Tutorial: Computational Graphs and MLPs

Marijn van Knippenberg   Vlado Menkovski

Eindhoven University of Technology

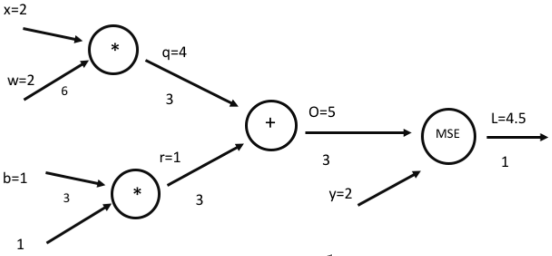
{*m.s.v.knippenberg,v.menkovski*}*@tue.nl*

March 29, 2018

Outline

1. Introduction
2. Simple Computational Nodes
3. Computation Graphs
4. Layered Networks

Computational Graph



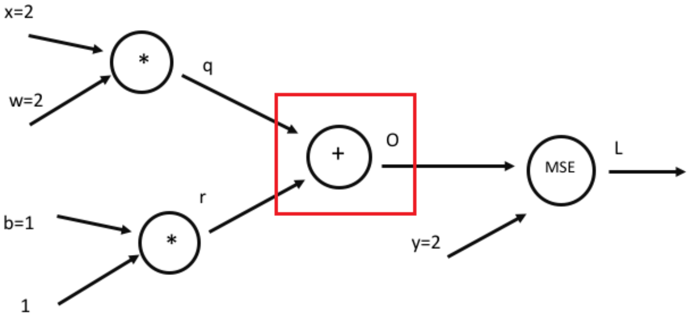
Generic Node

Node template with support for forward and backward propagation.

```python
class Node(object):

    def __init__(self, inputs):
        self.inputs = inputs

    @abstractmethod
    def forward(self):
        ''' Feed-forward the result '''
        raise NotImplementedError()

    @abstractmethod
    def backward(self, d):
        ''' Back-propagate the error
            d is the delta of the next node '''
        raise NotImplementedError()
```

Addition Node

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



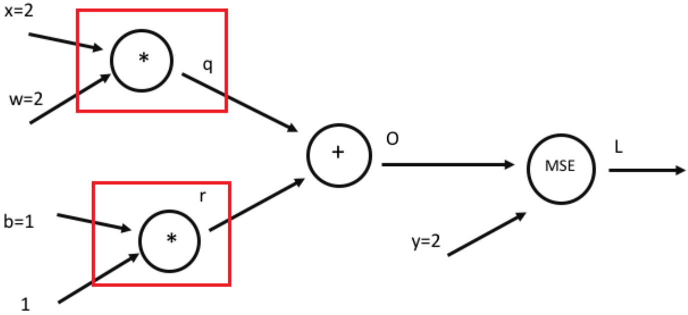
Addition Node

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

```
class AdditionNode(Node):

    def forward(self):
        self.output = sum([i.forward() for i in self.inputs])
        return self.output

    def backward(self, d):
        for i in self.inputs:
            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 $\frac{\partial r}{\partial b} = 1$.



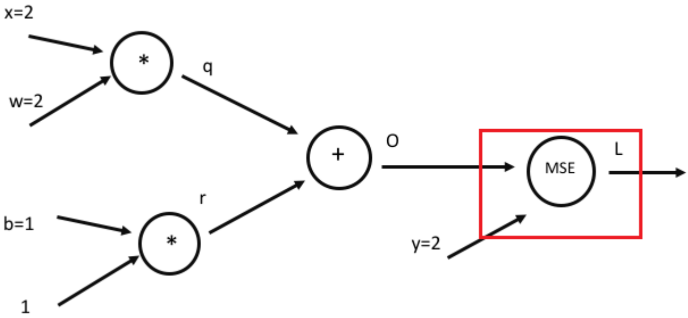
Multiplication Node

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 $\frac{\partial r}{\partial b} = 1$.

```
class MultiplicationNode(Node):

    def forward(self):
        self.output =
            self.inputs[0].forward() * self.inputs[1].forward()
        return self.output

    def backward(self, d):
        self.inputs[0].backward(d * self.inputs[1].output)
        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 $\frac{\partial L}{\partial y} = y - o$.

