University of Essex

School of Computer Science & Electronic Engineering

CE213 Artificial Intelligence – Lecture 15

Neural Networks: Part 3

Multilayer Feedforward Neural Networks

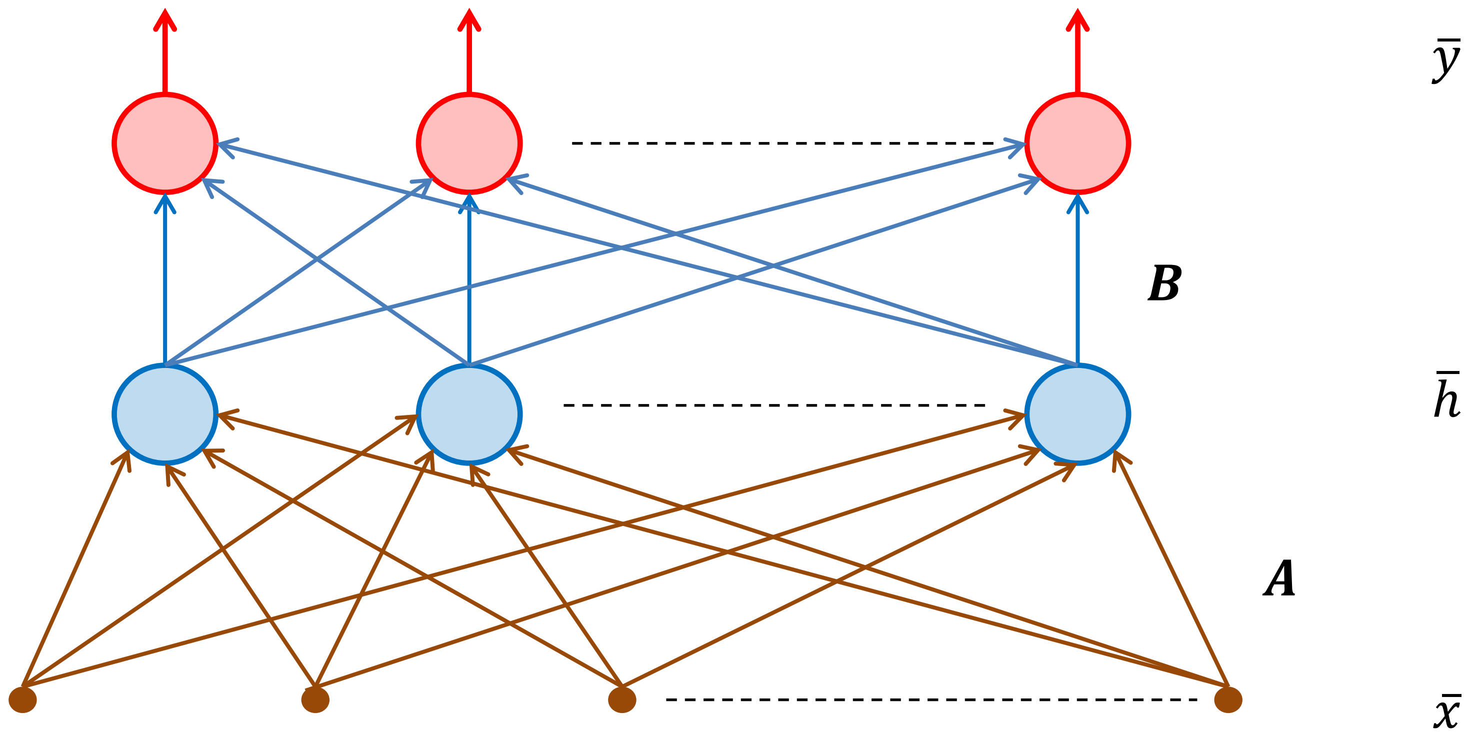
Error Back-Propagation Learning Algorithm

Overfitting Problem

[Warning: It may be difficult to understand the error back-propagation idea and related equations.]

