Neural Learning (1)

Never Stand Still

COMP9417 Machine Learning & Data Mining

Term 1, 2020

Lecturer: Dr Yang Song

Adapted from slides by Dr Michael Bain

Aims

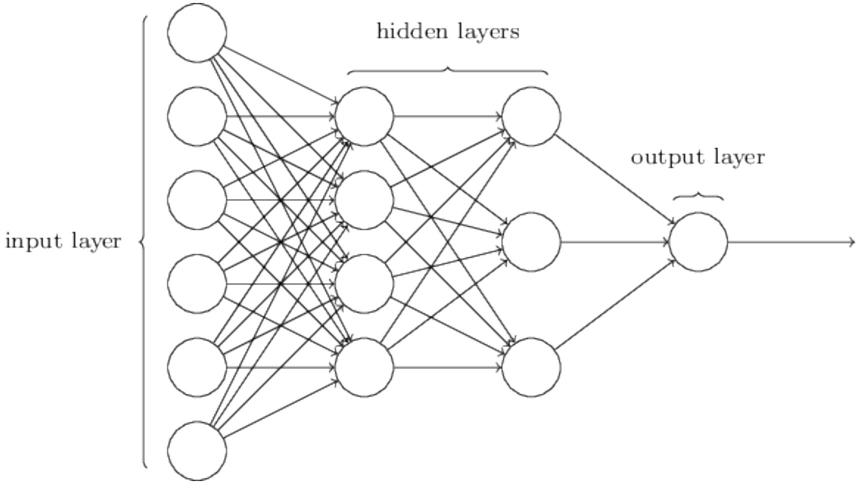
- Neural Network Learning
  - describe Perceptrons and how to train them
  - relate neural learning to optimization in machine learning
  - outline the problem of neural learning
  - derive the method of gradient descent for linear models
  - describe the problem of non-linear models with neural networks
  - outline the method of back-propagation training of a multi-layer
  - perceptron neural network
  - describe the application of neural learning for classification

Artificial Neural Networks

Artificial Neural Networks are inspired by human nervous system

NNs are composed of a large number of interconnected processing elements known as neurons

They use supervised error correcting rules with back-propagation to learn a specific task



http://statsmaths.github.io/stat665/lectures/lec12/lecture12.pdf

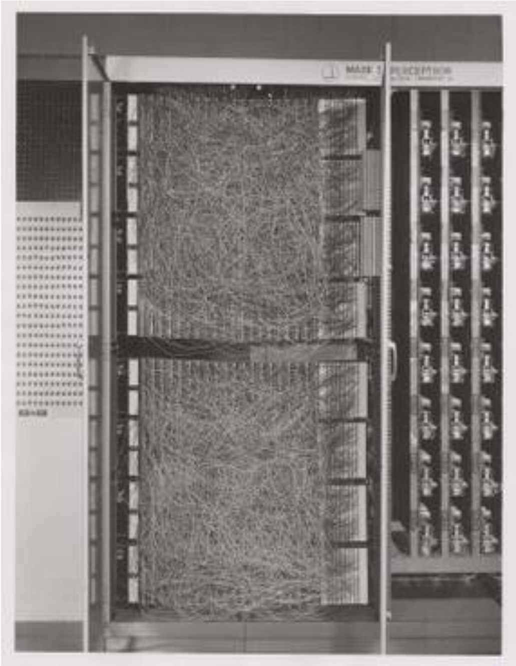
Perceptron

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

UNSW AUSTRALIA

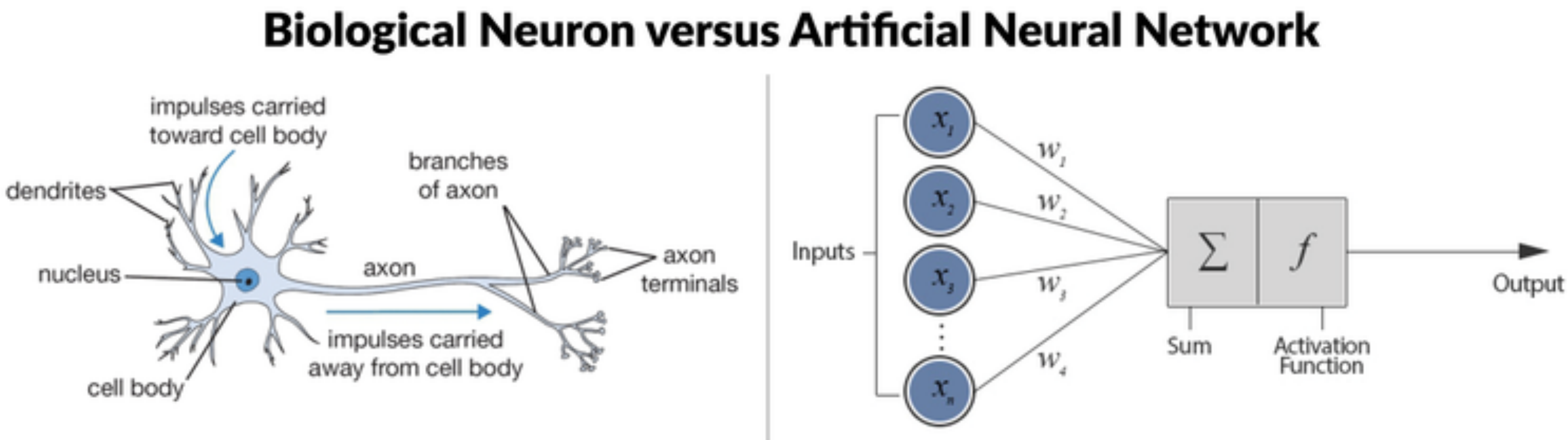
Perceptron

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

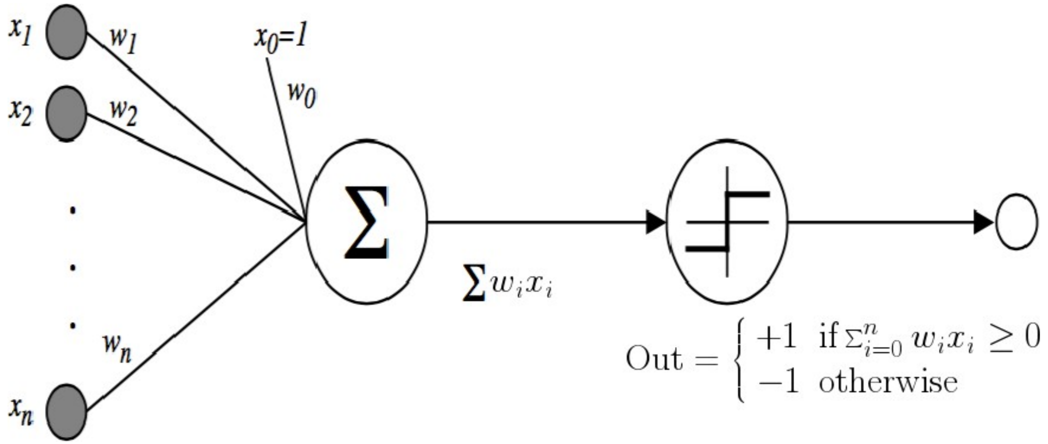


突触 轴突

Each neuron has multiple dendrites and a single axon. The neuron receives its inputs from its dendrites and transmits its output through its axon. Both inputs and outputs take the form of electrical impulses. The neuron sums up its inputs, and if the total electrical impulse strength exceeds the neuron's firing threshold, the neuron fires off a new impulse along its single axon. The axon, in turn, distributes the signal along its branching synapses which collectively reach thousands of neighboring neurons.

https://towardsdatascience.com/from-fiction-to-reality-a-beginners-guide-to-artificial-neural-networks-d0411777571b

