Neural Networks in Artificial Intelligence

Artificial Intelligence involves several different approaches in computer software, including

- **Symbolic artificial intelligence** includes methods based on high-level "symbolic" (human-readable) representations of problems, logic and search.
- **Bayesian decision networks** are models that represent a set of variables and their dependencies. Used, for example, to predict the likelihood that any one of several possible causes was the contributing factor to an event.
- **Evolutionary algorithms** use mechanisms inspired by biological evolution, such as reproduction, mutation, recombination, and selection. A fitness function determines the quality of candidate solutions. Evolution of the candidate solutions takes place after repeated application of the method.

Wikipedia

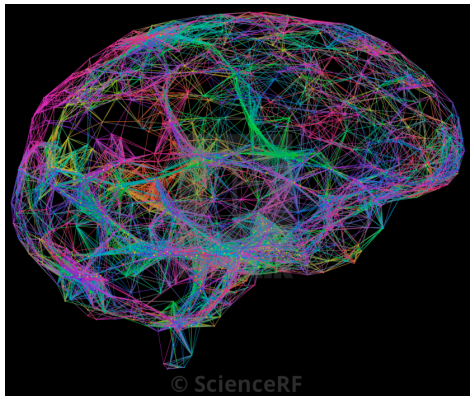
Here we illustrate a fourth approach

- **Artificial Neural Networks** - with basic introductory examples

MATLAB code used here is available at https://github.com/RichardHerz/neural-networks

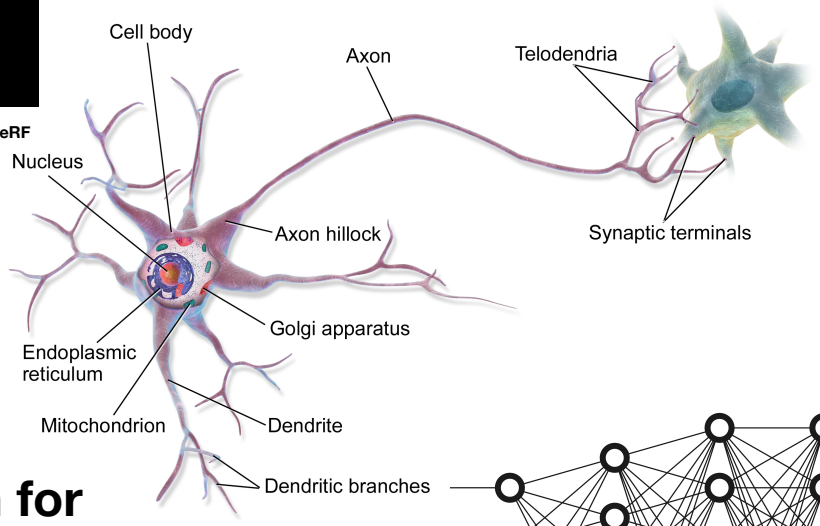
Our brains sense and think using connected networks of cells called neurons

www.braininjuryaustralia.org.au

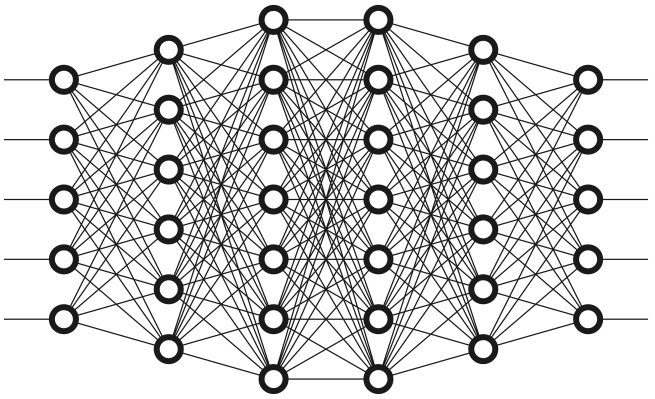


@ScienceRF

Blausen_0657_MultipolarNeuron



These networks are the inspiration for "artificial neural networks" which can be trained to solve complex problems such as object recognition in images and speech recognition



neural_network_shutterstock_all_is_magic.jpg

Neural Network

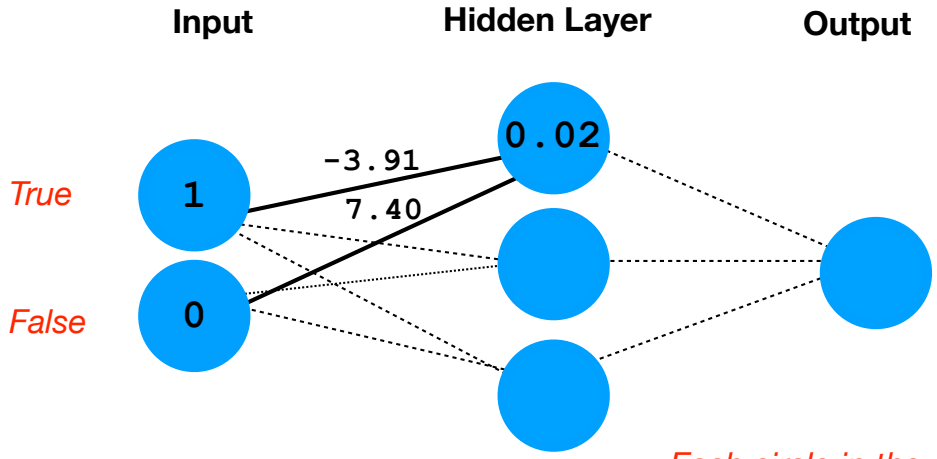
A simple example

Simulates XOR logic - exclusive or

Output is TRUE when one input is TRUE but not both

2 inputs, 1 output,
1 hidden layer
with 3 neurons &
9 synapses

This logic can be computed in a single IF statement in a procedural program but is useful here to start learning about neural networks



Each circle in the diagram represents a node or "neuron."
Each line represents a connection or "synapse."
The value in a neuron is its "activation."
Each synapse has a connection "weight."

EXAMPLE for input of 1
0

node value = sigmaFunc(sum of (connection weight * node activation))

where sigmaFunc(x) = exp(x) / (1 + exp(x)) >> converts all input x values into range 0 to 1

INPUT > HIDDEN LAYER

sigmaFunc((-3.91 * 1) + (7.40 * 0)) = 0.02 = hidden node 1 activation

Every node - neuron - has a connection - synapse - to every neuron in nearest-neighbor layers of neurons in this basic type of neural network.

