Reading

• http://cs231n.github.io/neural-networks-1/
• http://cs231n.github.io/neural-networks-2/
• http://cs231n.github.io/neural-networks-case-study/
• http://cs231n.github.io/convolutional-networks/
Announcements

P5 will be out tonight.

Get started before it’s released by reading:

Deep Learning with PyTorch: A 60 Minute Blitz
Goals

• Understand neural networks as a stack of linear classifiers with nonlinearities (activation functions) in between.

• Understand the basic menu of activations (Sigmoid, Tanh, ReLU)
  – Understand the vanishing gradients problem.

• Understand the motivation and behavior of convolutional layers in neural networks.
Backpropagation in Pytorch

- Your deep learning framework knows how to differentiate anything you might want to do.
- Example, in PyTorch:
  - Your classifier inherits from torch.nn.Module
  - You implement its forward() method
  - Torch generates a backward() method for you!
  - Training looks like this (pseudocode)
    ```python
def classifier(data)
    # uses W, b
    def loss_function(output, true_labels)
    loss.backward()  # (backprop magic here!)
    dW = w.grad
    db = b.grad
    W -= step_size * dW
    b -= step_size * dB
    ```
Backpropagation in Pytorch

• Example, in pytorch (pseudocode):
  ```python
  output = classifier(data, W, b)  # uses W, b
  loss = loss_function(output, true_labels)
  loss.backward()  # (backprop magic here!)
  dW = w.grad
  db = b.grad
  W -= step_size * dW
  b -= step_size * db
  ```

• In practice, an Optimizer performs the updates instead:
  ```python
  optimizer = torch.optim.SGD(net.parameters(), lr=0.0001)
  output = classifier(data)
  loss = loss_function(output, true_labels)
  loss.backward()
  optimizer.step()
  ```
Neural Networks: The Brain Stuff

Impulses carried toward cell body

dendrite

cell body

Impulses carried away from cell body

presynaptic terminal

image by Felipe Peruchois licensed under CC-BY 3.0

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung
Neural networks: without the brain stuff

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

Neural Network

Linear classifiers
Neural networks: without the brain stuff

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

(Now) 2-layer Neural Network \( f = W_2 \max(0, W_1x) \)
Neural networks: without the brain stuff

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

(Now) 2-layer Neural Network
\[ f = W_2 \max(0, W_1 x) \]
Neural networks: without the brain stuff

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

(Now) 2-layer Neural Network or 3-layer Neural Network
\( f = W_2 \max(0, W_1x) \)
\( f = W_3 \max(0, W_2 \max(0, W_1x)) \)
Training a 2 layer neural network in 20 lines of python

```python
import numpy as np
from numpy.random import randn

N, D_in, H, D_out = 64, 1000, 100, 10
x, y = randn(N, D_in), randn(N, D_out)
w1, w2 = randn(D_in, H), randn(H, D_out)

for t in range(2000):
    h = 1 / (1 + np.exp(-x.dot(w1)))
y_pred = h.dot(w2)
loss = np.square(y_pred - y).sum()
print(t, loss)

grad_y_pred = 2.0 * (y_pred - y)
grad_w2 = h.T.dot(grad_y_pred)
grad_h = grad_y_pred.dot(w2.T)
grad_w1 = x.T.dot(grad_h * h * (1 - h))
w1 -= 1e-4 * grad_w1
w2 -= 1e-4 * grad_w2
```
“Hidden Layers”

$W_1$, a $3 \times 4$ matrix converts input into hidden layer activations.

$W_2$, a $4 \times 2$ matrix transforms hidden layer activations to output scores.