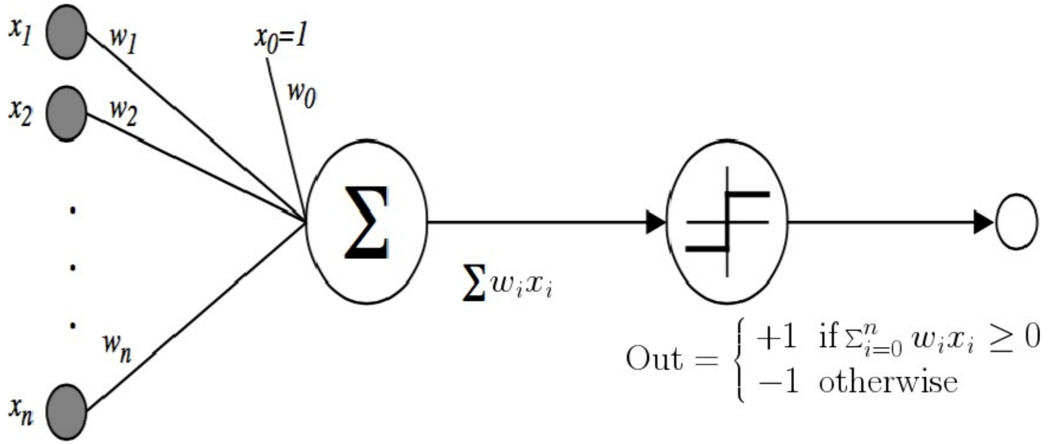
Perceptron



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

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

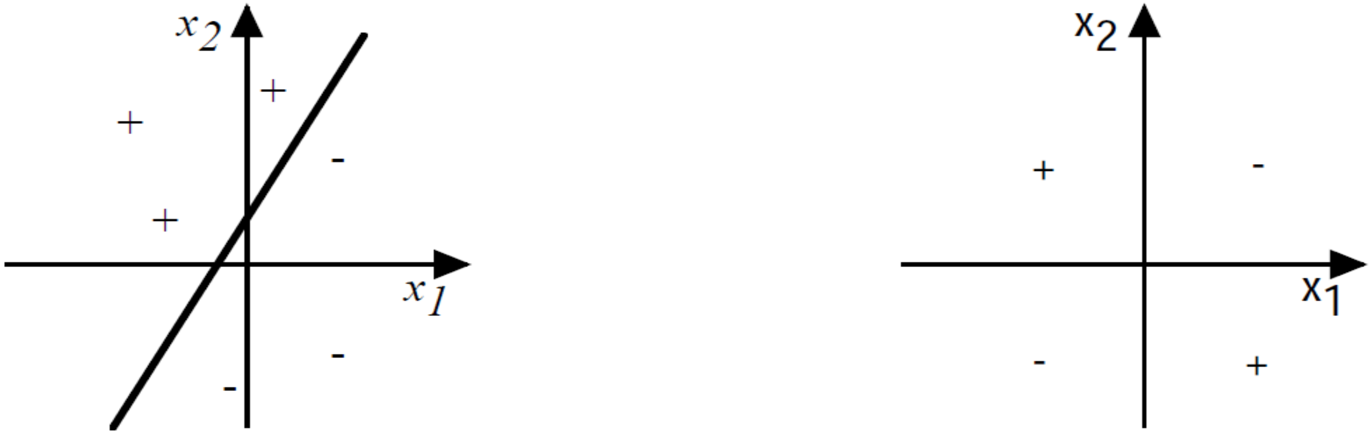
Perceptron



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



(a) (b)

Perceptron is able to represent some useful functions

- (a) Linearly separable
- (b) XOR – functions that are not linearly separable are not representable

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.

- Let $\mathbf{x}_i$ be a misclassified positive example, then we have $y_i = +1$ and $\mathbf{w} \cdot \mathbf{x}_i < 0$. 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 $\mathbf{w} \cdot \mathbf{x}_j > 0$. 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.

Training Perceptron

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 $\hat{y} = \text{sign}(\mathbf{w} \cdot \mathbf{x})$.

$\mathbf{w} \leftarrow \mathbf{0}$ // Other initialisations of the weight vector are possible

$converged \leftarrow$ false

while $converged =$ false **do**

    $converged \leftarrow$ true

    for $i = 1$ to $|D|$ **do**

        if $y_i \mathbf{w} \cdot \mathbf{x}_i \leq 0$ **then** // i.e., $\hat{y_i} \neq y_i$

            $\mathbf{w} \leftarrow \mathbf{w} + \eta y_i \mathbf{x}_i$

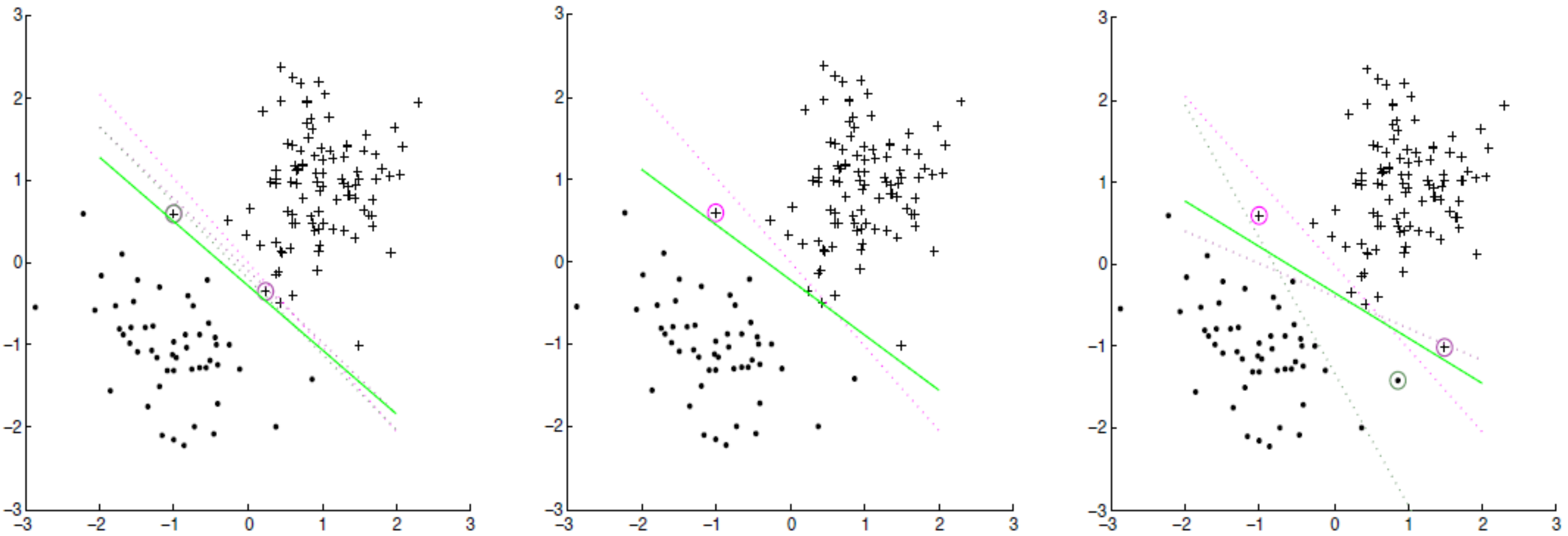
            $converged \leftarrow$ false // We changed $\mathbf{w}$ so haven't converged yet

        end

    end

end

Perceptron 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.

UNSW AUSTRALIA

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

UNSW AUSTRALIA

Perceptron Convergence

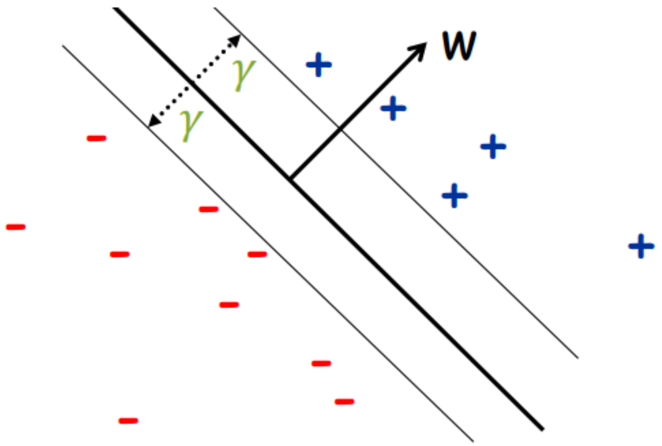
perceptron convergence: just a kind of proof of perceptron will converge at the end!

