

Lecture 4:

Backpropagation and Neural Networks part 1

Where we are...

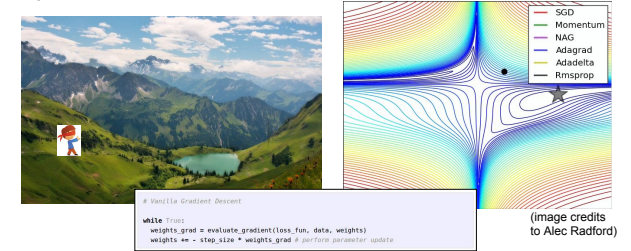
$$s = f(x; W) = Wx \quad \text{scores function}$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \quad \text{SVM loss}$$

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \sum_k W_k^2 \quad \text{data loss + regularization}$$

want $\nabla_W L$

Optimization



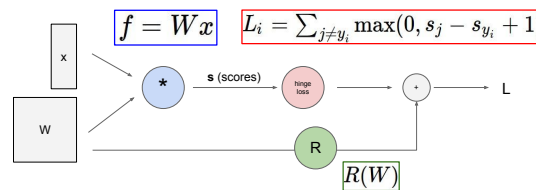
Gradient Descent

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

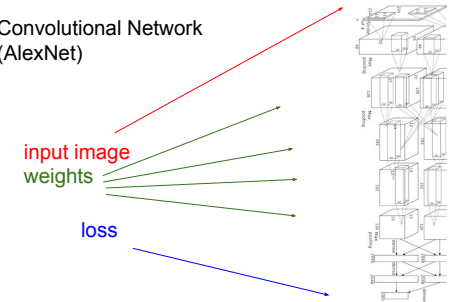
Numerical gradient: slow :, approximate :, easy to write :)
Analytic gradient: fast :, exact :, error-prone :)

In practice: Derive analytic gradient, check your implementation with numerical gradient

Computational Graph

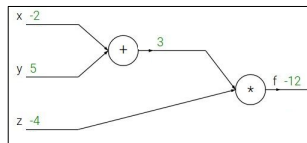


Convolutional Network (AlexNet)



$$f(x, y, z) = (x + y)z$$

e.g. $x = -2, y = 5, z = -4$



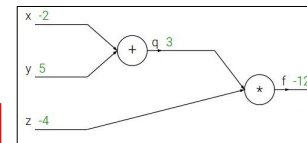
$$f(x, y, z) = (x + y)z$$

e.g. $x = -2, y = 5, z = -4$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



