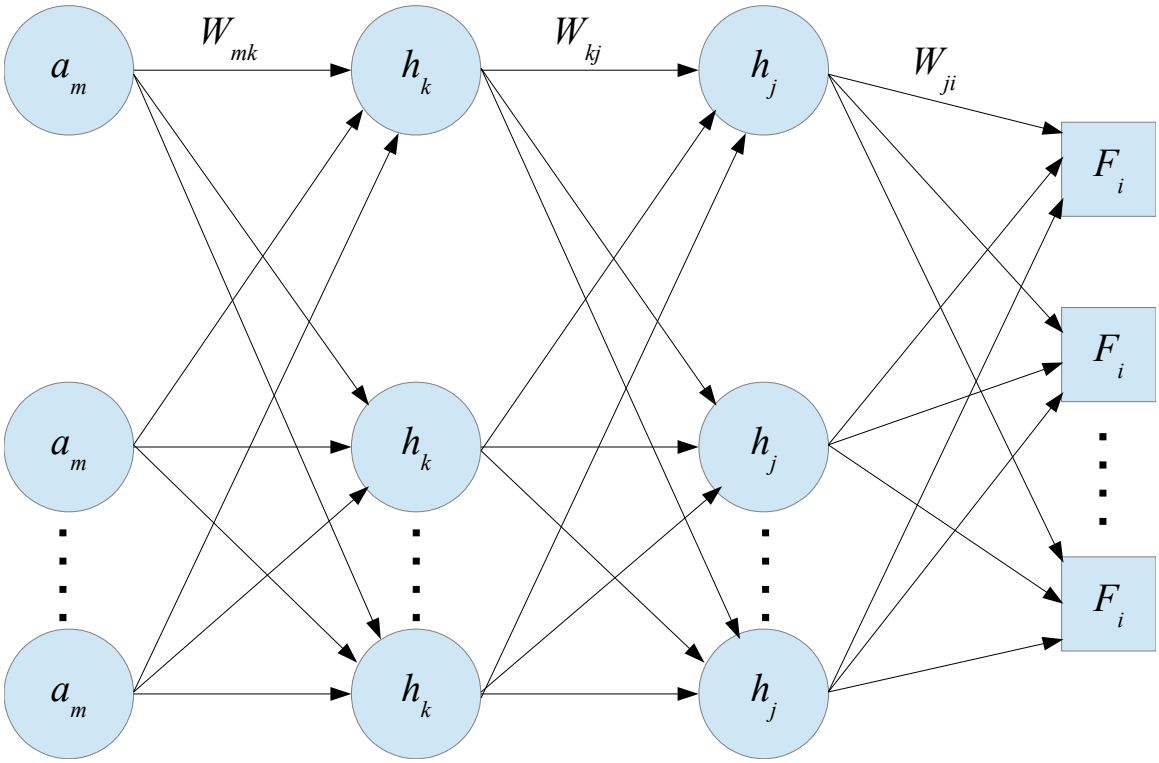
Consider a simply connected network that has an *input activation layer* (a_m), two *hidden activation layers* (h_k, h_j) and an *output activation layer* (F_i) as shown below. This configuration thus has three *connectivity layers*. Since there are **three connectivity layers**, this configuration is referred to as a **three-layer network**.



In general, you can see that for some set of activations on the right-side there are a collection of weights that connect them to another set of activations on the left-side. For each set of "output" (*aka*, right-side) activations we therefore have

