Tutorial: Computational Graphs and MLPs

Marijn van Knippenberg Vlado Menkovski

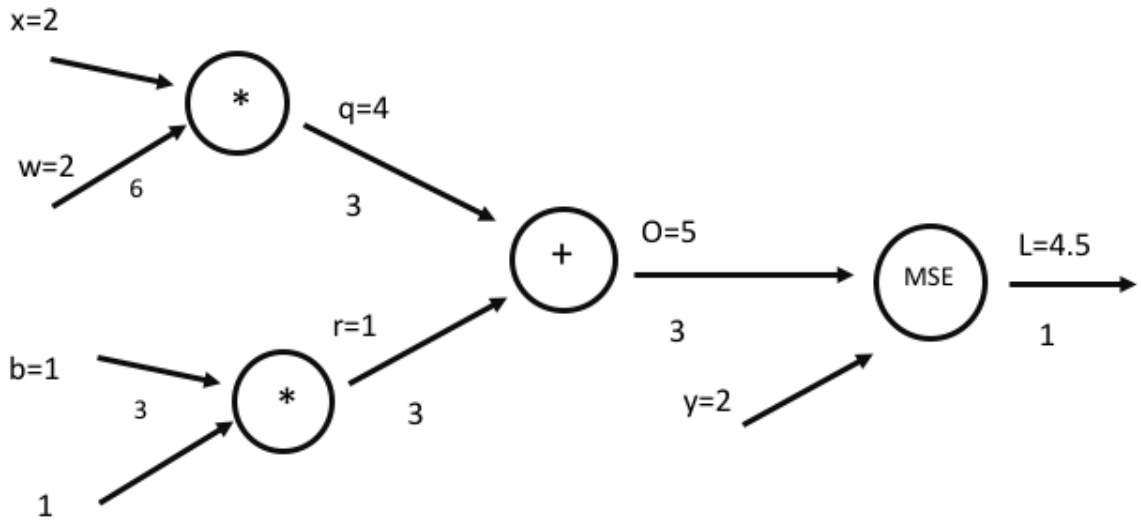
Eindhoven University of Technology

{m.s.v.knippenberg,v.menkovski}@tue.nl

April 3, 2018

Outline

Computational Graph



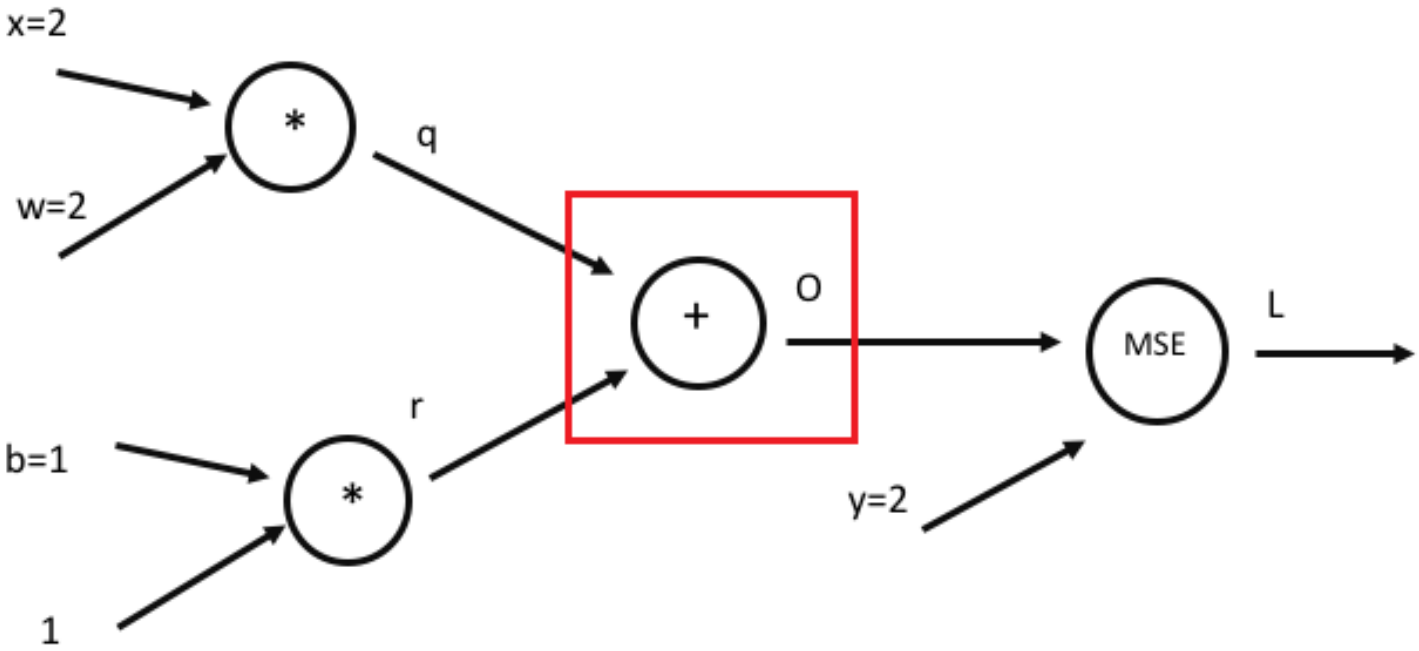
Generic Node

Node template with support for forward and backward propagation.

```
1  class Node(object):
2
3      def __init__(self, inputs):
4          self.inputs = inputs
5
6      @abstractmethod
7      def forward(self):
8          ''' Feed-forward the result '''
9          raise NotImplementedError()
10
11     @abstractmethod
12     def backward(self, d):
13         ''' Back-propagate the error
14             d is the delta of the next node '''
15         raise NotImplementedError()
