COMP 6321
Machine Learning

Instructor: Adam Krzyżak
Email: krzyzak@cs.concordia.ca

Lecture 3: Classification. Generative and Discriminative Learning. Logistic regression. Feed-forward Neural Networks

- Classification tasks
- Error functions for classification
- Generative vs. discriminative learning
- Naive Bayes
- Gaussian Discriminant analysis
- Discriminative learning: cross-entropy error function
- Logistic regression
- Feed-forward neural networks
- Backpropagation

Recall: Classification problems

- Given a data set $\langle \mathbf{x}_i, y_i \rangle$, where y_i *are discrete*, find a hypothesis which "best fits" the data
- If $y_i \in \{0, 1\}$, this is *binary classification* (very useful special case)
- If y_i can take more than two values, the problem is called *multi-class classification*
- Multi-class versions of most binary classification algorithms can be developed in a fairly straightforward way

Example: Text classification

- A very important practical problem, occurring in many different applications: information retrieval, spam filtering, news filtering, building web directories etc.
- A simplified problem description:
  - Given: a collection of documents, classified as “interesting” or “not interesting” by people
  - Goal: learn a classifier that can look at the text of a new document and provide a label for it, without human intervention
- How do we represent the data (documents)?

A simple data representation

- Consider all the possible “significant” words that can occur in the documents (words in the English dictionary, proper names, abbreviations)
- Typically, words that appear in all documents (called *stopwords*) are not considered (prepositions, common verbs like ”to be”, ”to do”...)
- In another preprocessing step, words are mapped to their root (process called *stemming*)

  E.g. learn, learned, learning are all represented by the root “learn”
- For each root, introduce a corresponding *binary feature*, specifying whether the word is present or not in the document.

Example

"Machine learning is fun" $\Longrightarrow$

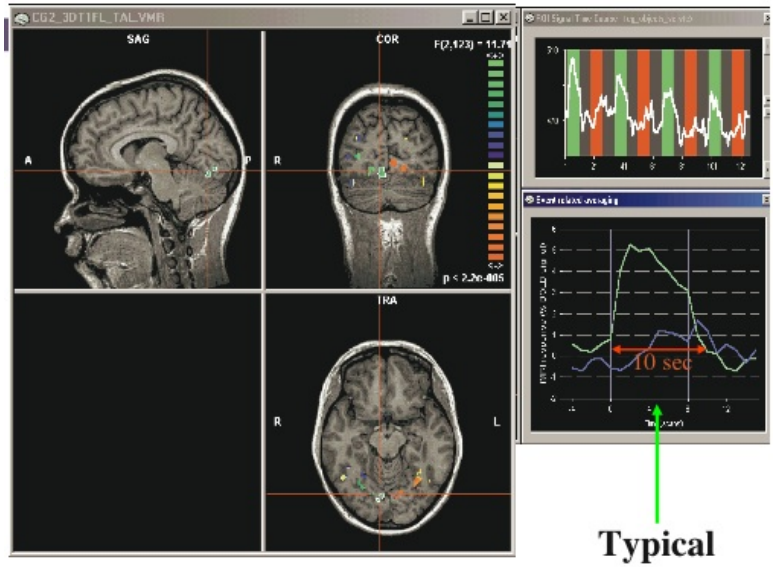
a	0
aardvark	0
⋮	⋮
fun	1
funel	0
⋮	⋮
learn	1
⋮	⋮
machine	1
⋮	⋮
zebra	0

What is special about this task?

- Lots of features! ≈ 100000 for any reasonable domain
- The feature vector is very sparse (a lot of 0 entries)
- It is difficult to get labeled data!

  This process is done by people, hence is very time consuming and tedious

Example: Mind reading (Mitchell et al., 2008)

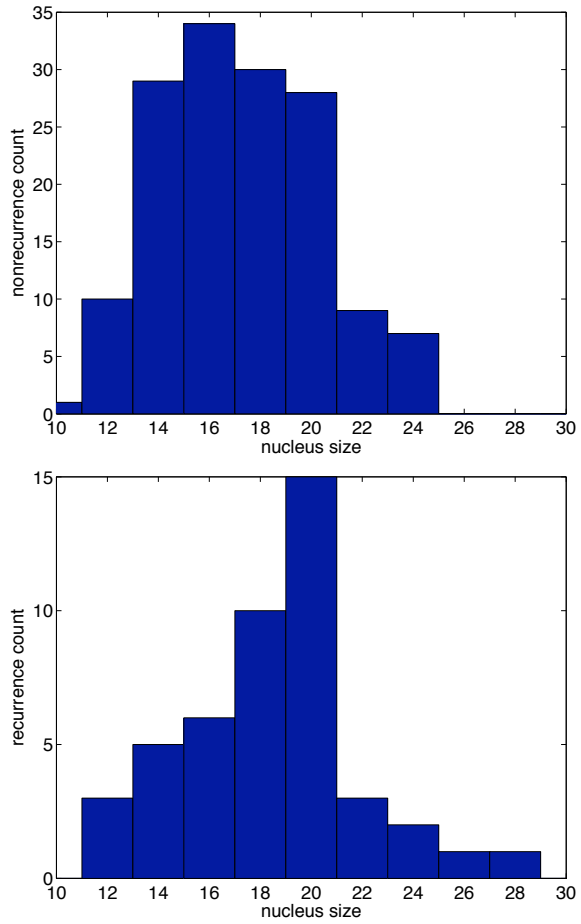


- Given MRI scans, identify what the person is thinking about
  - Roughly 15,000 voxels/image (1mm resolution)
  - 2 images/sec.
- E.g., people words vs. animal words

Classifier learning algorithms

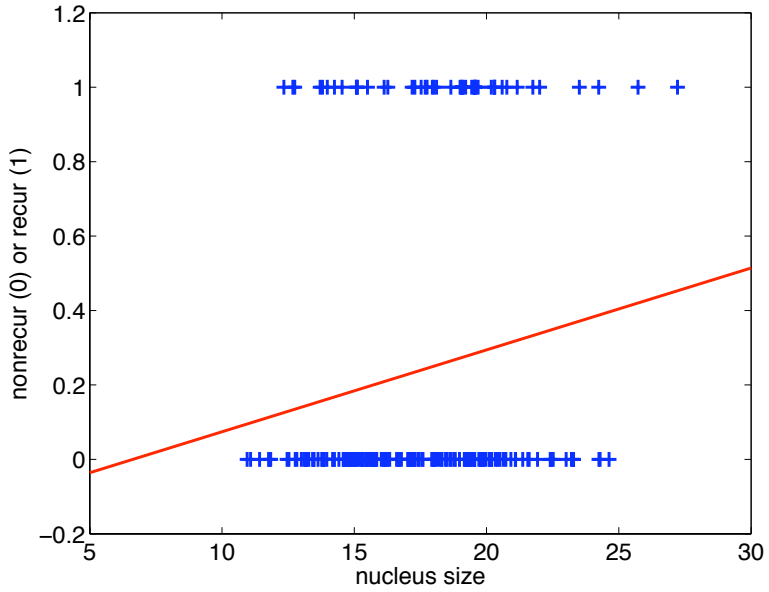
- What is a good error function for classification?
- What hypothesis classes can we use?
- What algorithms are useful for searching those hypotheses classes?

Example: Given “nucleus size” predict cancer recurrrence



Example: Solution by linear regression

- Univariate real input: nucleus size
- Output coding: non-recurrence = 0, recurrence = 1
- Sum squared error minimized by the red line



Linear regression for classification

- The predictor shows an increasing trend towards recurrence with larger nucleus size, as expected.
- Output *cannot be directly interpreted* as a class prediction.
- Thresholding output (e.g., at 0.5) could be used to predict 0 or 1. (In this case, prediction would be 0 except for extremely large nucleus size.)
- Output could be interpreted as probability. (Except that probabilities above 1 and below 0 may be output.)
- Both are somewhat artificial, so we'd like a way of learning that is more suited for the problem.