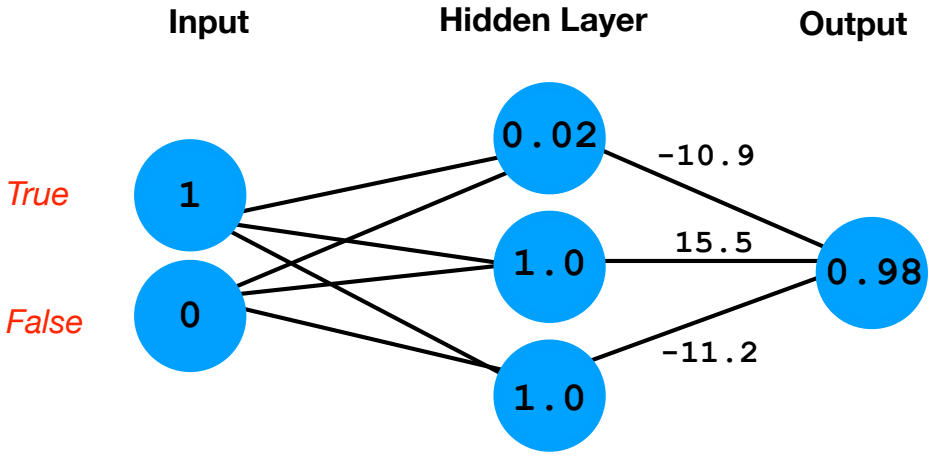
The values are held in memory locations and the CPU executes the math - there are no physical, hardware neurons and synapses.

Neural Network

Simulates XOR logic - exclusive or

Output is TRUE when one input is TRUE but not both

2 inputs, 1 output,
1 hidden layer
with 3 neurons &
9 synapses



True,
where network outputs
approximate desired values

HIDDEN LAYER > OUTPUT

sigmaFunc((-10.9 * 0.020) + (15.5 * 1.0) + (-11.2 * 1.0)) = 0.98 = output node

The MATLAB code to solve for the output remains the same as that below, regardless of the size of the network:

```
for i = 2 : numHiddenLayers + 2
    a{i} = sigmaFunc( W{i-1} * a{i-1} );
end
```

W *is a MATLAB cell array whose elements are the matrices of synapse weights for each layer;* ***a*** *is a cell array whose elements are the vectors of neuron activation values. Each set of* ***W*** *and* ***a*** *are matrix-multiplied to obtain the neuron activation values for the next layer in the series of neuron layers.*

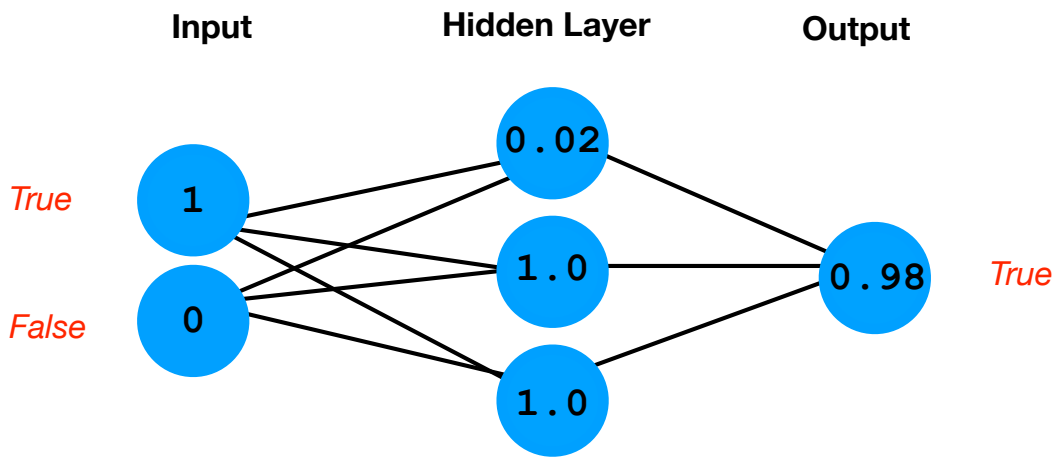
Matrix multiplication is well suited to being accelerated in hardware Graphical Processing Units, since graphic transformations also involve matrix multiplication.

