Lecture 6 Review: Review Over Parts 1 + 2
Course Logistics

- Assignment 1 is due tomorrow!
- Project proposal deadline is on Monday
Topic 1: Layers in CNNs
Recap: Convolution Layer

3x32x32 image

Don’t forget bias terms!

6 activation maps, each 1x28x28

6x3x5x5 filters

Convolution Layer

Stack activations to get a 6x28x28 output image!

Activation Function! (ReLU)

Slide inspiration: Justin Johnson
Recap: Pooling Layer

Single depth slice

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

pool with 2x2 filters and stride 2

Max Pooling

6
3
3.25
2

Average Pooling

8
4
5.25
2

Fei-Fei Li, Ehsan Adeli, Zane Durante

Lecture 6 - April 18, 2024
Components of CNNs

Convolution Layers

Pooling Layers

Fully-Connected Layers

Activation Function

Normalization

\[ \hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}} \]
Batch Normalization

**Input:** $x : N \times D$

- Per-channel mean, shape is $D$
  \[ \mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j} \]
- Per-channel var, shape is $D$
  \[ \sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2 \]
- Normalized $x$, Shape is $N \times D$
  \[ \hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} \]

[Ioffe and Szegedy, 2015]
Batch Normalization

Input: $x : N \times D$

\[ \mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j} \]  
Per-channel mean, shape is D

\[ \sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2 \]  
Per-channel var, shape is D

\[ \hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} \]  
Normalized x, Shape is N x D

\[ y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j \]  
Output, Shape is N x D

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

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

\[ \mu_j = \text{(Running) average of values seen during training} \quad \text{Per-channel mean, shape is } D \]

**Learnable scale and shift parameters:**

\[ \gamma, \beta : D \]

\[ \sigma^2_j = \text{(Running) average of values seen during training} \quad \text{Per-channel var, shape is } D \]

\[ \hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma^2_j + \varepsilon}} \quad \text{Normalized x, Shape is } N \times D \]

\[ y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j \quad \text{Output, Shape is } N \times D \]

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

Wu and He, “Group Normalization”, ECCV 2018
Topic 2: CNN Architectures
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

- Lin et al
- Sanchez & Perronnin
- Krizhevsky et al (AlexNet)
- Zeiler & Fergus
- Simonyan & Zisserman (VGG)
- Szegedy et al (GoogLeNet)
- He et al (ResNet)
- Shao et al
- Hu et al (SENet)
- Russakovsky et al

Shallow: 8 layers
8 layers
19 layers
22 layers
152 layers
152 layers
152 layers

2010: 28.2
2011: 25.8
2012: 16.4
2013: 11.7
2014: 7.3
2014: 6.7
2015: 3.6
2016: 3
2017: 2.3
2018: 5.1
Human: 5.1
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

- Lin et al (2010, 28.2)
- Sanchez & Perronnin (2011, 25.8)
- Krizhevsky et al (AlexNet) (2012, 16.4)
- Zeiler & Fergus (2013, 11.7)
- Simonyan & Zisserman (VGG) (2014, 7.3)
- Szegedy et al (GoogLeNet) (2014, 6.7)
- He et al (ResNet) (2015, 3.6)
- Shao et al (2016, 3)
- Hu et al (SENet) (2017, 2.3)
- Russakovsky et al (Human, 5.1)

Network depths:
- AlexNet: 8 layers
- VGG: 19 layers
- GoogLeNet: 22 layers
- ResNet: 152 layers
- SENet: 152 layers
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

- **2010**: Lin et al.
- **2011**: Sanchez & Perronnin
- **2012**: Krizhevsky et al. (AlexNet)
- **2013**: Zeiler & Fergus
- **2014**: Simonyan & Zisserman (VGG)
- **2014**: Szegedy et al. (GoogLeNet)
- **2015**: He et al. (ResNet)
- **2016**: Shao et al.
- **2017**: Hu et al. (SENet)
- **2018**: Russakovsky et al.