Probabilistic view

- Suppose we have two possible classes: $y \in \{0, 1\}$.
- What is the probability of a given input $\mathbf{x}$ to have class $y = 1$?
- Bayes Rule:

$$\begin{aligned} P(y=1|\mathbf{x}) &= \frac{P(\mathbf{x}, y=1)}{P(\mathbf{x})} = \frac{P(\mathbf{x}|y=1)P(y=1)}{P(\mathbf{x}|y=1)P(y=1) + P(\mathbf{x}|y=0)P(y=0)} \\ &= \frac{1}{1 + \frac{P(\mathbf{x}|y=0)P(y=0)}{P(\mathbf{x}|y=1)P(y=1)}} = \frac{1}{1+\exp(-a)} = \sigma(a) \end{aligned}$$

where

$$a = \ln \frac{P(\mathbf{x}|y=1)P(y=1)}{P(\mathbf{x}|y=0)P(y=0)}$$

- σ is the sigmoid function (also called "squashing") function
- a is the log-odds of the data being class 1 vs. class 0

Modelling for binary classification

$$P(y=1|\mathbf{x}) = \sigma\left(\ln \frac{P(\mathbf{x}|y=1)P(y=1)}{P(\mathbf{x}|y=0)P(y=0)}\right)$$

- One approach is to model $P(y)$ and $P(\mathbf{x}|y)$, then use the formula above for classification
- This is called *generative learning*, because we can actually use the model to generate (i.e. fantasize) data
- Another idea is to model directly $P(y|\mathbf{x})$
- This is called *discriminative learning*, because we only care about discriminating (i.e. separating) examples of the two classes.
- We will now consider algorithms of both types

Generative learning for document classification

- We can compute $P(y)$ by counting the number of interesting and uninteresting documents we have
- How do we compute $P(\mathbf{x}|y)$?
- Assuming about 100000 words, and not too many documents, this is hopeless!

  Most possible combinations of words will not appear in the data at all...
- Hence, we need to make some extra assumptions.

Naive Bayes assumption

- Suppose the features x_i are discrete
- Assume the x_i *are conditionally independent given* y.
- In other words, assume that:

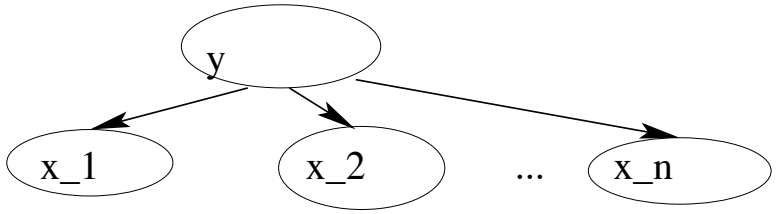
$$P(x_i|y) = P(x_i|y, x_j), \forall i, j$$

- Then, for any input vector $\mathbf{x}$, we have:

$$\begin{aligned} P(\mathbf{x}|y) = P(x_1, x_2, \ldots x_n|y) &= P(x_1|y)P(x_2|y, x_1) \cdots P(x_n|y, x_1, \ldots x_{n-1}) \\ &= P(x_1|y)P(x_2|y) \ldots P(x_n|y) \end{aligned}$$

- For binary features, instead of $O(2^n)$ numbers to describe a model, we only need $O(n)$!

A graphical representation of the naive Bayesian model



- The nodes corresponding to x_i are parameterized by $P(x_i|y)$.
- The node corresponding to y is parameterized by $P(y)$

More generally: Bayesian networks

- If not all conditional independence relations are true, we can introduce new arcs to show what dependencies are there
- At each node, we have the conditional probability distribution of the corresponding variable, given its parents.
- This more general type of graph, annotated with conditional distributions, is called a *Bayesian Network*
- More on this later in the term...

Naive Bayes for binary features

- The parameters of the model are
  $\theta_{i,1} = P(x_i = 1|y = 1)$, $\theta_{i,0} = P(x_i = 1|y = 0)$, $\theta_1 = P(y = 1)$.

- Class label has binomial distribution $P(y) = \theta_1^y(1 - \theta_1)^{1-y}$ and class conditional distributions are multivariate Bernoulli

$$P(\mathbf{x}|y = 1) = \prod_{i=1}^{n} \theta_{i,1}^{x_i}(1 - \theta_{i,1})^{1-x_i}$$

$$P(\mathbf{x}|y = 0) = \prod_{i=1}^{n} \theta_{i,0}^{x_i}(1 - \theta_{i,0})^{1-x_i}$$

- What is the decision surface?

$$\frac{P(y=1|\mathbf{x})}{P(y=0|\mathbf{x})} = \frac{P(y=1)P(\mathbf{x}|y=1)}{P(y=0)P(\mathbf{x}|y=0)}$$
$$= \frac{P(y=1)\prod_{i=1}^{n} P(x_i|y=1)}{P(y=0)\prod_{i=1}^{n} P(x_i|y=0)} = \frac{P(y=1)\prod_{i=1}^{n} \theta_{i,1}^{x_i}(1-\theta_{i,1})^{1-x_i}}{P(y=0)\prod_{i=1}^{n} \theta_{i,0}^{x_i}(1-\theta_{i,0})^{1-x_i}}$$

- Using the log trick, we get:

$$\log \frac{P(y=1|\mathbf{x})}{P(y=0|\mathbf{x})} = \log \frac{P(y=1)}{P(y=0)} + \sum_{i=1}^{n} \log \frac{P(x_i|y=1)}{P(x_i|y=0)}$$

- Note that in the equation above, the x_i would be 1 or 0

Decision boundary of Naive Bayes with binary features

$$
\begin{aligned}
\text{Let: } w_0 &= \log \frac{P(y=1)}{P(y=0)} = \log \frac{\theta_1}{\theta_0} \\
w_{i,1} &= \log \frac{P(x_i=1|y=1)}{P(x_i=1|y=0)} = \log \frac{\theta_{i,1}}{\theta_{i,0}} \\
w_{i,0} &= \log \frac{P(x_i=0|y=1)}{P(x_i=0|y=0)} = \log \frac{(1-\theta_{i,1})}{(1-\theta_{i,0})}
\end{aligned}
$$

We can re-write the decision boundary as:

$$
\log \frac{P(y=1|\mathbf{x})}{P(y=0|\mathbf{x})} = w_0 + \sum_{i=1}^{n} (w_{i,1}x_i + w_{i,0}(1-x_i)) = w_0 + \sum_{i=1}^{n} w_{i,0} + \sum_{i=1}^{n} (w_{i,1} - w_{i,0})x_i
$$

This is a *linear decision boundary*!

Learning the parameters of a Naive Bayes classifier

- We will find the parameters that *maximize the log likelihood of the training data*!
- The likelihood in this case is:

$$L(\theta_1, \theta_{i,1}, \theta_{i,0}) = \prod_{j=1}^{m} P(\mathbf{x}_j, y_j) = \prod_{j=1}^{m} \left[P(y_j) \prod_{i=1}^{n} P(x_{j,i}|y_j) \right]$$

- First, use the log trick:

$$\log L(\theta_1, \theta_{i,1}, \theta_{i,0}) = \sum_{j=1}^{m} \left(\log P(y_j) + \sum_{i=1}^{n} \log P(x_{j,i}|y_j) \right)$$

- Observe that each term in the sum depends on the values of y_j, $\mathbf{x}_j$ that appear in the jth instance. Also note $P(y) = \theta_1^y (1-\theta_1)^{1-y}$

Maximum likelihood parameter estimation

$$
\begin{aligned}
\log L(\theta_1, \theta_{i,1}, \theta_{i,0}) &= \sum_{j=1}^{m} \Big[\log \theta_1^{y_j} (1-\theta_1)^{1-y_j} \\
&+ \sum_{i=1}^{n} \log \theta_{i,1}^{y_j x_{j,i}} (1-\theta_{i,1})^{y_j(1-x_{j,i})} \cdot \theta_{i,0}^{(1-y_j)x_{j,i}} (1-\theta_{i,0})^{(1-y_j)(1-x_{j,i})} \Big] \\
&= \sum_{j=1}^{m} \Big[y_j \log \theta_1 + (1-y_j) \log(1-\theta_1) \\
&+ \sum_{i=1}^{n} y_j \Big(x_{j,i} \log \theta_{i,1} + (1-x_{j,i}) \log(1-\theta_{i,1}) \Big) \\
&+ \sum_{i=1}^{n} (1-y_j) \Big(x_{j,i} \log \theta_{i,0} + (1-x_{j,i}) \log(1-\theta_{i,0}) \Big) \Big]
\end{aligned}
$$

To estimate θ_1, we take the derivative of $\log L$ wrt θ_1 and set it to 0:

$$
\frac{\partial L}{\partial \theta_1} = \sum_{j=1}^{m} \left(\frac{y_j}{\theta_1} + \frac{1-y_j}{1-\theta_1}(-1) \right) = 0
$$

Maximum likelihood parameters estimation for naive Bayes

By solving for θ_1, we get:

$$\theta_1 = \frac{1}{m}\sum_{j=1}^{m} y_j = \frac{\text{number of examples of class 1}}{\text{total number of examples}}$$

Using a similar derivation, we get:

$$\theta_{i,1} = \frac{\text{number of instances for which } x_{j,i} = 1 \text{ and } y_j = 1}{\text{number of instances for which } y_j = 1}$$

$$\theta_{i,0} = \frac{\text{number of instances for which } x_{j,i} = 1 \text{ and } y_j = 0}{\text{number of instances for which } y_j = 0}$$

Text classification revisited

- Consider again the text classification example, where the features x_i correspond to words
- Using the approach above, we can compute probabilities for all the words which appear in the document collection
- But what about words that do not appear?

