Topics:
• Training Neural Networks (Part 3)
Administrative

- Project Proposal deadline today! **No grace period**
- No class next Tue (10/03)
- HW2/PS2 due next Thu (10/05) + 48hr grace period
Recap: “Xavier” Initialization

```
dims = [4096] * 7  # “Xavier” initialization: std = 1/sqrt(Din)
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

“Just right”: Activations are nicely scaled for all layers!

For conv layers, Din is

```
filter_size^2 * input_channels
```

Let: \( y = x_1 w_1 + x_2 w_2 + \ldots + x_{Din} w_{Din} \)

Assume: \( \text{Var}(x_1) = \text{Var}(x_2) = \ldots = \text{Var}(x_{Din}) \)

We want: \( \text{Var}(y) = \text{Var}(x_i) \)

\[
\text{Var}(y) = \text{Var}(x_1 w_1 + x_2 w_2 + \ldots + x_{Din} w_{Din})
= \text{Din} \text{Var}(x_i w_i)
= \text{Din} \text{Var}(x_i) \text{Var}(w_i)
\]

[Assume all \( x_i, w_i \) are iid]

So, \( \text{Var}(y) = \text{Var}(x_i) \) only when \( \text{Var}(w_i) = 1/\text{Din} \)

Scaling a normal distribution (std=1) to have \( \text{Var}=1/\text{Din} \) -> multiply by \( \sqrt{1/\text{Din}} \)
Recap: Kaiming / MSRA Initialization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) * np.sqrt(2/Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

ReLU correction: \( \text{std} = \sqrt{\frac{2}{\text{Din}}} \)

*Issue:* Half of the activation get killed.

*Solution:* make the non-zero output variance twice as large as input

He et al, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification", ICCV 2015

Visualize distribution of activations
Recap: Batch Normalization

“you want zero-mean unit-variance activations? just make them so.”

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

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

Learnable scale and shift parameters:
\( \gamma, \beta : \mathbb{R}^D \)

We want to give the model a chance to adjust batchnorm if the default is not optimal.
Learning \( \gamma = \sigma \) and \( \beta = \mu \) will recover the identity function!

\[
\begin{align*}
\mu_j &= \frac{1}{N} \sum_{i=1}^{N} x_{i,j} & \text{Per-channel mean, shape is D} \\
\sigma_j^2 &= \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2 & \text{Per-channel var, shape is D} \\
\hat{x}_{i,j} &= \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} & \text{Normalized x, Shape is N x D} \\
y_{i,j} &= \gamma_j \hat{x}_{i,j} + \beta_j & \text{Output, Shape is N x D}
\end{align*}
\]

[Ioffe and Szegedy, 2015]
Batch Normalization: Test-Time

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

Learnable scale and shift parameters: \( \gamma, \beta : \mathbb{R}^D \)

\[
\begin{align*}
\mu_j &= \text{(Moving) average of values seen during training} \\
\sigma_j^2 &= \text{(Moving) average of values seen during training} \\
\hat{x}_{i,j} &= \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} \\
y_{i,j} &= \gamma_j \hat{x}_{i,j} + \beta_j
\end{align*}
\]

Per-channel mean, shape is D
Per-channel var, shape is D
Normalized x, Shape is N x D
Output, Shape is N x D

During testing batchnorm becomes a linear operator!
Can be fused with the previous fully-connected or conv layer
Batch Normalization

- Makes deep networks much easier to train!
  - If you are interested in the theory, read https://arxiv.org/abs/1805.11604
  - TL;DR: makes optimization landscape smoother
- Allows higher learning rates, faster convergence
- More useful in deeper networks
- Networks become more robust to initialization
- Zero overhead at test-time: can be fused with conv!
- Behaves differently during training and testing: this is a very common source of bugs!
- Needs large batch size to calculate accurate stats
Group Normalization

Wu and He, “Group Normalization”, ECCV 2018
SGD + Momentum

Intuitions:
- Think of a ball (set of parameters) moving in space (loss landscape), with momentum keeping it going in a direction.
- Individual gradient step may be noisy, the general trend accumulated over a few steps will point to the right direction.
- Momentum can “push” the ball over saddle points or local minima.
SGD + Momentum:

continue moving in the general direction as the previous iterations

*SGD*

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

**SGD+Momentum**

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

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

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

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013
Nesterov Momentum

Momentum update:

Combine gradient at current point with velocity to get step used to update weights

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

“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction
Optimization: Problem #3 with SGD

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

Long, narrow ravines:

Loss function has high condition number: ratio of largest to smallest eigenvalue \( \frac{\lambda_{\text{max}}}{\lambda_{\text{min}}} \) of the Hessian matrix of a loss function is large
Small condition number in loss Hessian -> circular contour
Large condition number in loss Hessian -> skewed contour

Can we enable SGD to adapt to this skew-ness?

AdaGrad

Q2: What happens to the step size over long time?  Decays to zero
RMSProp: “Leaky AdaGrad”

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)
```
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))
```

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

Adam with $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\text{learning\_rate} = 1e-3$ or $5e-4$ is a great starting point for many models!

Adam

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

Q: Which one of these learning rates is best to use?

A: In reality, all of these are good learning rates.

Need finer adjustment closer to convergence, so we want to reduce learning rate over time to keep making progress.
Learning rate decays over time

**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.
Learning Rate Decay

**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:** \[ \alpha_t = \frac{1}{2} \alpha_0 \left( 1 + \cos \left( t \pi / T \right) \right) \]

- \( \alpha_0 \) : Initial learning rate
- \( \alpha_t \) : Learning rate at epoch \( t \)
- \( T \) : Total number of epochs

Loshchilov and Hutter, “SGDR: Stochastic Gradient Descent with Warm Restarts”, ICLR 2017
Radford et al, “Improving Language Understanding by Generative Pre-Training”, 2018
Feichtenhofer et al, “SlowFast Networks for Video Recognition”, arXiv 2018
Learning Rate Decay

**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:**

\[ \alpha_t = \frac{1}{2} \alpha_0 (1 + \cos\left(\frac{t\pi}{T}\right)) \]

- \( \alpha_0 \): Initial learning rate
- \( \alpha_t \): Learning rate at epoch \( t \)
- \( T \): Total number of epochs

Loshchilov and Hutter, “SGDR: Stochastic Gradient Descent with Warm Restarts”, ICLR 2017
Radford et al, “Improving Language Understanding by Generative Pre-Training”, 2018
Feichtenhofer et al, “SlowFast Networks for Video Recognition”, arXiv 2018
Learning Rate Decay

**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:** \[ \alpha_t = \frac{1}{2} \alpha_0 \left(1 + \cos\left(t\pi/T\right)\right) \]

**Linear:** \[ \alpha_t = \alpha_0 \left(1 - t/T\right) \]

\( \alpha_0 \): Initial learning rate
\( \alpha_t \): Learning rate at epoch t
\( T \): Total number of epochs

Devlin et al, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", 2018
Learning Rate Decay

**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

\[ \alpha_t = \frac{1}{2} \alpha_0 \left(1 + \cos \left(\frac{t\pi}{T}\right)\right) \]

**Cosine:**

\[ \alpha_t = \alpha_0 \left(1 - \frac{t}{T}\right) \]

**Linear:**

\[ \alpha_t = \alpha_0 / \sqrt{t} \]

**Inverse sqrt:**

- \( \alpha_0 \): Initial learning rate
- \( \alpha_t \): Learning rate at epoch t
- \( T \): Total number of epochs

Vaswani et al, “Attention is all you need”, NIPS 2017
First-Order Optimization
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 of $f(x)$ at $a$:

$$f(x) = f(a) + \frac{f'(a)}{1!} (x - a) + \frac{f''(a)}{2!} (x - a)^2$$

Newton’s method for optimization: solving for the critical point $f'(x) = 0$, we obtain the Newton update rule

$$f'(x) = f'(a) + f''(a)(x - a) = 0$$
$$x^* = a - \frac{1}{f''(a)} f'(a)$$

Think of $a$ as the current params, $x^*$ as the updated params
Second-Order Optimization (multivariate)

Second-order Taylor Expansion of $f(x)$ at $a$:

$$
f(w) = f(a) + (x - a)^T \nabla f + \frac{1}{2} (x - a)^T H(x - a)
$$

Newton’s method for optimization: solving for the critical point we obtain the Newton update rule:

$$\mathbf{x}^* = a - H^{-1} \nabla f$$
Second-Order Optimization (multivariate)

second-order Taylor Expansion of $f(x)$ at $a$:

$$f(w) = f(a) + (x - a)^T \nabla f + \frac{1}{2} (x - a)^T H(x - a)$$

Newton’s method for optimization: solving for the critical point we obtain the Newton update rule:

$$x^* = a - H^{-1} \nabla f$$

Q: Why is this bad for deep learning?
Hessian Matrix

$$H_f = \begin{bmatrix}
\frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \ldots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\
\frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \ldots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \ldots & \frac{\partial^2 f}{\partial x_n^2}
\end{bmatrix}$$
Second-Order Optimization

second-order Taylor expansion:

\[ f(x) = f(a) + (x - a)^T \nabla f + \frac{1}{2} (x - a)^T H (x - a) \]

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

\[ x^* = a - H^{-1} \nabla f \]

Hessian has \( O(N^2) \) elements
Inverting takes \( O(N^3) \)
\( N = \) Millions

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

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

- 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 second-order methods to large-scale, stochastic setting is an active area of research.

Ba et al, "Distributed second-order optimization using Kronecker-factored approximations", ICLR 2017
This Time:

**Training** Deep Neural Networks

- Details of the non-linear activation functions
- Data normalization
- Weight Initialization
- Batch Normalization
- Advanced Optimization
- Regularization
- Data Augmentation
- Transfer learning
- Hyperparameter Tuning
Regularization
Beyond Training Error

Better optimization algorithms help reduce training loss

But we really care about error on new data - how to reduce the gap?
Early Stopping: Always do this

Stop training the model when accuracy on the validation set decreases
Or train for a long time, but always keep track of the model snapshot that worked best on val
How to improve generalization?

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

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

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

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

Cat score
Regularization: Dropout
How can this possibly be a good idea?

Another interpretation:

Dropout is training a large \textit{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!

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

\[ y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz \]
Dropout: Test time

Compute the expectation

Consider a single neuron.

\[ y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz \]
Dropout: Test time

Consider a single neuron.

Without dropout: \[ E[a] = w_1 x + w_2 y \]
Consider a single neuron.

Without dropout: \( E[a] = w_1 x + w_2 y \)

With dropout 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

Consider a single neuron.

Without dropout:

\[ E[a] = w_1x + w_2y \]

With dropout we have:

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

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

At test time, **multiply** by dropout probability

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

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 train time**: The dropout masks are applied during the forward pass in the `train_step` function. This involves multiplying the activations by random masks generated from a uniform distribution.

- **Scale at test time**: The activations are scaled by the dropout probability `p` during the `predict` function. This scaling is a practical way to maintain the expected value of the activations, which is crucial for the model’s performance.

---

Note: The use of vanilla dropout is discouraged due to its limitations in reproducibility and gradient masking. Modern implementations often use stochastic depth or other techniques to mitigate these issues.
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!

Similar to BatchNorm, different behavior train vs test!
Regularization: A common strategy

**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: 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 x 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 (test-time augmentation)**:
take votes / average from a fixed set of crops
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
3. Make prediction on all crops, use the majority vote as the final output.
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

**Testing (deterministic)**:
- Take a center crop of $224 \times 224$.
- Or crop by longer dimension and resize.
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

1. 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!

Examples of data augmentations:
- translation
- rotation
- stretching
- shearing,
- chromatic aberration
- lens distortions, … (go crazy)
Automatic Data Augmentation

Cubuk et al., "AutoAugment: Learning Augmentation Strategies from Data", CVPR 2019
Gradient clipping: prevent large gradient step

Large gradient step will likely destabilize training (gradients are noisy!)
Large gradient update can be caused by many issues, e.g., large weights, large input, bad loss function / activation function, ...
Should always first try to fix the root cause (normalization, better loss / activation function, etc.)

But if all things fail ... just clip the gradient

\[ g_{\text{new}} = \min \left( 1, \frac{\lambda}{\| g \|} \right) \times g \]

\( g \): original gradient
\( \lambda \): clipping threshold

```python
# Zero the gradients.
optimizer.zero_grad()

# Perform forward pass.
outputs = model(inputs)

# Compute the loss.
loss = loss_function(outputs, targets)

# Perform backward pass (compute gradients).
loss.backward()

# Clip the gradients.
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

# Update the model parameters.
optimizer.step()
```
Transfer learning / Pretraining
“You need a lot of data if you want to train/use deep neural networks”
“You need a lot of data if you want to train/use deep neural networks”
Transfer Learning with CNNs
Transfer Learning with CNNs

AlexNet:
64 x 3 x 11 x 11

(More on this in Lecture 13)
Transfer Learning with CNNs

Test image  L2 Nearest neighbors in feature space

(More on this in Lecture 13)
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

   - FC-1000
   - FC-4096
   - FC-4096
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-256
   - Conv-256
   - 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-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-256
   - Conv-256
   - MaxPool
   - Conv-256
   - Conv-256
   - MaxPool
   - Conv-128
   - Conv-128
   - MaxPool
   - Conv-64
   - Conv-64
   - Image

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

1. Train on Imagenet

2. Small Dataset (C classes)


Transfer Learning with CNNs

1. Train on Imagenet

2. Small Dataset (C classes)

3. Bigger dataset

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

With bigger dataset, train more layers

Freeze these

Train these

Freeze these

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

Reinitialize this and train

Download image
<table>
<thead>
<tr>
<th>Task-agnostic</th>
<th>Task-specific</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>very little data</strong></td>
<td>?</td>
</tr>
<tr>
<td><strong>quite a lot of data</strong></td>
<td>?</td>
</tr>
<tr>
<td>Task-agnostic</td>
<td>very similar dataset</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------</td>
</tr>
<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>
<tr>
<td>Task</td>
<td>very similar dataset</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>very little data</td>
<td>Use Linear Classifier on top layer</td>
</tr>
<tr>
<td>quite a lot of data</td>
<td>Finetune a few layers</td>
</tr>
</tbody>
</table>

**Diagram:**
- **Task-specific**
  - FC-1000
  - FC-4096
  - Task
- **Task-agnostic**
  - Conv-64
  - Conv-128
  - Conv-256
  - Conv-512
  - MaxPool
  - Image
Transfer learning 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 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 is pervasive…
(it’s the norm, not an exception)

Object Detection
(Fast R-CNN)

Image Captioning: CNN + RNN

CNN pretrained on ImageNet

Word vectors pretrained with word2vec

Figure copyright Ross Girshick, 2015. Reproduced with permission.

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

Pre-training
MLM on unlabelled data

- word2vec
- GloVe
- skip-thought
- InferSent
- ELMo
- ULMFiT
- GPT
- BERT

Fine-tuning
Cross-entropy on task labels

- classification
- sequence labeling
- Q&A
- ....

Generic Language Model
Train with Task-specific Labels

https://ruder.io/recent-advances-lm-fine-tuning/
Preview: Pretrained Language Models

Devlin et al. in BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, 2019

https://huggingface.co/blog/rlhf
Preview: Self-Supervised Pretraining (pretraining tasks that do not need labels)

Example: learn to predict image transformations / complete corrupted images

1. Solving the pretext tasks allow the model to learn good features.
2. We can automatically generate labels for the pretext tasks.
Problem: finetuning still takes a lot of data, especially if the model is huge and/or the domain gap is large.

Fact: finetuning is just adding a $W_\delta$ to the existing weight matrix $W$, i.e., $W^* = W + W_\delta$

Hypothesis: $W_\delta$ is low-rank, meaning that $W_\delta$ can be decomposed into two smaller matrices $A$ and $B$, i.e., $W_\delta = A^T B$.

