CLASSIFICATION METHODS

CS5011- MACHINE LEARNING

CLASSIFICATION AND REGRESSION TREES

- The process of selecting a specific model, given a new input **x**, can be described by a sequential decision making process corresponding to the traversal of a binary tree (one that splits into two branches at each node).
- Here we focus on a particular tree-based framework called *classification and regression trees*, or *CART* (Breiman *et al.*, 1984)

CLASSIFICATION AND REGRESSION TREES

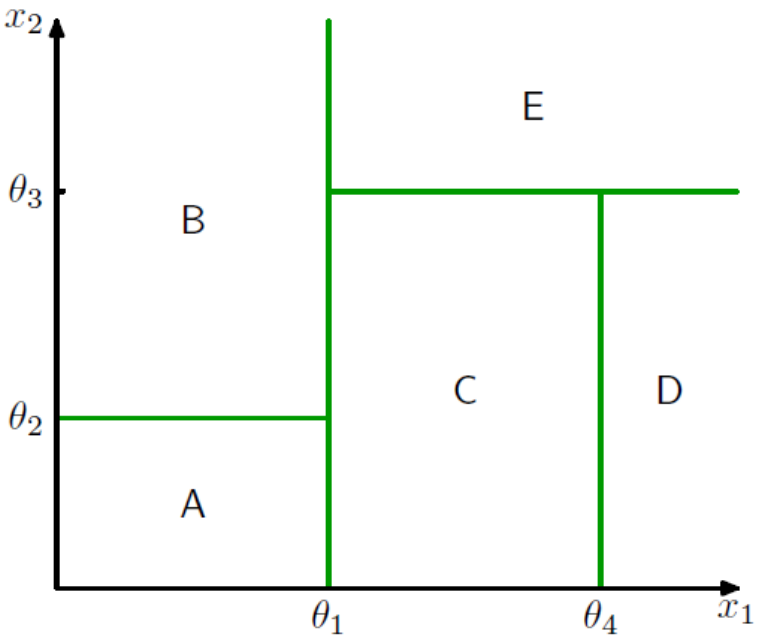
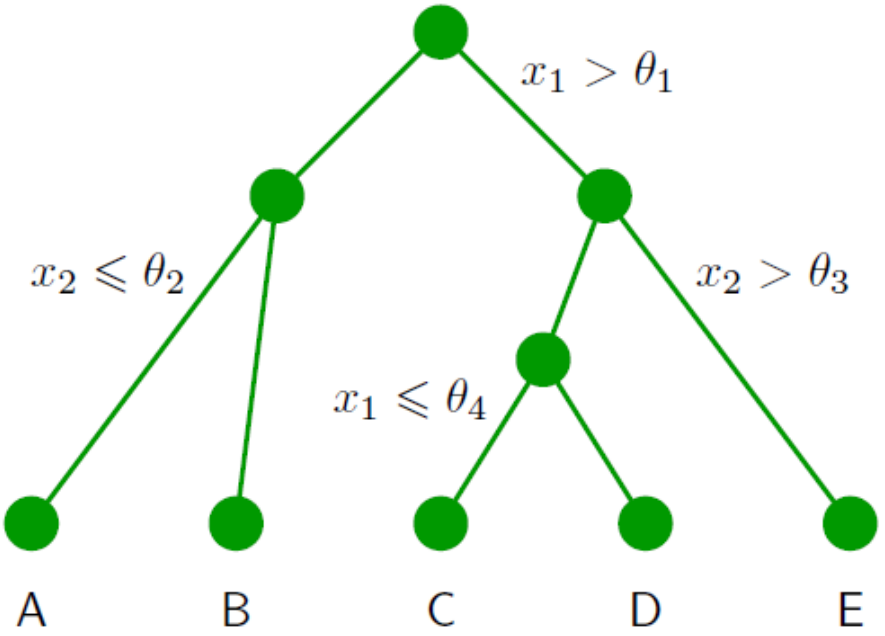


Illustration of a two-dimensional input space that has been partitioned into five regions using axis-aligned boundaries.



Binary tree corresponding to the partitioning of input space (eg BSP tree)

CLASSIFICATION AND REGRESSION TREES

- In the example given in previous slide, the first step divides the whole of the input space into two regions according to whether $x_1 \leq \theta_1$ or $x_1 > \theta_1$ where θ_1 is a parameter of the model.
- This creates two sub regions, each of which can then be subdivided independently.
- For instance, the region $x_1 \leq \theta_1$ is further subdivided according to whether $x_2 \leq \theta_2$ or $x_2 > \theta_2$, giving rise to the regions denoted A and B.
- For any new input **x**, we determine which region it falls into by starting at the top of the tree at the root node and following a path down to a specific leaf node according to the decision criteria at each node.

CLASSIFICATION AND REGRESSION TREES

- Within each region, there is a separate model to predict the target variable.
- For instance, in regression we might simply predict a constant over each region, or in classification we might assign each region to a specific class.
- **EXAMPLE**: For instance, to predict a patient's disease, we might
  - first ask "is their temperature greater than some threshold?". If the answer is yes, then
  - we might next ask "is their blood pressure less than some threshold?".

Each leaf of the tree is then associated with a specific diagnosis.