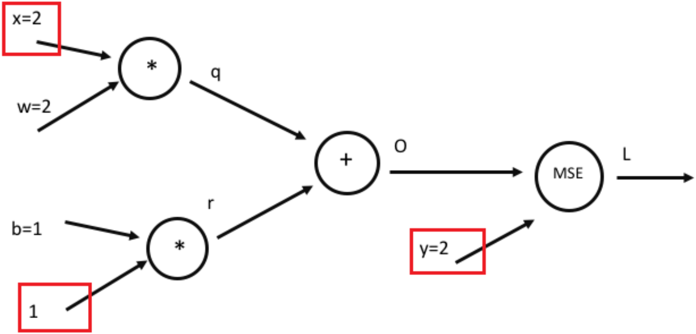


Mean Squared Error Node

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

```
class MSENode(Node):

    def forward(self):
        self.output = 0.5 * (
            self.inputs[0].forward() - self.inputs[1].forward()
            )**2
        return self.output

    def backward(self, d):
        self.inputs[0].backward(
            d * (self.inputs[0].output - self.inputs[1].output))
        self.inputs[1].backward(
            d * (self.inputs[1].output - self.inputs[0].output))
```

Constant Value Node

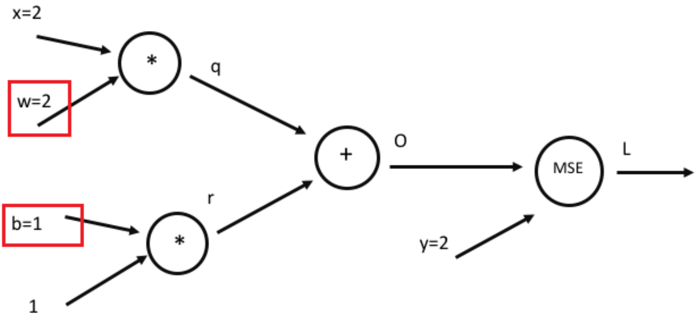


Constant Value Node

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).

```
class ConstantNode(Node):

    def __init__(self, val):
        self.output = val

    def forward(self):
        return self.output

    def backward(self, d):
        pass
```

Variable Value Node

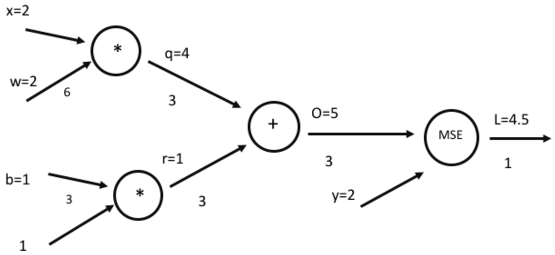


Variable Value Node

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 $\eta = 0.01$.

```
class VariableNode(Node):

    def __init__(self, val, name):
        self.output = val
        self.name = name

    def forward(self):
        return self.output

    def backward(self, d):
        self.output -= 1e-2 * d # Gradient Descent
```

