Week 2: Training Neural Networks

Erkut Erdem

Visualization of the cat face neuron, from Le et al. Building high-level features using large-scale unsupervised learning, ICML 2012
Today’s Lecture

• Part 1: Backpropagation and Neural Networks (Basics)

• Part 2: Training Neural Networks (Optimization, Learning tricks)
Part 1

Backpropagation and Neural Networks
Where we are...

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

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
```

(image credits to Alec Radford)
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
Computational Graph

\[ f = WX \]

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

input image
weights
loss
Neural Turing Machine

input tape

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

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

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

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

#### Chain rule:

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

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

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
activations

\( x \)  

\( y \)  

\( f \)  

\( z \)
activations

"local gradient"

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

\[ f \]

[Image]
activations

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

"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”

gradients
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
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

z
Another example:

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

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

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

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

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

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

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

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

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

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

\[
\begin{align*}
  \frac{df}{dx} &= 1
\end{align*}
\]

\[
\begin{align*}
  \frac{df}{dx} &= -1/x^2
\end{align*}
\]
Another example: \[ f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \]

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

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

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

\[ (1)(-0.53) = -0.53 \]
Another example:

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

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

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

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

\[
\begin{align*}
  f(x) & = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x \\
  f_a(x) & = ax \quad \rightarrow \quad \frac{df}{dx} = a \\
  f_c(x) & = c + x \quad \rightarrow \quad \frac{df}{dx} = 1
\end{align*}
\]
Another example: \[ f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \]

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

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

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

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

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

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

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

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

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

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

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

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

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

[local gradient] x [its gradient]

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

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

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

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

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

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

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

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

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

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

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

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

[local gradient] x [its gradient]

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

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

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

\[
\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left( \frac{1 + e^{-x} - 1}{1 + e^{-x}} \right) \left( \frac{1}{1 + e^{-x}} \right) = (1 - \sigma(x)) \sigma(x)
\]
\[
    f(w, x) = \frac{1}{1 + e^{- (w_0 x_0 + w_1 x_1 + w_2)}}
    \]

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

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

sigmoid function

sigmoid gate

\[
(0.73) \times (1 - 0.73) = 0.2
\]
Patterns in backward flow

**add** gate: gradient distributor

**max** gate: gradient router

**mul** gate: gradient… “switcher”?

![Diagram showing patterns in backward flow with nodes and arrows representing x, y, z, w, and their respective values.](image-url)
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

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

(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

<table>
<thead>
<tr>
<th>Layer Name</th>
<th>Function Description</th>
<th>Output Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolution2D</td>
<td>Performs 2D convolution</td>
<td></td>
</tr>
<tr>
<td>Pooling2D</td>
<td>Performs 2D pooling</td>
<td></td>
</tr>
<tr>
<td>BatchNorm</td>
<td>Normalizes the input</td>
<td></td>
</tr>
<tr>
<td>ReLU</td>
<td>Applies rectified linear activation</td>
<td></td>
</tr>
<tr>
<td>Dropout</td>
<td>Randomly sets some input values to zero</td>
<td></td>
</tr>
<tr>
<td>FullyConnected</td>
<td>Performs inner product of the input and weight</td>
<td></td>
</tr>
<tr>
<td>Softmax</td>
<td>Applies the softmax function to the input</td>
<td></td>
</tr>
</tbody>
</table>

... (Continued with more layers)
Caffe Sigmoid Layer

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

\[(1 - \sigma(x)) \sigma(x) \] \(*\text{top\_diff} \quad \text{chain rule}\)
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)

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

"local gradient"
Vectorized operations

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

(elementwise)

4096-d input vector

4096-d output vector
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-d input vector

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

4096-d output vector
Vectorized operations

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

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) \hspace{2cm} (elementwise)

4096-d output vector
Vectorized operations

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

\[ f(x) = \max(0, x) \quad \text{(elementwise)} \]

100 4096-d input vectors \[ \rightarrow \]

100 4096-d output vectors

\[ \text{i.e. Jacobian would technically be a } [409,600 \times 409,600] \text{ matrix :} \]
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: \( f = W x \)

(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

\( f = W_2 \max(0, W_1 x) \)
Neural Network: without the brain stuff

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

(Now) 2-layer Neural Network

\( f = W_2 \max(0, W_1x) \)
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/
sigmoid activation function

\[
\frac{1}{1 + e^{-x}}
\]
```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
```
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]
Human Visual System vs. ConvNets

Felleman & van Essen, 1991

http://josephpcohen.com/w/visualizing-cnn-architectures-side-by-side-with-mxnet/
Activation Functions

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

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

\text{ReLU} \quad \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"

"3-layer Neural Net", or "2-hidden-layer Neural Net"

"Fully-connected" layers
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

```
# 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:

\[
\lambda = 0.001 \quad \lambda = 0.01 \quad \lambda = 0.1
\]

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

forward-mode differentiation (if you want effect of one thing on many things)
\[
\frac{\partial y}{\partial x}
\]
for many different \(y\)
Part 2

Training Neural Networks
Where we are now...