CLASSIFICATION AND REGRESSION TREES

- Consider first a regression problem in which the goal is to predict a single target variable t from a *D*-dimensional vector $\mathbf{x} = (x_1, \ldots, x_D)^T$ of input variables.
- The training data consists of input vectors $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ along with the corresponding continuous labels $\{t_1, \ldots, t_N\}$.
- If the partitioning of the input space is given, and we minimize the sum-of-squares error function, then the optimal value of the predictive variable within any given region is just given by the average of the values of t_n for those data points that fall in that region.

WHEN TO STOP ADDING NODES

- A simple approach would be to stop when the reduction in residual error falls below some threshold.
- However, it is found empirically that often none of the available splits produces a significant reduction in error, and yet after several more splits a substantial error reduction is found.
- For this reason, it is common practice to grow a large tree, using a stopping criterion based on the number of data points associated with the leaf nodes, and then prune back the resulting tree.
- The pruning is based on a criterion that balances residual error against a measure of model complexity.

WHEN TO STOP ADDING NODES

- If we denote the starting tree for pruning by T_0, then we define $T \subset T_0$ to be a subtree of T_0 if it can be obtained by pruning nodes from T_0 (in other words, by collapsing internal nodes by combining the corresponding regions).
- Suppose the leaf nodes are indexed by $\tau = 1, \ldots, |T|$, with leaf node τ representing a region R_τ of input space having N_τ data points, and $|T|$ denoting the total number of leaf nodes.
- The optimal prediction for region R_τ is then given by

$$y_\tau = \frac{1}{N_\tau} \sum_{\mathbf{x}_n \in \mathcal{R}_\tau} t_n$$

WHEN TO STOP ADDING NODES

- and the corresponding contribution to the residual sum-of-squares is then

$$Q_\tau(T) = \sum_{\mathbf{x}_n \in \mathcal{R}_\tau} \{t_n - y_\tau\}^2$$

- The pruning criterion is then given by

$$C(T) = \sum_{\tau=1}^{|T|} Q_\tau(T) + \lambda |T|$$

- The regularization parameter λ determines the trade-off between the overall residual sum-of-squares error and the complexity of the model as measured by the number $|T|$ of leaf nodes, and its value is chosen by cross-validation.

WHEN TO STOP ADDING NODES

- For classification problems, the process of growing and pruning the tree is similar, except that the sum-of-squares error is replaced by a more appropriate measure of performance.
- If we define $p_{\tau k}$ to be the proportion of data points in region R_τ assigned to class k, where $k = 1, \ldots, K$, then two commonly used choices are the cross-entropy

$$Q_\tau(T) = \sum_{k=1}^{K} p_{\tau k} \ln p_{\tau k}$$

- and the *Gini index*

$$Q_\tau(T) = \sum_{k=1}^{K} p_{\tau k} (1 - p_{\tau k})$$

- These both vanish for $p_{\tau k} = 0$ and $p_{\tau k} = 1$ and have a maximum at $p_{\tau k} = 0.5$.

Advantages

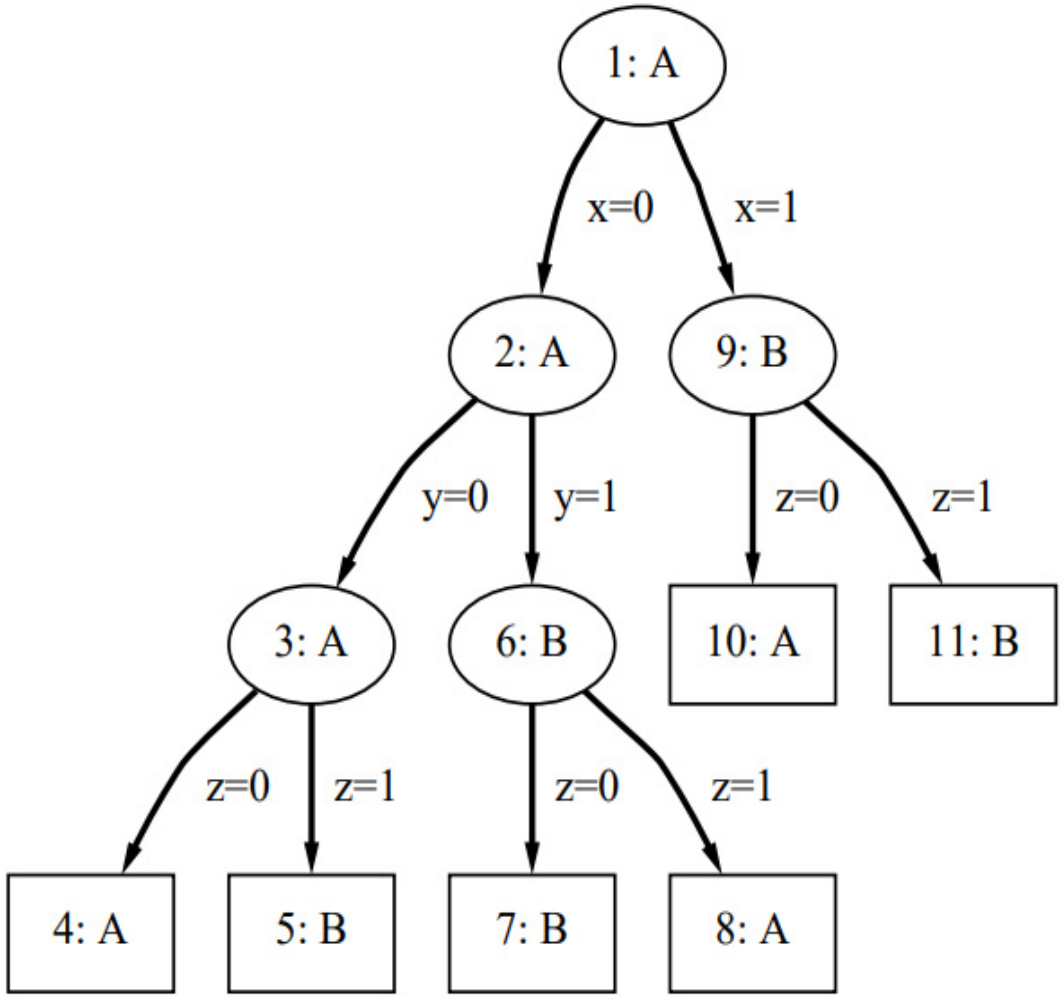
- The cross entropy and the Gini index are better measures than the misclassification rate for growing the tree because they are more sensitive to the node probabilities.
- Also, unlike misclassification rate, they are differentiable and hence better suited to gradient based optimization methods.
- The human interpretability of a tree model such as CART is often seen as its major strength.