**2010**: 28.2
**2011**: 25.8
**2012**: 16.4
**2013**: 11.7
**2014**: 7.3
**2014**: 6.7
**2015**: 3.6
**2016**: 3
**2017**: 2.3
**2018**: 5.1

- **2010**: shallow, 8 layers
- **2011**: 19 layers
- **2012**: 22 layers
- **2014**: 152 layers
- **2014**: 152 layers
- **2018**: 152 layers
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

- Sanchez & Perronnin (2011)
- Zeiler & Fergus (2013)
- Simonyan & Zisserman (VGG) (2014)
- Szegedy et al (GoogLeNet) (2014)
- Shao et al (2016)
- Russakovsky et al

“Revolution of Depth”

- Shallow models: 8 layers
- Increased depth: 19 layers, 22 layers, 152 layers

Fei-Fei Li, Ehsan Adeli, Zane Durante

Lecture 6 - 15

April 18, 2024
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]

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

Identity mapping: $H(x) = x$ if $F(x) = 0$

Use layers to fit residual $F(x) = H(x) - x$ instead of $H(x)$ directly.
Topic 3: Transfer Learning
You don’t always need a lot of data if you want to train/use 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

2. Small Dataset (C classes)

- Reinitialize this and train
- Freeze these

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

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

   - **Reinitialize this and train**

   - **Freeze these**

   - **Train these**

   - **With bigger dataset, train more layers**

   - **Freeze these**

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

Razavian et al., “CNN Features Off-the-Shelf: An Astounding Baseline for Recognition”, CVPR Workshops 2014
<table>
<thead>
<tr>
<th>More specific</th>
<th>More generic</th>
</tr>
</thead>
<tbody>
<tr>
<td>FC-1000</td>
<td>Image</td>
</tr>
<tr>
<td>FC-4096</td>
<td>Conv-64</td>
</tr>
<tr>
<td>FC-4096</td>
<td>Conv-64</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-512</td>
<td>Conv-512</td>
</tr>
<tr>
<td>Conv-512</td>
<td>Conv-512</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-256</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-256</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-128</td>
<td>Conv-128</td>
</tr>
<tr>
<td>Conv-128</td>
<td>Conv-128</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-64</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-64</td>
<td>Conv-64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>very little data</th>
<th>very similar dataset</th>
<th>very different dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ? )</td>
<td></td>
<td>( ? )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>quite a lot of data</th>
<th>very similar dataset</th>
<th>very different dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ? )</td>
<td></td>
<td>( ? )</td>
</tr>
<tr>
<td>More specific</td>
<td>More generic</td>
<td>very similar dataset</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------</td>
<td>----------------------</td>
</tr>
<tr>
<td><strong>very little data</strong></td>
<td>Use Linear Classifier on top layer</td>
<td>?</td>
</tr>
<tr>
<td><strong>quite a lot of data</strong></td>
<td>Finetune a few layers</td>
<td>?</td>
</tr>
<tr>
<td>More generic</td>
<td>More specific</td>
<td>very similar dataset</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>very little data</td>
<td>Use Linear Classifier on top layer</td>
<td>You’re in trouble… Try linear classifier from different stages</td>
</tr>
<tr>
<td>quite a lot of data</td>
<td>Finetune a few layers</td>
<td>Finetune a larger number of layers or start from scratch!</td>
</tr>
</tbody>
</table>
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 model 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

TensorFlow: [https://github.com/tensorflow/models](https://github.com/tensorflow/models)
PyTorch: [https://github.com/pytorch/vision](https://github.com/pytorch/vision)
Topic 4: Activation Functions in NNs
Standard Optimization Procedure

**Mini-batch SGD**

Loop:
1. **Sample** a batch of data
2. **Forward** prop it through the graph (network), get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient
Activation Functions

Sigmoid

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

- Squashes numbers to range \([0,1]\)
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

Key problem:

Saturated neurons “kill” the gradients
Activation Functions

-tanh(x) - Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

[LeCun et al., 1991]
Activation Functions

- Computes $f(x) = \max(0,x)$

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

ReLU
(Rectified Linear Unit)

[Krizhevsky et al., 2012]
Activation Functions

ReLU
(Rectified Linear Unit)

- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Not zero-centered output
- An annoyance:
  
  Dead ReLUs when $x < 0!$
Activation Functions

- Leaky ReLU
  \[ f(x) = \max(0.01x, x) \]

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- will not “die”.

Parametric Rectifier (PReLU)

\[ f(x) = \max(\alpha x, x) \]

backprop into \( \alpha \) (parameter)
Activation Functions

GELU (Gaussian Error Linear Unit)

- Computes \( f(x) = x \cdot \Phi(x) \)
- Very nice behavior around 0
- Smoothness facilitates training in practice
- Higher computational cost than ReLU
- Large negative values can still have gradient \( \rightarrow 0 \)

Source: [Hendrycks et al., 2016](https://en.m.wikipedia.org/wiki/File:ReLU_and_GELU.svg)
TLDR: In practice:

- Use **ReLU**. Be careful with your learning rates
- Try out **Leaky ReLU / PReLU / GELU**
  - To squeeze out some marginal gains
- Don’t use **sigmoid** or **tanh**
Topic 5: Data Preprocessing
TLDR: In practice for Images: center only

e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract per-channel mean and
  Divide by per-channel std (almost all modern models)
  (mean along each channel = 3 numbers)
Topic 6: Weight Initialization
Weight Initialization: Activation statistics

```python
dims = [4096] * 7  # Forward pass for a 6-layer net with hidden size 4096
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

All activations tend to zero for deeper network layers

**Q:** What do the gradients \( \frac{dL}{dW} \) look like?

\[
\frac{dL}{dW} = f'(a)x \times \text{upstream \_ grad}
\]

\[
a = \sum_i w_i x_i + t
\]
Weight Initialization: Activation statistics

```python
dims = [4096] * 7  # Increase std of initial weights from 0.01 to 0.05
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
x = np.tanh(x.dot(W))
hs.append(x)
```

All activations saturate

**Q:** What do the gradients look like?
Weight Initialization: “Xavier” Initialization

```
dims = [4096] * 7
hs = []

std = 1/sqrt(Din)
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)
```

Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010
Weight Initialization: “Xavier” 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(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

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

Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010
Weight Initialization: “Xavier” Initialization

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

```
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(Din)
x = np.tanh(x.dot(W))
hs.append(x)
```

“Xavier” initialization:
std = 1/sqrt(Din)

For conv layers, Din is filter_size^2 * input_channels

Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010
Weight Initialization: What about ReLU?

```python
dims = [4096] * 7  # Change from tanh to ReLU
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.maximum(0, x.dot(W))
    hs.append(x)
```
Weight Initialization: What about ReLU?

```python
 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(Din)
     x = np.maximum(0, x.dot(W))
     hs.append(x)
```

Xavier assumes zero centered activation function

Activations collapse to zero again, no learning 😞
Weight Initialization: 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)
```

"Just right": Activations are nicely scaled for all layers!

Topic 7: Training vs Testing
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**
  \[ R(W) = \sum_{k} \sum_{l} |W_{k,l}| \]
- **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

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

score

cat

score

score

score
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

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

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

\[ p = 0.5 \] # 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
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” → 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

**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
Regularization: Cutout

**Training**: Set random image regions to zero

**Testing**: Use full image

**Examples**: 
- Dropout
- Batch Normalization
- Data Augmentation
- Cutout / Random Crop

Works very well for small datasets like CIFAR, less common for large datasets like ImageNet

DeVries and Taylor, "Improved Regularization of Convolutional Neural Networks with Cutout", arXiv 2017
Topic 8: Hyperparameter Selection
Choosing Hyperparameters

**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-1, 1e-2, 1e-3, 1e-4
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 (next slides)
Accuracy still going up, you need to train longer

Accuracy vs. time graph with a note:

Train

Val
Huge train / val gap means overfitting! Increase regularization, get more data.
No gap between train / val means underfitting: train longer, can use a bigger model.
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
Random Search vs. Grid Search

**Grid Layout**

![Grid Layout](Image)

**Random Layout**

![Random Layout](Image)

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

Random Search for Hyper-Parameter Optimization
Bergstra and Bengio, 2012
Summary
We reviewed 8 topics at a high level:

1. Layers in CNNs
2. CNN Architectures (ResNets)
3. Transfer Learning (train on ImageNet first)
4. Activation Functions in NNs (ReLU, GELU, etc.)
Summary
We reviewed 8 topics at a high level:

5. Data Preprocessing (subtract mean, divide std)
6. Weight Initialization (Xavier vs Kaiming)
7. Training vs Testing (Regularization strategies)
8. Hyperparameter (Checking Losses + Random Search)