Multiple Layer Network Using Linear Units

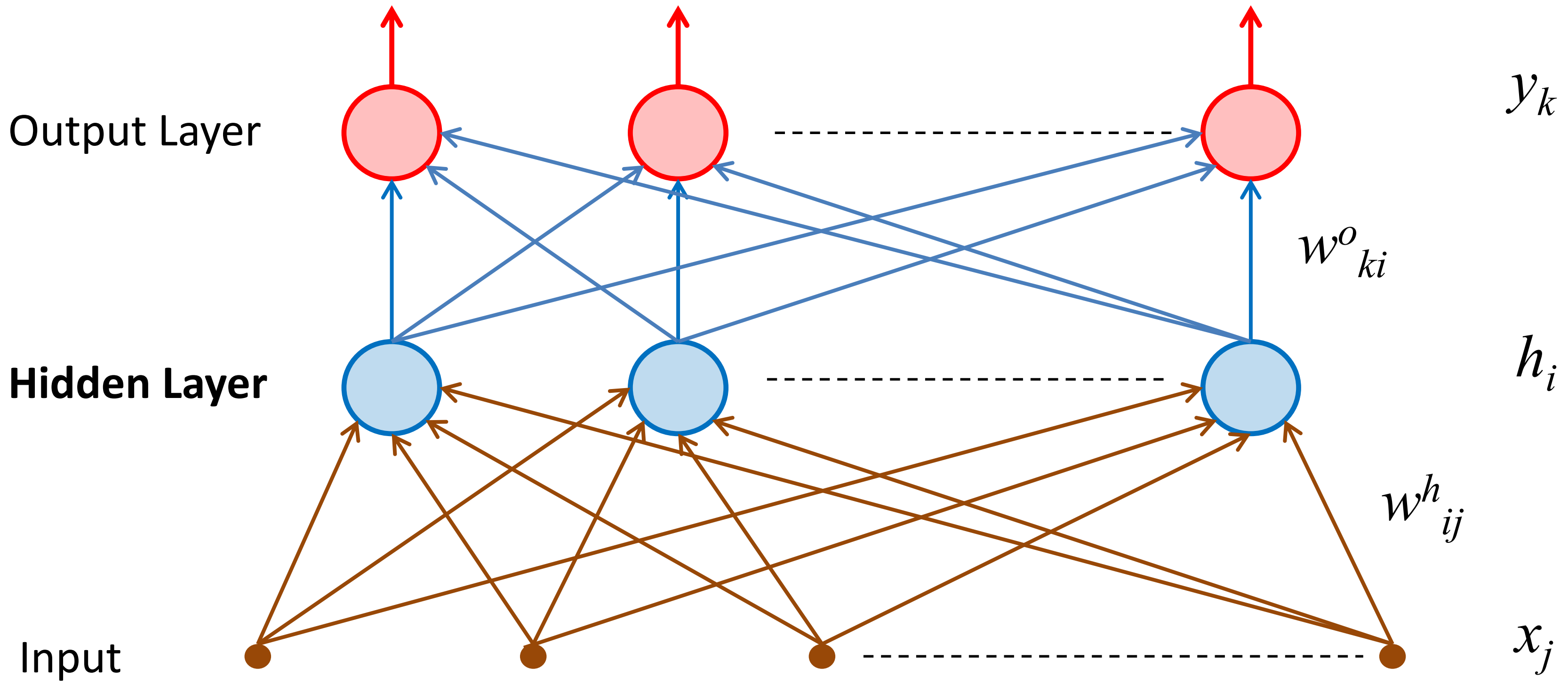
There is no point building a two-layer network of **linear** units, because it is functionally equivalent to a single layer network.



$$\bar{y} = \boldsymbol{B}\bar{h} = \boldsymbol{B}(\boldsymbol{A}\bar{x}) = (\boldsymbol{BA})\bar{x} = \boldsymbol{W}\bar{x}$$

Multilayer Feedforward Neural Network

However, a two-layer neural network of **non-linear** neurons can be much more powerful than a single layer network.



$$y_k = f(\sum_{i=1}^{n_h} w_{ki}^o \cdot h_i - \theta_k^o) = f(\sum_{i=1}^{n_h} w_{ki}^o \cdot \underline{f(\sum_{j=1}^{n} w_{ij}^h \cdot x_j - \theta_i^h) - \theta_k^o})$$

Multilayer Feedforward Neural Network (2)

There are two problems:

- The delta rule works for single layer neural networks or multilayer networks with linear units only.

