kptan86@gmail.com

Simple Implementation of fully connected deep-learning architecture

In [1]:
```
# import libraries
import uuid
import numpy as np
```

Define activation functions and its derivatives

In [2]:
```
################
### Logistic ###
################
def logistic(x):
    return 1./(1+np.exp(x))
# end def
def diff_logistic(x):
    return x*(1-x)
# end def


################
##### ReLU #####
################
def ReLU(x):
    return max(0, x)
# end def
def diff_ReLU(x):
    if x <= 0:
        return 0
    else:
        return 1
    # end if
# end def
```

Define network class

kptan86@gmail.com

In [3]:
```
class Net:
    def __init__(self):
        self.layers = []
        self.learning_rate = 0.05
    # end def

    def add_layer(self, layer):
        self.layers.append(layer)
        # register the neuron
        for neuron in self.layers[-1].get_all_neurons():
            neuron.net = self
        # end for
        self.reset_layers_index()
    # end def

    def reset_layers_index(self):
        for i in range(len(self.layers)):
            self.layers[-1].index = i
        # end for
    # end def

    # forward pass
    def forward(self):
        for layer in self.layers[1:]:
            for neuron in layer.neurons:
                neuron.forward()
            # end for
        # end for

        neurons = self.layers[-1].get_all_neurons()
        output = [_.value for _ in neurons]

        return np.array(output)
    # end def

    # backward pass
    def backprop(self):
        for layer in reversed(self.layers):
            for neuron in layer.neurons:
                neuron.backprop()
            # end for
        # end for
    # end def

    def get_all_neurons(self):
        for layer in self.layers:
            for neuron in layer.get_all_neurons():
                yield neuron
            # end for
        # end for
    # end def

    def delta_update(self):
        pass
    # end def

# end class
```

Define Layer class

kptan86@gmail.com

In [4]:
```
class Layer:
    def __init__(self, net):
        self.index = None

        self.net = net
        self.neurons = []
    # end def

    def add_neuron(self, neuron):
        self.neurons.append(neuron)
        self.neurons[-1].layer = self
        self.neurons[-1].net   = self.net
    # end def

    def size(self):
        return len(self.neurons)
    # end def

    def get_all_neurons(self):
        for neuron in self.neurons:
            yield neuron
        # end for
    # end def

    def fully_connect(self, next_layer):
        neuron_qs = list(next_layer.get_all_neurons())
        for neuron_p in self.get_all_neurons():
            neuron_p.add_children(neuron_qs)
        # end for
    # end def
# end class
```

Define Neuron class

kptan86@gmail.com

In [5]:
```
class Neuron:
    def __init__(self, name, func, diff_func):
        self.uuid = str(uuid.uuid4())
        self.name  = name
        self.net   = None
        self.layer = None

        self.value    = None
        self.delta    = None
        self.parents  = {}
        self.children = {}
        self.weights  = {}
        self.biases   = {}

        self.func      = func
        self.diff_func = diff_func
        self.learning_rate = None
    # end def

    def set_activation_func(self, func, diff_func):
        self.func      = func
        self.diff_func = diff_func
    # end def

    def add_parents(self, parents):
        for parent in parents:
            if parent.uuid not in self.parents:
                self.parents.update({parent.uuid: parent})
                parent.add_children([self])
            # end if
        # end for
    # end def

    def add_children(self, children):
        for child in children:
            if child.uuid not in self.children:
                self.children.update({child.uuid: child})
                child.add_parents([self])
            # end if
        # end for
    # end def

    def init_weights(self):
        '''Glorot initialization'''
        j = self.layer.index
        l = min(j+1, len(self.net.layers) - 1)
        nj = self.net.layers[j].size()
        nl = self.net.layers[l].size()

        lb = -np.sqrt(6) / np.sqrt(nj+nl)
        ub =  np.sqrt(6) / np.sqrt(nj+nl)

        for parent in self.parents:
            self.weights[parent] = np.random.uniform(lb, ub)
            self.biases [parent] = 0
        # end for
    # end def

    def forward(self):
        s = 0.0
        for parent_id, parent in self.parents.items():
            value = parent.value
            s += self.weights[parent_id]*value + self.biases[parent_id]
        # end for
        self.value = self.func(s)
    # end def
```


