CSCI 497P/597P: Computer Vision
Scott Wehrwein

Convolutional Neural Networks
Training Tricks and Architecture Innovations
Reading

• http://cs231n.github.io/convolutional-networks/
Announcements

• HW2 Out
  – Optional
  – Due Thursday night
  – Review in class Friday

• Today’s OH extended to a bit before 5

• I’ll extend OH tomorrow if there’s demand.
Goals

• Understand some of the common tricks and strategies for designing and training neural networks:
  – Batched training
  – Preprocessing / data augmentation
  – Momentum
  – Learning rate decay
  – Weight initialization and batch normalization
  – Ensembling
  – Dropout
Training CNNs

• Most of these things are practical heuristics that have been empirically discovered to work well:
  – Batched training
  – Preprocessing / data augmentation
  – Momentum
  – Learning rate decay
  – Weight initialization and batch normalization
  – Ensembling
  – Dropout
Data Augmentation

• When >1 million training images is not enough:
  – Randomly Flip, Scale, Crop, Rotate, Perturb brightness and color
  – Example:

```python
import torchvision.transforms as tvt
transforms = tvt.Compose([  
    tvt.Resize((224, 224)),
    tvt.ColorJitter(hue=.05, saturation=.05),
    tvt.RandomHorizontalFlip(),
    tvt.RandomRotation(20, resample=PIL.Image.BILINEAR)
])
```
Data Augmentation
Training CNNs

• Most of these things are practical heuristics that have been empirically discovered to work well:
  – Batched training
  – Preprocessing / data augmentation
  – Momentum
  – Learning rate decay
  – Weight initialization and batch normalization
  – Ensembling
  – Dropout
Weight Initialization

- Q: what happens when $W=$constant init is used?
- First idea: **Small random numbers**
  (gaussian with zero mean and 1e-2 standard deviation)

\[ W = 0.01 \times \text{np.random.randn}(D,H) \]
Weight Initialization

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 problems with deeper networks.
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.
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
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
Activations become zero!

What do the gradients look like?
Weight Initialization

\[ W = \text{np.random.randn}(\text{fan}_{\text{in}}, \text{fan}_{\text{out}}) / \sqrt{\frac{2}{\text{fan}_{\text{in}}}} \]

# fan_in = numel(input)
# fan_out = numel(output)
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 zero-mean unit-variance activations? just make them so."

consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, 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 zero-mean unit-variance activations? just make them so."

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

2. Normalize

\[ \hat{x}(k) = \frac{x(k) - \mathbf{E}[x(k)]}{\sqrt{\text{Var}[x(k)]}} \]
Problem: do we necessarily want a zero-mean unit-variance input?
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:

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

Details in the batch normalization paper:

- At test time, the answer shouldn’t depend on the batch:
  - Instead, use a global average (computed during training) of activation means and variances
Batch Normalization

**TL;DR:** Using batch normalization speeds up training and makes it less sensitive to weight initialization.

```python
torch.nn.BatchNorm2d(num_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

Applies Batch Normalization over a 4D input (a mini-batch of 2D inputs with additional channel dimension) as described in the paper *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*.

\[
y = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x]} + \epsilon} \ast \gamma + \beta
\]
Training CNNs

Most of these things are practical heuristics that have been empirically discovered to work well:

- Batched training
- Preprocessing / data augmentation
- Momentum
- Learning rate decay
- Weight initialization and batch normalization
- Ensembling
- Dropout
Model Ensembles

1. Train multiple independent models
2. At test time average their results (Take average of predicted probability distributions, then choose argmax)

Enjoy 2% extra performance

Why would this work?
• Using different random initializations results in training arriving at different local minima.
• Remarkable (empirical) fact: performance of each one is similar!
Model Ensembles: Tips and Tricks

Instead of training independent models, use multiple snapshots of a single model during training!

Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017
Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.
Model Ensembles: Tips and Tricks

Instead of training independent models, use multiple snapshots of a single model during training!

Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017
Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.

Cyclic learning rate schedules can make this work even better!
Training CNNs

• Most of these things are practical heuristics that have been empirically discovered to work well:
  – Batched training
  – Preprocessing / data augmentation
  – Momentum
  – Learning rate decay
  – Weight initialization and batch normalization
  – Ensembling
  – Dropout
Regularization: Recall

• Penalizes large weights to prevent the model from fitting training data *too* closely *(overfitting)*
  – Helps network generalize to unseen data

• L2 regularization forces parameters to be used “equally”
  – parameters with similar magnitudes will have a lower regularization cost than mostly zero with a few huge values.

• Another way to force the network to use all its parameters equally: randomly drop parameters each training iteration!
Another way to force the network to use all its parameters equally: **randomly drop parameters** each training iteration!

**Regularization: Dropout**

In each forward pass, randomly set some neurons to zero.

Probability of dropping is a hyperparameter; 0.5 is common.

---

Regularization: Dropout

$p = 0.5$ # 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)
```

Example forward pass with a 3-layer network using dropout
Regularization: Dropout
How can this possibly be a good idea?

