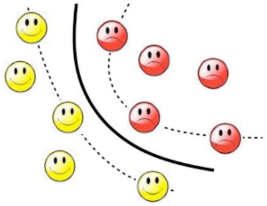


Machine Learning and Data Mining (COMP 5318)

Large-scale machine learning

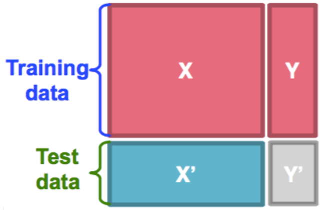
Nguyen Hoang Tran

Supervised Learning

Task: Given data **(X,Y)** build a model **f()** to predict **Y' ** based on **X'**

Strategy: Estimate $y = f(x)$ on (X, Y).

Hope that the same $f(x)$ also works to predict unknown Y'



- The **"hope"** is called **generalization**
  - **Overfitting: If f(x) predicts well Y but is unable to predict Y'**
- **We want to build a model that generalizes well to unseen data**

How can we do well on data we have never seen before?

Perceptron

Ullman's book, Chapter 12

(http://infolab.stanford.edu/~ullman/mmds/ch12.pdf)

Bishop's book, Chapter 5

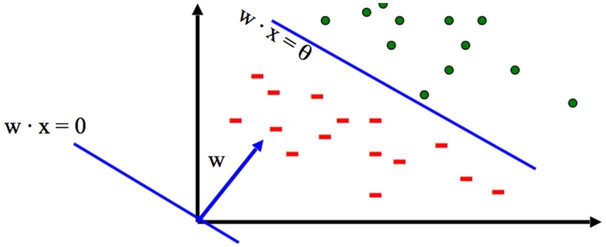
Linear models for classification

- **Binary classification:**

$$f(\mathbf{x}) = \begin{cases} +1 & \text{if} \quad w_1 x_1 + w_2 x_2 + \ldots w_d x_d \geq \theta \\ -1 & \text{otherwise} \end{cases}$$

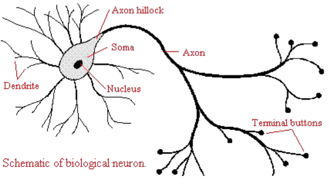
Decision boundary is **linear**

- **Input:** Vectors $\boldsymbol{x}^{(j)}$ and labels $\boldsymbol{y}^{(j)}$
  - Vectors $\boldsymbol{x}^{(j)}$ are real valued where $\|x\|_2 = 1$
- **Goal:** Find vector $w = (w_1, w_2, \ldots, w_d)$
  - Each w_i is a real number

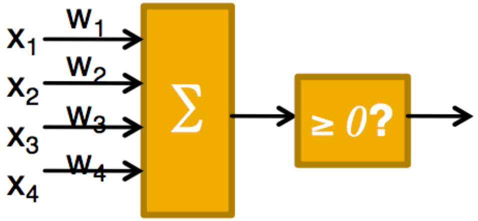


Perceptron [Rosenblatt '58]

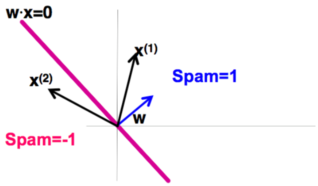
- **(very) Loose motivation: Neuron**
- Inputs are feature values
- Each feature has a weight w_i



- **Activation is the sum:**
  - $f(x) = \Sigma_i w_i x_i = w \cdot x$



- If the ***f(x)*** is:
  - **Positive:** Predict **+1**
  - **Negative:** Predict **-1**



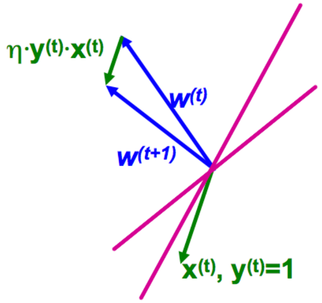
Perceptron: Estimating *w*

- **Perceptron: *y' = sign(w· x)***
- **How to find parameters *w*?**
  - Start with $w_0 = 0$
  - Pick training examples $x^{(t)}$ **one by one (from disk)**
  - Predict class of $x^{(t)}$ using current weights
    - $y' = sign(w^{(t)} \cdot x^{(t)})$
  - **If *y'* is correct (i.e., $y_t = y'$)**
    - No change: $w^{(t+1)} = w^{(t)}$
  - **If *y'* is wrong:** adjust $w^{(t)}$

$$w^{(t+1)} = w^{(t)} + \eta \cdot y^{(t)} \cdot x^{(t)}$$

- η is the learning rate parameter
- $x^{(t)}$ is the t-th training example
- $y^{(t)}$ is true t-th class label ({+1, -1})

Note that the Perceptron is a conservative algorithm: it ignores samples that it classifies correctly.