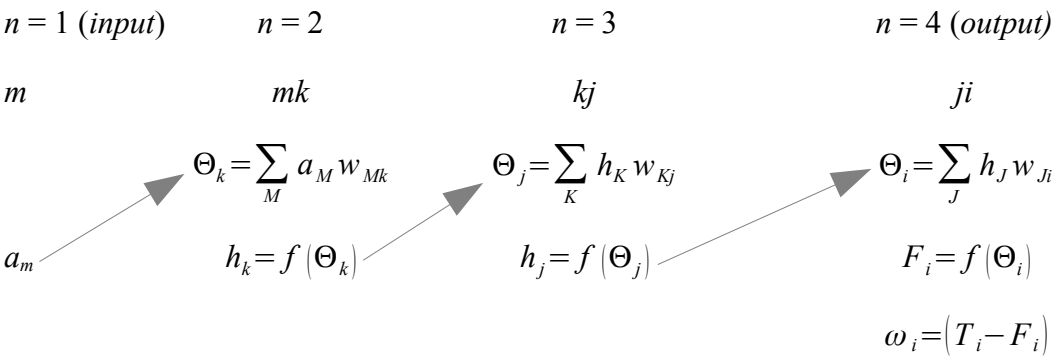
$h_k = f\left(\sum_m a_m w_{mk}\right)$, $h_j = f\left(\sum_k h_k w_{kj}\right)$ and $F_i = f\left(\sum_j h_j w_{ji}\right)$ where $f(x)$ is the threshold function (typically a sigmoid or hyperbolic tangent). It should be clear that we can generalize the connectivity between any two activation layers since each consists of an input (left-hand side) and an output (right-hand side) connected by a collection of weights. In general for any set of inbound and outbound activations we can write

$$Output_i = f\left(\sum_j Input_j Weight_{ji}\right).$$

Where the index order on the weights represent the input index (j) and the output index (i), respectively. Notice that there is really nothing special about the input and output activation layers in terms of the math involved in the connectivity layers. In principle, if the correct values for the weights can be found, then for any input there should be a predictable output, which I call T_i (for the truth).

Let's introduce some consolidating notion where $\Theta_i = \sum_j Activation_j Weight_{ji}$, so we then have $Output_i = f(\Theta_i)$.

A diagram for three connectivity layers (a three-layer network) is shown below to illustrate how the data flow though the network *via* the summations of the activations in each of the activation layers. Throughout all the documentation the *i*-index will be for the output activation layer, the *j*-index for the right-most hidden layer and the *k*-index for the activation layer to the left of the right-most hidden layer (which is the input layer in a two-layer network).



You should notice that you can fully specify the network connectivity model with only a few parameters

- Number of Input Nodes
- Number of Hidden Layers
- Number of Nodes in Each Hidden Layer
- Number of Output Nodes.

In Java the specification for the network would therefore need at most three parameters:

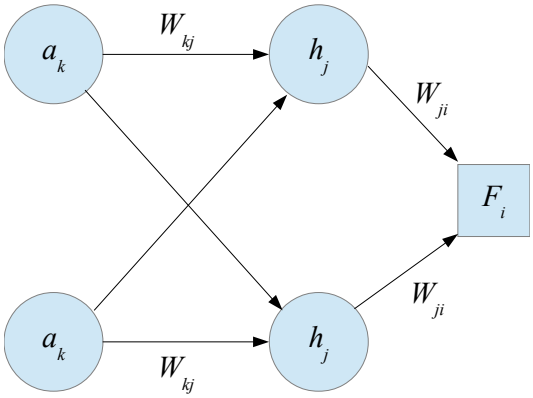
```
int inputNodes;
int[] hiddenLayerNodes;
int outputNodes;
```

From these parameters you can determine the dimensions of the array of weights for a fully connected feed-forward network (*i.e.*, a multilayer perceptron). You could also, of course, have a single array that incorporates both the input and output nodes, but at the moment we are going to strive for readability (make the code look like the design) so you will have an input activations array a[], a hidden activations array h[][] and and output activations array F[i].