Sample Graph



Sample Graph

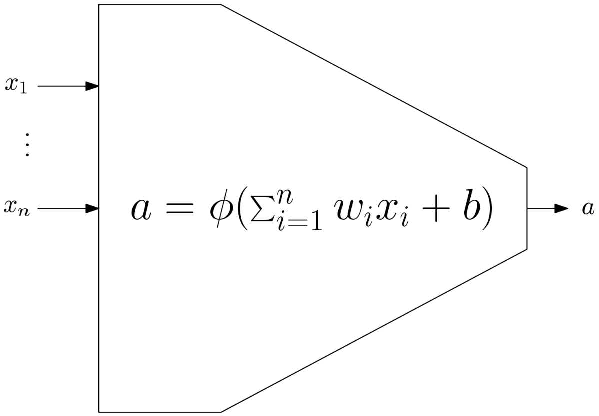
```python
class SampleGraph(object):

    def __init__(self, x, y, w, b):
        ''' x: input
            y: expected output
            w: initial weight
            b: initial bias '''
        self.graph = MSENode([
            AdditionNode([
                MultiplicationNode([
                    ConstantNode(x),
                    VariableNode(w, "w")
                ]),
                MultiplicationNode([
                    VariableNode(b, "b"),
                    ConstantNode(1)
                ])
            ]),
            ConstantNode(y)
        ])

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

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

Sample Graph

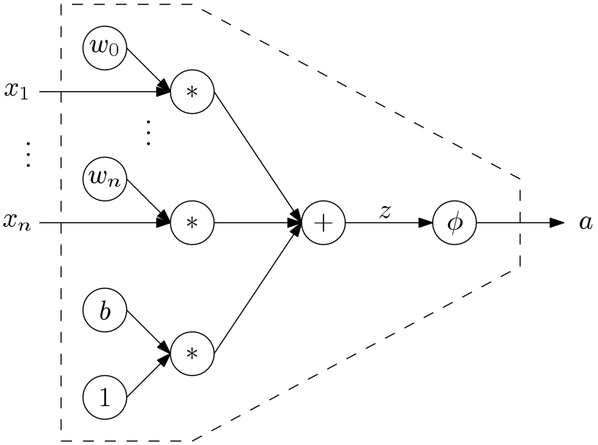
```
# Create sample graph with initial values
sg = SampleGraph(2, 2, 2, 1)
# Run single forward pass to obtain prediction
print(sg.forward())
# Run single backward pass to update weight and bias
# Note that dL/dL = 1.
sg.backward(1.0)
# Run another single forward pass to obtain updated prediction
print(sg.forward())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

Initial settings: $w = 2, b = 1, \eta = 0.01$.
First forward pass: $y = 2, \hat{y} = 4.5$.
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: $y = 2, \hat{y} \approx 4.06$.

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

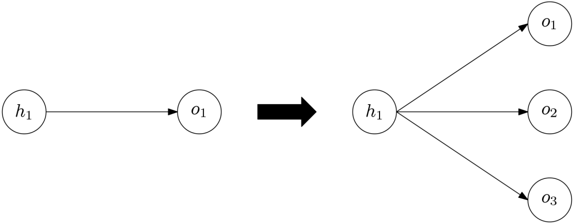
Calculate Sigmoid function $\phi(x) = \frac{1}{1+e^{-x}}$.
Derivative $\frac{\partial \phi(x)}{\partial x} = \phi(x) * (1 - \phi(x))$.

```
class SigmoidNode(Node):

    def forward(self):
        self.output = 1.0 / (
            1.0 + math.exp(-self.inputs[0].forward()))
        return self.output

    def backward(self, d):
        self.inputs[0].backward(
            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

```
class Node(object):

    def __init__(self, inputs):
        self.inputs = inputs
        for i in self.inputs:
            i.outputs.append(self)
        self.outputs = []

    @abstractmethod
    def forward(self):
        raise NotImplementedError()

    @abstractmethod
    def backward(self, d):
        raise NotImplementedError()
```

Input Layer

```
class InputLayer(Node):

    def __init__(self, num_inputs):
        self.nodes = [ConstantNode(0) for i in range(num_inputs)]

    def forward(self):
        pass

    def backward(self, i):
        for node in self.nodes:
            for output in node.outputs:
                outputs.backward(None)

    # Set sample x
    def set_inputs(self, values):
        for node, value in zip(self.nodes, values):
            node.output = value
```

Hidden Layer

```
class HiddenLayer(Node):

    def __init__(self, inputs, num_neurons, activation='sigmoid'):
        self.neurons = [Neuron(inputs.nodes, activation=activation)
                for i in num_neurons]

    def forward(self):
        pass

    def backward(self, i):
        pass
```

Output Layer (MSE)

```python
class OutputLayer(Node):

    def __init__(self, inputs, activation='mse'):
        self.expected = [ConstantNode(0) for i in inputs]
        self.nodes = [MSENode([i, e])
                for i, e in zip(inputs.nodes, self.expected)]
        self.graph = AdditionNode(self.nodes)

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

    def backward(self, i):
        pass

    # Set sample y
    def set_expected(self, values):
        for node, value in zip(self.expected, values):
            node.output = value
```

MLP

```python
# 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_out.forward()
# Back-propagate
network_in.backward(None)
```

MLP