Perceptron in hardware

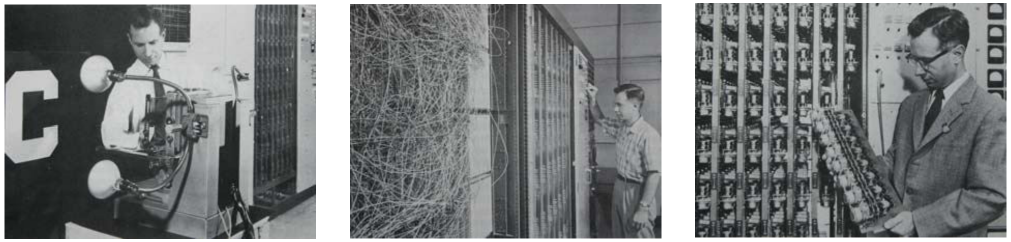


Figure 4.8 Illustration of the Mark 1 perceptron hardware. The photograph on the left shows how the inputs were obtained using a simple camera system in which an input scene, in this case a printed character, was illuminated by powerful lights, and an image focussed onto a 20×20 array of cadmium sulphide photocells, giving a primitive 400 pixel image. The perceptron also had a patch board, shown in the middle photograph, which allowed different configurations of input features to be tried. Often these were wired up at random to demonstrate the ability of the perceptron to learn without the need for precise wiring, in contrast to a modern digital computer. The photograph on the right shows one of the racks of adaptive weights. Each weight was implemented using a rotary variable resistor, also called a potentiometer, driven by an electric motor thereby allowing the value of the weight to be adjusted automatically by the learning algorithm.

(1958)
F. Rosenblatt

The perceptron: a probabilistic model for information storage and organization in the brain
Psychological Review 65:386–408

From: Pattern Recognition and Machine Learning by C. Bishop

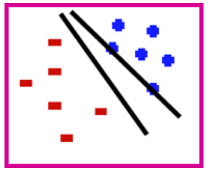
Perceptron Convergence

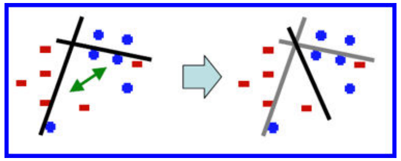
Perceptron Convergence Theorem:

- If there exist a set of weights that are consistent (i.e., the data is linearly separable) the Perceptron learning algorithm will converge

Perceptron Cycling Theorem:

- If the training data is not linearly separable the Perceptron learning algorithm will eventually repeat the same set of weights and therefore enter an infinite loop





Updating the Learning Rate

Perceptron will oscillate and won’t converge

When to stop learning?

(1) Slowly decrease the learning rate η

- A classic way is to: $\eta = c_1/(t + c_2)$
- But, we also need to determine constants c_1 and c_2

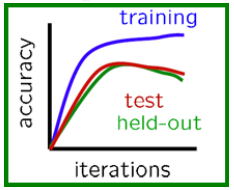
(2) Stop when the training error stops changing

(3) Have a small validation dataset and stop when the validation set error stops decreasing

(4) Stop when we reached some maximum number of passes over the data

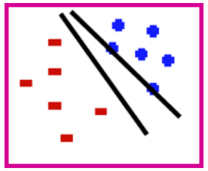
Issues with Perceptrons

Overfitting:

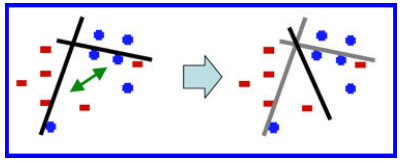


Regularisation:

If the data is separable

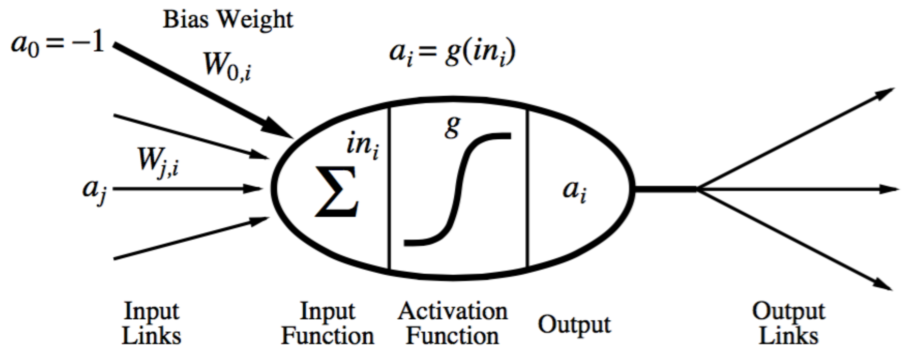


If the data is not separable weights dance around

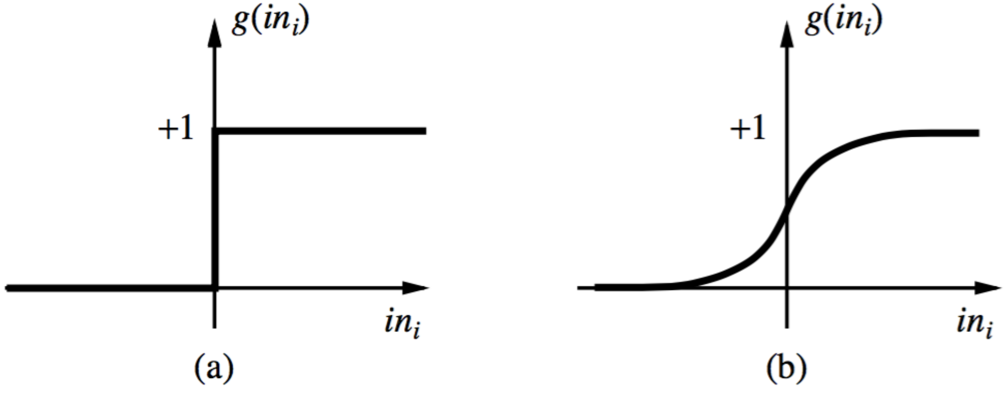


Activation Functions (1)

$$a_i \leftarrow g(in_i) = g\left(\Sigma_j W_{j,i} a_j\right)$$



Activation Functions (2)



(a) is a step function or threshold function
(b) is a sigmoid function $1/(1+e^{-x})$

Changing the bias weight $W_{0,i}$ moves the threshold location

Activation Functions (3)

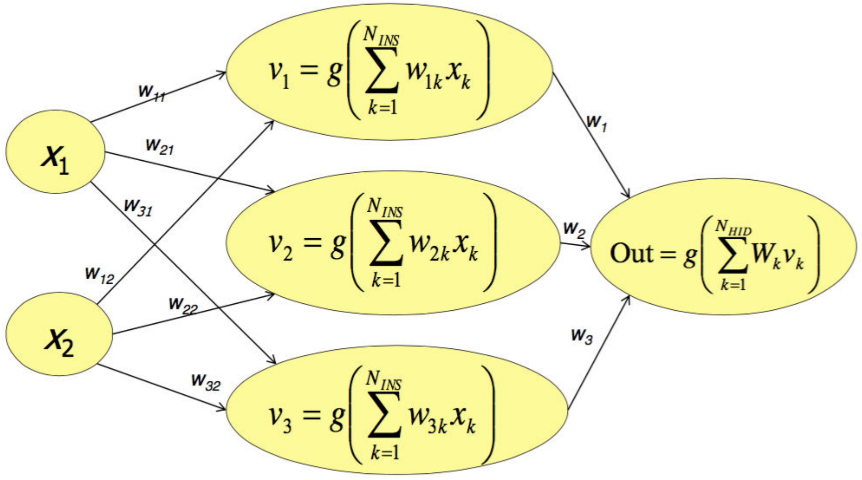
THE UNIVERSITY OF SYDNEY

$$
\begin{aligned}
g'_{\text{sigmoid}}(x) &= \frac{\partial}{\partial x}\left(\frac{1}{1+e^{-x}}\right) \\
&= \frac{e^{-x}}{(1+e^{-x})^2} \quad \text{(chain rule)} \\
&= \frac{e^{-x}+1-1}{(1+e^{-x})^2} \\
&= \frac{e^{-x}+1}{(1+e^{-x})^2} - \frac{1}{(1+e^{-x})^2} \\
&= \frac{1}{(1+e^{-x})} - \frac{1}{(1+e^{-x})^2} \\
&= g_{\text{sigmoid}}(x) - g_{\text{sigmoid}}(x)^2 \\
&= g_{\text{sigmoid}}(x)(1 - g_{\text{sigmoid}}(x))
\end{aligned}
$$

$F(x) = g(f(x))$, then $F'(x) = g'(f(x))f'(x)$.

