.16)$$

And prediction

$$\mathbf{y}(\mathbf{x}) = \widetilde{\mathbf{W}}^{\mathrm{T}}\widetilde{\mathbf{x}} = \mathbf{T}^{\mathrm{T}}\left(\widetilde{\mathbf{X}}^{\dagger}\right)^{\mathrm{T}}\widetilde{\mathbf{x}}. \qquad (4.17)$$