Lecture 9: CNN Architectures
Administrative

- A1 grades will be released hopefully by today: Check Piazza for regrade policy
- Project proposal grades will be released mid-week
Administrative

- A2 is due Friday April 30th, 11:59pm
Administrative: Midterm

Covers material through Lecture 10 (Thu April 29).

- Tues, May 4 and is worth 15% of your grade.
- available for 24 hours on Gradescope from May 4, 12PM PDT to May 5, 11:59 AM PDT.
- 3-hour consecutive timeframe
- Exam will be designed for 1.5 hours.
- Open book and open internet but no collaboration
- Only make private posts during those 24 hours
Administrative

Midterm review session: Fri April 30th discussion section

Sample midterm has been released on Piazza.

OAE accommodations: If you have not received an email from us, please reach out to the staff mailing list ASAP.
Last time: fancy optimizers

- SGD
- SGD+Momentum
- RMSProp
- Adam
Last time: learning rate scheduling

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

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

**Inverse sqrt:** 
\[ \alpha_t = \frac{\alpha_0}{\sqrt{t}} \]

\( \alpha_0 \) : Initial learning rate  
\( \alpha_t \) : Learning rate at epoch t  
\( T \) : Total number of epochs
Last time: dropout as a regularizer

\[ p = 0.5 \] # probability of keeping a unit active. higher = less dropout

```python
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
```

Example forward pass with a 3-layer network using dropout
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 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**: 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
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,
- lens distortions, … (go crazy)
Automatic Data Augmentation

Cubuk et al., “AutoAugment: Learning Augmentation Strategies from Data”, CVPR 2019
Regularization: A common pattern

**Training**: Add random noise

**Testing**: Marginalize over the noise

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

**Training**: Drop connections between neurons (set weights to 0)

**Testing**: Use all the connections

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

Wan et al, “Regularization of Neural Networks using DropConnect”, ICML 2013
Regularization: Fractional Pooling

**Training**: Use randomized pooling regions

**Testing**: Average predictions from several regions

**Examples**:

- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect
- Fractional Max Pooling

Graham, “Fractional Max Pooling”, arXiv 2014
Regularization: Stochastic Depth

Training: Skip some layers in the network
Testing: Use all the layer

Examples:
Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth (will become more clear in next week's lecture)

Regularization: Cutout

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

**Testing**: Use full image

**Examples**: Dropout, Batch Normalization, Data Augmentation, DropConnect, Fractional Max Pooling, Stochastic Depth, Cutout / Random Crop

DeVries and Taylor, “Improved Regularization of Convolutional Neural Networks with Cutout”, arXiv 2017

Works very well for small datasets like CIFAR, less common for large datasets like ImageNet
**Regularization: Mixup**

**Training:** Train on random blends of images

**Testing:** Use original images

**Examples:**
- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect
- Fractional Max Pooling
- Stochastic Depth
- Cutout / Random Crop
- Mixup

Randomly blend the pixels of pairs of training images, e.g. 40% cat, 60% dog

Target label: cat: 0.4
dog: 0.6

Zhang et al, "mixup: Beyond Empirical Risk Minimization", ICLR 2018
Regularization - In practice

Training: Add random noise
Testing: Marginalize over the noise

Examples:
- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect
- Fractional Max Pooling
- Stochastic Depth
- Cutout / Random Crop
- Mixup
- Consider dropout for large fully-connected layers
- Batch normalization and data augmentation almost always a good idea
- Try cutout and mixup especially for small classification datasets
Choosing Hyperparameters
(without tons of GPUs)
Choosing Hyperparameters

**Step 1: Check initial loss**

Turn off weight decay, sanity check loss at initialization
e.g. log(C) for softmax with C classes
Choosing Hyperparameters

**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
Loss explodes to Inf or NaN? LR too high, bad initialization
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 \( \sim 100 \) 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

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
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

Pick best models from Step 4, train them for longer (~10-20 epochs) without learning rate decay
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
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.
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 (L2/Dropout strength)
Random Search vs. Grid Search

Grid Layout

Random Layout

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

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

Wrapping up previous topics:
- Data augmentation to improve test time performance
- Transfer learning

New:
- CNN architecture design
Transfer learning
“You need a lot of data if you want to train/use CNNs”
“You need a lot of data if you want to train/use CNNs”

BUSTED
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

2. Small Dataset (C classes)

- Freeze these
- Reinitialize this and train

Transfer Learning with CNNs

1. Train on Imagenet

2. Small Dataset (C classes)

Finetuned from AlexNet

Reinitialize this and train

Freeze these

*Image*
Transfer Learning with CNNs

1. Train on Imagenet

2. Small Dataset (C classes)
   - Freeze these
   - Reinitialize this and train

3. Bigger dataset
   - Freeze these
   - Train these
   - With bigger dataset, train more layers
   - 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

