Lecture 04: CNNs II – Network Regularization & Architecture

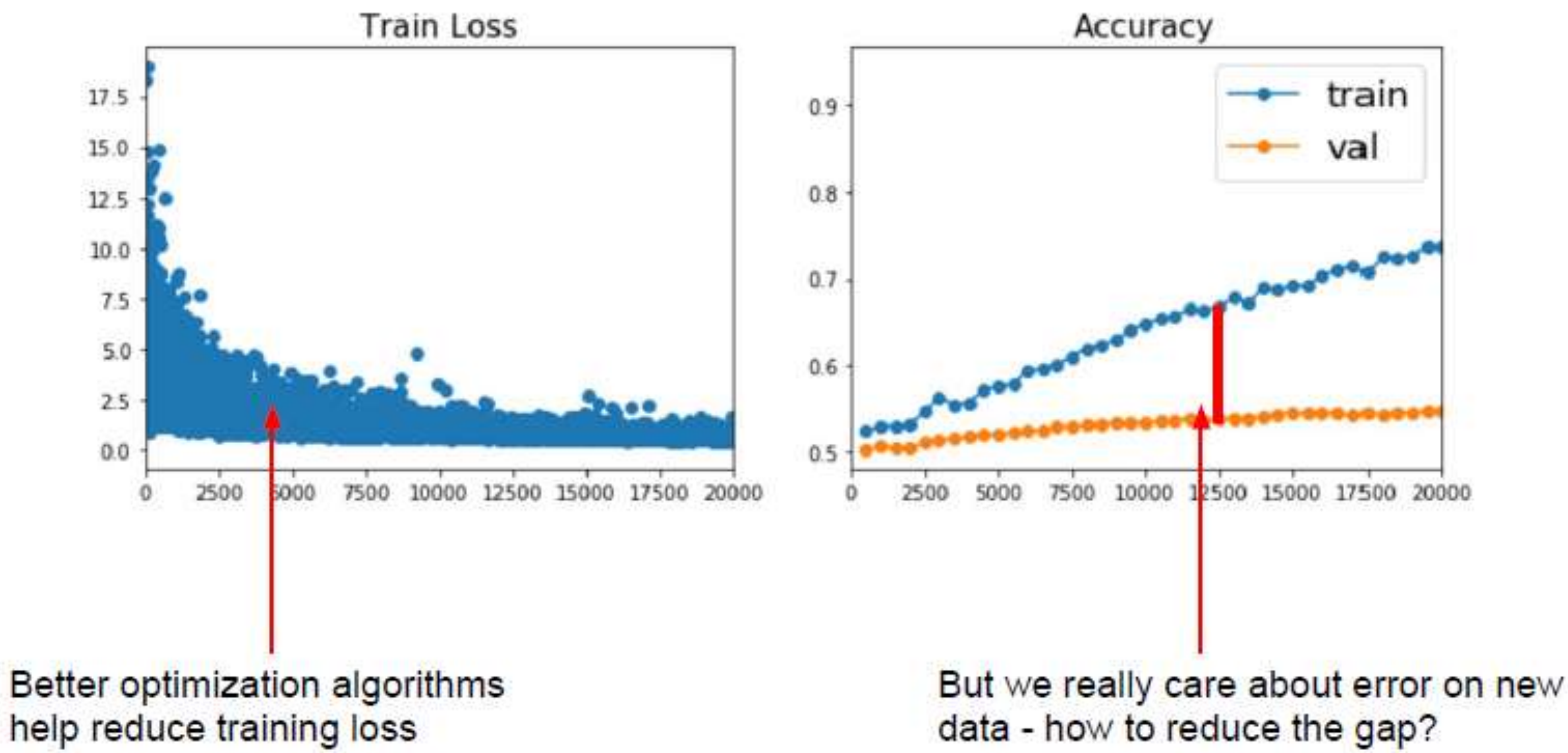
Lan Xu
SIST, ShanghaiTech
Fall, 2023

Training overview

- Two aspects of training networks
  - Optimization
    - How do we minimize the loss function effectively?
  - Generalization
    - How do we avoid overfitting?
- CNN training pipeline
  - Data processing
  - Weight initialization
  - Parameter updates
  - Batch normalization
- Avoid overfitting: Regularization

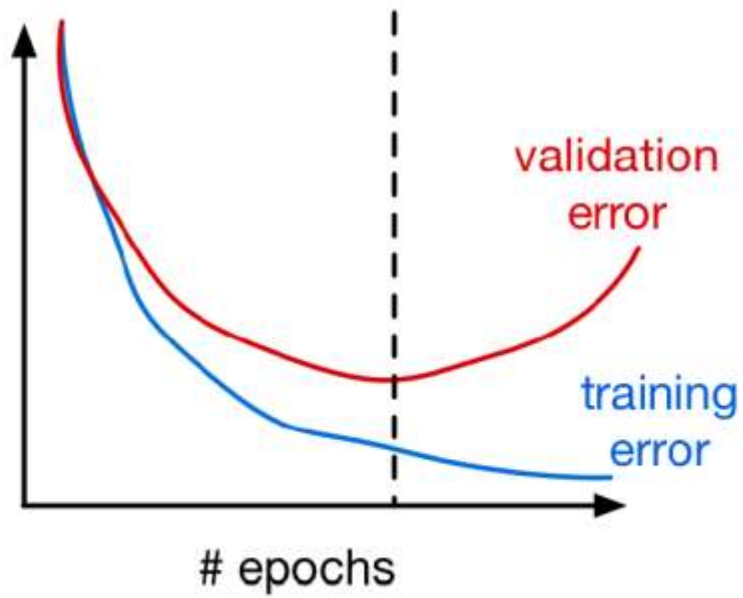
Beyond Training Error

- How do we generalize to unseen data?
  - Well studied but still poorly understood



Early Stopping

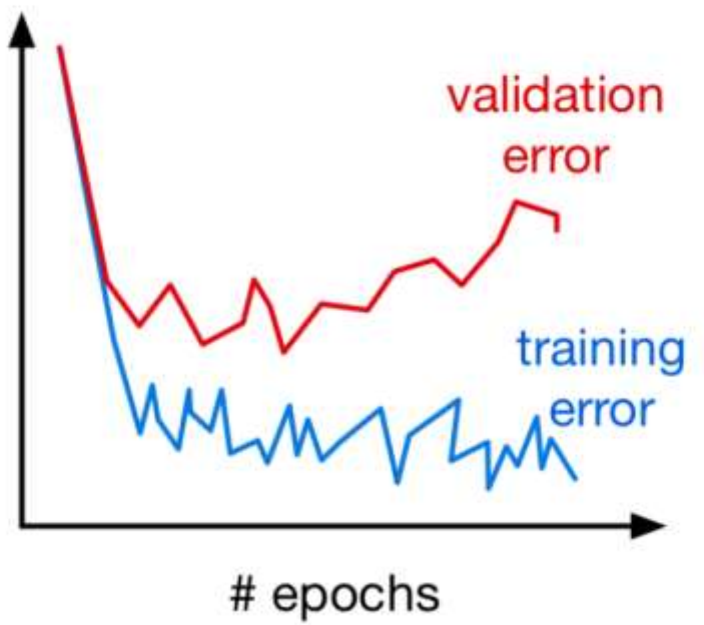
- Early stopping: monitor performance on a validation set, stop training when the validation error starts going up.
  - We don't always want to find a global (or even local) optimum of our cost function.



  - Weights start out small, so it takes time for them to grow large. Therefore, it has a similar effect to weight decay.

Early Stopping

- A slight catch: validation error fluctuates because of stochasticity in the updates.
  - Determining when the validation error has actually leveled off can be tricky.
  - May use temporal smoothing



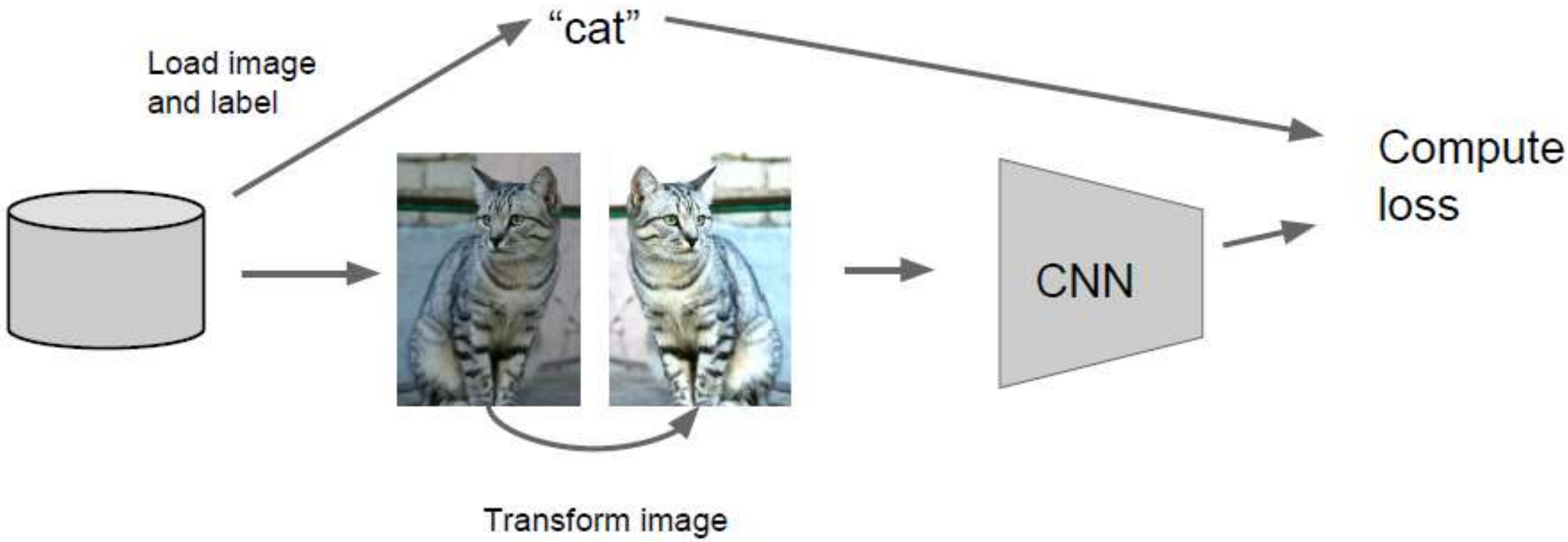
Outline

- Regularization in CNN training
  - Data Augmentation
  - Weight Regularization & Transfer Learning
  - Stochastic Regularization
  - Hyper-parameter optimization