Disadvantages

- In practice it is found that the particular tree structure that is learned is very sensitive to the details of the data set, so that a small change to the training data can result in a very different set of splits.

Decision Tree Pruning

Example

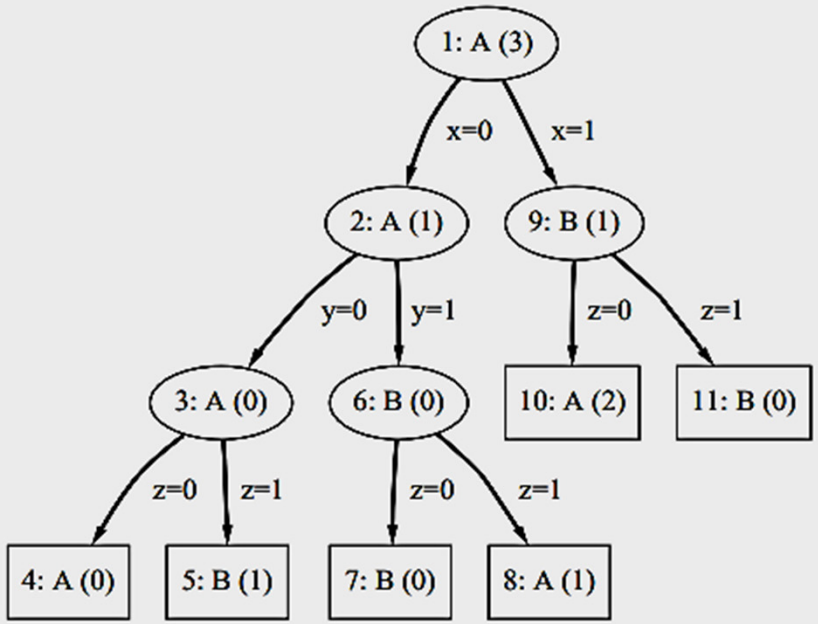


A decision tree with two classes A and B (with node numbers and class labels)

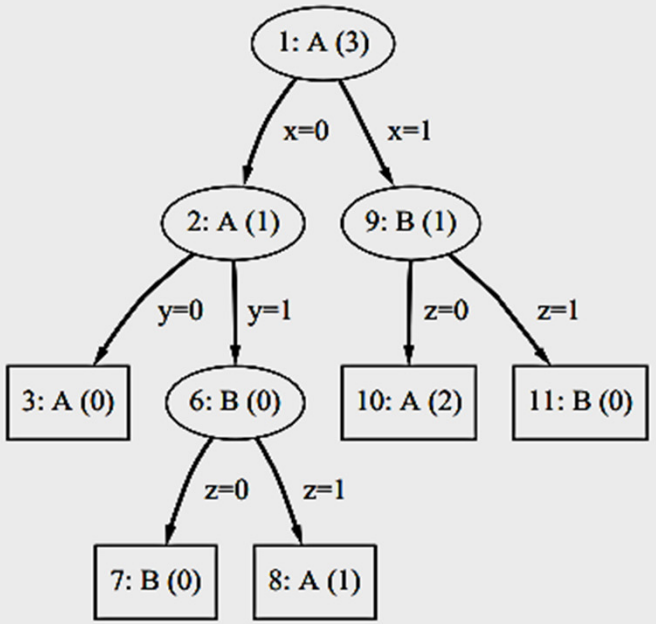
x	y	z	class
0	0	1	A
0	1	1	B
1	1	0	B
1	0	0	B
1	1	1	A