Mini-batch SGD

Loop:
1. **Sample** a batch of data
2. **Forward** prop it through the graph, get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient
Where we are now...

(image credits to Alec Radford)
Overview

1. One time setup
   activation functions, preprocessing, weight initialization, regularization, gradient checking

2. Training dynamics
   babysitting the learning process, parameter updates, hyperparameter optimization

3. Evaluation
   model ensembles
Activation Functions
Activation Functions

\[ f \left( \sum_{i} w_i x_i + b \right) \]

- \( x_0 \) axon from a neuron
- \( w_0 \) synapse
- \( w_0 x_0 \) dendrite
- \( w_1 x_1 \) cell body
- \( w_2 x_2 \) output axon
- \( \sum_{i} w_i x_i + b \) activation function
Activation Functions

**Sigmoid**

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

**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**

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

Sigmoid

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

- Squashes numbers to range $[0,1]$
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron
Activation Functions

**Sigmoid**

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

- Squashes numbers to range \([0, 1]\)
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients
What happens when \( x = -10 \)?
What happens when \( x = 0 \)?
What happens when \( x = 10 \)?
Activation Functions

Sigmoid

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

- Squashes numbers to range \([0,1]\)
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
Consider what happens when the input to a neuron (x) is always positive:

\[ f \left( \sum_i w_i x_i + b \right) \]

What can we say about the gradients on \( w \)?
Consider what happens when the input to a neuron is always positive...

\[ f \left( \sum_i w_i x_i + b \right) \]

What can we say about the gradients on \( w \)?
Always all positive or all negative :(
(this is also why you want zero-mean data!)
Activation Functions

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

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive
Activation Functions

- Squashes numbers to range \([-1,1]\)
- zero centered (nice)
- still kills gradients when saturated :(

\[ \text{tanh}(x) \]

[LeCun et al., 1991]
Activation Functions

- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

ReLU
(Rectified Linear Unit)

[Krizhevsky et al., 2012]
Activation Functions

- Computes $f(x) = \max(0,x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Not zero-centered output
- An annoyance:

hint: what is the gradient when $x < 0$?
What happens when $x = -10$?
What happens when $x = 0$?
What happens when $x = 10$?
active ReLU

dead ReLU will never activate
=> never update
=> people like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

active ReLU

dead ReLU will never activate
=> never update
Activation Functions

Leaky ReLU

\[ f(x) = \max(0.01x, x) \]

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- will not “die”.

[Mass et al., 2013]
[He et al., 2015]
Activation Functions

Leaky ReLU

\[ f(x) = \max(0.01x, x) \]

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- will not “die”.

Parametric Rectifier (PReLU)

\[ f(x) = \max(\alpha x, x) \]

backprop into \( \alpha \) (parameter)

[Mass et al., 2013]
[He et al., 2015]
Activation Functions

**Exponential Linear Units (ELU)**

- All benefits of ReLU
- Does not die
- Closer to zero mean outputs
- Computation requires \( \exp() \)

\[
f(x) = \begin{cases} 
  x & \text{if } x > 0 \\
  \alpha (\exp(x) - 1) & \text{if } x \leq 0 
\end{cases}
\]
Maxout “Neuron”
- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

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

Problem: doubles the number of parameters/neuron :(

[Goodfellow et al., 2013]
Summary: In practice:

- Use **ReLU**. Be careful with your learning rates
- Try out **Leaky ReLU / Maxout / ELU**
- Try out **tanh** but don’t expect much
- Don’t use **sigmoid**
Data Preprocessing
Step 1: Preprocess the data

(Assume X [NxD] is data matrix, each example in a row)
Step 1: Preprocess the data

In practice, you may also see **PCA** and **Whitening** of the data.

- **original data**
- **decorrelated data** (data has diagonal covariance matrix)
- **whitened data** (covariance matrix is the identity matrix)
Summary: In practice for Images: center only

- Subtract the mean image (e.g. AlexNet) (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet) (mean along each channel = 3 numbers)

Not common to normalize variance, to do PCA or whitening
Weight Initialization
Q: what happens when W=0 init is used?
- First idea: **Small random numbers**
  (gaussian with zero mean and 1e-2 standard deviation)

\[
W = 0.01 \times \text{np.random.randn}(D,H)
\]
- First idea: **Small random numbers**
  (gaussian with zero mean and $1e^{-2}$ standard deviation)

$$W = 0.01 \times \text{np.random.randn}(D,H)$$

Works ~okay for small networks, but can lead to non-homogeneous distributions of activations across the layers of a network.
Lets look at some activation statistics