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.

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

γ is typically referred to as the "margin"

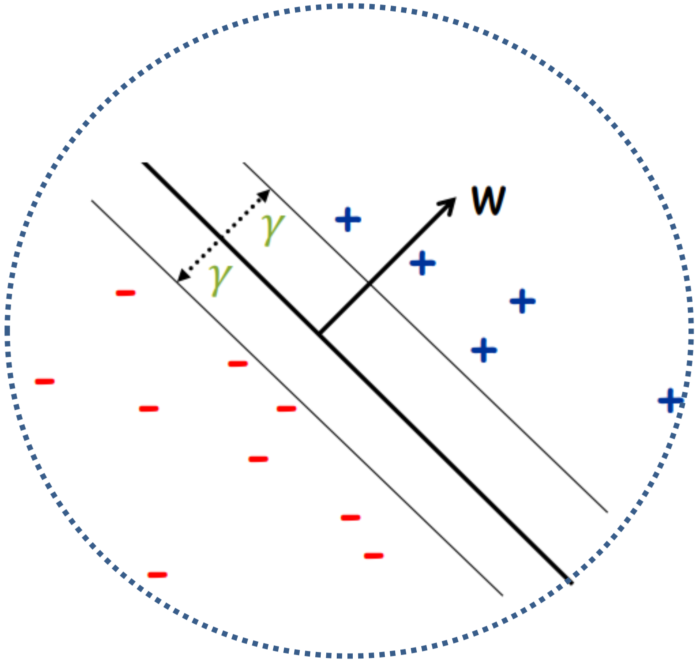


Perceptron Convergence

Perceptron Convergence Theorem (Novikoff, 1962)

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

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



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 relatively minor modification many perceptrons can be combined together to form one model
    - *multilayer perceptrons*, the classic "neural network"

Optimisation

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

A general optimisation 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 minimise a real-valued function f

Note: convergence criteria will be problem-specific.

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

Optimisation

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

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

Gradient Descent

To understand, consider the simple linear unit, where

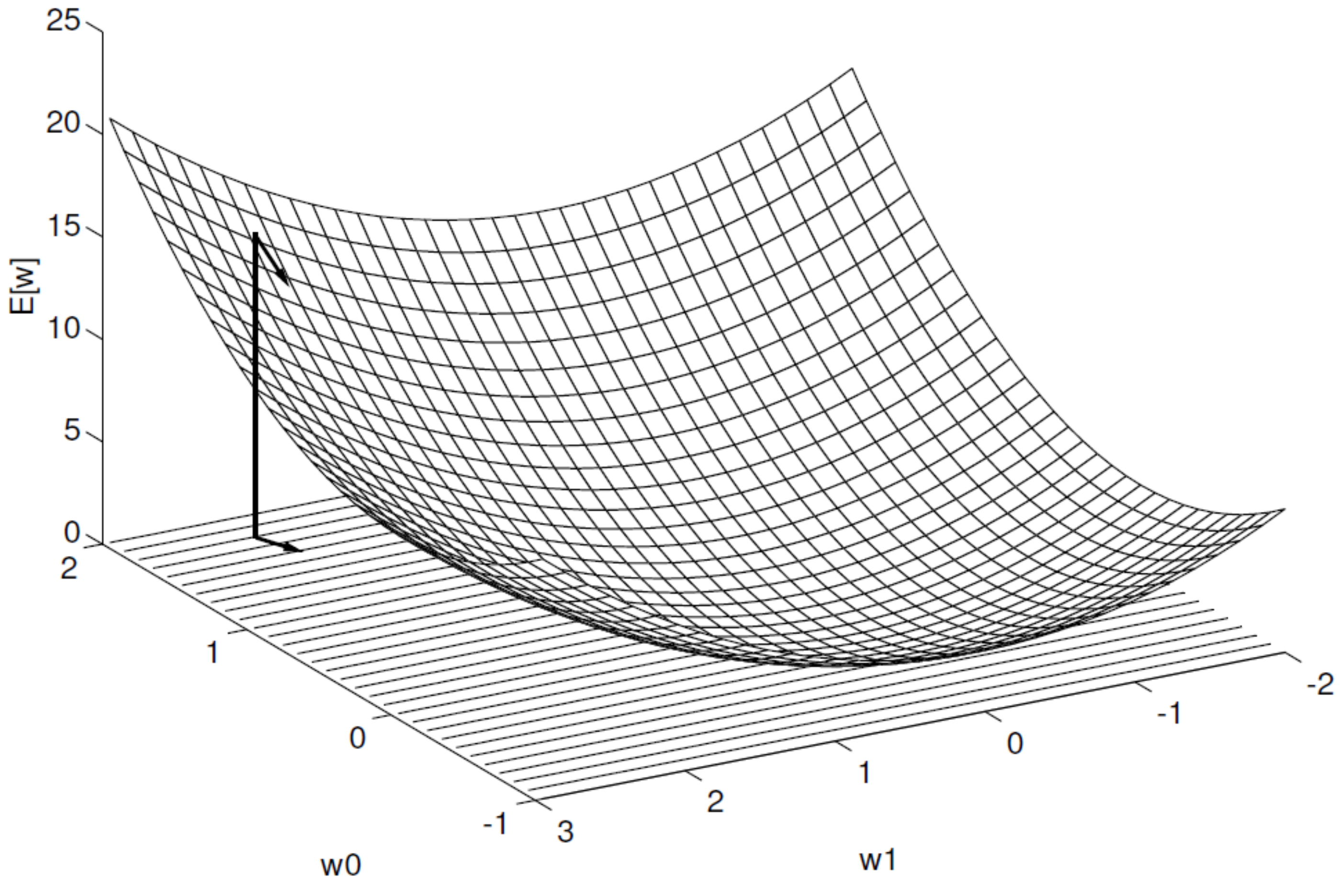
$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

Let's learn w_i that minimise the squared error

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

Where D is the set of training samples

Gradient Descent



UNSW AUSTRALIA

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

UNSW AUSTRALIA

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.* η *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 Δw_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

      $w_i \leftarrow w_i + \Delta w_i$

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

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

Incremental mode Gradient Descent:

Do until satisfied

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

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

UNSW AUSTRALIA

Incremental (Stochastic) Gradient Descent

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

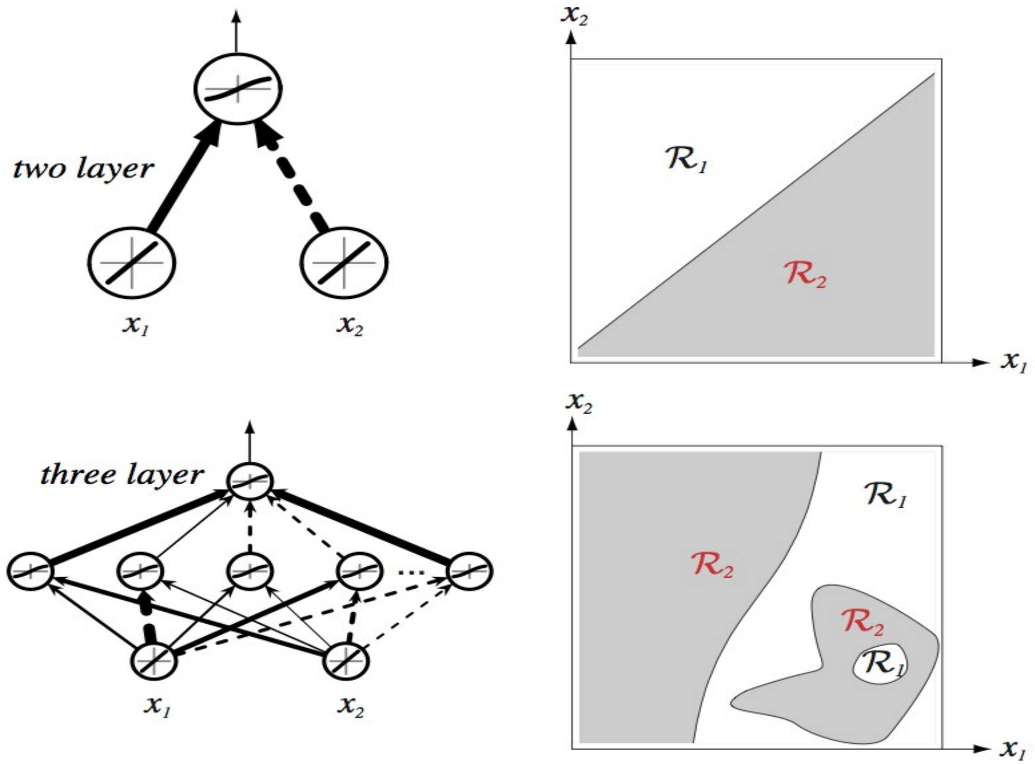


FIGURE 6.3. Whereas a two-layer network classifier can only implement a linear decision boundary, given an adequate number of hidden units, three-, four- and higher-layer networks can implement arbitrary decision boundaries. The decision regions need not be convex or simply connected. From: Richard O.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

Multilayer Networks

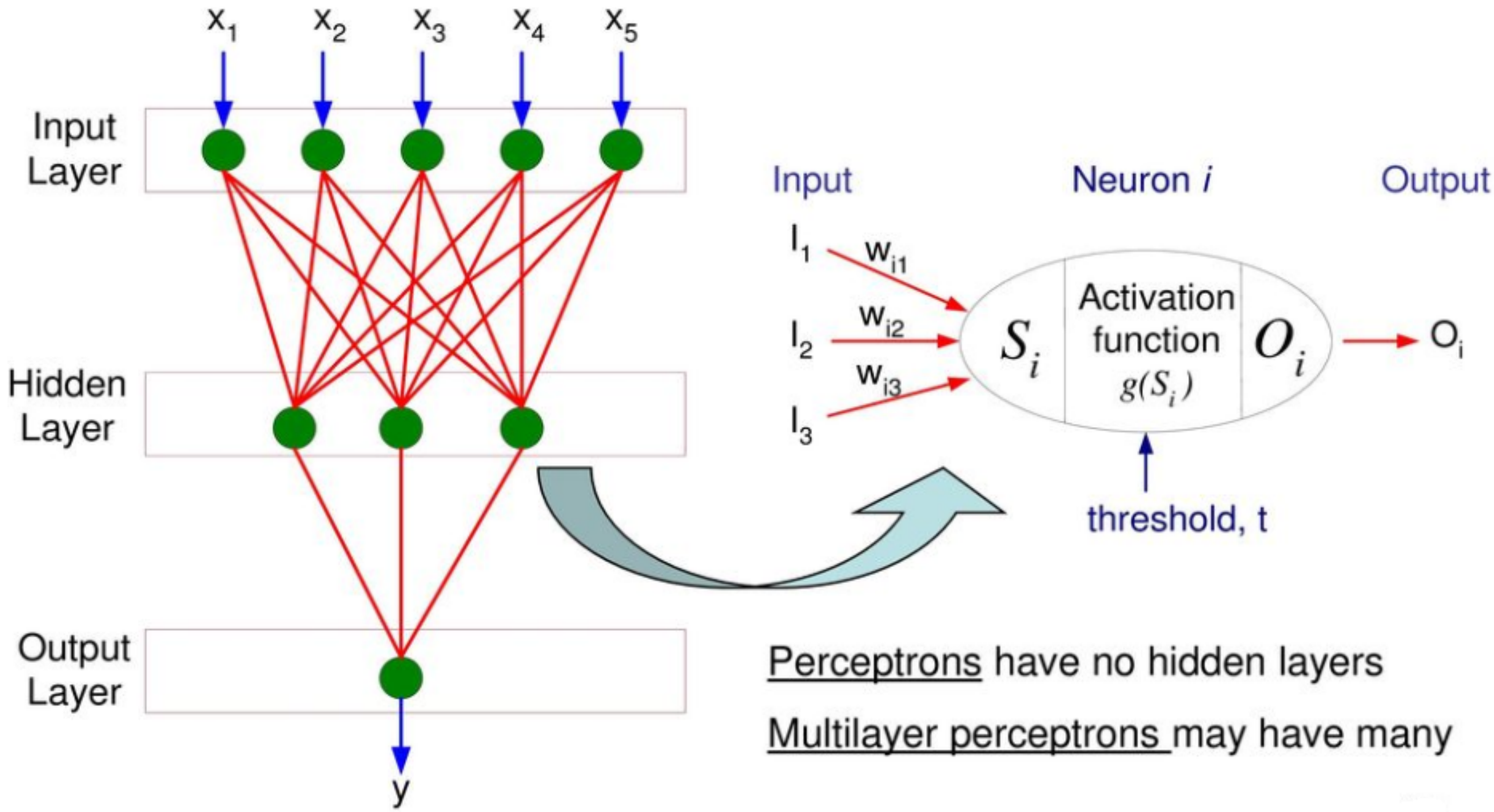
Multi-layer networks can represent arbitrary functions, but an effective learning algorithm for such networks was thought to be difficult

A typical multi-layer network consists of an input, hidden and output layer, each fully connected to the next, with activation feeding forward

The weights determine the function computed.

Given an arbitrary number of hidden units, any boolean function can be computed with a single hidden layer

General Structure of ANN



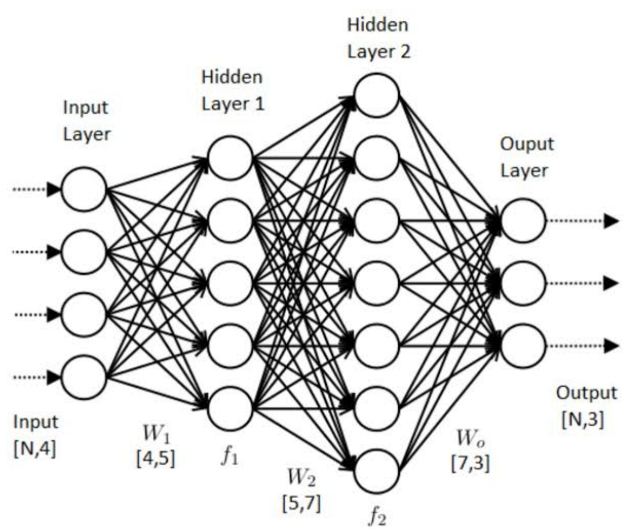
<u>Perceptrons</u> have no hidden layers

<u>Multilayer perceptrons</u> may have many

General Structure of ANN

Properties of Artificial Neural Networks (ANN’s):

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



Artificial Neural Network (Source: VIASAT)