So what?: $A$ and $B$ have a lot fewer parameters than the full $W$. Requires less data and faster to train.

---

Takeaway for your projects and beyond:

Transfer learning be like

Source: AI & Deep Learning Memes For Back-propagated Poets
Takeaway for your projects and beyond:
Have some dataset of interest but not big enough to train deep models?

1. Find a very large dataset that has similar data, train a big model there
2. Transfer learn to your dataset
3. Try LORA (low-rank finetuning) if necessary

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

TensorFlow: https://github.com/tensorflow/models
PyTorch (Vision): https://github.com/pytorch/vision
PyTorch (NLP): https://github.com/pytorch/text
Diagnose your training
(without tons of GPUs)
Diagnose your training

**Step 1:** Check initial loss

Turn off weight decay, sanity check loss at initialization

e.g. \( \log(C) \) for softmax with \( C \) classes

Reminder: \( L = - \log p = - \log(1/C) = \log(C) \)
Diagnose your training

**Step 1:** Check initial loss
**Step 2:** Overfit a small sample

Try to train to 100% training accuracy on a small sample of training data (~5-10 minibatches); fiddle with architecture, learning rate, weight initialization

Loss not going down? LR too low, bad initialization, bug in code or errors in training labels
Loss explodes to Inf or NaN? LR too high, bad initialization, bug in code
Diagnose your training

**Step 1:** Check initial loss
**Step 2:** Overfit a small sample
**Step 3:** Find LR that makes loss go down

Use the architecture from the previous step, use all training data, turn on small weight decay, find a learning rate that makes the loss drop significantly within ~100 iterations

Good learning rates to try: $1e^{-3}$, $3e^{-4}$, $1e^{-4}$
Diagnose your training

**Step 1:** Check initial loss  
**Step 2:** Overfit a small sample  
**Step 3:** Find LR that makes loss go down  
**Step 4:** Coarse grid, train for ~1-5 epochs

Choose a few values of learning rate and weight decay around what worked from Step 3, train a few models for ~1-5 epochs.

Good weight decay to try: 1e-4, 1e-5, 0
Diagnose your training

**Step 1:** Check initial loss
**Step 2:** Overfit a small sample
**Step 3:** Find LR that makes loss go down
**Step 4:** Coarse grid, train for ~1-5 epochs
**Step 5:** Refine grid, train longer

Pick best models from Step 4, train them for longer (~10-20 epochs) without learning rate decay
Diagnose your training

**Step 1:** Check initial loss
**Step 2:** Overfit a small sample
**Step 3:** Find LR that makes loss go down
**Step 4:** Coarse grid, train for ~1-5 epochs
**Step 5:** Refine grid, train longer
**Step 6:** Look at loss and accuracy curves
Accuracy still going up, you need to train longer

Train

Val

Accuracy

time
Huge train / val gap means overfitting! Increase regularization, get more data.
No gap between train / val means underfitting: train longer, use a bigger model, reduce regularization.

Accuracy

Train

Val

time
Losses may be noisy, use a scatter plot and also plot moving average to see trends better.
Cross-validation

We develop "command centers" to visualize all our models training with different hyperparameters.

check out weights and biases
You can plot all your loss curves for different hyperparameters on a single plot.
Don't look at accuracy or loss curves for too long!
Choosing Hyperparameters

Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid, train for ~1-5 epochs
Step 5: Refine grid, train longer
Step 6: Look at loss and accuracy curves
Step 7: GOTO step 5
Hyperparameters to play with:
- network architecture
- learning rate, its decay schedule, update type
- regularization (L1/L2/Dropout strength)
Summary

- Improve your training error:
  - Optimizers
  - Learning rate schedules

- Improve your test error:
  - Regularization
  - Choosing Hyperparameters
Summary

Training Deep Neural Networks
- Details of the non-linear activation functions
- Data normalization
- Weight Initialization
- Batch Normalization
- Advanced Optimization
- Regularization
- Data Augmentation
- Transfer learning
- Hyperparameter Tuning
Next time: Recurrent Neural Networks