Fei-Fei Li, Ranjay Krishna, Danfei Xu
<table>
<thead>
<tr>
<th>More generic</th>
<th>More specific</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FC-4096</strong></td>
<td><strong>FC-4096</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-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><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>
<tr>
<td><strong>Image</strong></td>
<td><strong>Image</strong></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>?</td>
</tr>
<tr>
<td>quite a lot of data</td>
<td>?</td>
</tr>
<tr>
<td>More generic</td>
<td>More specific</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>FC-1000</td>
<td>FC-4096</td>
</tr>
<tr>
<td>FC-4096</td>
<td>FC-4096</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>
<tr>
<td>Image</td>
<td>Image</td>
</tr>
</tbody>
</table>

### Table

<table>
<thead>
<tr>
<th>Dataset Type</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>very similar dataset</td>
<td>very different dataset</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>

?
<table>
<thead>
<tr>
<th>More general</th>
<th>More specific</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

Girshick, "Fast R-CNN", ICCV 2015
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

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)

1. Train CNN on ImageNet
2. Fine-Tune (1) for object detection on Visual Genome
3. Train BERT language model on lots of text
4. Combine (2) and (3), train for joint image / language modeling
5. Fine-tune (4) for image captioning, visual question answering, etc.

---

Figure copyright Luowei Zhou, 2020. Reproduced with permission.

Krishna et al. “Visual genome: Connecting language and vision using crowdsourced dense image annotations” IJCV 2017

Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding” ArXiv 2018

---

Fei-Fei Li, Ranjay Krishna, Danfei Xu
Lecture 9 - 61
April 27, 2021
Transfer learning with CNNs -
Architecture matters

We will discuss different architectures in detail today

Transfer learning with CNNs is pervasive…
But recent results show it might not always be necessary!

Training from scratch can work just as well as training from a pretrained ImageNet model for object detection. But it takes 2-3x as long to train.

They also find that collecting more data is better than finetuning on a related task.
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 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

TensorFlow: https://github.com/tensorflow/models
PyTorch: https://github.com/pytorch/vision
CNN Architectures
Review: LeNet-5

[LeCun et al., 1998]

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

32x32x3 image
3x3x3 filter $\mathcal{W}$

**Padding:**
Preserve input spatial dimensions in output activations

**Stride:**
Downsample output activations

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```

```
0 0 0 0 0 0
0 
0 
```
Review: Convolution

Each conv filter outputs a “slice” in the activation maps.
Review: Pooling

Single depth slice

max pool with 2x2 filters and stride 2
Today: CNN Architectures

Case Studies
- AlexNet
- VGG
- GoogLeNet
- ResNet

Also....
- SENet
- Wide ResNet
- ResNeXT
- DenseNet
- MobileNets
- NASNet
- EfficientNet
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

- **2010**: Lin et al (shallow)
- **2011**: Sanchez & Perronnin
- **2012**: Krizhevsky et al (AlexNet) - 8 layers
- **2013**: Zeiler & Fergus - 8 layers
- **2014**: Simonyan & Zisserman (VGG) - 19 layers
- **2014**: Szegedy et al (GoogLeNet) - 22 layers
- **2015**: He et al (ResNet) - 152 layers
- **2016**: Shao et al
- **2017**: Hu et al (SENet) - 152 layers
- **Human**: Russakovsky et al - 152 layers

*Note: The bar chart shows the accuracy improvements over time, with each year's winner achieving higher accuracy than the previous year.*
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

- First CNN-based winner: 2012
  - Lin et al
  - Sanchez & Perronnin
  - Krizhevsky et al (AlexNet)

- 2013
  - Zeiler & Fergus
  - Simonyan & Zisserman (VGG)

- 2014
  - Szegedy et al (GoogLeNet)

- 2015
  - He et al (ResNet)

- 2016
  - Shao et al

- 2017
  - Hu et al (SENet)

- Human

- Shallow
  - 8 layers

- 19 layers

- 22 layers

- 152 layers

- 152 layers

- 152 layers
Case Study: AlexNet

[Krizhevsky et al. 2012]

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

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

[Krizhevsky et al. 2012]

Input: 227x227x3 images

**First layer (CONV1):** 96 11x11 filters applied at stride 4
=>
Q: what is the output volume size? Hint: \((227 - 11)/4 + 1 = 55\)
Case Study: AlexNet

[Krizhevsky et al. 2012]

Input: 227x227x3 images

First layer (CONV1): 96 11x11 filters applied at stride 4

=>

Output volume [55x55x96]

\[ W' = \frac{(W - F + 2P)}{S} + 1 \]
Case Study: AlexNet

[Krizhevsky et al. 2012]

Input: 227x227x3 images

**First layer (CONV1):** 96 11x11 filters applied at stride 4

=>

Output volume **[55x55x96]**

Q: What is the total number of parameters in this layer?
Case Study: AlexNet

[Krizhevsky et al. 2012]

Input: 227x227x3 images

**First layer** (CONV1): 96 11x11 filters applied at stride 4

=>

Output volume **[55x55x96]**

Parameters: \((11 \times 11 \times 3 + 1) \times 96 = 35K\)

