Lecture 19:
Training CNNs, part 2
Overview

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

2. **Training dynamics**
   - starting the learning process, hyperparameter selection, parameter optimization, transfer learning

3. **Evaluation**
   - model ensembles
Overview

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

2. Training dynamics
   starting the learning process, hyperparameter
   selection, parameter optimization, transfer learning

3. Evaluation
   model ensembles
Gradient checking:

- Numerical gradient: approximate, slow, easy to write
- Analytic gradient: exact, fast, error-prone

=>

**In practice:** Always use analytic gradient, but check implementation with numerical gradient. This is called a gradient check.
Reminder: Activation functions

**Sigmoid**

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

**tanh**

$$\tanh(x)$$

**ReLU**

$$\max(0, x)$$

Good default choice

**Leaky ReLU**

$$\max(0.1x, x)$$

**Maxout**

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

**ELU**

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

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Before training: Preprocess the data

(Assume X [NxD] is data matrix, each example in a row)

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Xavier initialization

Reasonable initialization.
(Mathematical derivation assumes linear activations)
Overview

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

2. Training dynamics
   *starting the learning process, hyperparameter selection, parameter optimization, transfer learning*

3. Evaluation
   *model ensembles*
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[0.0])  # disable regularization
print loss
2.30261216167
```

- loss ~2.3.
- ‘correct ‘ for 10 classes
- returns the loss and the gradient for all parameters

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Let’s 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!
## Recall: setting hyperparameters

**Idea #1:** Choose hyperparameters that work best on the data

**BAD:** No idea how algorithm will perform on new data

### Your Dataset

| train | validation | test |

**Idea #2:** Split data into train and test, choose hyperparameters that work best on test data

**BAD:** No idea how algorithm will perform on new data

### train  | test

**Idea #3:** Split data into train, val, and test; choose hyperparameters on val and evaluate on test

**Good!**

### train  | validation  | test
Random Search vs. Grid Search

Grid Layout

Important Parameter

Unimportant Parameter

Random Layout

Important Parameter

Unimportant Parameter

Illustration of Bergstra et al., 2012 by Shayne Longpre, copyright CS231n 2017

Random Search for Hyper-Parameter Optimization
Bergstra and Bengio, 2012

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Monitor and visualize the loss curve

- **very high learning rate**
- **low learning rate**
- **good learning rate**

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
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 updates and values: \( \sim 0.0002 / 0.02 = 0.01 \) (about okay)
want this to be somewhere around 0.001 or so
Overview

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

2. Training dynamics
   - starting the learning process, hyperparameter selection, parameter optimization, transfer learning

3. Evaluation
   - model ensembles
# Vanilla Gradient Descent

```python
while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += - step_size * weights_grad  # perform parameter update
```
Optimization: Problems with SGD

What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Optimization: Problems with SGD

What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Optimization: Problems with SGD

What if the loss function has a local minima or saddle point?
Optimization: Problems with SGD

What if the loss function has a local minima or saddle point?

Zero gradient, gradient descent gets stuck
Optimization: Problems with SGD

What if the loss function has a local minima or saddle point?

Saddle points much more common in high dimension

Dauphin et al, “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization”, NeurIPS 2014
Optimization: Problems with SGD

Our gradients come from minibatches so they can be noisy!

\[
L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(x_i, y_i, W)
\]

\[
\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)
\]
SGD + Momentum

**SGD**

\[ x_{t+1} = x_t - \alpha \nabla f(x_t) \]

```
while True:
    dx = compute_gradient(x)
    x += learning_rate * dx
```

**SGD+Momentum**

\[ v_{t+1} = \rho v_t + \nabla f(x_t) \]
\[ x_{t+1} = x_t - \alpha v_{t+1} \]