Multi-Layer Neural Networks

- There are many ways to connect perceptrons into a network. One standard way is multi-layer neural nets
- 1 Hidden layer: we can't see the output; 1 output layer



x_1
x_2
w_{11}
w_{21}
w_{31}
w_{12}
w_{22}
w_{32}
$v_1 = g\left(\sum_{k=1}^{N_{INS}} w_{1k} x_k\right)$
$v_2 = g\left(\sum_{k=1}^{N_{INS}} w_{2k} x_k\right)$
$v_3 = g\left(\sum_{k=1}^{N_{INS}} w_{3k} x_k\right)$
W_1
W_2
W_3
$\text{Out} = g\left(\sum_{k=1}^{N_{HID}} W_k v_k\right)$

The power of Neural Nets

In theory

we don't need to many layers

1-hidden-layer NN with enough hidden units can represents any continuous function of the inputs with arbitrary accuracy

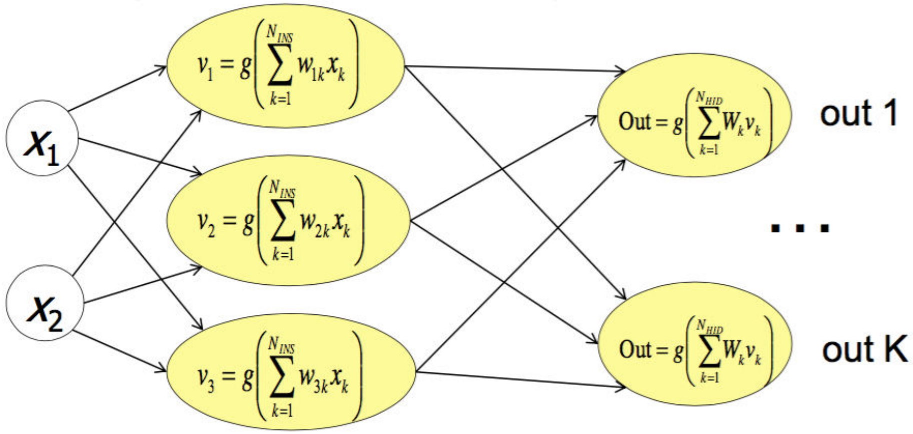
2-hidden-layer NN can even represents discontinuous function

Neural Net for Multi-Class Classification

- Use K output units. During training, encode a label y by an indicator vector with K entries.

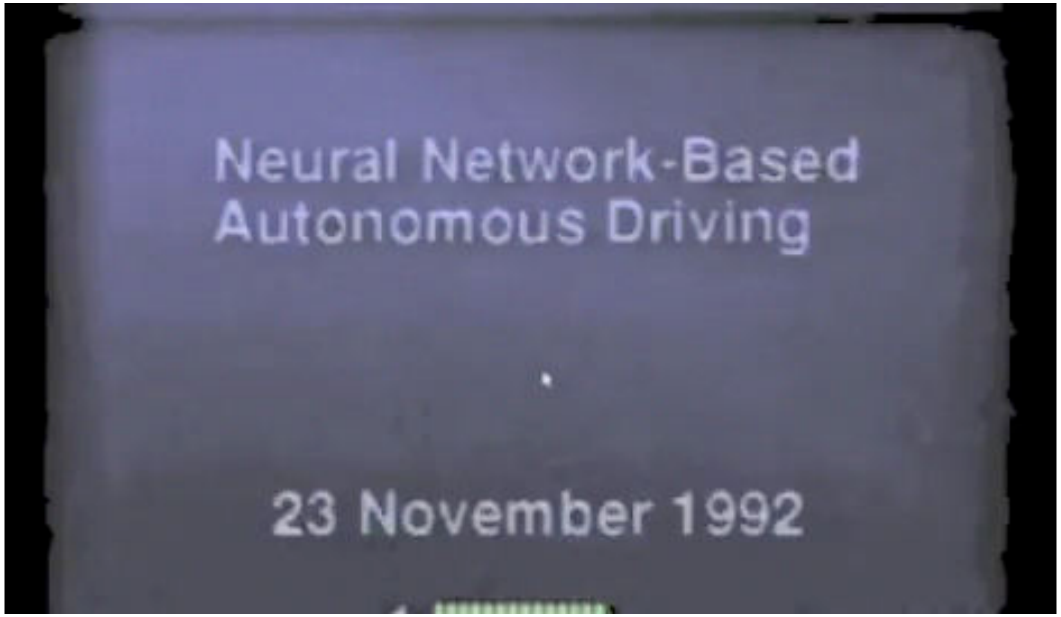
  class1=(1,0,0,...,0), class2=(0,1,0,...,0) etc.

- During test (encoding), choose the class corresponding to the largest output



x_1
x_2
$v_1 = g\left(\sum_{k=1}^{N_{INS}} w_{1k} x_k\right)$
$v_2 = g\left(\sum_{k=1}^{N_{INS}} w_{2k} x_k\right)$
$v_3 = g\left(\sum_{k=1}^{N_{INS}} w_{3k} x_k\right)$
$\text{Out} = g\left(\sum_{k=1}^{N_{HID}} W_k v_k\right)$
out 1
...
$\text{Out} = g\left(\sum_{k=1}^{N_{HID}} W_k v_k\right)$
out K

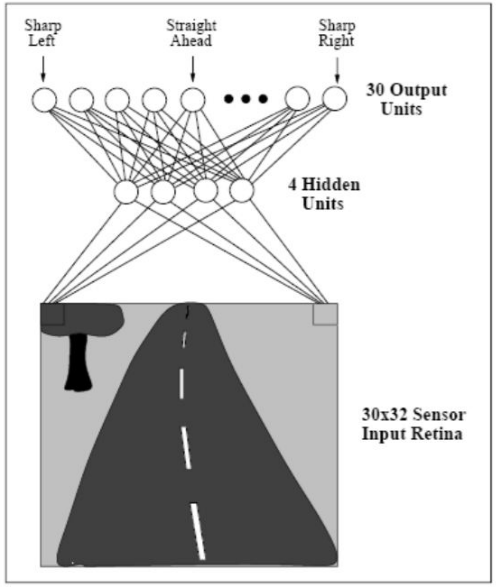
Classification for Autonomous Driving



Neural Network-Based Autonomous Driving

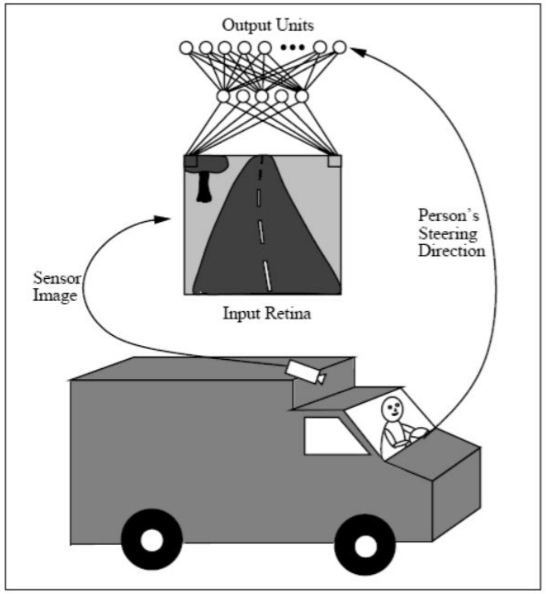
23 November 1992

Network structure



[Pomerleau, 1995]

Training data



[Pomerleau, 1995]

Learning a neural network

- Again we will minimise the error (K outputs):

$$E(W) = ½ \sum_{i=1..N} \sum_{c=1..K} (o_{ic}-Y_{ic})^2$$

- i: the i-th training point
- o_{ic}: the c-th output for the i-th training point
- Y_{ic}: the c-th element of the i-th label indicator vector
- Our variables are all the weights w on all the edges

- Apparent difficulty: we don't know the 'correct' output of hidden units
- It turns out to be OK: we can still do gradient descent. The trick you need is the chain rule
- The algorithm is known as **back-propagation**

Back-Propagation (1)

BACKPROPAGATION (training set, α, D, n_{hidden}, K)