Neural Network

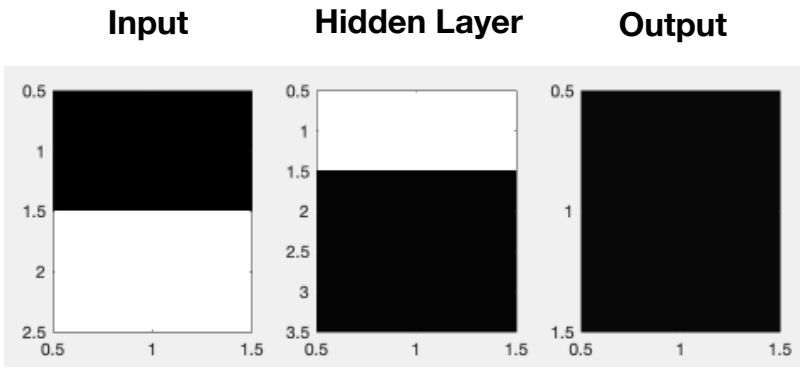
Simulates XOR logic - exclusive or

Output is TRUE when one input is TRUE but not both

2 inputs, 1 output,
1 hidden layer
with 3 neurons &
9 synapses



Visualization of neuron values - “activations” - for this input



1
0

0.02
1.00
1.00

0.98

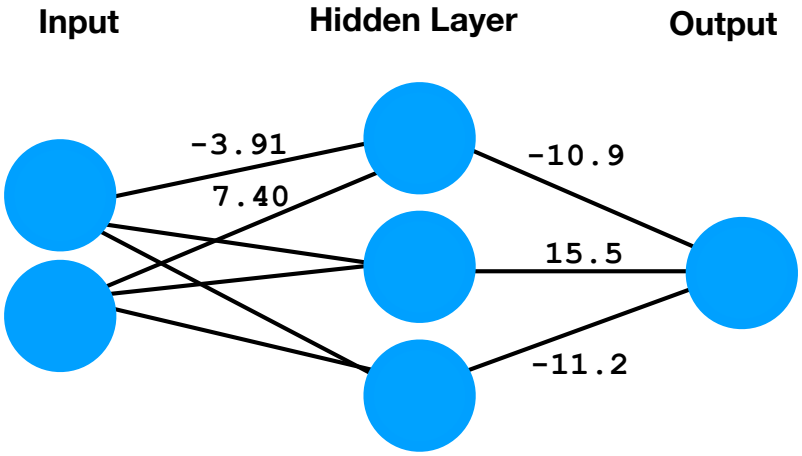
True

Neural Network

Simulates XOR logic - exclusive or

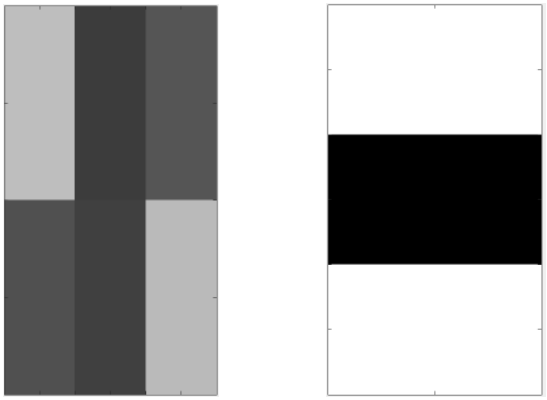
Output is TRUE when one input is TRUE but not both

2 inputs, 1 output,
1 hidden layer
with 3 neurons &
9 synapses



Visualization of synapse connection "weights" to hidden layer and to output
min = -11.2 (white), max = +15.5 (black)

-3.9117 9.3619 6.7424
7.4018 9.1350 -3.5200

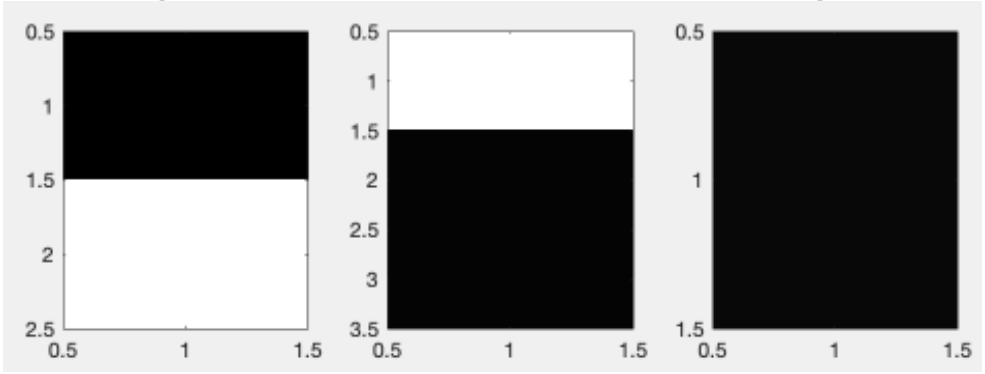


-10.9275
15.4564
-11.2180

The synapse connection weights were determined when the network was "trained"
using combinations of known inputs and outputs.
Training is discussed later in these notes.