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

[Krizhevsky et al. 2012]

Input: 227x227x3 images
After CONV1: 55x55x96

Second layer (POOL1): 3x3 filters applied at stride 2

Q: what is the output volume size? Hint: (55-3)/2+1 = 27
Case Study: AlexNet

[Krizhevsky et al. 2012]

Input: 227x227x3 images
After CONV1: 55x55x96

Second layer (POOL1): 3x3 filters applied at stride 2
Output volume: 27x27x96

Q: what is the number of parameters in this layer?

\[ W' = \frac{W - F + 2P}{S} + 1 \]

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

[Krizhevsky et al. 2012]

Input: 227x227x3 images
After CONV1: 55x55x96

Second layer (POOL1): 3x3 filters applied at stride 2
Output volume: 27x27x96
Parameters: 0!
Case Study: AlexNet

[Krizhevsky et al. 2012]

Input: 227x227x3 images
After CONV1: 55x55x96
After POOL1: 27x27x96

...
Case Study: AlexNet

[Krizhevsky et al. 2012]

Full (simplified) AlexNet architecture:

- [227x227x3] INPUT
- [55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
- [27x27x96] MAX POOL1: 3x3 filters at stride 2
- [27x27x96] NORM1: Normalization layer
- [27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
- [13x13x256] MAX POOL2: 3x3 filters at stride 2
- [13x13x256] NORM2: Normalization layer
- [13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
- [13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
- [13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
- [6x6x256] MAX POOL3: 3x3 filters at stride 2
- [4096] FC6: 4096 neurons
- [4096] FC7: 4096 neurons
- [1000] FC8: 1000 neurons (class scores)

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

[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

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.
Case Study: AlexNet

[Krizhevsky et al. 2012]

Full (simplified) AlexNet architecture:

- **INPUT**
  - [227x227x3] INPUT

- **CONV1**: 96 11x11 filters at stride 4, pad 0
- **MAX POOL1**: 3x3 filters at stride 2
- **NORM1**: Normalization layer
- **CONV2**: 256 5x5 filters at stride 1, pad 2
- **MAX POOL2**: 3x3 filters at stride 2
- **NORM2**: Normalization layer
- **CONV3**: 384 3x3 filters at stride 1, pad 1
- **CONV4**: 384 3x3 filters at stride 1, pad 1
- **CONV5**: 256 3x3 filters at stride 1, pad 1
- **MAX POOL3**: 3x3 filters at stride 2
- **FC6**: 4096 neurons
- **FC7**: 4096 neurons
- **FC8**: 1000 neurons (class scores)

**CONV1, CONV2, CONV4, CONV5:** Connections only with feature maps on same GPU

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)

CONV3, FC6, FC7, FC8: Connections with all feature maps in preceding layer, communication across GPUs

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

- Layer counts:
  - 8 layers:
    - Sanchez & Perronnin (2011)
    - Zeiler & Fergus (2013)
    - Simonyan & Zisserman (VGG) (2014)
    - Szegedy et al (GoogLeNet) (2014)
    - Shao et al (2016)
  - 152 layers:
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

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

**Key Models**:
- **AlexNet**: Shallow, 8 layers
- **VGG**: 19 layers
- **GoogLeNet**: 22 layers
- **ResNet**: 152 layers
- **SENet**: 152 layers

**Notable Improvements**:
- **ZFNet**: Improved hyperparameters over AlexNet - 8 layers
AlexNet but:
CONV1: change from (11x11 stride 4) to (7x7 stride 2)
CONV3,4,5: instead of 384, 384, 256 filters use 512, 1024, 512

ImageNet top 5 error: 16.4% -> 11.7%
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)
- Human

- Lin et al: 28.2
- Sanchez & Perronnin: 25.8
- Krizhevsky et al: 16.4
- Zeiler & Fergus: 11.7
- Simonyan & Zisserman: 7.3
- Szegedy et al: 6.7
- He et al: 3.6
- Shao et al: 3
- Hu et al: 2.3
- Human: 5.1

Shallow: 8 layers
Deeper Networks: 19 layers
Deeper Networks: 22 layers
Deeper Networks: 152 layers
Deeper Networks: 152 layers
Deeper Networks: 152 layers
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Small filters, Deeper networks

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

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

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

[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)
Case Study: VGGNet
[Simonyan and Zisserman, 2014]

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

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

Q: What is the effective receptive field of three 3x3 conv (stride 1) layers?
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Q: What is the effective receptive field of three 3x3 conv (stride 1) layers?

Conv1 (3x3)  Conv2 (3x3)  Conv3 (3x3)
Case Study: VGGNet
[Simonyan and Zisserman, 2014]

Q: What is the effective receptive field of three 3x3 conv (stride 1) layers?
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Q: What is the effective receptive field of three 3x3 conv (stride 1) layers?