```

Addition Node

Sum inputs $o = q + r$.
Partial derivatives $\frac{\partial o}{\partial q} = 1, \frac{\partial o}{\partial r} = 1$.



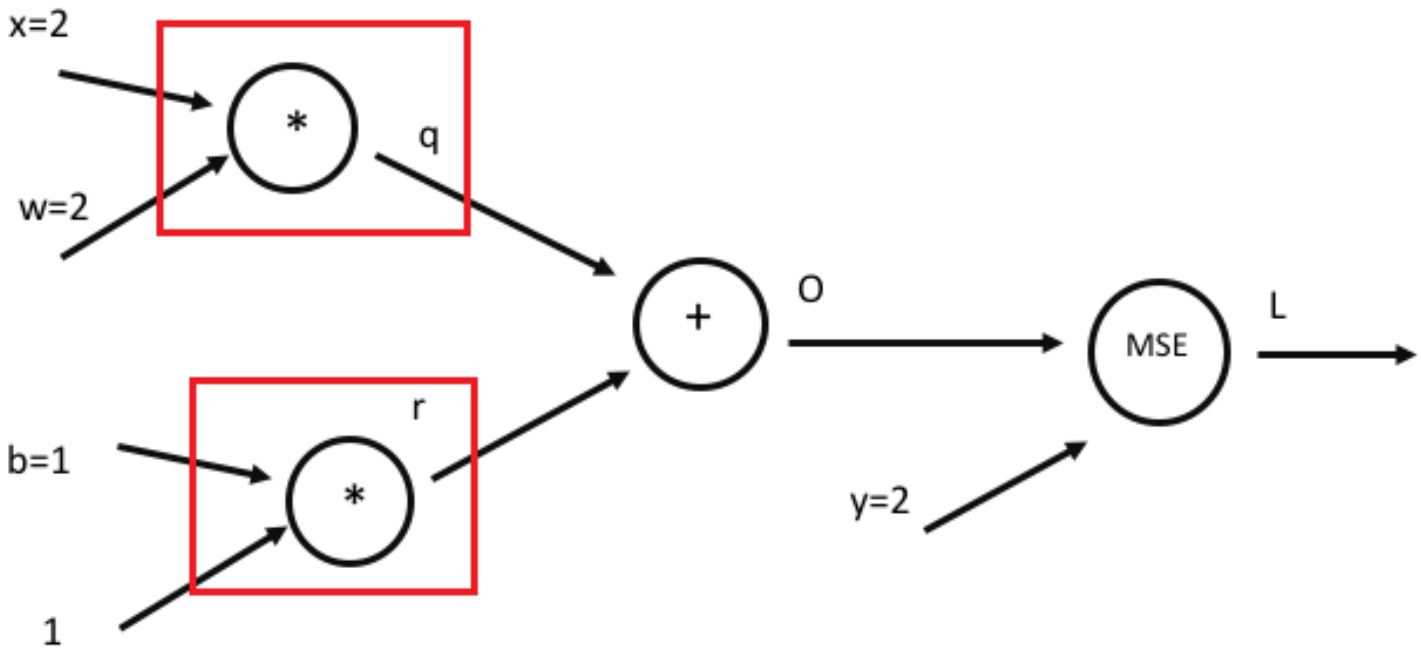
Addition Node

Sum inputs $o = q + r$.
Partial derivatives $\frac{\partial o}{\partial q} = 1, \frac{\partial o}{\partial r} = 1$.

```
1  class AdditionNode(Node):
2
3      def forward(self):
4          self.output = sum([i.forward() for i in self.inputs])
5          return self.output
6
7      def backward(self, d):
8          for i in self.inputs:
9              i.backward(d)
```

Multiplication Node

Multiply two inputs $q = x \cdot w, r = b \cdot 1$.
Partial derivatives $\frac{\partial q}{\partial x} = w$, $\frac{\partial q}{\partial w} = x$, and $\frac{\partial r}{\partial b} = 1$.



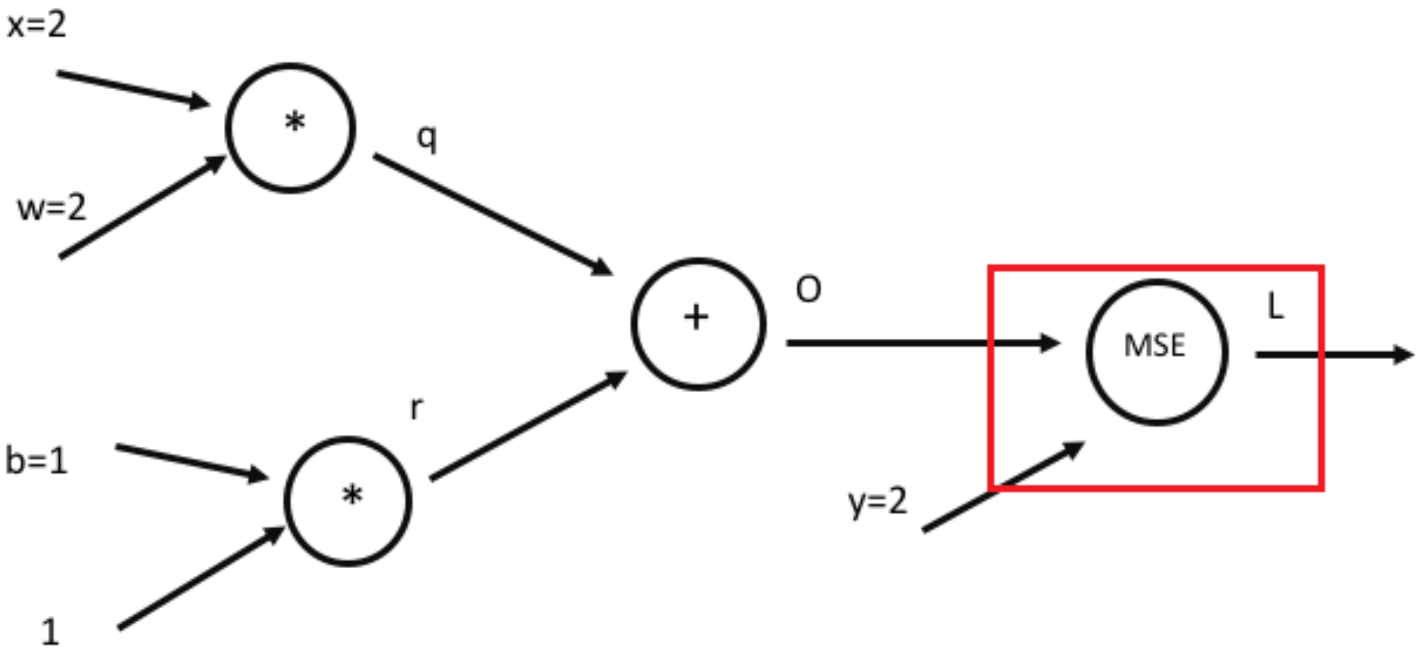
Multiplication Node

Multiply two inputs $q = x \cdot w, r = b \cdot 1$.
Partial derivatives $\frac{\partial q}{\partial x} = w$, $\frac{\partial q}{\partial w} = x$, and $\frac{\partial r}{\partial b} = 1$.

```
1  class MultiplicationNode(Node):
2
3      def forward(self):
4          self.output =
5              self.inputs[0].forward() * self.inputs[1].forward()
6          return self.output
7
8      def backward(self, d):
9          self.inputs[0].backward(d * self.inputs[1].output)
10         self.inputs[1].backward(d * self.inputs[0].output)
```

Mean Squared Error Node

Calculate MSE of two values $L = \frac{1}{2}(o - y)^2$.
Partial derivatives $\frac{\partial L}{\partial o} = o - y$ and $\frac{\partial L}{\partial y} = y - o$.

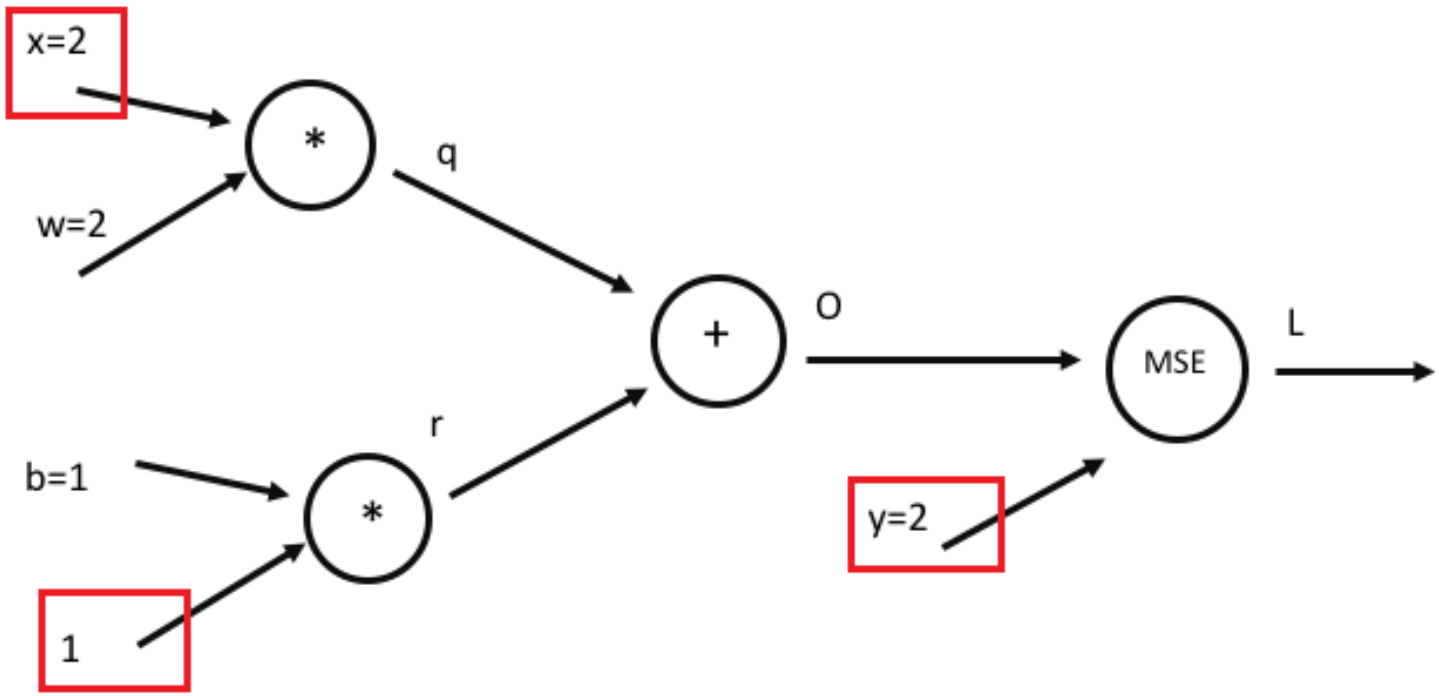


Mean Squared Error Node

Calculate MSE of two values $L = \frac{1}{2}(o - y)^2$.
Partial derivatives $\frac{\partial L}{\partial o} = o - y$ and $\frac{\partial L}{\partial y} = y - o$.

```
1  class MSENode(Node):
2
3      def forward(self):
4          self.output = 0.5 * (
5              self.inputs[0].forward() - self.inputs[1].forward()
6              )**2
7          return self.output
8
9      def backward(self, d):
10         self.inputs[0].backward(
11             d * (self.inputs[0].output - self.inputs[1].output))
12         self.inputs[1].backward(
13             d * (self.inputs[1].output - self.inputs[0].output))
```

Constant Value Node

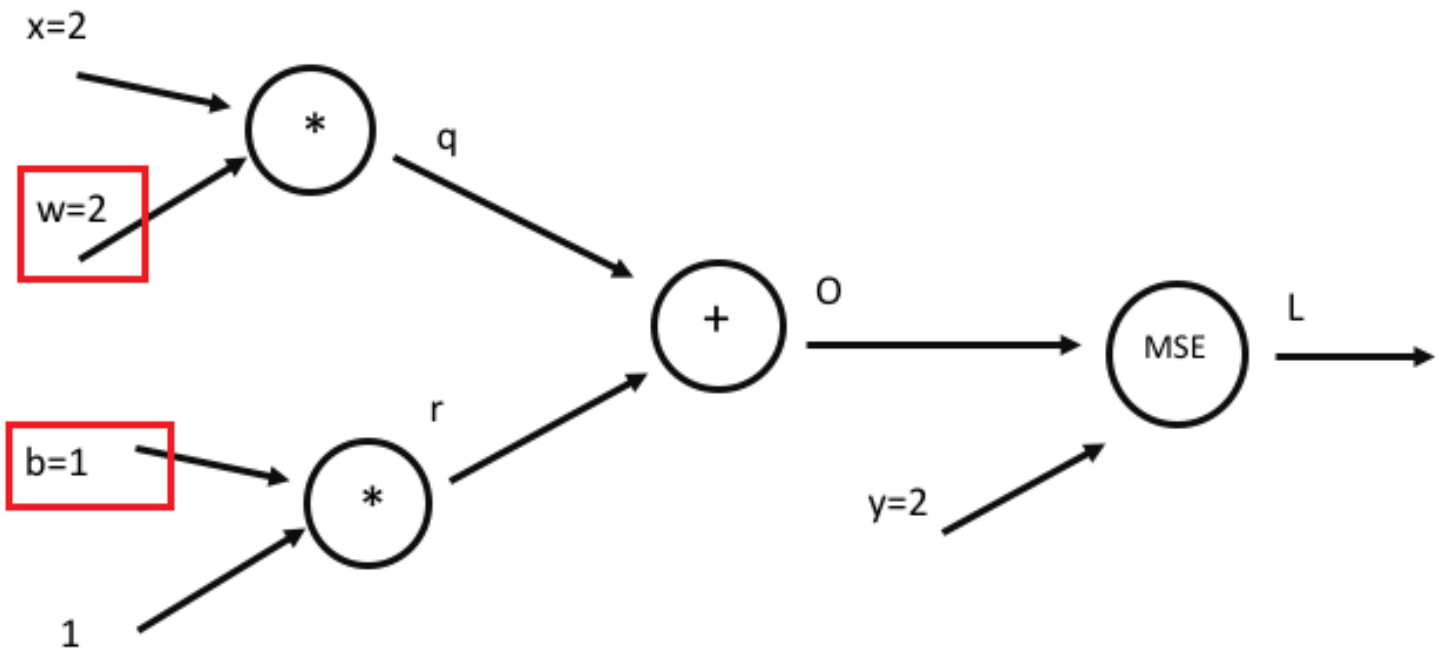


Constant Value Node

Input node, so no backward pass. Used to create data input points, or to set constants in the network (the multiplication factor for the bias, for example).

```
1  class ConstantNode(Node):
2
3      def __init__(self, val):
4          self.output = val
5
6      def forward(self):
7          return self.output
8
9      def backward(self, d):
10         pass
```

Variable Value Node

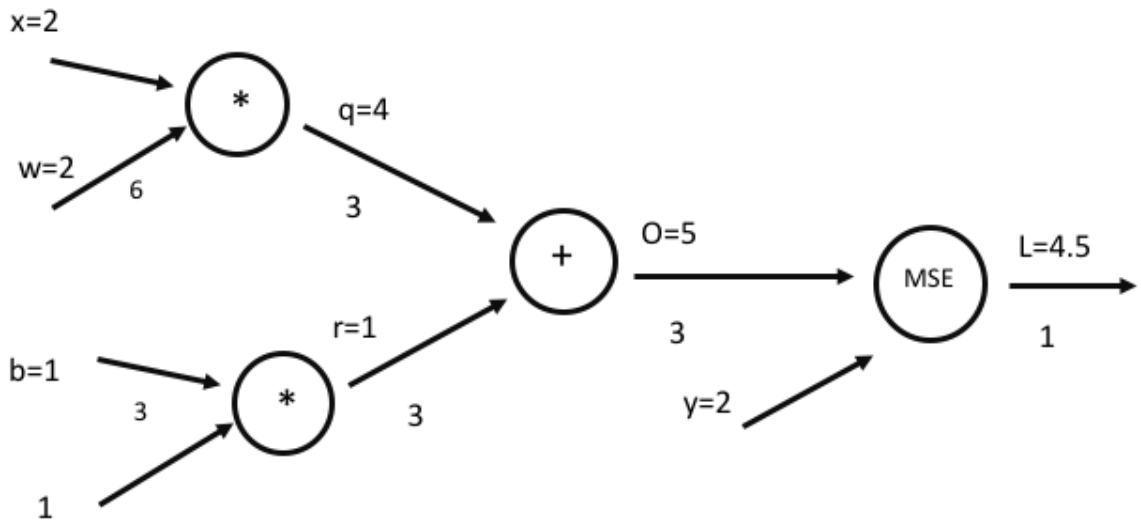


Variable Value Node

Input node with a trainable value. Like the constant node, does not pass backwards, but unlike constant node does update its own value. In this example, updates are performed with $\eta = 0.01$.

```
1  class VariableNode(Node):
2
3      def __init__(self, val, name):
4          self.output = val
5          self.name = name
6
7      def forward(self):
8          return self.output
9
10     def backward(self, d):
11         self.output -= 1e-2 * d # Gradient Descent
```

Sample Graph



Sample Graph

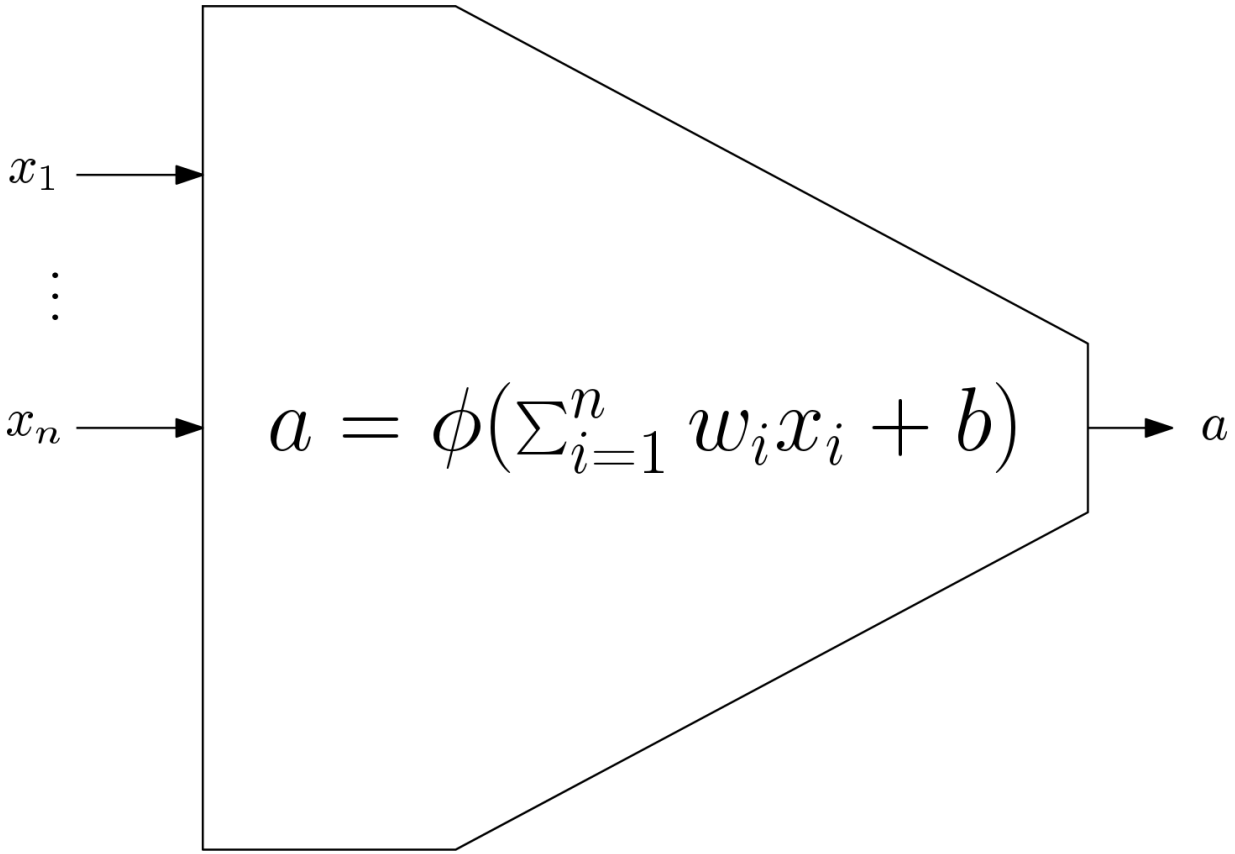
```
1  class SampleGraph(object):
2
3      def __init__(self, x, y, w, b):
4          ''' x: input
5              y: expected output
6              w: initial weight
7              b: initial bias '''
8          self.graph = MSENode([
9              AdditionNode([
10                 MultiplicationNode([
11                     ConstantNode(x),
12                     VariableNode(w, "w")
13                 ]),
14                 MultiplicationNode([
15                     VariableNode(b, "b"),
16                     ConstantNode(1)
17                 ])
18             ]),
19             ConstantNode(y)
20         ])
21
22     def forward(self):
23         return self.graph.forward()
24
25     def backward(self, d):
26         self.graph.backward(d)
```