Visualizations of node activations: input > hidden layer > output

Input | **Hidden Layer** | **Output**

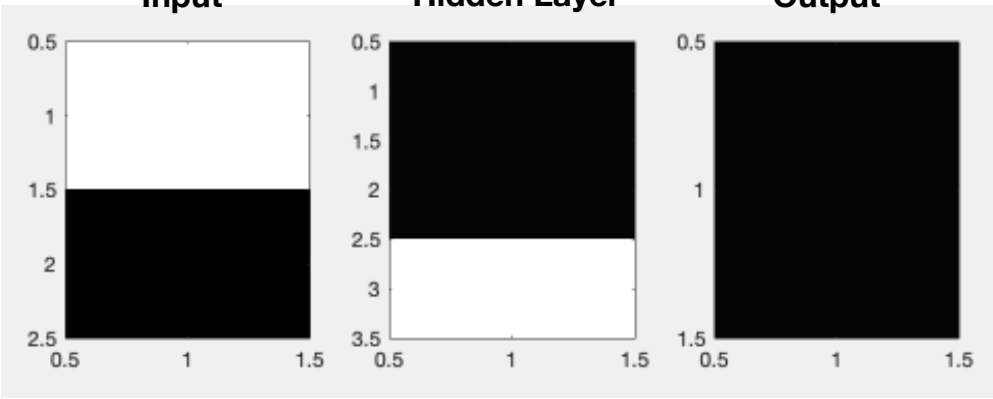


1
0

0.02
1.00
1.00

0.98
True

Input | **Hidden Layer** | **Output**

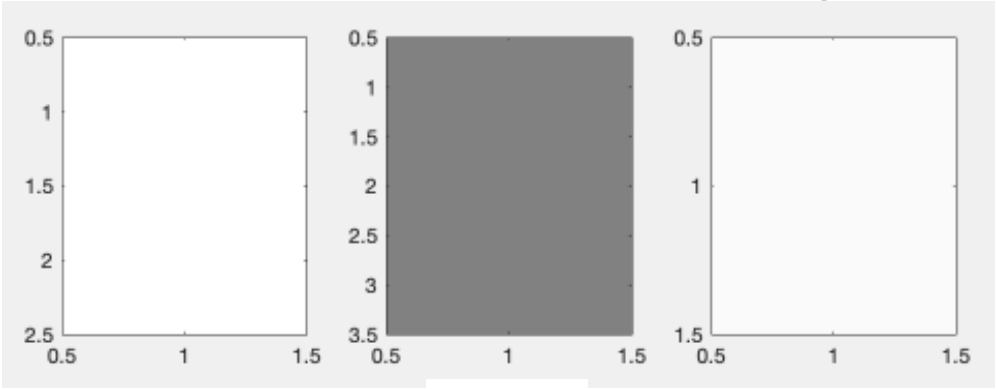


0
1

1.00
1.00
0.029

0.99
True

Input | **Hidden Layer** | **Output**

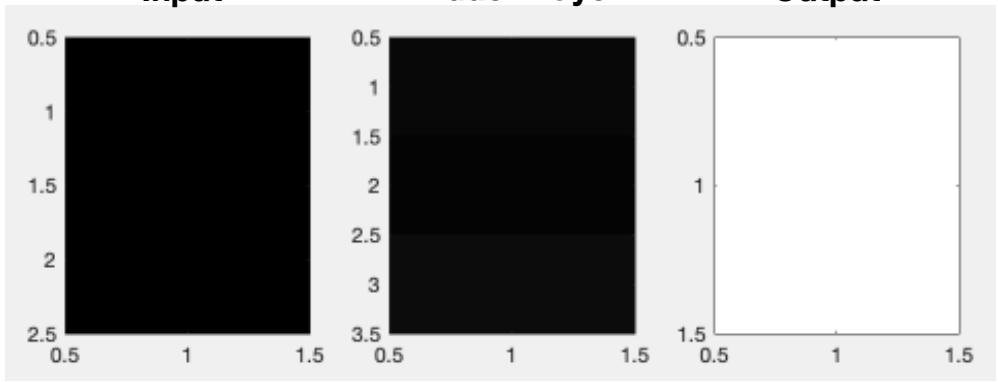


0
0

0.50
0.50
0.50

0.034
False

Input | **Hidden Layer** | **Output**



1
1

0.97
1.00
0.96

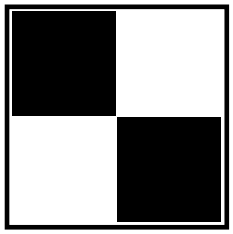
0.003
False

Output is TRUE when one input is TRUE but not both

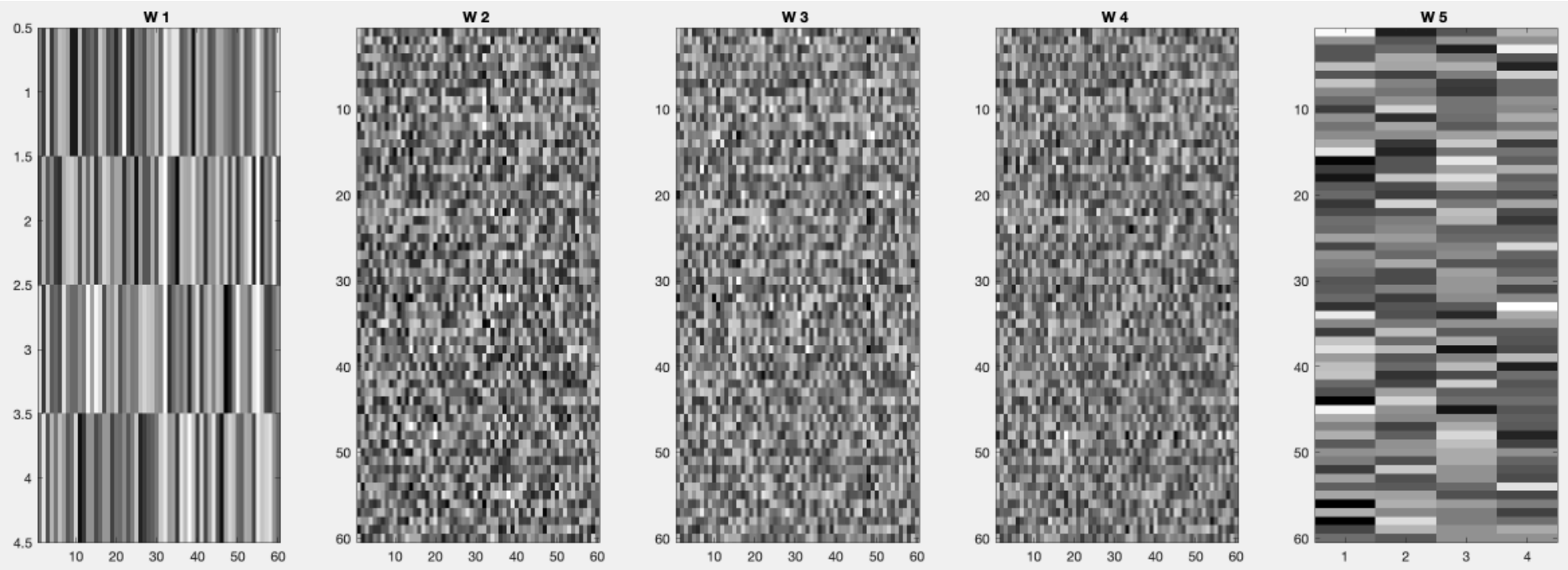
Neural Network

4 inputs, 4 outputs
4 hidden layers, each with 60 neurons = 240 neurons & 11,280 synapses

A more complex network which detects diagonal, horizontal and vertical inputs to a 2 x 2 "touch screen"

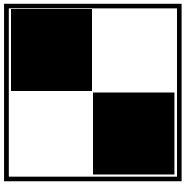


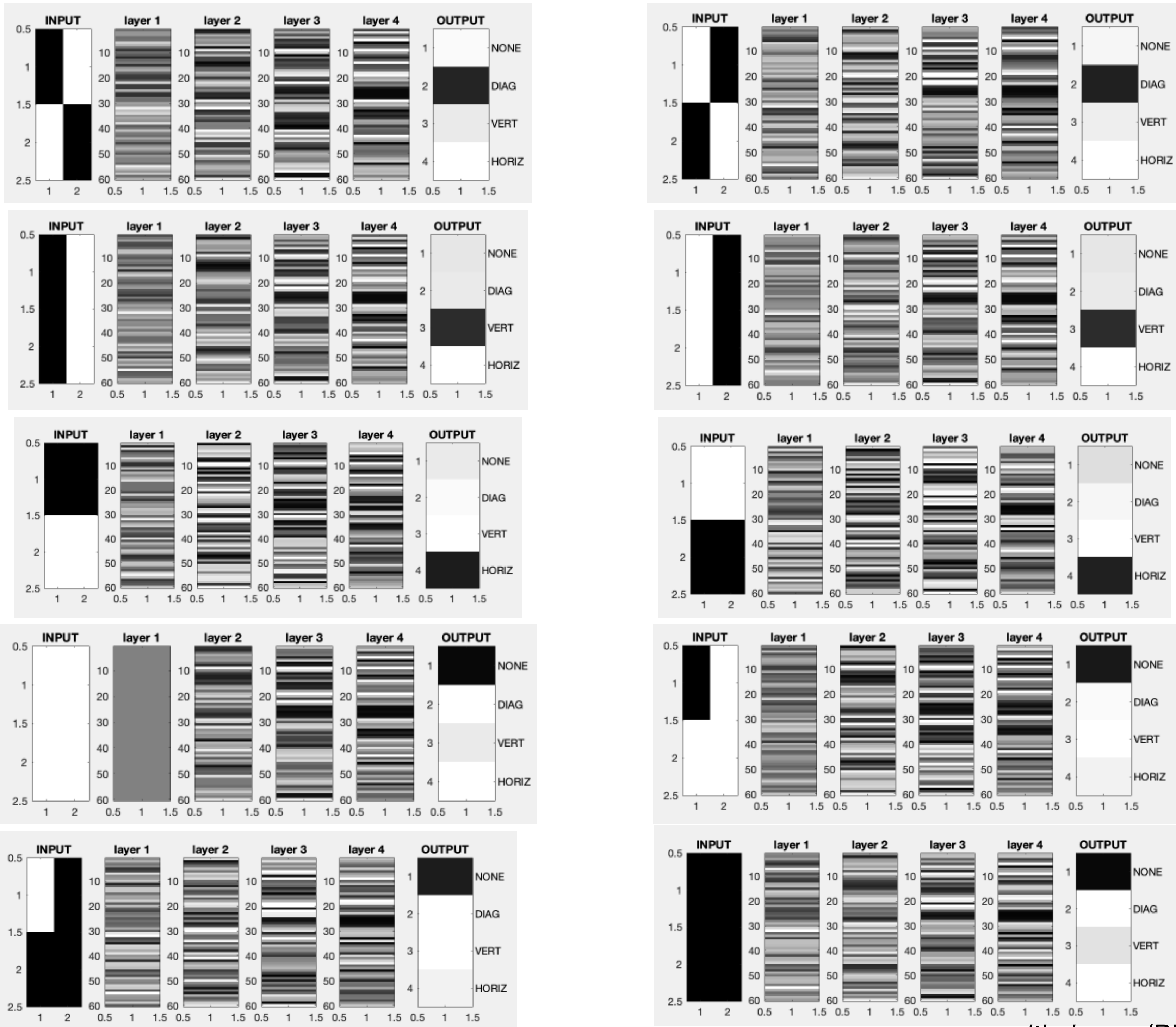
Visualization of synapse weights to hidden layers 1-4 and to output,
min = -1.23 (white), max = +1.25 (black)



