Lecture 4:
Backpropagation
and
Neural Networks part 1
Administrative

A1 is due Jan 20 (Wednesday). ~150 hours left
Warning: Jan 18 (Monday) is Holiday (no class/office hours)

Also note:
Lectures are non-exhaustive.
Read course notes for completeness.

I’ll hold make up office hours on Wed Jan20, 5pm @ Gates 259
Where we are...

\[ s = f(x; W) = Wx \]  

scores function

\[ L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \]  

SVM loss

\[ L = \frac{1}{N} \sum_{i=1}^{N} L_i + \sum_k W_k^2 \]  

data loss + regularization

want  \[ \nabla_W L \]
Optimization

# Vanilla Gradient Descent

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

\[ \frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \]

**Numerical gradient:** slow :(', approximate :, easy to write :)

**Analytic gradient:** fast :), exact :), error-prone :(.

In practice: Derive analytic gradient, check your implementation with numerical gradient.
\[ f = Wx \]

\[ L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \]
Convolutional Network
(AlexNet)

- input image
- weights
- loss
Neural Turing Machine
\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, \ y = 5, \ z = -4 \)
\[ f(x, y, z) = (x + y)z \]
e.g. \( x = -2, y = 5, z = -4 \)

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

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

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: \( \frac{\partial f}{\partial x}, \ \frac{\partial f}{\partial y}, \ \frac{\partial f}{\partial z} \)
\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, \ y = 5, \ z = -4 \)

\[
\begin{align*}
q &= x + y & \frac{\partial q}{\partial x} &= 1, \quad \frac{\partial q}{\partial y} &= 1 \\
\end{align*}
\]

\[
\begin{align*}
f &= qz & \frac{\partial f}{\partial q} &= z, \quad \frac{\partial f}{\partial z} &= q \\
\end{align*}
\]

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. \( x = -2, y = 5, z = -4 \)

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

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

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. \( x = -2, y = 5, z = -4 \)

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

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

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:** \( \frac{\partial f}{\partial x}, \ \frac{\partial f}{\partial y}, \ \frac{\partial f}{\partial z} \)
 \[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

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

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

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. \( x = -2, y = 5, z = -4 \)

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

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

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

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

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

Want: \[ \frac{\partial f}{\partial x}, \quad \frac{\partial f}{\partial y}, \quad \frac{\partial f}{\partial z} \]
\[ f(x, y, z) = (x + y)z \]
e.g. \( x = -2, y = 5, z = -4 \)

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

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

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. \( x = -2, y = 5, z = -4 \)

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

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

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

Chain rule:
\[
\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial x}
\]
activations

\[ f(x, y) \rightarrow z \]
activations

\( \frac{\partial z}{\partial x} \)

\( \frac{\partial z}{\partial y} \)

\( x \)

\( f \)

\( y \)

\( z \)

“local gradient”
activations

\[
\frac{\partial z}{\partial x}
\]

\[
\frac{\partial z}{\partial y}
\]

"local gradient"

\[
\frac{\partial L}{\partial z}
\]

gradients
activations

\[ \frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial x} \]

"local gradient"

\[ \frac{\partial z}{\partial x} \]

\[ \frac{\partial L}{\partial z} \]

gradients

Fei-Fei Li & Andrej Karpathy & Justin Johnson

Lecture 4 - 25

13 Jan 2016
The diagram illustrates the relationship between activations and gradients in a neural network. The equation $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x}$ shows how the gradient of the loss $L$ with respect to the input $x$ is decomposed into the gradient of the loss with respect to the intermediate variable $z$ and the gradient of $z$ with respect to $x$. Similarly, $\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial y}$ shows the relationship for the gradient with respect to $y$. The term "local gradient" highlights the focus on gradients around the intermediate variable $z$. The diagram helps visualize the flow of information and gradient computation in neural networks.
activations

\[ \frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x} \]

\[ \frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial y} \]

“local gradient”

gradients

\[ \frac{\partial L}{\partial z} \]
Another example: 

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

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

\[ f(x) = e^x \quad \Rightarrow \quad \frac{df}{dx} = e^x \]
\[ f_a (x) = ax \quad \Rightarrow \quad \frac{df}{dx} = a \]
\[ f_c(x) = c + x \quad \Rightarrow \quad \frac{df}{dx} = 1 \]
Another example:

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

\[
\begin{align*}
  f(x) &= e^x & \rightarrow & \quad \frac{df}{dx} = e^x \\
  f_a(x) &= ax & \rightarrow & \quad \frac{df}{dx} = a \\
  f_c(x) &= c + x & \rightarrow & \quad \frac{df}{dx} = 1 \\
  f(x) &= \frac{1}{x} & \rightarrow & \quad \frac{df}{dx} = -\frac{1}{x^2}
\end{align*}
\]
Another example:

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

\[
\begin{align*}
\left(-\frac{1}{1.37^2}\right)(1.00) &= -0.53 \\
\end{align*}
\]

\[
\begin{align*}
f(x) &= e^x \\
\frac{df}{dx} &= e^x \\
f_a(x) &= ax \\
\frac{df}{dx} &= a \\
f_c(x) &= c + x \\
\frac{df}{dx} &= 1 \\
\end{align*}
\]
Another example:

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

\[
\begin{align*}
  f(x) &= e^x & \rightarrow & & \frac{df}{dx} &= e^x \\
  f_a(x) &= ax & \rightarrow & & \frac{df}{dx} &= a \\
  f_c(x) &= c + x & \rightarrow & & \frac{df}{dx} &= 1
\end{align*}
\]

\[
\begin{align*}
  f(x) &= \frac{1}{x} & \rightarrow & & \frac{df}{dx} &= -\frac{1}{x^2}
\end{align*}
\]
Another example:

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

\[ (1)(-0.53) = -0.53 \]

\[
\begin{align*}
 f(x) &= e^x \\
 f_a(x) &= ax
\end{align*}
\]

\[
\begin{align*}
 \frac{df}{dx} &= e^x \\
 \frac{df}{dx} &= a
\end{align*}
\]

\[
\begin{align*}
 f(x) &= \frac{1}{x} \\
 f_c(x) &= c + x
\end{align*}
\]

\[
\begin{align*}
 \frac{df}{dx} &= -\frac{1}{x^2} \\
 \frac{df}{dx} &= 1
\end{align*}
\]
Another example:

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

\[
\begin{align*}
  f(x) &= e^x & \Rightarrow & & \frac{df}{dx} &= e^x \\
  f_a(x) &= ax & \Rightarrow & & \frac{df}{dx} &= a \\
  f_c(x) &= c + x & \Rightarrow & & \frac{df}{dx} &= 1 \\
  f(x) &= \frac{1}{x} & \Rightarrow & & \frac{df}{dx} &= -\frac{1}{x^2}
\end{align*}
\]
Another example: 

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

\[
(e^{-1})(-0.53) = -0.20
\]

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

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

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

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

Fei-Fei Li & Andrej Karpathy & Justin Johnson
Another example:

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

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

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

\[ f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1 \]
Another example:

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

\[ (-1) \times (-0.20) = 0.20 \]

\[
\begin{align*}
  f(x) &= e^x &\Rightarrow& &\frac{df}{dx} &= e^x \\
  f_a(x) &= ax &\Rightarrow& &\frac{df}{dx} &= a \\
  f_c(x) &= c + x &\Rightarrow& &\frac{df}{dx} &= 1 \\
  f(x) &= \frac{1}{x} &\Rightarrow& &\frac{df}{dx} &= -1/x^2
\end{align*}
\]
Another example:

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

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

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

\[ f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1 \]
Another example:

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

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

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

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

\[
[\text{local gradient}] \times [\text{its gradient}]
\]

\[
[1] \times [0.2] = 0.2
\]

\[
[1] \times [0.2] = 0.2 \quad \text{(both inputs!)}
\]
Another example:

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

\[
\begin{align*}
f(x) &= e^x & \Rightarrow & & \frac{df}{dx} = e^x & & f(x) = \frac{1}{x} & \Rightarrow & & \frac{df}{dx} = -\frac{1}{x^2} \\
f_a(x) &= ax & \Rightarrow & & \frac{df}{dx} = a & & f_c(x) = c + x & \Rightarrow & & \frac{df}{dx} = 1
\end{align*}
\]
Another example:

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

[local gradient] \times [its gradient]

\[ x_0: [2] \times [0.2] = 0.4 \]
\[ w_0: [-1] \times [0.2] = -0.2 \]

\[ f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x \]
\[ f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a \]
\[ f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -\frac{1}{x^2} \]
\[ f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1 \]
\[ f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \]

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

sigmoid function

sigmoid gate
The sigmoid function is defined as:

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

For the derivative of the sigmoid function, we have:

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

The sigmoid gate is demonstrated in the diagram, where the output is calculated as:

\[(0.73) \times (1 - 0.73) = 0.2\]
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)

```python
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

\[ \text{class MultiplyGate(object):} \]
\[ \quad \text{def forward(x, y):} \]
\[ \quad \quad z = x \times y \]
\[ \quad \quad \text{return } z \]
\[ \quad \text{def backward(dz):} \]
\[ \quad \quad \# dx = \ldots \text{ #todo} \]
\[ \quad \quad \# dy = \ldots \text{ #todo} \]
\[ \quad \quad \text{return } [dx, dy] \]

\( \frac{\partial L}{\partial z} \)
\( \frac{\partial L}{\partial x} \)

\( x, y, z \) are scalars
Implementation: forward/backward API