```
vx = 0
while True:
    dx = compute_gradient(x)
    vx = rho * vx + dx
    x += learning_rate * vx
```

- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99
SGD + Momentum

Local Minima  Saddle points

Poor Conditioning

Gradient Noise

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
SGD + Momentum

Momentum update:

Gradient

Velocity

actual step
Nesterov Momentum

Momentum update:

- Velocity
- actual step
- Gradient

Nesterov Momentum

- Gradient
- Velocity
- actual step

Nesterov, “A method of solving a convex programming problem with convergence rate $O(1/k^2)$”, 1983
Nesterov, “Introductory lectures on convex optimization: a basic course”, 2004
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Nesterov Momentum

\[ v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t) \]

\[ x_{t+1} = x_t + v_{t+1} \]
Nesterov Momentum

\[ v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t) \]

\[ x_{t+1} = x_t + v_{t+1} \]

Annoying, usually we want update in terms of \( x_t, \nabla f(x_t) \)
Nesterov Momentum

\[ v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t) \]
\[ x_{t+1} = x_t + v_{t+1} \]

Annoying, usually we want update in terms of \( x_t, \nabla f(x_t) \)

Change of variables \( \tilde{x}_t = x_t + \rho v_t \) and rearrange:

\[ v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t) \]
\[ \tilde{x}_{t+1} = \tilde{x}_t - \rho v_t + (1 + \rho) v_{t+1} \]
\[ = \tilde{x}_t + v_{t+1} + \rho (v_{t+1} - v_t) \]

```
dx = compute_gradient(x)
old_v = v
v = rho * v - learning_rate * dx
x += -rho * old_v + (1 + rho) * v
```
Nesterov Momentum

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

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

Duchi et al, "Adaptive subgradient methods for online learning and stochastic optimization", JMLR 2011

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
RMSProp

AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Tieleman and Hinton, 2012

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Adam

- SGD
- SGD+Momentum
- RMSProp
- Adam

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.
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.

**exponential decay:**
\[
\alpha = \alpha_0 e^{-kt}
\]

**1/t decay:**
\[
\alpha = \alpha_0 / (1 + kt)
\]
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.

More critical with SGD+Momentum, less common with Adam
First-Order Optimization

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
First-Order Optimization

(1) Use gradient form linear approximation
(2) Step to minimize the approximation
Second-Order Optimization

(1) Use gradient and Hessian to form quadratic approximation
(2) Step to the minima of the approximation
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).
Beyond Training Error

Better optimization algorithms help reduce training loss

But we really care about error on new data - how to reduce the gap?
Overview

1. One time setup
   gradient checking, activation functions, data preprocessing, weight initialization, regularization
2. Training dynamics
   starting the learning process, hyperparameter selection, parameter optimization, transfer learning
3. Evaluation
   model ensembles
Model Ensembles

1. Train multiple independent models
2. At test time average their results

Enjoy 2% extra performance
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.

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
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!
Model Ensembles: Tips and Tricks

Instead of using actual parameter vector, keep a moving average of the parameter vector and use that at test time (Polyak averaging)

```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
```


Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
How to improve single-model performance?

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Overview

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

2. Training dynamics
   - starting the learning process, hyperparameter selection, parameter optimization, transfer learning

3. Evaluation
   - model ensembles
Regularization: Add term to loss

\[ L = \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \lambda R(W) \]

In common use:

**L2 regularization**

\[ R(W) = \sum_k \sum_l W_{k,l}^2 \] (Weight decay)

**L1 regularization**

\[ R(W) = \sum_k \sum_l |W_{k,l}| \]

**Elastic net (L1 + L2)**

\[ R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}| \]
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

Credit: 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

Dropout makes our output random!

\[ y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz \]

Want to “average out” the randomness at test-time

But this integral seems hard …
Dropout: Test time

Want to approximate the integral

\[ y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz \]

Consider a single neuron.
Dropout: Test time

Want to approximate the integral

$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

Consider a single neuron.

At test time we have:

$$E[a] = w_1x + w_2y$$
Dropout: Test time

Want to approximate the integral

\[ y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z) \, dz \]

Consider a single neuron.

At test time we have:

\[ E[a] = w_1 x + w_2 y \]

During training we have:

\[ E[a] = \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) = \frac{1}{2}(w_1 x + w_2 y) \]
Dropout: Test time

Want to approximate the integral

\[ y = f(x) = \mathbb{E}_z[f(x, z)] = \int p(z)f(x, z)dz \]

Consider a single neuron.

At test time we have:

\[ \mathbb{E}[a] = w_1x + w_2y \]

During training we have:

\[ \mathbb{E}[a] = \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \]

\[ = \frac{1}{2}(w_1x + w_2y) \]

At test time, multiply by dropout probability

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Dropout: Test time

```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

Dropout involves dropping units in the forward pass with probability \( p \) and scaling the remaining activations at test time.