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 not important

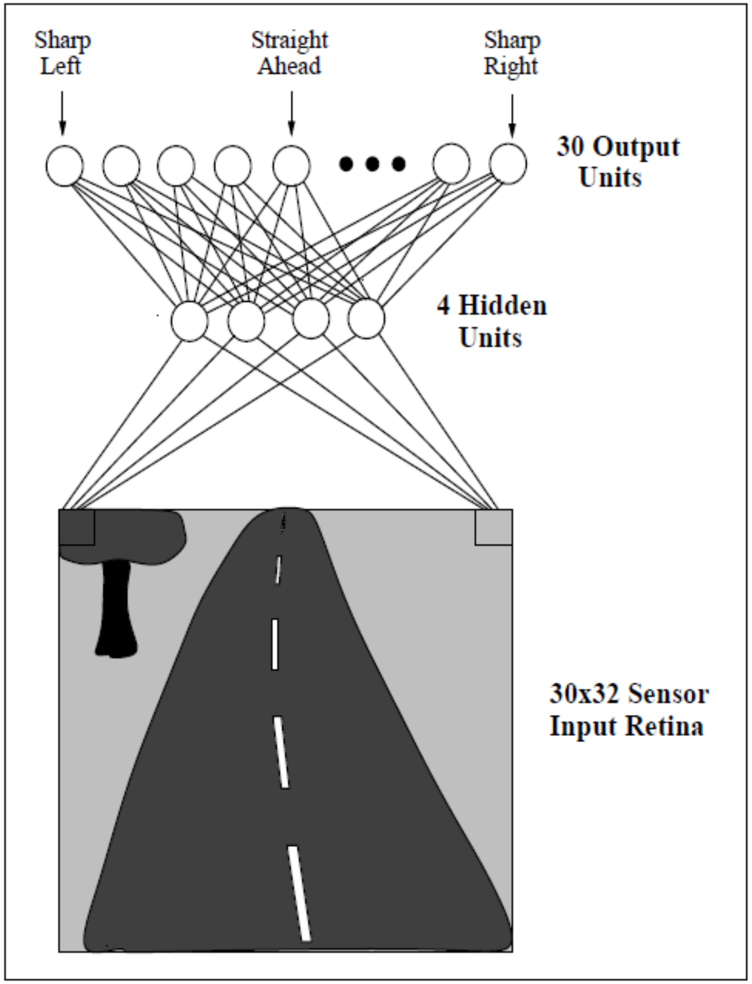
Examples:

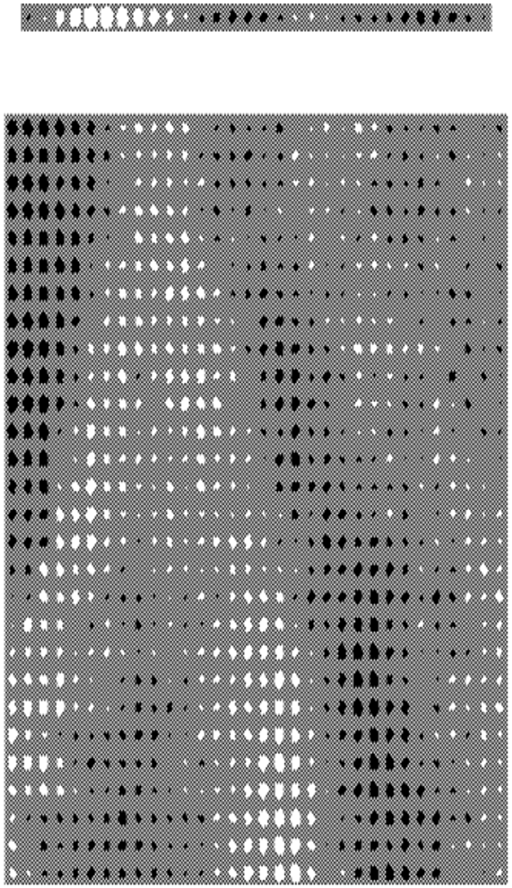
- Speech recognition
- Image classification
- many others . . .

