DEEP LEARNING GROUP - WEEK 2

BACKPROPAGATION

Material from http://cs231n.github.io/optimization-2/#intro

PUTTING THINGS IN CONTEXT

To compute the weights and biases of our network we need to minimize the cost function

- So far our reading has discussed L2 norm as cost function (but rarely used in practice)
- For deep neural networks to add information learning capability, hidden layers can't be linear functions
- Gradient can't be computed analytically, nor easily approximated. So use gradient descend

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

No need to use all data. Works well because examples in the training data are correlated.

```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

If mini-batch contains only a single example, the process is called Stochastic Gradient Descent (SGD) (or also sometimes on-line gradient descent). But mini-batch gradient descent is sometimes referred as SGD

PUTTING THINGS IN CONTEXT

```
class Network(object):
...
    def update_mini_batch(self, mini_batch, eta):
        """Update the network's weights and biases by applying
        gradient descent using backpropagation to a single mini batch.
        The "mini_batch" is a list of tuples "(x, y)", and "eta"
        is the learning rate."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
        self.weights = [w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)]
```

INTUITION

Let our example function $f(x, y, z) = (x + y)z.$

Be expressed as $q = x + y \quad f = qz.$

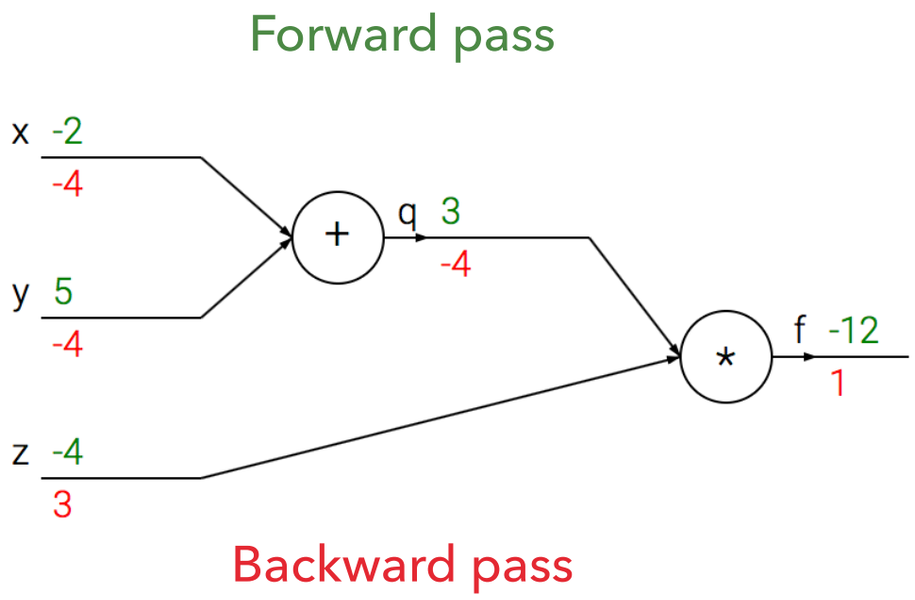
With partial derivatives $\frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q, \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1.$

That, using the chain rule, give us the gradient we are interested in $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$

