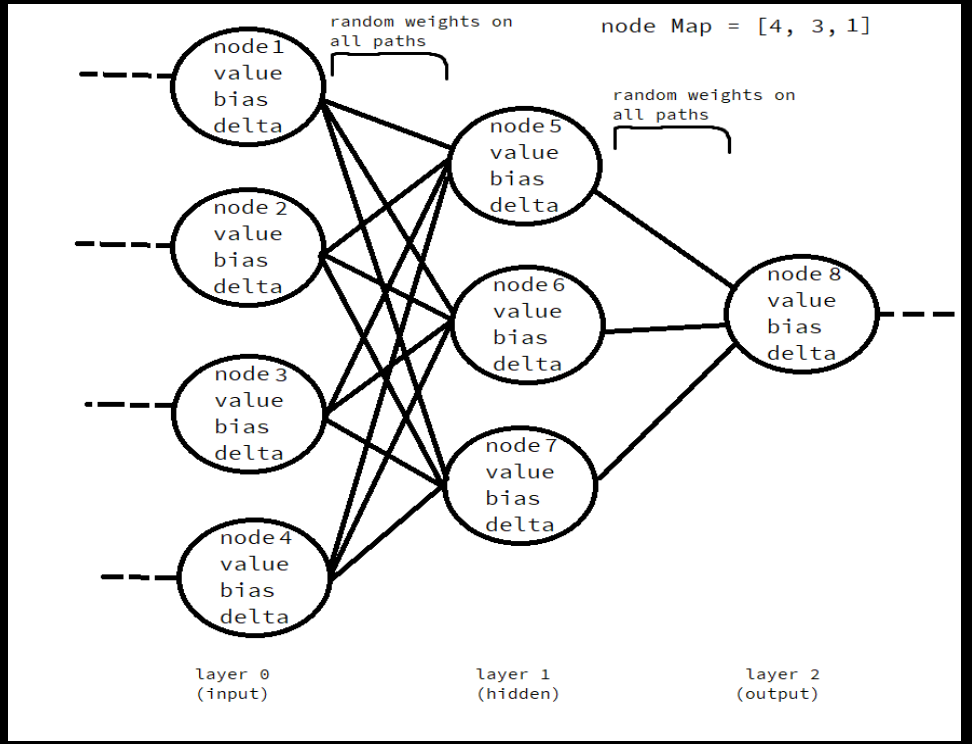
Artificial Neural Networks and How They Work

By Ariel Leston

Background Info – What is it?

- An Artificial Neural Network is a collection of biased nodes that are linked with weighted paths
- The biases and weights are incrementally adjusted while training
- To ultimately be able to pass inputs through these weights and biases so that the final nodes output is the correct output



Background Info – Key Functions

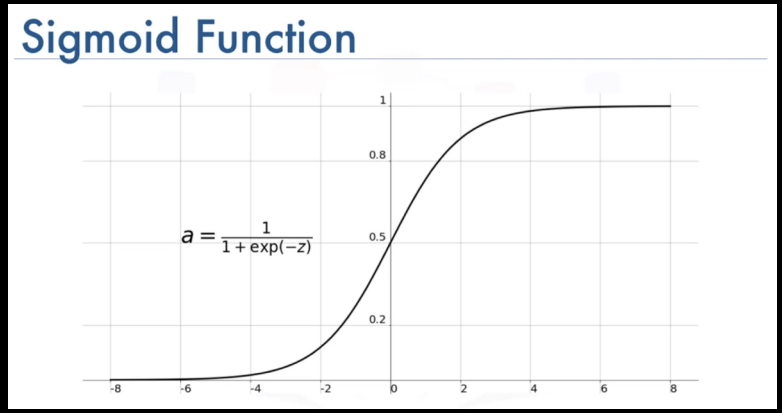
- Train
  - Activate
  - Back Propagate
  - Weight Update
- Test
  - Activate

```
def train(self, train, target, lr=0.15, maxLoops=10**5, convg=10**-5):
    startT4 = time.time()
    self.loops = 0
    while self.loops < maxLoops and self.maxDiff > convg:
        for x in range(len(train)):
            self.activate(train[x])
            self.backPropagate(target[x])
            self.weightUpdate(lr)
            self.maxDiff = max(self.diffs)
        self.loops += 1
    self.times[3] += time.time() - startT4
```

```
def test(self, xlist):
    self.activate(xlist)
    return self.layers[-1][-1].a
```