To construct the network you should start by allocating the space needed to hold the weights based on the configuration parameters. PDP models are memory intensive, **so you need to declare and dimension all the arrays you will be using up front and not define them on-the-fly** which is what you usually do in Java. **You will need to use arrays (jagged arrays are fine) and NOT the ArrayList data structure**. ArrayList

carries a huge amount of unneeded overhead. **You should also use the known number of activations in each layer that are provided by the user to configure the network and NOT use the Java `array.length` construct to ask the object for those values**. Since the arrays are only being created once, you will need to tactically zero the elements associated with the summations. These elements are referred to as accumulators and should be zeroed just prior to entering the summation loop. You will need to be able to randomize the weights for training purposes, so **create a method just like you did in the first week of AP CS A for generating random values within any user selectable range** (not just integers and don't assume you will always start at zero). Note that this action is NOT part of the construction of the network since once the optimal weights are known you would set them to those values. You will need to be able to perform the general activation-weight multiplication of the form $Output_i = f\left(\sum_j Activation_j Weight_{ji}\right)$ and have a high level routine that manages what constitutes inputs and outputs for this action. You will need the ability to dump the weights to the terminal (for debugging purposes) as well as a way of stating the structure of the network configuration (again for debugging). Lastly you will need a way of minimizing the error function based on the training set data, but that is the topic of the "Minimizing the Error Function" document.

The problem is now in figuring out what the value of all the weights is to be to make the network do something useful. To find a set of acceptable weights (and there can be more than one set that will work well enough for a given set of inputs), we are going to have to train our network. In order to construct a functioning example, let us build a simple network that we can use to solve boolean algebra problems such as AND, OR and XOR. This simple network has a two-node input activation layer, a two-node hidden activation layer and a single-node output layer and is designated as a 2-2-1 network.



We have the following relationships within the network

$$h_j = f\left(\sum_k a_k w_{kj}\right),$$

$$F_i = f\left(\sum_j h_j w_{ji}\right),$$

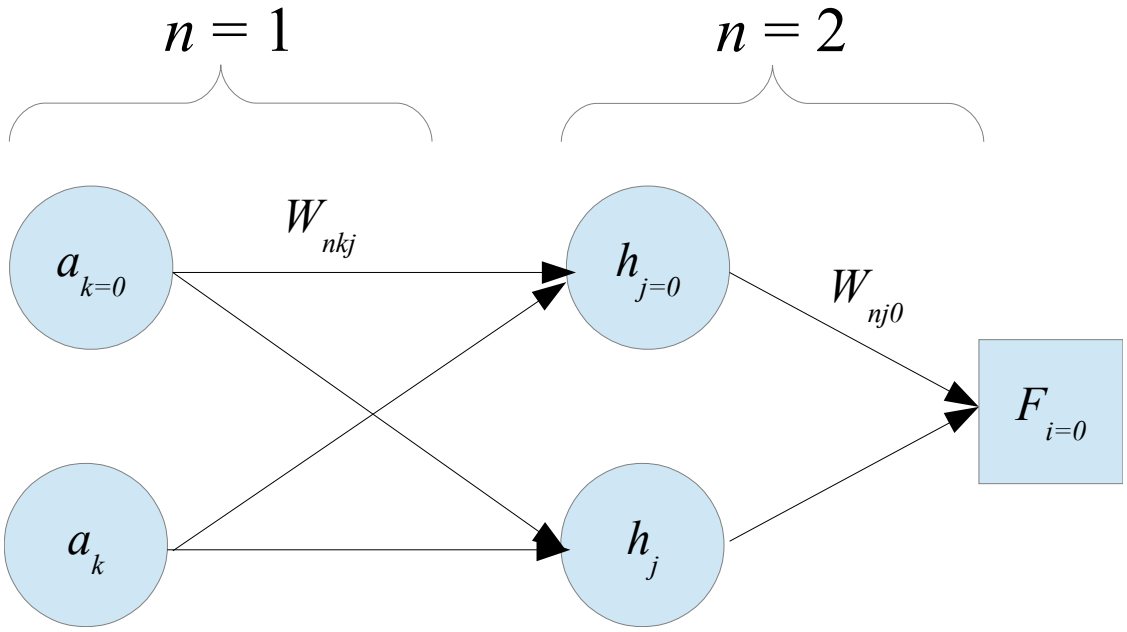
${}^tE = \frac{1}{2}\sum_i \left({}^tT_i - {}^tF_i\right)^2$ (more on how to interpret this expression later)

or using our theta notion where $\Theta_i = \sum_j a_j w_{ji}$, $\Theta_j = \sum_k a_k w_{kj}$, and letting ${}^t\omega_i = \left({}^tT_i - {}^tF_i\right)$ we have

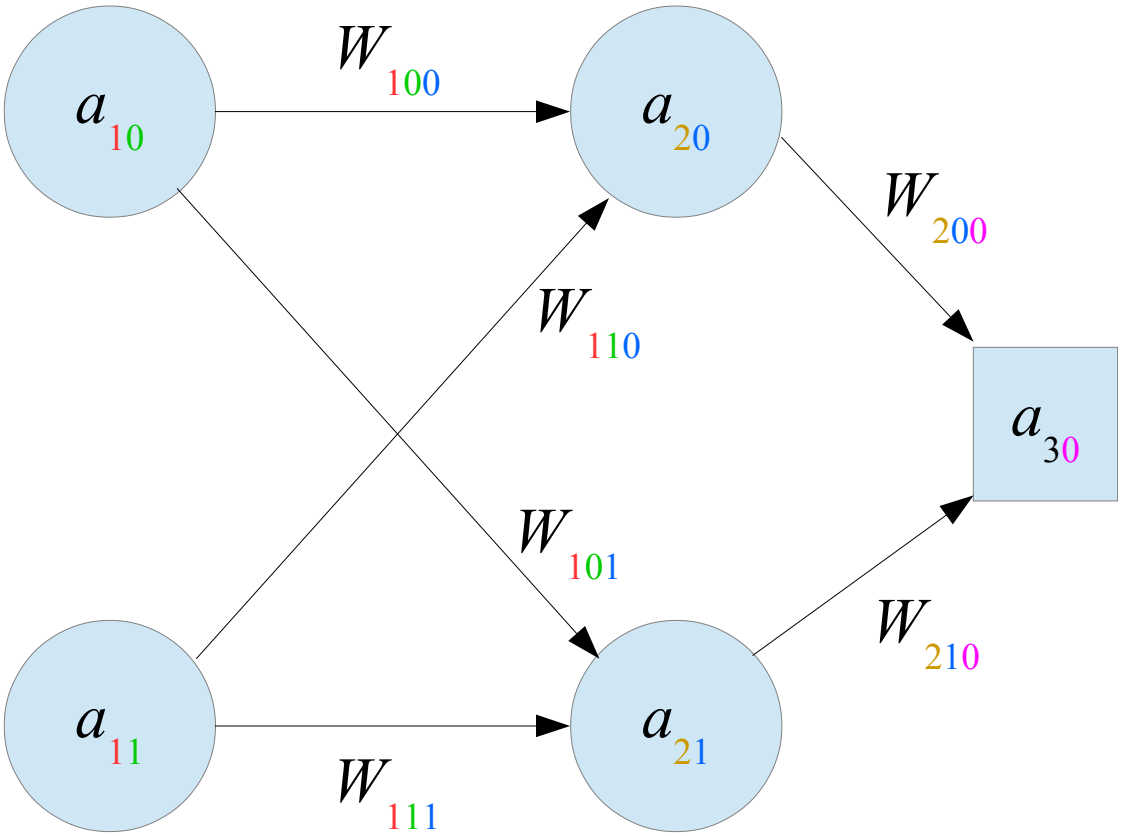
$h_j = f\left(\Theta_j\right)$, $F_i = f\left(\Theta_i\right)$, and ${}^tE = \frac{1}{2}\sum_i \left({}^t\omega_i\right)^2$.

where ${}^tE$ is the *error function* (*aka* cost function) for a single training activation case *t* which is a function of all the weights in the network and the T_i values, which are the target outputs expected from a known set of input values (these are our training data). There is a one-to-one correspondence to the outputs produced by the network and the expected values which is known as *supervised* training.