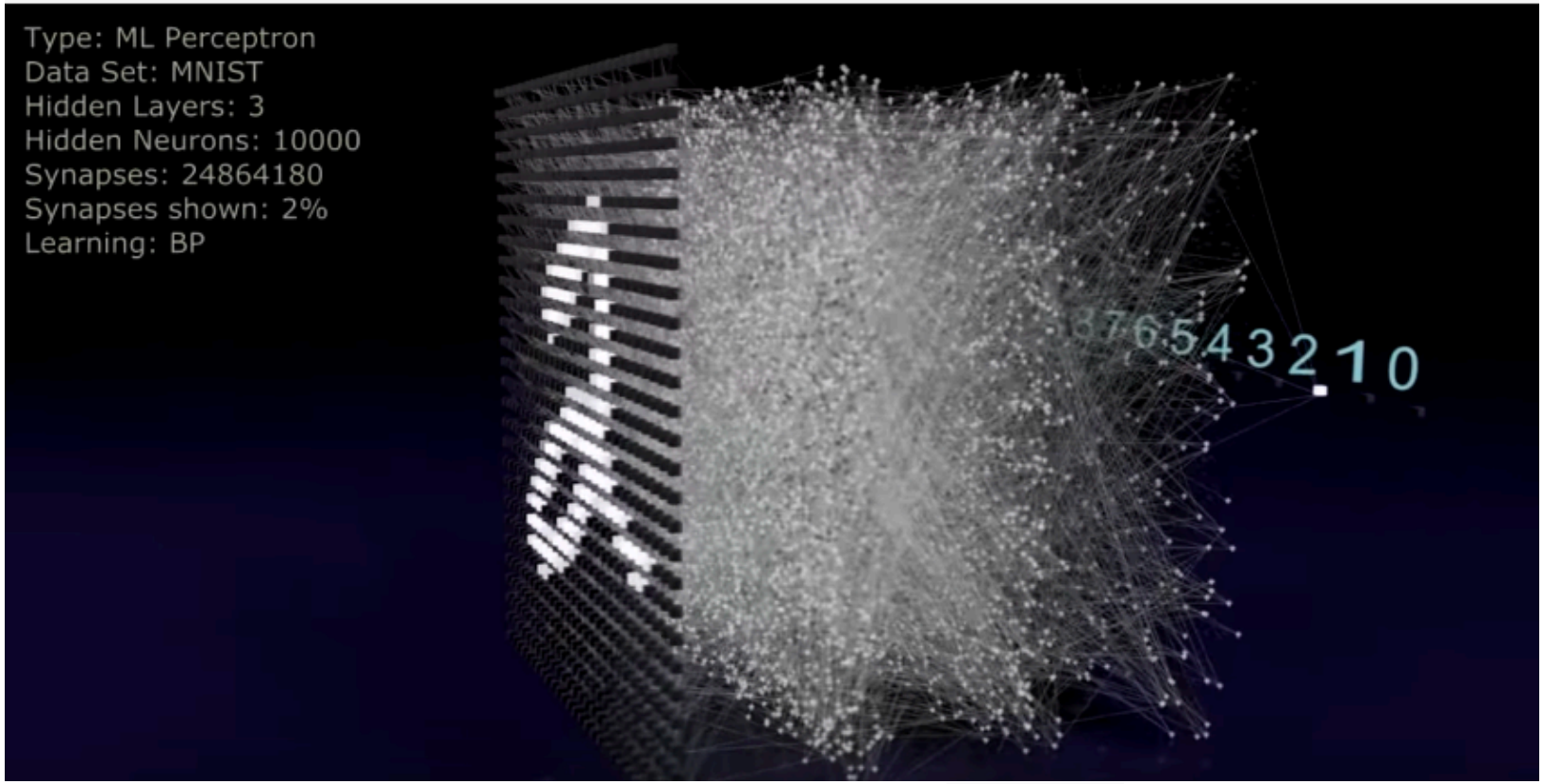
Prior to training the network with known input and output cases, the connection weights were assigned random values in the range -1 to +1. Then the weights were adjusted during training in order to match input cases with their corresponding outputs. The resulting weights are not random. Different sets of weights will be obtained with different random initializations

Visualization of node activations: input > 4 hidden layers > output





A neural network for a 28 x 28 “touch screen”
note that only 2% of the 24+ million “synapses” are shown



Denis Dmitriev **https://youtu.be/3JQ3hYko51Y**

This network was trained with a set of known inputs (drawn numbers) and outputs that is much smaller than all the possible ways in which numbers can be drawn. The power of a trained network is to be able to give correct outputs for inputs for which the network wasn’t trained. That’s the whole point of neural networks!

A neural network represents a large number of coupled equations which, when given a set of input values, can produce a set of desired output values.

The more neurons and synapses - the more equations - and the greater complexity of inputs and outputs which can be "fit" by the system of equations. Note the significant increase in complexity going from the XOR example to the 2 x 2 “touch screen” example to the 28 x 28 touch screen in the figure above.