  They would be assigned zero probability!
- As a result, the probability estimates for documents containing such words would be $0/0$ for both classes, and hence no decision can be made

Laplace smoothing

- Instead of the maximum likelihood estimate:

$$\theta_{i,1} = \frac{\text{number of instances for which } x_{j,i} = 1 \text{ and } y_j = 1}{\text{number of instances for which } y_j = 1}$$

  use:

$$\theta_{i,1} = \frac{(\text{number of instances for which } x_{j,i} = 1 \text{ and } y_j = 1) + 1}{(\text{number of instances for which } y_j = 1) + 2}$$

- Hence, if a word does not appear at all in the documents, it will be assigned prior probability 0.5.
- If a word appears in a lot of documents, this estimate is only slightly different from max. likelihood.
- This is an example of *Bayesian prior* for Naive Bayes

Example: 20 newsgroups

Given 1000 training documents from each group, learn to classify new documents according to which newsgroup they came from

comp.graphics	misc.forsale
comp.os.ms-windows.misc	rec.autos
comp.sys.ibm.pc.hardware	rec.motorcycles
comp.sys.mac.hardware	rec.sport.baseball
comp.windows.x	rec.sport.hockey
alt.atheism	sci.space
soc.religion.christian	sci.crypt
talk.religion.misc	sci.electronics
talk.politics.mideast	sci.med
talk.politics.misc	talk.politics.guns

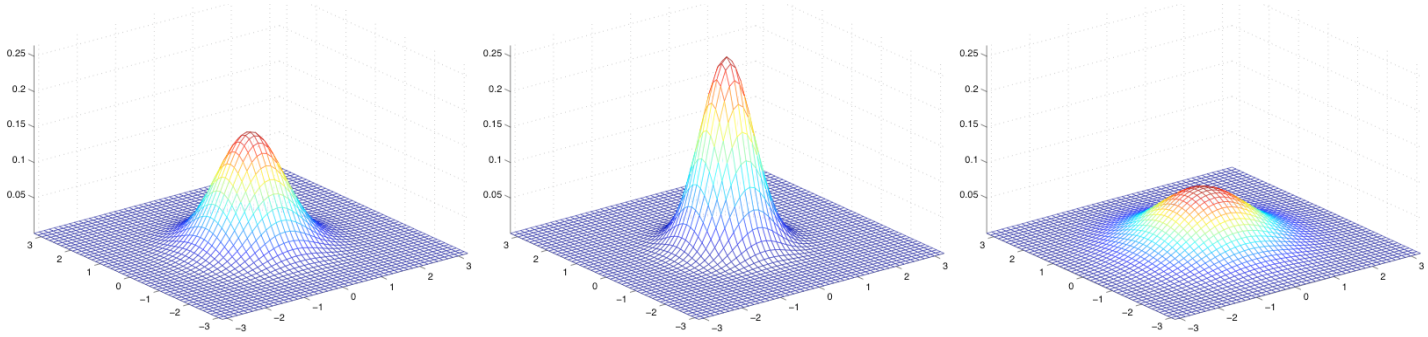
Naive Bayes: 89% classification accuracy - comparable to other state-of-art methods

Gaussian Discriminant Analysis

- This is a generative model for continuous inputs
- $P(y)$ is still assumed to be binomial
- $P(\mathbf{x}|y)$ is assumed to be a multivariate Gaussian (normal distribution), with mean $\mu \in \mathbb{R}^n$ and covariance matrix $\Sigma \in \mathbb{R}^n \times \mathbb{R}^n$.
- If we assume that the covariance matrix is diagonal, then the features are all conditionally independent and the model is also called *Gaussian Naive Bayes*

Example: Diagonal covariance

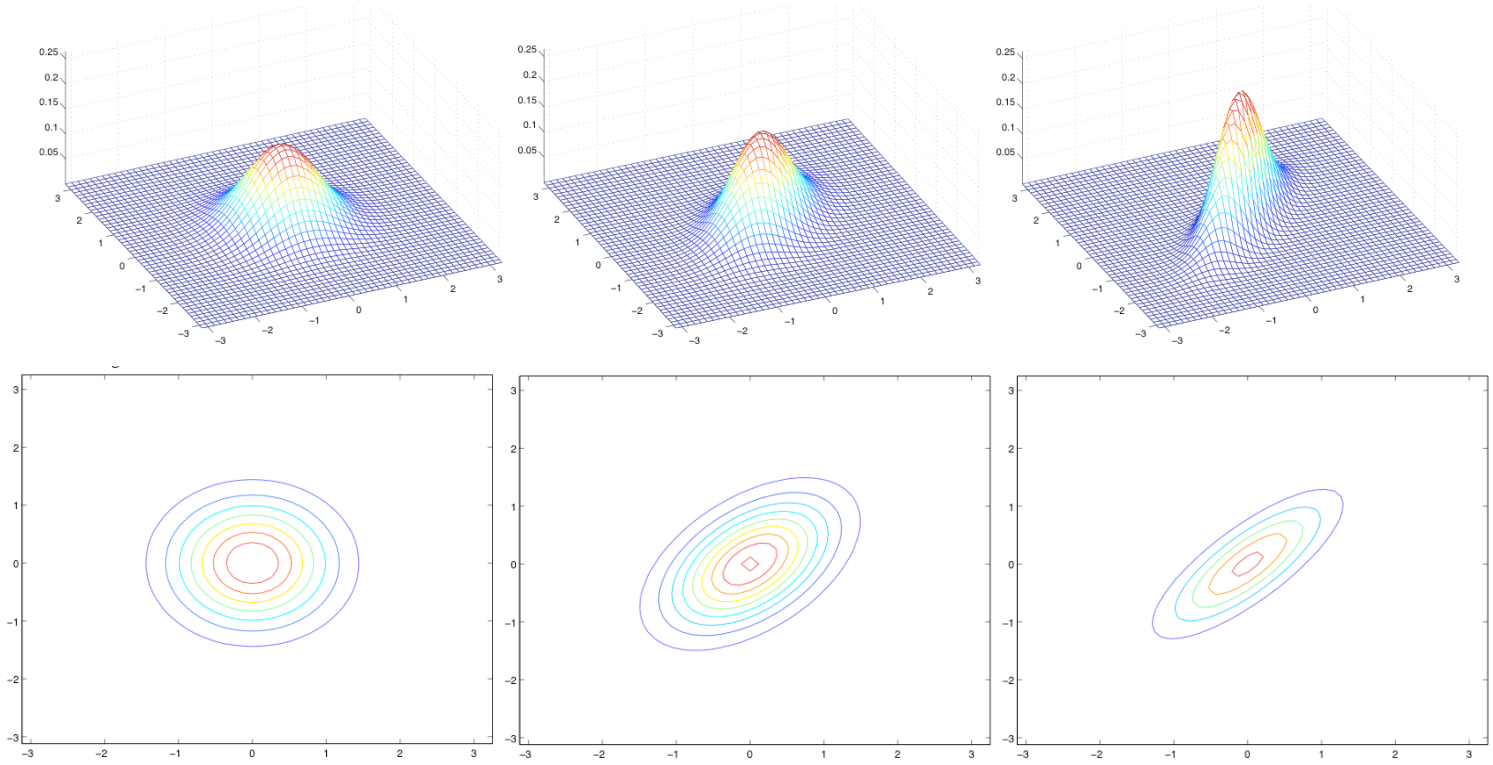
$$\Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \Sigma = \begin{bmatrix} 0.6 & 0 \\ 0 & 0.6 \end{bmatrix} \qquad \Sigma = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$