E.g. 10-layer net with 500 neurons on each layer, using tanh non-linearities, and initializing as described in last slide:

```python
# assume some unit gaussian 10-D input data
D = np.random.randn(1000, 500)
hidden_layer_sizes = [500]*10
nonlinearities = ['tanh']*len(hidden_layer_sizes)

act = {'relu': lambda x:np.maximum(0,x), 'tanh': lambda x:np.tanh(x)}
Hs = {}
for i in range(len(hidden_layer_sizes)):
    X = D if i == 0 else Hs[i-1] # input at this layer
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out) * 0.01 # layer initialization
    H = np.dot(X, W) # matrix multiply
    H = act[nonlinearities[i]](H) # nonlinearity
    Hs[i] = H # cache result on this layer

# look at distributions at each layer
print 'input layer had mean %f and std %f' % (np.mean(D), np.std(D))
layer_means = [np.mean(H) for i,H in Hs.items()]
layer_stds = [np.std(H) for i,H in Hs.items()]
for i,H in Hs.items():
    print 'hidden layer %d had mean %f and std %f' % (i+1, layer_means[i], layer_stds[i])

# plot the means and standard deviations
plt.figure()
plt.subplot(211)
plt.plot(Hs.keys(), layer_means, 'ob-')
plt.title('layer mean')
plt.subplot(212)
plt.plot(Hs.keys(), layer_stds, 'or-')
plt.title('layer std')

# plot the raw distributions
plt.figure()
for i,H in Hs.items():
    plt.subplot(1, len(Hs), i+1)
    plt.hist(H.ravel(), 30, range=(-1,1))
```
input layer had mean 0.000927 and std 0.998368
hidden layer 1 had mean -0.000117 and std 0.213061
hidden layer 2 had mean -0.000001 and std 0.647551
hidden layer 3 had mean -0.000002 and std 0.610639
hidden layer 4 had mean 0.000001 and std 0.002370
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
All activations become zero!

Q: think about the backward pass. What do the gradients look like?

Hint: think about backward pass for a $W \times X$ gate.
$W = \text{np.random.randn}(\text{fan\_in}, \text{fan\_out}) \times 1.0 \# \text{layer initialization}$

Almost all neurons completely saturated, either -1 and 1. Gradients will be all zero.

*1.0 instead of *0.01
**“Xavier initialization”**

[Glorot et al., 2010]

Reasonable initialization.
(Mathematical derivation assumes linear activations)
but when using the ReLU nonlinearity it breaks.
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization

He et al., 2015
(note additional /2)
He et al., 2015
(note additional /2)
Proper initialization is an active area of research...

*Understanding the difficulty of training deep feedforward neural networks* by Glorot and Bengio, 2010

*Exact solutions to the nonlinear dynamics of learning in deep linear neural networks* by Saxe et al, 2013

*Random walk initialization for training very deep feedforward networks* by Sussillo and Abbott, 2014

*Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification* by He et al., 2015

*Data-dependent Initializations of Convolutional Neural Networks* by Krähenbühl et al., 2015

*All you need is a good init*, Mishkin and Matas, 2015

...
Batch Normalization

“you want unit gaussian activations? just make them so.”

consider a batch of activations at some layer. To make each dimension unit gaussian, apply:

\[
\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}
\]

this is a vanilla differentiable function...
**Batch Normalization**

“you want unit gaussian activations? just make them so.”

1. compute the empirical mean and variance independently for each dimension.

2. Normalize

\[
\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}
\]

[ioffe and Szegedy, 2015]
Batch Normalization

Usually inserted after Fully Connected / (or Convolutional, as we’ll see soon) layers, and before nonlinearity.

Problem: do we necessarily want a unit gaussian input to a tanh layer?

\[
\hat{x}(k) = \frac{x(k) - E[x(k)]}{\sqrt{\text{Var}[x(k)]}}
\]
Batch Normalization

Normalize:

\[ \hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}} \]

And then allow the network to squash the range if it wants to:

\[ y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)} \]

Note, the network can learn:

\[ \gamma^{(k)} = \sqrt{\text{Var}[x^{(k)}]} \]
\[ \beta^{(k)} = E[x^{(k)}] \]

to recover the identity mapping.

[Ioffe and Szegedy, 2015]
Batch Normalization

**Input:** Values of \( x \) over a mini-batch: \( \mathcal{B} = \{x_1...x_m\} \);
Parameters to be learned: \( \gamma, \beta \)

**Output:** \( \{y_i = \text{BN}_{\gamma,\beta}(x_i)\} \)

\[
\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \quad \text{// mini-batch mean}
\]

\[
\sigma^2_B \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2 \quad \text{// mini-batch variance}
\]

\[
\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma^2_B + \epsilon}} \quad \text{// normalize}
\]

\[
y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \quad \text{// scale and shift}
\]

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

[ioffe and Szegedy, 2015]
Batch Normalization

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_1, \ldots, x_m\}$; Parameters to be learned: $\gamma, \beta$

**Output:** $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

\[
\begin{align*}
\mu_{\mathcal{B}} & \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i & \text{// mini-batch mean} \\
\sigma_{\mathcal{B}}^2 & \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 & \text{// mini-batch variance} \\
\hat{x}_i & \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} & \text{// normalize} \\
y_i & \leftarrow \gamma \hat{x}_i + \beta = \text{BN}_{\gamma, \beta}(x_i) & \text{// scale and shift}
\end{align*}
\]

---

**Note:** at test time BatchNorm layer functions differently:

The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.

