Neural Learning

COMP9417 Machine Learning and Data Mining

Term 2, 2020

Acknowledgements

Material derived from slides for the book
"Elements of Statistical Learning (2nd Ed.)" by T. Hastie,
R. Tibshirani & J. Friedman. Springer (2009)
`http://statweb.stanford.edu/~tibs/ElemStatLearn/`

Material derived from slides for the book
"Machine Learning: A Probabilistic Perspective" by P. Murphy
MIT Press (2012)
`http://www.cs.ubc.ca/~murphyk/MLbook`

Material derived from slides for the book
"Machine Learning" by P. Flach
Cambridge University Press (2012)
`http://cs.bris.ac.uk/~flach/mlbook`

Material derived from slides for the book
"Bayesian Reasoning and Machine Learning" by D. Barber
Cambridge University Press (2012)
`http://www.cs.ucl.ac.uk/staff/d.barber/brml`

Material derived from slides for the book
"Machine Learning" by T. Mitchell
McGraw-Hill (1997)
`http://www-2.cs.cmu.edu/~tom/mlbook.html`

Material derived from slides for the course
"Machine Learning" by A. Srinivasan
BITS Pilani, Goa, India (2016)

Aims

This lecture will enable you to describe and reproduce machine learning approaches to the problem of neural (network) learning. Following it you should be able to:

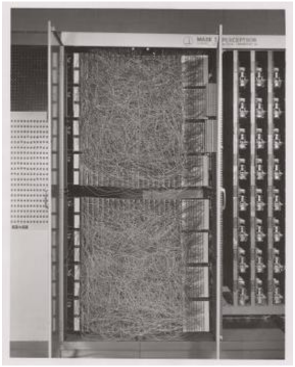
- describe Perceptrons and how to train them
- relate neural learning to optimization in machine learning
- outline the problem of neural learning
- derive the method of gradient descent for linear models
- describe the problem of non-linear models with neural networks
- outline the method of back-propagation training of a multi-layer perceptron neural network
- describe the application of neural learning for classification
- outline back-propagation in terms of computational graphs
- describe some issues arising when training deep neural networks

Perceptron

A linear classifier that can achieve perfect separation on linearly separable data is the *perceptron*, originally proposed as a simple *neural network* by F. Rosenblatt in the late 1950s.

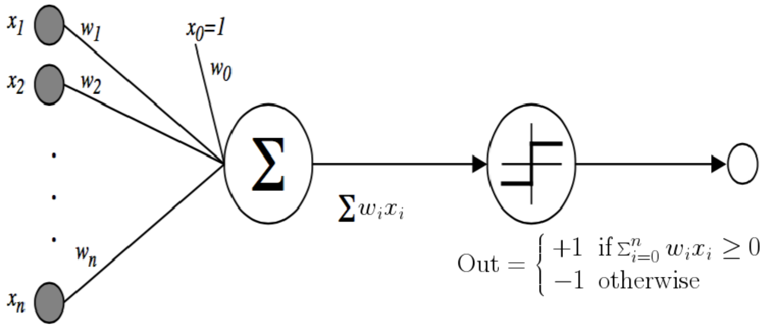
Perceptron

Originally implemented in software (based on the McCulloch-Pitts neuron from the 1940s), then in hardware as a 20x20 visual sensor array with potentiometers for adaptive weights.



Source http://en.wikipedia.org/w/index.php?curid=47541432

Perceptron

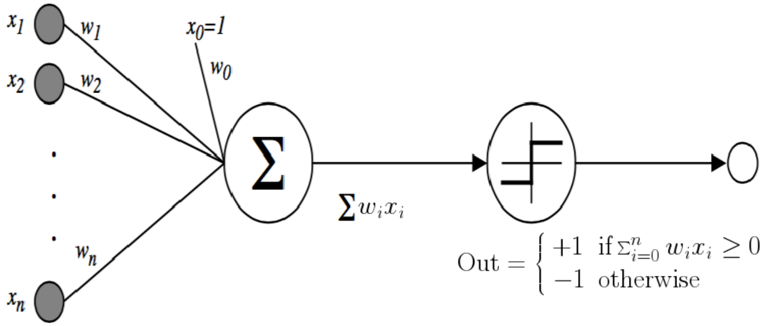


x_1
x_2
x_n
w_1
w_2
w_n
$x_0=1$
w_0
Σ
$\sum w_i x_i$
$\text{Out} = \begin{cases} +1 & \text{if } \Sigma_{i=0}^{n} w_i x_i \geq 0 \\ -1 & \text{otherwise} \end{cases}$

Output o is thresholded sum of products of inputs and their weights:

$$o(x_1, \ldots, x_n) = \begin{cases} +1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Perceptron

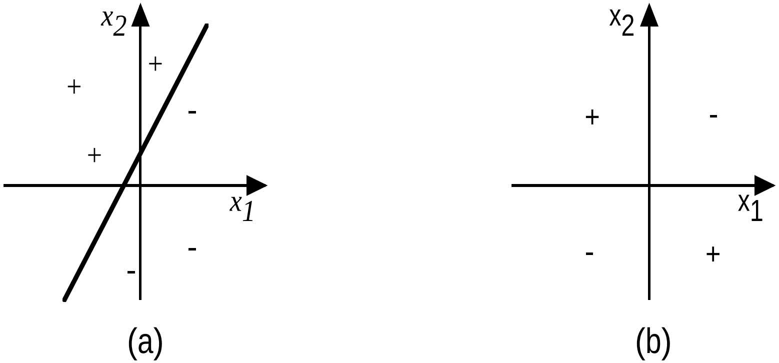


x_1
w_1
$x_0=1$
w_0
x_2
w_2
Σ
$\sum w_i x_i$
w_n
x_n
$\text{Out} = \begin{cases} +1 & \text{if } \Sigma_{i=0}^{n} w_i x_i \geq 0 \\ -1 & \text{otherwise} \end{cases}$

Or in vector notation:

$$o(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Decision Surface of a Perceptron



Represents some useful functions

- What weights represent $o(x_1, x_2) = AND(x_1, x_2)$?
- What weights represent $o(x_1, x_2) = XOR(x_1, x_2)$?