Sample Graph

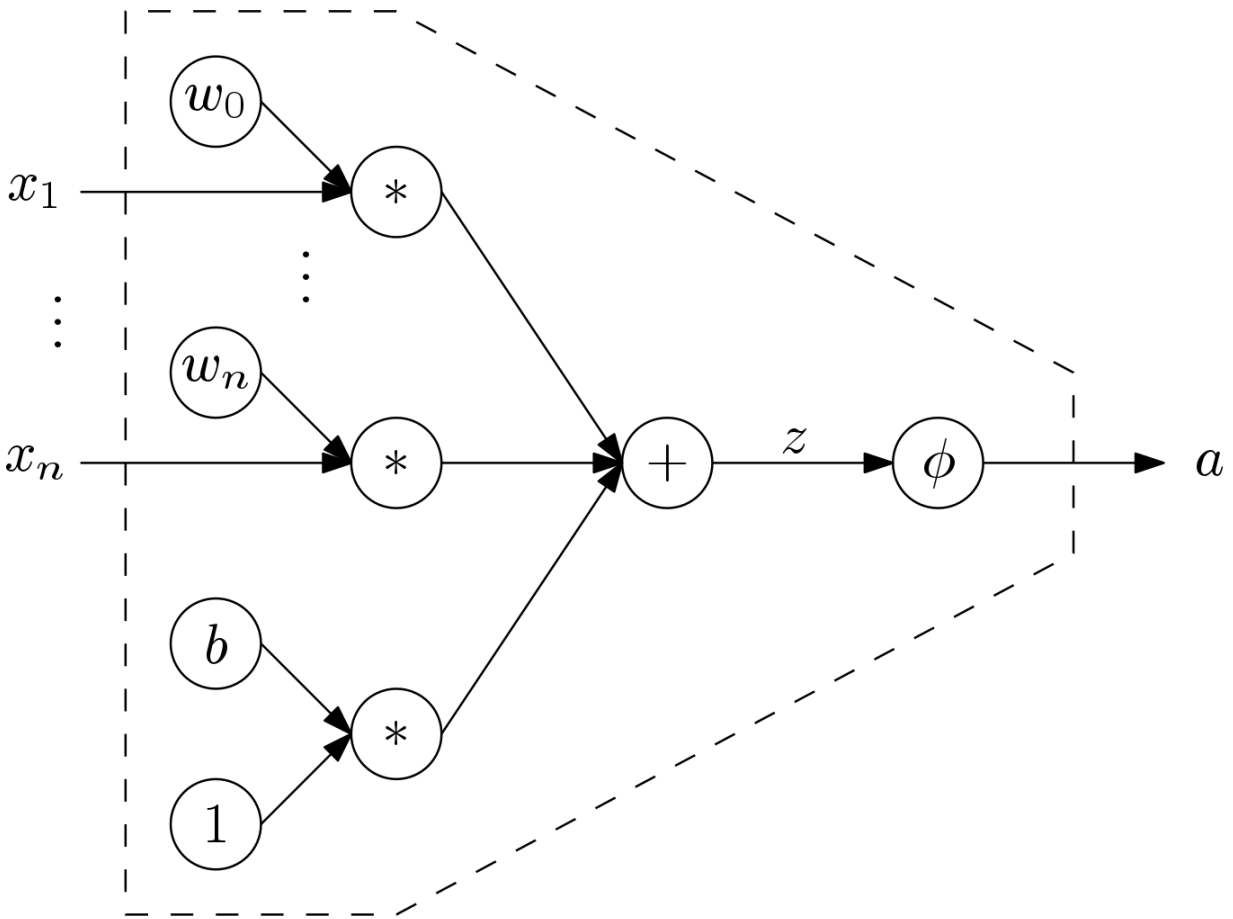
```
# Create sample graph with initial values
sg = SampleGraph(2, 2, 2, 1)
# Run single forward pass to obtain prediction
print(sg.forward())
# Run single backward pass to update weight and bias
# Note that dL/dL = 1.
sg.backward(1.0)
# Run another single forward pass to obtain updated prediction
print(sg.forward())
```

Initial settings: $w = 2, b = 1, \eta = 0.01$.
First forward pass: $L = 4.5$.
First backward pass: $\frac{\partial L}{\partial w} = 6, \frac{\partial L}{\partial b} = 3 \rightarrow w = 1.94, b = 0.97$
Second forward pass: $L \approx 4.06$.

Neuron



Neuron



Neuron

```python
class Neuron(Node):

    def __init__(self, num_inputs):
        self.x = [ConstantNode(0) for i in range(num_inputs)] # Set x later to change input
        # Initialize a weight for each input
        self.w = [VariableNode(0, f"w_{i}") for i in range(num_inputs)]
        # Initialize a single bias for all inputs
        self.b = VariableNode(0, "b")

        # Multiply each pair of inputs and weights
        mults = [MultiplicationNode([self.x[i], self.w[i]])
                 for i in range(num_inputs)]
        # Multiply bias by 1
        mults.append(MultiplicationNode([self.b, ConstantNode(1)]))

        # Sum all multiplication results and apply sigmoid function
        self.graph = SigmoidNode([
            AdditionNode(
                mults
            )
        ])

    def forward(self):
        return self.graph.forward()

    def backward(self, d):
        self.graph.backward(d)
```

Sigmoid Node

Calculate Sigmoid function $\phi(x) = \frac{1}{1+e^{-x}}$.

