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

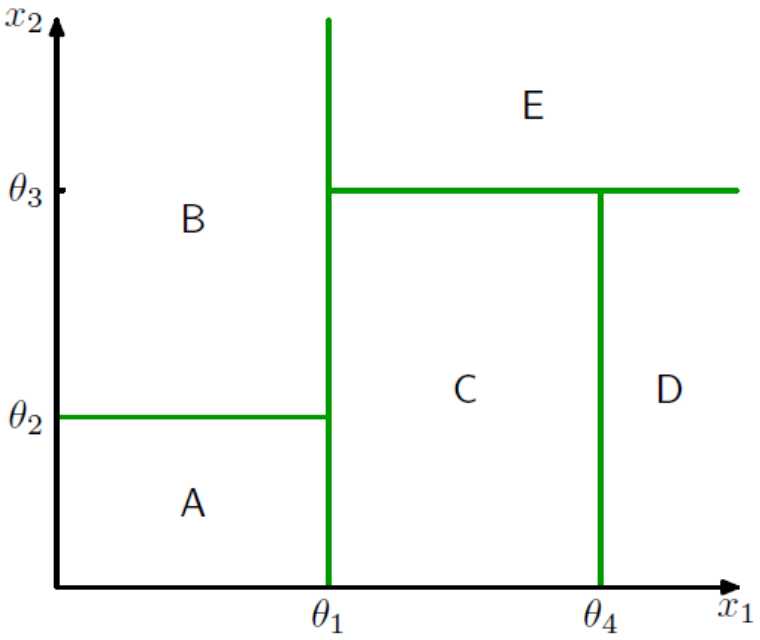
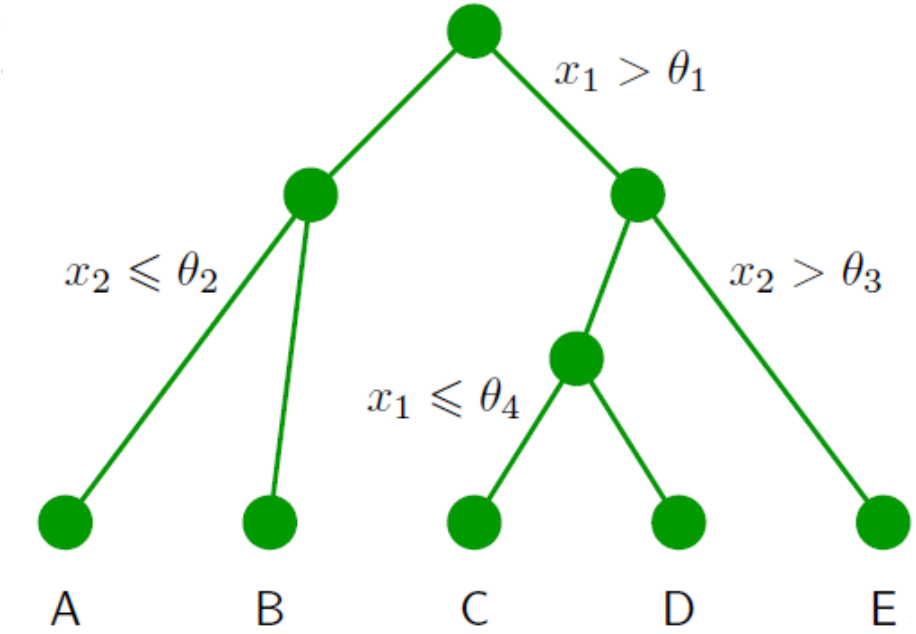


Illustration of a two-dimensional input space that has been partitioned into five regions using axis-aligned boundaries.



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.

Hastie Sec. 9.2 – Also Murphy Sec. 16.2

Regression Trees – popular method for tree-based regression and classification called **CART**

We choose the variable and split-point to achieve the best fit. Then one or both of these regions are split into two more regions, and this process is continued, until some stopping rule is applied. For example, in the top right panel of Figure 9.2, we first split at $X_1 = t_1$. Then the region $X_1 \leq t_1$ is split at $X_2 = t_2$ and the region $X_1 > t_1$ is split at $X_1 = t_3$. Finally, the region $X_1 > t_3$ is split at $X_2 = t_4$. The result of this process is a partition into the five regions $R_1, R_2, \ldots, R_5$ shown in the figure. The corresponding regression model predicts Y with a constant c_m in region R_m, that is,

$$\hat{f}(X) = \sum_{m=1}^{5} c_m I\{(X_1, X_2) \in R_m\}. \tag{9.9}$$

This same model can be represented by the binary tree in the bottom left panel of Figure 9.2. The full dataset sits at the top of the tree. Observations satisfying the condition at each junction are assigned to the left branch, and the others to the right branch. The terminal nodes or leaves of the tree correspond to the regions $R_1, R_2, \ldots, R_5$. The bottom right panel of Figure 9.2 is a perspective plot of the regression surface from this model.

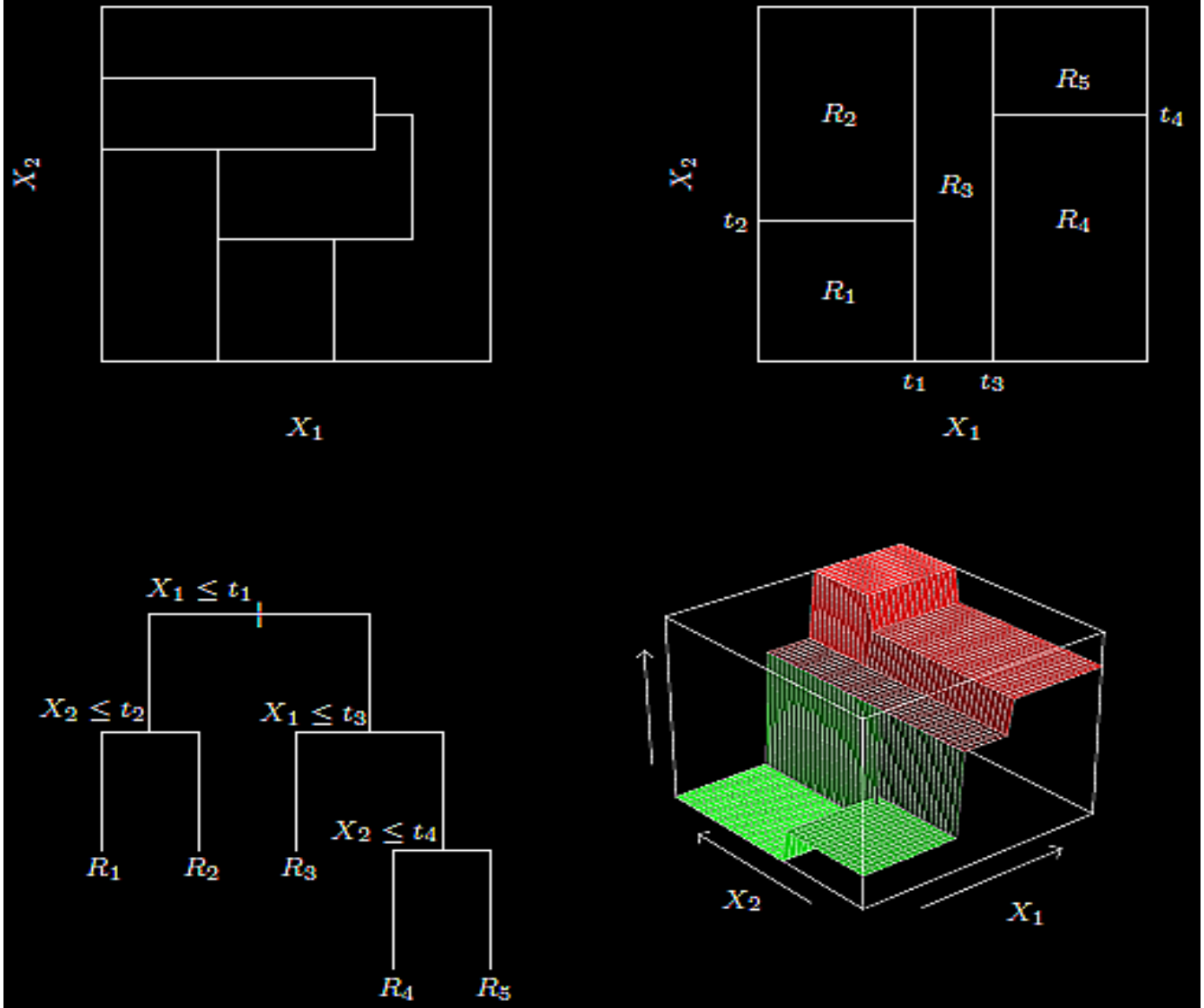
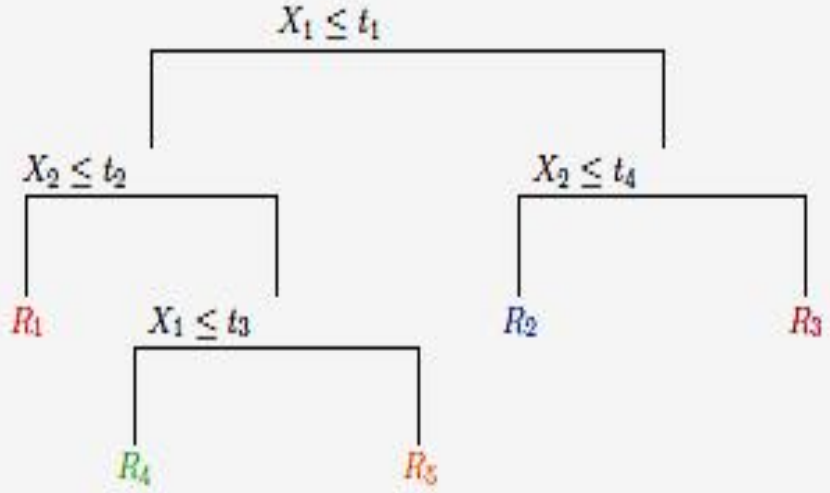
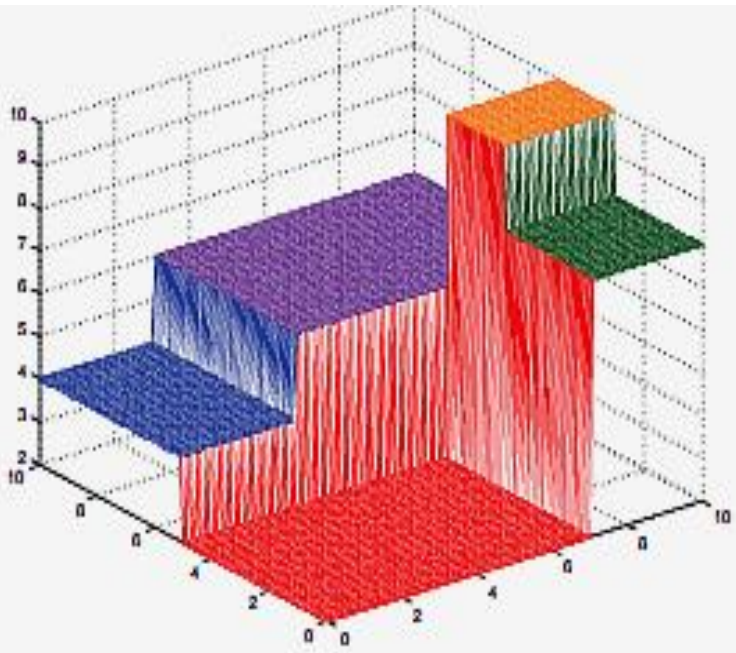


FIGURE 9.2. *Partitions and CART. Top right panel shows a partition of a two-dimensional feature space by recursive binary splitting, as used in CART, applied to some fake data. Top left panel shows a general partition that cannot be obtained from recursive binary splitting. Bottom left panel shows the tree corresponding to the partition in the top right panel, and a perspective plot of the prediction surface appears in the bottom right panel.*



(a)



(b)

Figure 16.1 A simple regression tree on two inputs. Based on Figure 9.2 of (Hastie et al. 2009).