A neural network might be thought of as a general function which can fit anything given enough terms…

In a sense, neural networks are math functions which can "fit" any desired input and output data given enough adjustable parameters, which are the "synapse" connection weights and, thus, enough neurons.

Using a neural network is somewhat similar to using a polynomial function to fit a series of data points (empirical fit) vs. using a functional form that represents the underlying physics (theoretical fit).

In a neural network, the functional form is fixed by the network structure. The values of the constants in the function are the connection weights, whose values are determined during training.

For the XOR network above, this is the Matlab code which computes the output a{3} given the input a{1}

```
for i = 2:3
    a{i} = sigmaFunc( W{i-1}*a{i-1} );
end
```

Matrix W{i-1} and vector a{i-1} are elements of the cell arrays W and a. They are matrix multiplied. The Matlab code is very compact. We can see the form of this network's function by looking at the expanded equation, which shows the individual terms. The output $a\{3\}$ is a function of the inputs $a\{1\}$:

$$a^{\{3\}} = f(a^{\{1\}}) = \sigma\left(W_1^{\{2\}}a_1^{\{2\}} + W_2^{\{2\}}a_2^{\{2\}} + W_3^{\{2\}}a_3^{\{2\}}\right)$$

$$= \sigma\left(W_1^{\{2\}}\sigma\left(W_{1,1}^{\{1\}}a_1^{\{1\}} + W_{1,2}^{\{1\}}a_2^{\{1\}}\right) + W_2^{\{2\}}\sigma\left(W_{2,1}^{\{1\}}a_1^{\{1\}} + W_{2,2}^{\{1\}}a_2^{\{1\}}\right) + W_3^{\{2\}}\sigma\left(W_{3,1}^{\{1\}}a_1^{\{1\}} + W_{3,2}^{\{1\}}a_2^{\{1\}}\right)\right)$$

where, for more compact notation, the superscript {n} of cell arrays a and W denotes a matrix in cell array element n, and the subscripts are the indices within that matrix. The hidden layer activations are $a\{2\}$. The nonlinear activation function for this network, which constrains activation values between 0 and 1, is

$$\sigma(x) = \frac{e^x}{1 + e^x}$$

For a larger neural network of this type, there are more terms but the functional form remains unchanged. With the continued development of computers, larger networks can be computed more rapidly.

Training Artificial Neural Networks

The networks shown earlier in these notes had already been "trained" to produce correct outputs given various inputs. The connection weights are fixed in the initial training stage. This is the hard part of neural networks!

First, a collection of paired inputs and their correct outputs is obtained to use in training the network.

Initial weights are set to random values. Then an input is fed to the network and an output is obtained. This output is compared to the correct output and an error value is computed. The initial error value will be large.

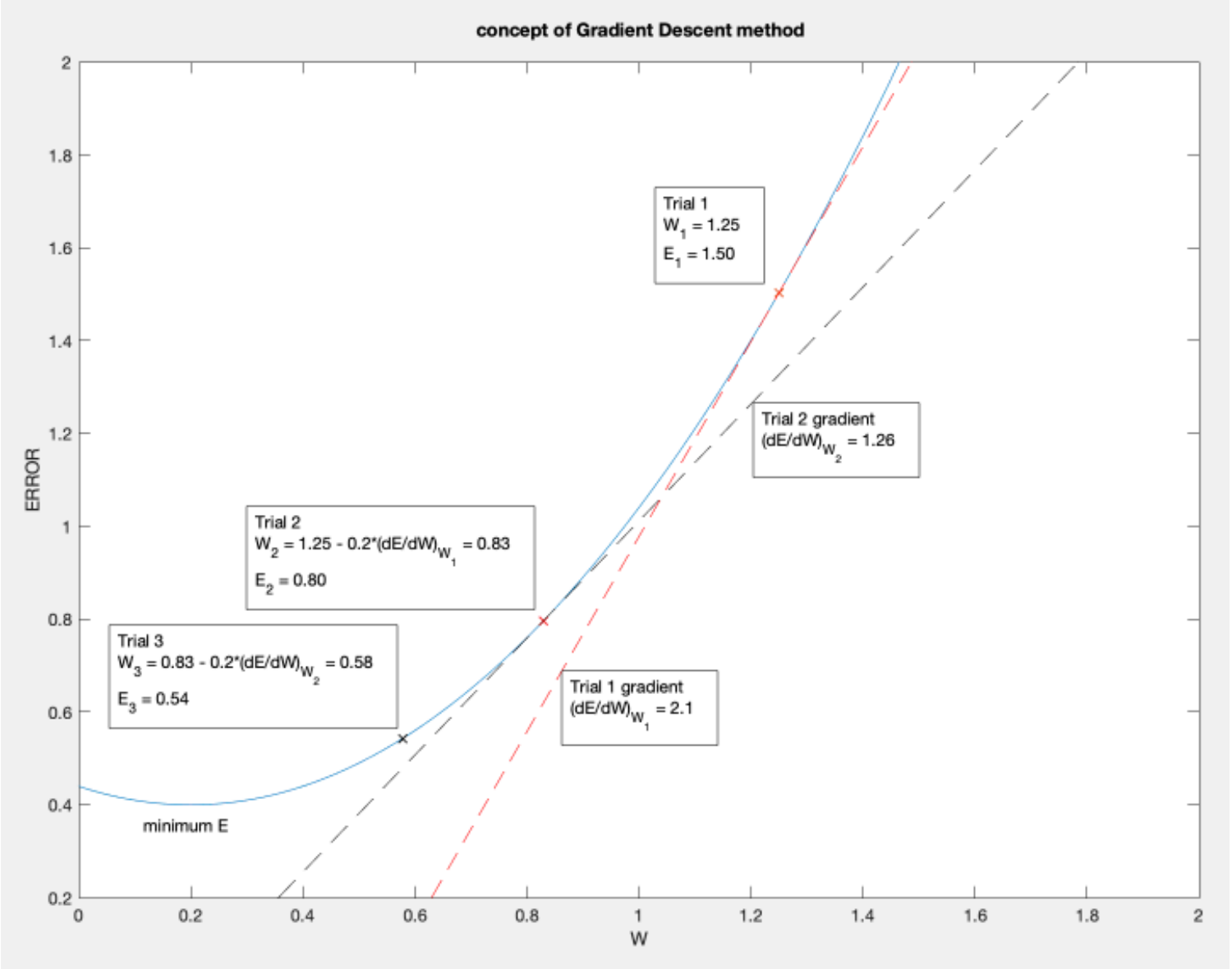
Next, for each connection weight in the network, the rate of change of the error with respect to a change in the value of that weight is computed. After the rates of change for all the weights are obtained, the weights are changed by multiplying a small constant times each of the rates of change, and then subtracting the result from corresponding weights. This small adjustment to the weights, when the input is again fed to the network, will result in a smaller error in the output.