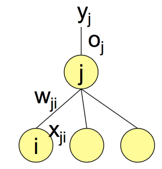
- Training set: {(X1,Y1), ..., (Xn,Yn)}, Xi is a feature vector of dimension D, Yi is an output vector of dimension K, α is the learning rate (step size in gradient descent), n_{hidden} is the number of hidden units
- Create a NN with D inputs, n_{hidden} hidden units, and K outputs. Connect each layer
- Initialise all weights to some small random numbers (e.g. between –0.05 and 0.05)
- Repeat the next page until the terminate condition is met…

Back-Propagation (2)

For each example (X,Y)

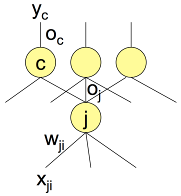
- Propagate the input forward through the network
  - Input X to the network, compute output o_u for any unit u in the network
- Propagate the errors backward through the network
  - For each output unit c, compute its error term

$$\delta_c \leftarrow (o_c - y_c)o_c(1-o_c)$$



  - For each hidden unit h, compute its error term

$$\delta_h \leftarrow \left(\sum_{i\in succ(h)} w_{ih}\delta_i\right) o_h(1-o_h)$$



  - Update each weights

$$w_{ji} \leftarrow w_{ji} - \alpha\delta_j x_{ji}$$

where x_{ji} is the input from unit i to unit j (o_u if i is a hidden unit; X_i if i is an input), w_{ji} is the corresponding weight.

Back-Propagation (3)

- For simplicity we assume online learning (as oppose to batch learning): 1-step gradient descent after seeing each training example (X,Y)
- For each (X,Y), the error is

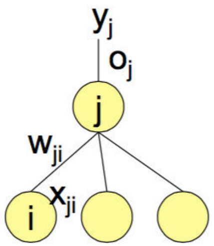
$$E(W) = ½ \sum_{c=1..K} (o_c - Y_c)^2$$

  - o_c: the c-th output unit (when input is X)
  - Y_c: the c-th element of the label indicator vector
  - Update each weights
- Use gradient descent to change all weights to minimise the errors. Separate the cases:
  - Case 1: w_{ji}, when j is an output unit
  - Case 2: w_{ji}, when j is a hidden unit

Case 1: weights of an output unit

$$\frac{\partial Error}{\partial w_{ji}} = \frac{\partial \frac{1}{2}(o_j - y_j)^2}{\partial w_{ji}} = \frac{\partial \frac{1}{2}(g(\sum_m w_{jm}x_{jm}) - y_j)^2}{\partial w_{ji}}$$

$$= (o_j - y_j)o_j(1-o_j)x_{ji}$$



- o_c: the c-th output unit (when input is X)
- Y_c: the c-th element of the label indicator vector

gradient descent: to minimise error, run away from the partial deviation

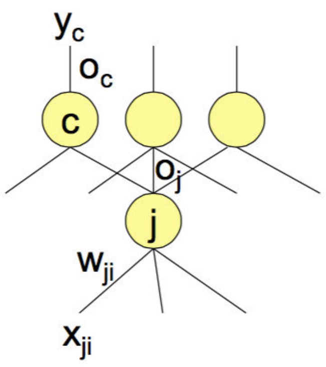
$$w_{ji} \leftarrow w_{ji} - \alpha\frac{\partial Error}{\partial w_{ji}} = w_{ji} - \alpha(o_j - y_j)o_j(1-o_j)x_{ji}$$

Case 2: weights of a hidden unit

$$\frac{\partial Error}{\partial w_{ji}} = \sum_{c=succ(j)} \frac{\partial E_c}{\partial o_c} \cdot \frac{\partial o_c}{\partial o_j} \cdot \frac{\partial o_j}{\partial w_{ji}}$$

$$= \sum_{c=succ(j)} (o_c - y_c) \cdot \frac{\partial g(\sum_m w_{cm} x_{cm})}{\partial x_{cm}} \cdot \frac{\partial g(\sum_n w_{jn} x_{jn})}{\partial w_{ji}}$$

$$= \sum_{c=succ(j)} (o_c - y_c) \cdot o_c (1 - o_c) w_{cj} \cdot o_j (1 - o_j) x_{ji}$$



Neural network learning issues

- When to terminate backpropagation? Overfitting and early stopping
  - After fixed number of iterations (ok)
  - When training error less than a threshold (wrong)
  - When holdout error starts to go up (ok)
- Local optima
  - The weights will converge to a local minimum
- Learning rate
  - Convergence sensitive to learning rate
  - Weight learning can be rather slow

Stochastic gradient descent

How to estimate *w*?

$$\min_{w,b} \tfrac{1}{2} w \cdot w + C \cdot \sum_{i=1}^{n} \xi_i$$

$$s.t. \forall i, y_i \cdot (x_i \cdot w + b) \geq 1 - \xi_i$$

- **Want to estimate *w* and *b*!**
  - **Standard way:** Use a solver!
    - **Solver:** software for finding solutions to "common" optimization problems
- **Use a quadratic solver:**
  - Minimize quadratic function
  - Subject to linear constraints
- **Problem:** Solvers are inefficient for big data!