DO NOT DISTRIBUTE

kptan86@gmail.com

Initialize a network

In [6]:
```
net = Net()
```

Initialize layers

In [7]:
```
l1 = Layer(net)
l2 = Layer(net)
l3 = Layer(net)
l4 = Layer(net)
```

In [8]:
```
_layers = [l1, l2, l3, l4]
```

Initialize neurons and add to layers

In [9]:
```
architecture = [3, 10, 5, 3]
```

In [10]:
```
# Input layer
for l in range(len(architecture)):
    # get corresponding layer
    layer = _layers[l]

    # initialize all neurons
    num_neurons = architecture[l]
    for i in range(num_neurons):
        name = str((l, i))
        _neuron = Neuron(
            name      = name,
            func      = logistic,
            diff_func = diff_logistic
        ) # end neuron
        # add to layer
        layer.add_neuron(_neuron)
    # end for
# end for
```

Adding layers to net

In [11]:
```
for _layer in _layers:
    net.add_layer(_layer)
# end for
```

Fully connect the layers

In [12]:
```
for i in range(len(net.layers)-1):
    layer_p = net.layers[i]
    layer_q = net.layers[i+1]
    layer_p.fully_connect(layer_q)
# end for

'''
# or more explictly:
l1.fully_connect(l2)
l2.fully_connect(l3)
l3.fully_connect(l4)
''';
```

kptan86@gmail.com

Initialize weights for all neurons

In [13]:
```
for neuron in net.get_all_neurons():
    neuron.init_weights()
# end for
```

Set input values at input layer

In [14]:
```
# assumed input values
x = (0.5, 0.3, 0.8)

layer_in = net.layers[0]
neurons_in = list(layer_in.get_all_neurons())
for i in range(len(x)):
    neurons_in[i].value = x[i]
# end for
```

Forward pass (compute value)

In [15]:
```
out = net.forward()
out
```

Out[15]:
```
array([0.38125397, 0.22368634, 0.66421561])
```

Compute error and values at output layer

In [16]:
```
# assumed target values
target = np.array([0.5, 0.5, 0.5])
```

In [17]:
```
layer_out = net.layers[-1]
neurons_out = list(layer_out.get_all_neurons())
for i in range(len(neurons_out)):
    neuron = neurons_out[i]

    # error
    err = neuron.value - target[i]
    # regularizer value
    l2_reg = 2*sum(neuron.weights.values())

    # compute delta
    delta = neuron.cal_grad() * (err+l2_reg)
    neuron.delta = delta
# end for
```

Backward pass (Backpropagation)

In [18]:
```
for layer in reversed(net.layers[:-1]):
    for neuron in layer.get_all_neurons():
        neuron.backprop()
    # end for
# end for
```

Demonstrating vanishing gradient

kptan86@gmail.com

In [19]:
```
print('Demonstrating vanishing gradient')
for i in range(len(net.layers)):
    deltas = [neuron.delta for neuron in net.layers[i].get_all_neurons()]
    abs_mean_delta = np.mean(np.abs(deltas))
    print ('layer-%d: delta=%4.4f' % (i+1, abs_mean_delta))
```

```
Demonstrating vanishing gradient
layer-1: delta=0.0039
layer-2: delta=0.0196
layer-3: delta=0.0616
layer-4: delta=0.4051
```

Update weights

In [20]:
```
for neuron in net.get_all_neurons():
    neuron.update_weights()
# end for
```

In [21]:
```
net.forward()
```

Out[21]:
```
array([0.38945027, 0.23961475, 0.65030059])
```

In [ ]:
```
```


DO NOT DISTRIBUTE