  How do we modify it so that it can work with non-linear neurons?

  Solution: **Modify the delta rule → The generalised delta rule**.

- We don't know what the desired outputs or errors from the hidden layer would be.

  How do we train the hidden layer (update its weights)?

  Solution: **Reasonably estimate the 'errors' of the hidden neurons based on errors of output neurons → The error back-propagation learning algorithm**.

The Generalised Delta Rule

The delta rule is defined as

$$w_j(t+1) = w_j(t) + \alpha\delta x_j$$

where x_j is the input to the neuron and

$$\delta \equiv z - y$$

Now suppose we replace the linear function $y = \overline{w}\bar{x}$ with some **non-linear function** f so that

$$y = f(\overline{w}\bar{x})$$

How can we modify the delta rule so that it will maximise the error reduction, no matter what the form of f is ?

(N.B. $\overline{w}$ and $\bar{x}$ are vectors. In some following slides, $\overline{w}\bar{x}$ is simply written as wx)

The Generalised Delta Rule (2)

We modify the delta rule to give the following ***generalised delta rule***:

$$\delta x_j \Rightarrow \delta \frac{dy}{dw_j} = (z - y)\frac{dy}{dw_j}$$

$$\boxed{why\ \frac{dy}{dw_j}\ ?}$$

dy/dw_j indicates how change in w_j will lead to change in y. In the case of a linear unit, this reduces to the simple delta rule since $y = \bar{w}\bar{x}$ and $dy/dw_j = x_j$.

This restricts our choice of f to differentiable functions because the function must have a derivative defined at all points.

So we cannot use a step function ( ) as in the original MP neuron.

It can be proved that the generalized delta rule (similar to gradient descent algorithm) has the same property as the original delta rule, i.e.,

It modifies the weights to minimize $|z - y|$.

(Strict proof is beyond the scope of CE213. Will give a graphical interpretation later.)

The Generalised Delta Rule (3)

Choosing a suitable non-linear function

We need a function that is

- differentiable (so we can find its derivative at any point)
- Monotonically increasing (so increased input always produces increased output)

We would prefer a function that

- Is asymptotic for large values of $\overline{w}\bar{x}$, so that the output remains within limits (similar to the step function used in the MP neuron).
- Has a readily computable first derivative.

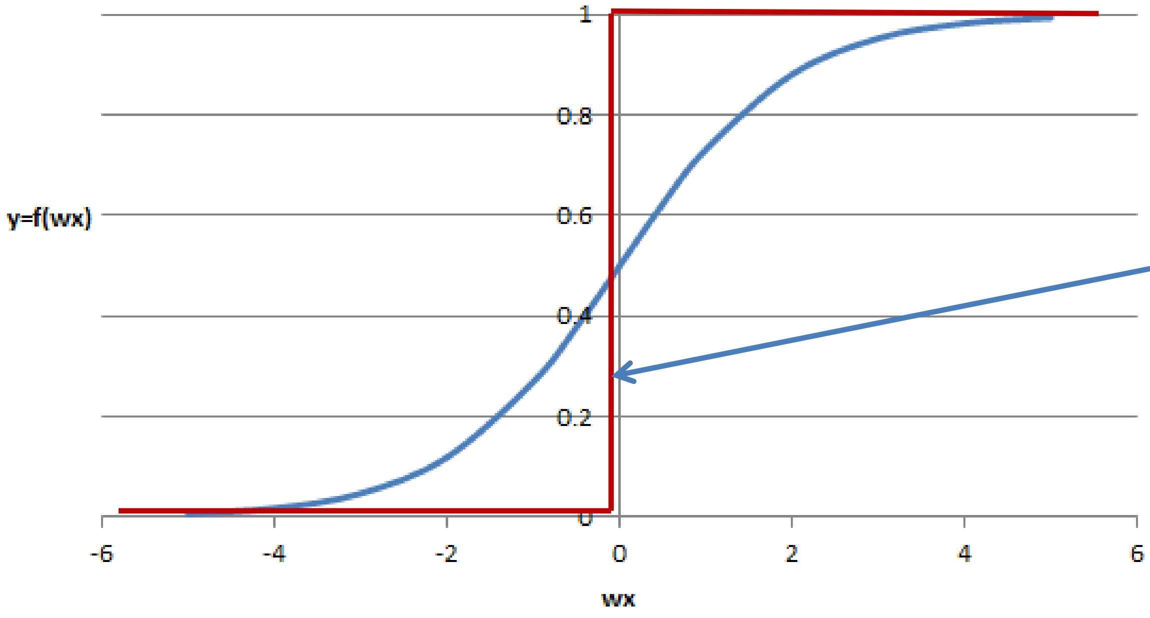
The Generalised Delta Rule (4)