Perceptron learning

Key idea:

Learning is “finding a good set of weights”

Perceptron learning is simply an iterative weight-update scheme:

$$w_i \leftarrow w_i + \Delta w_i$$

where the weight update Δw_i depends only on *misclassified* examples and is modulated by a “smoothing” parameter η typically referred to as the “learning rate”.

Perceptron learning

The perceptron iterates over the training set, updating the weight vector every time it encounters an incorrectly classified example.

- For example, let $\mathbf{x}_i$ be a misclassified positive example, then we have $y_i = +1$ and $\mathbf{w} \cdot \mathbf{x}_i < t$. We therefore want to find $\mathbf{w}'$ such that $\mathbf{w}' \cdot \mathbf{x}_i > \mathbf{w} \cdot \mathbf{x}_i$, which moves the decision boundary towards and hopefully past x_i.
- This can be achieved by calculating the new weight vector as $\mathbf{w}' = \mathbf{w} + \eta \mathbf{x}_i$, where $0 < \eta \leq 1$ is the *learning rate* (again, assume set to 1). We then have $\mathbf{w}' \cdot \mathbf{x}_i = \mathbf{w} \cdot \mathbf{x}_i + \eta \mathbf{x}_i \cdot \mathbf{x}_i > \mathbf{w} \cdot \mathbf{x}_i$ as required.
- Similarly, if $\mathbf{x}_j$ is a misclassified negative example, then we have $y_j = -1$ and $\mathbf{w} \cdot \mathbf{x}_j > t$. In this case we calculate the new weight vector as $\mathbf{w}' = \mathbf{w} - \eta \mathbf{x}_j$, and thus $\mathbf{w}' \cdot \mathbf{x}_j = \mathbf{w} \cdot \mathbf{x}_j - \eta \mathbf{x}_j \cdot \mathbf{x}_j < \mathbf{w} \cdot \mathbf{x}_j$.

Perceptron learning

- The two cases can be combined in a single update rule:

$$\mathbf{w}' = \mathbf{w} + \eta y_i \mathbf{x}_i$$

- Here y_i acts to change the sign of the update, corresponding to whether a positive or negative example was misclassified
- This is the basis of the *perceptron training algorithm* for linear classification
- The algorithm just iterates over the training examples applying the weight update rule until all the examples are correctly classified
- If there is a linear model that separates the positive from the negative examples, i.e., the data is linearly separable, it can be shown that the perceptron training algorithm will converge in a finite number of steps.

Perceptron training algorithm

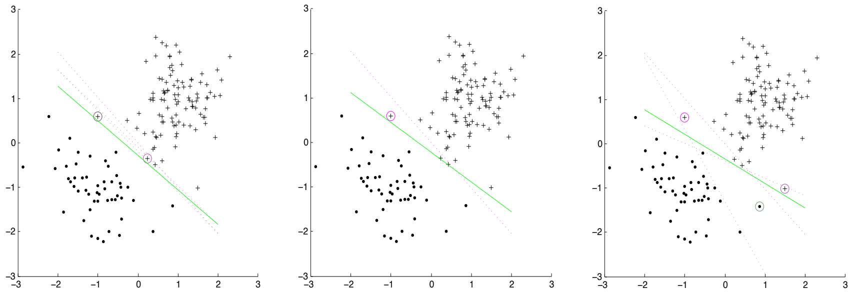
Algorithm Perceptron(D, η) // perceptron training for linear classification

Input: labelled training data D in homogeneous coordinates; learning rate η.

Output: weight vector $\mathbf{w}$ defining classifier $\hat{y} = \mathrm{sign}(\mathbf{w} \cdot \mathbf{x})$.

```
1  $\mathbf{w} \leftarrow \mathbf{0}$ // Other initialisations of the weight vector are possible
2  $converged \leftarrow$ false
3  while $converged =$ false do
4      $converged \leftarrow$ true
5      for $i = 1$ to $|D|$ do
6          if $y_i \mathbf{w} \cdot \mathbf{x}_i \leq 0$ then          // i.e., $\hat{y}_i \neq y_i$
7              $\mathbf{w} \leftarrow \mathbf{w} + \eta y_i \mathbf{x}_i$
8              $converged \leftarrow$ false  // We changed $\mathbf{w}$ so haven't converged yet
9          end
10     end
11 end
```

Perceptron training – varying learning rate



(left) A perceptron trained with a small learning rate ($\eta = 0.2$). The circled examples are the ones that trigger the weight update. (middle) Increasing the learning rate to $\eta = 0.5$ leads in this case to a rapid convergence. (right) Increasing the learning rate further to $\eta = 1$ may lead to too aggressive weight updating, which harms convergence.
The starting point in all three cases was the basic linear classifier.

Perceptron Convergence

Perceptron training will converge (under some mild assumptions) for linearly separable classification problems

A labelled data set is linearly separable if there is a linear decision boundary that separates the classes

Perceptron Convergence

Assume:

Dataset $D = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$

At least one example in D is labelled $+1$, and one is labelled -1.

