Lecture 7: Training Neural Networks, Part 2
Administrative

- Assignment 1 is being graded, stay tuned
- Project proposals due today by 11:59pm
- Assignment 2 is out, due Thursday May 4 at 11:59pm
Administrative: Google Cloud

- STOP YOUR INSTANCES when not in use!
Administrative: Google Cloud

- STOP YOUR INSTANCES when not in use!
- Keep track of your spending!
- GPU instances are much more expensive than CPU instances - only use GPU instance when you need it (e.g. for A2 only on TensorFlow / PyTorch notebooks)
Last time: Activation Functions

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

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

ReLU
\[ \text{max}(0, x) \]

Leaky ReLU
\[ \text{max}(0.1x, x) \]

Maxout
\[ \text{max}(w_1^T x + b_1, w_2^T x + b_2) \]

ELU
\[
\begin{align*}
    x & \quad x \geq 0 \\
    \alpha(e^x - 1) & \quad x < 0
\end{align*}
\]
Last time: 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**
\[ \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases} \]
Last time: Weight Initialization

**Initialization too small:**
Activations go to zero, gradients also zero,
No learning

**Initialization too big:**
Activations saturate (for tanh),
Gradients zero, no learning

**Initialization just right:**
Nice distribution of activations at all layers,
Learning proceeds nicely
Last time: Data Preprocessing
Last time: Data Preprocessing

**Before normalization**: classification loss very sensitive to changes in weight matrix; hard to optimize

**After normalization**: less sensitive to small changes in weights; easier to optimize
Last time: Batch Normalization

Input: \( x : N \times D \)

Learnable params:
\( \gamma, \beta : D \)

Intermediates:
\( \mu, \sigma : D \)
\( \hat{x} : N \times D \)

Output: \( y : N \times D \)

\[
\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}
\]

\[
\sigma^2_j = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2
\]

\[
\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma^2_j + \varepsilon}}
\]

\[
y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j
\]
Last time: Babysitting Learning
Last time: Hyperparameter Search

Grid Layout

Random Layout

Coarse to fine search

Important Parameter

Unimportant Parameter

Important Parameter

Unimportant Parameter

Fei-Fei Li & Justin Johnson & Serena Yeung

Lecture 7 - April 25, 2017
Today

- Fancier optimization
- Regularization
- Transfer Learning
# 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?

Loss function has high condition number: ratio of largest to smallest singular value of the Hessian matrix is large.
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.

Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large.
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”, NIPS 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**

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

**SGD+Momentum**

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

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

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
SGD + Momentum

Momentum update:

Velocity

actual step

Gradient
Nesterov Momentum

Momentum update:

Nesterov Momentum

Gradient

Velocity

actual step

Velocity

actual step

Gradient

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
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

\[
\begin{align*}
\nu_{t+1} &= \rho \nu_t - \alpha \nabla f(x_t + \rho \nu_t) \\
 x_{t+1} &= x_t + \nu_{t+1}
\end{align*}
\]

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:

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

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

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

- SGD
- SGD+Momentum
- Nesterov
AdaGrad

```python
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
AdaGrad

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

Q: What happens with AdaGrad?
AdaGrad

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

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

AdaGrad

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

RMSProp

```python
g = 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
RMSProp

- SGD
- SGD+Momentum
- RMSProp
Adam (almost)

```python
first_moment = 0
second_moment = 0
while True:
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7)
```

Adam (almost)

```python
firstMoment = 0
secondMoment = 0
while True:
    dx = compute_gradient(x)
    firstMoment = beta1 * firstMoment + (1 - beta1) * dx
    secondMoment = beta2 * secondMoment + (1 - beta2) * dx * dx
    x -= learning_rate * firstMoment / (np.sqrt(secondMoment) + 1e-7))
```

Momentum

AdaGrad / RMSProp

Sort of like RMSProp with momentum

Q: What happens at first timestep?

Adam (full form)

```
firstMoment = 0
secondMoment = 0
for t in range(num_iterations):
    dx = compute_gradient(x)
    firstMoment = beta1 * firstMoment + (1 - beta1) * dx
    secondMoment = beta2 * secondMoment + (1 - beta2) * dx * dx
    first_unbias = firstMoment / (1 - beta1 ** t)
    second_unbias = secondMoment / (1 - beta2 ** t)
    x = learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7)
```

Bias correction for the fact that first and second moment estimates start at zero

Momentum

AdaGrad / RMSProp

Bias correction

Adam (full form)

```python
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

- **Momentum**
- **Bias correction**
- **AdaGrad / RMSProp**

Bias correction for the fact that first and second moment estimates start at zero

Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!

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

Loss vs. $w_1$
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
Second-Order Optimization

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

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?

No hyperparameters!
No learning rate!
Second-Order Optimization

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:

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

Hessian has \( O(N^2) \) elements
Inverting takes \( O(N^3) \)
\( N = ( \text{Tens or Hundreds of} ) \text{ Millions} \)

Q2: Why is this bad for deep learning?
Second-Order Optimization

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

- **L-BFGS** (Limited memory BFGS):
  *Does not form/store the full inverse Hessian.*
Second-Order Optimization

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

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

- **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.

Le et al, "On optimization methods for deep learning, ICML 2011"
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?
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.
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
```

How to improve single-model performance?

Regularization
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**

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

**Elastic net (L1 + L2)**

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

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)
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

X

X

X

cat score

X
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 $2^{4096} \approx 10^{1233}$ possible masks!
Only $\approx 10^{82}$ atoms in the universe...
Dropout: Test time

Dropout makes our output random!

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

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:

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

\[
\begin{align*}
E[a] &= \frac{1}{4} (w_1 x + w_2 y) + \frac{1}{4} (w_1 x + 0y) \\
&\quad + \frac{1}{4} (0x + 0y) + \frac{1}{4} (0x + w_2 y) \\
&= \frac{1}{2} (w_1 x + w_2 y)
\end{align*}
\]
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:

\[ E[a] = \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \]

\[ = \frac{1}{2}(w_1 x + w_2 y) \]

At test time, multiply by dropout probability.
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
Vanilla Dropout: Not recommended implementation (see notes below)

\[ p = 0.5 \] # 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)
```

Dropout Summary

- **Dropout in Forward Pass**: Randomly set activations to zero with probability `p`.
- **Scale at Test Time**: Scale activations by `1/p` at test time to compensate for the dropout.