(e.g. can be estimated during training with running averages)

[Ioffe and Szegedy, 2015]
Babysitting the Learning Process
Step 1: Preprocess the data

(Assume X [NxD] is data matrix, each example in a row)
Step 2: Choose the architecture:
say we start with one hidden layer of 50 neurons:

CIFAR-10 images, 3072 numbers

50 hidden neurons

input layer

hidden layer

output layer

10 output neurons, one per class
Double check that the loss is reasonable:

```python
def init_two_layer_model(input_size, hidden_size, output_size):
    # initialize a model
    model = {}
    model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
    model['b1'] = np.zeros(hidden_size)
    model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
    model['b2'] = np.zeros(output_size)
    return model
```

```
model = init_two_layer_model(32*32*3, 50, 10)  # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train[0,0])  # disable regularization
print(loss)
```

```
2.30261216167
```

loss ~2.3. “correct“ for 10 classes
returns the loss and the gradient for all parameters
Double check that the loss is reasonable:

```python
def init_two_layer_model(input_size, hidden_size, output_size):
    # initialize a model
    model = {}
    model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
    model['b1'] = np.zeros(hidden_size)
    model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
    model['b2'] = np.zeros(output_size)
    return model
```

```python
model = init_two_layer_model(32*32*3, 50, 10)  # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train[1e3])  # crank up regularization
print loss
```

3.06859716482

loss went up, good. (sanity check)
Lets try to train now…

**Tip**: Make sure that you can overfit very small portion of the training data

The above code:
- take the first 20 examples from CIFAR-10
- turn off regularization (reg = 0.0)
- use simple vanilla ‘sgd’
Lets try to train now...

**Tip:** Make sure that you can overfit very small portion of the training data.

Very small loss, train accuracy 1.00, nice!
Lets try to train now...

I like to start with small regularization and find learning rate that makes the loss go down.

```python
model = init_two_layer_model(32*32*3, 50, 10)  # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
  model, two_layer_net,
  num_epochs=10, reg=0.000001,
  update='sgd', learning_rate_decay=1,
  sample_batches = True,
  learning_rate=1e-6, verbose=True)
```
Let's try to train now...

I like to start with small regularization and find learning rate that makes the loss go down.

```python
model = init_two_layer_model(32*32*3, 50, 10)  # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
model, two_layer_net,
num_epochs=10, reg=0.000001,
update='sgd', learning_rate_decay=1,
sample_batches=False,
learning_rate=1e-6, verbose=True)
```

Loss barely changing
Let's try to train now...

I like to start with small regularization and find learning rate that makes the loss go down.

Loss not going down: learning rate too low

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
    model, two_layer_net,
    num_epochs=10, reg=0.000001,
    update='sgd', learning_rate_decay=1,
    sample_batches=True,
    learning_rate=1e-6, verbose=True)
```

Loss barely changing: Learning rate is probably too low
Lets try to train now…

I like to start with small regularization and find learning rate that makes the loss go down.

loss not going down: learning rate too low

Loss barely changing: Learning rate is probably too low

Notice train/val accuracy goes to 20% though, what’s up with that? (remember this is softmax)
Lets try to train now...

I like to start with small regularization and find learning rate that makes the loss go down.

loss not going down: learning rate too low

Okay now lets try learning rate 1e6. What could possibly go wrong?
Lets try to train now...

I like to start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low

loss exploding:
learning rate too high

cost: NaN almost always means high learning rate...
Lets try to train now...

I like to start with small regularization and find learning rate that makes the loss go down.

**loss not going down:**
learning rate too low

**loss exploding:**
learning rate too high

3e-3 is still too high. Cost explodes....

=> Rough range for learning rate we should be cross-validating is somewhere [1e-3 ... 1e-5]
Hyperparameter Optimization
Cross-validation strategy

I like to do coarse -> fine cross-validation in stages

**First stage:** only a few epochs to get rough idea of what params work
**Second stage:** longer running time, finer search

… (repeat as necessary)

Tip for detecting explosions in the solver:
If the cost is ever > 3 * original cost, break out early
For example: run coarse search for 5 epochs

```python
max_count = 100
for count in range(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6)

    trainer = ClassifierTrainer()
    model = init_two_layer_model([32*32*3, 50, 10]) # input size, hidden size, number of classes
    trainer = ClassifierTrainer()

    best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                           model, two_layer_net,
                                           num_epochs=5, reg=reg,
                                           update='momentum', learning_rate_decay=0.9,
                                           sample_batches=True, batch_size=100,
                                           learning_rate=lr, verbose=False)
```

Note it’s best to optimize in log space!