ALVINN drives 70 mph on highways

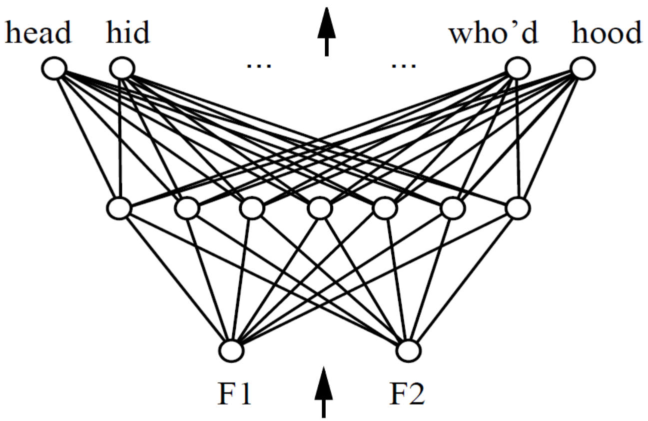
UNSW AUSTRALIA

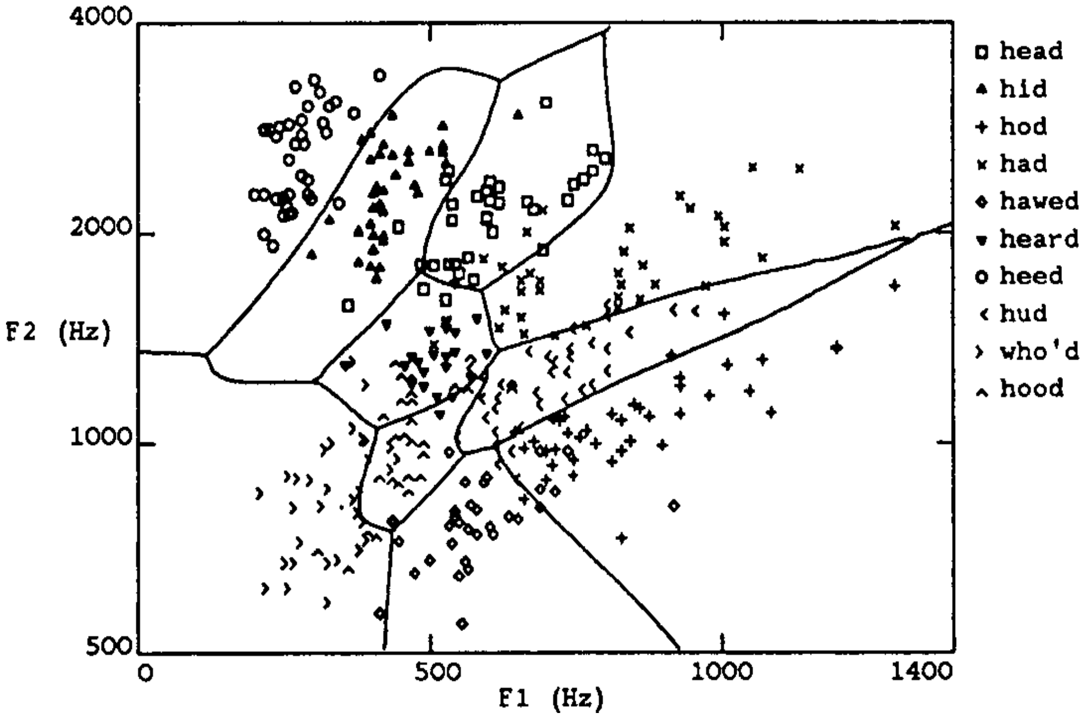
ALVINN


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



MLP Speech Recognition

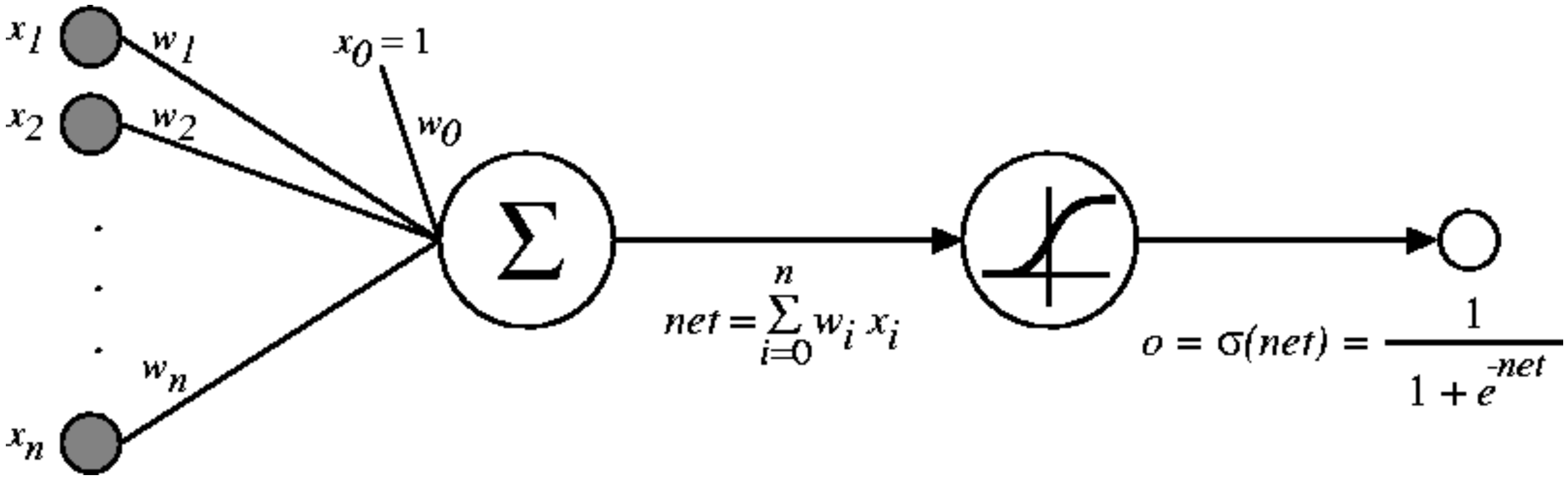




Decision Boundaries

Sigmoid Unit

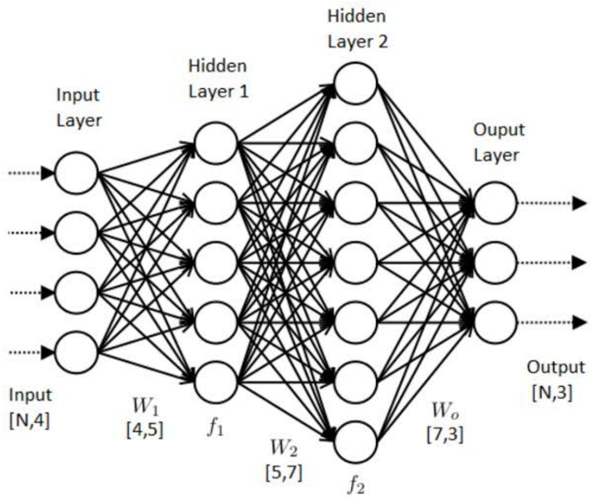
S形的



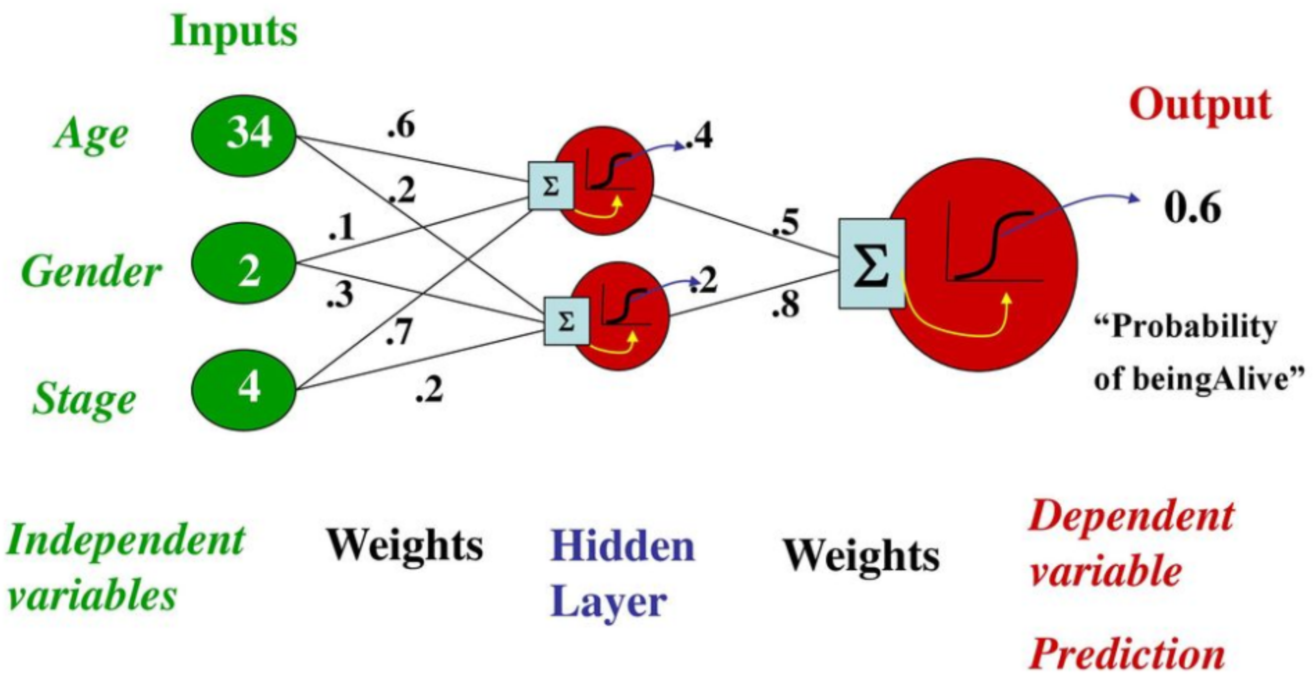
Sigmoid Unit

Same as a perceptron except that the step function has been replaced by a nonlinear sigmoid function.

Nonlinearity makes it easy for the model to generalise or adapt with variety of data and to differentiate between the output.



Sigmoid Unit



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 → Backpropagation 反向传播

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.

Error Gradient of Sigmoid Unit

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

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

Error Gradient of Sigmoid Unit

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

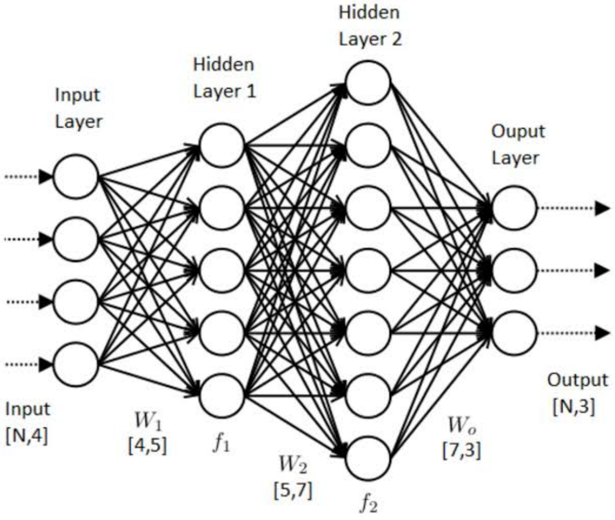
$$\delta_h \leftarrow o_h(1-o_h)\sum_{k \in outputs} w_{kh}\delta_k$$

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 generalised to arbitrary directed graphs
- Can learn probabilistic models by maximising likelihood

Minimises error over *all* training examples

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

More on Backpropagation

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

- Might exist 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

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

More on Backpropagation

Models can be very complex

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

Many ways to regularise 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 penalise 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 a network with single hidden layer
  指数的
- but might require exponential (in number of inputs) hidden units

Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by a 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]

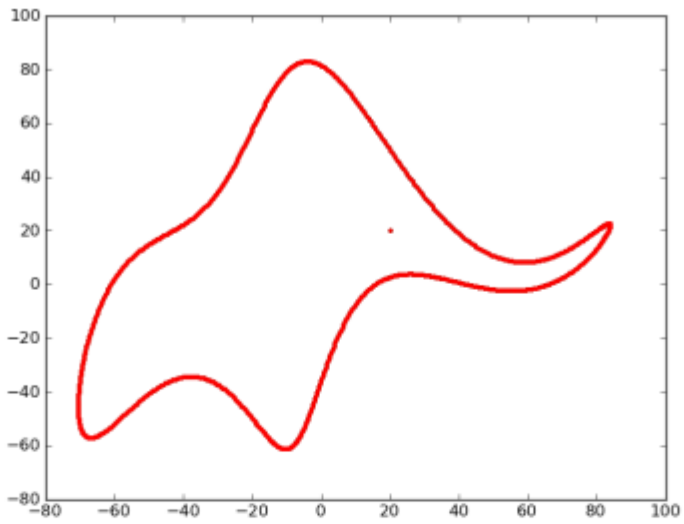
Being able to approximate any function is one thing, being able to *learn* it is another . . .