$R = \max_i ||\mathbf{x}_i||_2$

A weight vector $\mathbf{w}^*$ exists s.t. $||\mathbf{w}^*||_2 = 1$ and $\forall i \; y_i \mathbf{w}^* \cdot \mathbf{x}_i \geq \gamma$

Perceptron Convergence Theorem (Novikoff, 1962)

The number of mistakes made by the perceptron is at most $(\frac{R}{\gamma})^2$.

γ is typically referred to as the "margin".

Decision Surface of a Perceptron

Unfortunately, as a linear classifier perceptrons are limited in expressive power

So some functions not representable

- e.g., not linearly separable

For non-linearly separable data we'll need something else

However, with a fairly minor modification many perceptrons can be combined together to form one model

- *multilayer perceptrons*, the classic "neural network"

Optimization

Studied in many fields such as engineering, science, economics, . . .

A general optimization algorithm: [1]

1. start with initial point $\mathbf{x} = \mathbf{x}_0$
2. select a search direction $\mathbf{p}$, usually to decrease $f(\mathbf{x})$
3. select a step length η
4. set $\mathbf{s} = \eta\mathbf{p}$
5. set $\mathbf{x} = \mathbf{x} + \mathbf{s}$
6. go to step 2, unless convergence criteria are met

For example, could minimize a real-valued function $f : \mathbb{R}^n \rightarrow \mathbb{R}$

Note: convergence criteria will be problem-specific.

[1] B. Ripley (1996) "Pattern Recognition and Neural Networks", CUP.

Optimization

Usually, we would like the optimization algorithm to quickly reach an answer that is close to being the right one.

- typically, need to minimize a function
  - e.g., error or *loss*
  - optimization is known as *gradient descent* or *steepest descent*
- sometimes, need to maximize a function
  - e.g., probability or *likelihood*
  - optimization is known as *gradient ascent* or *steepest ascent*

Connectionist Models

Consider humans:

- Neuron switching time $\approx .001$ second
- Number of neurons $\approx 10^{10}$
- Connections per neuron $\approx 10^{4-5}$
- Scene recognition time $\approx .1$ second
- 100 inference steps doesn't seem like enough

$\rightarrow$ much parallel computation

Connectionist Models

Properties of artificial neural nets (ANN's):

- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically

When to Consider Neural Networks

- Input is high-dimensional discrete or real-valued (e.g., raw sensor input)
- Output can be discrete or real-valued
- Output can be a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability of result is unimportant

Examples:

- Speech recognition (now the standard method)
- Image classification (also now the standard method)
- many others . . .

Gradient Descent

To understand, consider simpler *linear unit*, where

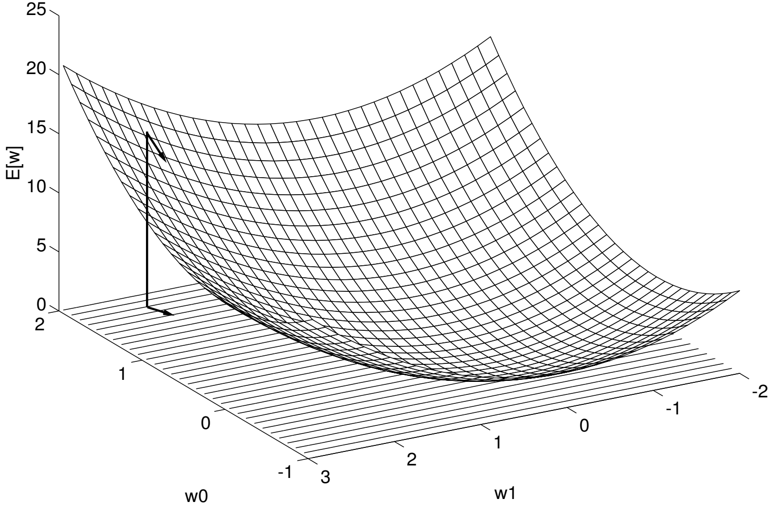
$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

Let's learn w_i's that minimize the squared error

$$E[\mathbf{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

Where D is set of training examples

Gradient Descent



Gradient Descent

Gradient

$$\nabla E[\mathbf{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n} \right]$$

Gradient vector gives direction of *steepest increase* in error E

Negative of the gradient, i.e., *steepest decrease*, is what we want

Training rule:

$$\Delta \mathbf{w} = -\eta \nabla E[\mathbf{w}]$$

i.e.,

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Gradient Descent

$$
\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \mathbf{w}\cdot\mathbf{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d})
\end{aligned}
$$

Gradient Descent

```
GRADIENT-DESCENT($training\_examples, \eta$)
*Each training example is a pair $\langle \mathbf{x}, t \rangle$, where $\mathbf{x}$ is the vector of input values, and $t$ is the target output value. $\eta$ is the learning rate (e.g., .05).*

Initialize each $w_i$ to some small random value

Until the termination condition is met, Do
    Initialize each $\Delta w_i$ to zero
    For each $\langle \mathbf{x}, t \rangle$ in $training\_examples$, Do
        Input the instance $\mathbf{x}$ to the unit and compute the output $o$
        For each linear unit weight $w_i$
            $\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$
    For each linear unit weight $w_i$
            $w_i \leftarrow w_i + \Delta w_i$
```