The Logistic Function

$$y = \frac{1}{1 + e^{-wx}}$$

Choosing the logistic function for neuron modelling led to a breakthrough in machine learning in 1980s .

It has a sigmoid shape:



Step function used in the MP neuron, with $\theta = 0$

The Generalised Delta Rule (5)

Properties of the logistic function:

- Differentiable
- Monotonically increasing
- $f(wx) \to +1$ as $wx \to +\infty$
- $f(wx) \to 0$ as $wx \to -\infty$

(asymptotic)

- $f(wx)$ = 0.5 when wx = 0 (good for thresholding if needed)
- Derivative has a simple form:

$$f(wx) \times \left(1 - f(wx)\right) \quad \text{or} \quad y(1\text{-}y)$$

Training the Hidden Layer

The 2nd Problem

How can we train the hidden neurons when we do not know what their desired outputs (or errors) would be?

(for output neurons we have $\delta \equiv z - y$)

Solution

Devise a way of making a plausible guess at what the errors of the hidden neurons would be.

[Training a neuron means finding appropriate values of its weights so that it will output as desired or the error between desired output and actual output will be minimised.]

Training the Hidden Layer (2)

Basic idea: Feeding back the error from the output layer

- Make the estimated errors of hidden neurons **proportional** to the errors of the output neurons. This is reasonable because:
  - If the output is perfect we won't mess it up by changing the hidden layer (no error, no change).
  - If the output error is small we will only make small changes to the hidden layer.

How much of the error should be fed back?

- Each hidden neuron can contribute to each of the output neurons.
- The **contribution of a particular hidden neuron** to a particular output neuron will depend on the **weight** of the connection between them.
- Therefore, it is reasonable to use weights to proportion the error feedback.

Training the Hidden Layer (3)

So, the solution to the 2nd problem (estimating errors of hidden neurons) is:

- Making the **estimated error of a hidden neuron** simply the **weighted sum of the errors of the output neurons**.
- Using the weights of the connections from the hidden neuron to the output neurons to form this weighted sum.

We can express this solution by introducing **a modified form of the generalised delta rule** for hidden neurons, described by the following equations:

$$w_{ij}^{h}(t+1) = w_{ij}^{h}(t) + \underline{\alpha\delta_i^h x_j}$$

$$\delta_i^h = \underline{\left(\sum_{k=1}^{m} \delta_k^o \cdot w_{ki}^o\right)} \cdot h_i \cdot (1-h_i)$$

$$\delta_k^o = \underline{(z_k - y_k)} \cdot y_k \cdot (1-y_k)$$

k : index for output units,

i : index for hidden units,

j : index for inputs

$\uparrow$ (Back propagation)