```
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
```
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!
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
Regularization: Data Augmentation

Load image and label

“cat”

CNN

Compute loss

This image by Nikita is licensed under CC-BY 2.0
Regularization: Data Augmentation

Load image and label → “cat”

Transform image → CNN → Compute loss
Data Augmentation
Horizontal Flips
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
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
Regularization: A common pattern

**Training:** Add random noise

**Testing:** Marginalize over the noise

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

Wan et al, "Regularization of Neural Networks using DropConnect", ICML 2013
Regularization: A common pattern

**Training**: Add random noise  
**Testing**: Marginalize over the noise

**Examples**:  
Dropout  
Batch Normalization  
Data Augmentation  
DropConnect  
Fractional Max Pooling

Graham, "Fractional Max Pooling", arXiv 2014
Regularization: A common pattern

**Training:** Add random noise

**Testing:** Marginalize over the noise

**Examples:**
- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect
- Fractional Max Pooling
- Stochastic Depth

Transfer Learning

“You need a lot of data if you want to train/use CNNs”
Transfer Learning

“You need a lot of data if you want to train/use CNNs” BUSTED
Transfer Learning with CNNs

1. Train on Imagenet

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

1. Train on Imagenet

2. Small Dataset (C classes)

- Reinitialize this and train
- Freeze these
Transfer Learning with CNNs

1. Train on Imagenet
   - FC-1000
   - FC-4096
   - FC-4096
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-128
   - Conv-128
   - MaxPool
   - Conv-256
   - Conv-256
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-64
   - Conv-64
   - Conv-64
   - Image

2. Small Dataset (C classes)
   - Reinitialize this and train
   - FC-C
   - FC-4096
   - FC-4096
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-128
   - Conv-128
   - MaxPool
   - Conv-256
   - Conv-256
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-64
   - Conv-64
   - Conv-64
   - Image

3. Bigger dataset
   - Train these
   - With bigger dataset, train more layers
   - Freeze these
   - FC-C
   - FC-4096
   - FC-4096
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-128
   - Conv-128
   - MaxPool
   - Conv-256
   - Conv-256
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-64
   - Conv-64
   - Conv-64
   - Image

Razavian et al., “CNN Features Off-the-Shelf: An Astounding Baseline for Recognition”, CVPR Workshops 2014
<table>
<thead>
<tr>
<th>More generic</th>
<th>More specific</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Image</strong></td>
<td><strong>Conv-64</strong></td>
</tr>
<tr>
<td><strong>Conv-64</strong></td>
<td><strong>Conv-512</strong></td>
</tr>
<tr>
<td><strong>MaxPool</strong></td>
<td><strong>MaxPool</strong></td>
</tr>
<tr>
<td><strong>Conv-512</strong></td>
<td><strong>Conv-512</strong></td>
</tr>
<tr>
<td><strong>Conv-512</strong></td>
<td><strong>Conv-256</strong></td>
</tr>
<tr>
<td><strong>MaxPool</strong></td>
<td><strong>MaxPool</strong></td>
</tr>
<tr>
<td><strong>Conv-256</strong></td>
<td><strong>Conv-256</strong></td>
</tr>
<tr>
<td><strong>Conv-256</strong></td>
<td><strong>Conv-128</strong></td>
</tr>
<tr>
<td><strong>MaxPool</strong></td>
<td><strong>MaxPool</strong></td>
</tr>
<tr>
<td><strong>Conv-128</strong></td>
<td><strong>Conv-128</strong></td>
</tr>
<tr>
<td><strong>MaxPool</strong></td>
<td><strong>MaxPool</strong></td>
</tr>
<tr>
<td><strong>Conv-128</strong></td>
<td><strong>Conv-128</strong></td>
</tr>
<tr>
<td><strong>MaxPool</strong></td>
<td><strong>MaxPool</strong></td>
</tr>
<tr>
<td><strong>Conv-64</strong></td>
<td><strong>Conv-64</strong></td>
</tr>
<tr>
<td><strong>Conv-64</strong></td>
<td><strong>Conv-64</strong></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>More generic</th>
<th>More specific</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FC-1000</strong></td>
<td><strong>Conv-64</strong></td>
</tr>
<tr>
<td><strong>FC-4096</strong></td>
<td><strong>Conv-64</strong></td>
</tr>
<tr>
<td><strong>FC-4096</strong></td>
<td><strong>MaxPool</strong></td>
</tr>
<tr>
<td><strong>Conv-512</strong></td>
<td><strong>Conv-64</strong></td>
</tr>
<tr>
<td><strong>Conv-512</strong></td>
<td><strong>Conv-64</strong></td>
</tr>
<tr>
<td><strong>Conv-512</strong></td>
<td><strong>MaxPool</strong></td>
</tr>
<tr>
<td><strong>MaxPool</strong></td>
<td><strong>Conv-512</strong></td>
</tr>
<tr>
<td><strong>Conv-512</strong></td>
<td><strong>Conv-512</strong></td>
</tr>
<tr>
<td><strong>MaxPool</strong></td>
<td><strong>MaxPool</strong></td>
</tr>
<tr>
<td><strong>Conv-256</strong></td>
<td><strong>Conv-256</strong></td>
</tr>
<tr>
<td><strong>Conv-256</strong></td>
<td><strong>Conv-256</strong></td>
</tr>
<tr>
<td><strong>MaxPool</strong></td>
<td><strong>MaxPool</strong></td>
</tr>
<tr>
<td><strong>Conv-128</strong></td>
<td><strong>Conv-128</strong></td>
</tr>
<tr>
<td><strong>Conv-128</strong></td>
<td></td>
</tr>
<tr>
<td><strong>MaxPool</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Conv-64</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Conv-64</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Image</strong></td>
<td></td>
</tr>
</tbody>
</table>

### Use Cases:

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td><strong>quite a lot of data</strong></td>
<td>Finetune a few layers</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.
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
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

- Optimization
  - Momentum, RMSProp, Adam, etc
- Regularization
  - Dropout, etc
- Transfer learning
  - Use this for your projects!
Next time: Deep Learning Software!