9.2.2 Regression Trees

We now turn to the question of how to grow a regression tree. Our data consists of p inputs and a response, for each of N observations: that is, (x_i, y_i) for $i = 1, 2, \ldots, N$, with $x_i = (x_{i1}, x_{i2}, \ldots, x_{ip})$. The algorithm needs to automatically decide on the splitting variables and split points, and also what topology (shape) the tree should have. Suppose first that we have a partition into M regions $R_1, R_2, \ldots, R_M$, and we model the response as a constant c_m in each region:

$$f(x) = \sum_{m=1}^{M} c_m I(x \in R_m). \tag{9.10}$$

If we adopt as our criterion minimization of the sum of squares $\sum (y_i - f(x_i))^2$, it is easy to see that the best $\hat{c}_m$ is just the average of y_i in region R_m:

$$\hat{c}_m = \text{ave}(y_i | x_i \in R_m). \tag{9.11}$$

Now finding the best binary partition in terms of minimum sum of squares is generally computationally infeasible. Hence we proceed with a greedy algorithm. Starting with all of the data, consider a splitting variable j and split point s, and define the pair of half-planes

$$R_1(j,s) = \{X | X_j \leq s\} \text{ and } R_2(j,s) = \{X | X_j > s\}. \tag{9.12}$$

Then we seek the splitting variable j and split point s that solve

$$\min_{j,\, s} \Big[\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2 \Big]. \tag{9.13}$$

For any choice j and s, the inner minimization is solved by

$$\hat{c}_1 = \text{ave}(y_i | x_i \in R_1(j,s)) \text{ and } \hat{c}_2 = \text{ave}(y_i | x_i \in R_2(j,s)). \tag{9.14}$$

For each splitting variable, the determination of the split point s can be done very quickly and hence by scanning through all of the inputs, determination of the best pair (j, s) is feasible.

Having found the best split, we partition the data into the two resulting regions and repeat the splitting process on each of the two regions. Then this process is repeated on all of the resulting regions.

How large should we grow the tree? Clearly a very large tree might overfit the data, while a small tree might not capture the important structure.

Tree size is a tuning parameter governing the model's complexity, and the optimal tree size should be adaptively chosen from the data. One approach would be to *split tree nodes only if the decrease in sum-of-squares due to the split exceeds some threshold*. This strategy is too short-sighted, however, since a seemingly worthless split might lead to a very good split below it.

The preferred strategy is to grow a large tree T0, stopping the splitting process only when some minimum node size (say 5) is reached. Then this large tree is pruned using cost-complexity pruning, which we describe later

9.2.3 Classification Trees

If the target is a classification outcome taking values $1, 2, \ldots, K$, the only changes needed in the tree algorithm pertain to the criteria for splitting classification. In a node m, representing a region R_m with N_m observations, let

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k),$$

the proportion of class k observations in node m. We classify the observations in node m to class $k(m) = \arg\max_k \hat{p}_{mk}$, the majority class in node m. Different measures $Q_m(T)$ of node impurity include the following:

Misclassification error:	$\frac{1}{N_m} \sum_{i \in R_m} I(y_i \neq k(m)) = 1 - \hat{p}_{mk(m)}.$
Gini index:	$\sum_{k \neq k'} \hat{p}_{mk} \hat{p}_{mk'} = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}).$
Cross-entropy or deviance:	$-\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}.$

(9.17)

vations in node m to class $k(m) = \arg\max_k \hat{p}_{mk}$, the majority class in node m. Different measures $Q_m(T)$ of node impurity include the following:

$$\begin{aligned}
&\text{Misclassification error:} && \frac{1}{N_m}\sum_{i\in R_m} I(y_i \neq k(m)) = 1 - \hat{p}_{mk(m)}. \\
&\text{Gini index:} && \sum_{k\neq k'} \hat{p}_{mk}\hat{p}_{mk'} = \sum_{k=1}^{K} \hat{p}_{mk}(1-\hat{p}_{mk}). \\
&\text{Cross-entropy or deviance:} && -\sum_{k=1}^{K} \hat{p}_{mk}\log\hat{p}_{mk}.
\end{aligned} \tag{9.17}$$

For two classes, if p is the proportion in the second class, these three measures are $1 - \max(p, 1-p)$, $2p(1-p)$ and $-p\log p - (1-p)\log(1-p)$,

The split function chooses the best feature, and the best value for that feature, as follows:

$$(j^*, t^*) = \arg \min_{j \in \{1,\ldots,D\}} \min_{t \in T_j} \text{cost}(\{\mathbf{x}_i, y_i : x_{ij} \leq t\}) + \text{cost}(\{\mathbf{x}_i, y_i : x_{ij} > t\}) \quad (16$$

The function that checks if a node is worth splitting can use several stopping heuristics, such as the following:

- is the reduction in cost too small? Typically we define the gain of using a feature to be a normalized measure of the reduction in cost:

$$\Delta \triangleq \text{cost}(\mathcal{D}) - \left(\frac{|\mathcal{D}_L|}{|\mathcal{D}|} \text{cost}(\mathcal{D}_L) + \frac{|\mathcal{D}_R|}{|\mathcal{D}|} \text{cost}(\mathcal{D}_R) \right) \quad (16.6)$$

- has the tree exceeded the maximum desired depth?
- is the distribution of the response in either $\mathcal{D}_L$ or $\mathcal{D}_R$ sufficiently homogeneous (e.g., all labels are the same, so the distribution is pure)?
- is the number of examples in either $\mathcal{D}_L$ or $\mathcal{D}_R$ too small?

Regression cost

In the regression setting, we define the cost as follows:

$$\text{cost}(\mathcal{D}) = \sum_{i \in \mathcal{D}} (y_i - \overline{y})^2$$

Classification cost

In the classification setting, there are several ways to measure the quality of a split. First, we fit a multinoulli model to the data in the leaf satisfying the test $X_j < t$ by estimating the class-conditional probabilities as follows:

$$\hat{\pi}_c = \frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} \mathbb{I}(y_i = c) \tag{16.8}$$

where $\mathcal{D}$ is the data in the leaf. Given this, there are several common error measures for evaluating a proposed partition:

- **Misclassification rate.** We define the most probable class label as $\hat{y}_c = \text{argmax}_c \, \hat{\pi}_c$. The corresponding error rate is then

$$\frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} \mathbb{I}(y_i \neq \hat{y}) = 1 - \hat{\pi}_{\hat{y}} \tag{16.9}$$

- **Entropy, or deviance:**

$$\mathbb{H}(\hat{\pi}) = -\sum_{c=1}^{C} \hat{\pi}_c \log \hat{\pi}_c \tag{16.10}$$

Note that minimizing the entropy is equivalent to **maximizing the information gain** (Quinlan 1986) between test $X_j < t$ and the class label Y, defined by

$$\begin{aligned} \text{infoGain}(X_j < t, Y) &\triangleq \mathbb{H}(Y) - \mathbb{H}(Y|X_j < t) && (16.11) \\ &= \left(-\sum_c p(y=c) \log p(y=c)\right) && (16.12) \\ &\quad + \left(\sum_c p(y=c|X_j < t) \log p(c|X_j < t)\right) && (16.13) \end{aligned}$$

since $\hat{\pi}_c$ is an MLE for the distribution $p(c|X_j < t)$.[1]

- **Gini index**

$$\sum_{c=1}^{C} \hat{\pi}_c(1-\hat{\pi}_c) = \sum_c \hat{\pi}_c - \sum_c \hat{\pi}_c^2 = 1 - \sum_c \hat{\pi}_c^2 \tag{16.14}$$

This is the expected error rate. To see this, note that $\hat{\pi}_c$ is the probability a random entry in the leaf belongs to class c, and $(1 - \hat{\pi}_c$ is the probability it would be misclassified.

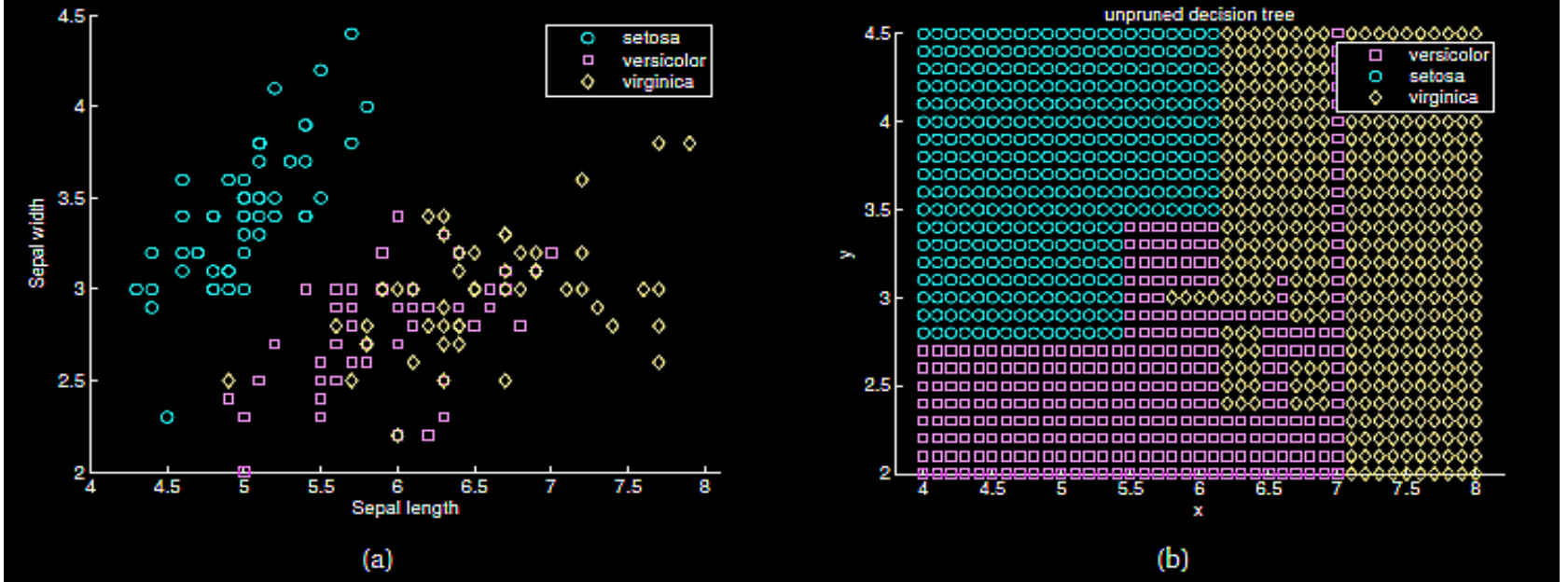


Figure 16.4 (a) Iris data. We only show the first two features, sepal length and sepal width, and ignore petal length and petal width. (b) Decision boundaries induced by the decision tree in Figure 16.5(a).