Background Info – Activation

- Activate function will push the input value of a node, added with its bias, through the Sigmoid calculation and set the result as that nodes output value
- Sigmoid calculation will squeeze given values to between 0 and 1
- The function repeats this process through all nodes, but not nodes on the first layer




```
def activate(self, xlist):
    startT1 = time.time()
    x, w = 0, 0
    for layer in self.layers:
        for node in layer:
            if node.position[0] == 0:
                node.a = xlist[x]
            else:
                sumInVal = 0
                for inputNode in node.inputs:
                    sumInVal += inputNode.a * self.paths[w][2]
                    w += 1
                node.a = 1 / (1 + (math.e ** -(sumInVal + node.bias)))
            x += 1
    self.times[0] += time.time() - startT1
```

Background Info – Back Propagate

- Calculates the error from activation function of each node while training
- And sets the delta values of each node to its error
- Is done in reverse, starting from last layer and moving upward

```
let sigmoid(x) = g(x) and g'(x) = g(x) * (1-g(x))
where x is input value
and y will be the expected or corresponding output for that input

so, delta value of a last layer node:
      delta = (y - g(x)) * g'(x)

and delta value of a hidden layer node (not on first or last layer):
     delta = (sumValue) * g'(x)
     where sumValue = Σ(delta value * path weight) of all linked output nodes

and lastly no delta value on a first layer node
```

```
def backPropagate(self, y):
    startT2 = time.time()
    self.layers.reverse(); self.paths.reverse()
    i, x = 0, 0
    for layer in self.layers:
        if i == 0:
            for endNode in layer:
                endNode.delta = (y - endNode.a) * (endNode.a * (1 - endNode.a))
        elif 0 < i < len(self.layers) - 1:
            for node in layer:
                sumVal = 0
                for output in node.outputs:
                    sumVal += self.paths[x][2] * output.delta
                    x += 1
                node.delta = (node.a * (1 - node.a)) * sumVal
        i += 1
    self.layers.reverse(); self.paths.reverse()
    self.times[1] += time.time() - startT2
```