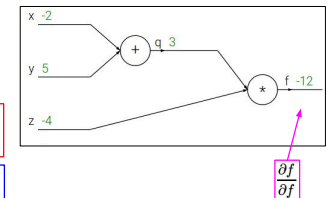
$$f(x, y, z) = (x + y)z$$

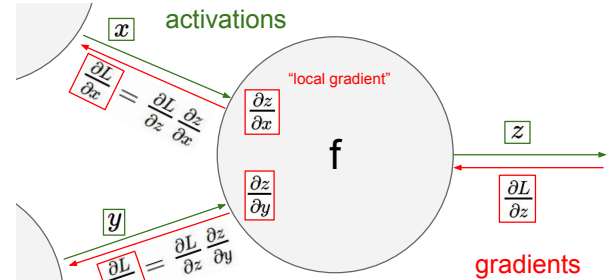
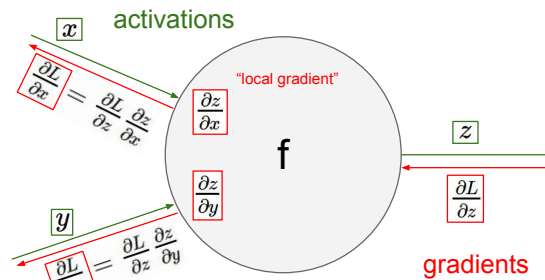
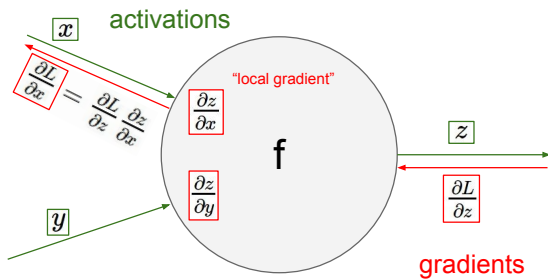
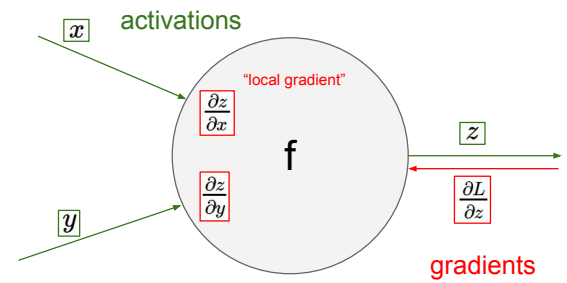
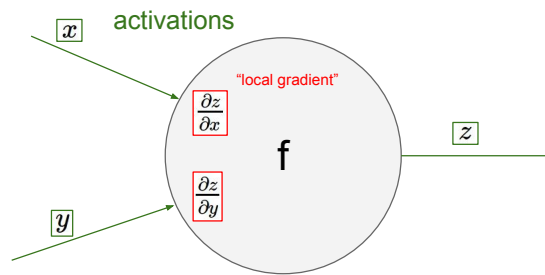
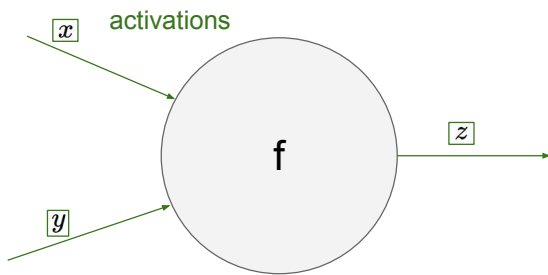
e.g. $x = -2, y = 5, z = -4$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

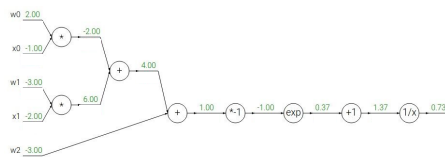
$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

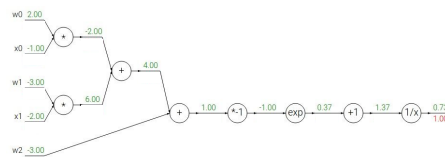




Another example: $f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2 x_2)}}$

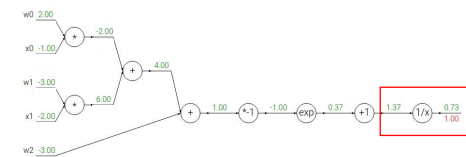


Another example: $f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2 x_2)}}$



$$\begin{array}{lcl} f(x) = e^x & \rightarrow & \frac{df}{dx} = e^x \\ f_a(x) = ax & \rightarrow & \frac{df}{dx} = a \end{array} \quad \left| \quad \begin{array}{lcl} f(x) = \frac{1}{x} & \rightarrow & \frac{df}{dx} = -1/x^2 \\ f_c(x) = c + x & \rightarrow & \frac{df}{dx} = 1 \end{array} \right.$$

Another example: $f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2 x_2)}}$



$$\begin{array}{lcl} f(x) = e^x & \rightarrow & \frac{df}{dx} = e^x \\ f_a(x) = ax & \rightarrow & \frac{df}{dx} = a \end{array} \quad \left| \quad \begin{array}{lcl} f(x) = \frac{1}{x} & \rightarrow & \frac{df}{dx} = -1/x^2 \\ f_c(x) = c + x & \rightarrow & \frac{df}{dx} = 1 \end{array} \right.$$

[illegible]

Diagram illustrating a neural network structure with 4 input nodes (w0, x0, w1, x1) and 1 output node (w2). The network consists of two hidden layers and an output layer.