```
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]
```

(x,y,z are scalars)
Example: Torch Layers
Example: Torch Layers
Example: Torch MulConstant

\[ f(X) = aX \]

- Initialization
- forward()
- backward()
Example: Caffe Layers
Caffe Sigmoid Layer

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

\[(1 - \sigma(x)) \sigma(x) \]

*top_diff (chain rule)
Gradients for vectorized code (x,y,z are now vectors)

This is now the Jacobian matrix (derivative of each element of z w.r.t. each element of x)

\[
\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x}
\]

“local gradient”

\[
\frac{\partial L}{\partial z}
\]

\[
\frac{\partial L}{\partial y}
\]

\[
\frac{\partial z}{\partial x}
\]

\[
\frac{\partial z}{\partial y}
\]

\[
f
\]

\[
x
\]

\[
y
\]

\[
z
\]

gradients
Vectorized operations

\[ f(x) = \max(0, x) \] (elementwise)

4096-d input vector \rightarrow 4096-d output vector
Vectorized operations

\[ f(x) = \max(0,x) \] (elementwise)

Q: what is the size of the Jacobian matrix?
Vectorized operations

\[ \frac{\partial L}{\partial x} = \begin{bmatrix} \frac{\partial f}{\partial x} \end{bmatrix} \frac{\partial L}{\partial f} \]

Jacobian matrix

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

Q2: what does it look like?

4096-d input vector

\( f(x) = \max(0,x) \) (elementwise)

4096-d output vector
Vectorized operations

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

\[ f(x) = \max(0, x) \] (elementwise)

100 4096-d input vectors \rightarrow \rightarrow \rightarrow \rightarrow 100 4096-d output vectors

I.e. Jacobian would technically be a \([409,600 \times 409,600]\) matrix \(\vdash\)
Assignment: Writing SVM/Softmax

Stage your forward/backward computation!

E.g. for the SVM:

```python
# receive W (weights), X (data)
# forward pass (we have 8 lines)
scores = #...
margins = #...
data_loss = #...
reg_loss = #...
loss = data_loss + reg_loss
# backward pass (we have 5 lines)
dmargins = # ... (optionally, we go direct to dscores)
dscores = #...
dW = #...
```

\[ f = Wx \]

\[ L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \]

Margins
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: $f = W x$

(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

\[ f = W_2 \max(0, W_1 x) \]

\[ f = W_3 \max(0, W_2 \max(0, W_1 x)) \]
Full implementation of training a 2-layer Neural Network needs ~11 lines:

```python
X = np.array([[0,0,1],[0,1,1],[1,0,1],[1,1,1]])
y = np.array([[0,1,1,0]]).T
syn0 = 2*np.random.random((3,4)) - 1
syn1 = 2*np.random.random((4,1)) - 1
for j in xrange(60000):
    l1 = 1/(1+np.exp(-(np.dot(X,syn0))))
    l2 = 1/(1+np.exp(-(np.dot(l1,syn1))))
    l2_delta = (y - l2)*(l2*(1-l2))
    l1_delta = l2_delta.dot(syn1.T) * (l1 * (1-l1))
    syn1 += l1.T.dot(l2_delta)
    syn0 += X.T.dot(l1_delta)
```

from @iamtrask, http://iamtrask.github.io/2015/07/12/basic-python-network/
Assignment: Writing 2layer Net

Stage your forward/backward computation!

```python
# 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 = #...
```
Neuron diagram:

- **Dendrites**: Impulses carried toward cell body.
- **Nucleus**: Central control center.
- **Cell Body**: Origin of the axon.
- **Axon**: Main conductive pathway that branches out.
- **Branches of axon**: Impulses carried away from cell body.
- **Axon terminals**: End points of the axon for synaptic connections.
impulses carried toward cell body

branches of axon

impulses carried away from cell body

dendrites

nucleus

cell body

axon

axon terminals

$\mathbf{x}_0 \rightarrow w_0$ synapse

$w_0 \mathbf{x}_0$ axon from a neuron
dendrite

cell body

$\sum_i w_i x_i + b \rightarrow f$ activation function

output axon

Fei-Fei Li & Andrej Karpathy & Justin Johnson  Lecture 4 - 72  13 Jan 2016
class Neuron:
    # ...
    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
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
Activation Functions

Sigmoid
\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

\text{tanh} \, \text{tanh}(x)

\text{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
\[ f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha (\exp(x) - 1) & \text{if } x \leq 0 \end{cases} \]
Neural Networks: Architectures

“2-layer Neural Net”, or “1-hidden-layer Neural Net”

“Fully-connected” layers

“3-layer Neural Net”, or “2-hidden-layer Neural Net”
Example Feed-forward computation of a Neural Network

```python
class Neuron:
    # ...
    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.
Example Feed-forward computation of a Neural Network

```python
# 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)
```
Setting the number of layers and their sizes

3 hidden neurons  

6 hidden neurons  

20 hidden neurons  

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

(\(\lambda = 0.001\))  \hspace{1cm} (\(\lambda = 0.01\))  \hspace{1cm} (\(\lambda = 0.1\))

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

More than you ever wanted to know about Neural Networks and how to train them.
reverse-mode differentiation (if you want effect of many things on one thing)
\[ \frac{\partial y}{\partial x} \text{ for many different } x \]

forward-mode differentiation (if you want effect of one thing on many things)
\[ \frac{\partial y}{\partial x} \text{ for many different } y \]