How to estimate *w*?

- **Want to minimize *f(w,b)*:**

$$f(w,b) = \tfrac{1}{2}\sum_{j=1}^{d}\left(w^{(j)}\right)^2 + \underbrace{C\sum_{i=1}^{n}\max\left\{0, 1 - y_i\left(\sum_{j=1}^{d} w^{(j)} x_i^{(j)} + b\right)\right\}}_{\text{Empirical loss } L(x_i\, y_i)}$$

- **Compute the gradient ∇(j) w.r.t. *w(j)***

$$\nabla f^{(j)} = \frac{\partial f(w,b)}{\partial w^{(j)}} = w^{(j)} + C\sum_{i=1}^{n}\frac{\partial L(x_i, y_i)}{\partial w^{(j)}}$$

$$\begin{aligned}\frac{\partial L(x_i, y_i)}{\partial w^{(j)}} &= 0 \qquad \text{if } y_i(\mathrm{w}\cdot x_i + b) \geq 1\\ &= -y_i x_i^{(j)} \qquad \text{else}\end{aligned}$$

How to estimate *w*?

Gradient descent:

Iterate until convergence:

- **For j = 1 … d**
  - **Evaluate:** $\nabla f^{(j)} = \frac{\partial f(w,b)}{\partial w^{(j)}} = w^{(j)} + C\sum_{i=1}^{n}\frac{\partial L(x_i, y_i)}{\partial w^{(j)}}$
  - **Update:**
    w(j) ← w(j) - η∇f(j)

η…learning rate parameter
C… regularisation parameter

Problem

computing gradients takes O(n) time

Storage of n training examples

n is the size of data

Gradient descent for MSE cost function

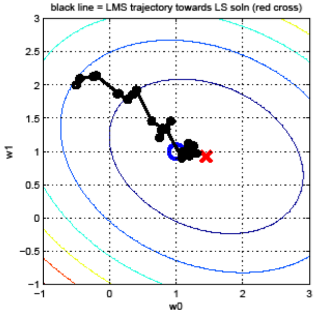
$$\begin{aligned}\mathbf{w}_{k+1} &= \mathbf{w}_k - \eta_k \mathbf{g}(\mathbf{w}_k)\\ \mathbf{g}(\mathbf{w}) &\propto \mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{y}) = \sum_{i=1}^{n}\mathbf{x}_i(\mathbf{w}^T\mathbf{x}_i - y_i)\end{aligned}$$

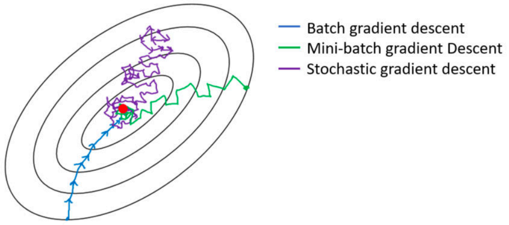
Stochastic gradient descent

- Approximate the gradient by looking at a single data case

$$\mathbf{g}(\mathbf{w}_k) \approx \mathbf{x}_i(\mathbf{w}^T\mathbf{x}_i - y_i)$$

- Can be used to learn online





Stochastic gradient descent

We just had:

$$\nabla f^{(j)} = w^{(j)} + C\sum_{i=1}^{n} \frac{\partial L(x_i,y_i)}{\partial w^{(j)}}$$

- **Stochastic Gradient Descent**
  - Instead of evaluating gradient over all examples evaluate it for each **individual** training example

$$\nabla f^{(j)}(x_i) = w^{(j)} + C \cdot \frac{\partial L(x_i,y_i)}{\partial w^{(j)}}$$

Notice: no summation over *i* anymore

- **Stochastic gradient descent:**

```
Iterate until convergence:
• For i = 1 … n
    • For j = 1 … d
        • Compute: $\nabla f^{(j)}(x_i)$
        • Update: $w^{(j)} \leftarrow w^{(j)} - \eta \nabla f^{(j)}(x_i)$
```