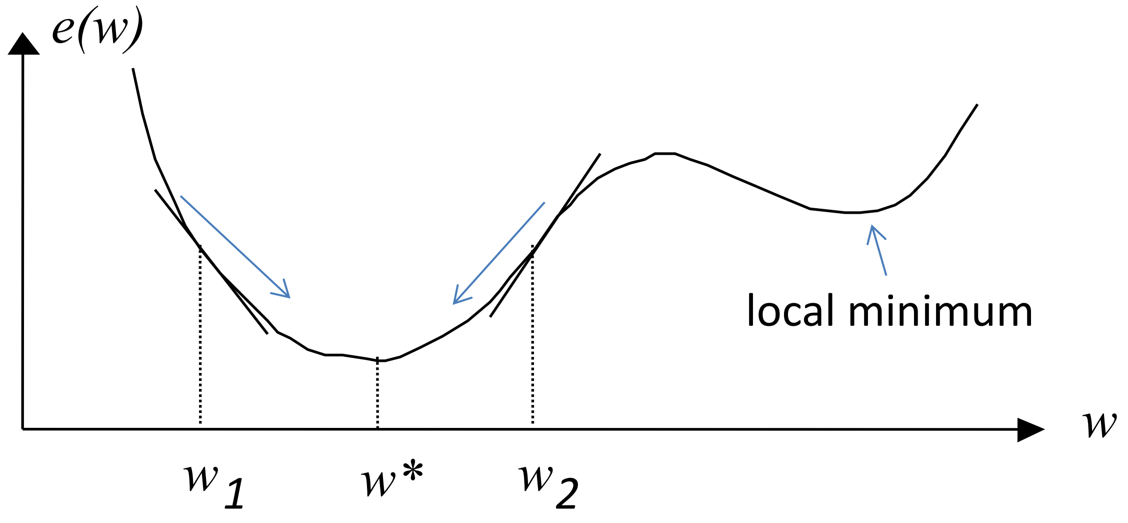
This process of passing back a weighted error is called ***error back-propagation.***

[Difficulty warning!]

Error Back-Propagation Learning Process Regarded as Error Gradient Descending

The delta rule and all its variants used in error back-propagation can be regarded as ***error gradient descending*** procedures, because

$$e = \frac{1}{2}(z-y)^2, \qquad \Delta w = -\alpha \frac{de}{dw} = \alpha(z-y)\frac{dy}{dw}$$



$$(\frac{de}{dw} < 0,\ \Delta w > 0) \qquad (\frac{de}{dw} > 0,\ \Delta w < 0)$$

Weight updating by Δw always reduce error if α is small enough.

When to Stop Error Back-Propagation?

Error back-propagation learning is an iterative process. How do you determine when the learning process should stop?

An obvious approach:

Monitor the average error for the output layer, $(z - y)$.

Stop learning when this falls below a small value ε.

Problem: It may happen that after a long period of learning we still have $|z - y| > \varepsilon$.

Alternative stopping criteria include:

Stop when the average error change is very small.