Here is an example implementation in Python:

```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
```

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
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!
```
Regularization: A common pattern

**Training:** Add some kind of randomness

\[
y = f_W(x, z)
\]

**Testing:** Average out randomness (sometimes approximate)

\[
y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz
\]
Regularization: A common pattern

**Training:** Add some kind of randomness

\[ y = f_W(x, z) \]

**Testing:** Average out randomness (sometimes approximate)

\[ y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz \]

**Example:** Batch Normalization

**Training:** Normalize using stats from random minibatches

**Testing:** Use fixed stats to normalize

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Regularization: Data Augmentation

Load image and label

“cat”

CNN

Compute loss

This image by Nikita is licensed under CC-BY 2.0

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Regularization: Data Augmentation

Load image and label

“cat”

Transform image

CNN

Compute loss

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Data Augmentation
Horizontal Flips

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Data Augmentation
Random crops and scales

**Training**: sample random crops / scales

ResNet:
1. Pick random $L$ in range $[256, 480]$
2. Resize training image, short side = $L$
3. Sample random $224 \times 224$ patch
Data Augmentation
Random crops and scales

**Training:** sample random crops / scales
ResNet:
1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch

**Testing:** average a fixed set of crops
ResNet:
1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224 x 224 crops: 4 corners + center, + flips

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Data Augmentation
Color Jitter

Simple: Randomize contrast and brightness
Data Augmentation

Color Jitter

Simple: Randomize contrast and brightness

More Complex:

1. Apply PCA to all [R, G, B] pixels in training set

2. Sample a “color offset” along principal component directions

3. Add offset to all pixels of a training image

(As seen in [Krizhevsky et al. 2012], ResNet, etc)
Data Augmentation
Get creative for your problem!

Random mix/combinations of:
- translation
- rotation
- stretching
- shearing,
- lens distortions, … (go crazy)
Regularization: A common pattern

**Training:** Add random noise

**Testing:** Marginalize over the noise

**Examples:**
- Dropout
- Batch Normalization
- Data Augmentation

Ioffe and Szegedy. "Batch normalization: accelerating deep network training by reducing internal covariate shift", ICML 2015

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Overview

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

2. Training dynamics
   starting the learning process, hyperparameter selection, parameter optimization, transfer learning

3. Evaluation
   model ensembles
Transfer Learning with CNNs

1. Train on ImageNet

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung

Transfer Learning with CNNs

1. Train on Imagenet

<table>
<thead>
<tr>
<th>FC-1000</th>
<th>FC-4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxPool</td>
<td>Conv-512</td>
</tr>
<tr>
<td>Conv-512</td>
<td>Conv-512</td>
</tr>
<tr>
<td>MaxPool</td>
<td>Conv-256</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-256</td>
</tr>
<tr>
<td>MaxPool</td>
<td>Conv-128</td>
</tr>
<tr>
<td>Conv-128</td>
<td>Conv-128</td>
</tr>
<tr>
<td>MaxPool</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-64</td>
<td></td>
</tr>
</tbody>
</table>

2. Small Dataset (C classes)

<table>
<thead>
<tr>
<th>FC-C</th>
<th>FC-4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxPool</td>
<td>Conv-512</td>
</tr>
<tr>
<td>Conv-512</td>
<td>Conv-512</td>
</tr>
<tr>
<td>MaxPool</td>
<td>Conv-256</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-256</td>
</tr>
<tr>
<td>MaxPool</td>
<td>Conv-128</td>
</tr>
<tr>
<td>Conv-128</td>
<td>Conv-128</td>
</tr>
<tr>
<td>MaxPool</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-64</td>
<td></td>
</tr>
</tbody>
</table>