How complex should the model be?

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

摆动

John von Neumann



“Goodness of fit” in ANNs

Can neural networks overfit/underfit ?

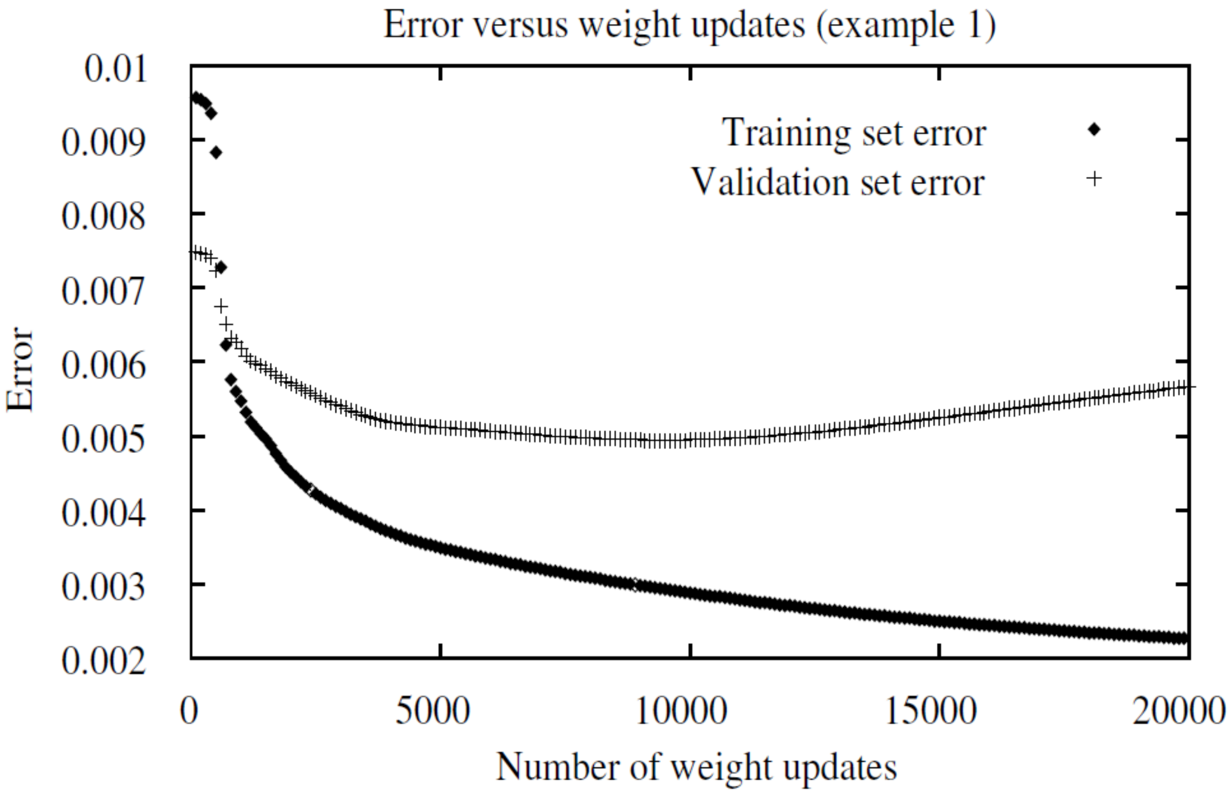
Next two slides: plots of “learning curves” for error as the network learns (shown by number of weight updates) on two different robot perception tasks.

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

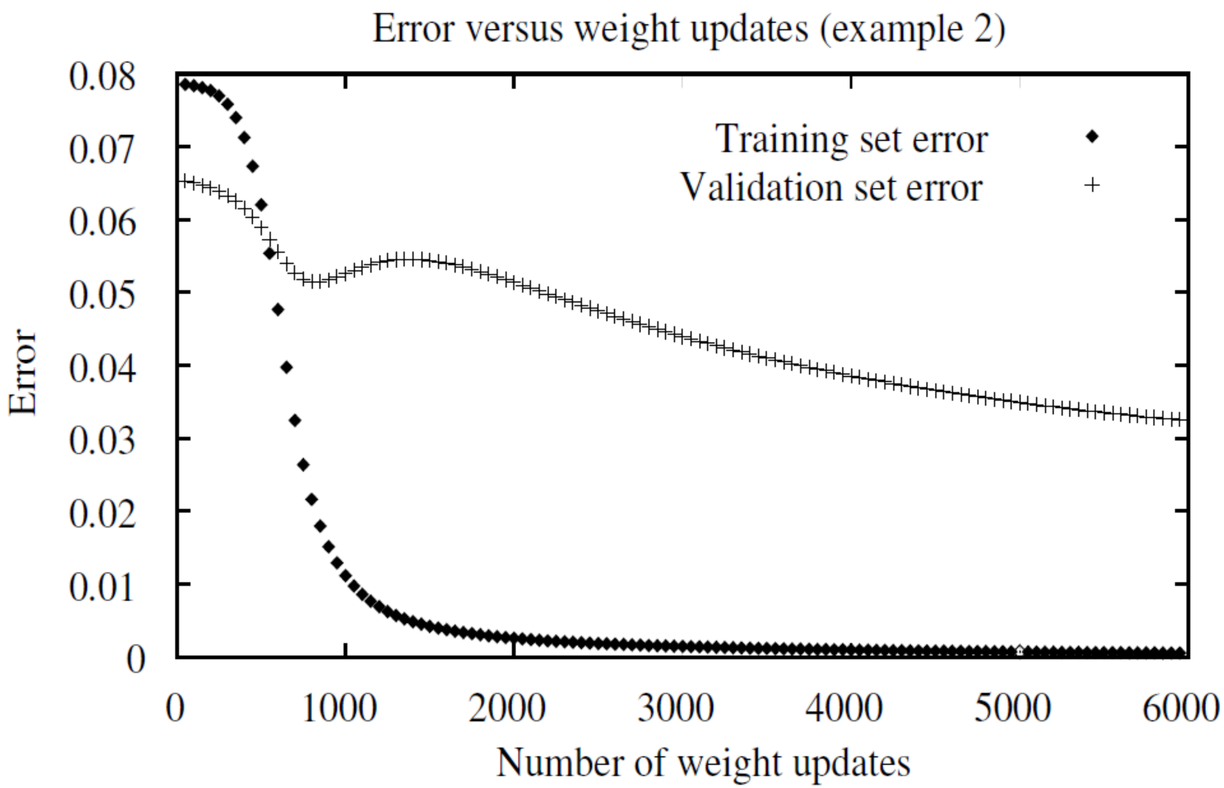
Note also that on second task validation set error continues to decrease after an initial increase — any regularisation (network simplification, or weight reduction) strategies need to avoid early stopping (underfitting).

UNSW AUSTRALIA

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.

