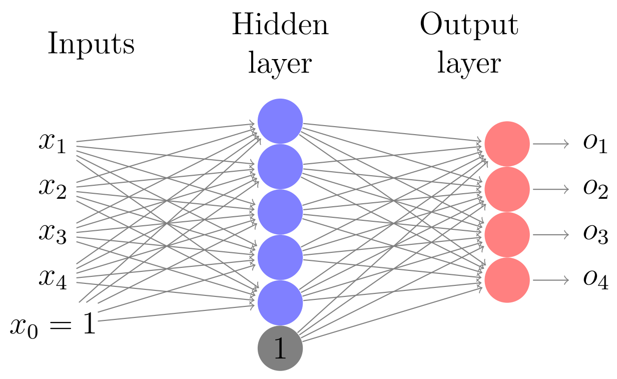

3 Feed Forward Neural Networks

- $d + 1$ input nodes (including bias)
- m hidden nodes
- k output nodes
- At hidden nodes: $\mathbf{w_j}, 1 \leq j \leq m$, $\mathbf{w_j} \in \mathbb{R}^{d+1}$
- At output nodes: $\mathbf{w_l}, 1 \leq l \leq k$, $\mathbf{w_l} \in \mathbb{R}^{m+1}$

The multi-layer neural network shown above is used in a feed forward mode, i.e., information only flows in one direction (forward). Each hidden node "collects" the inputs from all input nodes and computes a weighted sum of the inputs and then applies the sigmoid function to the weighted sum. The output of each hidden node is forwarded to every output node. The output node "collects" the inputs (from hidden layer nodes) and computes a weighted sum of its inputs and then applies the sigmoid function to obtain the final output. The class corresponding to the output node with the largest output value is assigned as the predicted class for the input.

For implementation, one can even represent the weights as two matrices, $W^{(1)}$ $(m \times d + 1)$ and $W^{(2)}$ $(k \times m + 1)$.

4 Backpropagation

- Assume that the network structure is predetermined (number of hidden nodes and interconnections)

- Objective function for N training examples:

$$J = \sum_{i=1}^{N} J_i = \frac{1}{2}\sum_{i=1}^{N}\sum_{l=1}^{k}(y_{il} - o_{il})^2$$

- y_{il} - Target value associated with l^{th} class for input $(\mathbf{x}_i)$
- $y_{il} = 1$ when k is true class for $\mathbf{x}_i$, and 0 otherwise
- o_{il} - Predicted output value at l^{th} output node for $\mathbf{x}_i$

What are we learning?
Weight vectors for all output and hidden nodes that minimize J

The first question that comes to mind is, why not use a standard gradient descent based minimization as the one that we saw in single perceptron unit learning. The reason is that the output at every output node (o_l) is directly dependent on the weights associated with the output nodes but not with weights at hidden nodes. But the input values are "used" by the hidden nodes and are not "visible" to the output nodes. To learn all the weights simultaneously, direct minimization is not possible. Advanced methods such as **Backpropagation** need to be employed.

1. Initialize all weights to *small values*
2. For each training example, $\langle \mathbf{x}, \mathbf{y} \rangle$:
   (a) **Propagate input forward** through the network
   (b) **Propagate errors backward** through the network

Gradient Descent

- Move in the opposite direction of the **gradient** of the objective function
- $-\eta \nabla J$

$$\nabla J = \sum_{i=1}^{N} \nabla J_i$$