```
val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```
Now run finer search...

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6)

    val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
    val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
    val_acc: 0.512000, lr: 8.600827e-04, reg: 1.349737e-02, (2 / 100)
    val_acc: 0.461000, lr: 1.028372e-04, reg: 1.220193e-02, (3 / 100)
    val_acc: 0.468000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
    val_acc: 0.498000, lr: 9.477776e-04, reg: 2.002293e-03, (5 / 100)
    val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
    val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
    val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
    val_acc: 0.489000, lr: 1.979168e-04, reg: 1.016889e-04, (9 / 100)
    val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
    val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287887e-01, (11 / 100)
    val_acc: 0.468000, lr: 1.135527e-04, reg: 3.905840e-02, (12 / 100)
    val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562888e-02, (13 / 100)
    val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
    val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
    val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
    val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
    val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850665e-03, (18 / 100)
    val_acc: 0.500000, lr: 2.412648e-04, reg: 4.997821e-04, (19 / 100)
    val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
    val_acc: 0.516000, lr: 8.639527e-04, reg: 1.528291e-02, (21 / 100)
```

53% - relatively good for a 2-layer neural net with 50 hidden neurons.
Now run finer search...

\[
\text{max_count = 100}
\]
\[
\text{for count in xrange(max_count):}
\]
\[
\quad \text{reg = 10**uniform(-5, 5)}
\]
\[
\quad \text{lr = 10**uniform(-3, -6)}
\]

\[
\begin{array}{cccc}
\text{val_acc} & \text{lr} & \text{reg} & \text{epochs} \\
0.527000 & 5.340517e-04 & 4.097824e-01 & 0 / 100 \\
0.492000 & 2.279484e-04 & 9.993425e-04 & 1 / 100 \\
0.512000 & 8.680027e-04 & 1.349727e-02 & 2 / 100 \\
0.461000 & 1.028372e-04 & 1.220193e-02 & 3 / 100 \\
0.468000 & 1.113730e-04 & 5.244389e-02 & 4 / 100 \\
0.498000 & 9.477776e-04 & 2.001293e-03 & 5 / 100 \\
0.469000 & 1.484369e-04 & 4.328313e-01 & 6 / 100 \\
0.522000 & 5.586261e-04 & 2.312685e-04 & 7 / 100 \\
0.530000 & 5.808183e-04 & 8.259964e-02 & 8 / 100 \\
0.489000 & 1.979168e-04 & 1.016899e-04 & 9 / 100 \\
0.499000 & 2.036031e-04 & 2.406271e-03 & 10 / 100 \\
0.475000 & 2.021162e-04 & 2.287807e-01 & 11 / 100 \\
0.468000 & 1.135527e-04 & 3.905040e-02 & 12 / 100 \\
0.515000 & 6.947668e-04 & 1.562888e-02 & 13 / 100 \\
0.531000 & 9.471549e-04 & 1.433695e-03 & 14 / 100 \\
0.509000 & 3.140688e-04 & 2.857518e-01 & 15 / 100 \\
0.514000 & 6.438349e-04 & 3.033781e-01 & 16 / 100 \\
0.502000 & 3.921784e-04 & 2.707126e-04 & 17 / 100 \\
0.509000 & 9.752279e-04 & 2.850656e-03 & 18 / 100 \\
0.500000 & 2.142048e-04 & 4.997821e-04 & 19 / 100 \\
0.466000 & 1.319314e-04 & 1.189915e-02 & 20 / 100 \\
0.516000 & 8.639527e-04 & 1.528291e-02 & 21 / 100 \\
\end{array}
\]

53% - relatively good for a 2-layer neural net with 50 hidden neurons.

But this best cross-validation result is worrying. Why?
Random Search vs. Grid Search

Grid Layout

Random Layout

Random Search for Hyper-Parameter Optimization
Bergstra and Bengio, 2012
Hyperparameters to play with:
- network architecture
- learning rate, its decay schedule, update type
- regularization (L2/Dropout strength)

neural networks practitioner
music = loss function
Monitor and visualize the loss curve
Bad initialization a prime suspect
Monitor and visualize the accuracy:

- **big gap** = overfitting
  => increase regularization strength?

- **no gap**
  => increase model capacity?
Track the ratio of weight updates / weight magnitudes:

```python
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW  # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update  # the actual update
print update_scale / param_scale  # want ~1e-3
```

ratio between the values and updates: \(~0.0002 / 0.02 = 0.01\) (about okay)
want this to be somewhere around 0.001 or so
So far..

We looked in detail at:

- Activation Functions (use ReLU)
- Data Preprocessing (images: subtract mean)
- Weight Initialization (use Xavier init)
- Batch Normalization (use)
- Babysitting the Learning process
- Hyperparameter Optimization
  (random sample hyperparams, in log space when appropriate)
NOW

- Parameter update schemes
- Learning rate schedules
- Model ensembles
- More on regularization: Dropout
Parameter Updates
Training a neural network, main loop:

```python
while True:
    data_batch = dataset.sample_data_batch()
    loss = network.forward(data_batch)
    dx = network.backward()
    x += -learning_rate * dx
```
Training a neural network, main loop:

```python
while True:
    data_batch = dataset.sample_data_batch()
    loss = network.forward(data_batch)
    dx = network.backward()
    x += -learning_rate * dx
```

simple gradient descent update now: complicate.
Suppose loss function is steep vertically but shallow horizontally:

Q: What is the trajectory along which we converge towards the minimum with SGD?
Suppose loss function is steep vertically but shallow horizontally:

Q: What is the trajectory along which we converge towards the minimum with SGD?
Suppose loss function is steep vertically but shallow horizontally:

Q: What is the trajectory along which we converge towards the minimum with SGD? very slow progress along flat direction, jitter along steep one
Momentum update

- Physical interpretation as ball rolling down the loss function + friction (mu coefficient).
- mu = usually ~0.5, 0.9, or 0.99 (Sometimes annealed over time, e.g. from 0.5 -> 0.99)

```python
# Gradient descent update
x += - learning_rate * dx

# Momentum update
v = mu * v - learning_rate * dx  # integrate velocity
x += v # integrate position
```
Momentum update

- Allows a velocity to “build up” along shallow directions
- Velocity becomes damped in steep direction due to quickly changing sign
SGD vs Momentum

notice momentum overshooting the target, but overall getting to the minimum much faster.
Nesterov Momentum update

# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position

Ordinary momentum update:

- momentum step
- gradient step
- actual step
Nesterov Momentum update

Momentum update

- momentum step
- gradient step
- actual step

Nesterov momentum update

- momentum step
- “lookahead” gradient step (bit different than original)
- actual step
Nesterov Momentum update

\[ v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} + \mu v_{t-1}) \]

\[ \theta_t = \theta_{t-1} + v_t \]
Nesterov Momentum update

\[ v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} + \mu v_{t-1}) \]

\[ \theta_t = \theta_{t-1} + v_t \]

Slightly inconvenient…
usually we have:

\[ \theta_{t-1}, \nabla f(\theta_{t-1}) \]

Variable transform and rearranging saves the day:

Replace all thetas with phis, rearrange and obtain:

\[ \phi_{t-1} = \theta_{t-1} + \mu v_{t-1} \]

\[ v_t = \mu v_{t-1} - \epsilon \nabla f(\phi_{t-1}) \]

\[ \phi_t = \phi_{t-1} - \mu v_{t-1} + (1 + \mu)v_t \]

# Nesterov momentum update rewrite

\[ v_{prev} = v \]
\[ v = \mu u * v - \text{learning rate} * dx \]
\[ x += -\mu u * v_{prev} + (1 + \mu u) * v \]
nag = Nesterov Accelerated Gradient
AdaGrad update

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

[Duchi et al., 2011]
AdaGrad update

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

Q: What happens with AdaGrad?
AdaGrad update

```python
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

Q2: What happens to the step size over long time?
RMSProp update

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

```
# RMSProp

cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```
rmsprop: A mini-batch version of rprop

- rprop is equivalent to using the gradient but also dividing by the size of the gradient.
  - The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?
- rmsprop: Keep a moving average of the squared gradient for each weight
  \[ \text{MeanSquare}(w, t) = 0.9 \text{MeanSquare}(w, t-1) + 0.1 \left( \frac{\partial E}{\partial w(t)} \right)^2 \]
- Dividing the gradient by \( \sqrt{\text{MeanSquare}(w, t)} \) makes the learning work much better (Tijmen Tieleman, unpublished).

Cited by several papers as:

adagrad
rmsprop
Adam update
(incomplete, but close)

```
# Adam
m = beta1*m + (1-beta1)*dx  # update first moment
v = beta2*v + (1-beta2)*(dx**2)  # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

[Kingma and Ba, 2014]
Adam update
(incomplete, but close)

Looks a bit like RMSProp with momentum

# Adam

\[
\begin{align*}
m &= \beta_1 m + (1-\beta_1)dx & \# \text{update first moment} \\
v &= \beta_2 v + (1-\beta_2)(dx^2) & \# \text{update second moment} \\
x &= x - \text{learning\_rate} \times m / (\sqrt{v} + 1e-7)
\end{align*}
\]

[Kingma and Ba, 2014]
Adam update

(incomplete, but close)

Adam update formula:

\[
\begin{align*}
m &= \text{betal} \cdot m + (1 - \text{betal}) \cdot \text{dx} \quad \# \text{update first moment} \\
v &= \text{beta2} \cdot v + (1 - \text{beta2}) \cdot (\text{dx}^2) \quad \# \text{update second moment} \\
\text{x} &= -\text{learning\_rate} \cdot m / (\text{np.sqrt}(v) + 1e-7)
\end{align*}
\]

Looks a bit like RMSProp with momentum

RMSProp update formula:

\[
\begin{align*}
\text{cache} &= \text{decay\_rate} \cdot \text{cache} + (1 - \text{decay\_rate}) \cdot \text{dx}^2 \\
\text{x} &= \text{learning\_rate} \cdot \text{dx} / (\text{np.sqrt}(\text{cache}) + 1e-7)
\end{align*}
\]

[Kingma and Ba, 2014]
The bias correction compensates for the fact that \( m, v \) are initialized at zero and need some time to “warm up”.