An example pruning set

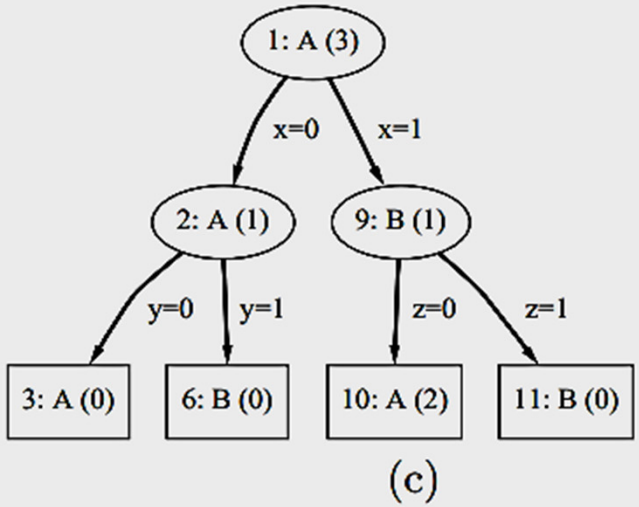
- The idea is to hold out some of the available instances—the “pruning set”—when the tree is built, and prune the tree until the classification error on these independent instances starts to increase.
- Because **the instances in the pruning set are not used for building the decision tree**, they provide a less biased estimate of its error rate on future instances than the training data.



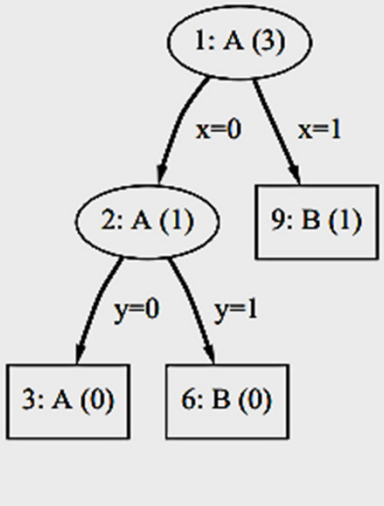
(a)



(b)



(c)



(d)

x	y	z	class
0	0	1	A
0	1	1	B
1	1	0	B
1	0	0	B
1	1	1	A

- In each tree, the number of instances in the pruning data that are misclassified by the individual nodes are given in parentheses.
- Assuming that the tree is traversed left-to-right, the pruning procedure first considers for removal the subtree attached to node 3.
- Because the subtree's error on the pruning data (1 error) exceeds the error of node 3 itself (0 errors), node 3 is converted to a leaf.
- Next, node 6 is replaced by a leaf for the same reason.
- Having processed both of its successors, the pruning procedure then considers node 2 for deletion. However, because the subtree attached to node 2 makes fewer mistakes (0 errors) than node 2 itself (1 error), the subtree remains in place. Next, the subtree extending from node 9 is considered for pruning, resulting in a leaf.
- In the last step, node 1 is considered for pruning, leaving the tree unchanged.

Feed-forward Network Functions

- The linear models for regression and classification are based on linear combinations of fixed nonlinear basis functions $\phi_j(\mathbf{x})$ and take the form

$$y(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=1}^{M} w_j \phi_j(\mathbf{x})\right)$$

- where $f(\cdot)$ is a nonlinear activation function in the case of classification and is the identity in the case of regression.
- Our goal is to extend this model by making the basis functions $\phi_j(\mathbf{x})$ depend on parameters and then to allow these parameters to be adjusted, along with the coefficients $\{w_j\}$, during training.

Feed-forward Network Functions

- The basic neural network model can be described a series of functional transformations. First we construct *M* linear combinations of the input variables $x_1, \ldots, x_D$ in the form

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}$$

- where $j = 1, \ldots, M$, and the superscript (1) indicates that the corresponding parameters are in the first 'layer' of the network.
- We shall refer to the parameters $w_{ji}^{(1)}$ as *weights* and the parameters $w_{j0}^{(1)}$ as *biases.*
- The quantities a_j are known as *activations*.

Feed-forward Network Functions

- Each of them is then transformed using a differentiable, nonlinear *activation function* $h(\cdot)$ to give

$$z_j = h(a_j).$$

- These quantities, in the context of neural networks, are called *hidden units*.
- The nonlinear functions $h(\cdot)$ are generally chosen to be sigmoidal functions such as the logistic sigmoid or the 'tanh'.

- These values are again linearly combined to give *output unit activations*

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)}$$

where $k = 1, \ldots, K,$ and K is the total number of outputs.

- This transformation corresponds to the second layer of the network, and again the $w_{k0}^{(2)}$ are bias parameters.

Feed-forward Network Functions

- Finally, the output unit activations are transformed using an appropriate activation function to give a set of network outputs y_k.
- The choice of activation function is determined by the nature of the data and the assumed distribution of target variables and follows the same considerations as for linear models.
- Thus for standard regression problems, the activation function is the identity so that $y_k = a_k$.
- Similarly, for multiple binary classification problems, each output unit activation is transformed using a logistic sigmoid function so that