- The Gaussian bump is centered at the mean, and the contours of equal probability are axis-aligned (hyper)ellipsoids
- If the values of the diagonal elements are the same, the hyper-ellipsoids are “round”, otherwise they will be elongated
- The magnitude of the diagonal elements controls how spread out the bump is

Example

$$\Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \Sigma = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix} \Sigma = \begin{bmatrix} 1 & 0.8 \\ 0.8 & 1 \end{bmatrix}$$



Gaussian Naive Bayes model

- The class label is modeled as $P(y) = \theta^y(1-\theta)^{1-y}$ (binomial, like before)
- The two classes are modeled as:

$$P(\mathbf{x}|y=0) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu_\mathbf{0})^T\Sigma^{-1}(\mathbf{x}-\mu_\mathbf{0})\right)$$

$$P(\mathbf{x}|y=1) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu_\mathbf{1})^T\Sigma^{-1}(\mathbf{x}-\mu_\mathbf{1})\right)$$

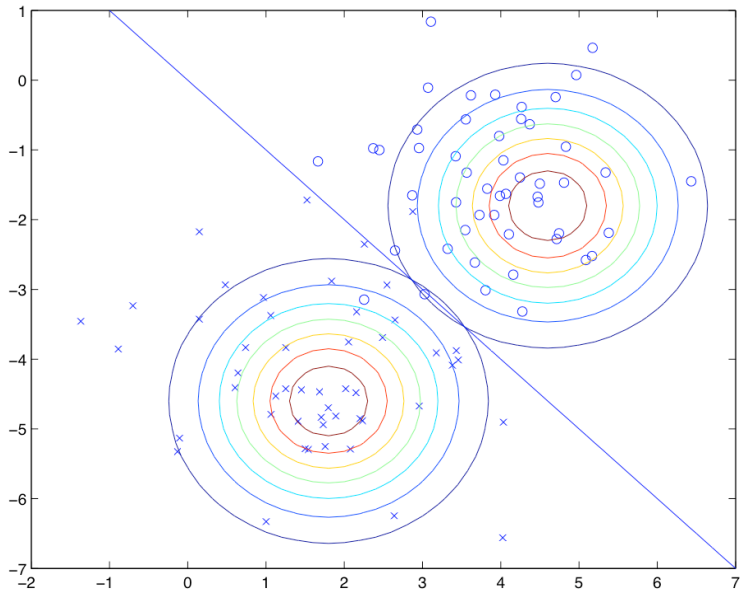
- The parameters to estimate are: θ, $\mu_\mathbf{0}$, $\mu_\mathbf{1}$, Σ
- Note that the covariance is considered the same!

Determining the parameters

- We can write down the likelihood function, like before
- We take the derivatives wrt the parameters and set them to 0
- The parameter θ is just the empirical frequency of class 1
- The means μ_0 and μ_1 are just the empirical means for examples of class 0 and 1 respectively
- The covariance matrix is the empirical estimate from the data:

$$\Sigma = \frac{1}{m}\sum_{i=1}^{m}(\mathbf{x}_i - \mu_{y_i})(\mathbf{x}_i - \mu_{y_i})^T$$

Example



The line is the decision boundary between the classes (on the line, both have equal probability)

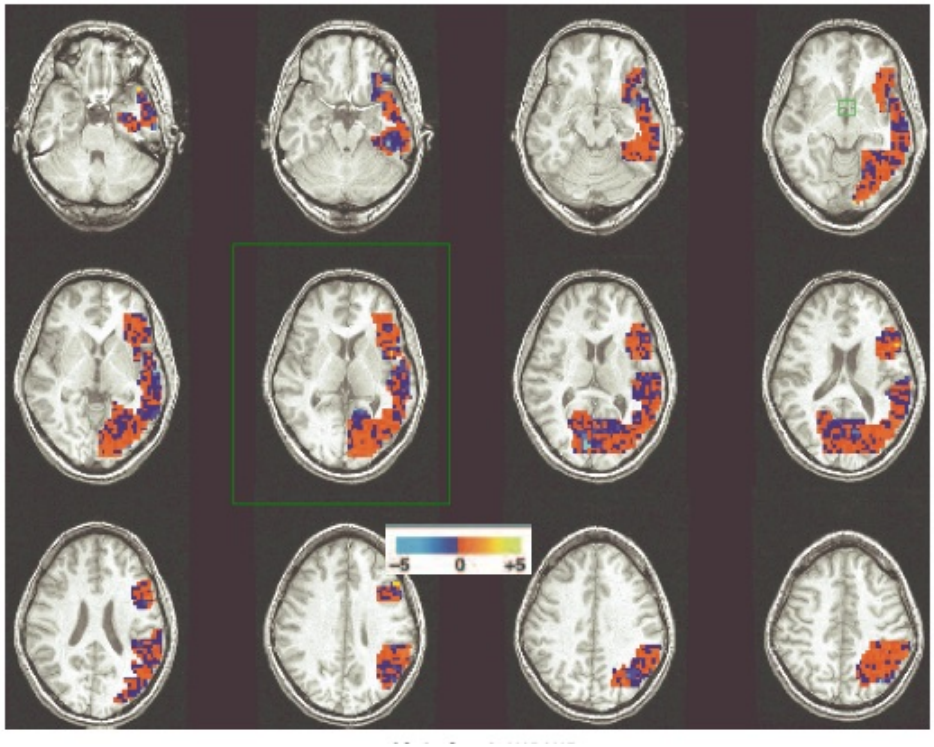
Other variations

- Covariance matrix can be different for the two classes, if we have enough data to estimate it
- Covariance matrix can be restricted to diagonal, or mostly diagonal with few off-diagonal elements, based on prior knowledge.
- The shape of the covariance is influenced both by assumptions about the domain and by the amount of data available.

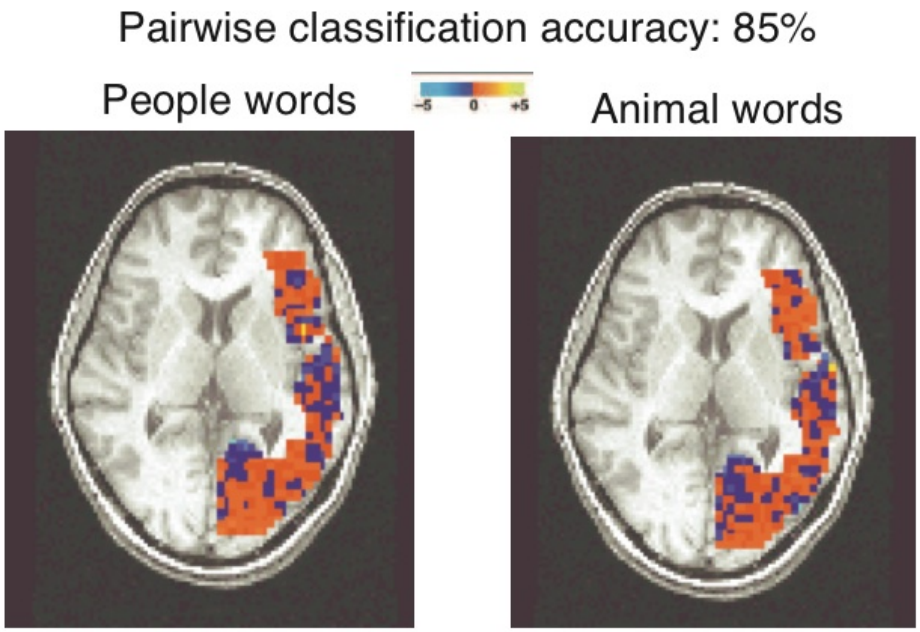
Recall: Generative learning

- Learn a model of $P(\mathbf{x}, y)$ (the joint distribution of the input and output), typically as $P(y)$ and $P(\mathbf{x}|y)$
- This distribution can be used both to classify data and to *generate* data
- Different methods make assumptions about the form of these distributions
  - E.g. Naive Bayes
  - E.g. Gaussian Naive Bayes
  - E.g. Gaussian discriminant analysis

Mind reading revisited: Word models



Mind reading revisited: Average class distributions



Error function

- Maximum likelihood classification assumes that we will find the hypothesis that maximizes the (log) likelihood of the training data:

$$\arg\max_h \log P(\langle \mathbf{x}_i, y_i \rangle_{i=1\ldots m} | h) = \arg\max_h \sum_{i=1}^{n} \log P(\mathbf{x}_i, , y_i) | h)$$

  (using the usual i.i.d. assumption)

- But if we use discriminative learning, we have *no model of the input distribution*
- Instead, we want to maximize the *conditional probability* of the labels, given the inputs:

$$\arg\max_h \sum_{i=1}^{m} \log P(y_i | \mathbf{x}_i, h)$$

The cross-entropy error function

- Suppose we interpret the output of the hypothesis, $h(\mathbf{x}_i)$, as the probability that $y_i = 1$
- Then the log-likelihood of a hypothesis h is:

$$\begin{aligned}\log L(h) &= \sum_{i=1}^{m} \log P(y_i|\mathbf{x}_i, h) = \sum_{i=1}^{m} \begin{cases} \log h(\mathbf{x}_i) & \text{if } y_i = 1 \\ \log(1 - h(\mathbf{x}_i)) & \text{if } y_i = 0 \end{cases} \\ &= \sum_{i=1}^{m} y_i \log h(\mathbf{x}_i) + (1 - y_i) \log(1 - h(\mathbf{x}_i))\end{aligned}$$

- The *cross-entropy error function* is the opposite quantity:

$$J(h) = -\left(\sum_{i=1}^{m} y_i \log h(\mathbf{x}_i) + (1 - y_i) \log(1 - h(\mathbf{x}_i))\right)$$

Logistic regression

- Suppose we represent the hypothesis itself as a logistic function of a linear combination of inputs:

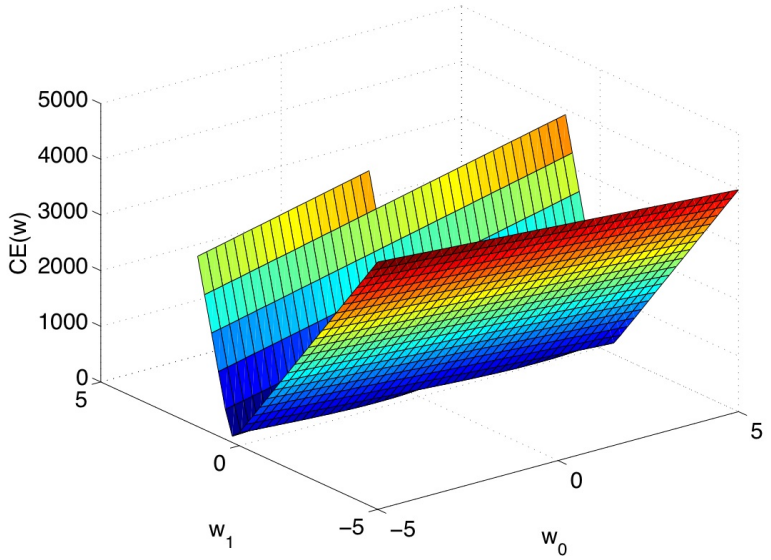
$$h_w(\mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{w}^T\mathbf{x})} = \sigma(\mathbf{w}^T\mathbf{x}).$$

  This is also known as a *sigmoid neuron*
- Suppose we interpret $h(\mathbf{x})$ as $P(y = 1|\mathbf{x})$
- Then the log-odds ratio,

$$\ln\left(\frac{P(y = 1|\mathbf{x})}{P(y = 0|\mathbf{x})}\right) = \mathbf{w}^T\mathbf{x}$$

  which is linear (nice!)
- We can find the optimum weight by minimizing cross-entropy (maximizing the conditional likelihood of the data)

Cross-entropy error surface for logistic function



$$J(h) = -\left(\sum_{i=1}^{m} y_i \log \sigma(\mathbf{w}^T \mathbf{x}_i) + (1 - y_i) \log(1 - \sigma(\mathbf{w}^T \mathbf{x}_i))\right)$$

Nice error surface, unique minimum, but *cannot solve in closed form*

Maximization procedure: Gradient ascent

- First we compute the gradient of $\log L(\mathbf{w})$ wrt $\mathbf{w}$ and use the fact

$$\nabla_w h_w(x) = (1 - h_w(x))h_w(x)x$$

- We get

$$\begin{aligned}
\nabla \log L(\mathbf{w}) &= \sum_i y_i \frac{1}{h_{\mathbf{w}}(\mathbf{x}_i)} h_{\mathbf{w}}(\mathbf{x}_i)(1 - h_{\mathbf{w}}(\mathbf{x}_i))\mathbf{x}_i \\
&\quad + (1 - y_i)\frac{1}{1 - h_{\mathbf{w}}(\mathbf{x}_i)} h_{\mathbf{w}}(\mathbf{x}_i)(1 - h_{\mathbf{w}}(\mathbf{x}_i))\mathbf{x}_i(-1) \\
&= \sum_i \mathbf{x}_i(y_i - y_i h_{\mathbf{w}}(\mathbf{x}_i) - h_{\mathbf{w}}(\mathbf{x}_i) + y_i h_{\mathbf{w}}(\mathbf{x}_i)) \\
&= \sum_i (y_i - h_{\mathbf{w}}(\mathbf{x}_i))\mathbf{x}_i
\end{aligned}$$

Maximization procedure: Gradient ascent

- The update rule (because we maximize) is:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \nabla \log L(\mathbf{w}) = \mathbf{w} + \alpha \sum_{i=1}^{m} (y_i - h_{\mathbf{w}}(\mathbf{x_i}))\mathbf{x_i}$$

  where $\alpha \in (0, 1)$ is a step-size or learning rate parameter
- This is called *logistic regression*

Another algorithm for optimization

- Recall Newton's method for finding the zero of a function $g : \mathbb{R} \to \mathbb{R}$
- At point w^i, approximate the function by a straight line (its tangent)
- Solve the linear equation for where the tangent equals 0, and move the parameter to this point:

$$w^{i+1} = w^i - \frac{g(w^i)}{g'(w^i)}$$

Application to machine learning

- Suppose for simplicity that the error function J has only one parameter
- We want to optimize J, so we can apply Newton's method to find the zeros of $J' = \frac{d}{dw}J$
- We obtain the iteration:

$$w^{i+1} = w^i - \frac{J'(w^i)}{J''(w^i)}$$

- Note that there is *no step size parameter*!
- This is a *second-order method*, because it requires computing the second derivative
- But, if our error function is quadratic, this will find the global optimum in one step!

Second-order methods: Multivariate setting

- If we have an error function J that depends on many variables, we can compute the *Hessian matrix*, which contains the second-order derivatives of J:

$$H_{ij} = \frac{\partial^2 J}{\partial w_i \partial w_j}$$

- The inverse of the Hessian gives the "optimal" learning rates
- The weights are updated as:

$$\mathbf{w} \leftarrow \mathbf{w} - H^{-1} \nabla_{\mathbf{w}} J$$

- This is also called Newton-Raphson method

Which method is better?