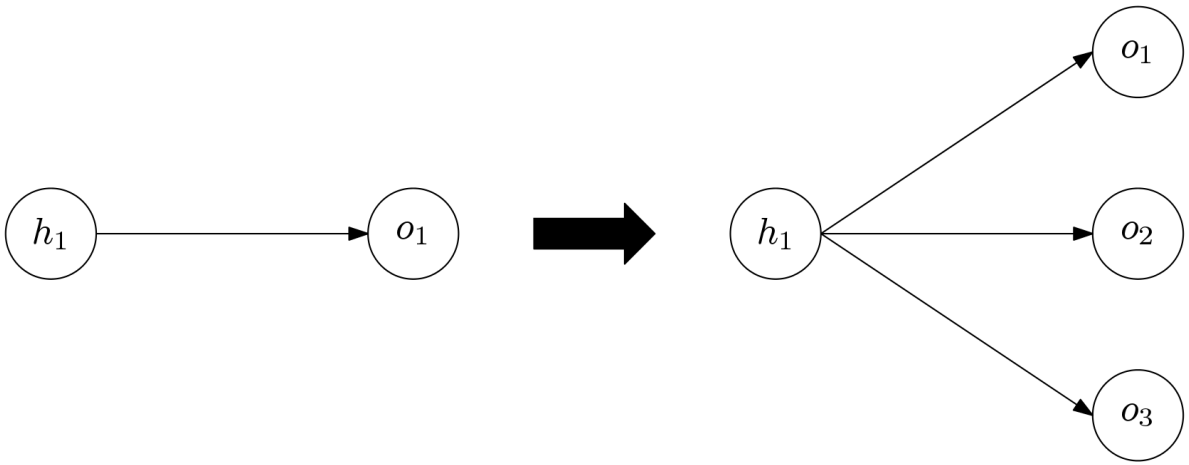
Derivative $\frac{\partial \phi(x)}{\partial x} = \phi(x) * (1 - \phi(x))$.

```
1  class SigmoidNode(Node):
2
3      def forward(self):
4          self.output = 1.0 / (
5              1.0 + math.exp(-self.inputs[0].forward()))
6          return self.output
7
8      def backward(self, d):
9          self.inputs[0].backward(
10             d * self.output * (1.0 - self.output))
```

Neuron for MLP

In an MLP, the output of each neuron is used multiple times:

$$\frac{\partial L_{total}}{\partial a_{h_1}} = \frac{\partial L_{o_1}}{\partial a_{o_1}} + \frac{\partial L_{o_2}}{\partial a_{o_2}} + \frac{\partial L_{o_3}}{\partial a_{o_3}}$$



Each node needs to know where it output goes in order to back-propagate.

Extended Node Definition

Instead of calling `backward` from the last node in the graph to update the weights, call it recursively from the first node of the graph.

```
1  class Node(object):
2
3      def __init__(self, inputs):
4          self.inputs = inputs
5          for i in self.inputs:
6              i.outputs.append(self)
7          self.outputs = []
8
9      @abstractmethod
10     def forward(self):
11         raise NotImplementedError()
12
13     @abstractmethod
14     def backward(self, i):
15         ''' i: Which input to backpropagate for '''
16         # Call backward on output node(s)
17         # Return local gradient for input i
18         raise NotImplementedError()
```

Input Layer

```
1  class InputLayer(Node):
2
3      def __init__(self, num_inputs):
4          self.nodes = [ConstantNode(0) for i in range(num_inputs)]
5
6      def forward(self):
7          pass
8
9      def backward(self, i):
10         for node in self.nodes:
11             for output in node.outputs:
12                 outputs.backward(None)
13
14     # Set sample x
15     def set_inputs(self, values):
16         for node, value in zip(self.nodes, values):
17             node.output = value
```

Hidden Layer

```
1  class HiddenLayer(Node):
2
3      def __init__(self, inputs, num_neurons, activation='sigmoid'):
4          self.neurons = [Neuron(inputs.nodes, activation=activation)
5                  for i in num_neurons]
6
7      def forward(self):
8          pass
9
10     def backward(self, i):
11         pass
```

Output Layer (MSE)

```
1  class OutputLayer(Node):
2
3      def __init__(self, inputs):
4          self.expected = [ConstantNode(0) for i in inputs]
5          self.nodes = [MSENode([i, e])
6                  for i, e in zip(inputs.nodes, self.expected)]
7          self.graph = AdditionNode(self.nodes)
8
9      def forward(self):
10         self.output = self.graph.forward()
11         return self.output
12
13     def backward(self, i):
14         pass
15
16     # Set sample y
17     def set_expected(self, values):
18         for node, value in zip(self.expected, values):
19             node.output = value
```

MLP

```
# Input layer with 5 input values
network_in = InputLayer(5)
# First hidden layer with 10 neurons
network_out = HiddenLayer(network_in, 10)
# Second hidden layer with 10 neurons
network_out = HiddenLayer(network_in, 10)
# Ouput layer with single output: loss
network_out = OutputLayer(network_out)
# Forward-propagate
network_in.set_inputs([1, 2, 3, 4, 5])
network_out.set_expected([1, 0, 1, 1, 2, 1, 2, 2, 3, 2])
network_out.forward()
# Back-propagate
network_in.backward(None)
```

MLP