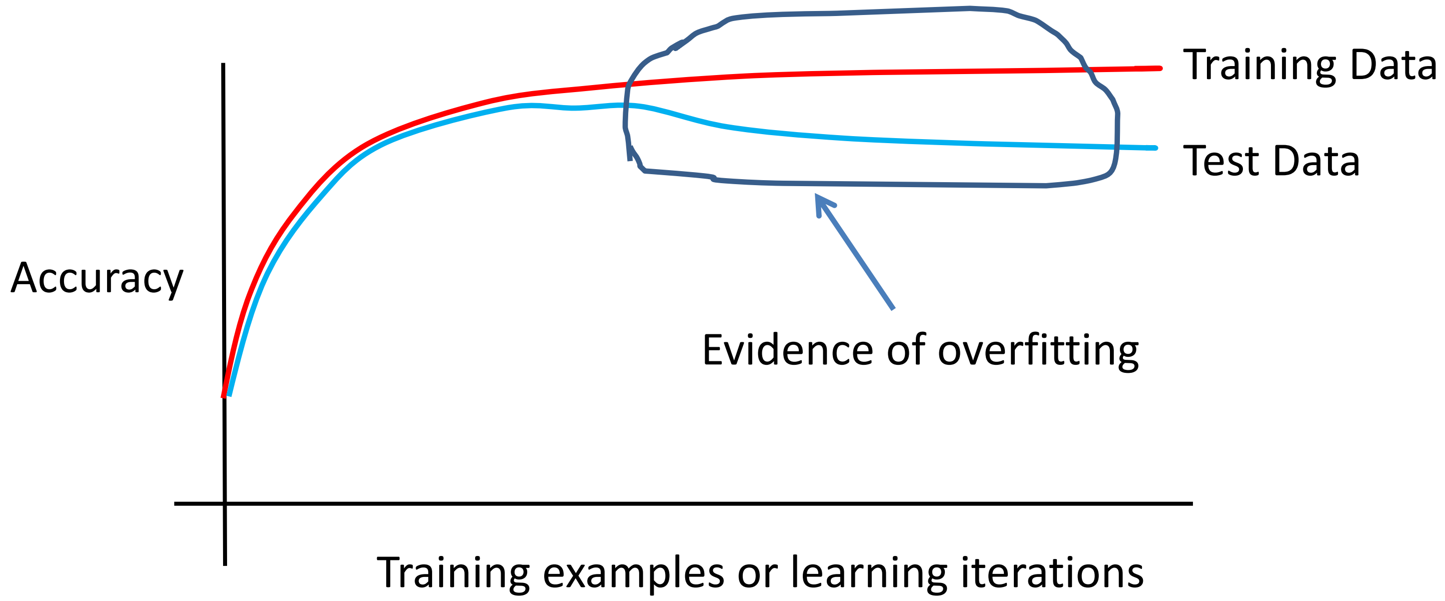
or

Stop when a preset number of iterations have been reached.

Overfitting in Error Back-Propagation Networks

As in other forms of learning (e.g., decision tree induction), overfitting is a potential problem in error back-propagation neural networks.

Overfitting occurs because a built model is supposed to reflect the characteristics of the whole population, but it may reflect the characteristics peculiar to the training dataset only.



Overfitting in Error Back-Propagation Networks (2)

Main reasons for overfitting:

- Model used for machine learning is too flexible and powerful
- Number of training samples is too small (not representative)
- Training samples are too noisy (of low quality)

Methods for reducing overfitting:

- Keep the number of hidden units (and thus number of weights) small
- Add weight decay

  Decrease weight values by a small factor each iteration.
  Thus weights tend to stay small.
- Use a large number of high-quality representative training samples.

The first two techniques **restrict the complexity** of the model that can be developed.

Overfitting in Error Back-Propagation Networks (3)

Dealing with overfitting through cross-validation:

Apart from the methods mentioned in the previous slide, cross-validation is a useful technique for reducing overfitting.

Split the available training data into two datasets:

Training dataset:	Used for weight updating.
Validation dataset:	Used to assess performance throughout learning.

Assess performance using validation dataset throughout learning.

Save weights for best performance so far.

Continue learning until performance on validation dataset has clearly dropped.

Restore weights to values saved from best performance.

Finally, assess performance using a separate ***Test dataset***.

Cross-validation can be also used to determine optimal neural network structure, leading to ***parametric learning combined with structural learning***.

Applications of Error Back-Propagation Networks

Error back-propagation neural networks are very powerful in approximation, prediction, and classification.

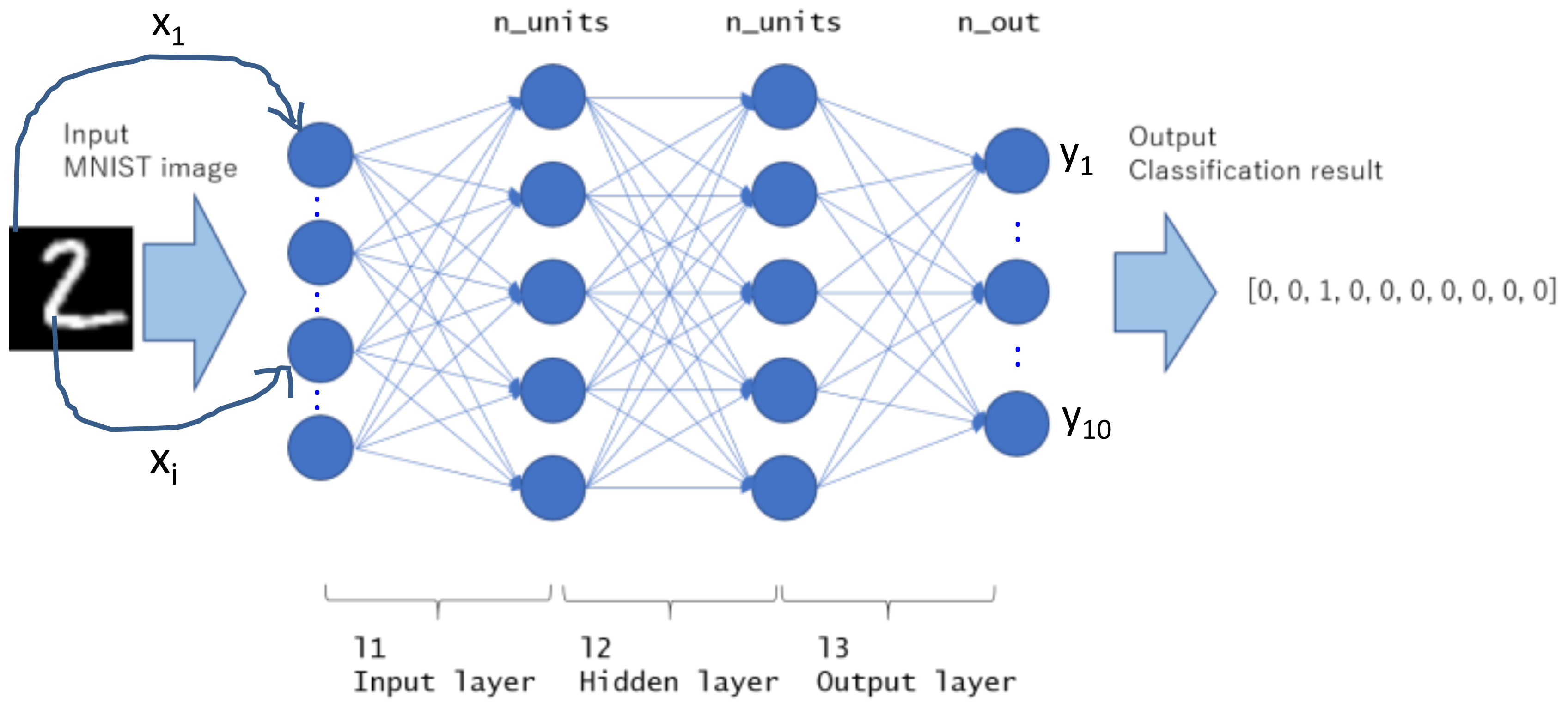
For classification problems, the desired output is like 0 … 0 1 0 …0, with the only 1 representing a certain class. As actual output y takes continuous values, for classification problems, y should be converted to binary values by thresholding, e.g., 1 if it is greater than 0.5, and 0 if it is less than 0.5.

Unfortunately, it is impossible to demonstrate, step by step, the learning procedure of an error back-propagation neural network using a small set of training samples, like what we did for decision tree induction.

A demo: https://www.youtube.com/watch?v=hB3b1CoZuR4 (classification)

Lab exercise 3 at your own time provides another example.

An Application Example: Handwritten Digit Recognition



What would be happening during machine learning?
What would be different for face recognition?

Summary

Multilayer Neural Networks Using Error Back-Propagation

Necessity of non-linear neurons

The generalized delta rule

Error back-propagation for training hidden neurons

Overfitting in Error Back-Propagation Neural Networks

What is overfitting?

How to deal with overfitting?