$$y_k = \sigma(a_k) \text{ where } \sigma(a) = \frac{1}{1 + \exp(-a)}.$$

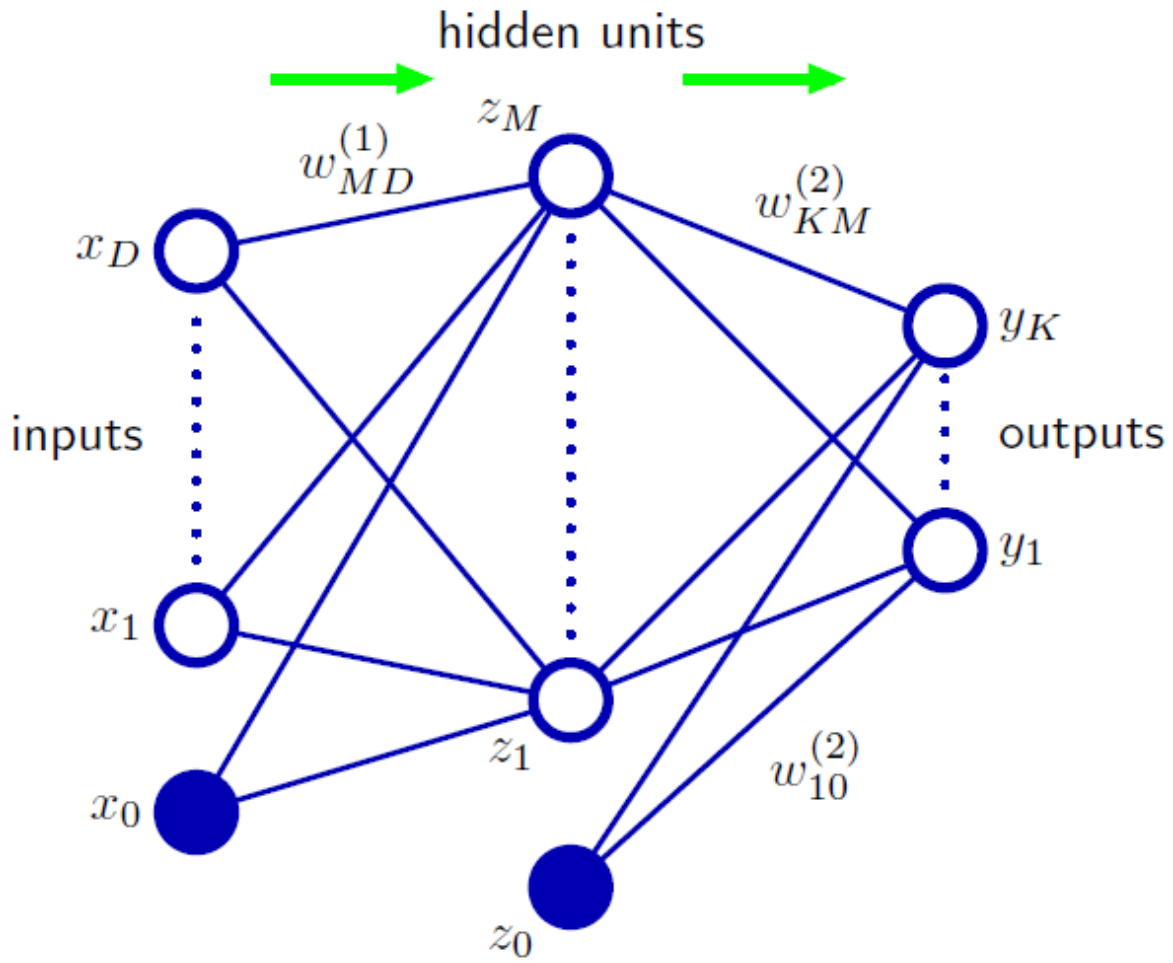
Feed-forward Network Functions

- Finally, for multiclass problems, a softmax activation function is used.
- We can combine these various stages to give the overall network function that, for sigmoidal output unit activation functions, takes the form

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right)$$

where the set of all weight and bias parameters have been grouped together into a vector **w**.

- Thus the neural network model is simply a nonlinear function from a set of input variables $\{x_i\}$ to a set of output variables $\{y_k\}$ controlled by a vector **w** of adjustable parameters.



Network diagram for the two layer neural network. The input, hidden, and output variables are represented by nodes, and the weight parameters are represented by links between the nodes, in which the bias parameters are denoted by links coming from additional input and hidden variables x_0 and z_0.
Arrows denote the direction of information flow through the network during forward propagation.

Feed-forward Network Functions

- The process of evaluating

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right)$$

can be interpreted as a *forward propagation* of information through the network.

- The bias parameters can be absorbed into the set of weight parameters by defining an additional input variable x_0 whose value is clamped at $x_0 = 1$, so that

$$a_j = \sum_{i=0}^{D} w_{ji}^{(1)} x_i.$$

Feed-forward Network Functions

- We can similarly absorb the second-layer biases into the second-layer weights, so that the overall network function becomes

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=0}^{M} w_{kj}^{(2)} h\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\right)\right).$$

- If the activation functions of all the hidden units in a network are taken to be linear, then for any such network we can always find an equivalent network without hidden units.
- Neural networks are said to be *universal approximators*. For example, a two-layer network with linear outputs can uniformly approximate any continuous function on a compact input domain to arbitrary accuracy provided the network has a sufficiently large number of hidden units.

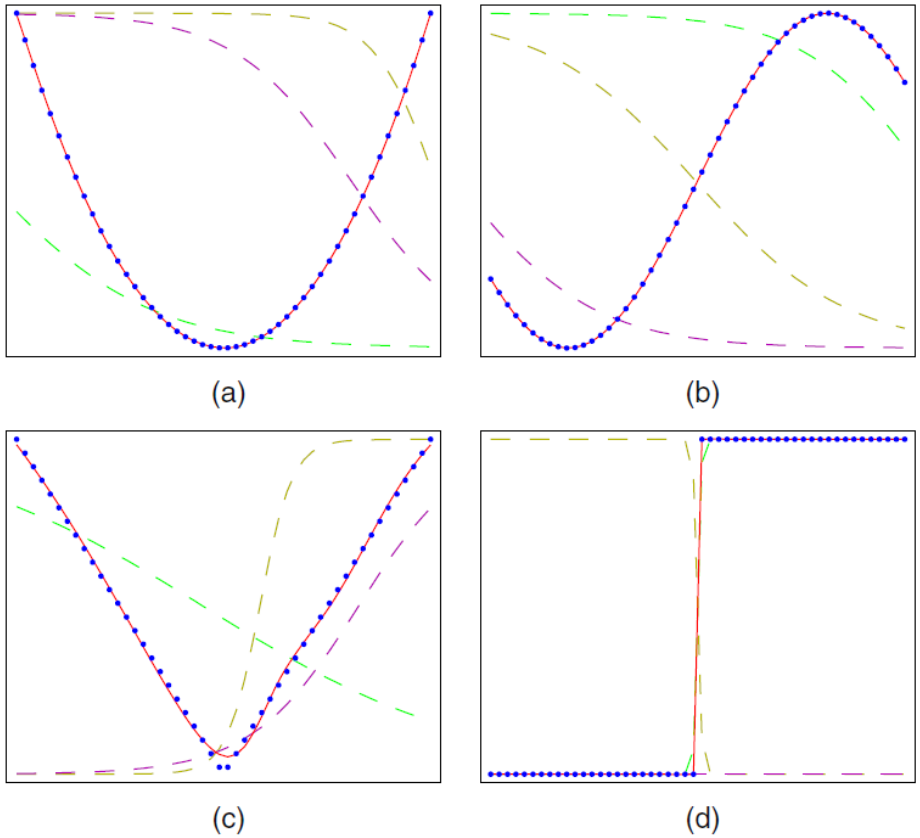


Illustration of the capability of a multilayer perceptron to approximate four different functions comprising (a) $f(x) = x^2$, (b) $f(x) = \sin(x)$, (c), $f(x) = |x|$, and (d) $f(x) = H(x)$ where $H(x)$ is the Heaviside step function. In each case, $N = 50$ data points, shown as blue dots, have been sampled uniformly in x over the interval $(-1, 1)$ and the corresponding values of $f(x)$ evaluated. These data points are then used to train a two layer network having 3 hidden units with 'tanh' activation functions and linear output units. The resulting network functions are shown by the red curves, and the outputs of the three hidden units are shown by the three dashed curves.

Network training

- Given a training set comprising a set of input vectors $\{\mathbf{x}_n\}$, where $n = 1, \ldots, N$, together with a corresponding set of target vectors $\{\mathbf{t}_n\}$ *for regression*, we minimize the error function

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N} \|\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{t}_n\|^2.$$

- Now consider the case of binary classification in which we have a single target variable t such that $t = 1$ denotes class C_1 and $t = 0$ denotes class C_2.
- Consider a network having a single output whose activation function is a logistic sigmoid

$$y = \sigma(a) \equiv \frac{1}{1 + \exp(-a)}$$

so that $0 \leq y(\mathbf{x}, \mathbf{w}) \leq 1$.

Network training

- We can interpret $y(\mathbf{x}, \mathbf{w})$ as the conditional probability $p(C_1|\mathbf{x})$, with $p(C_2|\mathbf{x})$ given by $1 - y(\mathbf{x}, \mathbf{w})$.
- The conditional distribution of targets given inputs is then a Bernoulli distribution of the form

$$p(t|\mathbf{x}, \mathbf{w}) = y(\mathbf{x}, \mathbf{w})^t \{1 - y(\mathbf{x}, \mathbf{w})\}^{1-t}.$$

- If we consider a training set of independent observations, then the error function, which is given by the negative log likelihood, is then a *cross-entropy* error function of the form

$$E(\mathbf{w}) = -\sum_{n=1}^{N} \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\}$$

where y_n denotes $y(\mathbf{x}_n, \mathbf{w})$.

Network training

- Using the cross-entropy error function instead of the sum-of-squares for a classification problem leads to faster training as well as improved generalization.
- If we have K separate binary classifications to perform, then we can use a network having K outputs each of which has a logistic sigmoid activation function.
- Associated with each output is a binary class label $t_k \in \{0, 1\}$, where $k = 1, \ldots, K$.
- If we assume that the class labels are independent, given the input vector, then the conditional distribution of the targets is

$$p(\mathbf{t}|\mathbf{x}, \mathbf{w}) = \prod_{k=1}^{K} y_k(\mathbf{x}, \mathbf{w})^{t_k} \left[1 - y_k(\mathbf{x}, \mathbf{w})\right]^{1-t_k}.$$

Network training

- Taking the negative logarithm of the corresponding likelihood function then gives the following error function

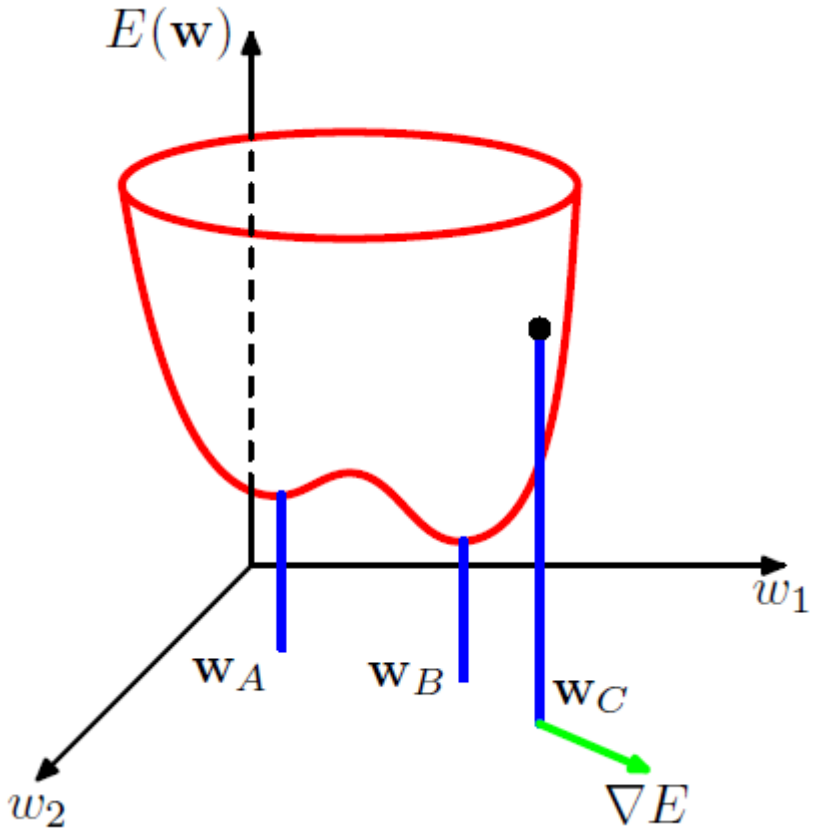
$$E(\mathbf{w}) = -\sum_{n=1}^{N}\sum_{k=1}^{K}\{t_{nk}\ln y_{nk} + (1-t_{nk})\ln(1-y_{nk})\}$$

  where y_{nk} denotes $y_k(\mathbf{x}_n, \mathbf{w})$.

- Finally, we consider the standard multiclass classification problem in which each input is assigned to one of K mutually exclusive classes.
- The binary target variables $t_k \in \{0, 1\}$ have a 1-of-K coding scheme indicating the class, and the network outputs are interpreted as $y_k(\mathbf{x}, \mathbf{w}) = p(t_k = 1|\mathbf{x})$, leading to the following error function

Network training

$$E(\mathbf{w}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{kn} \ln y_k(\mathbf{x}_n, \mathbf{w}).$$



Geometrical view of the error function $E(\mathbf{w})$ as a surface sitting over weight space. Point $\mathbf{w}_A$ is a local minimum and $\mathbf{w}_B$ is the global minimum. At any point $\mathbf{w}_C$, the local gradient of the error surface is given by the vector ∇E.

Network training

- The output unit activation function is given by the softmax function

$$y_k(\mathbf{x}, \mathbf{w}) = \frac{\exp(a_k(\mathbf{x}, \mathbf{w}))}{\sum_j \exp(a_j(\mathbf{x}, \mathbf{w}))}$$

- which satisfies $0 \leq y_k \leq 1$ and $\sum_k y_k = 1$.

	Outputs	
	Real Values	Probabilities
Output Activation	Linear	Softmax
Loss Function	Squared Error	Cross Entropy

Gradient descent optimization

- The simplest approach to using gradient information is to choose the weight update to comprise a small step in the direction of the negative gradient, so that

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)})$$

where the parameter $\eta > 0$ is known as the *learning rate*.

- After each such update, the gradient is re-evaluated for the new weight vector and the process repeated.
- Note that the error function is defined with respect to a training set, and so each step requires that the entire training set be processed in order to evaluate ∇E.
- At each step the weight vector is moved in the direction of the greatest rate of decrease of the error function, and so this approach is known as *gradient descent* or *steepest descent*.

Gradient descent optimization

- On-line gradient descent, also known as *sequential gradient descent* or *stochastic gradient descent*, makes an update to the weight vector based on **one data point at a time**, so that

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E_n(\mathbf{w}^{(\tau)}).$$

Error Backpropagation

- Our goal in this section is to find an efficient technique for evaluating the gradient of an error function $E(\mathbf{w})$ for a feed-forward neural network.
- We shall see that this can be achieved using a local message passing scheme in which information is sent alternately forwards and backwards through the network and is known as ***error backpropagation***, or sometimes simply as *backprop*.
- We now derive the backpropagation algorithm for a general network having arbitrary feed-forward topology, arbitrary differentiable nonlinear activation functions, and a broad class of error function.
- The resulting formulae will then be illustrated using a simple layered network structure having a single layer of sigmoidal hidden units together with a sum-of-squares error.

Error Backpropagation

- Many error functions of practical interest, for instance those defined by maximum likelihood for a set of i.i.d. data, comprise a sum of terms, one for each data point in the training set, so that

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}).$$

- Here we shall consider the problem of evaluating $\nabla E_n(\mathbf{w})$ for one such term in the error function.
- This may be used directly for sequential optimization, or the results can be accumulated over the training set in the case of batch methods.