Training Perceptron vs. Linear unit

Perceptron training rule guaranteed to succeed if

- Training examples are linearly separable
- Sufficiently small learning rate η

Linear unit training rule uses gradient descent

- Guaranteed to converge to hypothesis with minimum squared error
- Given sufficiently small learning rate η
- Even when training data contains noise
- Even when training data not separable by H

Incremental (Stochastic) Gradient Descent

Batch mode Gradient Descent:
Do until satisfied

- Compute the gradient $\nabla E_D[\mathbf{w}]$
- $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E_D[\mathbf{w}]$

Incremental mode Gradient Descent:
Do until satisfied

- For each training example d in D
    - Compute the gradient $\nabla E_d[\mathbf{w}]$
    - $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E_d[\mathbf{w}]$

Incremental (Stochastic) Gradient Descent

Batch:

$$E_D[\mathbf{w}] \equiv \frac{1}{2}\sum_{d\in D}(t_d - o_d)^2$$

Incremental:

$$E_d[\mathbf{w}] \equiv \frac{1}{2}(t_d - o_d)^2$$

Incremental or *Stochastic Gradient Descent* (SGD) can approximate *Batch Gradient Descent* arbitrarily closely, if η made small enough

Very useful for training large networks, or online learning from data streams

Stochastic implies examples should be selected at random

Multilayer Networks of Sigmoid Units

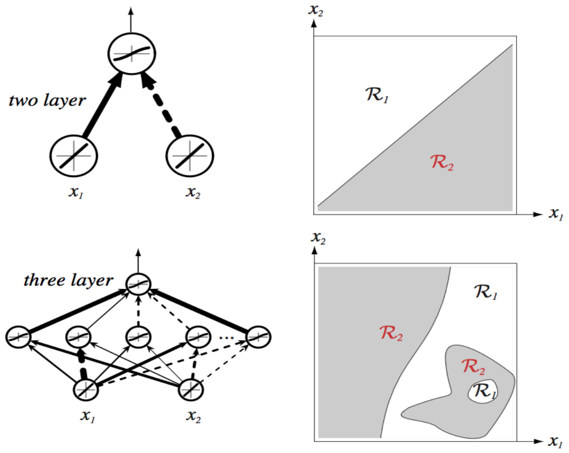
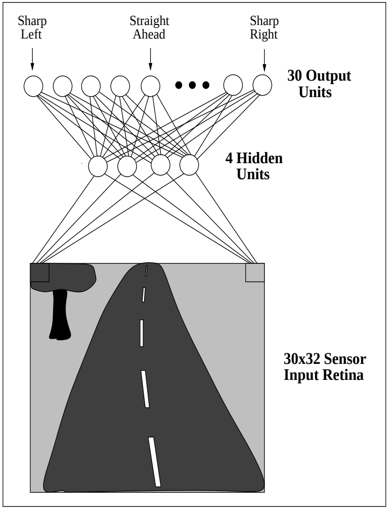
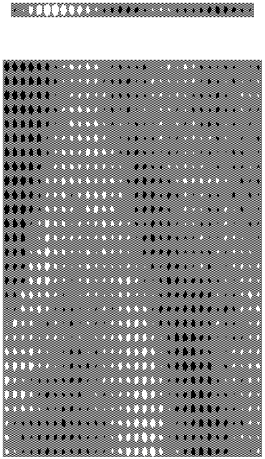


FIGURE 6.3. Whereas a two-layer network classifier can only implement a linear decision boundary, given an adequate number of hidden units, three-, four- and higher-layer networks can implement arbitrary decision boundaries. The decision regions need not be convex or simply connected. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

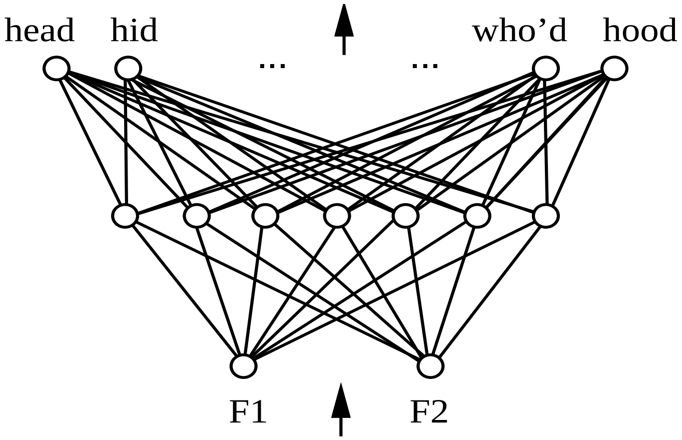
ALVINN drives 70 mph on highways

ALVINN

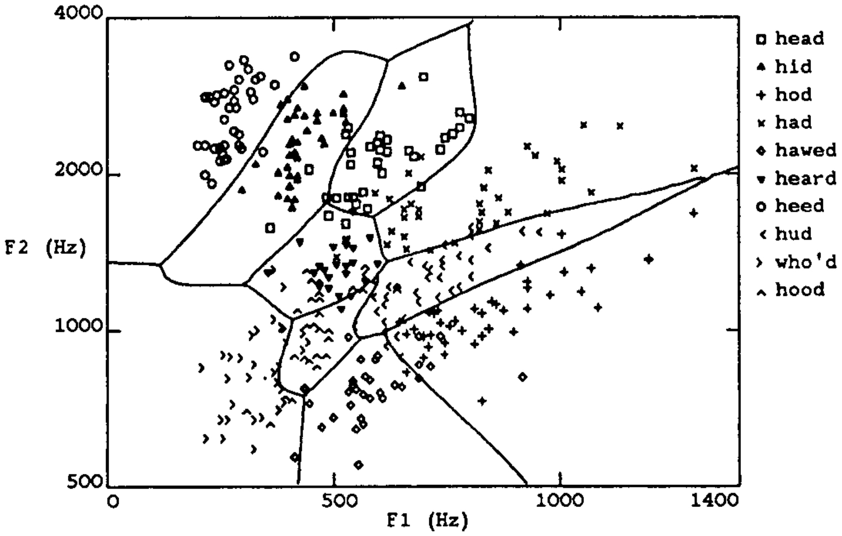

Sharp Left
Straight Ahead
Sharp Right
30 Output Units
4 Hidden Units
30x32 Sensor Input Retina