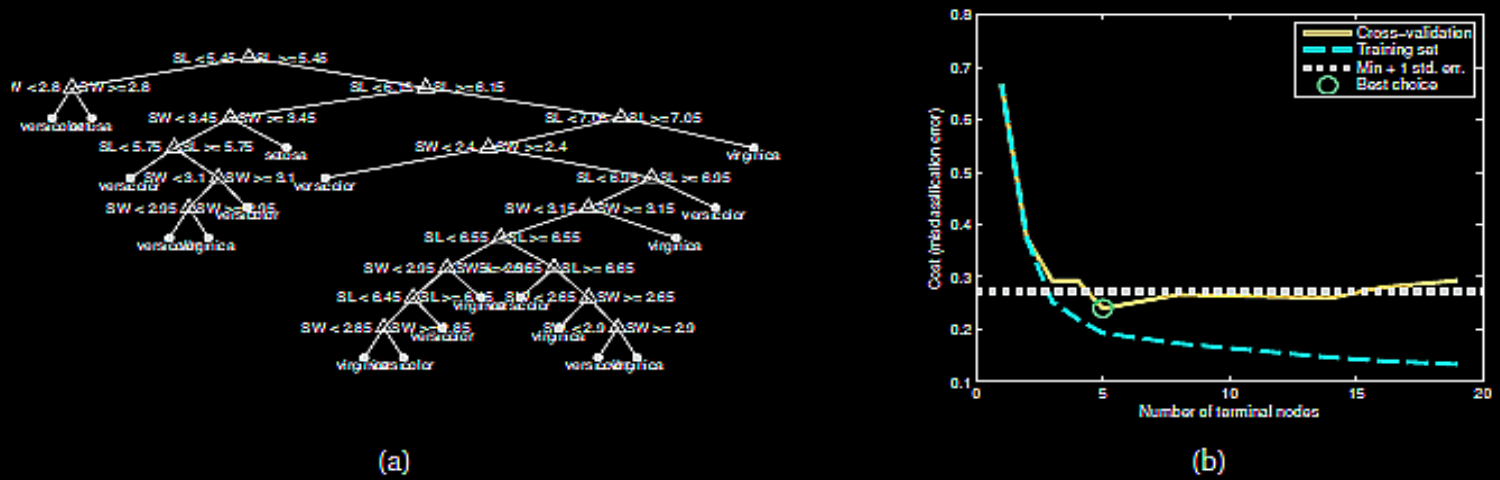


Figure 16.5 (a) Unpruned decision tree for Iris data. (b) Plot of misclassification error rate vs depth of tree. Figure generated by `dtreeDemoIris`.

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*

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

- 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

We define a subtree $T \subset T_0$ to be any tree that can be obtained by pruning T_0, that is, collapsing any number of its internal (non-terminal) nodes. We index terminal nodes by m, with node m representing region R_m. Let $|T|$ denote the number of terminal nodes in T. Letting

$$\begin{aligned} N_m &= \#\{x_i \in R_m\}, \\ \hat{c}_m &= \frac{1}{N_m} \sum_{x_i \in R_m} y_i, \\ Q_m(T) &= \frac{1}{N_m} \sum_{x_i \in R_m} (y_i - \hat{c}_m)^2, \end{aligned} \tag{9.15}$$

we define the cost complexity criterion

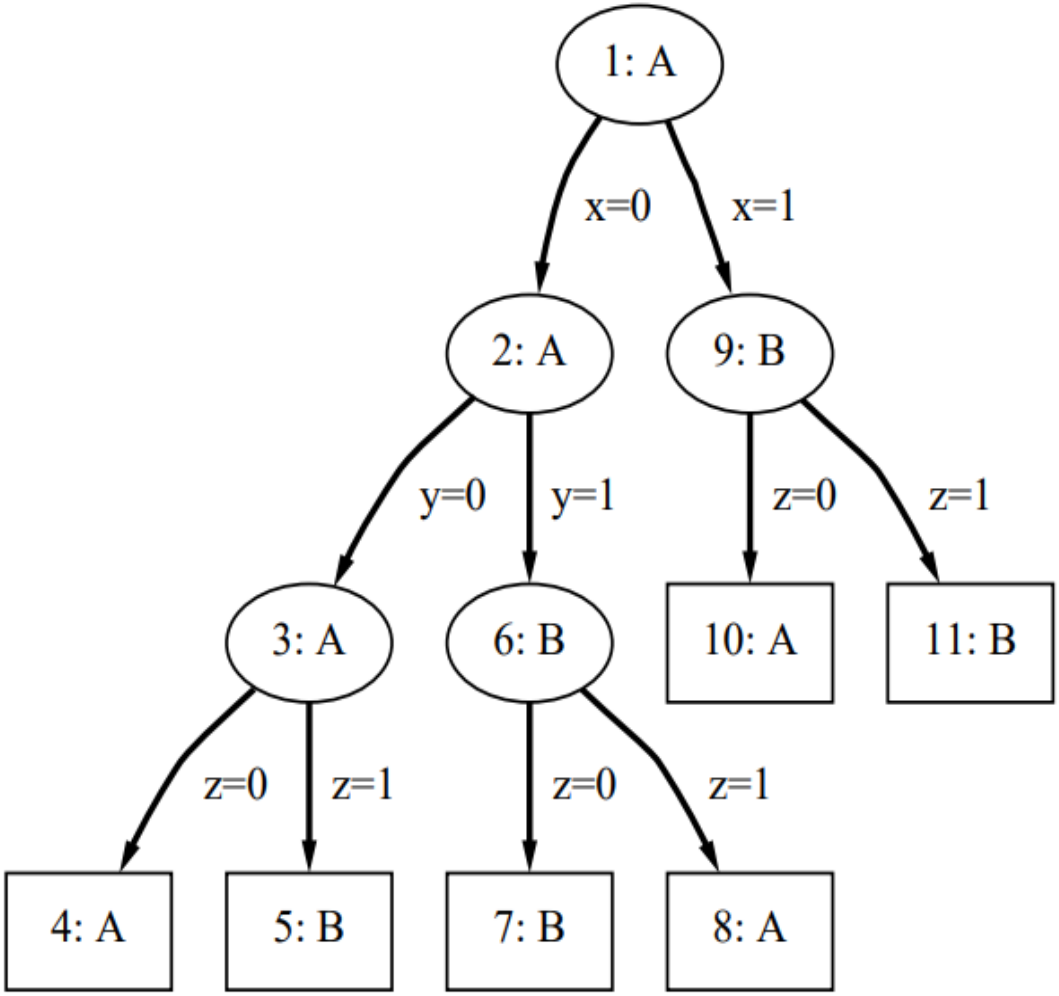
$$C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha|T|. \tag{9.16}$$

The idea is to find, for each α, the subtree $T_\alpha \subseteq T_0$ to minimize $C_\alpha(T)$. The tuning parameter $\alpha \geq 0$ governs the tradeoff between tree size and its goodness of fit to the data. Large values of α result in smaller trees T_α, and conversely for smaller values of α. As the notation suggests, with $\alpha = 0$ the

Either **the Gini index or cross-entropy** should be used when growing the tree.

To guide cost-complexity pruning, any of the three measures can be used, but typically it is the misclassification rate.

The Gini index can be interpreted in two interesting ways. Rather than classify observations to the majority class in the node, we could classify them to class k with probability $\hat{p}_{mk}$. Then the training error rate of this rule in the node is $\sum_{k \neq k'} \hat{p}_{mk}\hat{p}_{mk'}$—the Gini index. Similarly, if we code each observation as 1 for class k and zero otherwise, the variance over the node of this 0-1 response is $\hat{p}_{mk}(1 - \hat{p}_{mk})$. Summing over classes k again gives the Gini index.

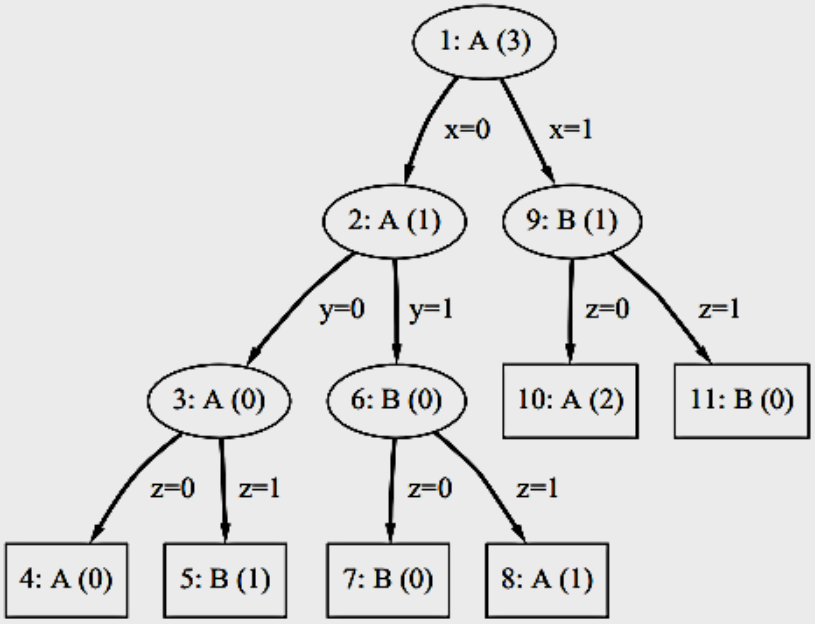


A decision tree with two classes A and B (with node numbers and class labels)