This procedure is called the “gradient descent method.” The rates of change are also called gradients. You wish to "descend" to smaller errors.

This gradient descent procedure is repeated until the error reaches a minimum value. The concept is illustrated in the figure on the next slide.

Gradient Descent Method

The blue line represents how the output error E changes with this connection weight W. In Trial 1, the value of W is randomly specified. Then E and the gradient (dE/dW) is computed. The value of W in Trial 2 is computed from the value in Trial 1 minus a factor (0.2 here) times the gradient from Trial 1. This results in a smaller error. The process repeats until E approaches the minimum value.



Training Artificial Neural Networks

After training of a network that is properly structured for the problem, the network will give approximately correct results, even for inputs that are not in the training set.

In fact, that is the whole point of neural networks: giving correct outputs for inputs that are not in the training set!

Note that the simple XOR and 2x2 touch screen networks had all possible inputs used in training, whereas the 28x28 touch screen was trained with a finite set of the extraordinarily large possible combinations of pixel activations that are possible.

The 28x28 touch screen network shown in a previous slide was trained with the MNIST data set http://yann.lecun.com/exdb/mnist/. Other sets of data are available on the web for other types of problems such as image and speech analysis.

The gradient descent method is applied to all pairs of inputs and outputs in the training set, either individually or in batches.

Next, we will examine how the gradient values in the gradient descent method are obtained.

Training Artificial Neural Networks

In the simple network structure we are considering, information signals move in one direction: from the input to the output. A change in one connection weight near the input causes a change in the signal that propagates through the rest of the network and eventually results in a change in the final output error.

The way that this change in signal propagates through the network is determined by the connection weights and nodes through which it passes. Those weights and nodes are known to us.

One way to determine the gradients would be to make a change in each connection weight separately, then recompute the outputs and obtain the change in error. Then repeat for each connection weight in the network. This would be a lot of computational work!

A more efficient way to compute the same gradient values is to work backwards from the output to each preceding connection weight in a procedure called "back propagation." This is possible because of the straightforward structure of the network.

This method proceeds from the output layer to the last hidden layer, then back to each preceding pair of hidden layers, computing the gradients of error with change in connection weight at each step by simple analytical differentiation. This process is more efficient because the entire network from input to output doesn't have to be computed for each connection weight.

After all the gradients are computed, then all the weights are updated. Then the input is fed to the modified network and a new output and error is computed.

The gradient descent process is repeated and continues until only small changes in output error are obtained.

Back Propagation and Gradient Descent in Network Training

Below, instead of standard math notation, we use MATLAB pseudo-code in order to make easier comparison of this derivation to the MATLAB code, which is available at https://github.com/RichardHerz/neural-networks.

A derivative similar to `dX_dY` below are the *partial* derivatives of array `X` with respect to array `Y`. We do not use `∂X/∂Y` since that shows division in code and we can't use ∂ in MATLAB code.

Relationships in the XOR network:

```
layer            input       hidden          output

activation       a{1}          a{2}           a{3}
weight                  W{1}          W{2}
```

The errors at the output nodes, given a training input `a{1}`, are

```
E = 0.5 * (y - a{3}).^2
```

where `y` is the array of correct outputs in the training set, which correspond to the inputs `a{1}`.

The original paper introducing back propagation: Rumelhart, D., Hinton, G. & Williams, R. Learning representations by back-propagating errors. Nature 323, 533–536 (1986). https://www.nature.com/articles/323533a0

Back Propagation and Gradient Descent in Network Training

The rates of change of errors `E` with respect to the outputs are

```
dE_da{3} = -(y - a{3})
```

Define the inputs `I` from layer 2 going to the output layer 3 as

```
I{2} = W{2} * a{2}
```

where

```
a{3} = sigmaFunc(I{2})

da{3}_dI{2} = dsigmaFunc(I{2})_dI{2}

            = a{3} .* (1 - a{3})
```

The rates of change of error `E` with respect to `I{2}`, the signals from layer 2 to layer 3, are

```
dE_dI{2} = dE_da{3} * da{3}_dI{2}

         = -(y - a{3}) .* a{3} .* (1 - a{3})
```

Back Propagation and Gradient Descent in Network Training

Working backward toward the inputs, in "back propagation" of the error...

```
I{1} = W{1} * a{1}

a{2} = sigmaFunc(I{1})

da{2}_dI{1} = dsigmaFunc(I{1})_dI{1}

            = a{2} .* (1 - a{2})

dE_da{2} = dI{2}_da{2}' * dE_dI{2}

dI{2}_da{2} = W{2}

dE_da{2} = W{2}' * dE_dI{2}
```

where `W{2}'` is the transpose of `W{2}` and `*` is matrix multiplication in Matlab such that the results are the sums of the contributions of each node in layer 2.

Back Propagation and Gradient Descent in Network Training

The rates of change of error `E` with respect to `I{1}`, the signals from layer 1 to layer 2, are

```
dE_dI{1} = dE_da{2} * da{2}_I{1}

dE_dI{1} = W{2}' * dE_dI{2} .* a{2} .* (1 - a{2})
```

For the XOR network with only one hidden layer, we stop here in getting the `d`'s.

For a network with more hidden layers, we would continue the process by which we got `d{1}`, with corresponding changes in array index.

Back Propagation and Gradient Descent in Network Training

Now that we have the rates of change of error `E` with respect to the inputs to each layer, we can compute the rates of change of error `E` with respect to the weights themselves.

```
dE_dW{i} = dE_dI{i} * dI{i}_dW{i}'

dI{i}_dW{i} = a{i}

dE_dW{i} = dE_dI{i} * a{i}'
```

where `a{i}'` is the transpose of `a{i}` and `*` is matrix multiplication in Matlab such that the results are the sums of the contributions to each weight.

Back Propagation and Gradient Descent in Network Training

Now we can compute the new values of the connection weights `W{i}`, which will be used in the next iteration of the gradient descent method

```
W{i} = W{i} - alpha * dE_dW{i}
```

where `alpha` is a constant value in the range 0-1.