Let us fully expand out our diagram with all the proper index values. The problem is that we are using *k*, *j* and *i* to help keep track of the activation layer and when we put in the numbers we won't know where we are (*e.g.*, is '1' for the *i*, *k* or *j* layer?). We need another index to keep track of the activation and connectivity layers, so we are going to redraw our diagram to use a more general notation where all activations are simply labeled as a_{ni} with an activation layer index *n* and a node index *i*, *j*, or *k*, and all the weights are W_{nkj} and W_{nji}, where *n* is now the connectivity layer, *k* is an input node index, *j* is a hidden node index and *i* is an output node index.



Now we can put in the index values for this simple example where I have used color coding to make the associations easier to see.



For each of the right-side activations we can now expand our example.

$$h_j = a_{2j} = f\left(\sum_{k=0}^{1} a_{1k} w_{1kj}\right), \text{ so}$$

$$h_0 = a_{20} = f\left(\sum_{k=0}^{1} a_{1k} w_{1k0}\right) = f\left(a_{10} w_{100} + a_{11} w_{110}\right),$$

$$h_1 = a_{21} = f\left(\sum_{k=0}^{1} a_{1k} w_{1k1}\right) = f\left(a_{10} w_{101} + a_{11} w_{111}\right)$$

and

$$F_i = a_{3i} = f\left(\sum_{j=0}^{1} a_{2j} w_{2ji}\right), \text{ so}$$

$$F_0 = a_{30} = f\left(\sum_{j=0}^{1} a_{2j} w_{2j0}\right) = f\left(a_{20} w_{200} + a_{21} w_{210}\right)$$

But what about the error function? There are two ways to view the error function. The first is to consider the full error for all the training data. This minimization is mathematically rigorous in that you want to minimize the error with respect to the *N*-dimensional weight space for all training cases simultaneously. It is also computationally

expensive. A simpler approach is to cycle through the training data and adjust the weights after each training instance (a case) in the training set so the changes to the weights are made incrementally, which is how we are going to proceed. For the XOR problem, the input and output activation values are clearly defined.

Input Activations		XOR Output Activations ${}^{t}T_0$
0	0	${}^{1}T_0 = 0$
0	1	${}^{2}T_0 = 1$
1	0	${}^{3}T_0 = 1$
1	1	${}^{4}T_0 = 0$

These values represent our full training set with each row representing a single training case. Note that you must have a network constructed before you can train it. Training is NOT part of the network; it is simply a way to determine the values of the weights that should be used by the network. For this complete training set our simple example becomes

$${}^{t}E = \frac{1}{2}\sum_{i=0}^{0}\left({}^{t}T_i - {}^{t}F_i\right)^2, \text{ so}$$

$${}^{1}E = \frac{1}{2}\left({}^{1}T_0 - {}^{1}F_0\right)^2,\ {}^{2}E = \frac{1}{2}\left({}^{2}T_0 - {}^{2}F_0\right)^2,\ {}^{3}E = \frac{1}{2}\left({}^{3}T_0 - {}^{3}F_0\right)^2,\ {}^{4}E = \frac{1}{2}\left({}^{4}T_0 - {}^{4}F_0\right)^2$$

or

$${}^{1}E = \frac{1}{2}\left(0.0 - {}^{1}F_0\right)^2,\ {}^{2}E = \frac{1}{2}\left(1.0 - {}^{2}F_0\right)^2,\ {}^{3}E = \frac{1}{2}\left(1.0 - {}^{3}F_0\right)^2,\ {}^{4}E = \frac{1}{2}\left(0.0 - {}^{4}F_0\right)^2$$

You now have a complete example of a simple XOR network that can be put into Excel and coded in your favorite language. The two approaches should agree at each step along the way to the output value and the error function for a known set of weights.