Application: Face Recognition

Dataset: 624 images of faces of 20 different people

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

Raw images compressed to 30x32 pixels

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

Application: Face Recognition


left


straight

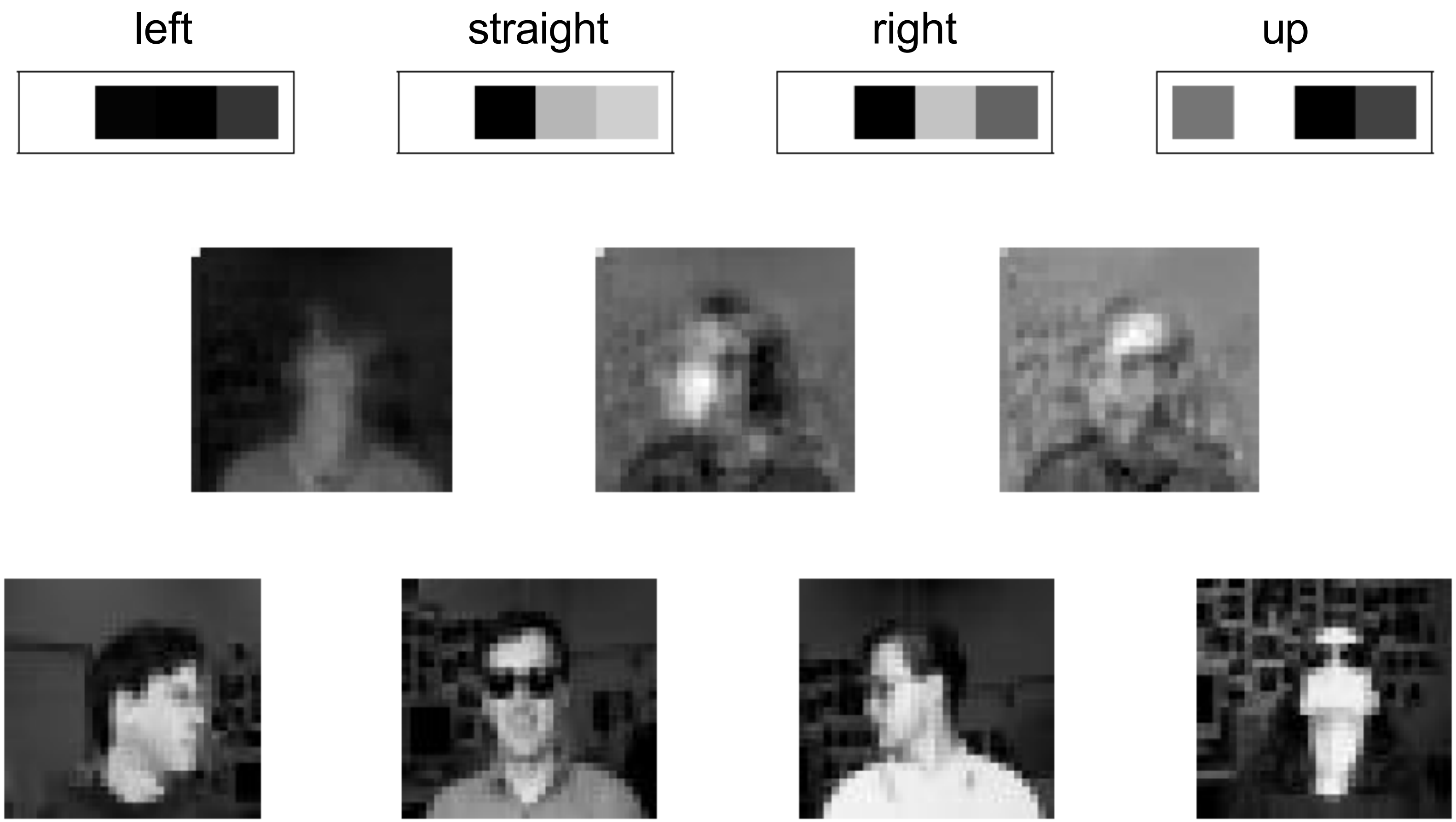

right


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

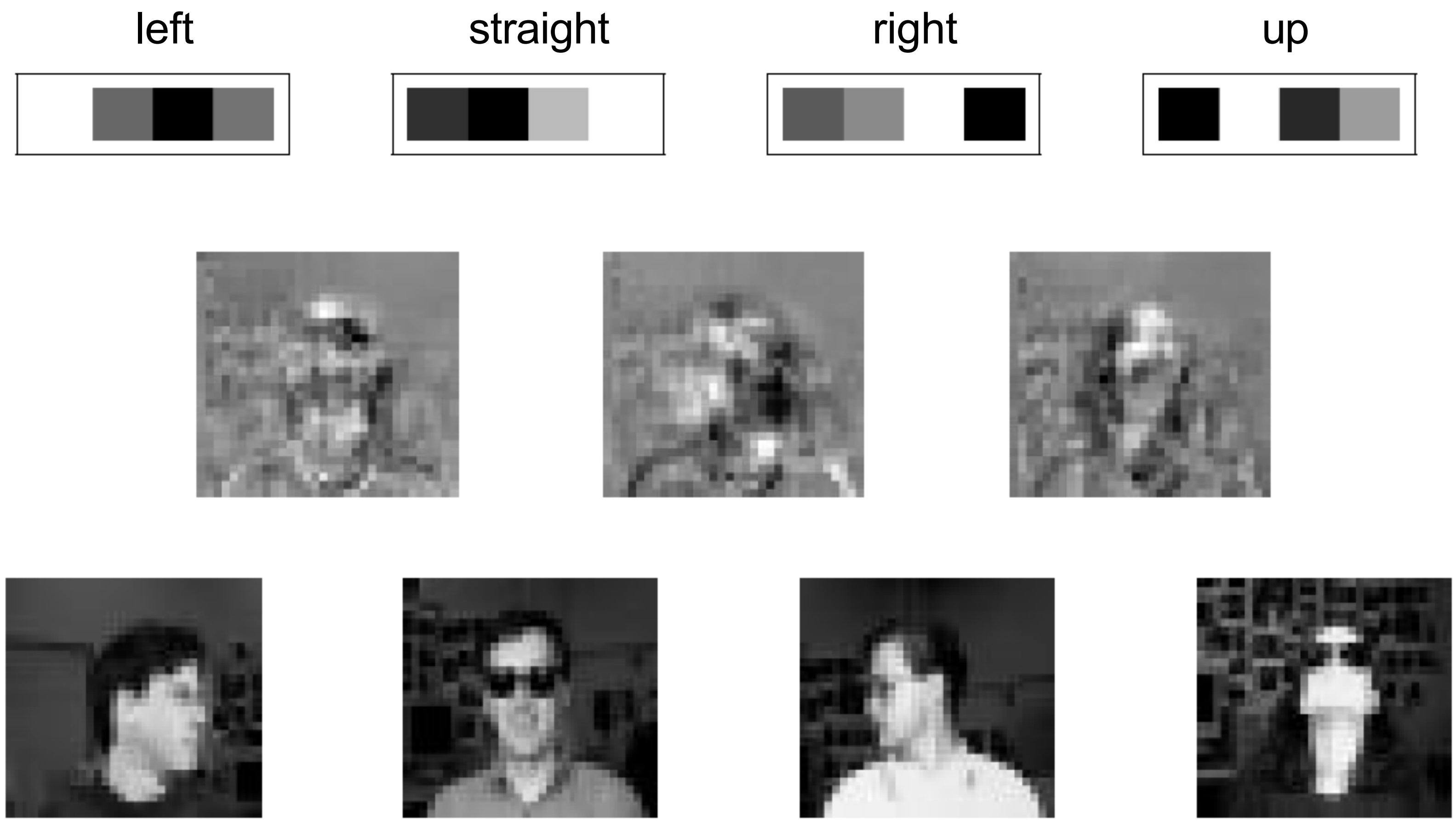
Application: Face Recognition

After 1 epoch



Application: Face Recognition

After 100 epochs



Application: Face Recognition

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

Summary

Artificial Neural Networks

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.

Next lecture: Part 2 – Deep Learning (CNNs)

Acknowledgement

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)

Slides from CISC 4631/6930 Data Mining https://slideplayer.com/slide/13508539/