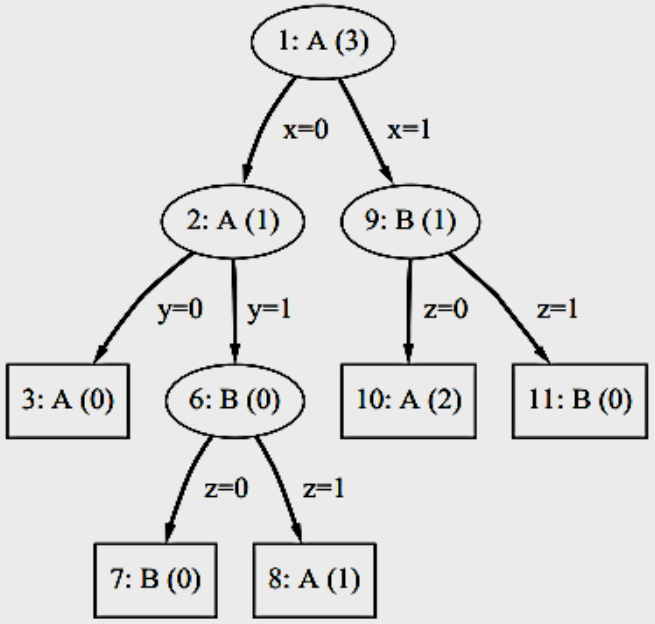
x	y	z	class
0	0	1	A
0	1	1	B
1	1	0	B
1	0	0	B
1	1	1	A

An example pruning set

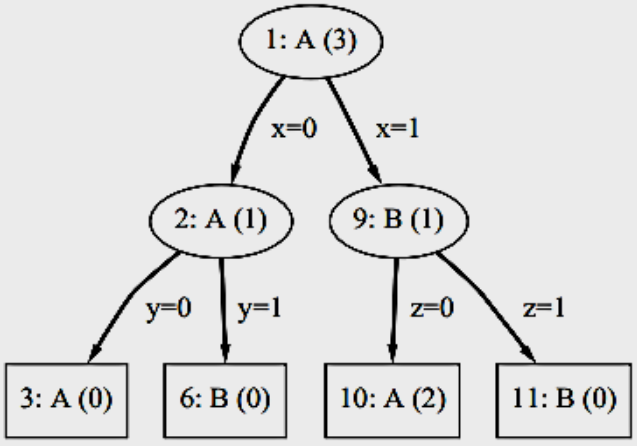
- The idea is to hold out some of the available instances—the “pruning set”—when the tree is built, and prune the tree until the classification error on these independent instances starts to increase.
- Because **the instances in the pruning set are not used for building the decision tree**, they provide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



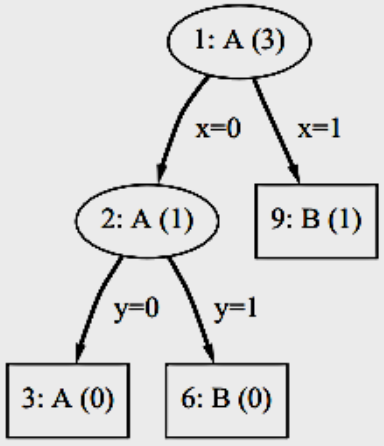
(a)



(b)



(c)



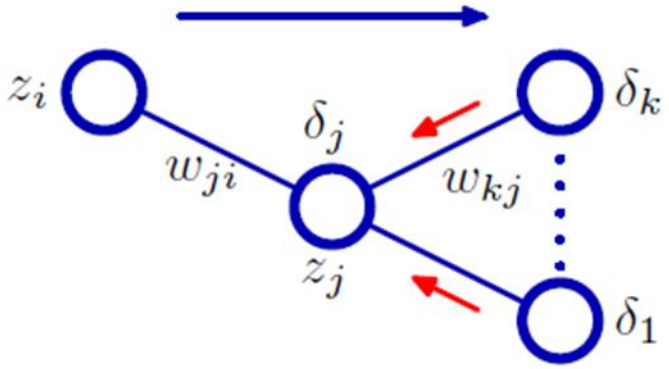
(d)

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 - Notations

- $\mathbf{x}$ - input; $\mathbf{y}$ - output; $\mathbf{w}$ - weights
- N - no. of samples
- $\mathbf{x}_n$ - n^{th} sample
- $\mathbf{t}_n$ - target output for $\mathbf{x}_n$
- $\mathbf{y}_n = \mathbf{y}(\mathbf{x}_n, \mathbf{w})$ - predicted output for $\mathbf{x}_n$
- M - no. of nodes in the hidden layer
- K - no. of classes
- D - dimension of input $\mathbf{x}$
- w_{ji} -weight from node i in layer l to node j in layer $(l+1)$
- a_j - linear combination of input variables with weights (pre-activation)
- $\mathbf{y}_k(\mathbf{x}, \mathbf{w})$ - output at node k (output layer)
- $\mathbf{y}_{nk} = \mathbf{y}_k(\mathbf{x}_n, \mathbf{w})$ - output at node k for n^{th} input ; likewise for $\mathbf{t}_{nk}$
- τ - time step
- z_j - activation of a_j ($h(a_j)$ where h is the activation function)
- δ_k - error term with respect to output node k
- δ_j - error term with respect to hidden node j