- Newton's method usually requires significantly fewer iterations than gradient descent
- Computing the Hessian requires a batch of data, so there is no natural on-line algorithm
- Inverting the Hessian explicitly is expensive, but almost never necessary
- Computing the product of a Hessian with a vector can be done in linear time (Schraudolph, 1994)

Newton-Raphson for logistic regression

- Leads to a nice algorithm called *recursive least squares*
- The Hessian has the form:

$$\mathbf{H} = \mathbf{\Phi}^T \mathbf{R} \mathbf{\Phi}$$

  where $\mathbf{R}$ is the diagonal matrix of $h(\mathbf{x}_i)(1 - h(\mathbf{x}_i))$
- The weight update becomes:

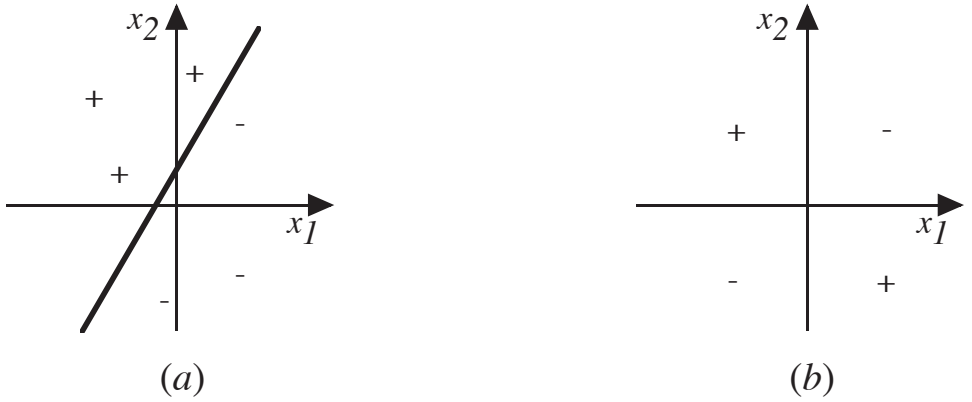
$$\mathbf{w} \leftarrow (\mathbf{\Phi}^T \mathbf{R} \mathbf{\Phi})^{-1} \mathbf{\Phi}^T \mathbf{R} (\mathbf{\Phi}\mathbf{w} - \mathbf{R}^{-1}(\mathbf{\Phi}\mathbf{w} - \mathbf{y}))$$

Logistic vs. Gaussian Discriminant

- Comprehensive study done by Ng and Jordan (2002)
- If the Gaussian assumption is correct, as expected, Gaussian discriminant is better
- If the assumption is violated, Gaussian discriminant suffers from bias (unlike logistic regression)
- In practice, Gaussian discriminant, tends to converge quicker to a less helpful solution
- Note that they optimize *different error functions*!

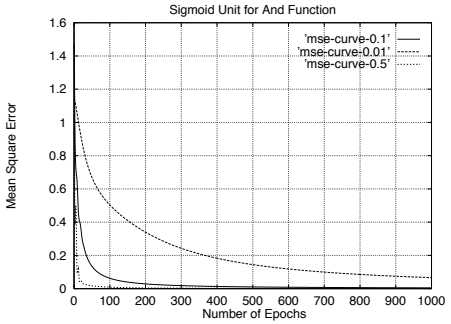
From neurons to networks

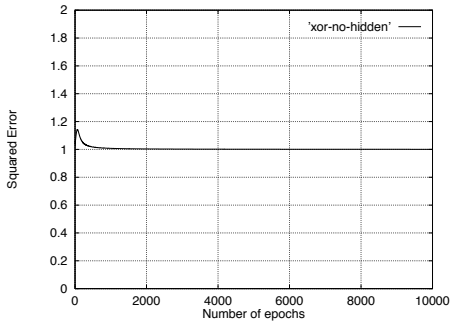
- Logistic regression can be used to learn *linearly separable* problems



- If the instances are not linearly separable, the function cannot be learned correctly (e.g. XOR)
- One solution is to provide fixed, non-linear basis functions instead of inputs (like we did with linear regression)
- Today: learn such non-linear functions from data

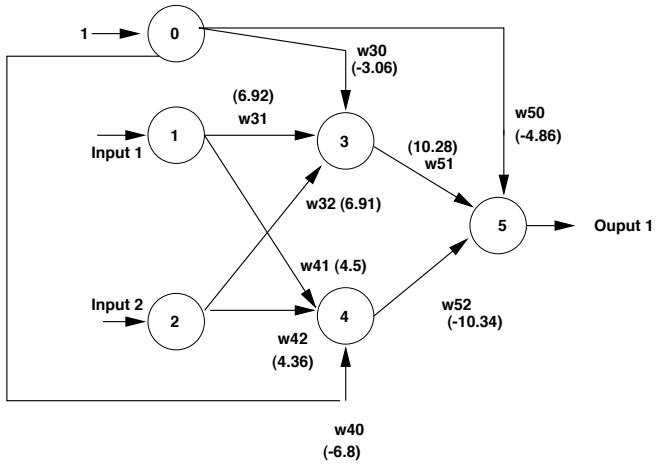
Example: Logical functions of two variables



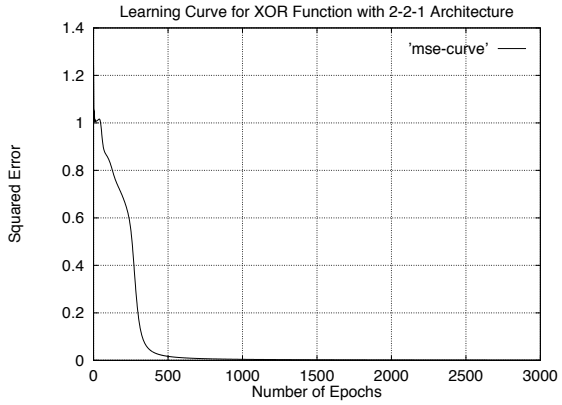


- One sigmoid neuron can learn the AND function (left) but not the XOR function (right)
- In order to learn discrimination in data sets that are not linearly separable, we need *networks of sigmoid units*

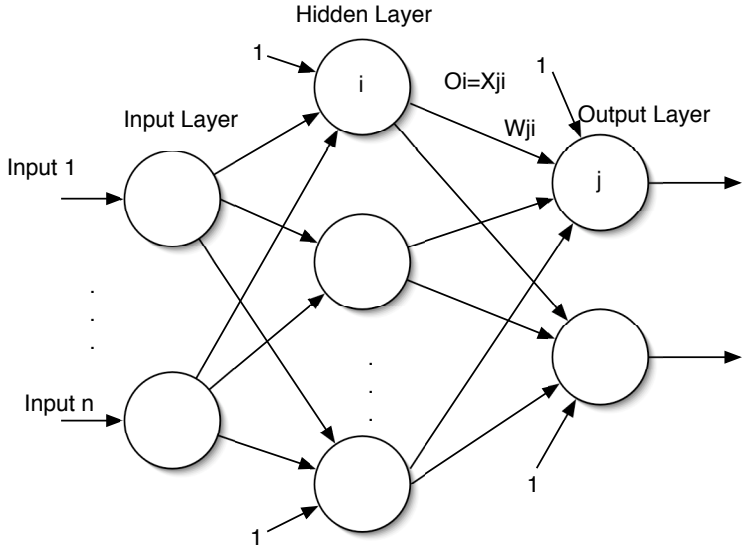
Example: A network representing the XOR function



Input1	Input2	o3	o4	Ouput 1
0	0	0.04	0.001	0.011
0	1	0.98	0.08	0.99
1	0			
1	1			



Feed-forward neural networks

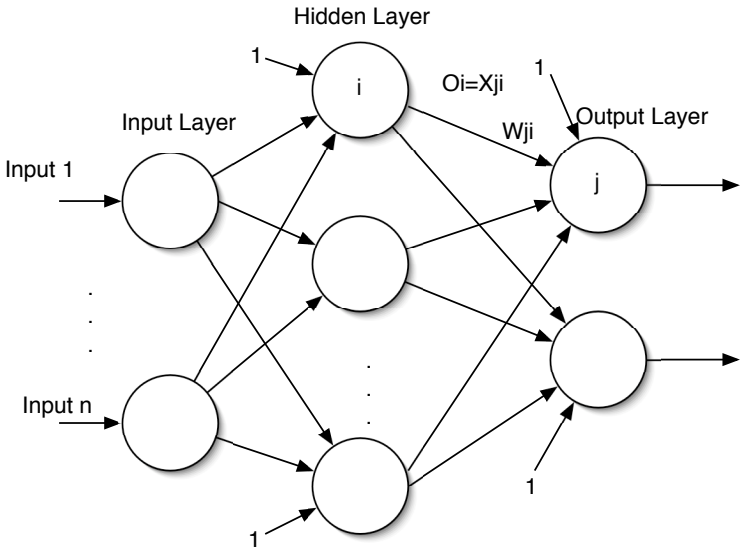


- A collection of units (neurons) with sigmoid or sigmoid-like activations, arranged in *layers*
- Layers 0 is the *input layer*, and its units just copy the inputs (by convention)
- The last layer, K, is called *output layer*, since its units provide the result of the network
- Layers $1, \ldots K-1$ are usually called *hidden layers* (their presence cannot be detected from outside the network)

Why this name?