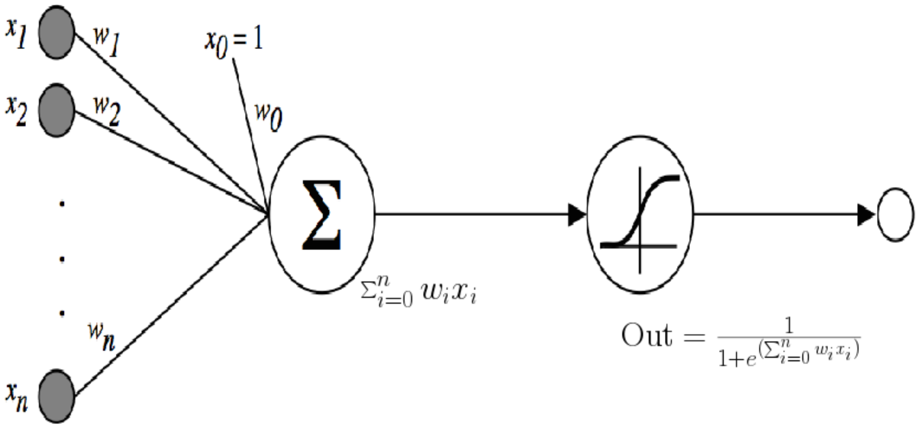
A Multilayer Perceptron for Speech Recognition: Model



A Multilayer Perceptron for Speech Recognition: Decision Boundaries



Sigmoid Unit



x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$\Sigma_{i=0}^{n} w_i x_i$
$\text{Out} = \frac{1}{1+e^{(\Sigma_{i=0}^{n} w_i x_i)}}$

Sigmoid Unit

Same as a perceptron except that the step function has been replaced by a smoothed version, a sigmoid function.

Note: in practice, particularly for deep networks, sigmoid functions are less common than other non-linear activation functions that are easier to train, but sigmoids are mathematically convenient.

Sigmoid Unit

Why use the sigmoid function $\sigma(x)$?

$$\frac{1}{1+e^{-x}}$$

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x))$

We can derive gradient descent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units $\rightarrow$ Backpropagation

Start by assuming we want to minimize squared error $\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2$ over a set of training examples D.

Error Gradient for a Sigmoid Unit

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
&= \sum_d (t_d - o_d) \left(- \frac{\partial o_d}{\partial w_i} \right) \\
&= - \sum_d (t_d - o_d) \frac{\partial o_d}{\partial net_d} \frac{\partial net_d}{\partial w_i}
\end{aligned}$$

Error Gradient for a Sigmoid Unit

But we know:

$$\frac{\partial o_d}{\partial net_d} = \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1 - o_d)$$

$$\frac{\partial net_d}{\partial w_i} = \frac{\partial (\mathbf{w} \cdot \mathbf{x}_d)}{\partial w_i} = x_{i,d}$$

So:

$$\frac{\partial E}{\partial w_i} = -\sum_{d \in D} (t_d - o_d) o_d (1 - o_d) x_{i,d}$$

Backpropagation Algorithm

```
Initialize all weights to small random numbers.
Until satisfied, Do
  For each training example, Do
    Input the training example to the network and
        compute the network outputs
    For each output unit $k$
        $\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$
    For each hidden unit $h$
        $\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k$
    Update each network weight $w_{ji}$
        $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$
      where
        $\Delta w_{ji} = \eta \delta_j x_{ji}$
```

More on Backpropagation

A solution for learning highly complex models . . .

- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Can learn probabilistic models by maximising likelihood

Minimizes error over *all* training examples

- Training can take thousands of iterations $\rightarrow$ slow!
- Using network after training is very fast

More on Backpropagation

Will converge to a local, not necessarily global, error minimum

- May be many such local minima
- In practice, often works well (can run multiple times)
- Often include weight *momentum* α

$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1)$$

- Stochastic gradient descent using “mini-batches”

Nature of convergence

- Initialize weights near zero
- Therefore, initial networks near-linear
- Increasingly non-linear functions possible as training progresses

More on Backpropagation

Models can be very complex

- Will network generalize well to subsequent examples?
    - may *underfit* by stopping too soon
    - may *overfit* . . .

Many ways to regularize network, making it less likely to overfit

- Add term to error that increases with magnitude of weight vector

$$E(\mathbf{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$

- Other ways to penalize large weights, e.g., weight decay
- Using "tied" or shared set of weights, e.g., by setting all weights to their mean after computing the weight updates
- Many other ways . . .

Expressive Capabilities of ANNs

Boolean functions:

- Every Boolean function can be represented by network with single hidden layer
- but might require exponential (in number of inputs) hidden units

Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one 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].

Being able to approximate any function is one thing, being able to *learn* it is another . . .

How complex should the model be ?