Acknowledgement: Feifei Li's cs231n notes

Data Augmentation

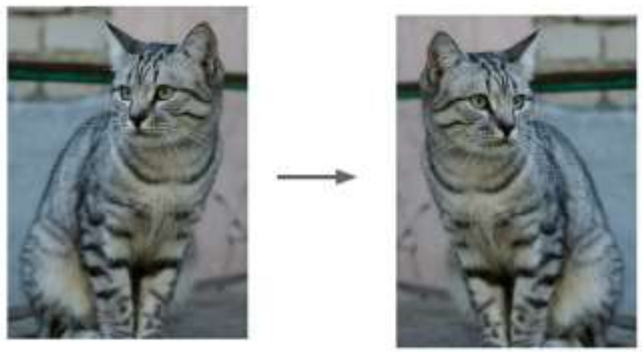
- Create more data for regularization



Data Augmentation

- Create more data for regularization

Horizontal Flips



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:

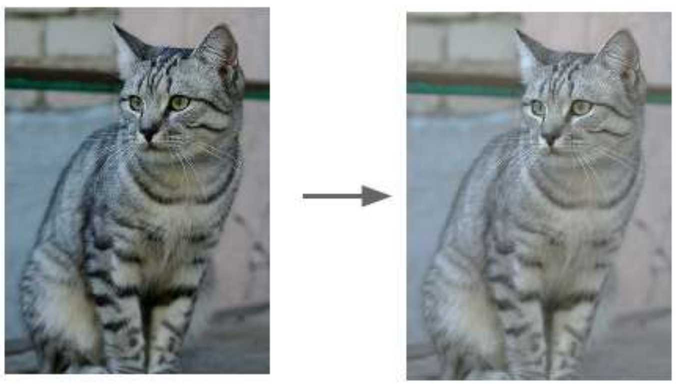
1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224 x 224 crops: 4 corners + center, + flips

Data Augmentation

- Create more data for regularization

Color Jitter

Simple: Randomize contrast and brightness



More Complex:

1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a “color offset” along principal component directions
3. Add offset to all pixels of a training image

(As seen in *[Krizhevsky et al. 2012]*, ResNet, etc)

Data Augmentation

- Create more data for regularization
- Examples (for visual recognition)
  - translation
  - horizontal or vertical
  - flip
  - rotation
  - smooth warping
  - noise (e.g. flip random pixels)
- The choice of transformations depends on the task.
  - E.g. horizontal flip for object recognition, but not handwritten digit recognition.

Data Augmentation

- AutoAugment (Cubuk et al, Arxiv 2018)
  - An automatic way to design custom data augmentation policies for computer vision datasets,
  - Selecting an optimal policy from a search space of 2.9 x 10^{32} image transformation possibilities.
    - E.g., guiding the selection of basic image transformation operations, such as flipping an image horizontally/vertically, rotating an image, changing the color of an image, etc.
  - Using reinforcement learning strategy (More later…)
- Results
  - New state of the art: ImageNet: 83.54% top1 accuracy; SVHN: error rate 1.02%.
  - AutoAugment policies are found to be transferable to other vision datasets.

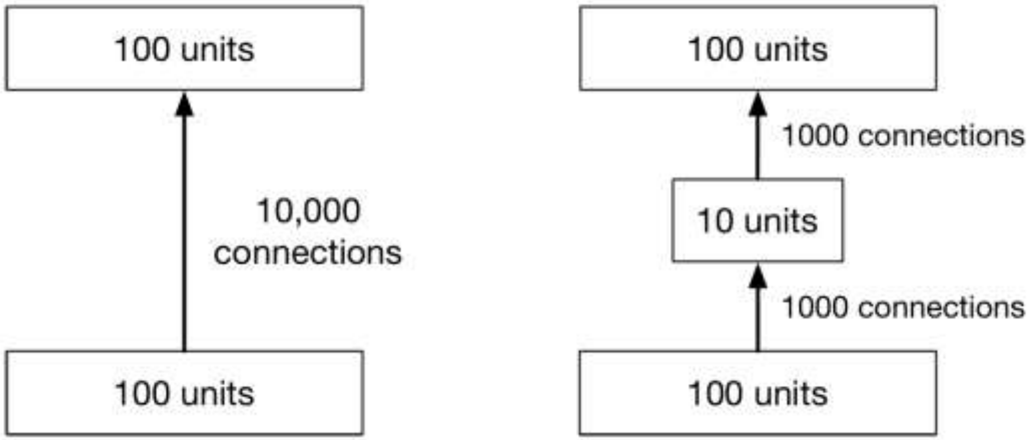
Outline

- Regularization in CNN training
  - Data Augmentation
  - Weight Regularization & Transfer Learning
  - Stochastic Regularization
  - Hyper-parameter optimization

Acknowledgement: Feifei Li's cs231n notes

Reducing # of Parameters

- Reducing the number of layers or the number of parameters per layer.
- Adding a linear bottleneck layer:



  - The first network is strictly more expressive than the second (i.e. it can represent a strictly larger class of functions). (Why?)
  - Remember how linear layers don't make a network more expressive? They might still improve generalization.

Weight Regularization

- L_2 regularization / weight decay
  - Encouraging the weights to be small in magnitude

$$\mathcal{E}_{\text{reg}} = \mathcal{E} + \lambda \mathcal{R} = \mathcal{E} + \frac{\lambda}{2} \sum_j w_j^2$$

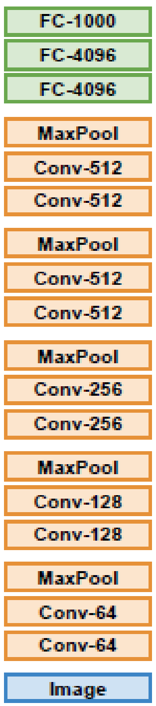
  - The gradient update can be interpreted as weight decay

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} - \alpha \left(\frac{\partial \mathcal{E}}{\partial \mathbf{w}} + \lambda \frac{\partial \mathcal{R}}{\partial \mathbf{w}} \right) \\ &= \mathbf{w} - \alpha \left(\frac{\partial \mathcal{E}}{\partial \mathbf{w}} + \lambda \mathbf{w} \right) \\ &= (1 - \alpha\lambda)\mathbf{w} - \alpha \frac{\partial \mathcal{E}}{\partial \mathbf{w}} \end{aligned}$$

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

Transfer Learning with CNNs

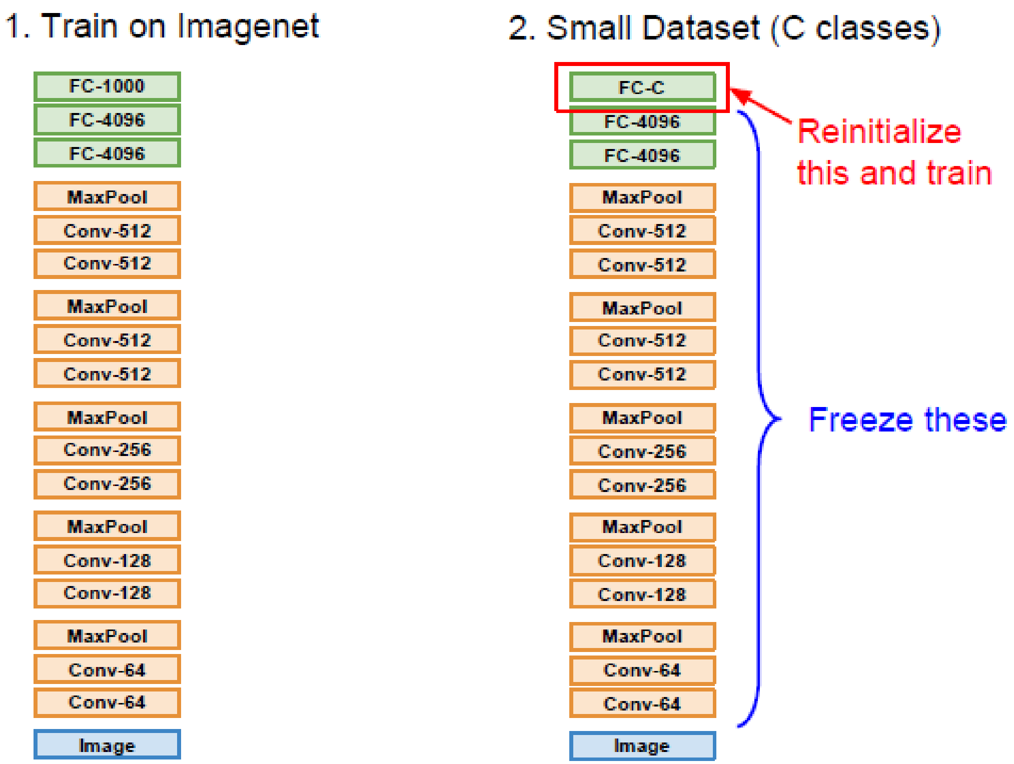
1. Train on Imagenet



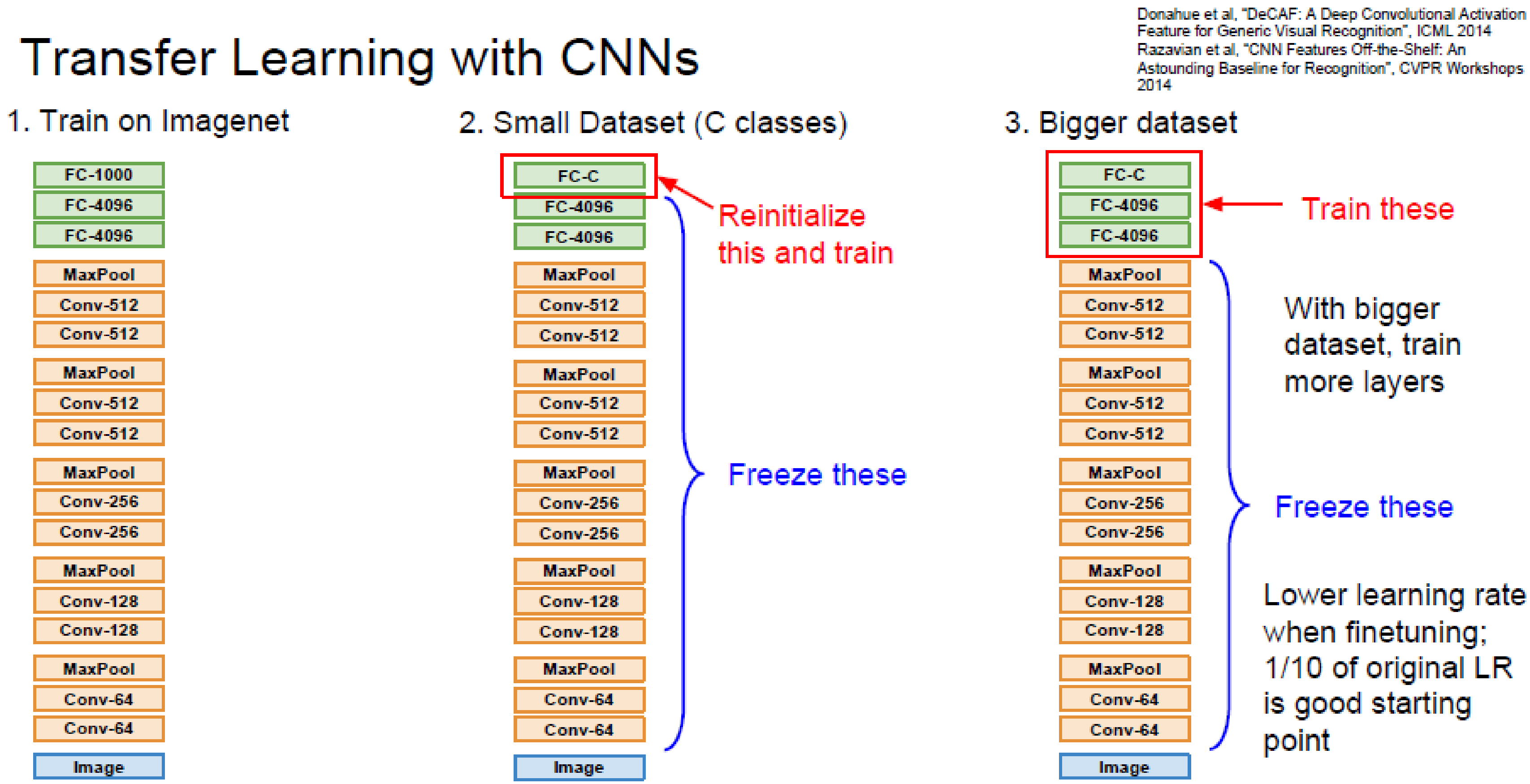
Transfer Learning

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

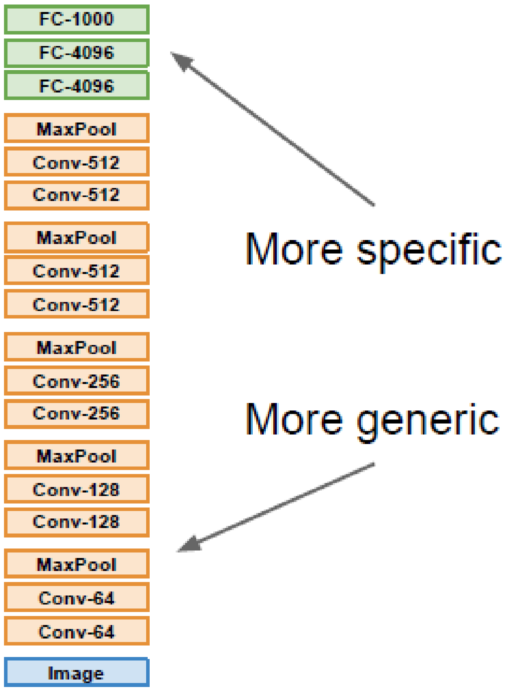
Transfer Learning with CNNs



Transfer Learning



Transfer Learning



	very similar dataset	**very different dataset**
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

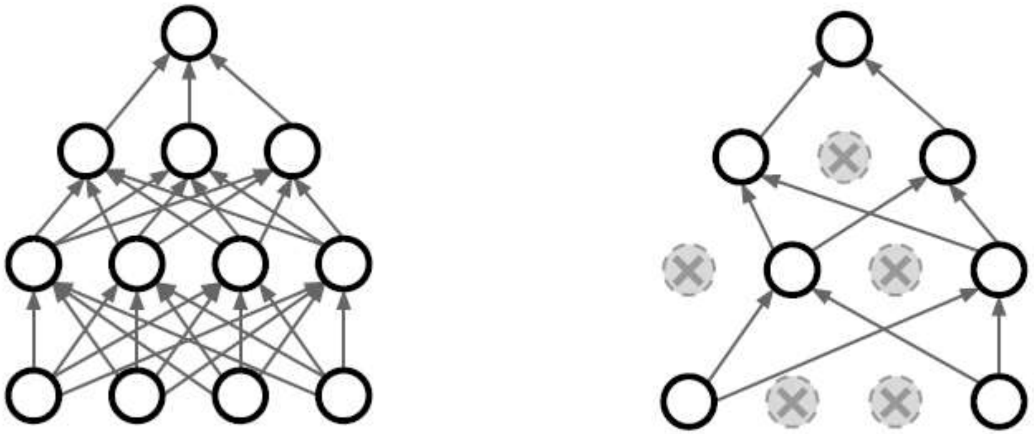
Outline

- Regularization in CNN training
  - Data Augmentation
  - Weight Regularization & Transfer Learning
  - Stochastic Regularization
  - Hyper-parameter optimization
- Network Architectures

Acknowledgement: Feifei Li's cs231n notes

Stochastic Regularization

- For a network to overfit, its computations need to be really precise. This suggests regularizing them by injecting noise into the computations, a strategy known as stochastic regularization.
- Dropout is a stochastic regularizer which randomly deactivates a subset of the units



Srivastava et al, "Dropout: A simple way to prevent neural networks from overfitting", JMLR 2014

Dropout

- Operations

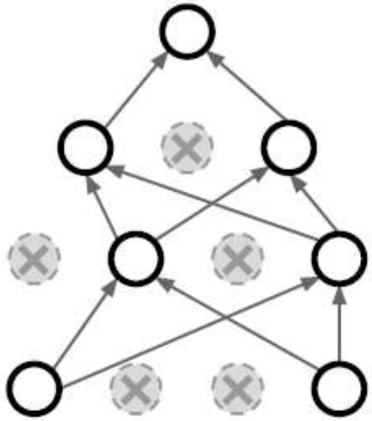
$$h_i = m_i \cdot \phi(z_i),$$

where m_i is a Bernoulli random variable, independent for each hidden unit.

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)
```

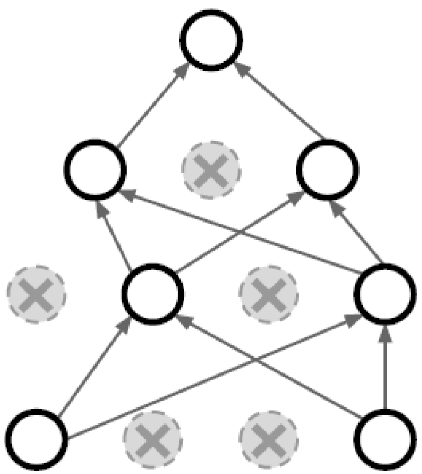
Example forward pass with a 3-layer network using dropout



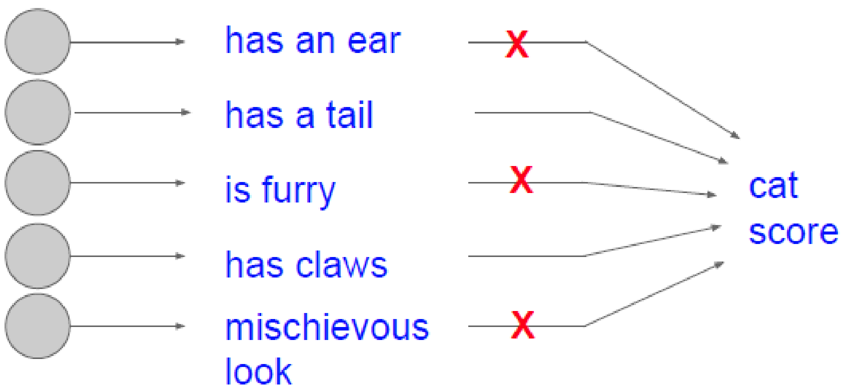
Understanding Dropout

Regularization: Dropout

How can this possibly be a good idea?

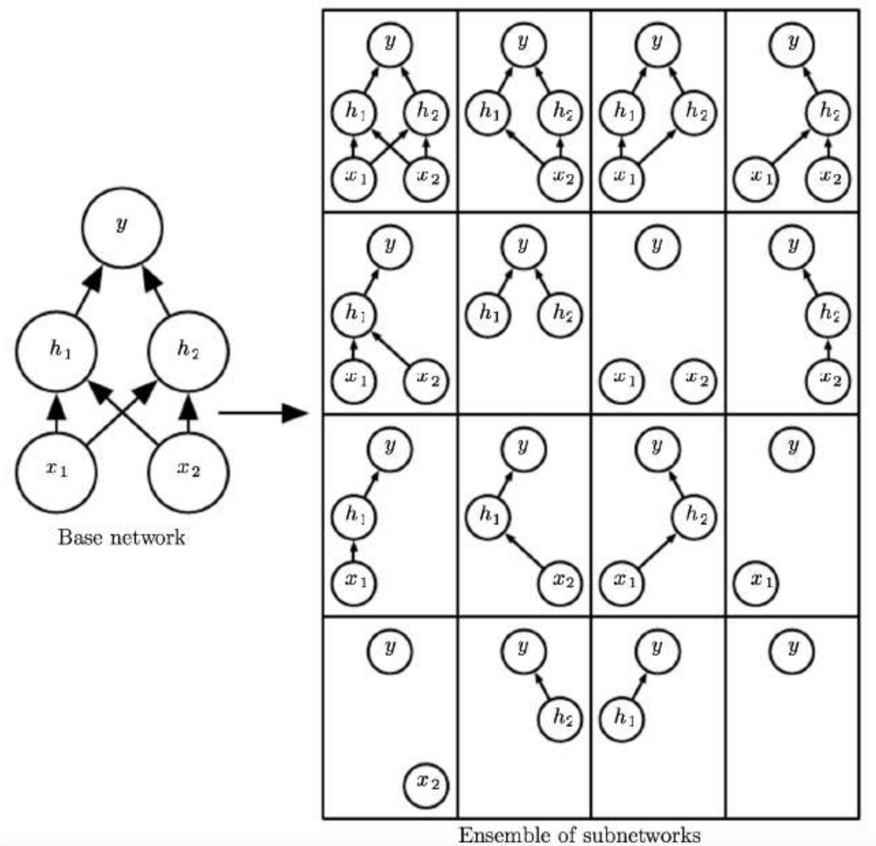


Forces the network to have a redundant representation;
Prevents co-adaptation of features



Understanding Dropout

- Dropout can be seen as training an ensemble of 2^D different architectures with shared weights (where D is the number of units):



— Goodfellow et al., *Deep Learning*

Dropout

- Dropout at test time

Dropout makes our output random! $\boxed{y} = f_W(\boxed{x}, \boxed{z})$

Output (label) — Input (image) — Random mask

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

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

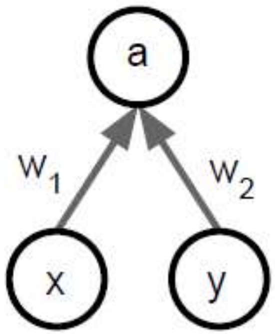
But this integral seems hard …

Dropout

- Dropout at test time

Want to approximate the integral

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



Consider a single neuron.

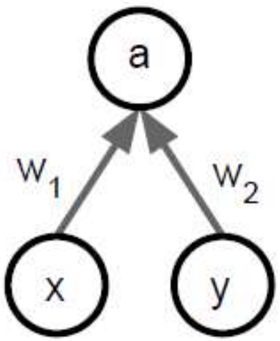
At test time we have: $E\left[a\right] = w_1x + w_2y$

Dropout

- Dropout at test time

Want to approximate the integral

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



Consider a single neuron.

At test time we have: $E\big[a\big] = w_1 x + w_2 y$

During training we have:
$$\begin{aligned} E\big[a\big] =& \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) \\ &+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\ =& \frac{1}{2}(w_1 x + w_2 y) \end{aligned}$$

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

Dropout

- Dropout at test time

```python
def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

At test time all neurons are active always
=> We must scale the activations so that for each neuron:
<u>output at test time</u> = <u>expected output at training time</u>

Dropout

- Implementation: Inverted dropout

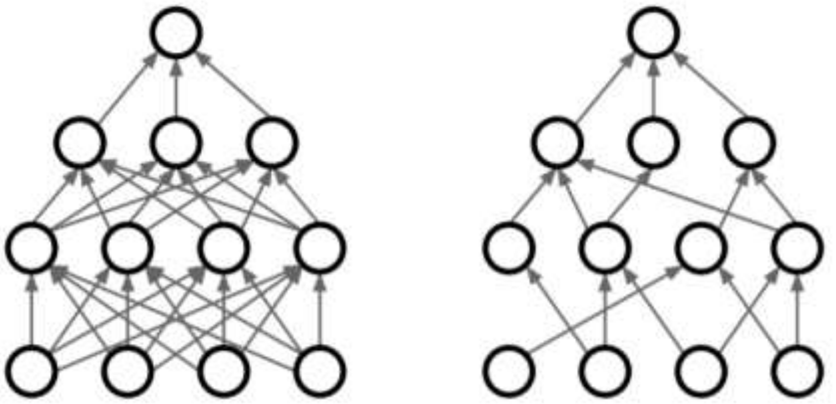
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

test time is unchanged!

Stochastic Regularization

- Lots of other stochastic regularizers have been proposed:
  - DropConnect drops connections instead of activations.

    - Training: Drop connections between neurons (set weights to 0)
    - Testing: Use all the connections

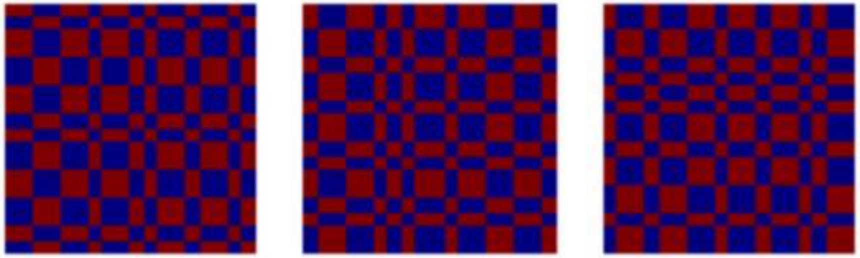


Wan et al, “Regularization of Neural Networks using DropConnect”, ICML 2013

Stochastic Regularization

- Lots of other stochastic regularizers have been proposed:
  - Fractional Pooling

    - Training: Use randomized pooling regions
    - Testing: Average predictions from several regions

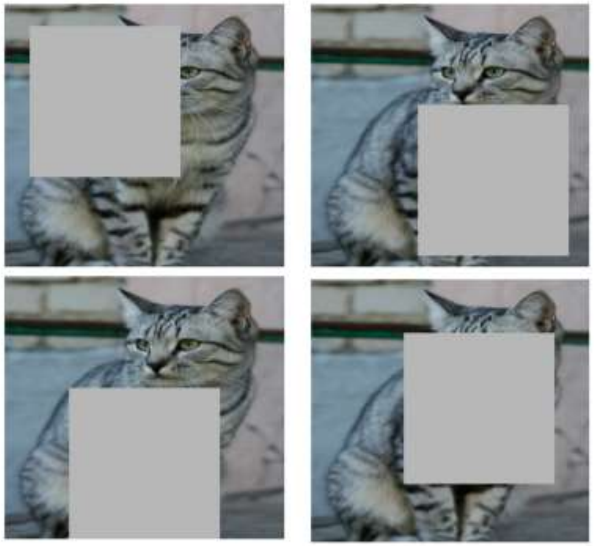


Graham, "Fractional Max Pooling", arXiv 2014

Stochastic Regularization

- Lots of other stochastic regularizers have been proposed:
  - Cutout

    - Training: Set random image regions to zero
    - Testing: Use full image predictions from several regions



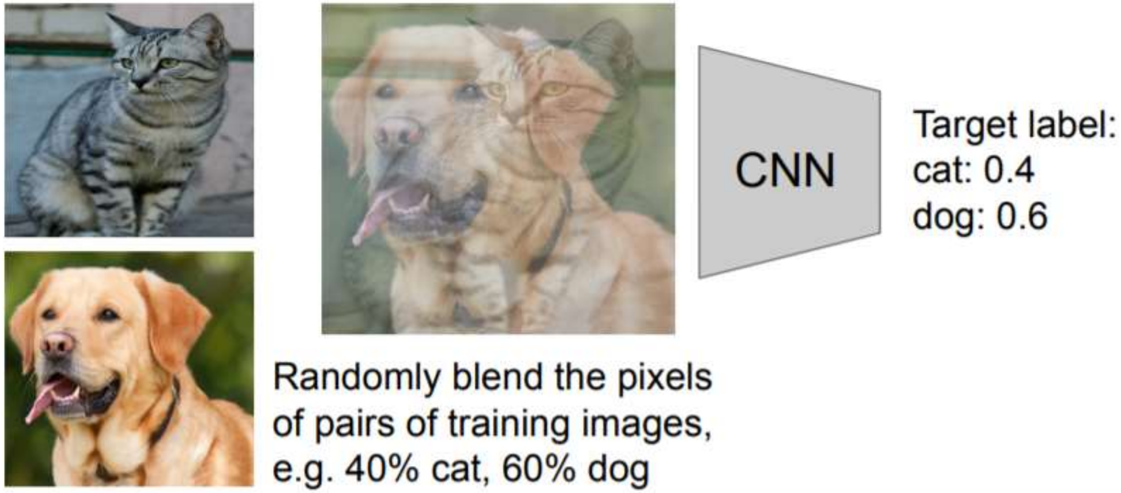
Works very well for small datasets like CIFAR, less common for large datasets like ImageNet

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

Stochastic Regularization

- Lots of other stochastic regularizers have been proposed:
  - Mixup

    - Training: Train on random blends of images
    - Testing: Use original images



Zhang et al, “mixup: Beyond Empirical Risk Minimization”, ICLR 2018

Stochastic Regularization

- Lots of other stochastic regularizers have been proposed:
  - Training: Add random noise
  - Testing: Marginalize over the noise
- In practice
  - 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

Outline

- Regularization in CNN training
  - Data Augmentation
  - Weight Regularization & Transfer Learning
  - Stochastic Regularization
  - Hyper-parameter optimization

Acknowledgement: Feifei Li's cs231n notes

Hyperparameter optimization

- (Cross-)validation strategy

coarse -> fine cross-validation in stages

First stage: only a few epochs to get rough idea of what params work
Second stage: longer running time, finer search
… (repeat as necessary)

Tip for detecting explosions in the solver:
If the cost is ever > 3 * original cost, break out early

Hyperparameter optimization

For example: run coarse search for 5 epochs

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)

      trainer = ClassifierTrainer()
      model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
      trainer = ClassifierTrainer()
      best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                   model, two_layer_net,
                                   num_epochs=5, reg=reg,
                                   update='momentum', learning_rate_decay=0.9,
                                   sample_batches = True, batch_size = 100,
                                   learning_rate=lr, verbose=False)
```

note it's best to optimize in log space!

nice

```
val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val_acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

Hyperparameter optimization

Now run finer search...

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)
```

adjust range

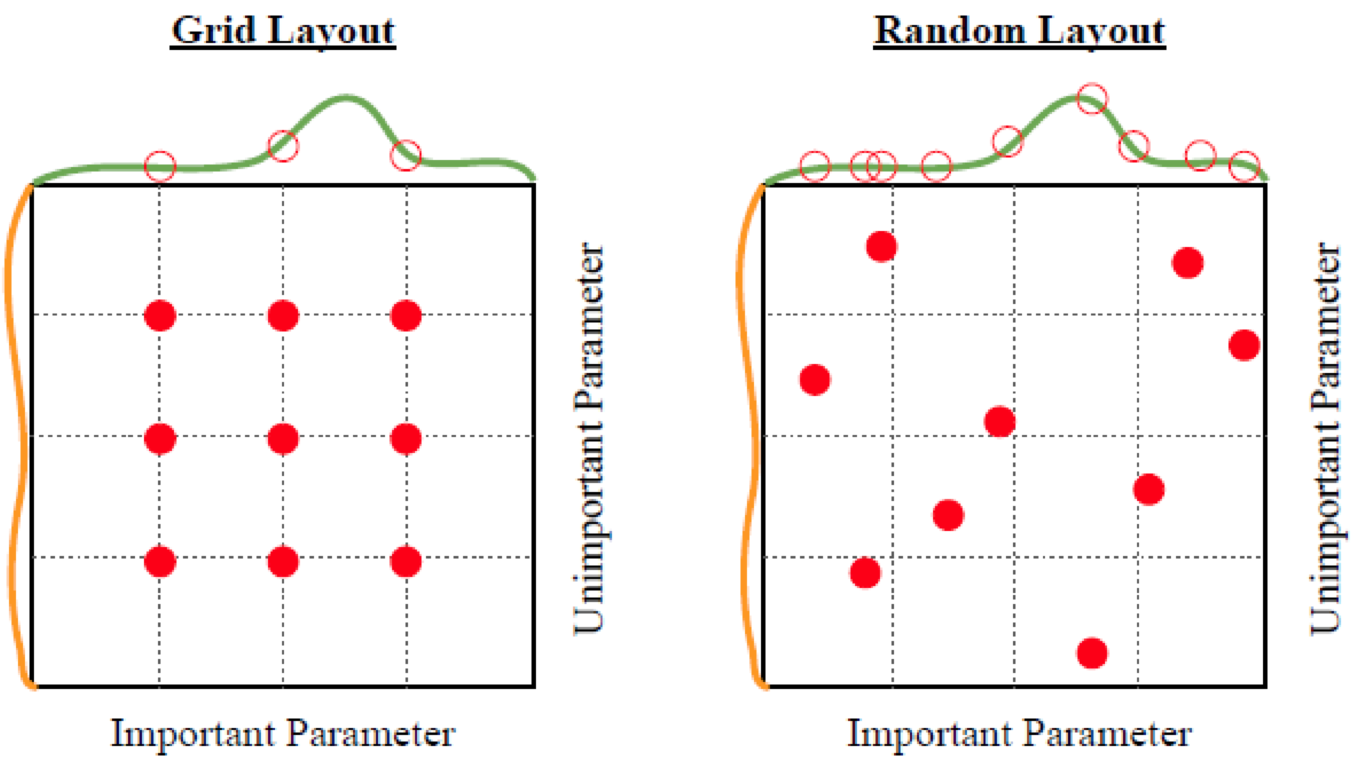
```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-4, 0)
      lr = 10**uniform(-3, -4)
```

```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

53% - relatively good for a 2-layer neural net with 50 hidden neurons.

Hyperparameter optimization

- Random search vs. Grid search



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

Hyperparameter optimization

- **Hyperparameters to play with:**
  - network architecture
  - learning rate, its decay schedule, update type
  - regularization (L2/Dropout strength)

- Other hyperparameter optimization methods
  - Shahriari, et al. "Taking the human out of the loop: A review of Bayesian optimization." Proceedings of the IEEE 104.1 (2016): 148-175.

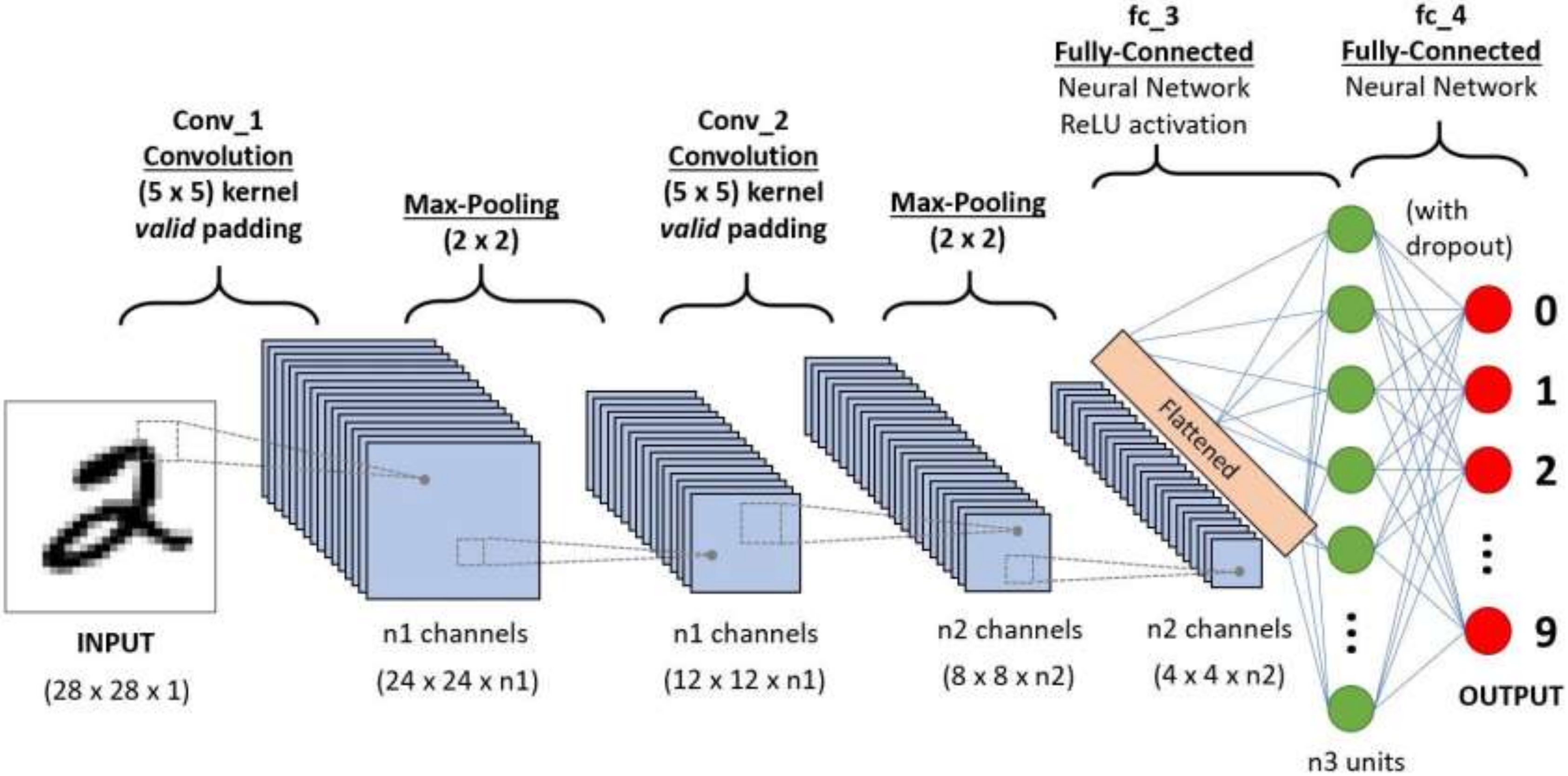
Outline

- CNN architectures
  - Sequential structure: LeNet/AlexNet/VGGNet
  - Parallel branches: GoogLeNet
  - Residual structure: ResNet/DenseNet
  - Network Architecture Search

Acknowledgement: Zemel et al’s CSC411 and Feifei Li et al’s cs231n notes

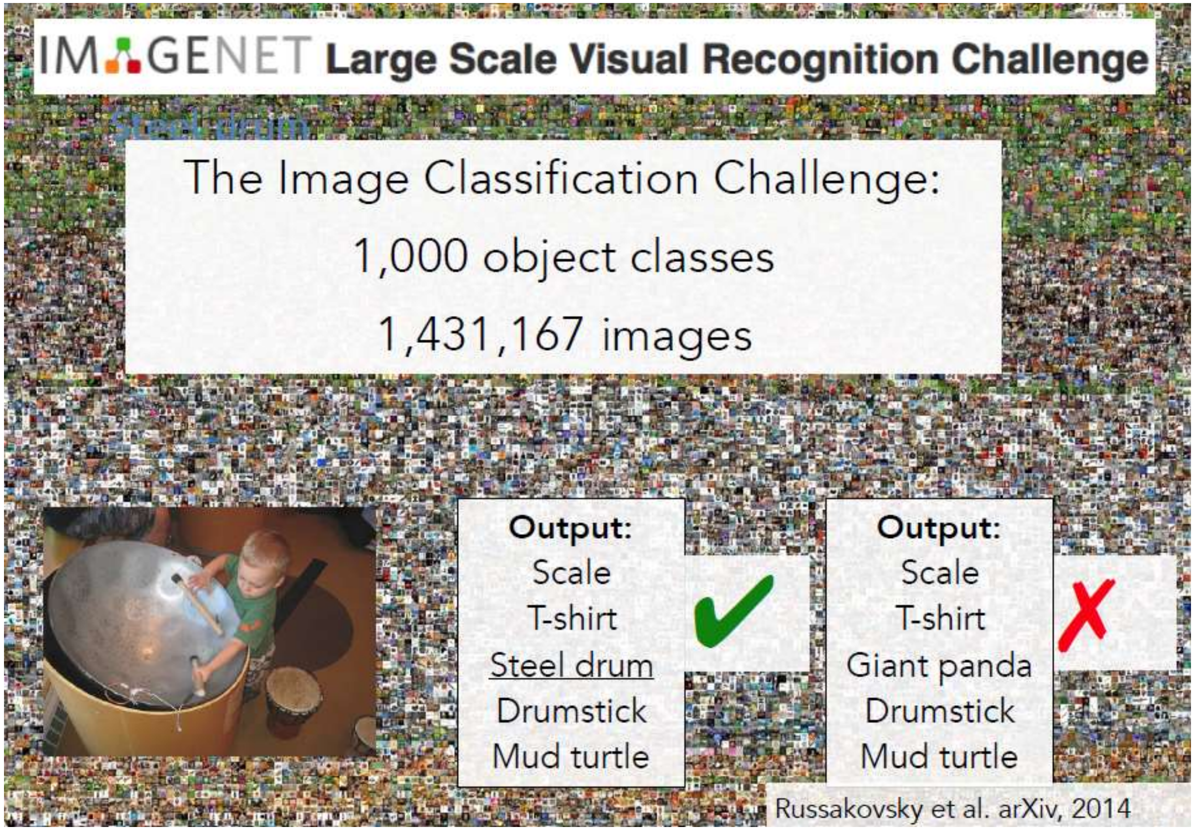
LeNet-5

- Handwritten digit recognition
- LeCun et al., 1998

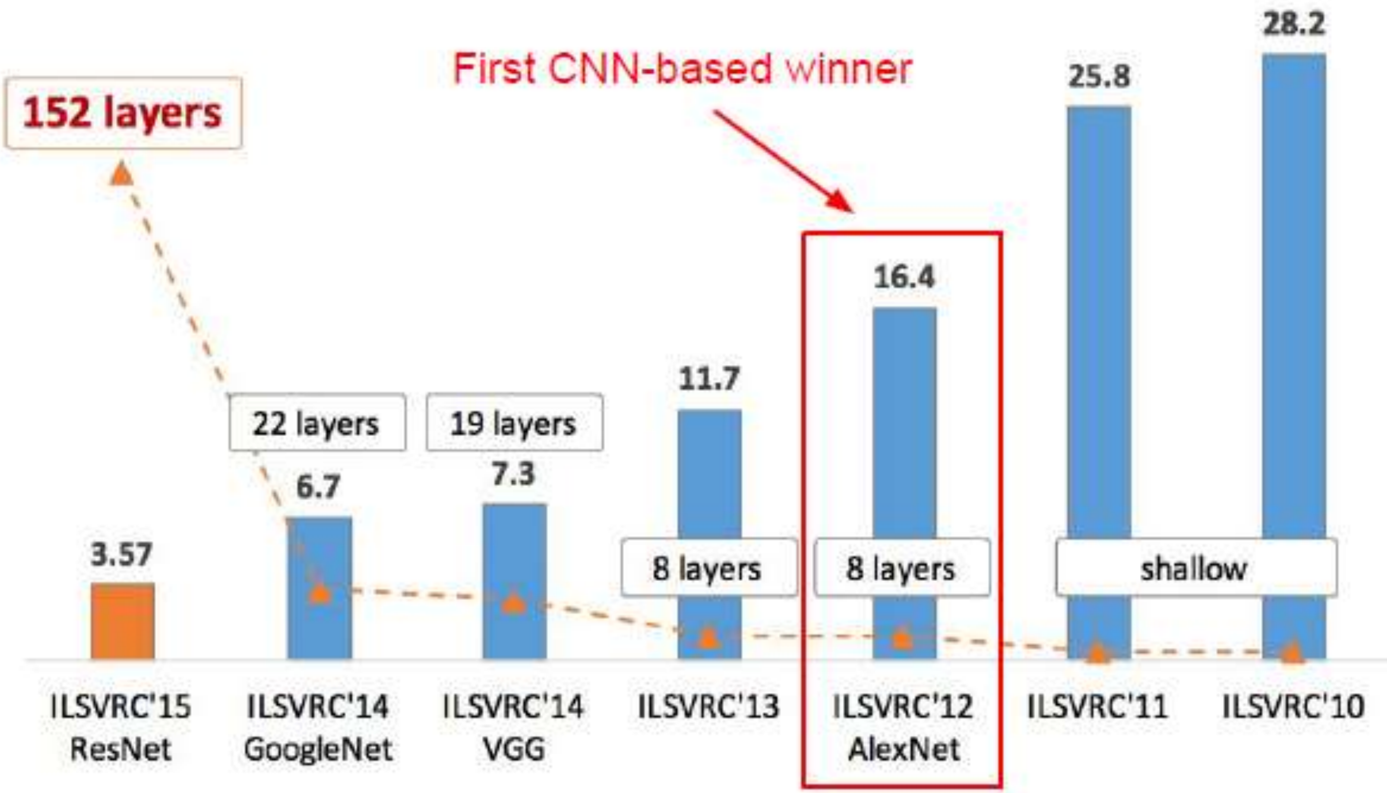


Background: Image/Object Classification

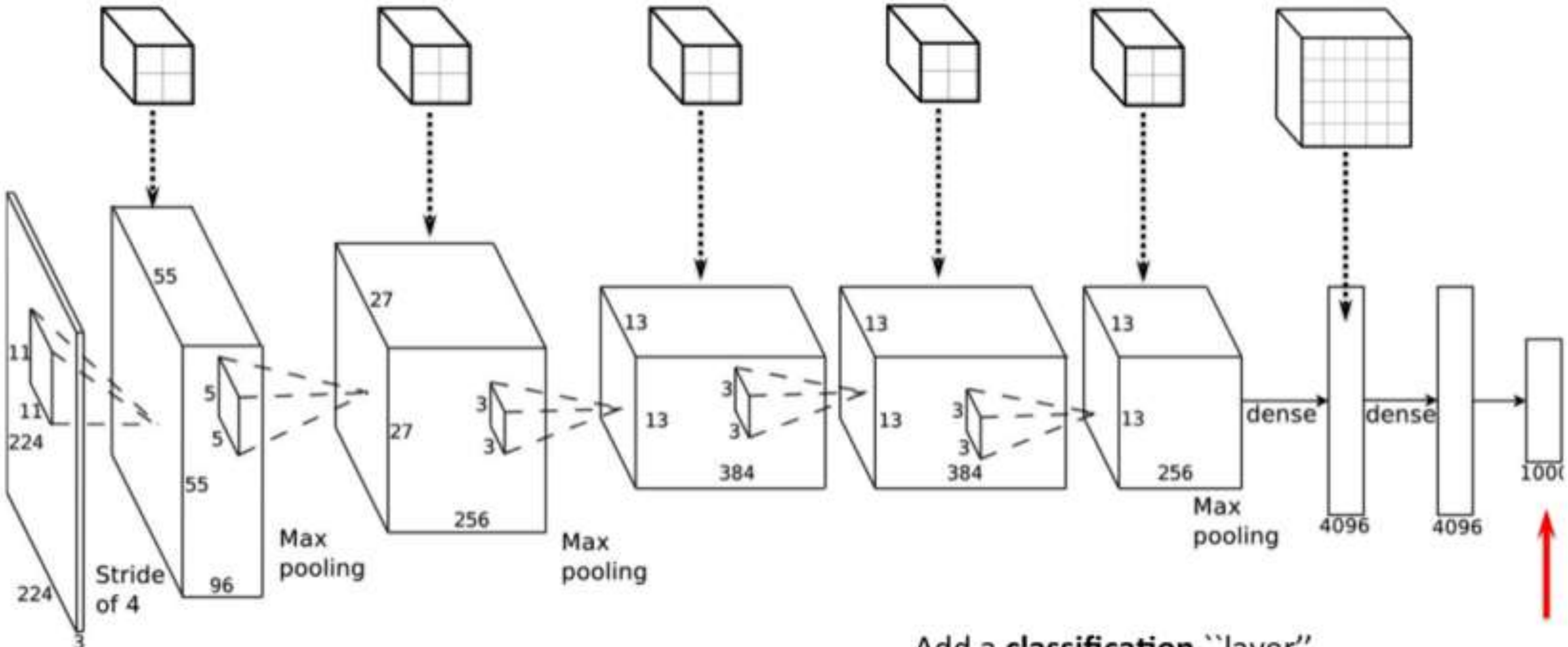
- Problem Setup
  - Input: Image
  - Output: Object class



ImageNet (ILSVRC)



AlexNet



Add a **classification** ``layer''.

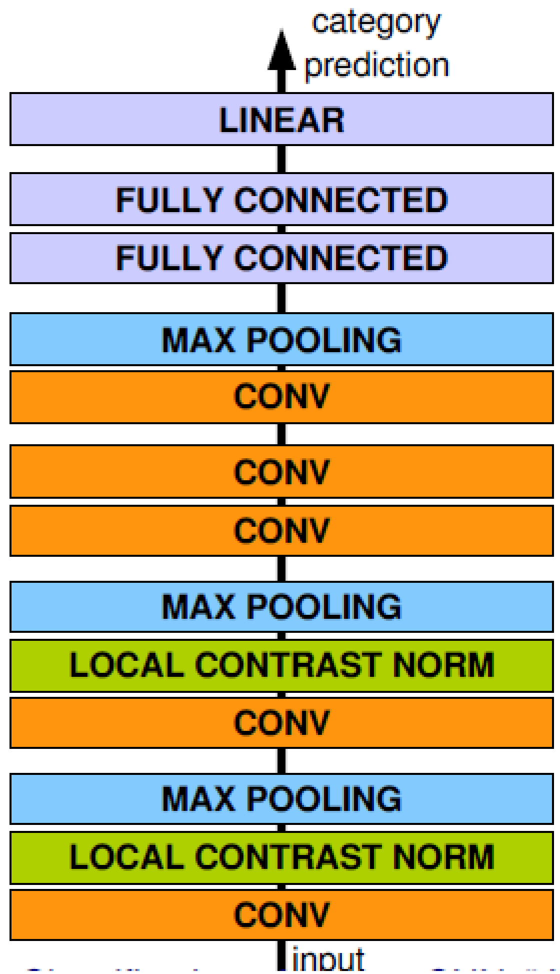
For an input image, the value in a particular dimension of this vector tells you the probability of the corresponding object class.

- 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%

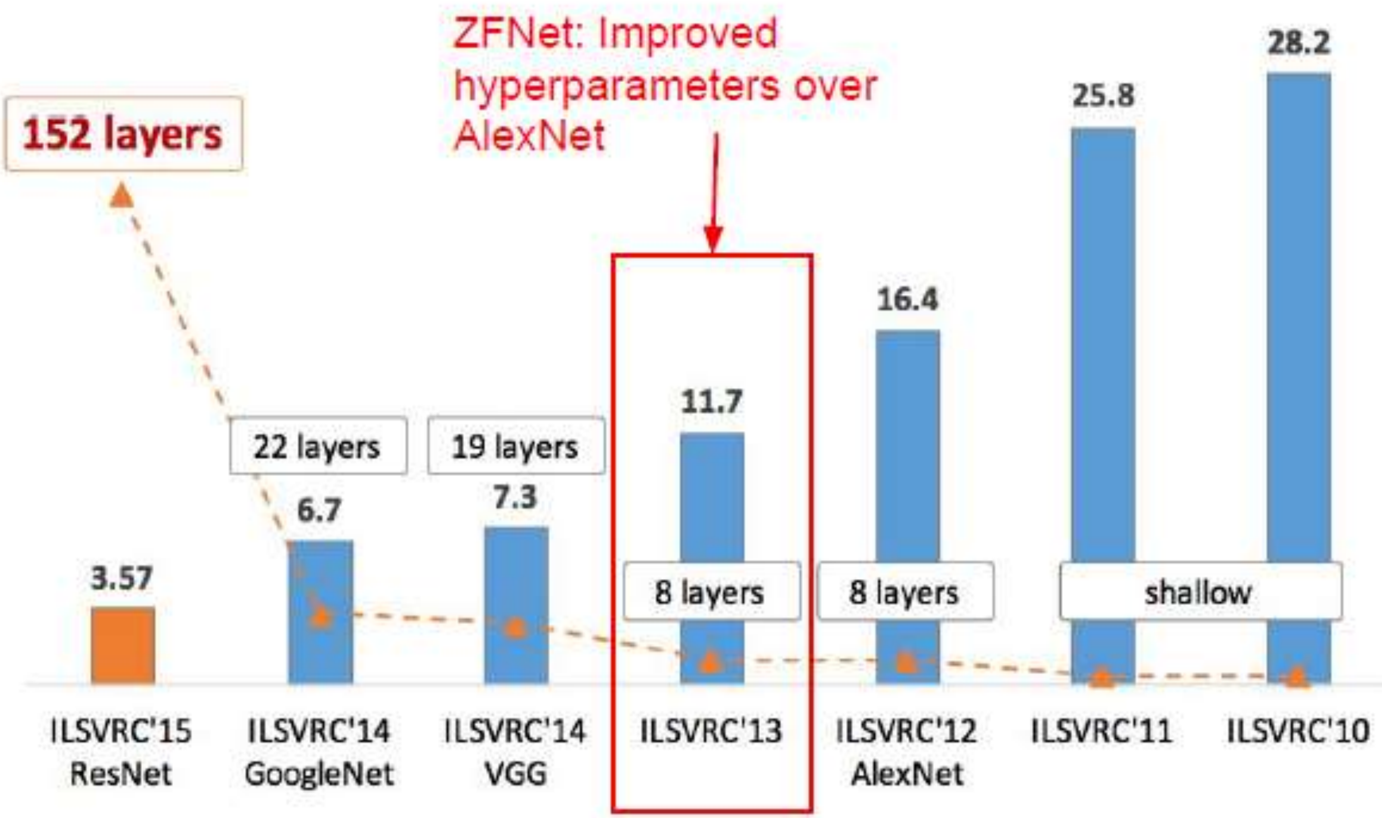
AlexNet

- Deeper network structure
  - More convolution layers
  - Local contrast normalization
  - ReLu instead of Tanh
  - Dropout as regularization

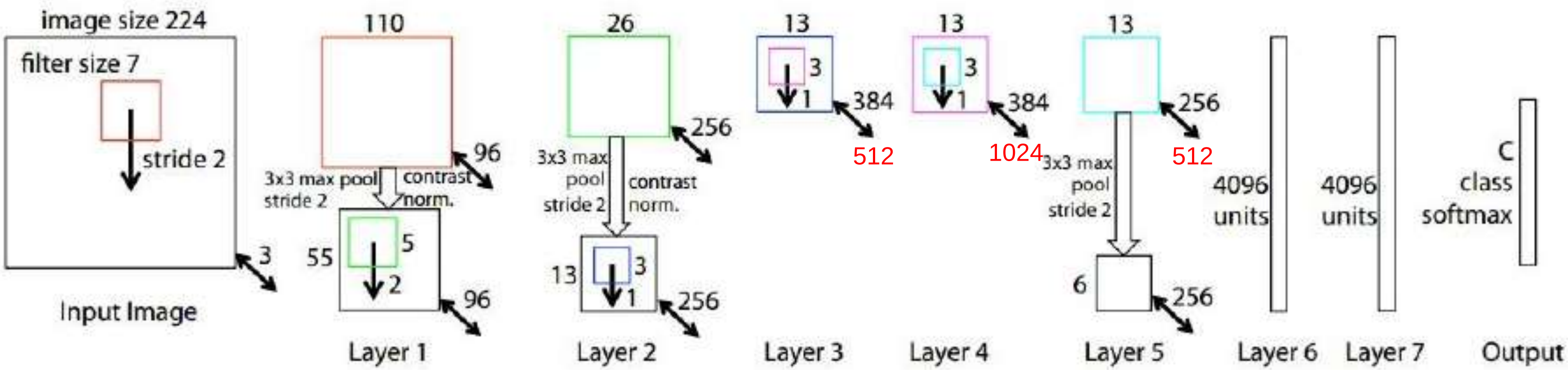
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] 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)



ImageNet (ILSVRC)



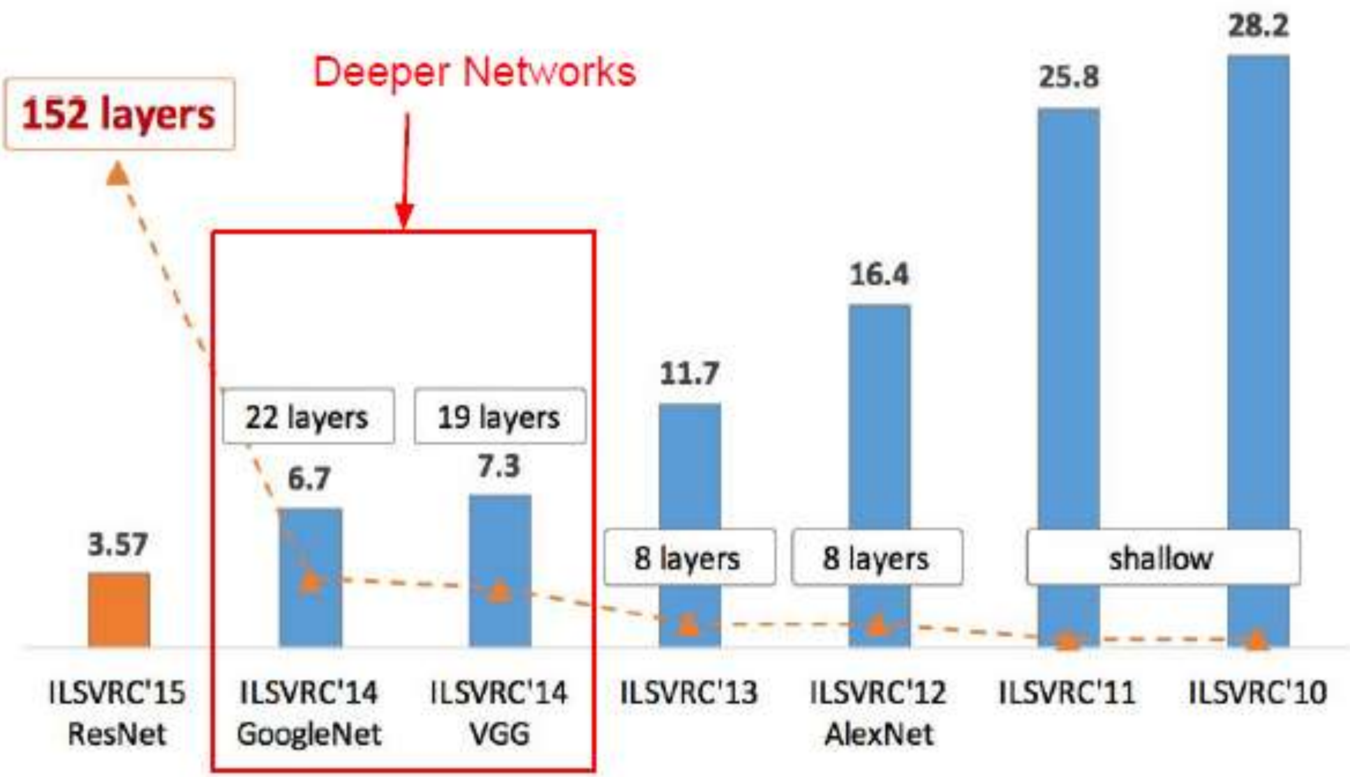
ZFNet



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 (ILSVRC)



Case Study: VGGNet

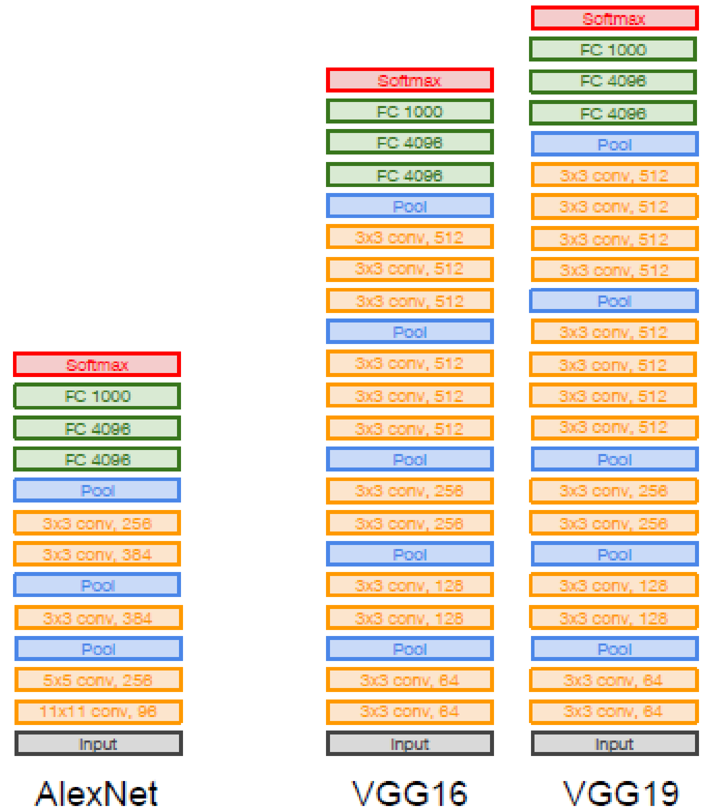
[Simonyan and Zisserman, 2014]

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



VGGNet

Case Study: VGGNet

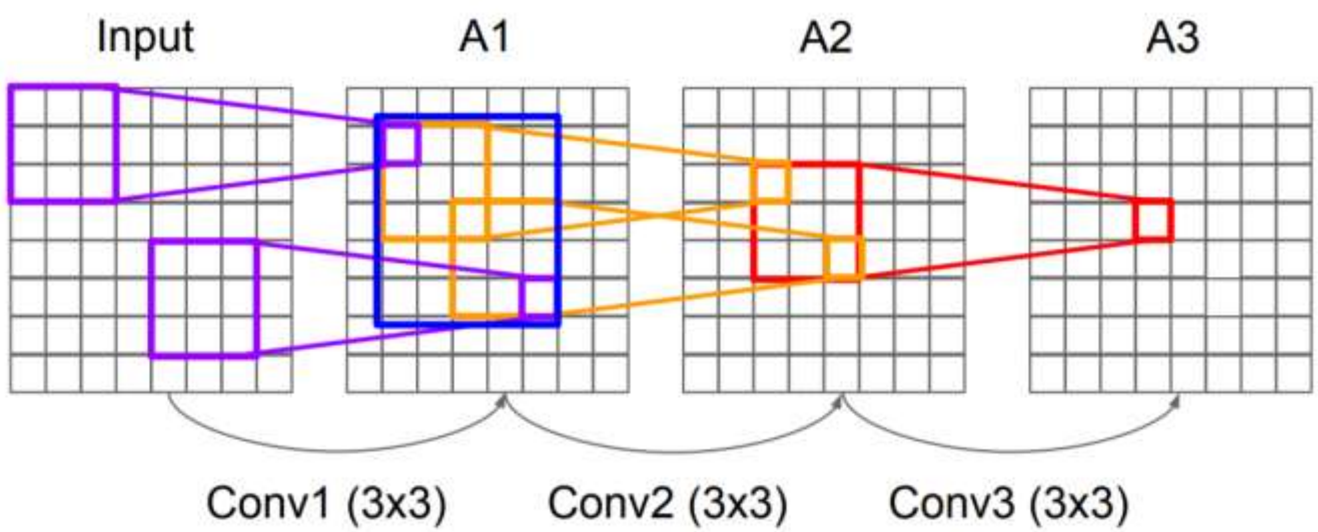
[Simonyan and Zisserman, 2014]

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: $3 * (3^2C^2)$ vs. 7^2C^2 for C channels per layer