RMSProp-like bias correction (only relevant in first few iterations when \( t \) is small)

momentum
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.

Q: Which one of these learning rates is best to use?
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.

=> **Learning rate decay over time!**

**step decay:**
e.g. decay learning rate by half every few epochs.

$$\alpha = \alpha_0 e^{-kt}$$

**exponential decay:**

$$\alpha = \alpha_0 / (1 + kt)$$

**1/t decay:**
Second order optimization methods

second-order Taylor expansion:

\[
J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^\top \nabla_\theta J(\theta_0) + \frac{1}{2} (\theta - \theta_0)^\top H(\theta - \theta_0)
\]

Solving for the critical point we obtain the Newton parameter update:

\[
\theta^* = \theta_0 - H^{-1} \nabla_\theta J(\theta_0)
\]

Q: what is nice about this update?
Second order optimization methods

second-order Taylor expansion:

\[ J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^\top \nabla_{\theta} J(\theta_0) + \frac{1}{2} (\theta - \theta_0)^\top H(\theta - \theta_0) \]

Solving for the critical point we obtain the Newton parameter update:

\[ \theta^* = \theta_0 - H^{-1} \nabla_{\theta} J(\theta_0) \]

notice:
no hyperparameters! (e.g. learning rate)

Q2: why is this impractical for training Deep Neural Nets?
Second order optimization methods

\[ \theta^* = \theta_0 - H^{-1} \nabla_{\theta} J(\theta_0) \]

- Quasi-Newton methods \textbf{(BGFS most popular)}: instead of inverting the Hessian \((O(n^3))\), approximate inverse Hessian with rank 1 updates over time \((O(n^2)\) each).

- \textbf{L-BFGS} (Limited memory BFGS): Does not form/store the full inverse Hessian.
L-BFGS

- Usually works very well in full batch, deterministic mode i.e. if you have a single, deterministic $f(x)$ then L-BFGS will probably work very nicely

- Does not transfer very well to mini-batch setting. Gives bad results. Adapting L-BFGS to large-scale, stochastic setting is an active area of research.
In practice:

- **Adam** is a good default choice in most cases

- If you can afford to do full batch updates then try out **L-BFGS** (and don’t forget to disable all sources of noise)
Evaluation:
Model Ensembles
1. Train multiple independent models
2. At test time average their results

Enjoy 2% extra performance

One disadvantage of model ensembles is that they take longer to evaluate on test example!
Model Ensembles:

- Same model, different initializations
- Top models discovered during cross-validation
- Running average of parameters during training
- Different checkpoints of a single model
Model Ensembles:

- **Same model, different initializations**
  - Use cross-validation to determine the best hyperparameters
  - train multiple models with the best set of hyperparameters but with different random initialization
  - Con: Variety is only due to initialization.
Model Ensembles:

- **Top models discovered during cross-validation**
  - Use cross-validation to determine the best hyperparameters
  - pick the top few (e.g. 10) models to form the ensemble.
  - improves the variety of the ensemble
  - In practice, this can be easier to perform since it doesn't require additional retraining of models after cross-validation
  - Con: has the danger of including suboptimal models
Model Ensembles:

- **Running average of parameters during training**
  - maintain a second copy of the network's weights in memory that maintains an exponentially decaying sum of previous weights during training.
  - you're averaging the state of the network over last several iterations.
  - this "smoothed" version of the weights over last few steps almost always achieves better validation error.
Model Ensembles:

- **Different checkpoints of a single model**
  - take different checkpoints of a single network over time (e.g. after every epoch) and use them to form an ensemble.
  - keep track of (and use at test time) a running average parameter vector

```python
while True:
    data_batch = dataset.sample_data_batch()
    loss = network.forward(data_batch)
    dx = network.backward()
    x += -learning_rate * dx
    x_test = 0.995*x_test + 0.005*x  # use for test set
```

- Pro: very cheap, prefer if training is very expensive, works reasonably well in practice
- Con: suffers from some lack of variety.
Regularization (dropout)
Regularization: **Dropout**

“randomly set some neurons to zero in the forward pass”

(a) Standard Neural Net

(b) After applying dropout.

[Srivastava et al., 2014]
p = 0.5  # probability of keeping a unit active. higher = less dropout

```python
def train_step(X):
    """ X contains the data """

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p  # first dropout mask
    H1 *= U1  # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p  # second dropout mask
    H2 *= U2  # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)
```
Waaaait a second…
How could this possibly be a good idea?
Waaaaait a second…
How could this possibly be a good idea?