With four parameters I can fit an elephant, and with five I can make him wiggle his trunk.

John von Neumann

"Goodness of fit" in ANNs

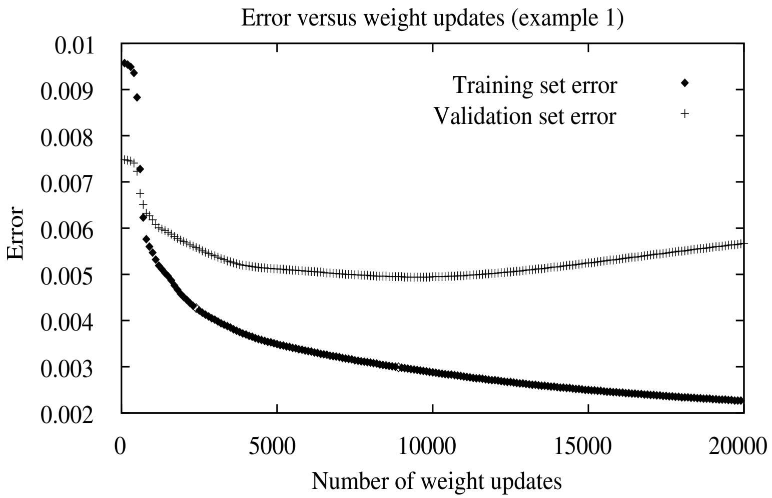
Can neural networks overfit/underfit ?

Next two slides: plots of "learning curves" for error as the network learns (shown by number of weight updates) on two different robot perception tasks.

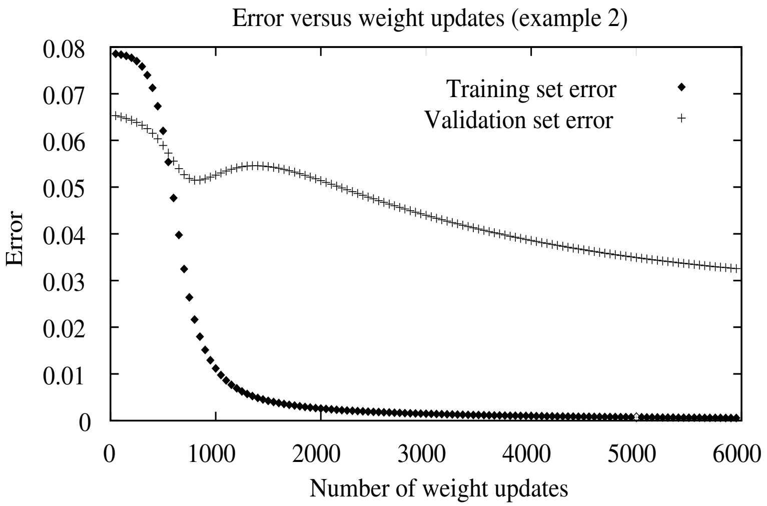
Note difference between training set and off-training set (validation set) error on both tasks !

Note also that on second task validation set error continues to decrease after an initial increase — any regularisation (network simplification, or weight reduction) strategies need to avoid early stopping (underfitting).

Overfitting in ANNs



Underfitting in ANNs



Neural networks for classification

Sigmoid unit computes output $o(\mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x})$

Output ranges from 0 to 1

Example: binary classification

$$o(\mathbf{x}) = \begin{cases} \text{predict class } 1 & \text{if } o(\mathbf{x}) \geq 0.5 \\ \text{predict class } 0 & \text{otherwise.} \end{cases}$$

Questions:

- what error (loss) function should be used ?
- how can we train such a classifier ?

Neural networks for classification

Minimizing square error (as before) does not work so well for classification

If we take the output $o(\mathbf{x})$ as the *probability* of the class of $\mathbf{x}$ being 1, the preferred loss function is the *cross-entropy*

$$-\sum_{d \in D} t_d \log o_d + (1 - t_d) \log (1 - o_d)$$

where:

$t_d \in \{0, 1\}$ is the class label for training example d, and o_d is the output of the sigmoid unit, interpreted as the probability of the class of training example d being 1.

To train sigmoid units for classification using this setup, can use *gradient ascent* with a similar weight update rule as that used to train neural networks by gradient descent – this will yield the *maximum likelihood* solution.

A practical application: Face Recognition

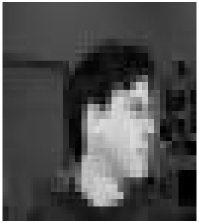
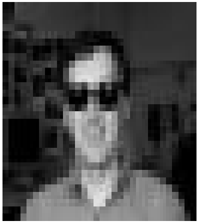
Dataset: 624 images of faces of 20 different people.

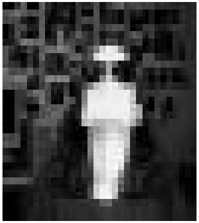
- image size 120x128 pixels
- grey-scale, 0-255 pixel value range
- different poses
- different expressions
- wearing sunglasses or not

Raw images compressed to 30x32 pixels, each is mean of 4x4 pixels.

MLP structure: 960 inputs $\times$ 3 hidden nodes $\times$ 4 output nodes.