Reinitialize this and train

Freeze these

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung


Razavian et al., “CNN Features Off-the-Shelf: An Astounding Baseline for Recognition”, CVPR Workshops 2014
Transfer Learning with CNNs

1. Train on Imagenet
   - FC-1000
   - FC-4096
   - FC-4096

   MaxPool
   Conv-512
   Conv-512
   MaxPool
   Conv-256
   Conv-256
   MaxPool
   Conv-128
   Conv-128
   MaxPool
   Conv-64
   Conv-64
   Image

2. Small Dataset (C classes)
   - FC-C
   - FC-4096
   - FC-4096

   MaxPool
   Conv-512
   Conv-512
   MaxPool
   Conv-256
   Conv-256
   MaxPool
   Conv-128
   Conv-128
   MaxPool
   Conv-64
   Conv-64
   Image

   Reinitialize this and train

3. Bigger dataset
   - FC-C
   - FC-4096
   - FC-4096

   MaxPool
   Conv-512
   Conv-512
   MaxPool
   Conv-256
   Conv-256
   MaxPool
   Conv-128
   Conv-128
   MaxPool
   Conv-64
   Conv-64
   Image

   Train these

   With bigger dataset, train more layers

   Freeze these

   Lower learning rate when finetuning; 1/10 of original LR is good starting point


Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
More generic

More specific

<table>
<thead>
<tr>
<th></th>
<th>very similar dataset</th>
<th>very different dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>very little data</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>quite a lot of data</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>More generic</td>
<td>More specific</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
<td></td>
</tr>
<tr>
<td>FC-1000</td>
<td>FC-4096</td>
<td></td>
</tr>
<tr>
<td>FC-4096</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MaxPool</td>
<td>Conv-512</td>
<td></td>
</tr>
<tr>
<td>Conv-512</td>
<td>Conv-512</td>
<td></td>
</tr>
<tr>
<td>MaxPool</td>
<td>Conv-256</td>
<td></td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-256</td>
<td></td>
</tr>
<tr>
<td>MaxPool</td>
<td>Conv-128</td>
<td></td>
</tr>
<tr>
<td>Conv-128</td>
<td>Conv-128</td>
<td></td>
</tr>
<tr>
<td>MaxPool</td>
<td>Conv-64</td>
<td></td>
</tr>
<tr>
<td>Conv-64</td>
<td>Conv-64</td>
<td></td>
</tr>
<tr>
<td>Image</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>very similar dataset</th>
<th>very different dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>very little data</td>
<td>Use Linear Classifier on top layer</td>
</tr>
<tr>
<td>quite a lot of data</td>
<td>?</td>
</tr>
<tr>
<td>More specific</td>
<td>More generic</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------</td>
</tr>
<tr>
<td><strong>very little data</strong></td>
<td></td>
</tr>
<tr>
<td><strong>quite a lot of data</strong></td>
<td></td>
</tr>
</tbody>
</table>

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
<table>
<thead>
<tr>
<th>More specific</th>
<th></th>
<th>More generic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv-64</td>
<td>Conv-64</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-64</td>
<td>Conv-64</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-512</td>
<td>Conv-512</td>
<td>Conv-512</td>
</tr>
<tr>
<td>Conv-512</td>
<td>Conv-512</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-256</td>
<td>Conv-256</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-256</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-128</td>
<td>Conv-128</td>
<td>Conv-128</td>
</tr>
<tr>
<td>Conv-128</td>
<td>Conv-128</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-64</td>
<td>Conv-64</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-64</td>
<td>Conv-64</td>
<td>Image</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>very similar dataset</th>
<th>very different dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>very little data</td>
<td>Use Linear Classifier on top layer</td>
<td>?</td>
</tr>
<tr>
<td>quite a lot of data</td>
<td>Finetune a few layers</td>
<td>Finetune a larger number of layers</td>
</tr>
</tbody>
</table>

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
<table>
<thead>
<tr>
<th>More specific</th>
<th>More generic</th>
<th>very similar dataset</th>
<th>very different dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>very little data</strong></td>
<td>Use Linear Classifier on top layer</td>
<td>You’re in trouble… Try linear classifier from different stages</td>
<td></td>
</tr>
<tr>
<td><strong>quite a lot of data</strong></td>
<td>Finetune a few layers</td>
<td>Finetune a larger number of layers</td>
<td></td>
</tr>
</tbody>
</table>
Transfer learning with CNNs is pervasive…
(it’s the norm, not an exception)

Object Detection
(Fast R-CNN)

Image Captioning: CNN + RNN

Figure copyright Ross Girshick, 2015. Reproduced with permission.

Figure copyright IEEE, 2015. Reproduced for educational purposes.

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Transfer learning with CNNs is pervasive…
(it’s the norm, not an exception)

Object Detection
(Fast R-CNN)

CNN pretrained on ImageNet

Image Captioning: CNN + RNN

Figure copyright Ross Girshick, 2015. Reproduced with permission.

Figure copyright IEEE, 2015. Reproduced for educational purposes.
Transfer learning with CNNs is pervasive…
(it’s the norm, not an exception)

Object Detection
(Fast R-CNN)

CNN pretrained on ImageNet

Image Captioning: CNN + RNN

Word vectors pretrained with word2vec

Figure copyright Ross Girshick, 2015. Reproduced with permission.

Figure copyright IEEE, 2015. Reproduced for educational purposes.
Takeaway for your projects and beyond:
Have some dataset of interest but it has < ~1M images?

1. Find a very large dataset that has similar data, train a big ConvNet there
2. Transfer learn to your dataset

Deep learning frameworks provide a “Model Zoo” of pretrained models so you don’t need to train your own

Caffe: https://github.com/BVLC/caffe/wiki/Model-Zoo
TensorFlow: https://github.com/tensorflow/models
PyTorch: https://github.com/pytorch/vision
Summary

1. One time setup
   gradient checking: do
   activation functions: use ReLU
   data preprocessing: subtract mean of the image
   weight initialization: use Xavier init
   regularization: use L2+dropout+data augmentation

2. Training dynamics
   starting the learning process: lots of sanity-checks
   hyperparameter selection: random sample in log space
   parameter optimization: use Adam
   transfer learning: use freely

3. Evaluation
   model ensembles: simple 2% boost
CNN architectures
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]
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

152 layers

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung

Figure copyright Kaiming He, 2016. Reproduced with permission.
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

First CNN-based winner

152 layers

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
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.

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
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)
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)

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

[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)

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 (ILSVRC) winners

First CNN-based winner

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung

Figure copyright Kaiming He, 2016. Reproduced with permission.
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

ZFNet: Improved hyperparameters over AlexNet

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

Deeper Networks

<table>
<thead>
<tr>
<th>Year</th>
<th>Model</th>
<th>Layers</th>
<th>Top-5 Error Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010</td>
<td>ILSVRC'10</td>
<td>shallow</td>
<td>28.2</td>
</tr>
<tr>
<td>2011</td>
<td>ILSVRC'11</td>
<td>25.8</td>
<td></td>
</tr>
<tr>
<td>2012</td>
<td>ILSVRC'12</td>
<td>16.4</td>
<td></td>
</tr>
<tr>
<td>2013</td>
<td>ILSVRC'13</td>
<td>11.7</td>
<td></td>
</tr>
<tr>
<td>2014</td>
<td>GoogleNet</td>
<td>19 layers</td>
<td>7.3</td>
</tr>
<tr>
<td>2014</td>
<td>VGG</td>
<td>22 layers</td>
<td>6.7</td>
</tr>
<tr>
<td>2015</td>
<td>ResNet</td>
<td>152 layers</td>
<td>3.57</td>
</tr>
</tbody>
</table>

Figure copyright Kaiming He, 2016. Reproduced with permission.
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
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)