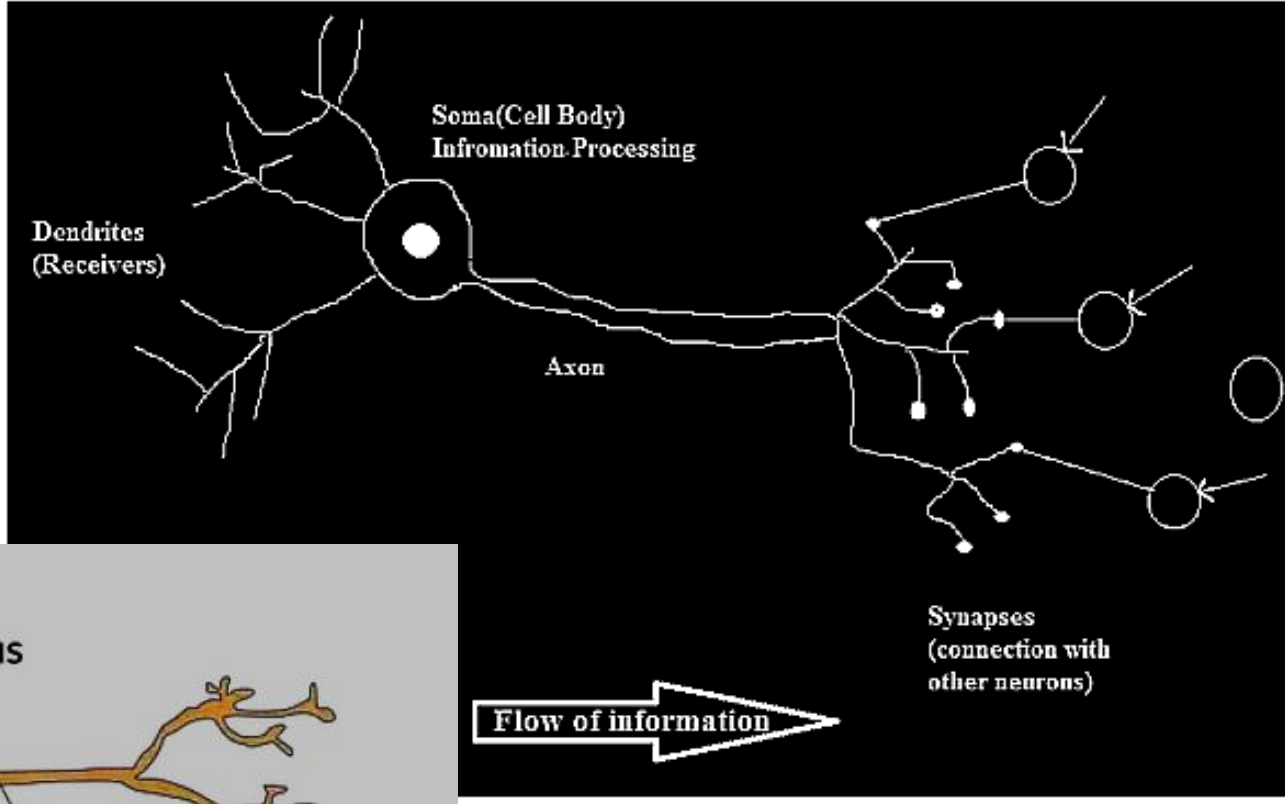


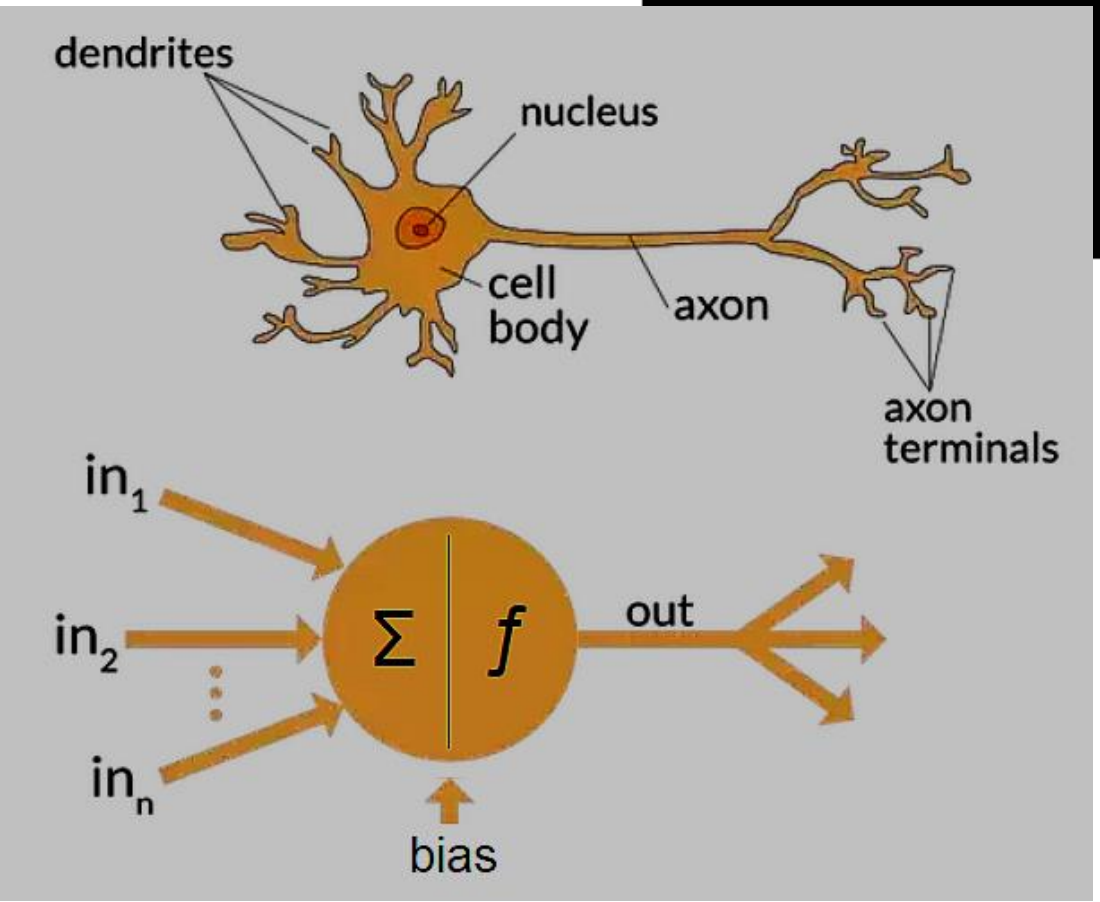
Feed-forward Network Functions

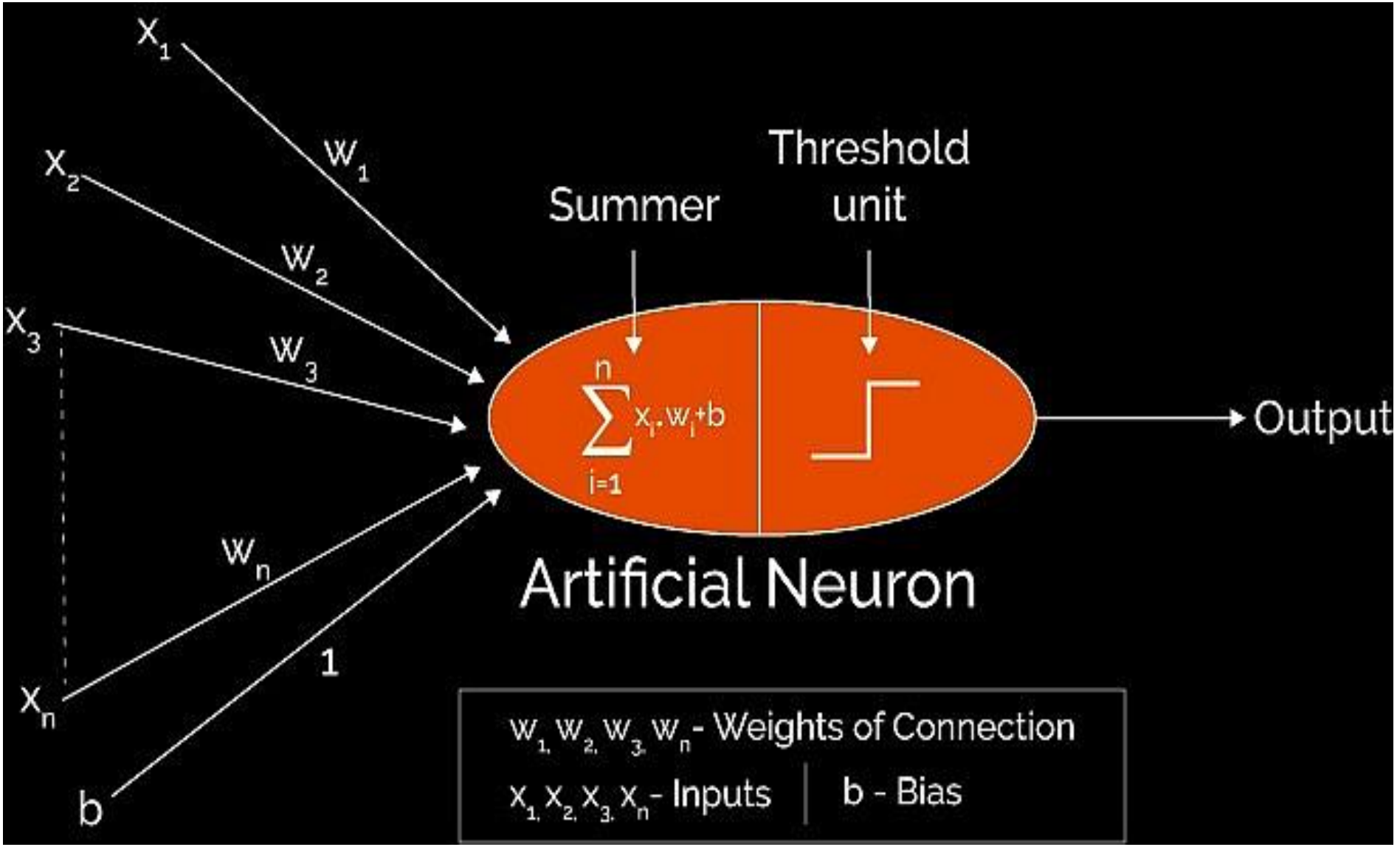
- The linear models for regression and classification are based on linear combinations of fixed nonlinear basis functions $\phi_j(\mathbf{x})$ and take the form

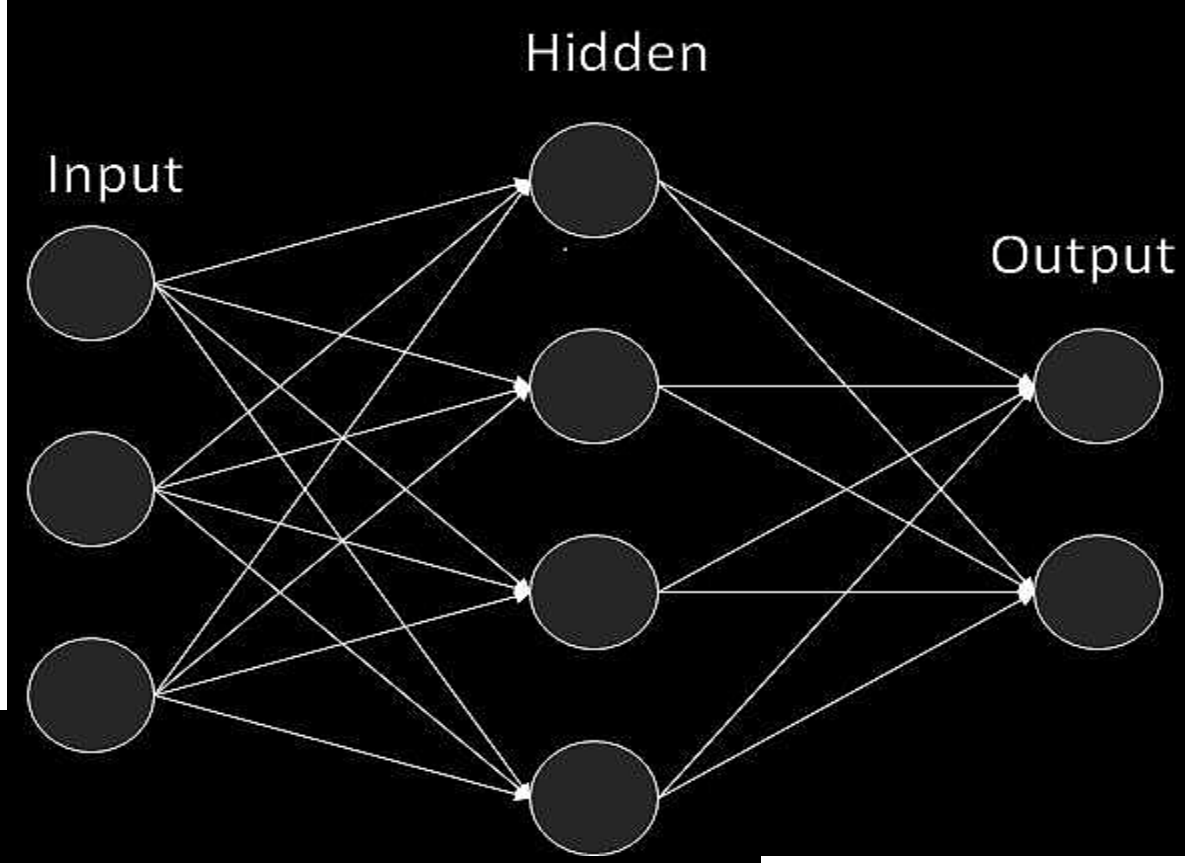
$$y(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=1}^{M} w_j \phi_j(\mathbf{x})\right)$$

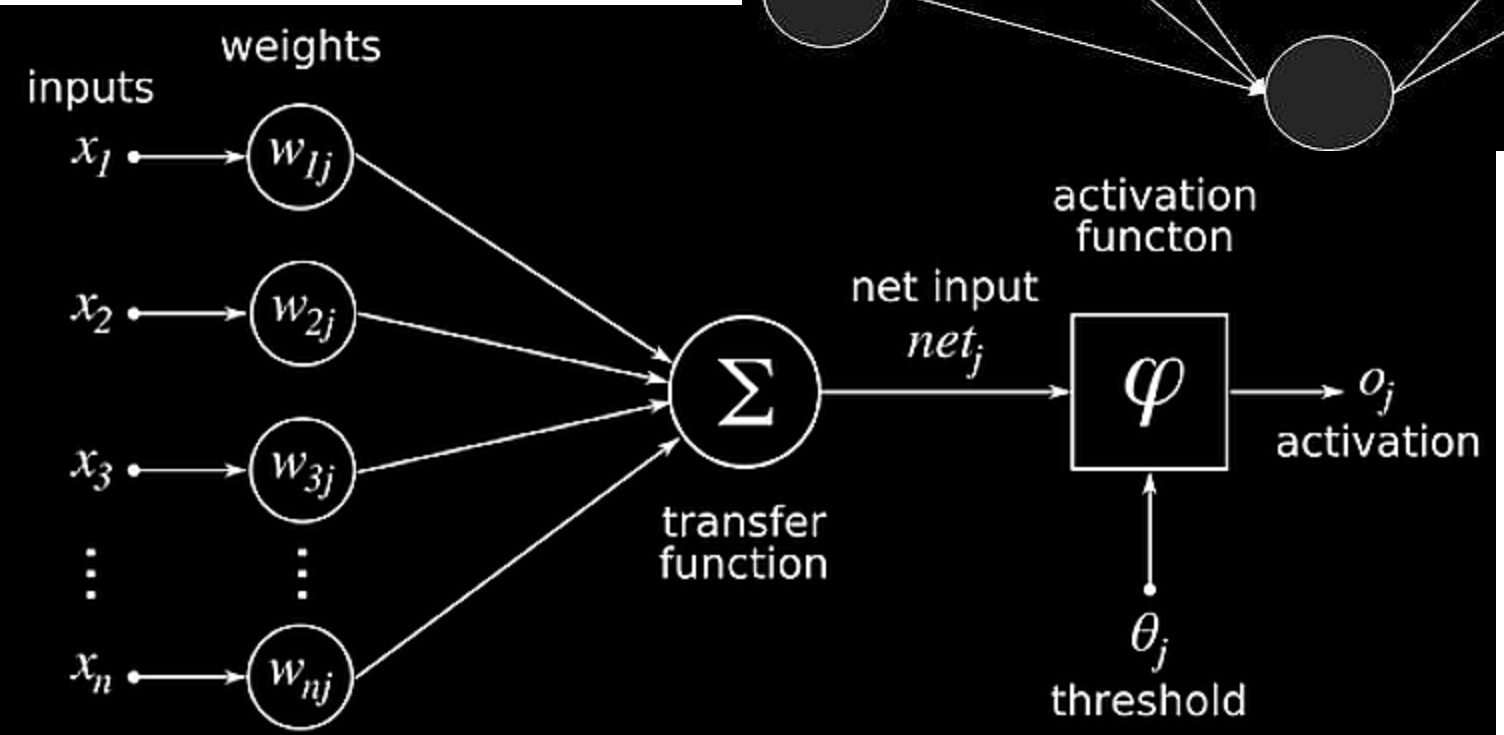
- where $f(\cdot)$ is a nonlinear activation function in the case of classification and is the identity in the case of regression.
- Our goal is to extend this model by making the basis functions $\phi_j(\mathbf{x})$ depend on parameters and then to allow these parameters to be adjusted, along with the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