Figures: Fei-Fei Li, Justin Johnson, & Serena Yeung
Neural Networks: Nonlinear Classifiers built from Linear Classifiers

Figures: Fei-Fei Li, Justin Johnson, & Serena Yeung
Neural networks: without the brain stuff

(Before) Linear score function: \[ f = WX \]

(Now) 2-layer Neural Network or 3-layer Neural Network
\[ f = W_2 \max(0, W_1 x) \]
\[ f = W_3 \max(0, W_2 \max(0, W_1 x)) \]

???
Activation Functions

\[ f(x, W) = Wx \]
Activation Functions

\[ f(x, W) = Wx \]
Activation Functions

\[ f(x, W) = Wx \]
\[ f(x, W_1, W_2) = W_1(W_2x) \]
Activation Functions

\[ f(x, W) = Wx \]

\[ f(x, W_1, W_2) = W_1(W_2x) \]

\[ W \leftarrow W_1W_2 \]

\[ f(x, W) = Wx \]
Activation Functions

\[ f(x, W) = Wx \]
\[ f(x, W_1, W_2) = W_1(W_2x) \]

\[ W \leftarrow W_1W_2 \]
\[ f(x, W) = Wx \]
Activation Functions

\[ f(x, W) = Wx \]
\[ f(x, W_1, W_2) = W_1(W_2x) \]

\[ W \leftarrow W_1W_2 \]
\[ f(x, W) = Wx \]
Activation Functions

\[ f(x, W_1, W_2, W_3) = W_3 \max(0, W_2 \max(0, W_1 x)) \]
Neural Networks

Neural Network

Linear classifiers
Neural Networks

Linear classifiers

Nonlinearities!
Neural Networks: Nonlinear Classifiers built from Linear Classifiers

Figure: Fei-Fei Li, Justin Johnson, & Serena Yeung
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**
\[ \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases} \]
Taking a step back: Image Recognition

• We have images; ML works on vectors.
• To do machine learning, we need a function that takes an image and converts it into a vector.

\[ \phi \left( \begin{array}{c} \text{image} \\ \text{vector} \end{array} \right) = \text{vector in high dimensional space} \]

• Given an image, use \( \phi \) to get a vector representing a point in high dimensional space
Classifying Images: Pipeline

1. Represent the image in some feature space

\[ \phi \left( \begin{array}{c} \text{image} \end{array} \right) = \begin{array}{c} \text{feature representation} \end{array} \]

2. Classify the image based on its feature representation.

- \[ h\left( \begin{array}{c} \text{feature representation} \end{array} \right) = \text{"dog"} \]
Two important pieces

• The feature extractor ($\phi$)

• The classifier ($h$)
  – (this is what we’ve been talking about this whole time: linear classifiers, now neural networks)
Let’s make the simplest possible $\phi$

- Represent an image as a vector in $\mathbb{R}^d$
- Step 1: convert image to gray-scale and resize to fixed size
Linear classifiers on pixels are bad
Linearly separable classes

\[ f(x_i, W, b) = W x_i + b \]
Linear classifiers on pixels are bad

How do we fix it?

• **Solution 1**: Better feature vectors
• **Solution 2**: Non-linear classifiers
Recap: Life Before Deep Learning

- **Input Pixels**
- **Extract Hand-Crafted Features**
- **Concatenate into a vector $x$**
- **Linear Classifier**

Figure: Karpathy 2016
Linear classifiers on pixels are bad

How do we fix it?

• Solution 1: Better feature vectors
• Solution 2: Non-linear classifiers
A Linear Classifier

- \( y = Wx + b \)
- Every row of \( y \) corresponds to a hyperplane in \( x \) space

The case when \( d_{\text{in}} = 2 \). A single row in \( y \) plotted for every possible value of \( x \)
A Neural Network

- Key idea: build complex functions by composing simple functions

\[ f(x) = Wx, \quad g(x) = \max(x, 0) \]

1 row of \( z \) plotted for every value of \( x \)

1 row of \( y \) plotted for every value of \( x \)
Multilayer perceptron on images

• An example network for cat vs dog
Linear Classifier: Parameter Count

- How many parameters does a linear function have? Suppose:
  - # pixels = 256*256 = 65536
  - # classes = 1024

The case when $d_{\text{in}} = 2$. A single row in $y$ plotted for every possible value of $x$. 
The linear function for images
Linear Classifier: Parameter Count