- In *feed-forward networks* the outputs of units in layer k become inputs for units in layers with index $> k$
- There are no cross-connections between units in the same layer
- There are no backward ("recurrent") connections from layers downstream
- Typically, units in layer k provide input *only* to units in layer $k + 1$
- In *fully connected networks*, all units in layer k are connected to all units in layer $k + 1$

Notation



- $w_{j,i}$ is the weight on the connection from unit i to unit j
- By convention, $x_{j,0} = 1, \forall j$
- The output of unit j, denoted o_j, is computed using a sigmoid: $o_j = \sigma(\mathbf{w}_j^T \mathbf{x}_j)$ where $\mathbf{w}_j$ is the vector of weights on the connections *entering* unit j and $\mathbf{x}_j$ is the vector of inputs to unit j
- By the definition of the connections, $x_{j,i} = o_i$

Computing the output of the network

- Suppose that we want the network to make a prediction for instance $\langle \mathbf{x}, y \rangle$
- In a feed-forward network, this can be done in one *forward pass*:

  For layer $k = 1$ to K

  1. Compute the output of all neurons in layer k:

$$o_j \leftarrow \sigma(\mathbf{w}_j^T \mathbf{x}_j), \forall j \in \text{Layer } k$$

  2. Copy these outputs as inputs to the next layer:

$$x_{j,i} \leftarrow o_i, \forall i \in \text{Layer } k, \forall j \in \text{Layer } k+1$$

Expressiveness of feed-forward neural networks

- A single sigmoid neuron has the same representational power as a perceptron: Boolean AND, OR, NOT, but not XOR
- Every Boolean function can be represented by a network with a single hidden layer, but might require a number of hidden units that is exponential in the number of inputs
- Every bounded continuous function can be approximated with arbitrary precision by a network with one, sufficiently large hidden layer [Cybenko 1989; Hornik et al. 1989]
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].

NN are universal approximators in sup norm

- We have the following universal approximation result in sup norm due to Hornik, Stinchcombe and White 1989, Cybenko 1989 and Hornik 1991

  Theorem 1. *Let $\phi(\cdot)$ be any continuous, bounded and nonconstant function from R to R. Denote by $C(X)$ the space of continuous functions on any compact subset X of R^n. Then, given any function $f \in C(X)$ and $\epsilon > 0$, there exist an integer H and real constants $a_i, b_i \in R$, $w_i \in R^n$, $i = 1, \ldots, H$ such that neural network*

$$F(x) = \sum_{i=1}^{H} a_i \phi(w_i^T x + b_i) \tag{1}$$

  approximates f in sup norm, i.e. functions of the form (1) are dense in $C(X)$ or

$$\sup_{x \in X} |F(x) - f(x)| \leq \epsilon$$

NN are universal approximators in $L_p(\mu)$ norm

- We have the following universal approximation result in $L_p(\mu)$ due to Hornik 1991, where μ is any finite measure on R^n

  Theorem 2. *Let $\phi(\cdot)$ be any bounded and nonconstant function from R to R. Then, given any function $f \in L_p(\mu)$ and $\epsilon > 0$, there exist an integer H and real constants $a_i, b_i \in R$, $w_i \in R^n$, $i = 1, \ldots, H$ such that neural network*

$$F(x) = \sum_{i=1}^{H} a_i \phi(w_i^T x + b_i) \tag{2}$$

  approximates f in $L_p(\mu)$, i.e. functions of the form (2) are dense in $L_p(\mu)$ or

$$\int_{R^n} |F(x) - f(x)|^p d\mu(x) \leq \epsilon$$

- It's the multilayer feedforward *architecture* itself rather than specific choice of the activation function that gives neural networks universal approximation power

Rate of approximation by NN in $L_p(\mu)$ norm

- Consider the class of functions $f \in R^n$ with the following Fourier representation

$$f(x) = \int_{R^n} \exp(i\omega^T x)\tilde{f}(\omega)d\omega$$

where $\tilde{f}(\omega)$ is the Fourier transform of $f(x)$ and let

$$C_f = \int_{R^n} ||\omega|||\tilde{f}(\omega)|d\omega$$

- Let Γ_C be a class of functions f that satisfy the following bound

$$C_f \leq C$$

Rate of approximation by NN in $L_p(\mu)$ norm

- We have the following rate of approximation result in $L_p(\mu)$ due to Barron 1993

  Theorem 3. *For every function f with C_f finite, and every $H \geq 1$, there exists a neural network*

$$F(x) = \sum_{i=1}^{H} a_i \phi(w_i^T x + b_i) \tag{3}$$

  such that

$$\int_{B_r} (F(x) - f(x))^2 d\mu(x) \leq \frac{(2rC_f)^2}{H}, \tag{4}$$

  where B_r is a ball of radius r centered at 0. If $f \in \Gamma_C$, then output coefficients of the network can be restricted to satisfy

$$\sum_{i=1}^{H} |a_i| \leq 2rC.$$

State of art of universal approximation with feed-forward neural networks

TABLE 1
The main characteristics of the reviewed approaches

	Network type	Decomposition in ridge functions	Ridge function approximation	Error bound	Constructive approach
Cybenko (1989), Hornik (1990, 1993)	Three layers, ridge activations		Hahn–Banach theorem	None	No
Katsuura and Sprecher (1994)	Four layers, ridge activatons		Kolmogorov's theorem	None	Yes
Kurkova (1992)	Four layers, ridge activations		Kolmogorov's theorem	$O(n^{-1/(d+2)})$	Yes
Chui and Li (1992)	Four layers, ridge activatons	Sums of powers	Hahn–Banach	None	For polynomials theorem and only the decomposition part
Chen et al. (1995), Carroll and Dickinson (1989)	Three layers, ridge activations	Radom Transform	*step* function argument	$O(n^{-1/(d-1)})$	Not completely
Barron (1993)	Three layers, ridge activation	Fourier distrbutions	*step* function argument	$O(n^{-1/2})$	No
Mhaskar and Micchelli (1992, 1994)	Three layers, ridge activations	Fourier series	Fourier series	$O(n^{-1/4})$	Not completely
Park and Sandberg (1991, 1993)	Three layers, RBFs		ϵ-mollifiers	$O(n^{-1/d})$	Yes
Our algorithm 1	Three layers, ridge activations	Sums of powers	*step* function argument	$O(n^{-1})$	Yes, for polynomials
Our algorithm 2	Three layers, ridge activations	Sums of powers	eqn (24)	Any degree of precision by a finite number of neurons	Yes, for polynomials

Learning in feed-forward neural networks

- Usually, the network structure (units and interconnections) is specified by the designer
- The learning problem is finding a *good set of weights*
- The answer: *gradient descent*, because the form of the hypothesis formed by the network, $h_{\mathbf{w}}$, is
  - Differentiable! Because of the choice of sigmoid units
  - Very complex! Hence, direct computation of the optimal weights is not possible
- Both mean-squared error and cross-entropy are used routinely as error functions.