VGG16

VGG19
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Q: What is the effective receptive field of three 3x3 conv (stride 1) layers?
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

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

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

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

[7x7]
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: [224x224x3]  memory: 224*224*3=150K  params: 0  (not counting biases)
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
INPUT: [224x224x3]  memory: 224*224*3=150K  params: 0  (not counting biases)

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 ~= 96MB / image (for a forward pass)
TOTAL params: 138M parameters
INPUT: [224x224x3]  memory: 224*224*3 = 150K  params: 0 (not counting biases)

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 ≈ 96MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters

Note:
Most memory is in early CONV
Most params are in late FC
<table>
<thead>
<tr>
<th>Layer Type</th>
<th>Input Size</th>
<th>Memory (bytes)</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT</td>
<td>[224x224x3]</td>
<td>224<em>224</em>3=150K</td>
<td>0</td>
</tr>
<tr>
<td>CONV3-64</td>
<td>[224x224x64]</td>
<td>224<em>224</em>64=3.2M</td>
<td>(3<em>3</em>3)*64 = 1,728</td>
</tr>
<tr>
<td>CONV3-64</td>
<td>[224x224x64]</td>
<td>224<em>224</em>64=3.2M</td>
<td>(3<em>3</em>64)*64 = 36,864</td>
</tr>
<tr>
<td>POOL2</td>
<td>[112x112x64]</td>
<td>112<em>112</em>64=800K</td>
<td>0</td>
</tr>
<tr>
<td>CONV3-128</td>
<td>[112x112x128]</td>
<td>112<em>112</em>128=1.6M</td>
<td>(3<em>3</em>64)*128 = 73,728</td>
</tr>
<tr>
<td>CONV3-128</td>
<td>[112x112x128]</td>
<td>112<em>112</em>128=1.6M</td>
<td>(3<em>3</em>128)*128 = 147,456</td>
</tr>
<tr>
<td>POOL2</td>
<td>[56x56x128]</td>
<td>56<em>56</em>128=400K</td>
<td>0</td>
</tr>
<tr>
<td>CONV3-256</td>
<td>[56x56x256]</td>
<td>56<em>56</em>256=800K</td>
<td>(3<em>3</em>128)*256 = 294,912</td>
</tr>
<tr>
<td>CONV3-256</td>
<td>[56x56x256]</td>
<td>56<em>56</em>256=800K</td>
<td>(3<em>3</em>256)*256 = 589,824</td>
</tr>
<tr>
<td>CONV3-256</td>
<td>[56x56x256]</td>
<td>56<em>56</em>256=800K</td>
<td>(3<em>3</em>256)*256 = 589,824</td>
</tr>
<tr>
<td>POOL2</td>
<td>[28x28x256]</td>
<td>28<em>28</em>256=200K</td>
<td>0</td>
</tr>
<tr>
<td>CONV3-512</td>
<td>[28x28x512]</td>
<td>28<em>28</em>512=400K</td>
<td>(3<em>3</em>256)*512 = 1,179,648</td>
</tr>
<tr>
<td>CONV3-512</td>
<td>[28x28x512]</td>
<td>28<em>28</em>512=400K</td>
<td>(3<em>3</em>512)*512 = 2,359,296</td>
</tr>
<tr>
<td>CONV3-512</td>
<td>[28x28x512]</td>
<td>28<em>28</em>512=400K</td>
<td>(3<em>3</em>512)*512 = 2,359,296</td>
</tr>
<tr>
<td>POOL2</td>
<td>[14x14x512]</td>
<td>14<em>14</em>512=100K</td>
<td>0</td>
</tr>
<tr>
<td>CONV3-512</td>
<td>[14x14x512]</td>
<td>14<em>14</em>512=100K</td>
<td>(3<em>3</em>512)*512 = 2,359,296</td>
</tr>
<tr>
<td>CONV3-512</td>
<td>[14x14x512]</td>
<td>14<em>14</em>512=100K</td>
<td>(3<em>3</em>512)*512 = 2,359,296</td>
</tr>
<tr>
<td>POOL2</td>
<td>[7x7x512]</td>
<td>7<em>7</em>512=25K</td>
<td>0</td>
</tr>
<tr>
<td>FC</td>
<td>[1x1x4096]</td>
<td>4096</td>
<td>0</td>
</tr>
<tr>
<td>FC</td>
<td>[1x1x4096]</td>
<td>4096</td>
<td>0</td>
</tr>
<tr>
<td>FC</td>
<td>[1x1x1000]</td>
<td>1000</td>
<td>0</td>
</tr>
</tbody>
</table>

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

**TOTAL params:** 138M parameters
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

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

- **2010**: Lin et al. (28.2)
- **2011**: Sanchez & Perronnin (25.8)
- **2012**: Krizhevsky et al. (AlexNet) (16.4)
- **2013**: Zeiler & Fergus (11.7)
- **2014**: Simonyan & Zisserman (VGG) (7.3)
- **2014**: Szegedy et al. (GoogLeNet) (6.7)
- **2015**: He et al. (ResNet) (3.6)
- **2016**: Shao et al. (3.0)
- **2017**: Hu et al. (SENet) (2.3)
- **Human**: 5.1