Forward pass equations shown below the diagram:

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

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

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

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

Diagram illustrating the forward pass of a neural network layer. The input vector $x = [-2.00, -2.00, -3.00, -1.00, 2.00]^T$ is multiplied by the weight matrix $W = \begin{bmatrix} -2.00 & 1.00 & -1.00 & 0.37 & -0.53 \\ -3.00 & 0.00 & 0.00 & -0.53 & 1.37 \\ -3.00 & 6.00 & 4.00 & -0.53 & 1.00 \end{bmatrix}$ to produce the pre-activation vector $z = [1.00, -1.00, -0.53]^T$. The activation function $\sigma(z) = \frac{1}{1 + e^{-z}}$ is applied to each element of z to produce the output vector $a = [0.73, 0.27, 0.62]^T$.

[illegible]

The diagram shows a neural network with three input nodes (w0, w1, w2), two hidden nodes, and one output node. The forward pass values are: w0=2.00, w1=-3.00, w2=-3.00, w0*=-1.00, w1*=2.00, w2*=1.00, w0**=-0.37, w1**=-0.53, w2**=-0.53, w0***=0.73, w1***=1.00, w2***=-1.00. The backpropagation values are: w0*=-1.00, w1*=2.00, w2*=1.00, w0**=-0.37, w1**=-0.53, w2**=-0.53, w0***=0.73, w1***=1.00, w2***=-1.00. The output node is highlighted with a red box.

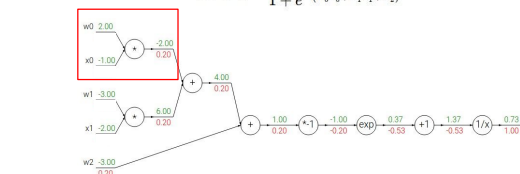
Diagram illustrating the forward pass of a neural network layer for the function $f(x) = \exp(-1/x^2)$. The inputs x_0 and x_1 are processed through a series of operations (multiplication and addition) to produce intermediate values w_0 , w_1 , and w_2 . These are then combined in a final sum and passed through an \exp node to produce the output 0.73. A red box highlights the \exp node and its inputs, with a red text label $(-1) * (-0.20) = 0.20$ indicating the value of the inner expression.

[illegible]

Diagram illustrating the forward pass of a neural network layer with 4 input nodes and 4 output nodes. The input nodes are labeled w_0, w_1, w_2, x_1 . The output nodes are labeled x_1, x_2, x_3, x_4 . The weights are: $w_0=2.00, w_1=-3.00, w_2=-2.00, x_1=-2.00, x_2=-1.00, x_3=1.00, x_4=1.00$. The bias is 0.20. The activation function is sigmoid. The output of the sigmoid function is 0.20. The diagram shows the calculation of the local gradient (0.20) and its gradient (0.20) for the input w_0 . The local gradient is calculated as $1/(1+exp(-x))$ where x is the weighted sum of inputs plus bias. The gradient is the derivative of the sigmoid function, which is $x*(1-x)$.

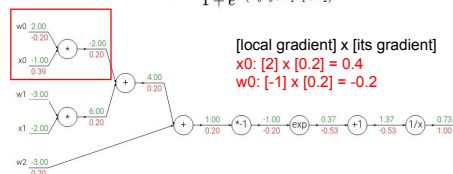
Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 39 13 Jan 2016

Another example: $f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2x_2)}}$



$$\begin{aligned} f(x) &= e^x & \rightarrow & \quad \frac{df}{dx} = e^x & \quad \left| \quad f(x) = \frac{1}{x} & \rightarrow \quad \frac{df}{dx} = -1/x^2 \right. \\ f_s(x) &= ax & \rightarrow & \quad \frac{df}{dx} = a & \quad \left| \quad f_c(x) = c + x & \rightarrow \quad \frac{df}{dx} = 1 \right. \end{aligned}$$

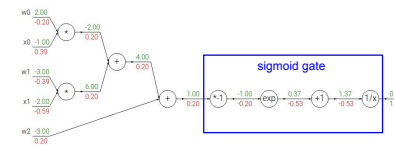
Another example: $f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2x_2)}}$



$$\begin{aligned} f(x) &= e^x & \rightarrow & \quad \frac{df}{dx} = e^x & \quad \left| \quad f(x) = \frac{1}{x} & \rightarrow \quad \frac{df}{dx} = -1/x^2 \right. \\ f_s(x) &= ax & \rightarrow & \quad \frac{df}{dx} = a & \quad \left| \quad f_c(x) = c + x & \rightarrow \quad \frac{df}{dx} = 1 \right. \end{aligned}$$