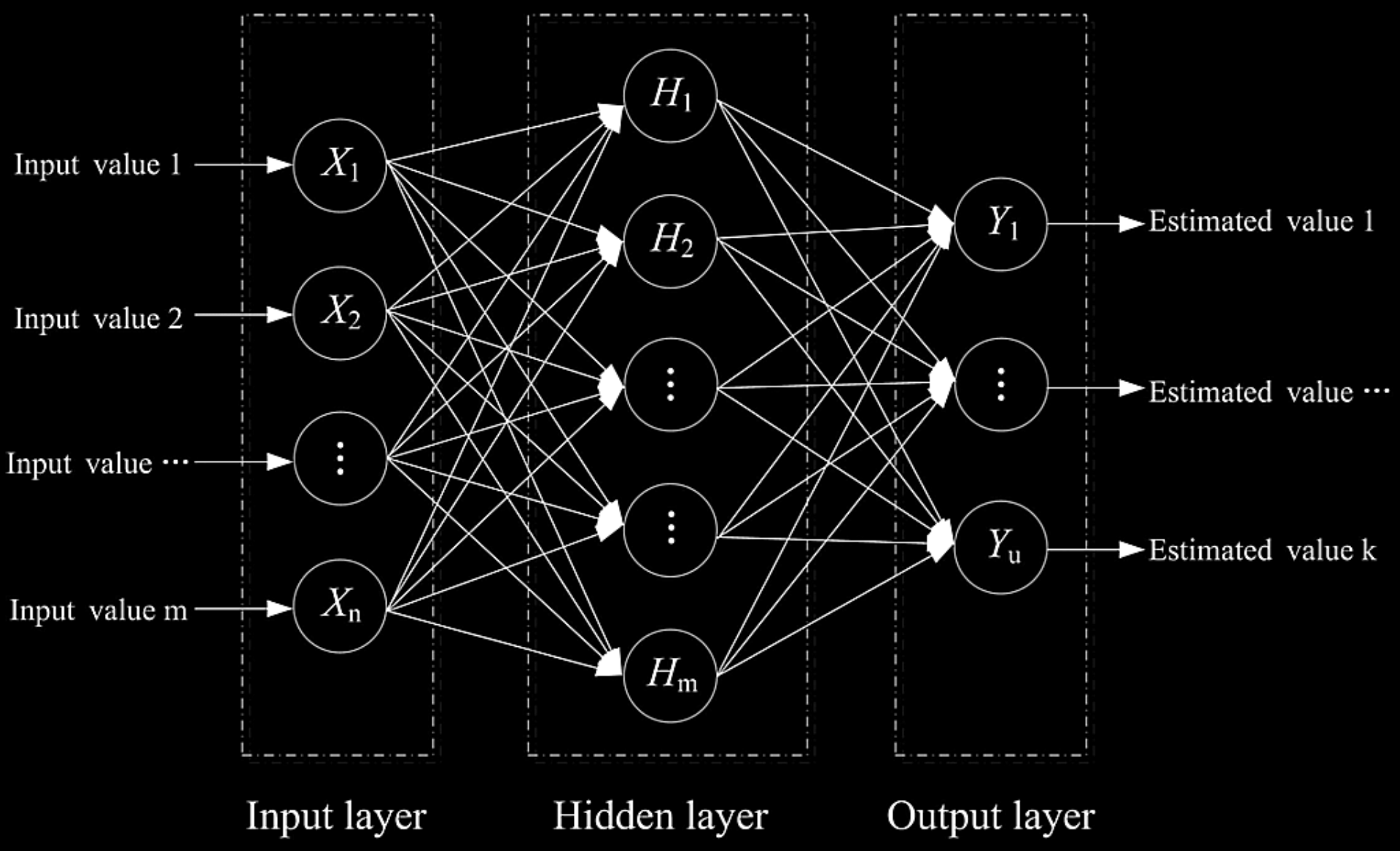

Soma(Cell Body)
Infromation Processing
Dendrites
(Receivers)
Axon
Synapses
(connection with
other neurons)
Flow of information