```
# set some inputs
x = -2; y = 5; z = -4

# perform the forward pass
q = x + y # q becomes 3
f = q * z # f becomes -12

# perform the backward pass (backpropagation) in reverse order:
# first backprop through f = q * z
dfdz = q # df/dz = q, so gradient on z becomes 3
dfdq = z # df/dq = z, so gradient on q becomes -4
# now backprop through q = x + y
dfdx = 1.0 * dfdq # dq/dx = 1. And the multiplication here is the chain rule!
dfdy = 1.0 * dfdq # dq/dy = 1
```



SIGMOID EXAMPLE

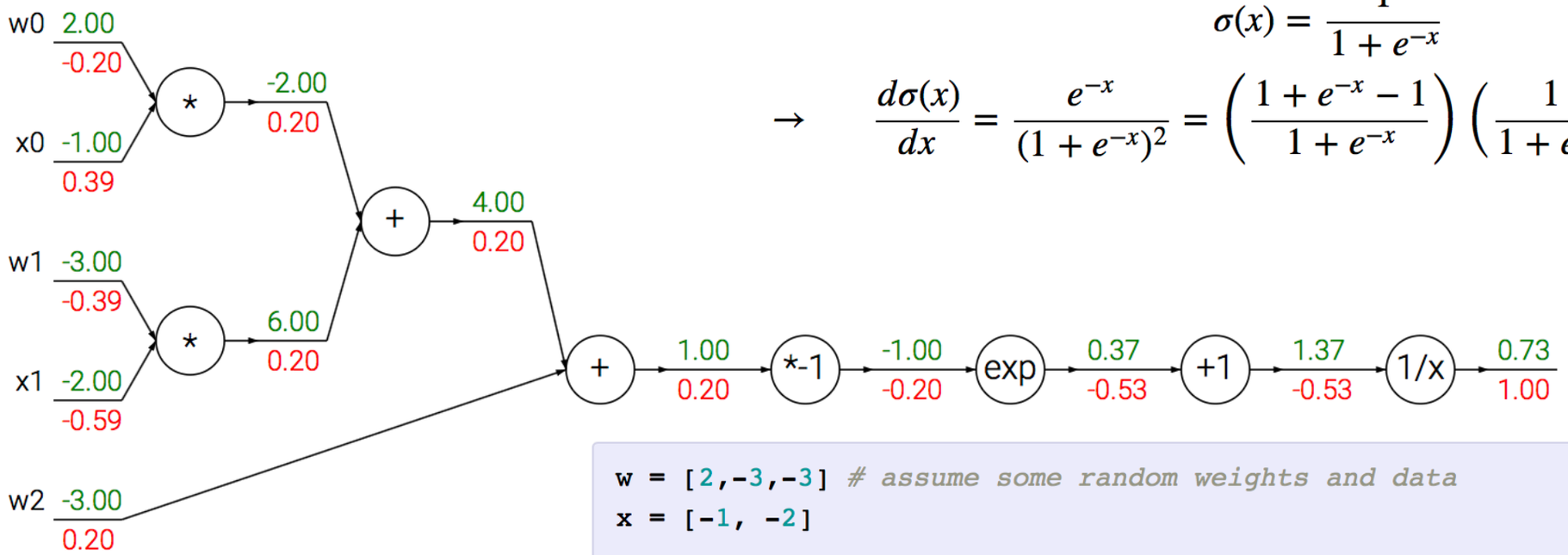
$$f(w,x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$



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

$$\rightarrow \quad \frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left(\frac{1 + e^{-x} - 1}{1 + e^{-x}}\right)\left(\frac{1}{1 + e^{-x}}\right) = (1 - \sigma(x))\,\sigma(x)$$

```
w = [2,-3,-3] # assume some random weights and data
x = [-1, -2]