With these new values of weights, the training inputs are used to compute new outputs. The new errors are computed, and the process is repeated.

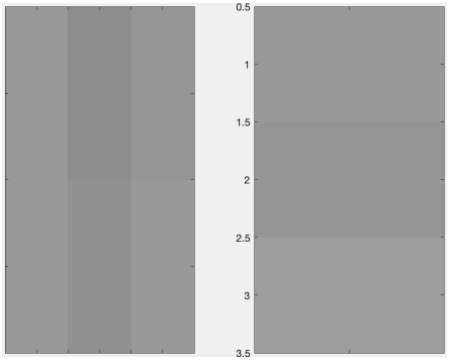
The process is stopped when some specified criterion is met with regard to the approach to the minimum error or a maximum number of iterations.

At that point, the artificial neural network has been trained. The performance of the network should then be tested by using additional inputs for which the outputs are known.

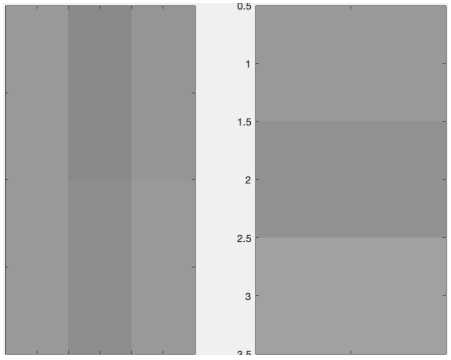
On the next slide is a visualization of how the XOR network weights change during training.

XOR network weight changes during training

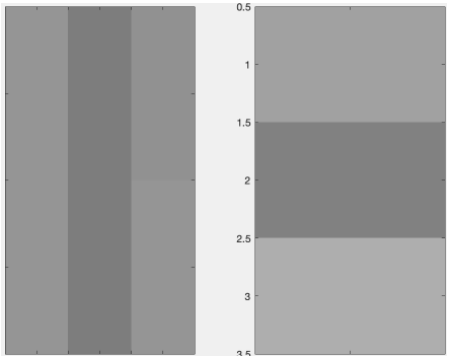
Visualization of the weights during training, starting from and ending at the weights shown above. These are for a particular random initialization. Other initializations will result in different weight patterns, but similar final error, E, which is shown for only the [1;0] input, although the training used all four possible inputs. Some elements of the initial pattern persist during training.



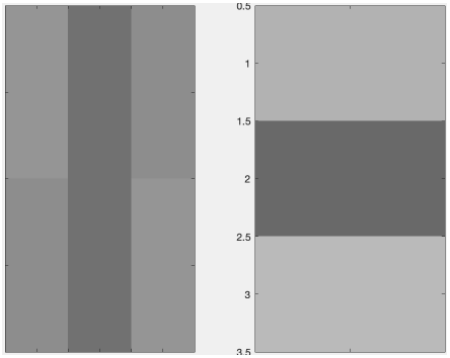
initial, E = 0.1672



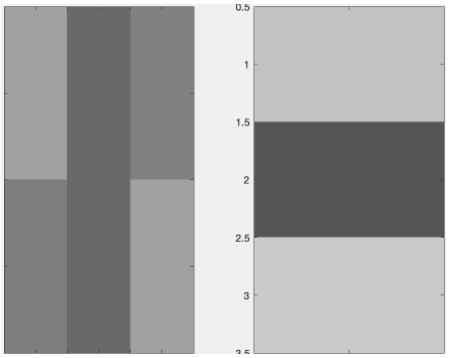
10%, E = 0.1167



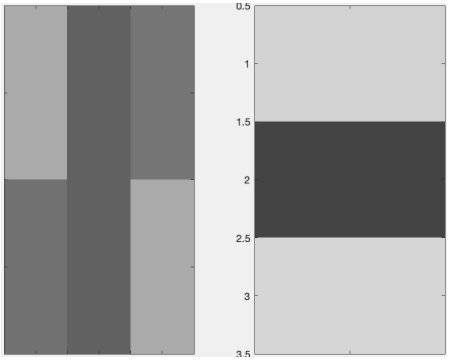
20%, E = 0.0988



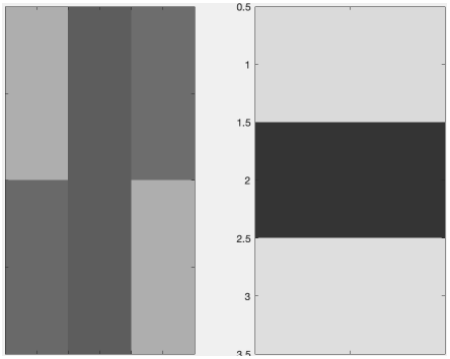
30%, E = 0.0690



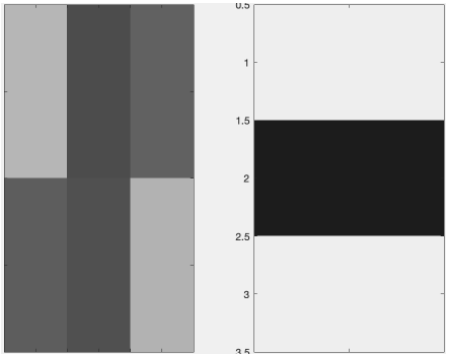
40%, E = 0.0296



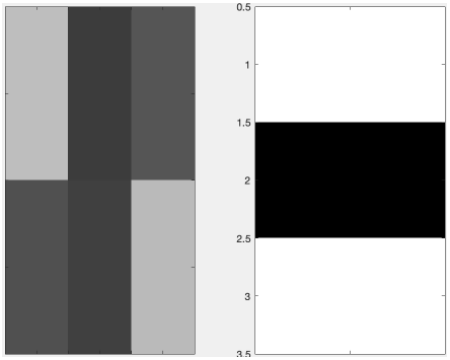
50%, E = 0.0085



60%, E = 0.0035



80%, E = 0.00075



100%, E = 0.00015

More Complex Neural Networks

These notes have discussed the simplest structure used in neural networks.

There are many more network structures to learn about, including

- Deep networks with many hidden layers used in “deep learning” - essentially very complex functions to relate complex inputs and outputs.

- Convolutional networks, in which various matrices are mathematically convoluted with input submatrices and scanned across an input matrix in order to help identify specific features, such as faces in an image or words in a sound spectrum.

- Recurrent networks with feedforward and feedback of information, useful in analyzing sequential inputs such as speech.

- Different types of networks arranged in parallel and series.

Search the website of the MathWorks, makers of MATLAB, for their neural network tools
https://mathworks.com