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.
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
INPUT: \([224 \times 224 \times 3]\) memory: 224*224*3=150K params: 0

<table>
<thead>
<tr>
<th>Layer Type</th>
<th>Output Size</th>
<th>Memory</th>
<th>Params</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONV3-64: ([224 \times 224 \times 64])</td>
<td>224<em>224</em>64=3.2M</td>
<td>(3<em>3</em>3)*64 = 1,728</td>
<td></td>
</tr>
<tr>
<td>CONV3-128: ([112 \times 112 \times 128])</td>
<td>112<em>112</em>128=1.6M</td>
<td>(3<em>3</em>64)*128 = 73,728</td>
<td></td>
</tr>
<tr>
<td>POOL2: ([56 \times 56 \times 128])</td>
<td>56<em>56</em>128=800K</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>CONV3-256: ([28 \times 28 \times 512])</td>
<td>28<em>28</em>512=400K</td>
<td>(3<em>3</em>512)*512 = 2,359,296</td>
<td></td>
</tr>
<tr>
<td>POOL2: ([7 \times 7 \times 512])</td>
<td>7<em>7</em>512=25K</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>FC: ([1 \times 1 \times 4096])</td>
<td>4096</td>
<td>7<em>7</em>512*4096 = 102,760,448</td>
<td></td>
</tr>
<tr>
<td>FC: ([1 \times 1 \times 4096])</td>
<td>4096</td>
<td>4096*4096 = 16,777,216</td>
<td></td>
</tr>
<tr>
<td>FC: ([1 \times 1 \times 1000])</td>
<td>1000</td>
<td>4096*1000 = 4,096,000</td>
<td></td>
</tr>
</tbody>
</table>

**TOTAL memory: 24M * 4 bytes ~ = 96MB / image (only forward! ~*2 for bwd)**

**TOTAL params: 138M parameters**

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
INPUT: [224 x 224 x 3] memory: 224 x 224 x 3 = 150K params: 0

CONV3-64: [224 x 224 x 64] memory: 224 x 224 x 64 = 3.2M params: 3 x 3 x 3 x 64 = 1,728

CONV3-64: [224 x 224 x 64] memory: 224 x 224 x 64 = 3.2M params: 3 x 3 x 64 x 64 = 36,864

POOL2: [112 x 112 x 64] memory: 112 x 112 x 64 = 800K params: 0

CONV3-128: [112 x 112 x 128] memory: 112 x 112 x 128 = 1.6M params: 3 x 3 x 64 x 128 = 73,728

CONV3-128: [112 x 112 x 128] memory: 112 x 112 x 128 = 1.6M params: 3 x 3 x 128 x 128 = 147,456

POOL2: [56 x 56 x 128] memory: 56 x 56 x 128 = 400K params: 0

CONV3-256: [56 x 56 x 256] memory: 56 x 56 x 256 = 800K params: 3 x 3 x 128 x 256 = 294,912

CONV3-256: [56 x 56 x 256] memory: 56 x 56 x 256 = 800K params: 3 x 3 x 256 x 256 = 589,824

CONV3-256: [56 x 56 x 256] memory: 56 x 56 x 256 = 800K params: 3 x 3 x 256 x 256 = 589,824

POOL2: [28 x 28 x 256] memory: 28 x 28 x 256 = 200K params: 0

CONV3-512: [28 x 28 x 512] memory: 28 x 28 x 512 = 400K params: 3 x 3 x 256 x 512 = 1,179,648

CONV3-512: [28 x 28 x 512] memory: 28 x 28 x 512 = 400K params: 3 x 3 x 512 x 512 = 2,359,296

CONV3-512: [28 x 28 x 512] memory: 28 x 28 x 512 = 400K params: 3 x 3 x 512 x 512 = 2,359,296

POOL2: [14 x 14 x 512] memory: 14 x 14 x 512 = 100K params: 0

CONV3-512: [14 x 14 x 512] memory: 14 x 14 x 512 = 100K params: 3 x 3 x 512 x 512 = 2,359,296

CONV3-512: [14 x 14 x 512] memory: 14 x 14 x 512 = 100K params: 3 x 3 x 512 x 512 = 2,359,296

CONV3-512: [14 x 14 x 512] memory: 14 x 14 x 512 = 100K params: 3 x 3 x 512 x 512 = 2,359,296

POOL2: [7 x 7 x 512] memory: 7 x 7 x 512 = 25K params: 0

FC: [1 x 1 x 4096] memory: 4096 params: 7 x 7 x 512 x 4096 = 102,760,448

FC: [1 x 1 x 4096] memory: 4096 params: 4096 x 4096 = 16,777,216

FC: [1 x 1 x 1000] memory: 1000 params: 4096 x 1000 = 4,096,000

TOTAL memory: 24M x 4 bytes ~ 96MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters
INPUT: [224x224x3] memory: 224*224*3=150K params: 0

CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*3)*64 = 1,728

CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*64)*64 = 36,864

POOL2: [112x112x64] memory: 112*112*64=800K params: 0

CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728

CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456

POOL2: [56x56x128] memory: 56*56*128=400K params: 0

CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912

CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824

CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824

POOL2: [28x28x256] memory: 28*28*256=200K params: 0

CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648

CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296

CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296