Deriving a gradient descent update for networks using MSE

- Suppose for simplicity that we have a fully connected network with N input units (of indices $1, \ldots N$), H hidden units (of indices $N + 1, \ldots N + H$) and one output unit (index $N + H + 1$)
- Suppose we want to compute the weight update after seeing one instance $\langle \mathbf{x}, y \rangle$
- Let $o_i, i = 1, \ldots N + H + 1$ be the outputs of all units in the network for the given inputs $\mathbf{x}$ (we omit the argument for simplicity of notation)
- The sum-squared error function is:

$$J(\mathbf{w}) = \frac{1}{2}(y - h_{\mathbf{w}}(\mathbf{x}))^2 = \frac{1}{2}(y - o_{N+H+1})^2$$

Deriving a gradient descent update for networks using MSE(2)

- The derivative with respect to the weights $w_{N+H+1,j}$ entering o_{N+H+1} is computed like usual:

$$\frac{\partial J}{\partial w_{N+H+1,j}} = -(y - o_{N+H+1})o_{N+H+1}(1 - o_{N+H+1})x_{N+H+1,j}$$

- For convenience, let:

$$\delta_{N+H+1} = (y - o_{N+H+1})o_{N+H+1}(1 - o_{N+H+1})$$

- Hence, we can write:

$$\frac{\partial J}{\partial w_{N+H+1,j}} = -\delta_{N+H+1}x_{N+H+1,j}$$

Deriving a gradient descent update for networks (3)

- The derivative wrt the other weights, $w_{l,j}$ where $j = 1, \ldots N$ and $l = N+1, \ldots N+H$, can be computed using chain rule:

$$\begin{aligned}\frac{\partial J}{\partial w_{l,j}} &= -(y - o_{N+H+1}) o_{N+H+1}(1 - o_{N+H+1}) \frac{\partial}{\partial w_{l,j}} \mathbf{w}_{N+H+1}^T \mathbf{x}_{N+H+1} \\ &= -\delta_{N+H+1} w_{N+H+1,l} \frac{\partial}{\partial w_{l,j}} x_{N+H+1,l}\end{aligned}$$

- Recall that $x_{N+H+1,l} = o_l$. Hence, we have:

$$\frac{\partial}{\partial w_{l,j}} x_{N+H+1,l} = o_l(1 - o_l) x_{l,j}$$

- Putting these together, and using similar notation as before, we have:

$$\frac{\partial J}{\partial w_{l,j}} = -o_l(1 - o_l) \delta_{N+H+1} w_{N+H+1,l} x_{l,j} = -\delta_l x_{l,j}$$

Backpropagation algorithm

- Just gradient descent over all weights in the network!
- We put together the two phases described above:
  1. *Forward pass:* Compute the outputs of all units in the network, $o_k, k = N+1, \ldots N+H+1$, going in increasing order of the layers
  2. *Backward pass:* Compute the δ_k updates described before, going from $k = N+H+1$ down to $k = N+1$ (in decreasing order of the layers)
  3. Update to all the weights in the network:

$$w_{i,j} \leftarrow w_{i,j} + \alpha_{i,j}\delta_i x_{i,j}$$

For the cross-entropy error function, drop the $o(1-o)$ terms from the δ

Backpropagation algorithm

1. Initialize all weights to small random numbers.
2. Repeat until satisfied:
   (a) Pick a training example
   (b) Input example to the network and compute the outputs o_k
   (c) For each output unit k, $\delta_k \leftarrow o_k(1 - o_k)(y_k - o_k)$
   (d) For each hidden unit l

$$\delta_l \leftarrow o_l(1 - o_l) \sum_{k \in outputs} w_{lk}\delta_k$$

   (e) Update each network weight: $w_{ij} \leftarrow w_{ij} + \alpha_{ij}\delta_j x_{ij}$
   - x_{ij} is the input from unit i into unit j (for the output neurons, these are signals received from the hidden layer neurons)
   - α_{ij} is the learning rate or step size

Backpropagation variations

- The previous version corresponds to incremental (stochastic) gradient descent
- An analogous batch version can be used as well:
  - Loop through the training data, accumulating weight changes
  - Update weights
- One pass through the data set is called *epoch*
- Generalization is easy both for different error functions and for other network structures.

Convergence of backpropagation

- Backpropagation performs gradient descent over all the parameters in the network
- Hence, if the learning rate is appropriate, the algorithm is guaranteed to converge to a *local minimum* of the cost function
  - NOT the global minimum
  - Can be much worse than global minimum
  - There can be MANY local minima (Auer et al, 1997)
- Partial solution: *restarting* $=$ train multiple nets with different initial weights.
- In practice, quite often the solution found is very good

Adding momentum

- On the t-th training sample, instead of the update:

$$\Delta w_{ij} \leftarrow \alpha_{ij}\delta_j x_{ij}, \text{ we do:}$$

$$\Delta w_{ij}(t) \leftarrow \alpha_{ij}\delta_j x_{ij} + \beta \Delta w_{ij}(t-1)$$

  The second term is called *momentum*

- Advantages:
  - Easy to pass small local minima
  - Keeps the weights moving in areas where the error is flat
  - Increases the speed where the gradient stays unchanged
- Disadvantages:
  - With too much momentum, it can get out of a global maximum!
  - One more parameter to tune, and more chances of divergence

Choosing the learning rate

- Backprop is *very sensitive* to the choice of learning rate
  - Too large $\Rightarrow$ divergence
  - Too small $\Rightarrow$ VERY slow learning
  - The learning rate also influences the ability to escape local optima
- Very often, different learning rates are used for units inn different layers
- It can be shown that each unit has its own optimal learning rate

Adjusting the learning rate: Delta-bar-delta

- Heuristic method, works best in batch mode (though there have been attempts to make it incremental)
- The intuition:
  - If the gradient direction is stable, the learning rate should be increased
  - If the gradient flips to the opposite direction the learning rate should be decreased
- A running average of the gradient and a separate learning rate is kept for each weight
- If the new gradient and the old average have the same sign, increase the learning rate by a constant amount
- If they have opposite sign, decay the learning rate exponentially

Second-order methods for neural nets

- Based on computing both first and second-order derivatives of the error function
- Quickprop: assume all weights are independent, optimize the learning rate independently for each one
- Conjugate gradient
  - Pick a first direction, e.g. regular gradient, and go to a local optimum
  - Look for a direction such that the gradient only varies in magnitude, not orientation.
- This ensures that no previous improvements are lost

Choosing a good input encoding: Discrete inputs

- Discrete inputs with k possible values are often encoded using a "1 hot" or "1-of-k" encoding:
  - k input bits are associated with the variable (one for each possible value)
  - For any instance, all bits are 0 except the one corresponding to the value found in the data, which is set to 1
  - If the value is missing, all inputs are set to 0

Choosing a good input encoding: Real-valued inputs

- For continuous (numeric) inputs, it is important to *scale* the inputs so they are all in a common, reasonable range
- One standard transformation is to "normalize" the data, i.e., make it such that it has mean 0, unit variance, by subtracting the mean and dividing by the standard deviation
- When is this good? When is this bad?

  Good if the data is roughly normal, but bad if we have outliers
- Alternative representations:
  - 1-hot encoding - what is the disadvantage?
    Potentially lots of values! Also, the ordering of values is lost
  - Thermometer encoding (nonnegative integer n is encoded as $n-1$ ones followed by a zero)

Output

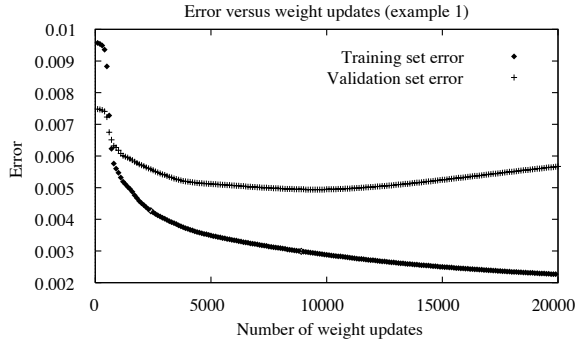
- A network can have several output units
- This can be useful when we want to predict a real-valued variable, and it has several ranges

  E.g. Dean Pomerleau's autonomous driving car
- Alternatively, we can allow one output unit, without a sigmoid function
- Normalization can be used if the output data is roughly normal

How large should the network be?

- Overfitting occurs if there are too many parameters compared to the amount of data available
- Choosing the number of hidden units:
  - Too few hidden units do not allow the concept to be learned
  - Too many lead to slow learning and overfitting
  - If the n inputs are binary, $\log n$ is a good heuristic choice
- Choosing the number of layers
  - Always start with one hidden layer
  - Never go beyond 2 hidden layers, unless the task structure suggests something different

Overtraining in feed-forward networks



- Traditional overfitting is concerned with the number of parameters vs. the number of instances
- In neural networks there is an additional phenomenon called *overtraining* which occurs when weights take on large magnitudes, pushing the sigmoids into saturation
- Effectively, as learning progresses, the network has more actual parameters
- *Use a validation set to decide when to stop training!*
- Regularization is also very effective