Error Backpropagation

- Consider first a simple linear model in which the outputs y_k are linear combinations of the input variables x_i so that

$$y_k = \sum_i w_{ki} x_i$$

together with an error function that, for a particular input pattern n, takes the form

$$E_n = \frac{1}{2} \sum_k (y_{nk} - t_{nk})^2$$

where, $y_{nk} = y_k(\mathbf{x}_n, \mathbf{w})$.

The gradient of this error function with respect to a weight w_{ji} is given by

$$\frac{\partial E_n}{\partial w_{ji}} = (y_{nj} - t_{nj}) x_{ni}$$

Error Backpropagation

- In a general feed-forward network, each unit computes a weighted sum of its inputs of the form

$$a_j = \sum_i w_{ji} z_i$$

- where z_i is the activation of a unit, or input, that sends a connection to unit j, and w_{ji} is the weight associated with that connection.
- This sum is transformed by a nonlinear activation function $h(\cdot)$ to give the activation z_j of unit j in the form

$$z_j = h(a_j).$$

- Now consider the evaluation of the derivative of E_n with respect to a weight w_{ji}.

Error Backpropagation

- First we note that E_n depends on the weight w_{ji} only via the summed input a_j to unit j. We can therefore apply the chain rule for partial derivatives to give

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}.$$

- We now introduce a useful notation

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j}$$

where the δ's are often referred to as *errors.*

- Using $a_j = \sum_i w_{ji} z_i$ we can write $\frac{\partial a_j}{\partial w_{ji}} = z_i.$

Error Backpropagation

- We thus obtain

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i.$$

- For the output units, we have

$$\delta_k = y_k - t_k$$

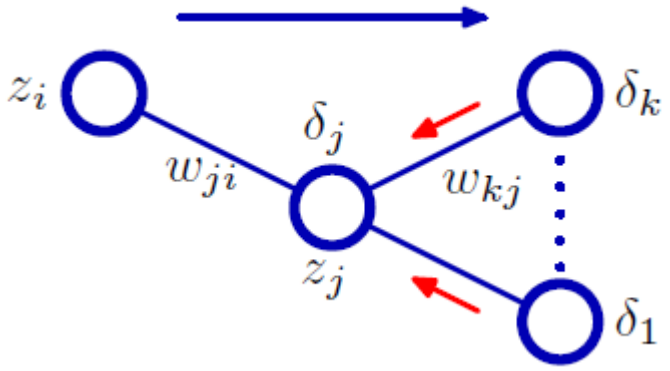


Illustration of the calculation of δ_j for hidden unit j by backpropagation of the δ's from those units k to which unit j sends connections. The blue arrow denotes the direction of information flow during forward propagation, and the red arrows indicate the backward propagation of error information.

Error Backpropagation

- To evaluate the δ's for hidden units, we again make use of the chain rule for partial derivatives,

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j}$$

where the sum runs over all units k to which unit j sends connections.

- If we now substitute the definition of δ we obtain the following *backpropagation* formula

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$$

Error Backpropagation: Summary

The backpropagation procedure can therefore be summarized as follows:

- Apply an input vector $\mathbf{x}_n$ to the network and forward propagate through the network to find the activations of all the hidden and output units.
- Evaluate the δ_k for all the output units.
- Backpropagate the δ's to obtain δ_j for each hidden unit in the network.
- Evaluate the required derivatives.

For batch methods, the derivative of the total error E can then be obtained by repeating the above steps for each pattern in the training set and then summing over all patterns:

$$\frac{\partial E}{\partial w_{ji}} = \sum_n \frac{\partial E_n}{\partial w_{ji}}.$$

Backpropagation Algorithm: Definitions

- Each training example is a pair of the form $(\vec{x}, \vec{t})$, where $\vec{x}$ is the vector of network input values, and $\vec{t}$ is the vector of target network output values.
- η is the learning rate (e.g., 0.05). , D is the number of network inputs, M the number of units in the hidden layer, and K the number of output units. The input from unit p into unit q is denoted x_{qp}, and the weight from unit p to unit q is denoted w_{qp}.

Backpropagation Algorithm

- Create a feed-forward network with D inputs, M hidden units, and K output units.
- Initialize all network weights to small random numbers.
- Until the termination condition is met, Do
  - For each $(\vec{x}, \vec{t})$ in training examples, Do
    - Propagate the input forward through the network:
      1. Input the instance $\vec{x}$ to the network and compute the output y_k, of every unit k in the network.
    - Propagate the errors backward through the network:

Backpropagation Algorithm

- Propagate the errors backward through the network:
  2. For each network output unit k, calculate its error term δ_k
     $\delta_k \leftarrow y_k(1 - y_k)(t_k - y_k)$
  3. For each hidden unit z_j, calculate its error term δ_z
     $$\delta_z \leftarrow z_j(1 - z_j) \sum_{k \in outputs} w_{kj}\delta_k$$
  4. Update each network weight w_{qp}
     $$w_{qp} \leftarrow w_{qp} + \nabla w_{qp}$$
     where $\nabla w_{qp} = \eta \delta_q x_{qp}$