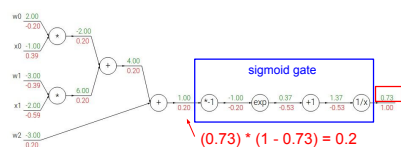
$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2x_2)}} \quad \sigma(x) = \frac{1}{1 + e^{-x}} \quad \text{sigmoid function}$$

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



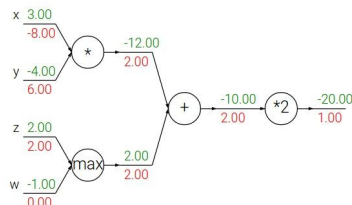
$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2x_2)}} \quad \sigma(x) = \frac{1}{1 + e^{-x}} \quad \text{sigmoid function}$$

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

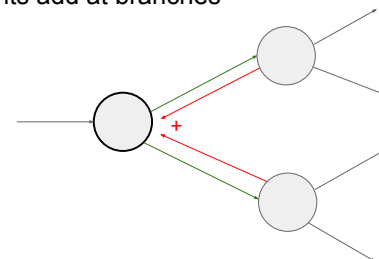


Patterns in backward flow

add gate: gradient distributor
max gate: gradient router
mul gate: gradient... "switcher"?



Gradients add at branches



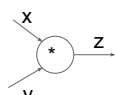
Implementation: forward/backward API



Graph (or Net) object. (Rough psuedo code)

```
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. pass inputs to input gates...
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```

Implementation: forward/backward API

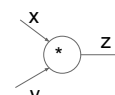


(x,y,z are scalars)

```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        return z
    def backward(dz):
        # dx = ... #todo
        # dy = ... #todo
        return [dx, dy]
```

$$\frac{\partial L}{\partial x} \quad \frac{\partial L}{\partial y}$$

Implementation: forward/backward API

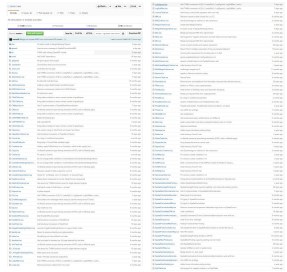


(x,y,z are scalars)

```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        self.x = x # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```

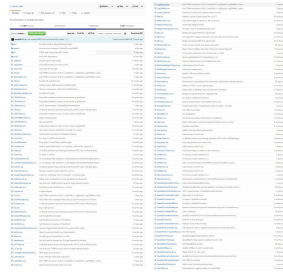


Example: Torch Layers

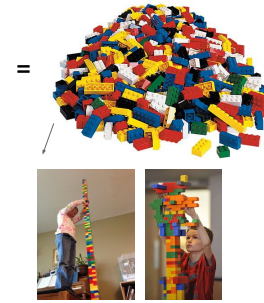


Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 49 13 Jan 2016

Example: Torch Layers



Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 50 13 Jan 2016



Example: Torch MulConstant

$$f(X) = aX$$

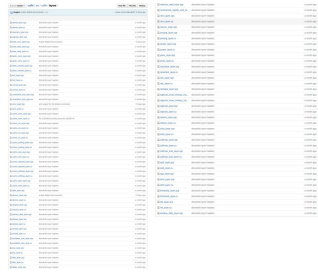
initialization

forward()

backward()

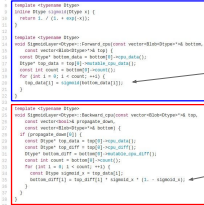
Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 51 13 Jan 2016