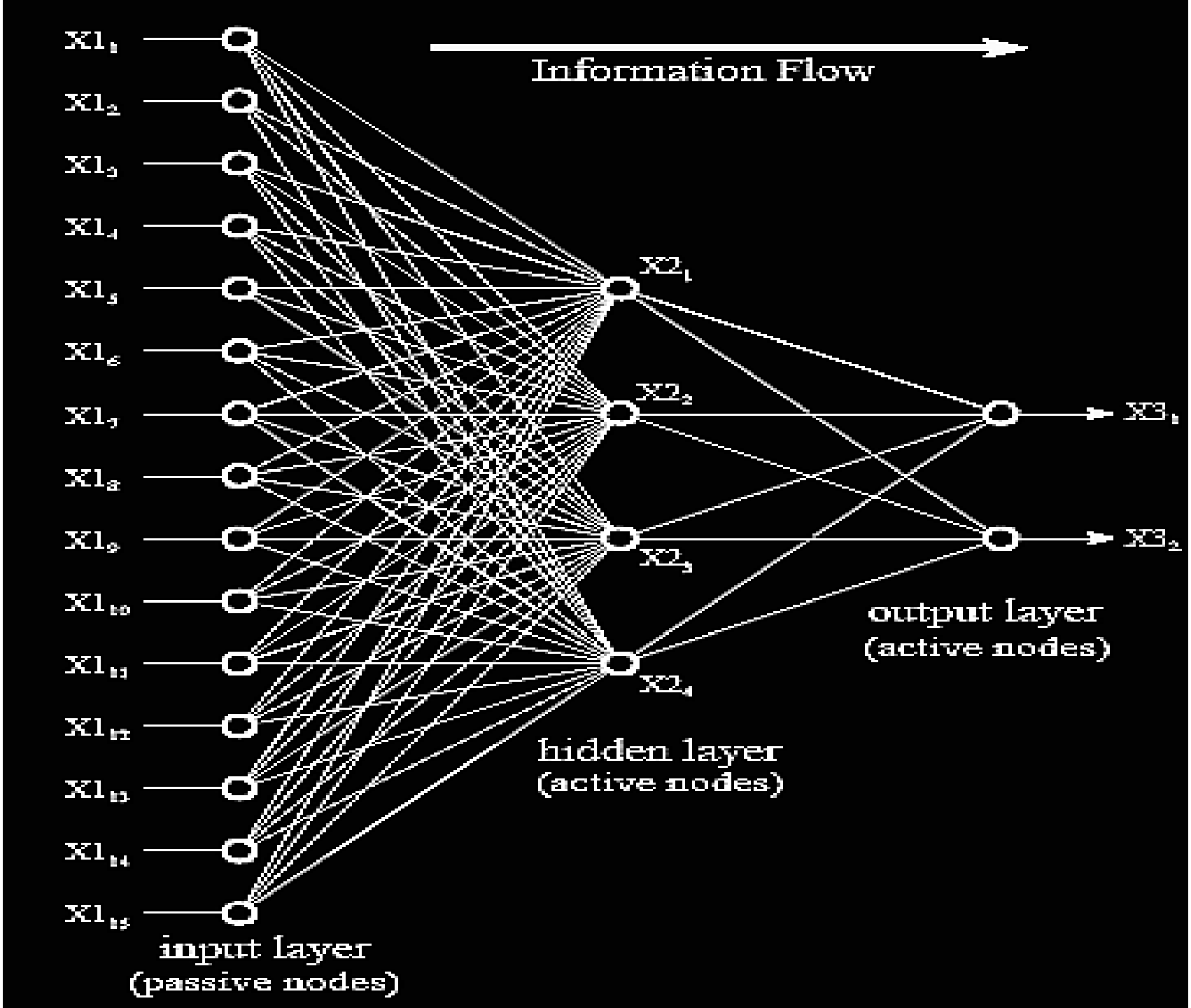

dendrites
nucleus
cell
body
axon
axon
terminals
in1
in2
inn
Σ
f
out
bias

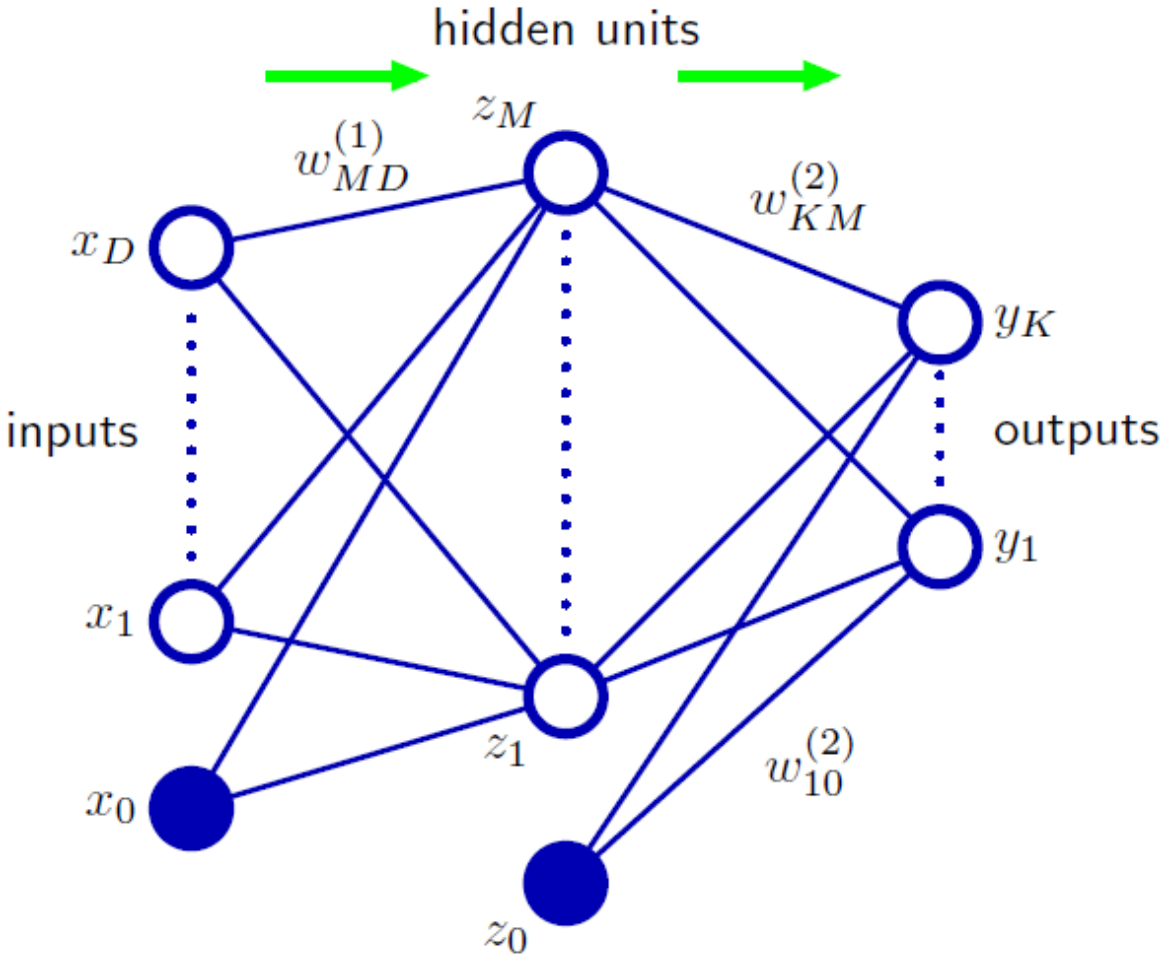

x_1
x_2
x_3
x_n
1
b
w_1
w_2
w_3
w_n
Summer
Threshold unit
$\sum_{i=1}^{n} x_i.w_i+b$
Output
Artificial Neuron
w_1, w_2, w_3, w_n - Weights of Connection
x_1, x_2, x_3, x_n - Inputs
b - Bias


Hidden
Input
Output


weights
inputs
x_1
x_2
x_3
x_n
w_{1j}
w_{2j}
w_{3j}
w_{nj}
Σ
transfer
function
net input
net_j
activation
functon
φ
o_j
activation
θ_j
threshold


Input value 1
Input value 2
Input value ···
Input value m
X_1
X_2
X_n
H_1
H_2
H_m
Y_1
Y_u
Estimated value 1
Estimated value ···
Estimated value k
Input layer
Hidden layer
Output layer


Information Flow
X1₁
X1₂
X1₃
X1₄
X1₅
X1₆
X1₇
X1₈
X1₉
X1₁₀
X1₁₁
X1₁₂
X1₁₃
X1₁₄
X1₁₅
X2₁
X2₂
X2₃
X2₄
X3₁
X3₂
input layer
(passive nodes)
hidden layer
(active nodes)
output layer
(active nodes)



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 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

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

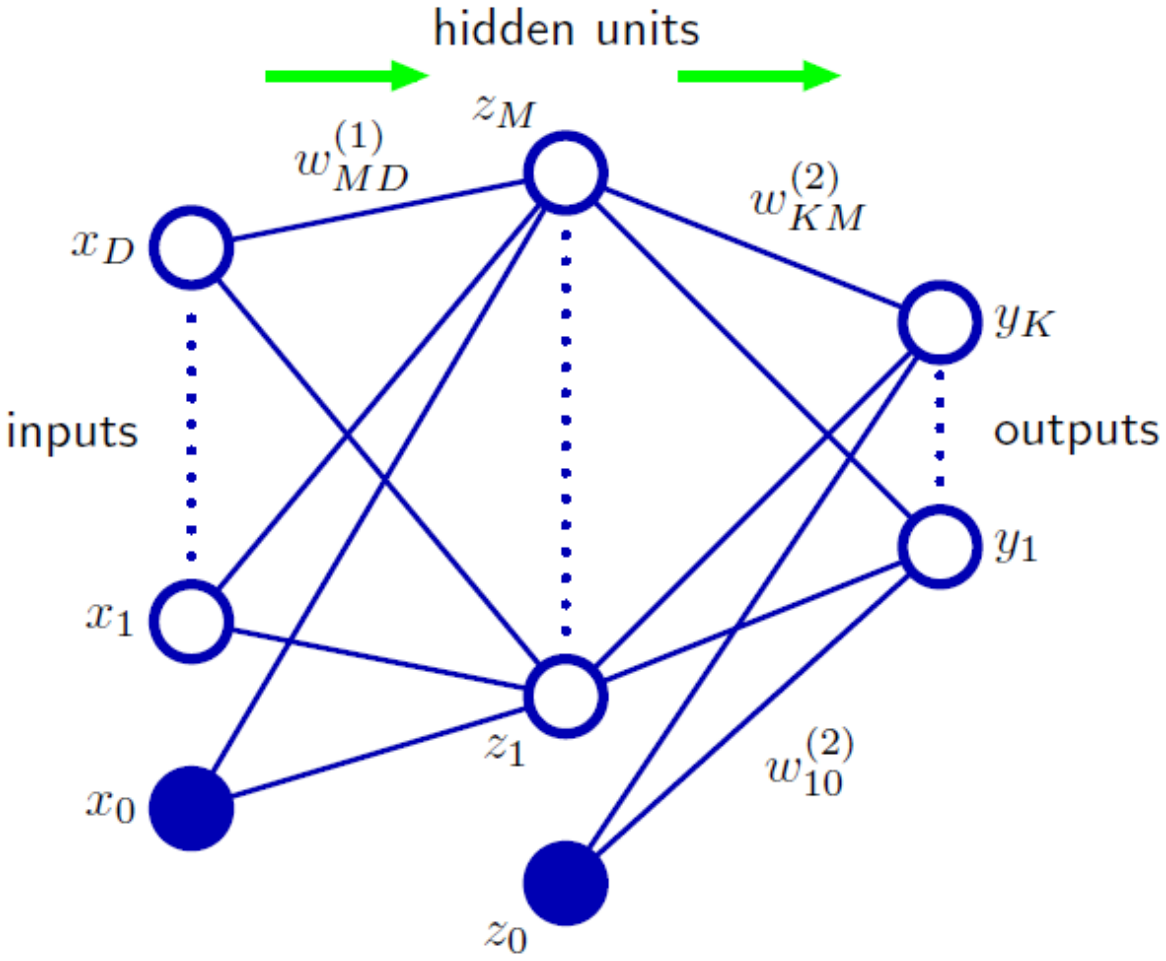
Feed-forward Network Functions

- 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.

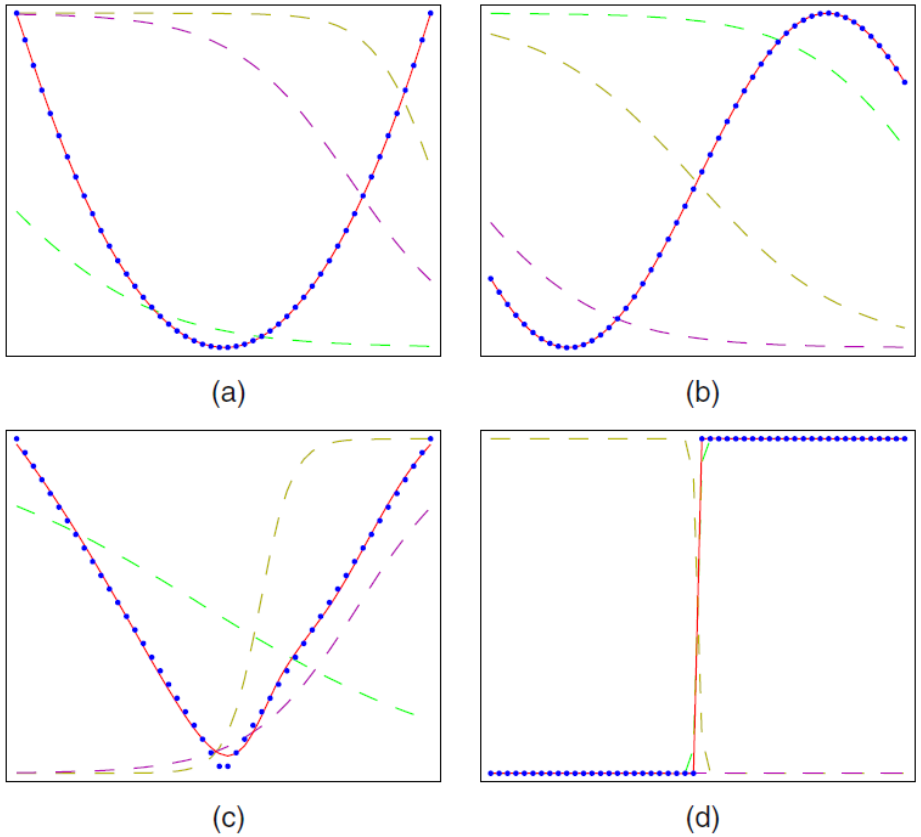


Illustration of the capability of a multilayer perceptron to approximate four different functions comprising (a) $f(x) = x^2$, (b) $f(x) = \sin(x)$, (c), $f(x) = |x|$, and (d) $f(x) = H(x)$ where $H(x)$ is the Heaviside step function. In each case, $N = 50$ data points, shown as blue dots, have been 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

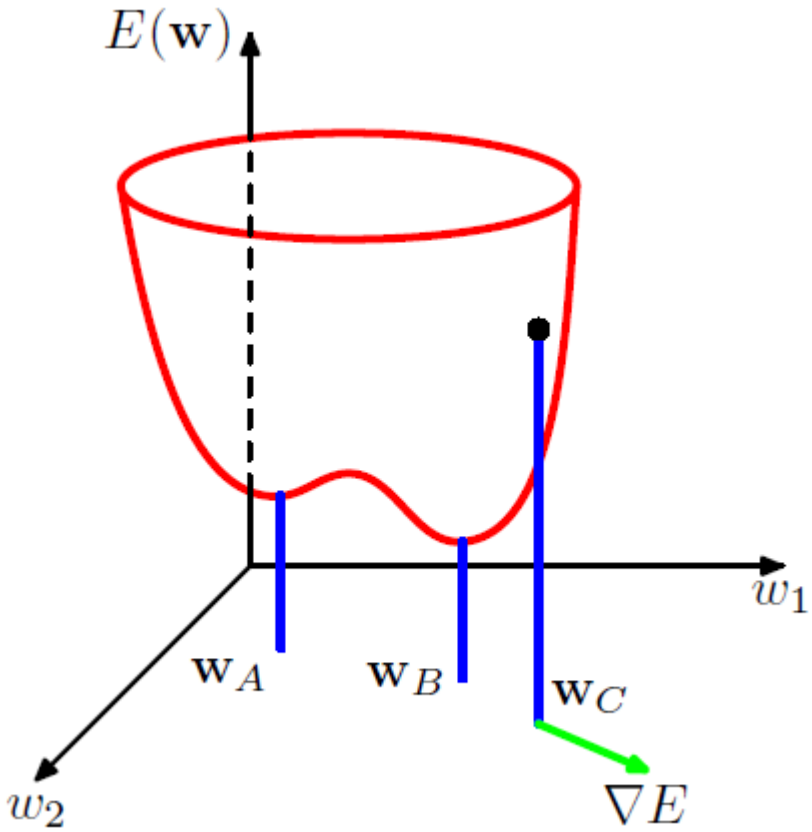
$$E(\mathbf{w}) = -\sum_{n=1}^{N}\sum_{k=1}^{K}\{t_{nk}\ln y_{nk} + (1-t_{nk})\ln(1-y_{nk})\}$$

  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 with* τ being the timestep.