Forces the network to have a redundant representation.

- has an ear
- has a tail
- is furry
- has claws
- mischievous look

Waaaaait a second…
How could this possibly be a good idea?

Waaaaait a second…
How could this possibly be a good idea?

Waaaaait a second…
How could this possibly be a good idea?

Waaaaait a second…
How could this possibly be a good idea?

Waaaaait a second…
How could this possibly be a good idea?

Waaaaait a second…
How could this possibly be a good idea?

Waaaaait a second…
How could this possibly be a good idea?

Waaaaait a second…
How could this possibly be a good idea?

Waaaaait a second…
How could this possibly be a good idea?

Waaaaait a second…
How could this possibly be a good idea?

Waaaaait a second…
How could this possibly be a good idea?

Waaaaait a second…
How could this possibly be a good idea?

Waaaaait a second…
How could this possibly be a good idea?

Waaaaait a second…
How could this possibly be a good idea?

Waaaaait a second…
How could this possibly be a good idea?

Waaaaait a second…
How could this possibly be a good idea?
Waaaait a second…
How could this possibly be a good idea?

Another interpretation:

Dropout is training a large ensemble of models (that share parameters).

Each binary mask is one model, gets trained on only ~one datapoint.
Dropout prevents co-adaptation by making the presence of other hidden units unreliable. This results in more useful features!

![Features learned on MNIST with one hidden layer autoencoders having 256 rectified linear units.](image)

(a) Without dropout  
(b) Dropout with $p = 0.5$.  

Figure 7: Features learned on MNIST with one hidden layer autoencoders having 256 rectified linear units.
At test time....

**Ideally:**
want to integrate out all the noise

**Monte Carlo approximation:**
do many forward passes with different dropout masks, average all predictions
At test time....
Can in fact do this with a single forward pass! (approximately)
Leave all input neurons turned on (no dropout).

(this can be shown to be an approximation to evaluating the whole ensemble)
At test time….
Can in fact do this with a single forward pass! (approximately)
Leave all input neurons turned on (no dropout).

Q: Suppose that with all inputs present at test time the output of this neuron is $x$.

What would its output be during training time, in expectation? (e.g. if $p = 0.5$)
At test time....
Can in fact do this with a single forward pass! (approximately)
Leave all input neurons turned on (no dropout).

during test: \( a = w_0 x + w_1 y \)
during train:
\[
E[a] = \frac{1}{4} \left( w_0^0 x + w_1^0 y \\
w_0 x + w_1 y \\
w_0^0 x + w_1^0 y \\
w_0 x + w_1 y \right) \\
= \frac{1}{4} \left( 2 w_0 x + 2 w_1 y \right) \\
= \frac{1}{2} \left( w_0 x + w_1 y \right)
\]
At test time…
Can in fact do this with a single forward pass! (approximately)
Leave all input neurons turned on (no dropout).

During test:
\[ a = w_0 x + w_1 y \]

During train:
\[
E[a] = \frac{1}{4} \left( w_0 \cdot 0 + w_1 \cdot 0 \right) + \frac{1}{4} \left( w_0 \cdot y + w_1 \cdot y \right) + \frac{1}{4} \left( w_0 \cdot x + w_1 \cdot 0 \right) + \frac{1}{4} \left( w_0 \cdot x + w_1 \cdot y \right) \\
= \frac{1}{4} \left( 2 \cdot w_0 \cdot x + 2 \cdot w_1 \cdot y \right) \\
= \frac{1}{2} \cdot (w_0 \cdot x + w_1 \cdot y)
\]

With \( p=0.5 \), using all inputs in the forward pass would inflate the activations by 2x from what the network was “used to” during training!
=> Have to compensate by scaling the activations back down by \( \frac{1}{2} \).
We can do something approximate analytically

```python
def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p  # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p  # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```

At test time all neurons are active always
=> We must scale the activations so that for each neuron:
output at test time = expected output at training time
**Dropout Summary**

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

def train_step(X):
    """ X contains the data """

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```

- **Drop in forward pass**
- **Scale at test time**
More common: “Inverted dropout”

```python
p = 0.5  # probability of keeping a unit active. higher = less dropout

def train_step(X):
    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p  # first dropout mask. Notice /p!
    H1 *= U1  # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p  # second dropout mask. Notice /p!
    H2 *= U2  # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1)  # no scaling necessary
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
```
test time is unchanged!
Summary

• Part 1: Backpropagation and Neural Networks (Basics)

• Part 2: Training Neural Networks (Optimization, Learning tricks)

What about the mathematics of deep learning? Why non-convexity does not create a problem in practice?
Local Minima

Distribution of test losses

The Loss Surfaces of Multilayer Networks
Next Lecture:
Convolutional Neural Networks