Example: Caffe Layers



Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 52 13 Jan 2016

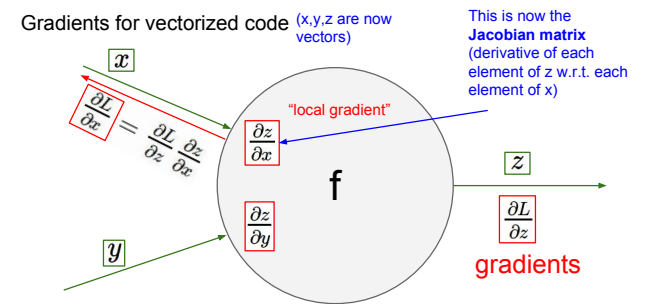
Caffe Sigmoid Layer



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

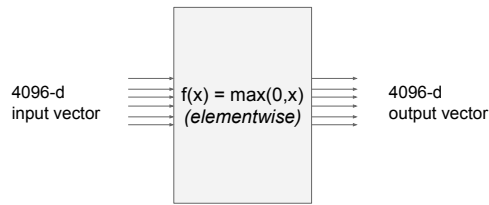
$$(1 - \sigma(x)) \sigma(x) \text{ *top_diff (chain rule)}$$

Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 53 13 Jan 2016



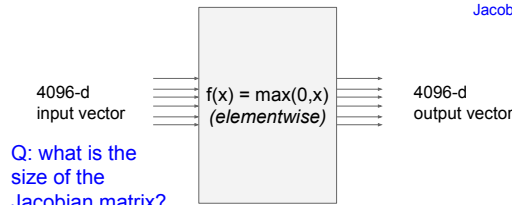
Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 54 13 Jan 2016

Vectorized operations



Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 55 13 Jan 2016

Vectorized operations



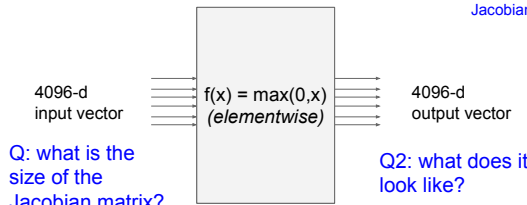
Q: what is the size of the Jacobian matrix?

Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 56 13 Jan 2016

$$\frac{\partial L}{\partial x} = \frac{\partial f}{\partial x} \frac{\partial L}{\partial f}$$

Jacobian matrix

Vectorized operations



Q: what is the size of the Jacobian matrix?
[4096 x 4096!]

Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 57 13 Jan 2016

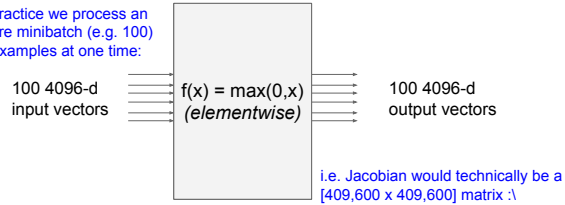
$$\frac{\partial L}{\partial x} = \frac{\partial f}{\partial x} \frac{\partial L}{\partial f}$$

Jacobian matrix

Q2: what does it look like?

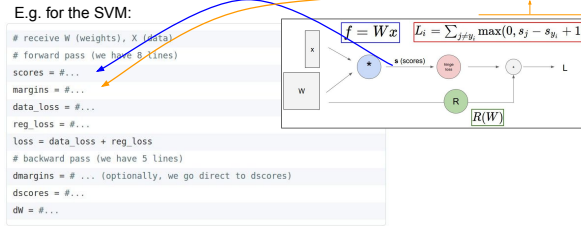
Vectorized operations

in practice we process an entire minibatch (e.g. 100) of examples at one time:



Assignment: Writing SVM/Softmax

Stage your forward/backward computation!



Summary so far

- neural nets will be very large: no hope of writing down gradient formula by hand for all parameters
- **backpropagation** = recursive application of the chain rule along a computational graph to compute the gradients of all inputs/parameters/intermediates
- implementations maintain a graph structure, where the nodes implement the **forward()** / **backward()** API.
- **forward**: compute result of an operation and save any intermediates needed for gradient computation in memory
- **backward**: apply the chain rule to compute the gradient of the loss function with respect to the inputs.