Neural Nets for Face Recognition - Task

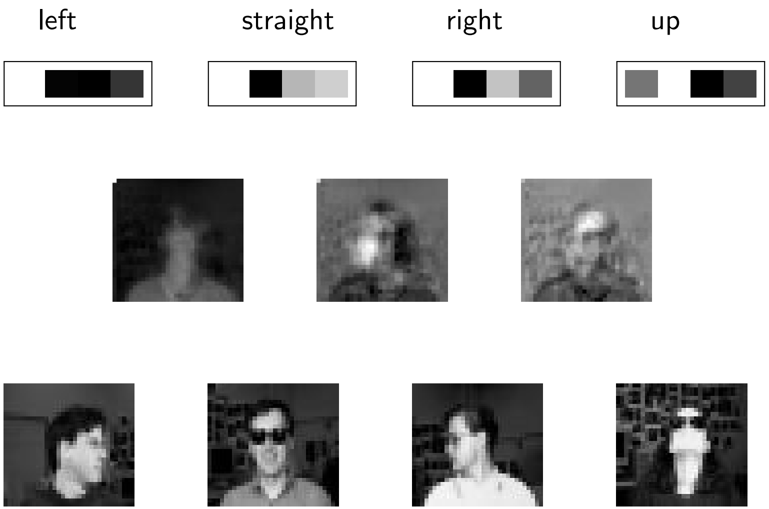





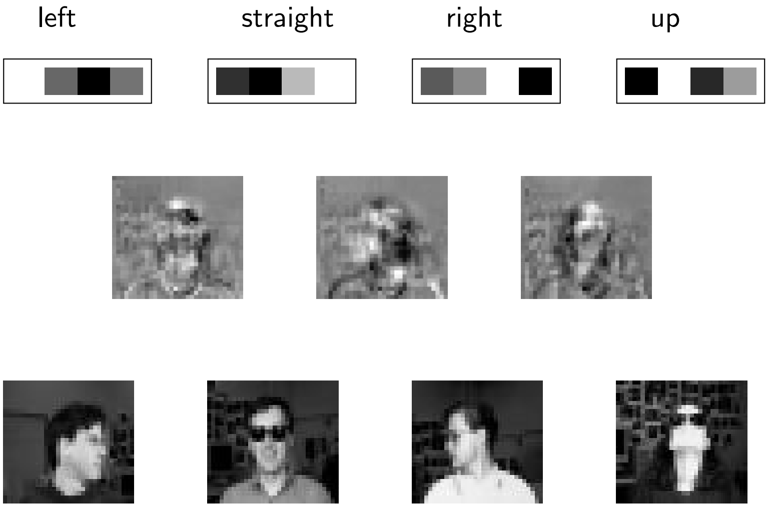
left straight right up

Four pose classes: looking left, straight ahead, right or upwards.
Use a 1-of-n encoding: more parameters; can give confidence of prediction.
Selected single hidden layer with 3 nodes by experimentation.

Neural Nets for Face Recognition - after 1 epoch



Neural Nets for Face Recognition - after 100 epochs



Neural Nets for Face Recognition - Results

Each output unit (left, straight, right, up) has four weights, shown by dark (negative) and light (positive) blocks.

Leftmost block corresponds to the bias (threshold) weight

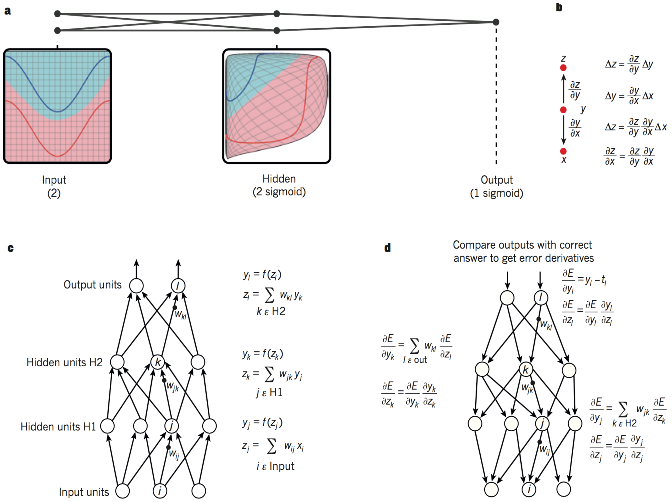
Weights from each of 30x32 image pixels into each hidden unit are plotted in position of corresponding image pixel.

Classification accuracy: 90% on test set (default: 25%)

Question: what has the network learned ?

For code, data, etc. see `http://www.cs.cmu.edu/~tom/faces.html`

Deep Learning


a
Input (2)
Hidden (2 sigmoid)
Output (1 sigmoid)
b
z
y
x
$\frac{\partial z}{\partial y}$
$\frac{\partial y}{\partial x}$
$\Delta z = \frac{\partial z}{\partial y}\Delta y$
$\Delta y = \frac{\partial y}{\partial x}\Delta x$
$\Delta z = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\Delta x$
$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$
c
Output units
Hidden units H2
Hidden units H1
Input units
l k j i
w_{kl} w_{jk} w_{ij}
$y_l = f(z_l)$
$z_l = \sum_{k \in H2} w_{kl} y_k$
$y_k = f(z_k)$
$z_k = \sum_{j \in H1} w_{jk} y_j$
$y_j = f(z_j)$
$z_j = \sum_{i \in Input} w_{ij} x_i$
d
Compare outputs with correct answer to get error derivatives
$\frac{\partial E}{\partial y_l} = y_l - t_l$
$\frac{\partial E}{\partial z_l} = \frac{\partial E}{\partial y_l}\frac{\partial y_l}{\partial z_l}$
$\frac{\partial E}{\partial y_k} = \sum_{l \in out} w_{kl}\frac{\partial E}{\partial z_l}$
$\frac{\partial E}{\partial z_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial z_k}$
$\frac{\partial E}{\partial y_j} = \sum_{k \in H2} w_{jk}\frac{\partial E}{\partial z_k}$
$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j}$

Y. Lecun et al. (2015) Nature (521) 436–444.

Deep Learning

Deep learning is a vast area that has exploded in the last 15 years.