POOL2: [14x14x512] memory: 14*14*512=100K params: 0

CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296

CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296

CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296

POOL2: [7x7x512] memory: 7*7*512=25K params: 0

FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = 102,760,448

FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216

FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

TOTAL memory: 24M * 4 bytes \(\approx\) 96MB / image (only forward! \(\sim\)2 for bwd)

TOTAL params: 138M parameters
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

Deeper Networks

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

"Revolution of Depth"

Figure copyright Kaiming He, 2016. Reproduced with permission.
Case Study: ResNet

[He et al., 2015]

Very deep networks using residual connections

- 152-layer model for ImageNet
- ILSVRC’15 classification winner (3.57% top 5 error)
- Swept all classification and detection competitions in ILSVRC’15 and COCO’15!
Case Study: ResNet

[He et al., 2015]

What happens when we continue stacking deeper layers on a “plain” convolutional neural network?
Case Study: ResNet

[He et al., 2015]

What happens when we continue stacking deeper layers on a “plain” convolutional neural network?

Q: What’s strange about these training and test curves?
Case Study: ResNet

[He et al., 2015]

What happens when we continue stacking deeper layers on a “plain” convolutional neural network?

56-layer model performs worse on both training and test error
-> The deeper model performs worse, but it’s not caused by overfitting!
Case Study: ResNet

[He et al., 2015]

Hypothesis: the problem is an *optimization* problem, deeper models are harder to optimize

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Case Study: ResNet

[He et al., 2015]

Hypothesis: the problem is an *optimization* problem, deeper models are harder to optimize

The deeper model should be able to perform at least as well as the shallower model.

A solution by construction is copying the learned layers from the shallower model and setting additional layers to identity mapping.
Case Study: ResNet

[He et al., 2015]

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping.

"Plain" layers:

\[ H(x) \]

\[ \text{conv} \]

\[ \text{relu} \]

\[ \text{conv} \]

\[ X \]

Residual block:

\[ F(x) + x \]

\[ \text{relu} \]

\[ \text{conv} \]

\[ \text{identity} \]

\[ X \]
Case Study: ResNet

[He et al., 2015]

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping

\[ H(x) = F(x) + x \]

“Plain” layers

Residual block

\[ F(x) = H(x) - x \]

Use layers to fit residual instead of \( H(x) \) directly
Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:
- Stack residual blocks
- Every residual block has two 3x3 conv layers
Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:
- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)
Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:
- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)
- Additional conv layer at the beginning
Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:
- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)
- Additional conv layer at the beginning
- No FC layers at the end (only FC 1000 to output classes)

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Case Study: ResNet

[He et al., 2015]

Total depths of 34, 50, 101, or 152 layers for ImageNet
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Comparing complexity...


Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Comparing complexity...

Inception-v4: Resnet + Inception!


Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Comparing complexity...

VGG: Highest memory, most operations


Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Comparing complexity...


Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Comparing complexity...

**AlexNet:**
Smaller compute, still memory heavy, lower accuracy


Comparing complexity...

ResNet:
Moderate efficiency depending on model, highest accuracy


Credit: Fei-Fei Li & Justin Johnson & Serena Yeung