# forward pass
dot = w[0]*x[0] + w[1]*x[1] + w[2]
f = 1.0 / (1 + math.exp(-dot)) # sigmoid function

# backward pass through the neuron (backpropagation)
ddot = (1 - f) * f # gradient on dot variable, using the sigmoid gradient derivation
dx = [w[0] * ddot, w[1] * ddot] # backprop into x
dw = [x[0] * ddot, x[1] * ddot, 1.0 * ddot] # backprop into w
# we're done! we have the gradients on the inputs to the circuit
```

BACKPROPAGATION

STAGED

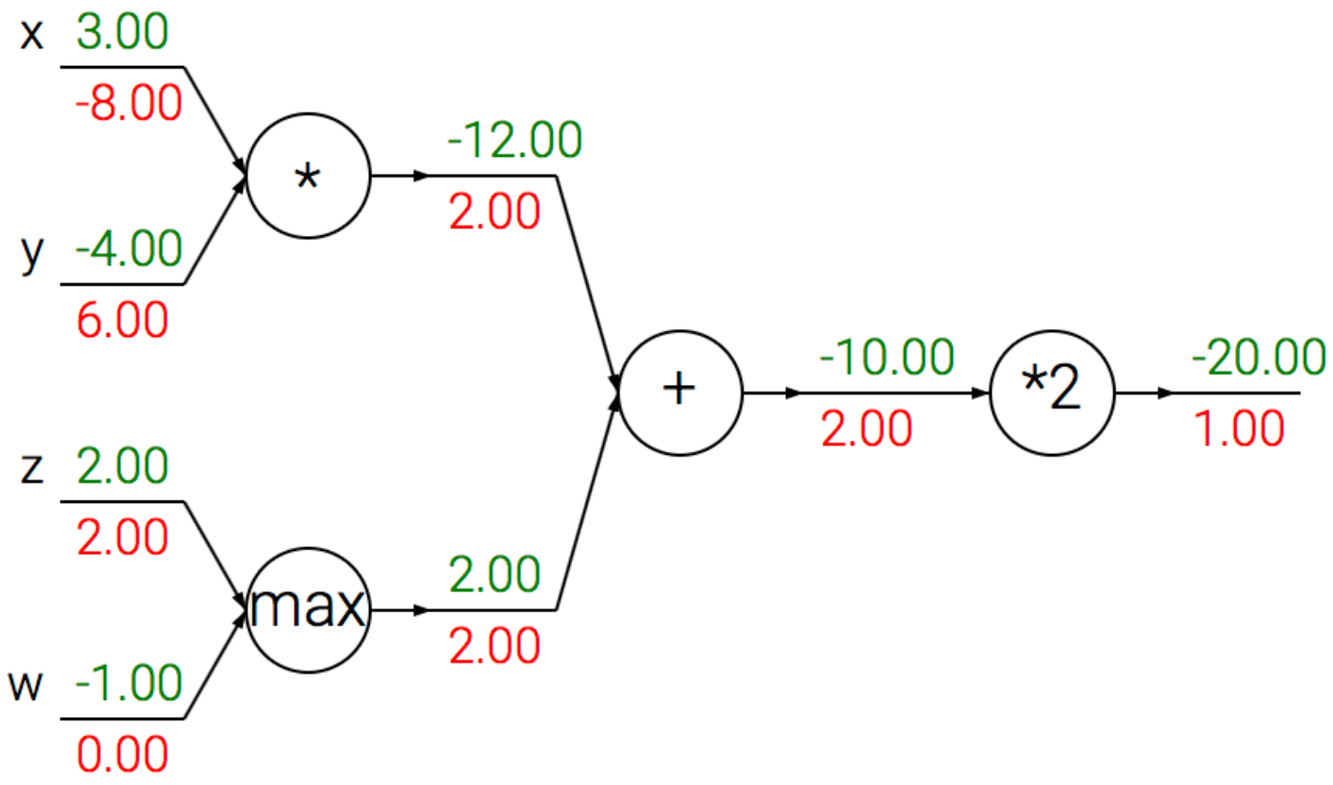
$$f(x,y) = \frac{x + \sigma(y)}{\sigma(x) + (x+y)^2}$$

```
x = 3 # example values
y = -4

# forward pass
sigy = 1.0 / (1 + math.exp(-y)) # sigmoid in numerator   #(1)
num = x + sigy # numerator                               #(2)
sigx = 1.0 / (1 + math.exp(-x)) # sigmoid in denominator #(3)
xpy = x + y                                              #(4)
xpysqr = xpy**2                                          #(5)
den = sigx + xpysqr # denominator                        #(6)
invden = 1.0 / den                                       #(7)
f = num * invden # done!                                 #(8)
```

```
# backprop f = num * invden
dnum = invden # gradient on numerator                    #(8)
dinvden = num                                            #(8)
# backprop invden = 1.0 / den
dden = (-1.0 / (den**2)) * dinvden                       #(7)
# backprop den = sigx + xpysqr
dsigx = (1) * dden                                       #(6)
dxpysqr = (1) * dden                                     #(6)
# backprop xpysqr = xpy**2
dxpy = (2 * xpy) * dxpysqr                               #(5)
# backprop xpy = x + y
dx = (1) * dxpy                                          #(4)
dy = (1) * dxpy                                          #(4)
# backprop sigx = 1.0 / (1 + math.exp(-x))
dx += ((1 - sigx) * sigx) * dsigx # Notice += !! See notes below  #(3)
# backprop num = x + sigy
dx += (1) * dnum                                         #(2)
dsigy = (1) * dnum                                       #(2)
# backprop sigy = 1.0 / (1 + math.exp(-y))
dy += ((1 - sigy) * sigy) * dsigy                        #(1)
# done! phew
```

EXAMPLE



An example circuit demonstrating the intuition behind the operations that backpropagation performs during the backward pass in order to compute the gradients on the inputs. Sum operation distributes gradients equally to all its inputs. Max operation routes the gradient to the higher input. Multiply gate takes the input activations, swaps them and multiplies by its gradient.