Beyond scope of this course to cover in detail.

See: "Deep Learning" I. Goodfellow *et al.* (2017) – there is an online copy freely available.

Course COMP9444 Neural Networks (next semester).

Deep Learning

Question: How much of what we have seen carries over to deep networks ?

Answer: Most of the basic concepts.

We mention some important issues that differ in deep networks.

Deep Learning: Architectures

Most successful deep networks *do not* use the fully connected network architecture we outlined above.

Instead, they use more specialised architectures for the application of interest (*inductive bias*).

Example: Convolutional neural nets (CNNs) have an alternating layer-wise architecture inspired by the brain's visual cortex. Works well for image processing tasks, but also for applications like text processing.

Example: Long short-term memory (LSTM) networks have recurrent network structure designed to capture long-range dependencies in *sequential* data, as found, e.g., in natural language (although now often superseded by *transformer* architectures).

Example: Autoencoders are are kind of *unsupervised* learning method. They learn a mapping from input examples to *the same examples as output* via a compressed (lower dimension) hidden layer, or layers.

Deep convolutional networks learn features

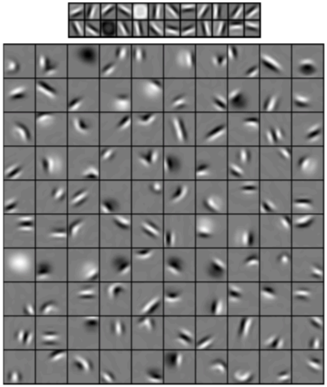
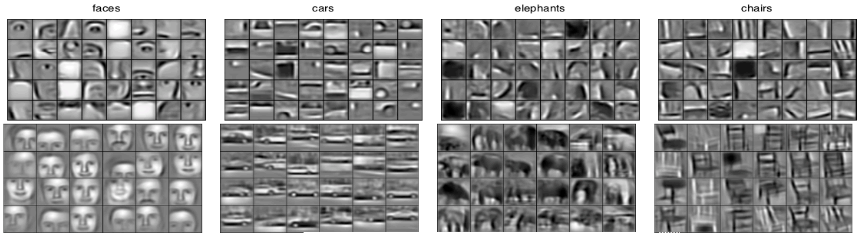


Figure 2. The first layer bases (top) and the second layer bases (bottom) learned from natural images. Each second layer basis (filter) was visualized as a weighted linear combination of the first layer bases.

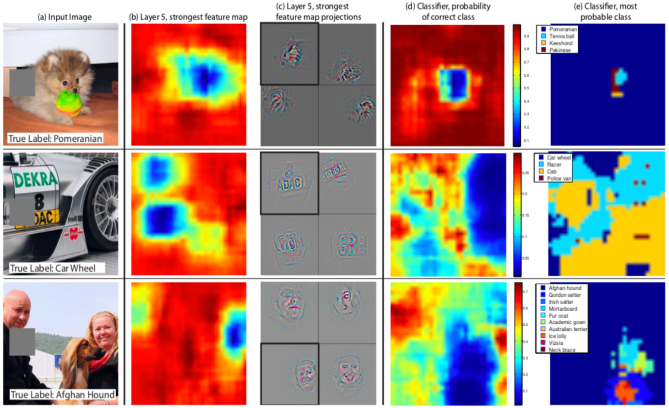
H. Lee et al. (2009) ICML.

Deep convolutional networks learn features



H. Lee et al. (2009) ICML.

Deep convolutional networks learn features



M. Zeiler & R. Fergus (2014) ECCV.

Deep Learning: Activation Functions

Problem: in very large networks, sigmoid activation functions can *saturate*, i.e., can be driven close to 0 or 1 and then the gradient becomes almost 0 – effectively halts updates and hence learning for those units.

Solution: use activation functions that are non-saturating., e.g., "Rectified Linear Unit" or ReLu, defined as $f(x) = \max(0, x)$.

Problem: sigmoid activation functions are not zero-centred, which can cause gradients and hence weight updates become "non-smooth".

Solution: use zero-centred activation function, e.g., $tanh$, with range $[-1, +1]$. Note that $tanh$ is essentially a re-scaled sigmoid.

Derivative of a ReLu is simply

$$\frac{\partial f}{\partial x} = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{otherwise.} \end{cases}$$

Deep Learning: Regularization

Deep networks can have millions or billions or parameters.

Hard to train, prone to overfit.

What techniques can help ?

Example: dropout

- for each unit u in the network, with probability p, “drop” it, i.e., ignore it and its adjacent edges during training
- this will simplify the network and prevent overfitting
- can take longer to converge
- but will be quicker to update on each epoch
- also forces exploration of different sub-networks formed by removing p of the units on any training run

Back-propagation and computational graphs

See the accompanying hand-written notes, based on Strang (2019).

Summary

- ANNs since 1940s; popular in 1980s, 1990s; recently a revival

  Complex function fitting. Generalise core techniques from machine learning and statistics based on linear models for regression and classification.

  Learning is typically stochastic gradient descent. Networks are too complex to fit otherwise.

  Many open problems remain. How are these networks actually learning ? How can they be improved ? What are the limits to neural learning ?

Strang, G. (2019). *Linear Algebra and Learning from Data*. Wellesley - Cambridge Press.