Forces the network to have a redundant representation;
Prevents co-adaptation of features

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

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung
Regularization: Dropout

How can 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.

An FC layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!

Only $\sim 10^{82}$ atoms in the universe...
Dropout: Test time

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
Reinforcement Learning

**Vanilla Dropout:** Not recommended implementation (see notes below)

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

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
```

**Dropout Summary**

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

```
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!
Training CNNs

• Most of these things are practical heuristics that have been empirically discovered to work well:
  – Batched training
  – Preprocessing / data augmentation
  – Momentum
  – Learning rate decay
  – Weight initialization and batch normalization
  – Ensembling
  – Dropout
Next Up: CNN Architecture Tour

• What happened since AlexNet?
• There’s a general theme:

WE NEED TO GO DEEPER
Review: LeNet-5

[LeCun et al., 1998]

Conv filters were 5x5, applied at stride 1
Subsampling (Pooling) layers were 2x2 applied at stride 2
i.e. architecture is [CONV-POOL-CONV-POOL-FC-FC]
Case Study: AlexNet

[Krizhevsky et al. 2012]

Architecture:
CONV1
MAX POOL1
NORM1
CONV2
MAX POOL2
NORM2
CONV3
CONV4
CONV5
Max POOL3
FC6
FC7
FC8

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.
Case Study: AlexNet

[Krizhevsky et al. 2012]

Full (simplified) AlexNet architecture:

[227x227x3] INPUT

[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0

[27x27x96] MAX POOL1: 3x3 filters at stride 2

[27x27x96] NORM1: Normalization layer

[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2

[13x13x256] MAX POOL2: 3x3 filters at stride 2

[13x13x256] NORM2: Normalization layer

[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1

[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1

[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1

[6x6x256] MAX POOL3: 3x3 filters at stride 2

[4096] FC6: 4096 neurons

[4096] FC7: 4096 neurons

[1000] FC8: 1000 neurons (class scores)

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.
Case Study: AlexNet

Full (simplified) AlexNet architecture:

[227x227x3] INPUT

[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2

[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

Details/Retrospectives:
- first use of ReLU
- used Norm layers (not common anymore)
- heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 5e-4
- 7 CNN ensemble: 18.2% -> 15.4%

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.
Case Study: AlexNet

[Full (simplified) AlexNet architecture:]

[227x227x3] INPUT

[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

[55x55x48] x 2

Historical note: Trained on GTX 580 GPU with only 3 GB of memory. Network spread across 2 GPUs, half the neurons (feature maps) on each GPU.

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.
ImageNet Large Scale Visual Recognition Challenge (ILSVRRC) winners

2010: Lin et al
2011: Sanchez & Perronnin
2012: Krizhevsky et al (AlexNet)
2013: Zeiler & Fergus
2014: Simonyan & Zisserman (VGG)  Szegedy et al (GoogleNet)
2015: He et al (ResNet)
2016: Shao et al
2017: Hu et al (SENet)

First CNN-based winner: 2012

152 layers 152 layers 152 layers

Human: 5.1
Not deeper! Just better tuned

ZFNet

[Zeiler and Fergus, 2013]

ImageNet top 5 error: 16.4% -> 11.7%

AlexNet but:
CONV1: change from (11x11 stride 4) to (7x7 stride 2)
CONV3,4,5: instead of 384, 384, 256 filters use 512, 1024, 512

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung
WE NEED TO GO DEEPER
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

- 2010: Lin et al
- 2011: Sanchez & Perronnin
- 2012: Krizhevsky et al (AlexNet)
- 2013: Zeiler & Fergus
- 2014: Simonyan & Zisserman (VGG) (GoogLeNet)
- 2014: Szegedy et al
- 2015: He et al (ResNet)
- 2016: Shao et al
- 2017: Hu et al (SENet)

Deeper Networks:
- 152 layers in 2014: Simonyan & Zisserman (VGG) (GoogLeNet)
- 152 layers in 2014: Szegedy et al
- 152 layers in 2017: Hu et al (SENet)

Human: 5.1
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Small filters, Deeper networks

8 layers (AlexNet)
-> 16 - 19 layers (VGG16Net)

Only 3x3 CONV stride 1, pad 1
and 2x2 MAX POOL stride 2

11.7% top 5 error in ILSVRC’13
(ZFNet)
-> 7.3% top 5 error in ILSVRC’14
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)

Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer

Q: What is the effective receptive field of three 3x3 conv (stride 1) layers?
Case Study: VGGNet
[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)

Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer

But deeper, more non-linearities

And fewer parameters: $3 \times (3^2C^2)$ vs. $7^2C^2$ for C channels per layer
<table>
<thead>
<tr>
<th>Layer</th>
<th>Input Size</th>
<th>Memory</th>
<th>Params</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT</td>
<td>[224x224x3]</td>
<td>224<em>224</em>3=150K</td>
<td>0</td>
</tr>
<tr>
<td>CONV3-64</td>
<td>[224x224x64]</td>
<td>224<em>224</em>64=3.2M</td>
<td>(3*3)*64 = 1,728</td>
</tr>
<tr>
<td>CONV3-64</td>
<td>[224x224x64]</td>
<td>224<em>224</em>64=3.2M</td>
<td>(3*3)*64 = 36,864</td>
</tr>
<tr>
<td>POOL2</td>
<td>[112x112x64]</td>
<td>112<em>112</em>64=800K</td>
<td>0</td>
</tr>
<tr>
<td>CONV3-128</td>
<td>[112x112x128]</td>
<td>112<em>112</em>128=1.6M</td>
<td>(3*3)*128 = 73,728</td>
</tr>
<tr>
<td>CONV3-128</td>
<td>[112x112x128]</td>
<td>112<em>112</em>128=1.6M</td>
<td>(3*3)*128 = 147,456</td>
</tr>
<tr>
<td>POOL2</td>
<td>[56x56x128]</td>
<td>56<em>56</em>128=400K</td>
<td>0</td>
</tr>
<tr>
<td>CONV3-256</td>
<td>[56x56x256]</td>
<td>56<em>56</em>256=800K</td>
<td>(3*3)*256 = 294,912</td>
</tr>
<tr>
<td>CONV3-256</td>
<td>[56x56x256]</td>
<td>56<em>56</em>256=800K</td>
<td>(3*3)*256 = 589,824</td>
</tr>
<tr>
<td>POOL2</td>
<td>[28x28x256]</td>
<td>28<em>28</em>256=200K</td>
<td>0</td>
</tr>
<tr>
<td>CONV3-512</td>
<td>[28x28x512]</td>
<td>28<em>28</em>512=400K</td>
<td>(3*3)*512 = 1,179,648</td>
</tr>
<tr>
<td>CONV3-512</td>
<td>[28x28x512]</td>
<td>28<em>28</em>512=400K</td>
<td>(3*3)*512 = 2,359,296</td>
</tr>
<tr>
<td>POOL2</td>
<td>[14x14x512]</td>
<td>14<em>14</em>512=100K</td>
<td>0</td>
</tr>
<tr>
<td>CONV3-512</td>
<td>[14x14x512]</td>
<td>14<em>14</em>512=100K</td>
<td>(3*3)*512 = 2,359,296</td>
</tr>
<tr>
<td>CONV3-512</td>
<td>[14x14x512]</td>
<td>14<em>14</em>512=100K</td>
<td>(3*3)*512 = 2,359,296</td>
</tr>
<tr>
<td>POOL2</td>
<td>[7x7x512]</td>
<td>7<em>7</em>512=25K</td>
<td>0</td>
</tr>
<tr>
<td>FC: [1x1x4096]</td>
<td></td>
<td>4096</td>
<td>7<em>7</em>512*4096 = 102,760,448</td>
</tr>
<tr>
<td>FC: [1x1x4096]</td>
<td></td>
<td>4096</td>
<td>4096*4096 = 16,777,216</td>
</tr>
<tr>
<td>FC: [1x1x1000]</td>
<td></td>
<td>1000</td>
<td>4096*1000 = 4,096,000</td>
</tr>
</tbody>
</table>

TOTAL memory: 24M * 4 bytes ~ 96MB / image (for a forward pass)
TOTAL params: 138M parameters
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Details:
- ILSVRC’14 2nd in classification, 1st in localization
- Similar training procedure as Krizhevsky 2012
- No Local Response Normalisation (LRN)
- Use VGG16 or VGG19 (VGG19 only slightly better, more memory)
- Use ensembles for best results
- FC7 features generalize well to other tasks