**Deeper Networks**

- **2014**: 22 layers
- **2015**: 152 layers
- **2016**: 152 layers
- **2017**: 152 layers

**Shallow**

- **2010**: 8 layers
- **2011**: 8 layers
- **2012**: 8 layers
- **2013**: 19 layers
- **2014**: 152 layers
Case Study: GoogLeNet

[Szegedy et al., 2014]

Deeper networks, with computational efficiency

- ILSVRC’14 classification winner (6.7% top 5 error)
- 22 layers
- Only 5 million parameters!
  12x less than AlexNet
  27x less than VGG-16
- Efficient “Inception” module
- No FC layers

Inception module
Case Study: GoogLeNet

[Szegedy et al., 2014]

“Inception module”: design a good local network topology (network within a network) and then stack these modules on top of each other
Case Study: GoogLeNet

[Szegedy et al., 2014]

Naive Inception module

Apply parallel filter operations on the input from previous layer:
- Multiple receptive field sizes for convolution (1x1, 3x3, 5x5)
- Pooling operation (3x3)

Concatenate all filter outputs together channel-wise
Case Study: GoogLeNet

[Szegedy et al., 2014]

Naive Inception module

Apply parallel filter operations on the input from previous layer:
- Multiple receptive field sizes for convolution (1x1, 3x3, 5x5)
- Pooling operation (3x3)

Concatenate all filter outputs together channel-wise

Q: What is the problem with this?
[Hint: Computational complexity]
Case Study: GoogLeNet

[ Szegedy et al., 2014 ]

Q: What is the problem with this?
[ Hint: Computational complexity ]

Example:

Naive Inception module
Case Study: GoogLeNet
[Szegedy et al., 2014]

Example:

Q1: What are the output sizes of all different filter operations?

Naive Inception module

Module input: 28x28x256

Input

1x1 conv, 128
3x3 conv, 192
5x5 conv, 96
3x3 pool

Filter concatenation

Q: What is the problem with this? [Hint: Computational complexity]
Case Study: GoogLeNet

[Szegedy et al., 2014]

Naive Inception module

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q1: What are the output sizes of all different filter operations?

Module input: 28x28x256

Input

1x1 conv, 128
3x3 conv, 192
5x5 conv, 96
3x3 pool

28x28x128
28x28x192
28x28x96
28x28x256

Filter concatenation
Case Study: GoogLeNet

[Szegedy et al., 2014]

**Naive Inception module**

*Input*

- 3x3 pool
- 5x5 conv, 96
- 3x3 conv, 192
- 1x1 conv, 128

*Filter concatenation*

*Module input: 28x28x256*

Q: What is the problem with this? [Hint: Computational complexity]

**Q2: What is output size after filter concatenation?**

- 28x28x128
- 28x28x192
- 28x28x96
- 28x28x256
Case Study: GoogLeNet
[Szegedy et al., 2014]

Example:

Q2: What is output size after filter concatenation?

28x28x(128+192+96+256) = 28x28x672

Naive Inception module

Q: What is the problem with this?
[Hint: Computational complexity]
Case Study: GoogLeNet

[Szegedy et al., 2014]

Naive Inception module

Example:

Q: What is output size after filter concatenation?

\[28 \times 28 \times (128 + 192 + 96 + 256) = 28 \times 28 \times 672\]

Conv Ops:

- [1x1 conv, 128]: 28x28x128x1x1x256
- [3x3 conv, 192]: 28x28x192x3x3x256
- [5x5 conv, 96]: 28x28x96x5x5x256

Total: 854M ops

Q: What is the problem with this? [Hint: Computational complexity]
Case Study: GoogLeNet

[ Szegedy et al., 2014 ]

Example:

Q: What is output size after filter concatenation?

\[ 28 \times 28 \times 128 + 192 + 96 + 256 = 28 \times 28 \times 672 \]

Conv Ops:

- [1x1 conv, 128] \( 28 \times 28 \times 128 \times 1 \times 1 \times 256 \)
- [3x3 conv, 192] \( 28 \times 28 \times 192 \times 3 \times 3 \times 256 \)
- [5x5 conv, 96] \( 28 \times 28 \times 96 \times 5 \times 5 \times 256 \)

Total: 854M ops

Very expensive compute

Pooling layer also preserves feature depth, which means total depth after concatenation can only grow at every layer!
Case Study: GoogLeNet
[Szegedy et al., 2014]

Example:

Q2: What is output size after filter concatenation?