Example: Text categorisation

Example by Leon Bottou:

- **Reuters RCV1** document corpus
  - Predict a category of a document
  - One **vs.** the rest classification
- ***n*** **= 781,000** training examples (documents)
- 23,000 test examples
- ***d*** **= 50,000** features
  - One feature per word
  - Remove stop-words
  - Remove low frequency words

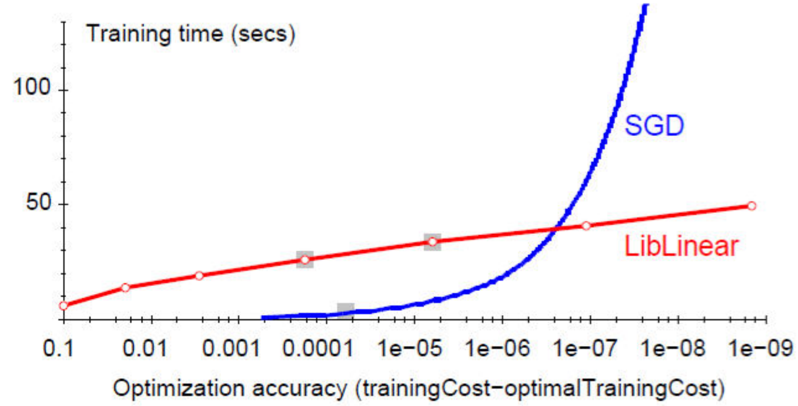
Example: Text categorisation

- **Questions:**
  - **(1)** Is **SGD** successful at minimising ***f(w,b)***?
  - **(2)** How quickly does **SGD** find the min of ***f(w,b)***?
  - **(3)** What is the error on a test set?

	Training time	*Value of f(w,b)*	*Test error*
Standard SVM	23,642 secs	0.2275	6.02%
"Fast SVM"	66 secs	0.2278	6.03%
SGD SVM	1.4 secs	0.2275	6.02%

(1) SGD-SVM is successful at minimising the value of ***f(w,b)***
(2) SGD-SVM is super fast
(3) SGD-SVM test set error is comparable

Optimisation "Accuracy"



For optimising *f(w,b) within reasonable* quality
SGD-SVM is super fast

Practical Considerations

Need to choose learning rate η and t_0

$$w_{t+1} \leftarrow w_t - \frac{\eta_t}{t+t_0}\left(w_t + C\frac{\partial L(x_i, y_i)}{\partial w}\right)$$

Leon suggests:

Choose $\mathbf{t}_0$ so that the expected initial updates are comparable with the expected size of the weights

Choose η:

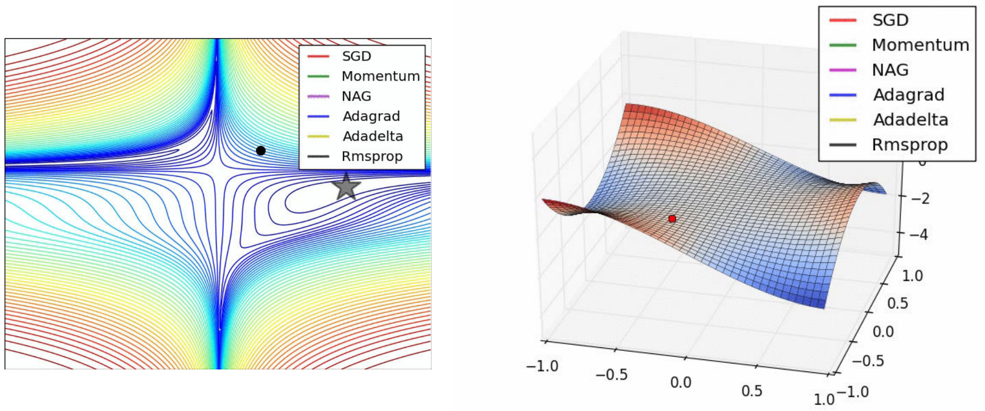
- Try various rates η (e.g., 10, 1, 0.1, 0.01, ...)
- Pick the one that most reduces the cost
- Use η for next 100k iterations on the full dataset

Practical Considerations

Stopping criteria:
How many iterations of SGD?

- **Early stopping with cross validation**
  - Create a validation set
  - Monitor cost function on the validation set
  - Stop when loss stops decreasing

Variations of SGD



Check: http://ruder.io/optimizing-gradient-descent/