- How many parameters does a linear function have? Suppose:
  - # pixels = 256*256 = 65536 = $2^{16}$
  - # classes = 1024 = $2^{10}$
Linear Classifier: Parameter Count

• How many parameters does a linear function have? Suppose:
  - # pixels = 256*256 = 65536 = 2^{16}
  - # classes = 1024 = 2^{10}

• $2^{26}$ parameters for a one-layer network on a tiny image.
Idea 1: local connectivity

• Pixels only related to nearby pixels
Idea 2: Translation invariance

- Pixels only related to nearby pixels
- Weights should not depend on the location of the neighborhood
Linear function + translation invariance = convolution

- Local connectivity determines kernel size

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4</td>
<td>0.1</td>
<td>3.6</td>
</tr>
<tr>
<td>1.8</td>
<td>2.3</td>
<td>4.5</td>
</tr>
<tr>
<td>1.1</td>
<td>3.4</td>
<td>7.2</td>
</tr>
</tbody>
</table>
Convolution Layer

32x32x3 image -> preserve spatial structure

- Height: 32
- Width: 32
- Depth: 3
Convolution Layer

32x32x3 image

5x5x3 filter

**Convolve** the filter with the image i.e. “slide over the image spatially, computing dot products”
Convolution Layer

32x32x3 image

5x5x3 filter

Filters always extend the full depth of the input volume

Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”
Convolution Layer

32x32x3 image
5x5x3 filter $w$

1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image (i.e. $5\times5\times3 = 75$-dimensional dot product + bias)

$$w^T x + b$$
Convolution Layer

32x32x3 image
5x5x3 filter

convolve (slide) over all spatial locations

activation map

3 32
32

1 28
Convolution Layer

32x32x3 image
5x5x3 filter

consider a second, green filter

convolve (slide) over all spatial locations

activation maps
Convolution as a general layer

For example, if we had 6 5x5 filters, we’ll get 6 separate activation maps:

We stack these up to get a “new image” of size 28x28x6!
Convolutional Neural Networks

- Convolution layers interspersed with activation functions.
Convolution as a primitive
Convolution as a feature detector

- score at \((x,y)\) = dot product (filter, image patch at \((x,y)\))
- Response represents similarity between filter and image patch
Kernel sizes and padding
Kernel sizes and padding

• Valid convolution decreases size by \((k-1)/2\) on each side
  – Pad by \((k-1)/2\), or
  – Allow spatial dimensions to shrink.
torch.nn.Conv2d

- torch.nn.Conv2d(
  in_channels,  # channels in input feature map
  out_channels, # filters to learn (== channels in the output)
  kernel_size,  # size of each filter kernel
  stride=1,     # move this many pixels when sliding filter
  padding=0,    # pad the input by this much (can be tuple)
  dilation=1,
  groups=1,
  bias=True     # add a bias after convolution?
)
Convolutional Layers

• Feature maps (“hidden layers”, “activations”, etc.) are no longer column vectors but 3D blobs:
  – Input # 256x256x3
  – Conv2d(in: 3, out:10) # 255x255x10
  – Conv2d(in: 10, out:20) # 255x255x20
  – ...

Convolutional Layers

- Feature maps ("hidden layers", "activations", etc.) are no longer column vectors but 3D blobs:
  - Input # 256x256x3
  - Conv2d(in: 3, out:10) # 255x255x10
  - Conv2d(in: 10, out:20) # 254x254x20
  - ... this could get large quickly, and we ultimately need a vector that we can apply a linear classifier to.
Convoluotional Networks

- Feature maps ("hidden layers", "activations", etc.) are no longer column vectors but 3D blobs:
  - Input # 256x256x3
  - Conv2d(in: 3, out:10) # 255x255x10
  - Subsample (2x2)
  - Conv2d(in: 10, out:20) # 127x127x20
  - ...
  - Conv/subsample until 1x1xC
  - Or at some point, just unravel HxWxC into HWCx1 vector.
  - Then apply a linear classifier!
CNNs before they were cool: LeNet-5 [LeCun et al., 1998]

• Today’s architectures still look a lot like this!