28x28x(128+192+96+256) = 529k

Solution: “bottleneck” layers that use 1x1 convolutions to reduce feature channel size
Review: 1x1 convolutions

1x1 CONV with 32 filters

(each filter has size 1x1x64, and performs a 64-dimensional dot product)
Review: 1x1 convolutions

1x1 CONV with 32 filters

(Each filter has size 1x1x64, and performs a 64-dimensional dot product)

Alternatively, interpret it as applying the same FC layer on each input pixel.
Review: 1x1 convolutions

1x1 CONV with 32 filters preserves spatial dimensions, reduces depth!

Projects depth to lower dimension (combination of feature maps)

Alternatively, interpret it as applying the same FC layer on each input pixel.
Case Study: GoogLeNet

[Szegedy et al., 2014]

Naive Inception module

Inception module with dimension reduction
Case Study: GoogLeNet

[Szegedy et al., 2014]

Naive Inception module

Inception module with dimension reduction
Case Study: GoogLeNet

[Szegedy et al., 2014]

Inception module with dimension reduction

Using same parallel layers as naive example, and adding “1x1 conv, 64 filter” bottlenecks:

**Conv Ops:**
- [1x1 conv, 64] 28x28x64x1x1x256
- [1x1 conv, 64] 28x28x64x1x1x256
- [1x1 conv, 128] 28x28x128x1x1x256
- [3x3 conv, 192] 28x28x192x3x3x64
- [5x5 conv, 96] 28x28x96x5x5x64
- [1x1 conv, 64] 28x28x64x1x1x256

**Total: 358M ops**

Compared to 854M ops for naive version
Bottleneck can also reduce depth after pooling layer
Case Study: GoogLeNet

[ Szegedy et al., 2014 ]

Stack Inception modules with dimension reduction on top of each other

Inception module
Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

Stem Network:
Conv-Pool-
2x Conv-Pool
Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

Stacked Inception Modules
Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

Classifier output
Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

Note: after the last convolutional layer, a global average pooling layer is used that spatially averages across each feature map, before final FC layer. No longer multiple expensive FC layers!
Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

Auxiliary classification outputs to inject additional gradient at lower layers
(AvgPool-1x1Conv-FC-FC-Softmax)
Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

22 total layers with weights
(parallel layers count as 1 layer => 2 layers per Inception module. Don’t count auxiliary output layers)
Case Study: GoogLeNet

[Szegedy et al., 2014]

Deeper networks, with computational efficiency

- 22 layers
- Efficient “Inception” module
- Avoids expensive FC layers
- 12x less params than AlexNet
- 27x less params than VGG-16
- ILSVRC’14 classification winner (6.7% top 5 error)
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

“Revolution of Depth”

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

- 8 layers
- 19 layers
- 22 layers
- 152 layers
- 152 layers
- 152 layers

Shallow models vs. deep models:
- Shallow
- 8 layers
- 8 layers
- 152 layers
- 152 layers
- 152 layers

He et al introduced ResNet, which revolutionized deep learning by enabling the training of much deeper networks.
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?
Case Study: ResNet

[He et al., 2015]

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

![Graph showing test and training error for 56-layer and 20-layer models](image)

56-layer model performs worse on both test and training error

-> The deeper model performs worse, but it’s not caused by overfitting!
Case Study: ResNet

[He et al., 2015]

Fact: Deep models have more representation power (more parameters) than shallower models.

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

[He et al., 2015]

Fact: Deep models have more representation power (more parameters) than shallower models.

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

What should the deeper model learn to be at least as good 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

\[
\begin{align*}
H(x) &= \text{conv} \\
& \quad \text{relu} \\
& \quad \text{conv} \\
& \quad X
\end{align*}
\]

“Plain” layers
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

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

Identity mapping:
\[ H(x) = x \text{ if } F(x) = 0 \]
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:
\[ H(x) = x \text{ if } F(x) = 0 \]

Use layers to fit residual
\[ F(x) = H(x) - x \]

instead of
\[ H(x) \text{ directly} \]
Case Study: ResNet

[He et al., 2015]

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

Residual block:
F(x) + x

F(x)
relu
3x3 conv
X
identity
relu
3x3 conv
X
Case Study: ResNet

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)
Reduce the activation volume by half.
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 (stem)
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 (stem)
- No FC layers at the end (only FC 1000 to output classes)
- (In theory, you can train a ResNet with input image of variable sizes)
Case Study: ResNet
[He et al., 2015]

Total depths of 18, 34, 50, 101, or 152 layers for ImageNet
Case Study: ResNet

[He et al., 2015]

For deeper networks (ResNet-50+), use “bottleneck” layer to improve efficiency (similar to GoogLeNet)
Case Study: ResNet

[He et al., 2015]

For deeper networks (ResNet-50+), use “bottleneck” layer to improve efficiency (similar to GoogLeNet)

1x1 conv, 256 filters projects back to 256 feature maps (28x28x256)
3x3 conv operates over only 64 feature maps
1x1 conv, 64 filters to project to 28x28x64

BN, relu
Case Study: ResNet
[He et al., 2015]

Training ResNet in practice:

- Batch Normalization after every CONV layer
- Xavier initialization from He et al.
- SGD + Momentum (0.9)
- Learning rate: 0.1, divided by 10 when validation error plateaus
- Mini-batch size 256
- Weight decay of 1e-5
- No dropout used
Case Study: ResNet

[He et al., 2015]

Experimental Results
- Able to train very deep networks without degrading (152 layers on ImageNet, 1202 on Cifar)
- Deeper networks now achieve lower training error as expected
- Swept 1st place in all ILSVRC and COCO 2015 competitions

MSRA @ ILSVRC & COCO 2015 Competitions

• 1st places in all five main tracks
  • ImageNet Classification: “Ultra-deep” (quote Yann) 152-layer nets
  • ImageNet Detection: 16% better than 2nd
  • ImageNet Localization: 27% better than 2nd
  • COCO Detection: 11% better than 2nd
  • COCO Segmentation: 12% better than 2nd
Case Study: ResNet

[He et al., 2015]

Experimental Results
- Able to train very deep networks without degrading (152 layers on ImageNet, 1202 on Cifar)
- Deeper networks now achieve lower training error as expected
- Swept 1st place in all ILSVRC and COCO 2015 competitions

MSRA @ ILSVRC & COCO 2015 Competitions

- 1st places in all five main tracks
  - ImageNet Classification: “Ultra-deep” (quote Yann) 152-layer nets
  - ImageNet Detection: 16% better than 2nd
  - ImageNet Localization: 27% better than 2nd
  - COCO Detection: 11% better than 2nd
  - COCO Segmentation: 12% better than 2nd

ILSVRC 2015 classification winner (3.6% top 5 error) -- better than “human performance”! (Russakovsky 2014)
Comparing complexity...


Comparing complexity...

Inception-v4: ResNet + Inception!


Comparing complexity...


VGG: most parameters, most operations

Comparing complexity...


Comparing complexity...

AlexNet:
Smaller compute, still memory heavy, lower accuracy


Comparing complexity...


ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

- **2010**: 28.2
  - Lin et al
- **2011**: 25.8
  - Sanchez & Perronnin
- **2012**: 16.4
  - Krizhevsky et al (AlexNet)
- **2013**: 11.7
  - Zeiler & Fergus
- **2014**: 7.3
  - Simonyan & Zisserman (VGG)
- **2014**: 6.7
  - Szegedy et al (GoogLeNet)
- **2015**: 3.6
  - He et al (ResNet)
- **2016**: 3
  - Shao et al
- **2017**: 2.3
  - Hu et al (SENet)
- **2018**: 5.1
  - Russakovsky et al

Network ensembling

- 152 layers
- 152 layers
- 152 layers

Shallow: 8 layers
- 2012
- 2013

8 layers
- 2014
- 2014

19 layers
- 2013

22 layers
- 2014
Improving ResNets...

“Good Practices for Deep Feature Fusion”
[Shao et al. 2016]

- Multi-scale ensembling of Inception, Inception-Resnet, Resnet, Wide Resnet models
- ILSVRC’16 classification winner

<table>
<thead>
<tr>
<th></th>
<th>Inception-v3</th>
<th>Inception-v4</th>
<th>Inception-Resnet-v2</th>
<th>Resnet-200</th>
<th>Wrn-68-3</th>
<th>Fusion (Val.)</th>
<th>Fusion (Test)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err. (%)</td>
<td>4.20</td>
<td>4.01</td>
<td>3.52</td>
<td>4.26</td>
<td>4.65</td>
<td><strong>2.92 (-0.6)</strong></td>
<td><strong>2.99</strong></td>
</tr>
</tbody>
</table>
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 (Human) (2021)

Adaptive feature map reweighting
Improving ResNets...

Squeeze-and-Excitation Networks (SENet)

[Hu et al. 2017]

- Add a “feature recalibration” module that learns to adaptively reweight feature maps
- Global information (global avg. pooling layer) + 2 FC layers used to determine feature map weights
- ILSVRC’17 classification winner (using ResNeXt-152 as a base architecture)
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

<table>
<thead>
<tr>
<th>Year</th>
<th>Winner</th>
<th>Layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010</td>
<td>Lin et al</td>
<td>shallow</td>
</tr>
<tr>
<td>2011</td>
<td>Sanchez &amp; Perronnin</td>
<td>8 layers</td>
</tr>
<tr>
<td>2012</td>
<td>Krizhevsky et al (AlexNet)</td>
<td>8 layers</td>
</tr>
<tr>
<td>2013</td>
<td>Zeiler &amp; Fergus</td>
<td>19 layers</td>
</tr>
<tr>
<td>2014</td>
<td>Simonyan &amp; Zisserman (VGG)</td>
<td>22 layers</td>
</tr>
<tr>
<td>2014</td>
<td>Szegedy et al (GoogLeNet)</td>
<td></td>
</tr>
<tr>
<td>2015</td>
<td>He et al (ResNet)</td>
<td>152 layers</td>
</tr>
<tr>
<td>2016</td>
<td>Shao et al</td>
<td>152 layers</td>
</tr>
<tr>
<td>2017</td>
<td>Hu et al (SENet)</td>
<td>152 layers</td>
</tr>
<tr>
<td>2018</td>
<td>Russakovsky et al</td>
<td></td>
</tr>
</tbody>
</table>

Human performance: 5.1
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

Completion of the challenge:
Annual ImageNet competition no longer held after 2017 -> now moved to Kaggle.
But research into CNN architectures is still flourishing
Improving ResNets...