- 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

$$\delta_k = y_k - t_k$$

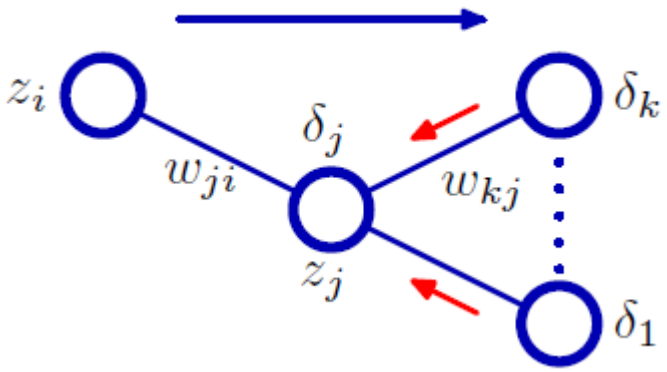


Illustration of the calculation of δ_j for hidden unit j by backpropagation of the δ's from those units k to which unit j sends connections. The blue arrow denotes the direction of information flow during forward propagation, and the red arrows indicate the backward propagation of error information.

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

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

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j} \tag{5.55}$$

- From equation 5.51 which is

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} \tag{5.51}$$

- $\frac{\partial E_n}{\partial a_k}$ can be written as δ_k. i.e., $\frac{\partial E_n}{\partial a_k} = \delta_k$ ----- (1)
  - According to chain rule, $\frac{\partial a_k}{\partial a_j} = \frac{\partial a_k}{\partial z_j} \cdot \frac{\partial z_j}{\partial a_j}$
- From equations 5.48 and 5.49 we have:

$$a_j = \sum_i w_{ji} z_i \tag{5.48}$$

$$z_j = h(a_j). \tag{5.49}$$

- $\frac{\partial a_k}{\partial z_j} = w_{kj}$ and $\frac{\partial z_j}{\partial a_j} = h'(a_j)$ ----- (2)
- Substituting (1) and (2) in eqn 5.55 we get

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k \tag{5.56}$$

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 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:

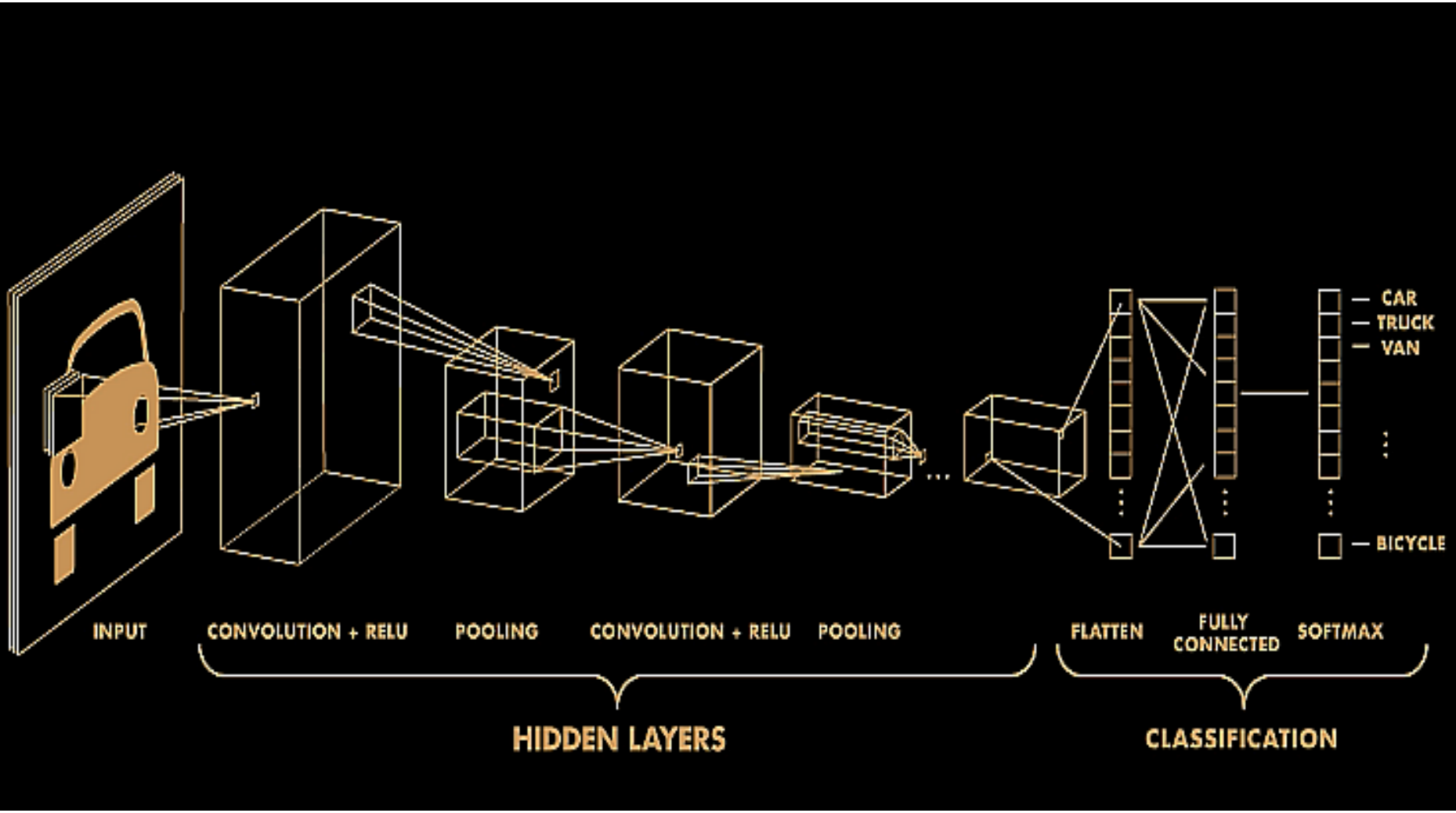
Backpropagation Algorithm

- Propagate the errors backward through the network:
  2. For each network output unit k, calculate its error term δ_k

     $$\delta_k \leftarrow (t_k - y_k)$$
  3. For each hidden unit z_j, calculate its error term δ_z

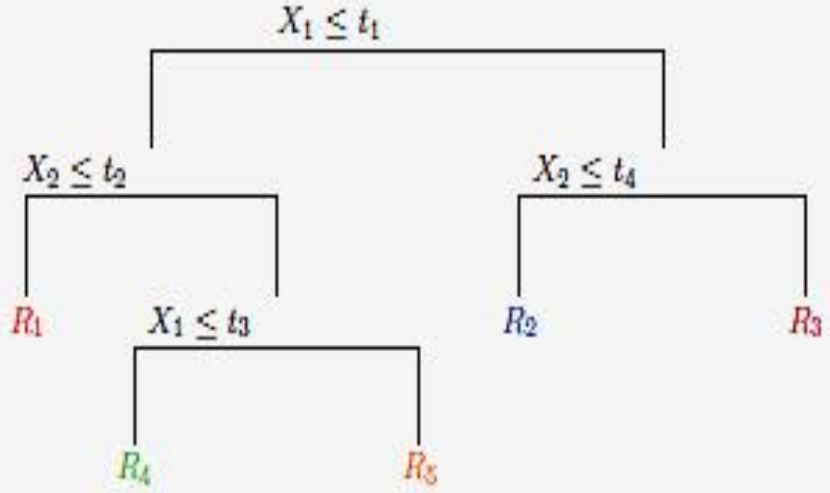
     $$\delta_z \leftarrow h'(a_j) \sum_{k \in outputs} w_{kj}\delta_k$$
  4. Update each network weight w_{qp}

     $$w_{qp} \leftarrow w_{qp} - \eta \nabla \mathrm{E}(w_{qp})$$

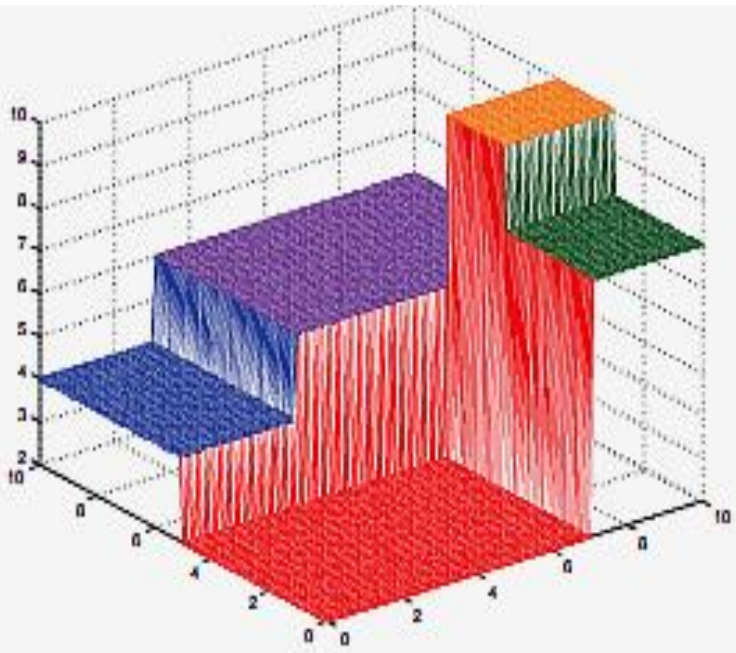

CAR
TRUCK
VAN
BICYCLE
INPUT
CONVOLUTION + RELU
POOLING
CONVOLUTION + RELU
POOLING
FLATTEN
FULLY CONNECTED
SOFTMAX
HIDDEN LAYERS
CLASSIFICATION

Various types of ANN Architectures:

- **Boltzmann Machine,**
- **Hopfield Network**
- **CAM (Content Addressable memories);**
- **BAM (Bidirectional associative memory)**
- **SOM (self-organizing maps)**
- **Deep Belief Networks**
- **RBM, RBF**
- **CNN, Relu**
- **GAN**
- **Auto-encoders (AE)**
- **LSTM**



(a)



(b)

Figure 16.1 A simple regression tree on two inputs. Based on Figure 9.2 of (Hastie et al. 2009).