Neural Network: without the brain stuff

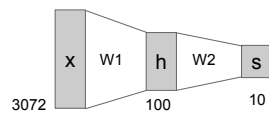
(Before) Linear score function: $f = Wx$

Neural Network: without the brain stuff

(Before) Linear score function: $f = Wx$
(Now) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

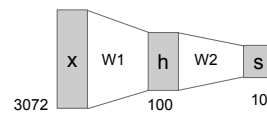
Neural Network: without the brain stuff

(Before) Linear score function: $f = Wx$
(Now) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$



Neural Network: without the brain stuff

(Before) Linear score function: $f = Wx$
(Now) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$



Neural Network: without the brain stuff

(Before) Linear score function: $f = Wx$
(Now) 2-layer Neural Network or 3-layer Neural Network $f = W_3 \max(0, W_2 \max(0, W_1 x))$

Full implementation of training a 2-layer Neural Network needs ~11 lines:

```
01. X = np.array([ [0,0,1], [0,1,1], [1,0,1], [1,1,1] ])
02. y = np.array([ [0,1,1,0] ]).T
03. syn0 = 2*np.random.random((3,4)) - 1
04. syn1 = 2*np.random.random((4,1)) - 1
05. for j in xrange(60000):
06.     l1 = 1/(1+np.exp(-(np.dot(X,syn0))))
07.     l2 = 1/(1+np.exp(-(np.dot(l1,syn1))))
08.     l2_delta = (y - l2)*(12*(1-l2))
09.     l1_delta = l2_delta.dot(syn1.T) * (l1 * (1-l1))
10.     syn1 += l1.T.dot(l2_delta)
11.     syn0 += X.T.dot(l1_delta)
```

from @iamtrask, <http://iamtrask.github.io/2015/07/12/basic-python-network/>

Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 67 13 Jan 2016

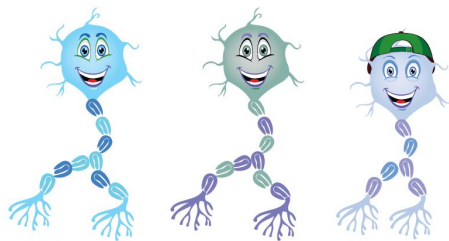
Assignment: Writing 2layer Net Stage your forward/backward computation!

```
# receive W1,W2,b1,b2 (weights/biases), X (data)
# forward pass:
h1 = #... function of X,W1,b1
scores = #... function of h1,W2,b2
loss = #... (several lines of code to evaluate Softmax loss)
# backward pass:
dscores = #...
dh1,dW2,db2 = #...
dW1,db1 = #...
```

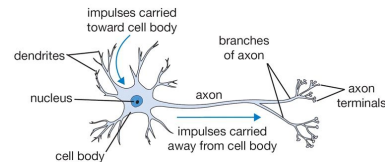
Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 68 13 Jan 2016



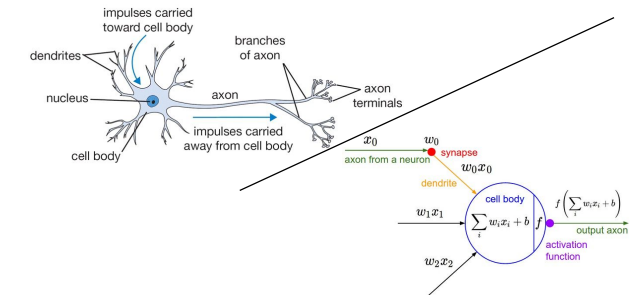
Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 69 13 Jan 2016



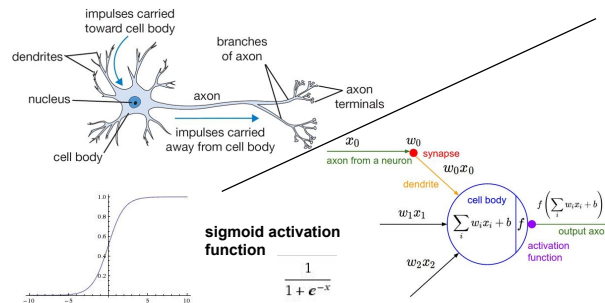
Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 70 13 Jan 2016



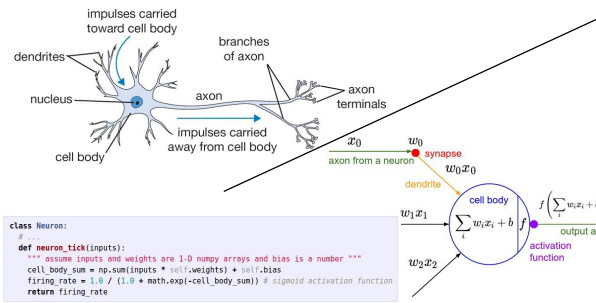
Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 71 13 Jan 2016



Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 72 13 Jan 2016



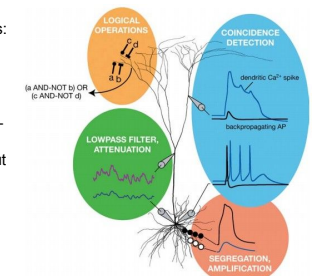
Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 73 13 Jan 2016



Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 74 13 Jan 2016

Be very careful with your Brain analogies:

- Biological Neurons:**
- Many different types
 - Dendrites can perform complex non-linear computations
 - Synapses are not a single weight but a complex non-linear dynamical system
 - Rate code may not be adequate



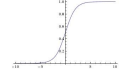
[Dendritic Computation, London and Hausser]

Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 75 13 Jan 2016

Activation Functions

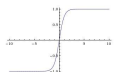
Sigmoid

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



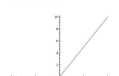
tanh

$$\tanh(x)$$



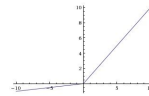
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$



Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

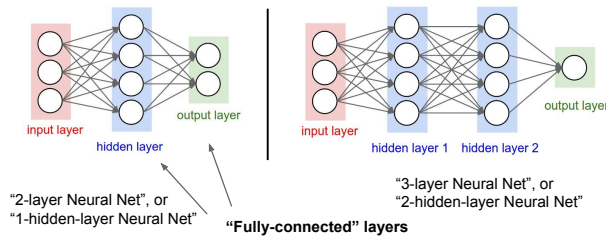
ELU

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$



Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 76 13 Jan 2016

Neural Networks: Architectures



Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 77 13 Jan 2016

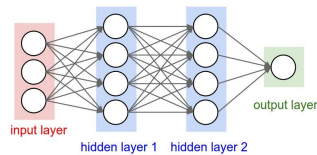
Example Feed-forward computation of a Neural Network

```
class Neuron:
    def __init__(self, weights, bias):
        self.weights = weights
        self.bias = bias
    def neuron_tick(inputs):
        """ assume inputs and weights are 1-D numpy arrays and bias is a number """
        cell_body_sum = np.sum(inputs * self.weights) + self.bias
        firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid activation function
        return firing_rate
```

We can efficiently evaluate an entire layer of neurons.

Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 78 13 Jan 2016

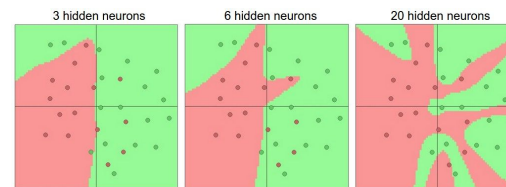
Example Feed-forward computation of a Neural Network



```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 79 13 Jan 2016

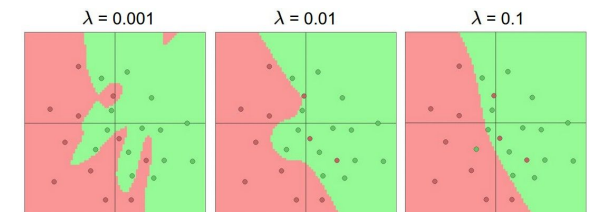
Setting the number of layers and their sizes



more neurons = more capacity

Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 80 13 Jan 2016

Do not use size of neural network as a regularizer. Use stronger regularization instead:



(you can play with this demo over at ConvNetJS: <http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html>)

Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 81 13 Jan 2016

Summary

- we arrange neurons into fully-connected layers
- the abstraction of a **layer** has the nice property that it allows us to use efficient vectorized code (e.g. matrix multiplies)
- neural networks are not really *neural*
- neural networks: bigger = better (but might have to regularize more strongly)

Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 82 13 Jan 2016