Identity Mappings in Deep Residual Networks

[He et al. 2016]

- Improved ResNet block design from creators of ResNet
- Creates a more direct path for propagating information throughout network
- Gives better performance
Improving ResNets...

Wide Residual Networks

[Zagoruyko et al. 2016]

- Argues that residuals are the important factor, not depth
- User wider residual blocks (F x k filters instead of F filters in each layer)
- 50-layer wide ResNet outperforms 152-layer original ResNet
- Increasing width instead of depth more computationally efficient (parallelizable)
Improving ResNets...

Aggregated Residual Transformations for Deep Neural Networks (ResNeXt)

[Xie et al. 2016]

- Also from creators of ResNet
- Increases width of residual block through multiple parallel pathways ("cardinality")
- Parallel pathways similar in spirit to Inception module
Other ideas...

Densely Connected Convolutional Networks (DenseNet)

[Huang et al. 2017]

- Dense blocks where each layer is connected to every other layer in feedforward fashion
- Alleviates vanishing gradient, strengthens feature propagation, encourages feature reuse
- Showed that shallow 50-layer network can outperform deeper 152 layer ResNet
Efficient networks...

**MobileNets: Efficient Convolutional Neural Networks for Mobile Applications [Howard et al. 2017]**

- Depthwise separable convolutions replace standard convolutions by factorizing them into a depthwise convolution and a 1x1 convolution.
- Much more efficient, with little loss in accuracy.
- Follow-up MobileNetV2 work in 2018 (Sandler et al.).
Learning to search for network architectures...

Neural Architecture Search with Reinforcement Learning (NAS) [Zoph et al. 2016]

- “Controller” network that learns to design a good network architecture (output a string corresponding to network design)
- Iterate:
  1) Sample an architecture from search space
  2) Train the architecture to get a “reward” $R$ corresponding to accuracy
  3) Compute gradient of sample probability, and scale by $R$ to perform controller parameter update (i.e. increase likelihood of good architecture being sampled, decrease likelihood of bad architecture)
Learning to search for network architectures...

Learning Transferable Architectures for Scalable Image Recognition

[Zoph et al. 2017]

- Applying neural architecture search (NAS) to a large dataset like ImageNet is expensive
- Design a search space of building blocks (“cells”) that can be flexibly stacked
- NASNet: Use NAS to find best cell structure on smaller CIFAR-10 dataset, then transfer architecture to ImageNet
- Many follow-up works in this space e.g. AmoebaNet (Real et al. 2019) and ENAS (Pham, Guan et al. 2018)
But sometimes smart heuristic is better than NAS ...

**EfficientNet: Smart Compound Scaling**

[Tan and Le. 2019]

- Increase network capacity by scaling width, depth, and resolution, while balancing accuracy and efficiency.
- Search for optimal set of compound scaling factors given a compute budget (target memory & flops).
- Scale up using smart heuristic rules

\[
\begin{align*}
\text{depth: } & d = \alpha^\phi \\
\text{width: } & w = \beta^\phi \\
\text{resolution: } & r = \gamma^\phi \\
\end{align*}
\]

\[\alpha \cdot \beta^2 \cdot \gamma^2 \approx 2\]

\[\alpha \geq 1, \beta \geq 1, \gamma \geq 1\]
Efficient networks...

https://openai.com/blog/ai-and-efficiency/
Summary: CNN Architectures

Case Studies
- AlexNet
- VGG
- GoogLeNet
- ResNet

Also....
- SENet
- Wide ResNet
- ResNeXT
- DenseNet
- MobileNets
- NASNet
Main takeaways

**AlexNet** showed that you can use CNNs to train Computer Vision models. **ZFNet, VGG** shows that bigger networks work better. **GoogLeNet** is one of the first to focus on efficiency using 1x1 bottleneck convolutions and global avg pool instead of FC layers. **ResNet** showed us how to train extremely deep networks:
- Limited only by GPU & memory!
- Showed diminishing returns as networks got bigger

After ResNet: CNNs were better than the human metric and focus shifted to efficient networks:
- Lots of tiny networks aimed at mobile devices: **MobileNet, ShuffleNet**

**Neural Architecture Search** can now automate architecture design.
Summary: CNN Architectures

- Many popular architectures available in model zoos
- ResNet and SENet currently good defaults to use
- Networks have gotten increasingly deep over time
- Many other aspects of network architectures are also continuously being investigated and improved

- Next time: Recurrent neural networks
Next time: Recurrent Neural Networks