Background Info – Weight Update

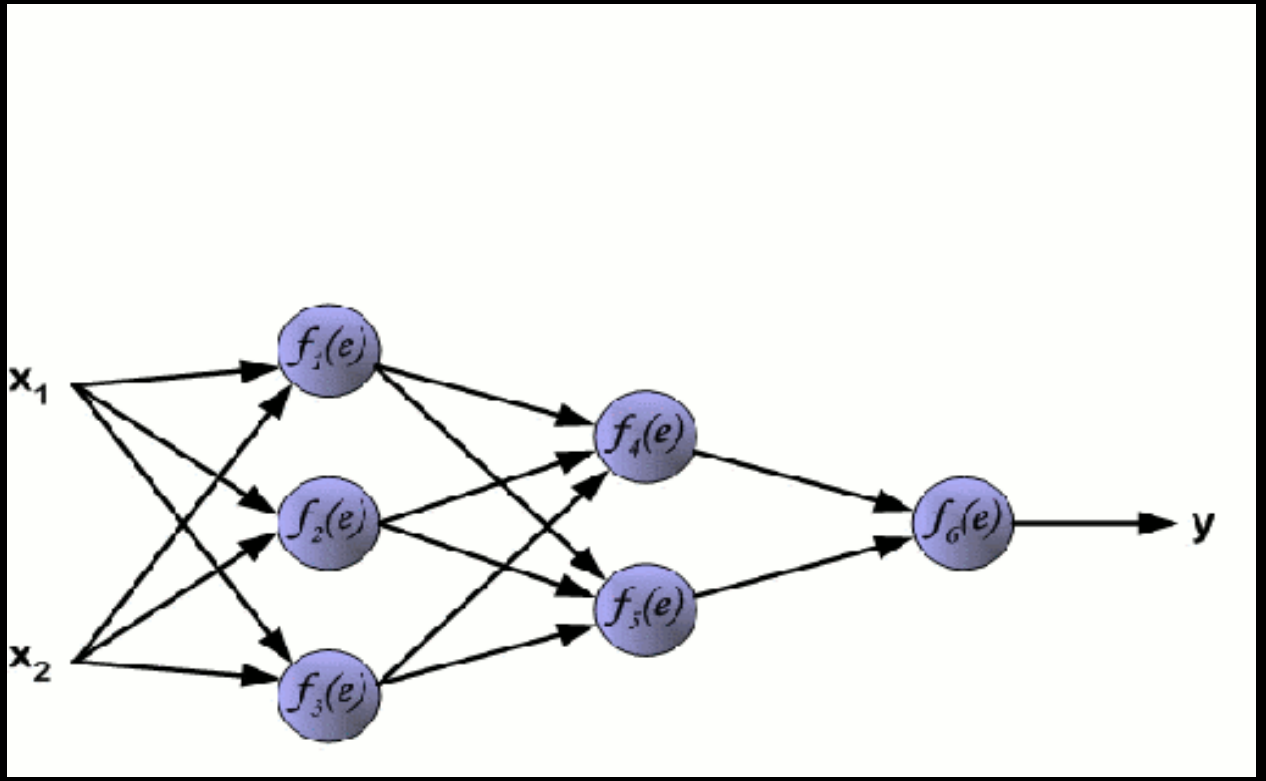
- Uses the delta values (error) to adjust all the biases and weights
- Aims for less error on next loop, over and over
- Also keeps track of the difference between old weights and new weights so the training function can see when it has converged

```
for each path (ex. node1 --> node2)
newWeight = current weight + (learning rate *  node1 output * node2 delta)

for each nodes bias value (not first layer though)
newBias = current bias + (learning rate * this nodes delta)
```

```python
def weightUpdate(self, lr):
    startT3 = time.time()
    self.diffs = []
    for path in self.paths:
        inNode, outNode, weight = path[0], path[1], path[2]
        newWeight = weight + (lr * inNode.a * outNode.delta)
        self.diffs.append(abs(newWeight - weight))
        path[2] = newWeight
    for layer in self.layers[1:]:
        for node in layer:
            newBias = node.bias + (lr * node.delta)
            self.diffs.append(abs(newBias - node.bias))
            node.bias = newBias
    self.times[2] += time.time() - startT3
```

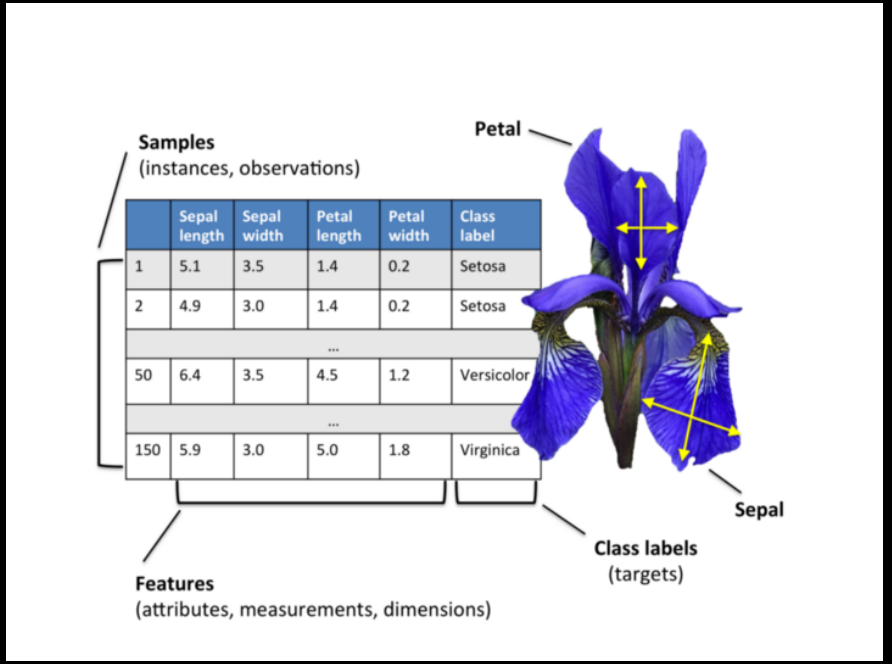
Visual of Training



gifer.com/en/6fyP

Testing the Model

- Testing model with flower Iris dataset
- Set up a Binary Classification Problem
- To predict the correct flower type between two types of flower based on its measurements



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica