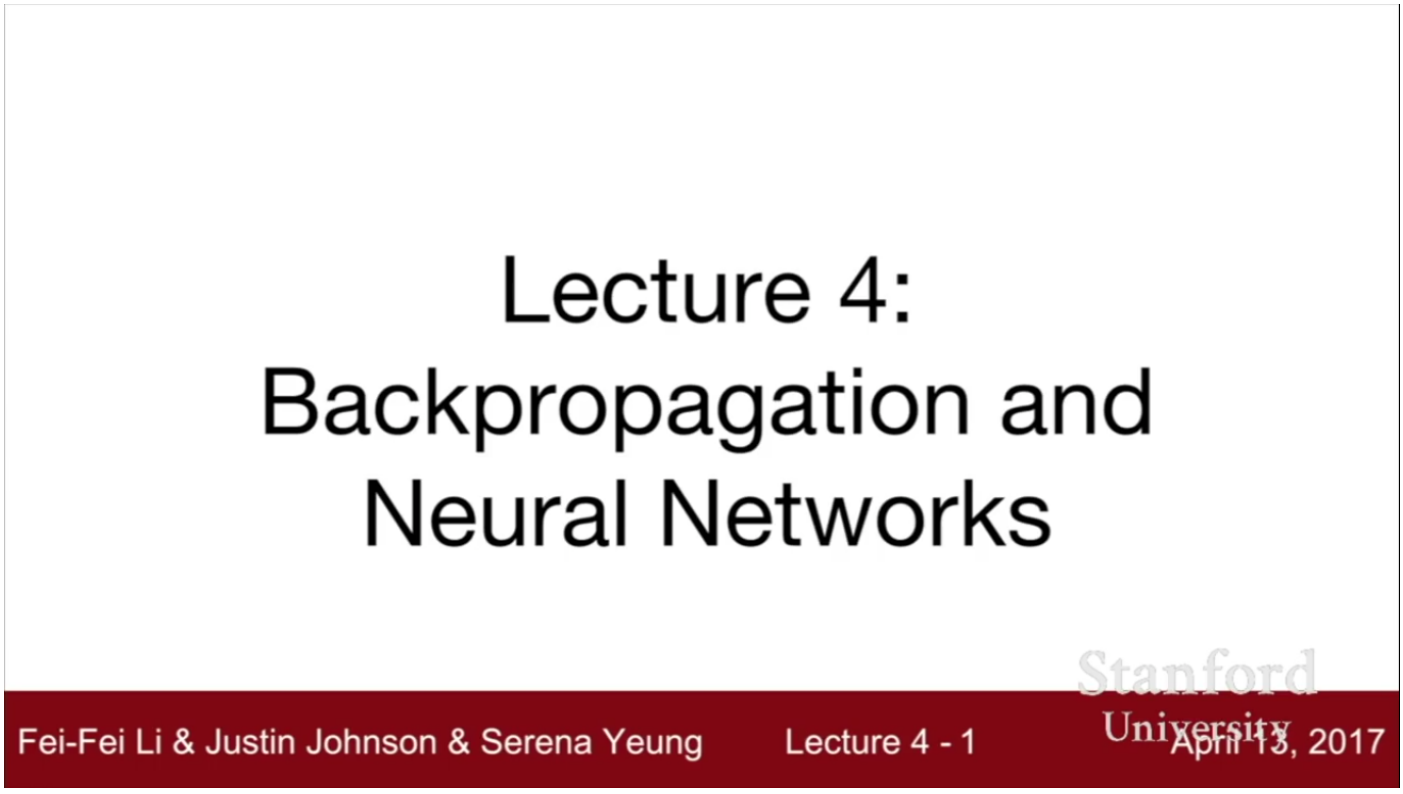
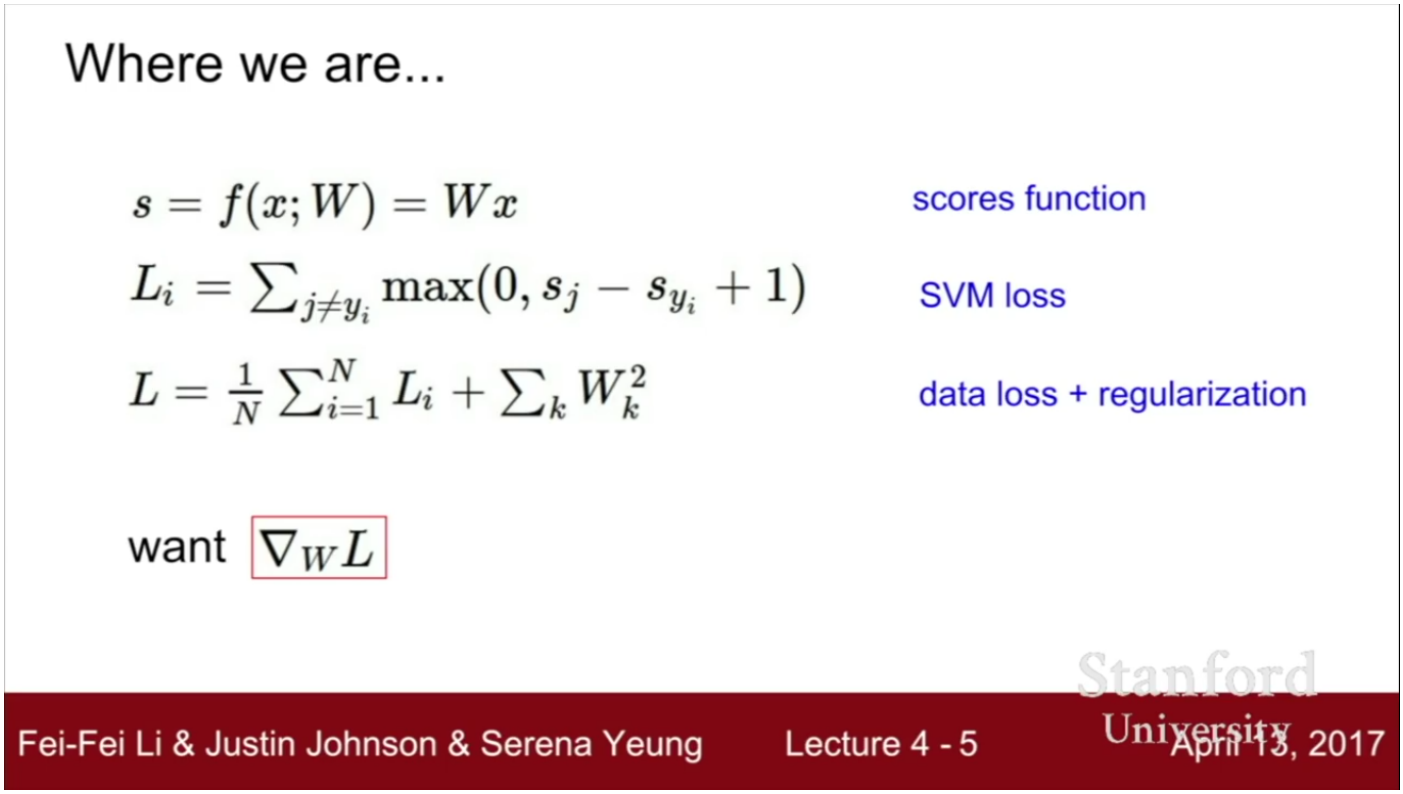
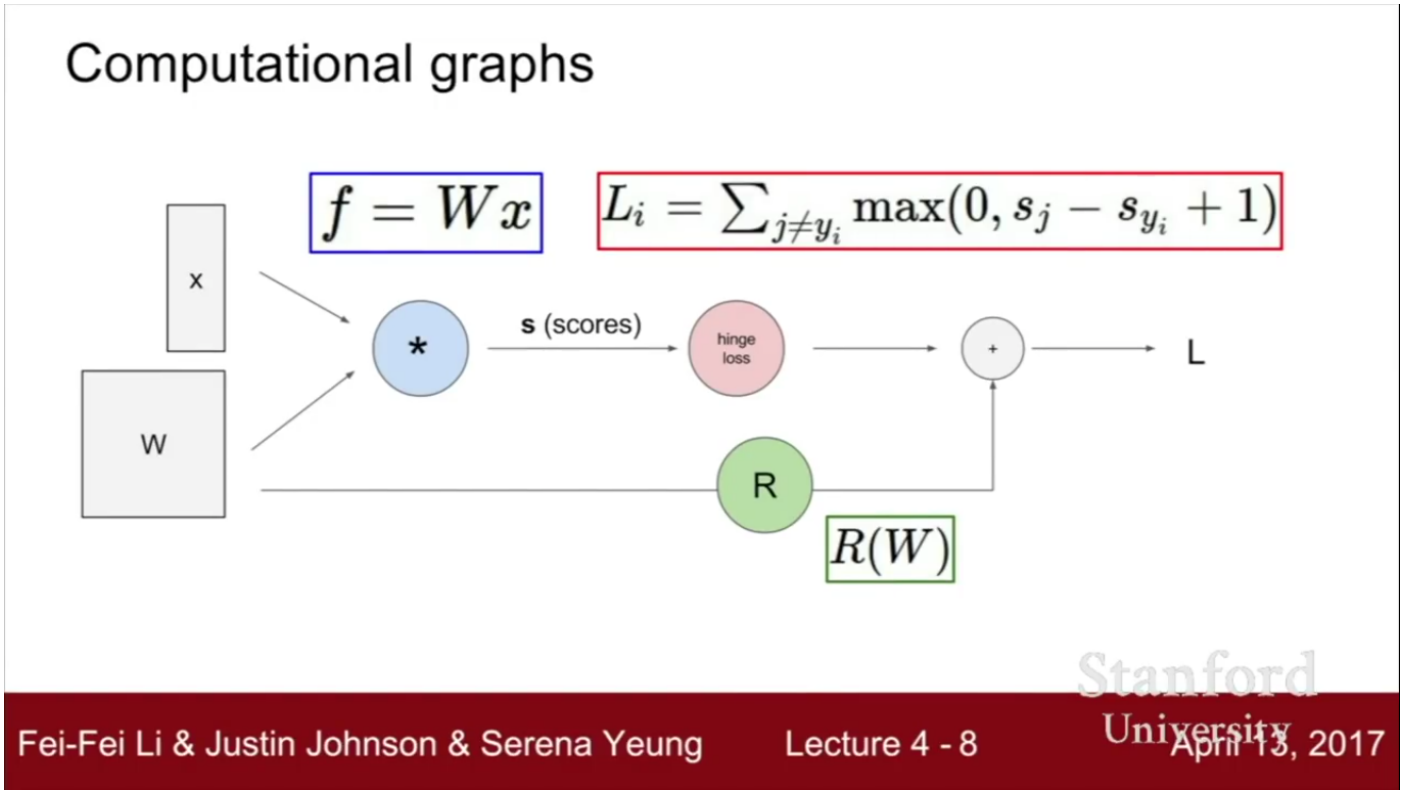
lec4

Creation Date: 31/12/2019 21:53
Last Updated: 03/01/2020 12:53

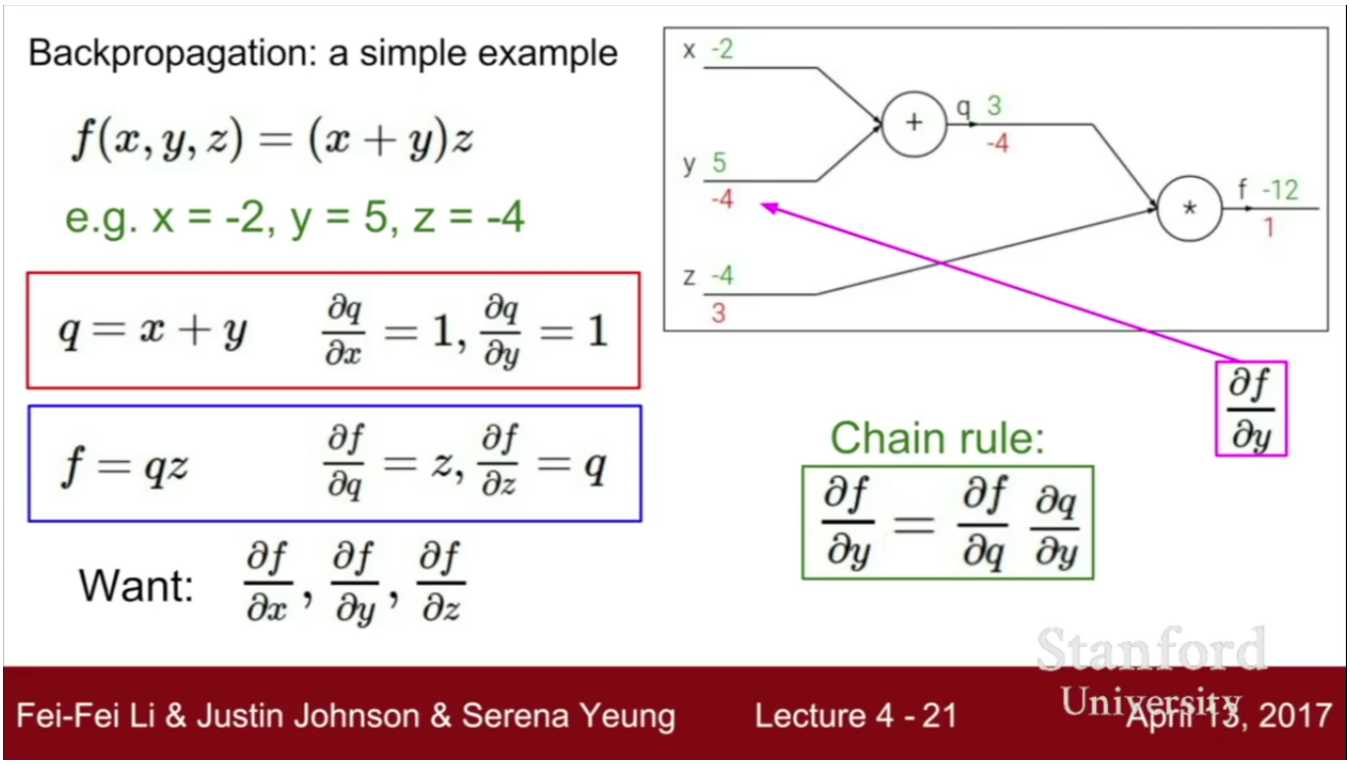
Lec 4: Backpropagation and Neural Networks



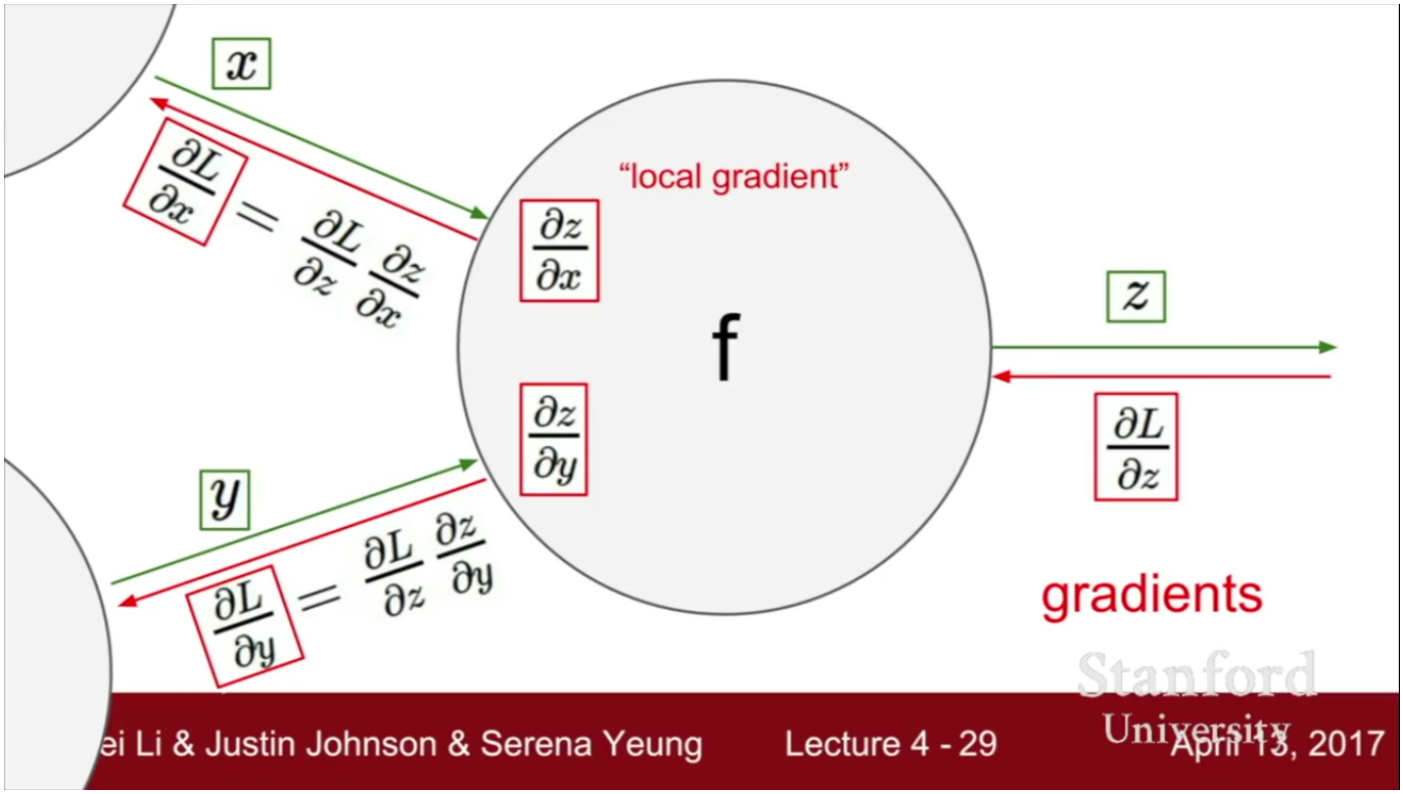




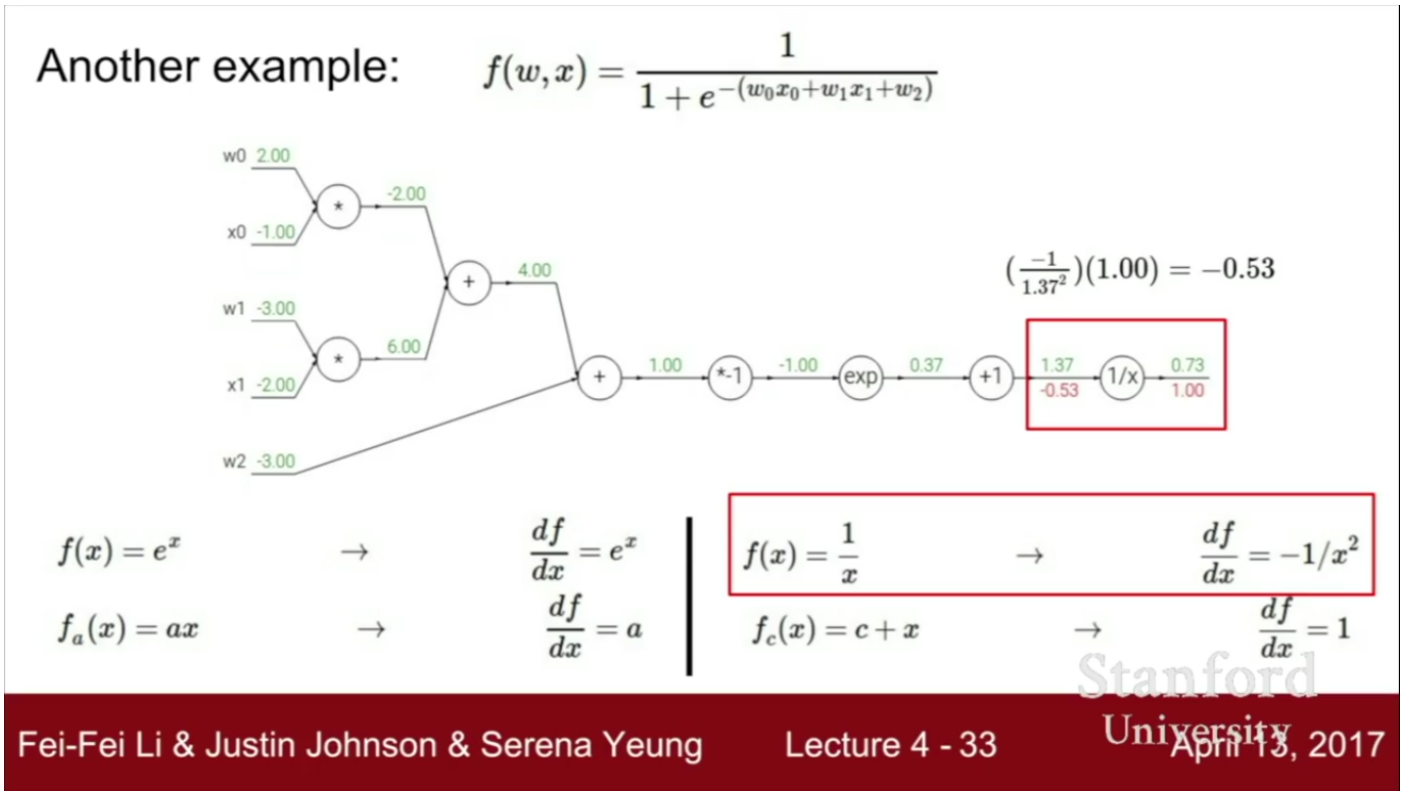
- We use computational graphs to compute the gradients needed.
- It helps in backpropagation using chain rule.



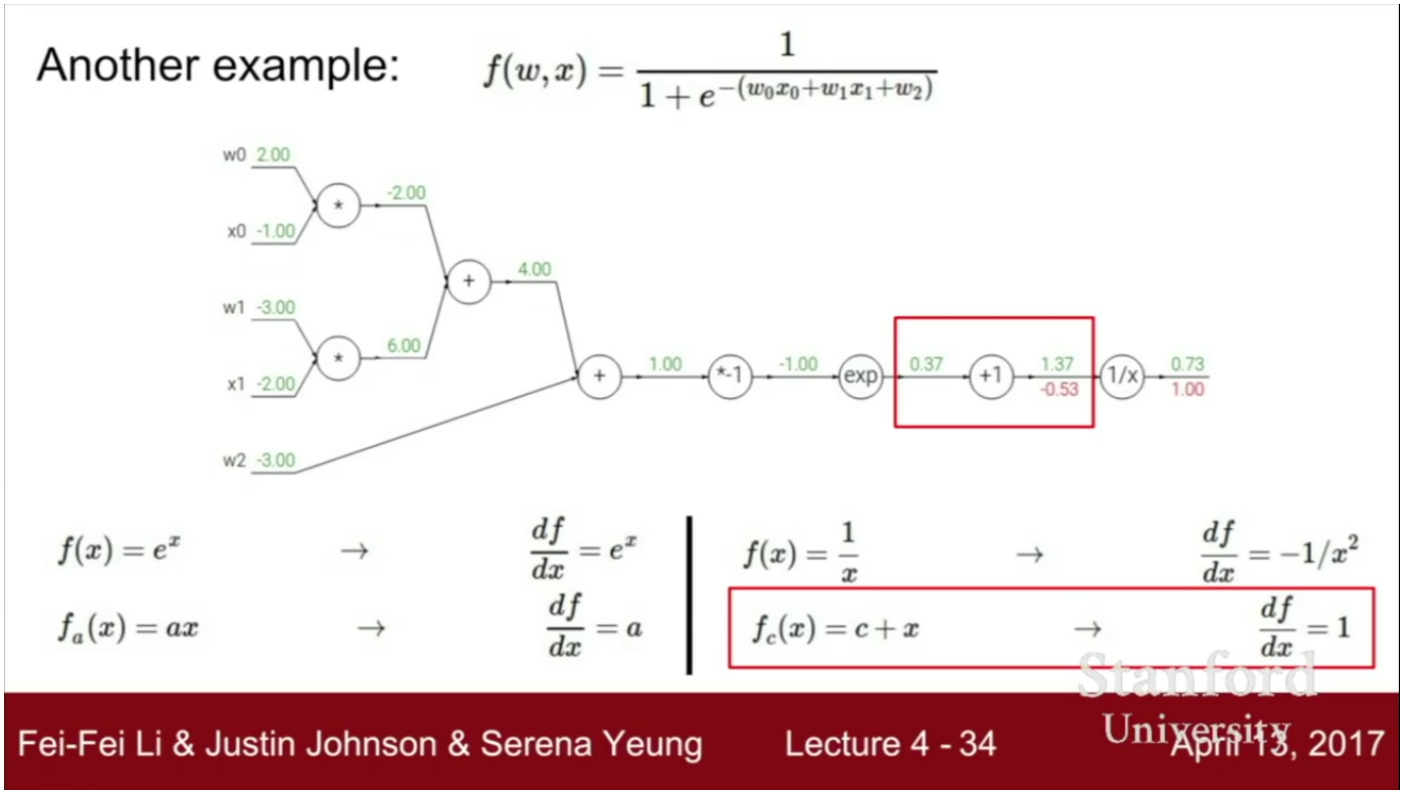
- Chain rules helps in backpropagation. We can see above that f depends on y, but we have an intermediate node q. so to find derivative of f w.r.t to y, we first find the derivative of f with q, and then multiply it with the derivative of q with y [shown in green box above].
- **We dont want to derive the derivative at each node, Because it gets simplified if we use chain rule. We just need to derive the derivative at the node.**



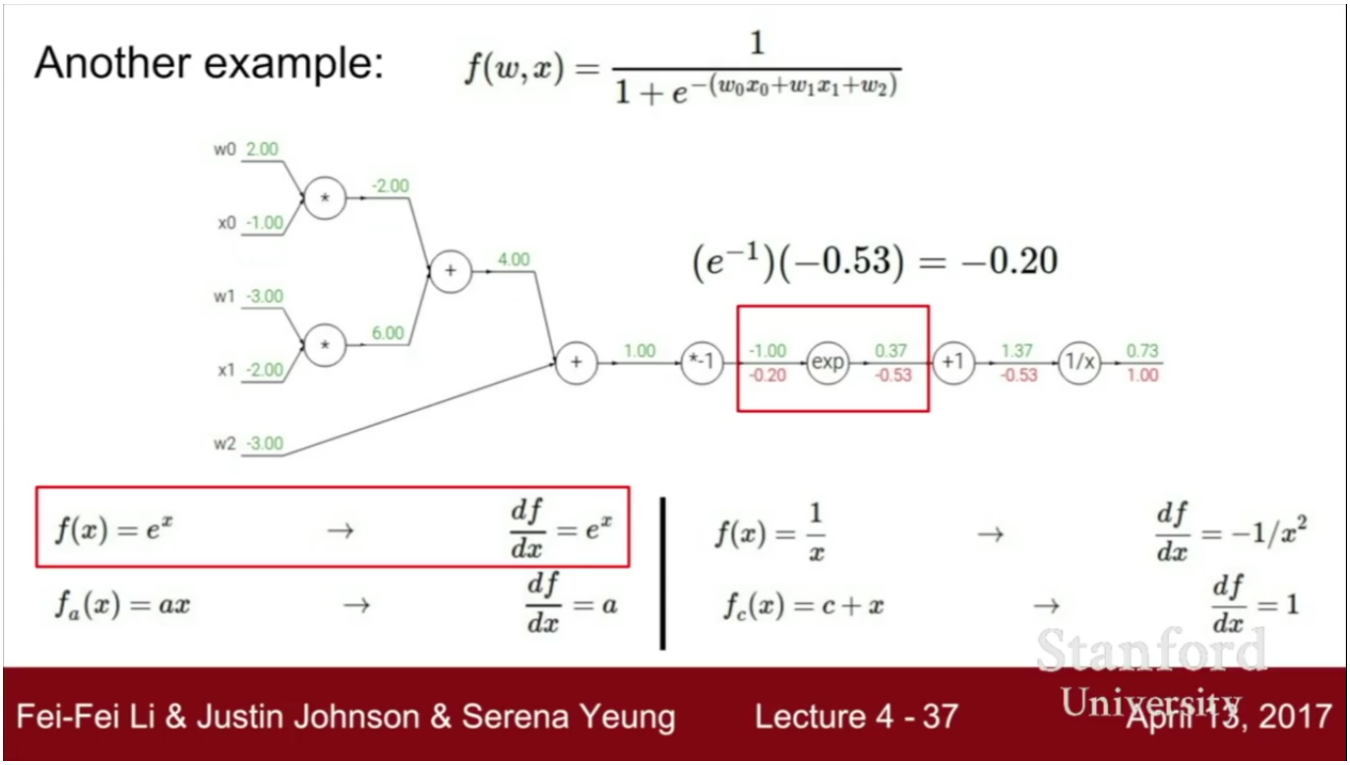
- At each node we get the derivative from the upstream(from output back to the previous node during backpropagation). We just have to multiply the received upstream derivative with the derivative of the current node w.r.t its available input. Here the f has the upstream of dL/dz, and to compute the gradient of dL/dx we just have to locally compute dz/dx and dz/dy and multiply it with the upstream derivtive dL/dz.
- The derivative dz/dx and dz/dy are usually simple in nature like multiplication and addition operations.



- Here $L = f(z) = z$. So $dL/dz = dz/dz = 1$. So when we compute the gradient at $1/x$ node. We have to compute dL/dx which will be $dL/dx = (dL/dz) * (dz/dx)$, where $dL/dz = 1$ and dz/dx is to compute the derivative of $1/x^2$.

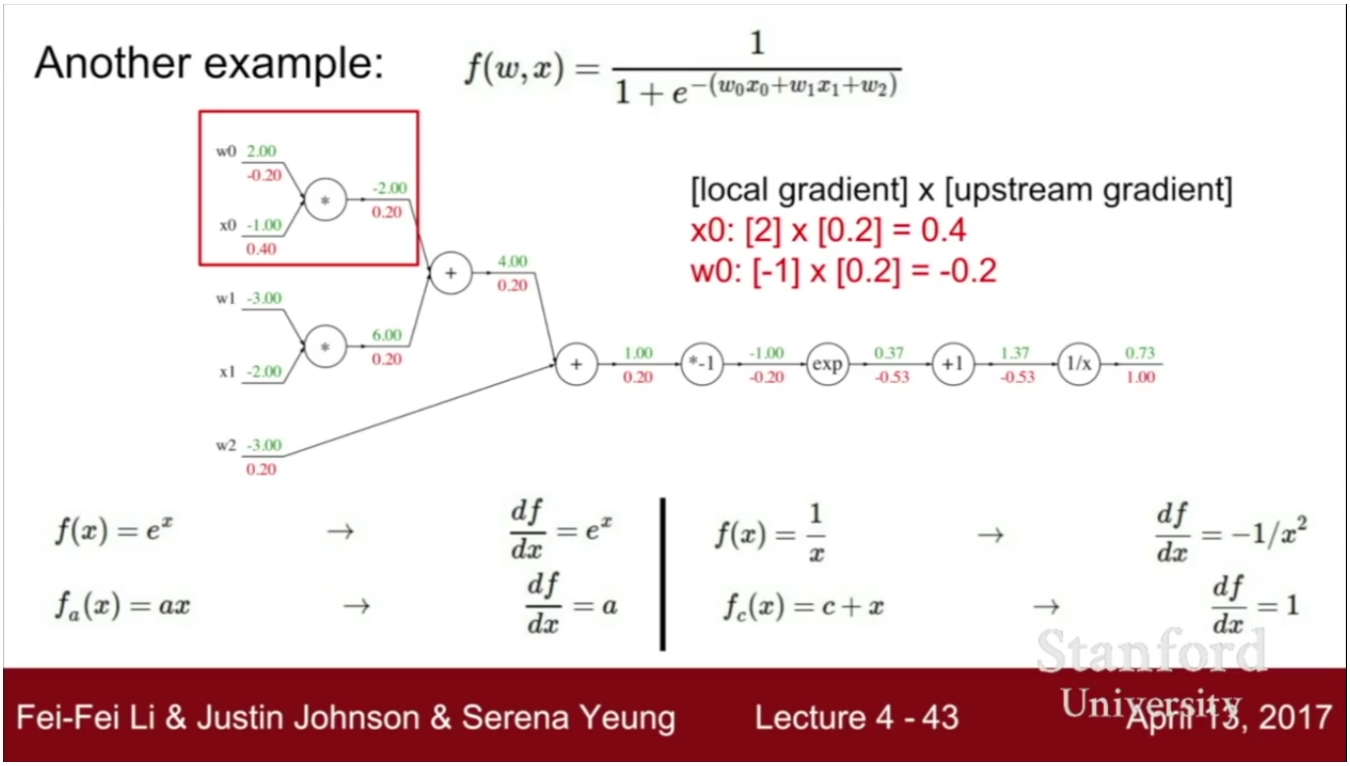


- For this above node, the derivative will be the product of the upstream gradient i.e. 0.53 which will get multiplied with the local gradient of 1 (Since its the function of $a + x$). So the gradient at this node will be $1 * 0.53$.

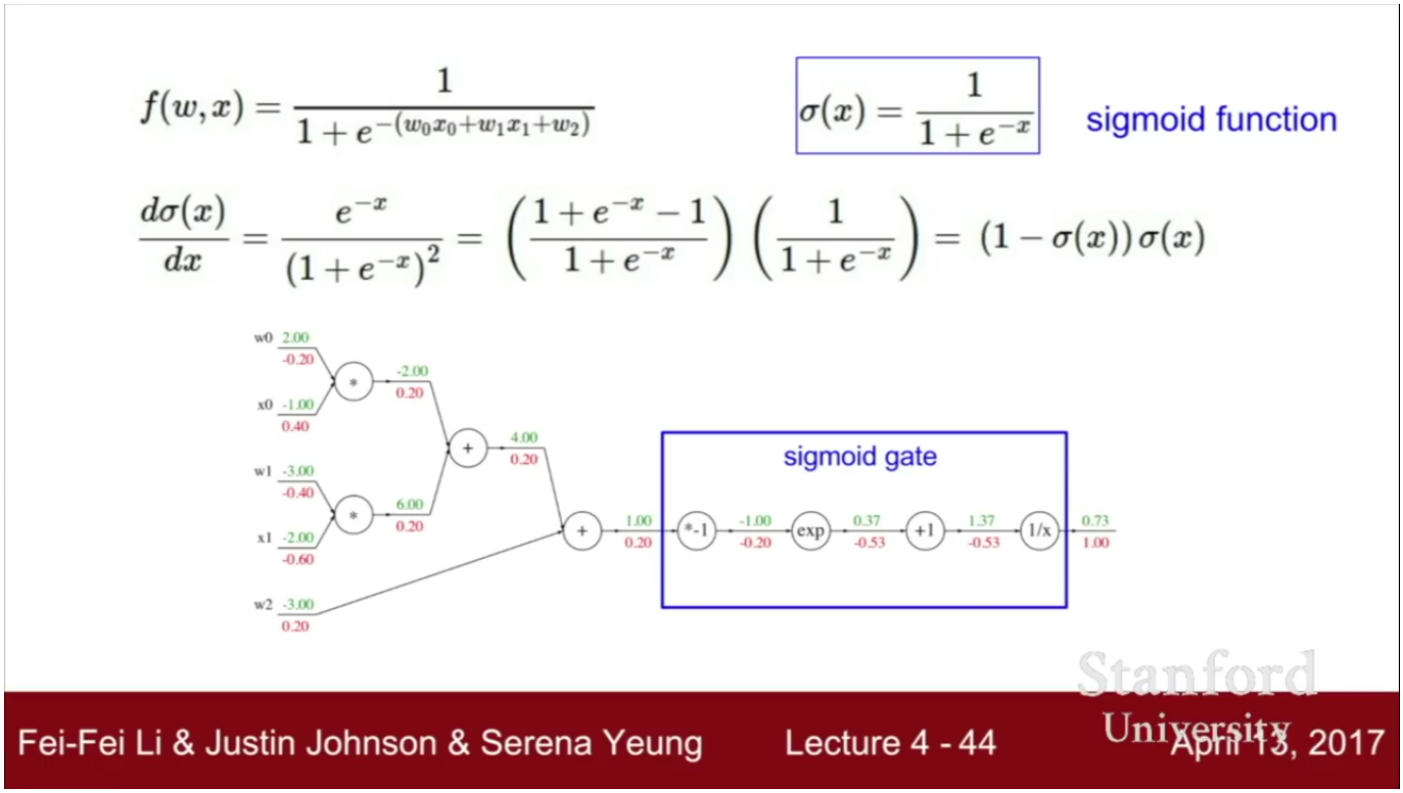


- For the node with the exponential, it's the same case. We take the upstream gradient of 0.53 and multiply it with the local gradient, which is e^x, Since the input is -1, we substitute the value -1 for

e^x.



- Similarly we find the gradient w.r.t the inputs w_0 etc.

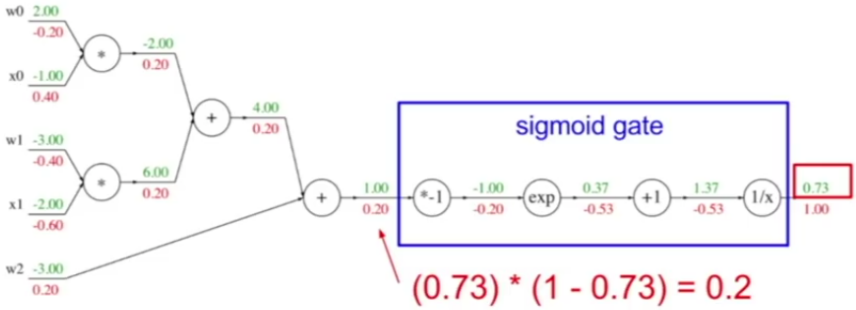


- **We can break the computational graph into any granularity we need. We can either expand the sigmoid function into constituent nodes or just use it as a single node. Since we have a simple derivative for the sigmoid.**
- As long as we can express the local gradient easily we can find the derivative at the given node. This is the case of sigmoid node, it's easier to find the local gradient.

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

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

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



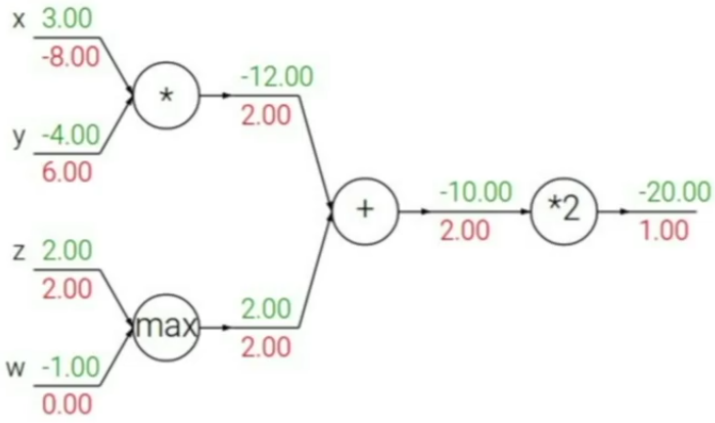
- We can see that the output matches if we either use the sigmoid node or break it into its constiuent intermediate nodes.

Patterns in backward flow

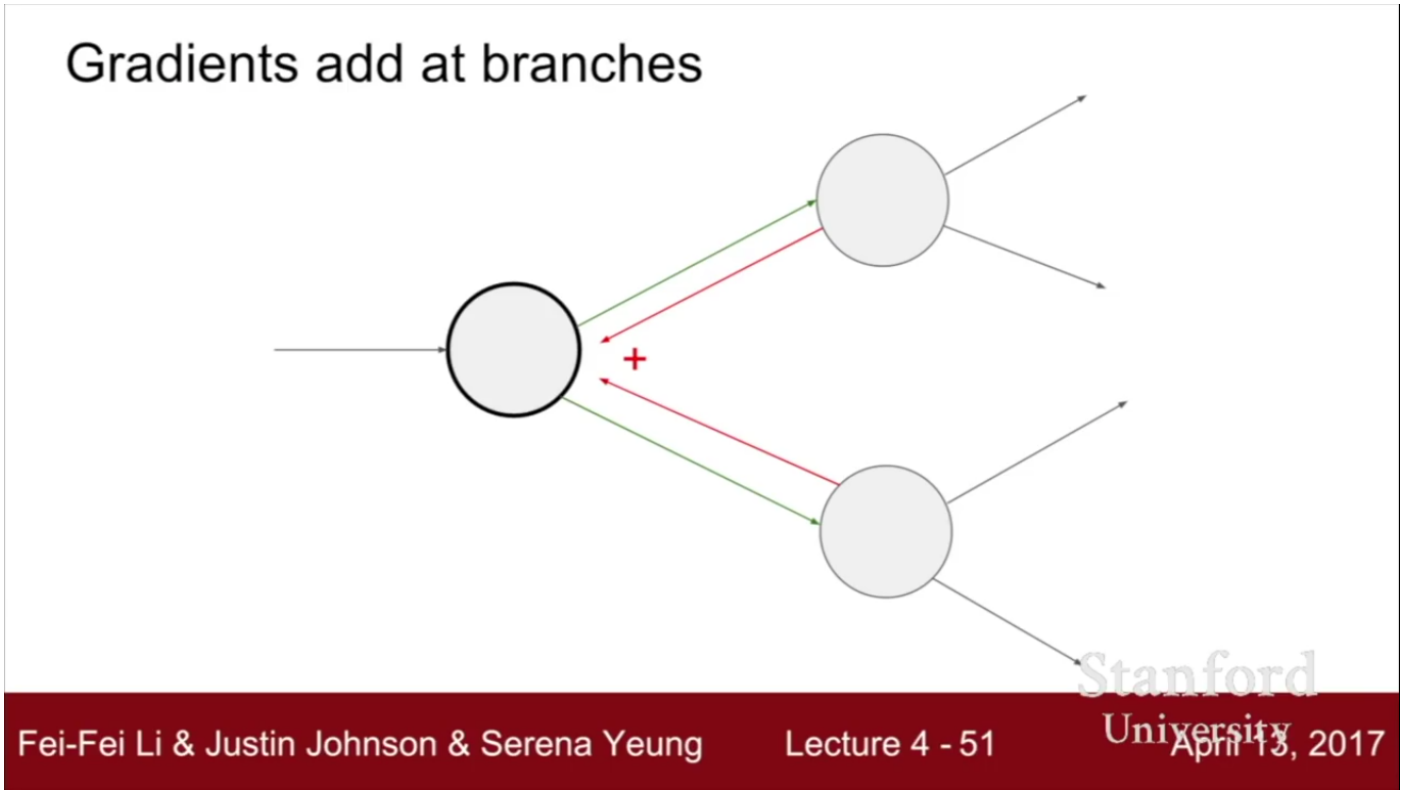
add gate: gradient distributor

max gate: gradient router

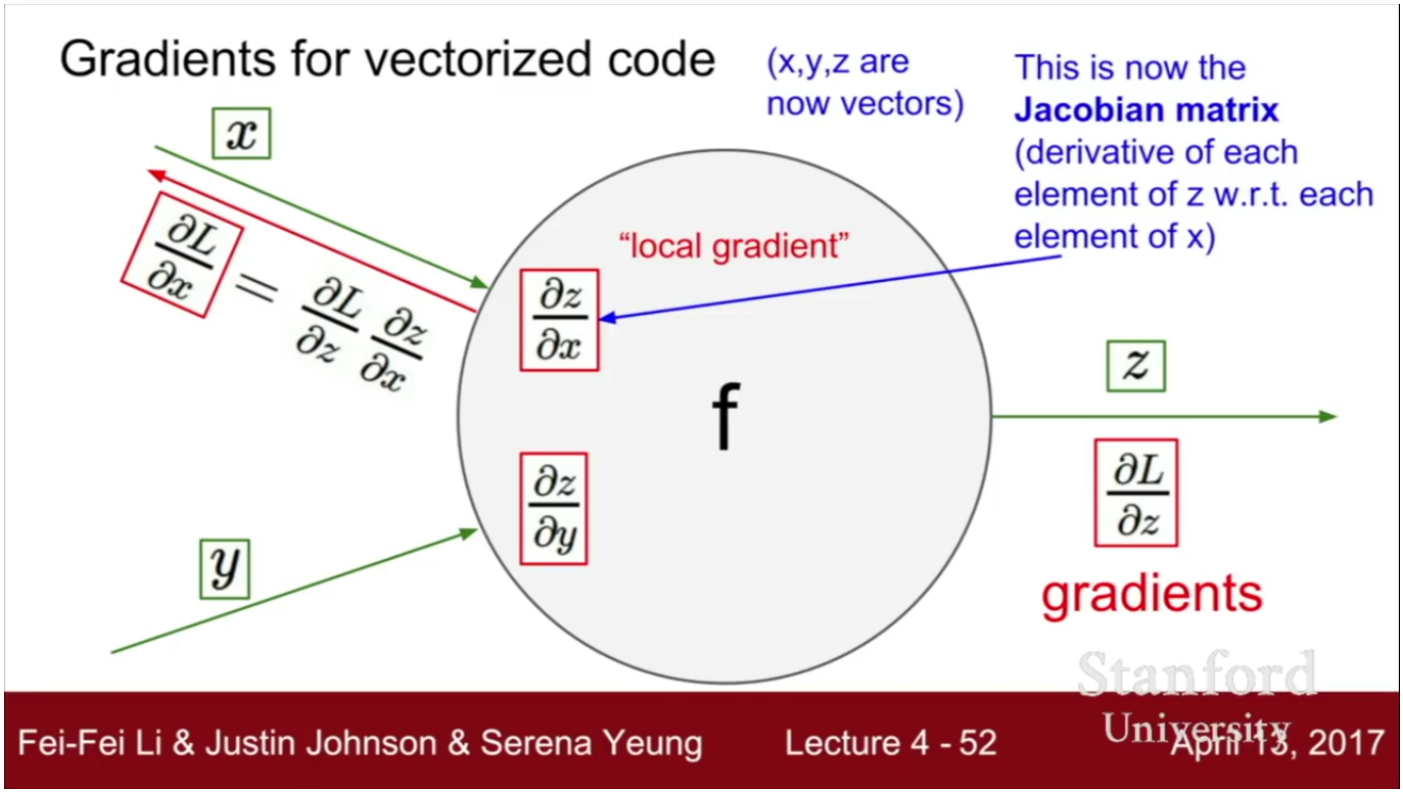
mul gate: gradient switcher



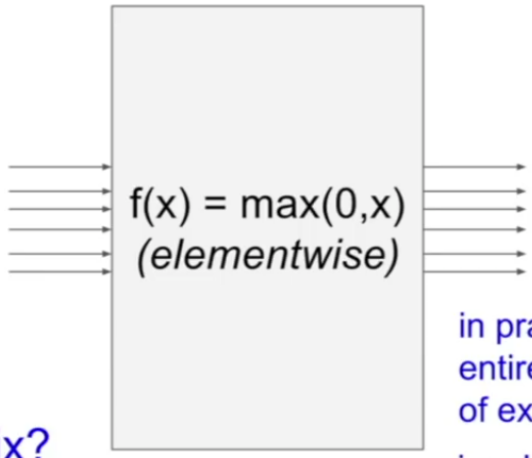
- The max gate will just route the gradient to which it came from (the value which was chosen). **Because during the forward pass only the maximum value was passed through the network, so during the backward pass the gradient should only affect that.**
- The mul gate is gradient switcher because only the other variable value remains after differentiation and the local gradient gets scaled by that and gets back-propagated.



- If a node distributes to multiple nodes then during backward pass, the gradients get added.



Vectorized operations



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

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

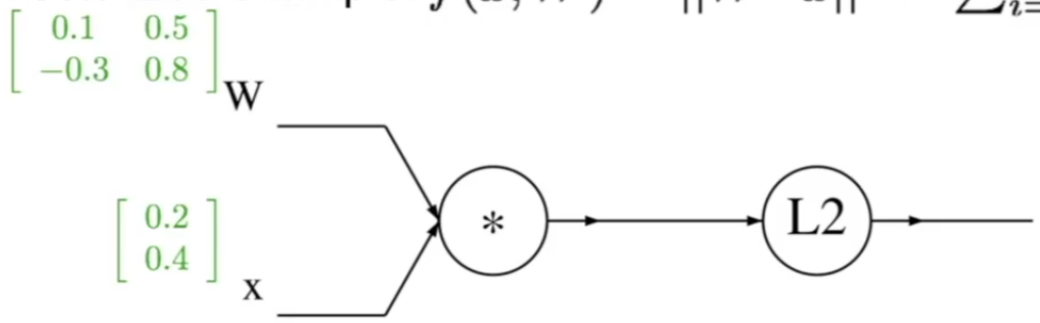
i.e. Jacobian would technically be a [409,600 x 409,600] matrix :\

- In practice we dont have to compute the Jacobian for such huge matrices. Since the element wise operation with the parameters gives rise to **DIAGONAL** matrix.

A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^{n} (W \cdot x)_i^2$

$\in \mathbb{R}^n$ $\in \mathbb{R}^{n \times n}$

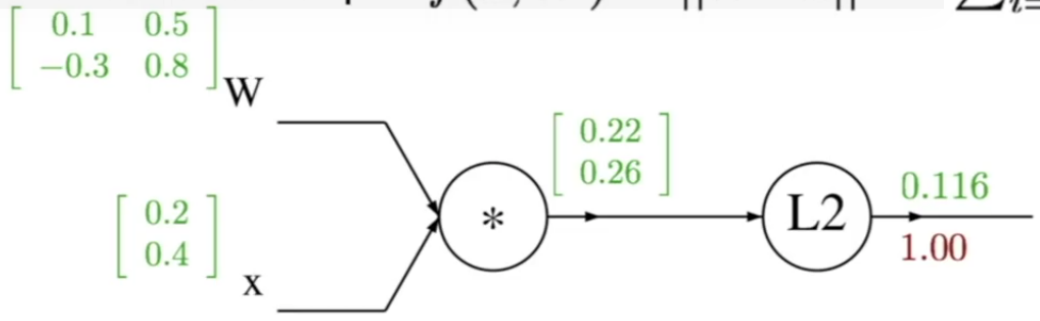
A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^{n} (W \cdot x)_i^2$



$$q = W \cdot x = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$f(q) = ||q||^2 = q_1^2 + \cdots + q_n^2$$

A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^{n} (W \cdot x)_i^2$



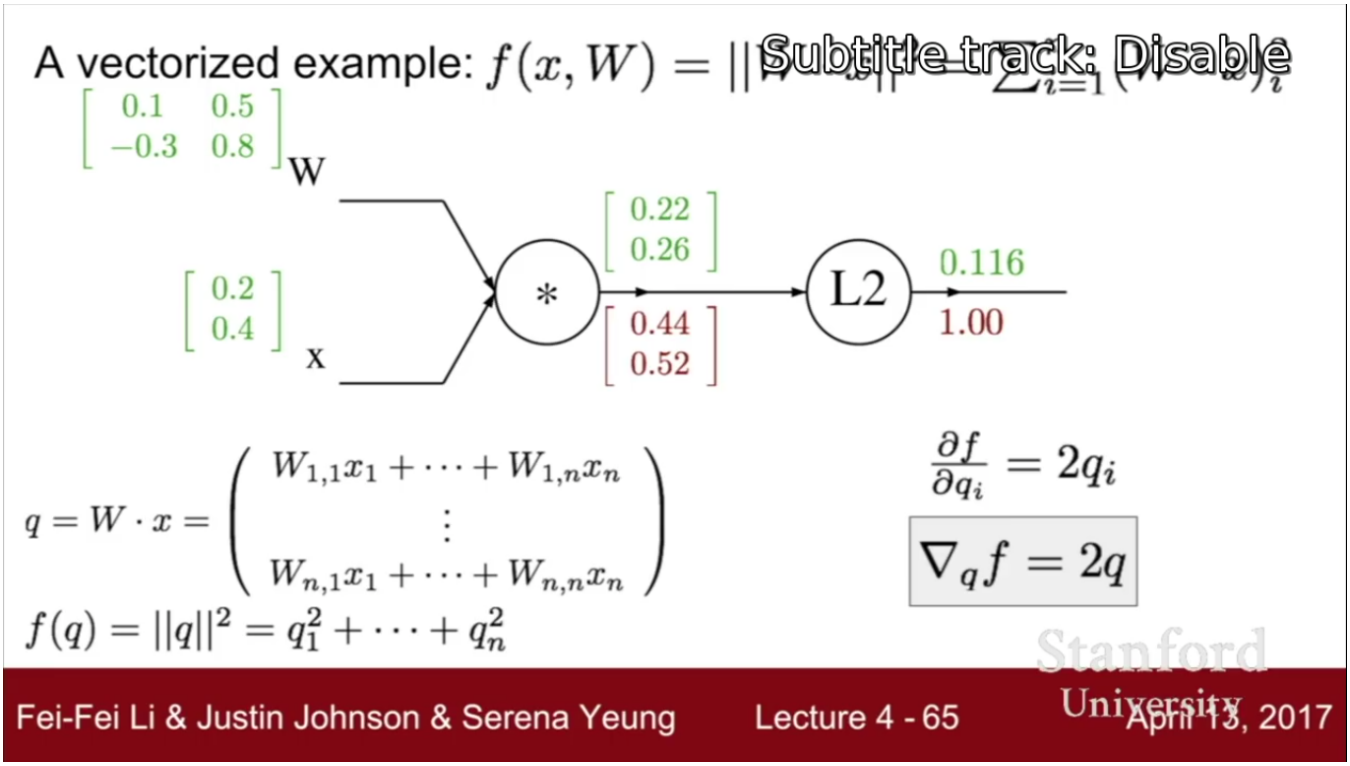
$$q = W \cdot x = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$f(q) = ||q||^2 = q_1^2 + \cdots + q_n^2$$

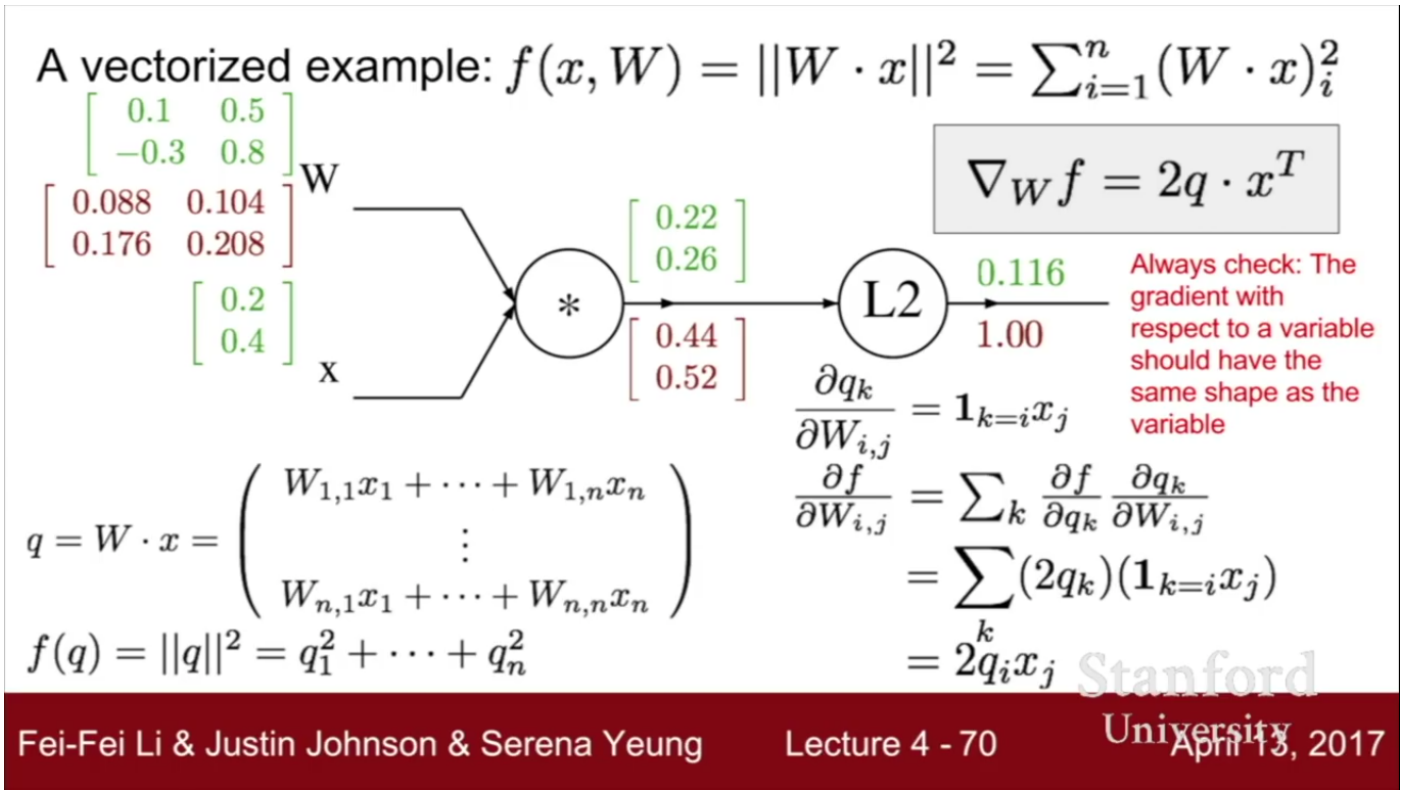
$$\frac{\partial f}{\partial q_i} = 2q_i$$

$$\boxed{\nabla_q f = 2q}$$

- The gradient of the last node will be 1.

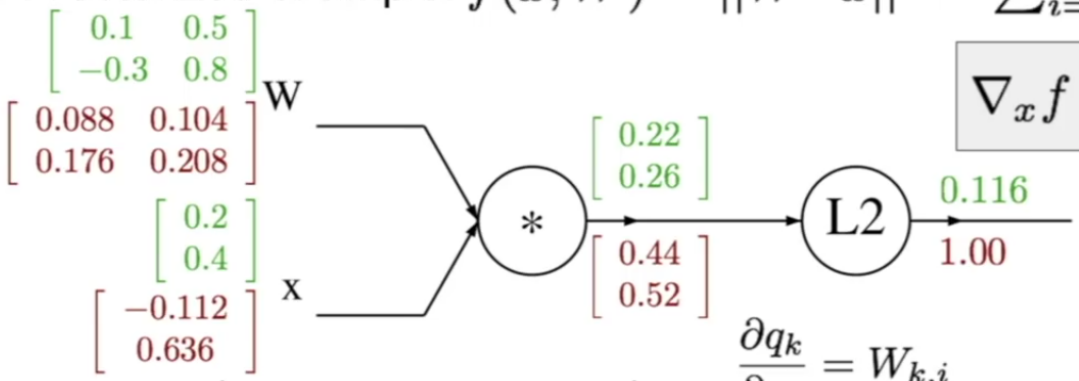


- Then previous node to that has the gradient of $2q$ so its values are also mentioned there as $[0.44, 0.52]$.



A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^{n} (W \cdot x)_i^2$

$$\nabla_x f = 2W^T \cdot q$$



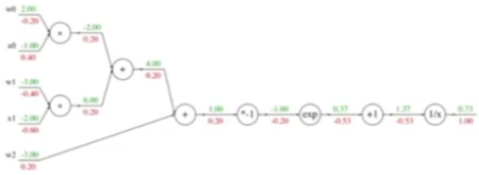
$$q = W \cdot x = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$f(q) = ||q||^2 = q_1^2 + \cdots + q_n^2$$

$$\frac{\partial q_k}{\partial x_i} = W_{k,i}$$

$$\frac{\partial f}{\partial x_i} = \sum_k \frac{\partial f}{\partial q_k}\frac{\partial q_k}{\partial x_i} = \sum_k 2q_k W_{k,i}$$

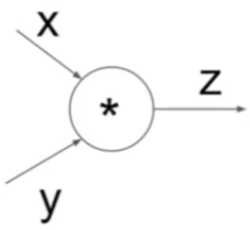
Modularized implementation: forward / backward API



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
```

Modularized implementation: forward / backward API



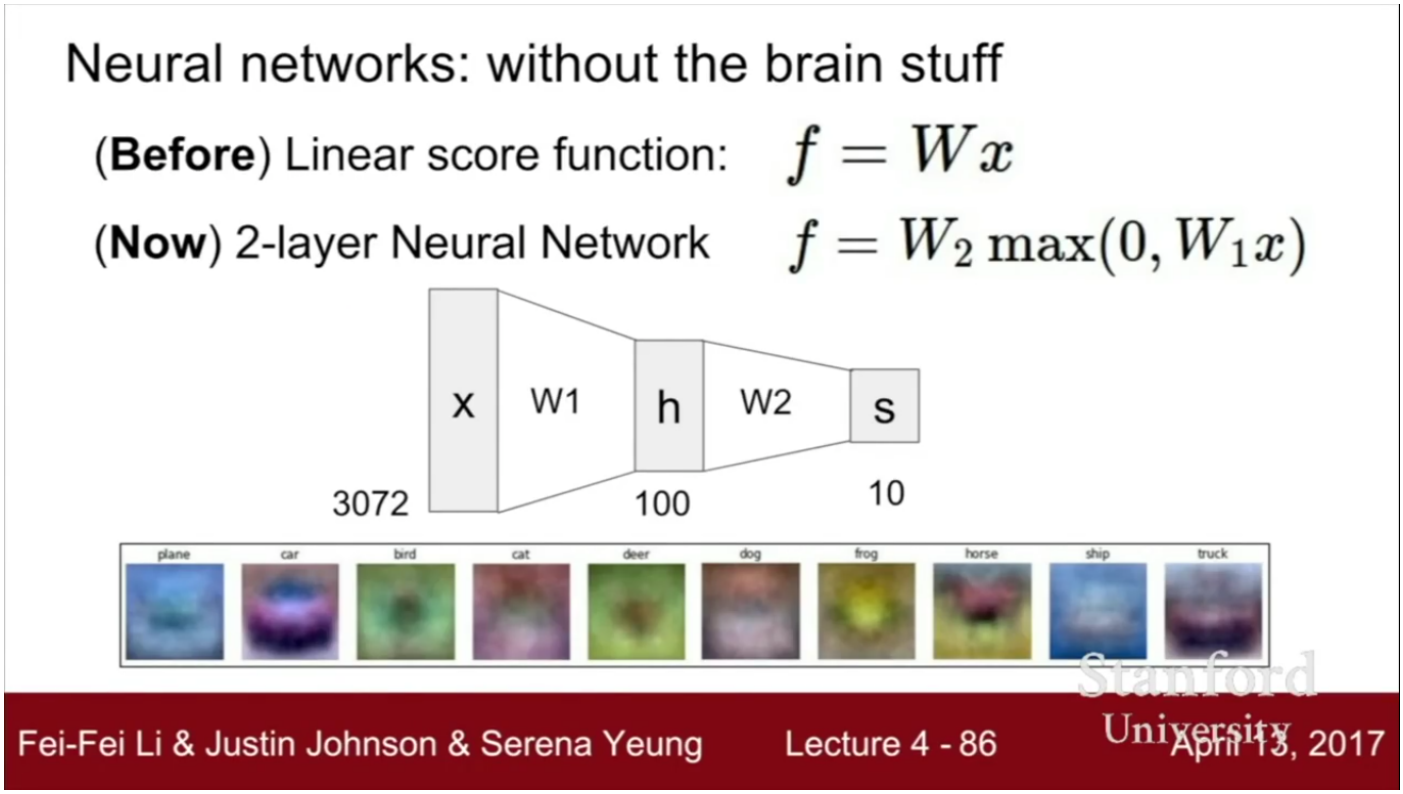
(x,y,z are scalars)

```
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]
```

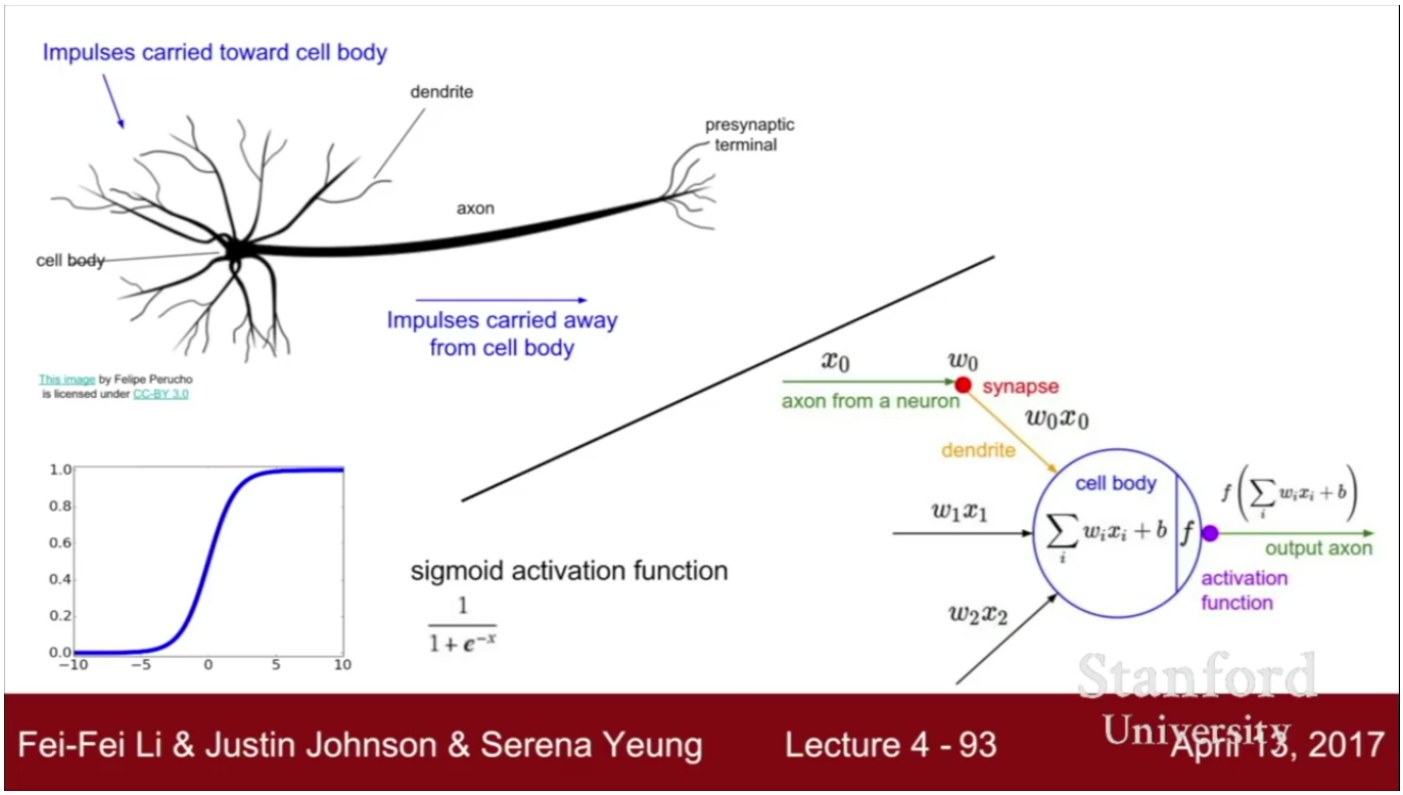

Summary so far...

- neural nets will be very large: impractical to write 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 Networks.



- In Neural Networks we stack up these combination of Linear Function and activation layers.
- In just Linear Function, the matrix W described a template (each row of W). Since we had only 1 Linear Function as the hypothesis, the template was fixed(n number of rows of W means we have n templates).
- In NN, since we have layers of Linear Functions, we are performing a weighted combination of these templates, hence we are learning more complex templates by combining various other templates.



- The sigmoid and ReLU functions mimics the activation of a real neuron.

Be very careful with your brain analogies!

Biological Neurons:

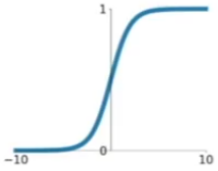
- Many different types
- Dendrites can perform complex non-linear computations
- Synapses are not a single weight but a complex non-linear dynamical system
- Rate code may not be adequate

[Dendritic Computation. London and Hausser]

Activation functions

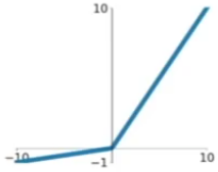
Sigmoid

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



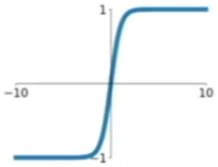
Leaky ReLU

$\max(0.1x, x)$



tanh

$\tanh(x)$

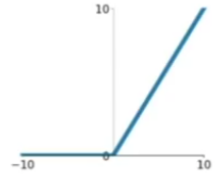


Maxout

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

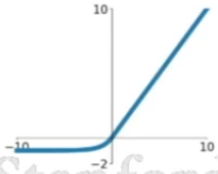
ReLU

$\max(0, x)$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



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*
- Next time: Convolutional Neural Networks