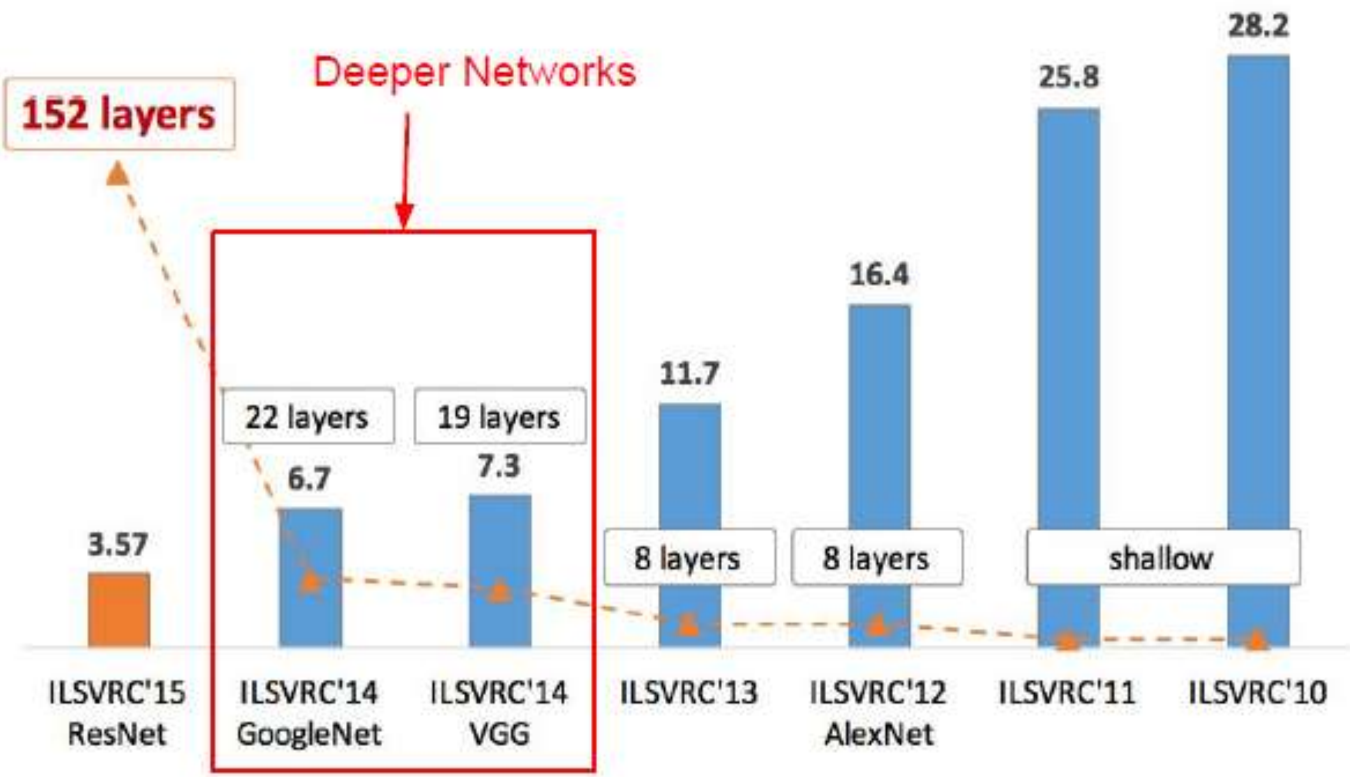


Outline

- CNN architectures
  - Sequential structure: LeNet/AlexNet/VGGNet
  - Parallel branches: GoogLeNet
  - Residual structure: ResNet/DenseNet
  - Network Architecture Search

Acknowledgement: Zemel et al's CSC411 and Feifei Li et al's cs231n notes

ImageNet (ILSVRC)



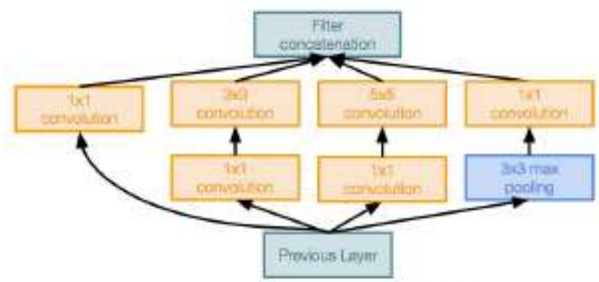
GoogLeNet

Case Study: GoogLeNet

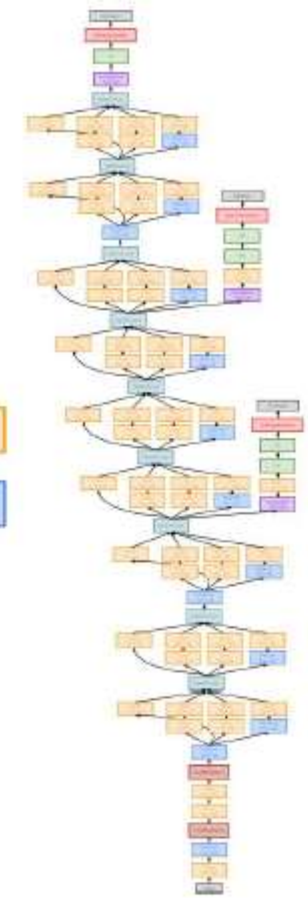
[Szegedy et al., 2014]

Deeper networks, with computational efficiency

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



Inception module

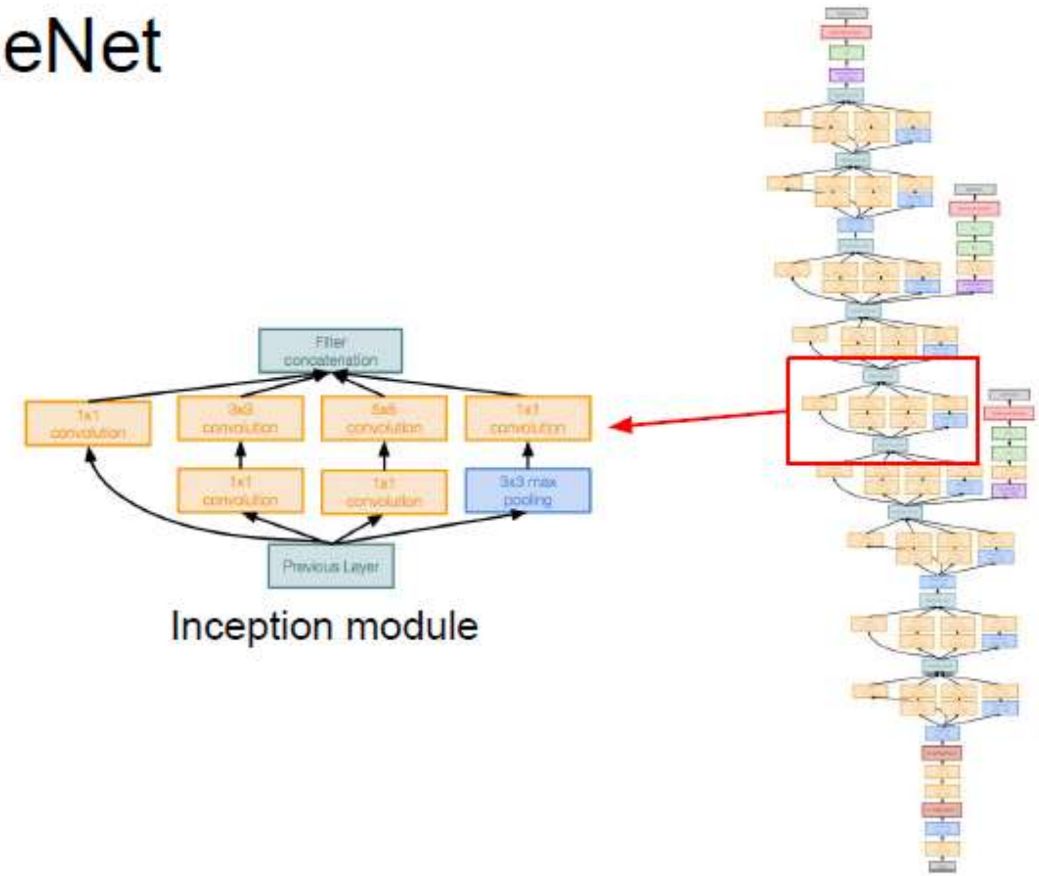


GoogLeNet

Case Study: GoogLeNet

[Szegedy et al., 2014]

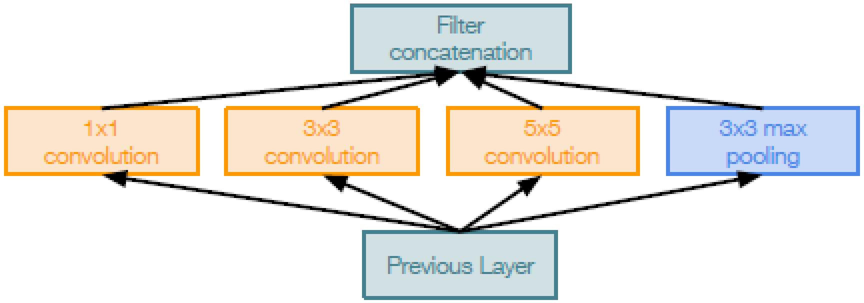
"Inception module": design a good local network topology (network within a network) and then stack these modules on top of each other



Inception module

GoogLeNet

- Inception Module



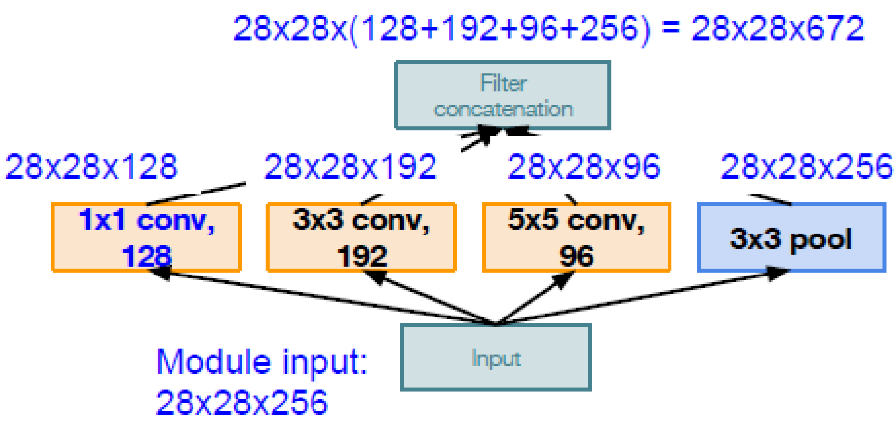
Naive Inception module

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 depth-wise

GoogLeNet

- Inception Module



Naive Inception module

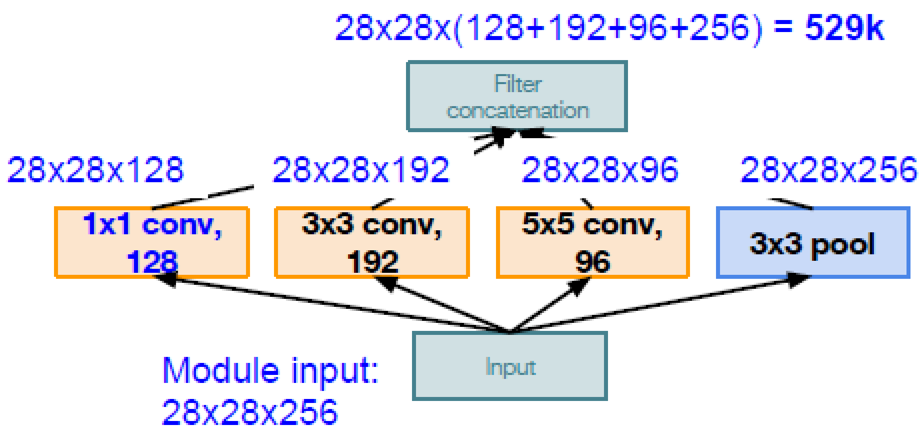
Conv Ops:
[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x256
[5x5 conv, 96] 28x28x96x5x5x256
Total: 854M ops

Very expensive compute

Pooling layer also preserves feature depth, which means total depth after concatenation can only grow at every layer!

GoogLeNet

- Inception Module

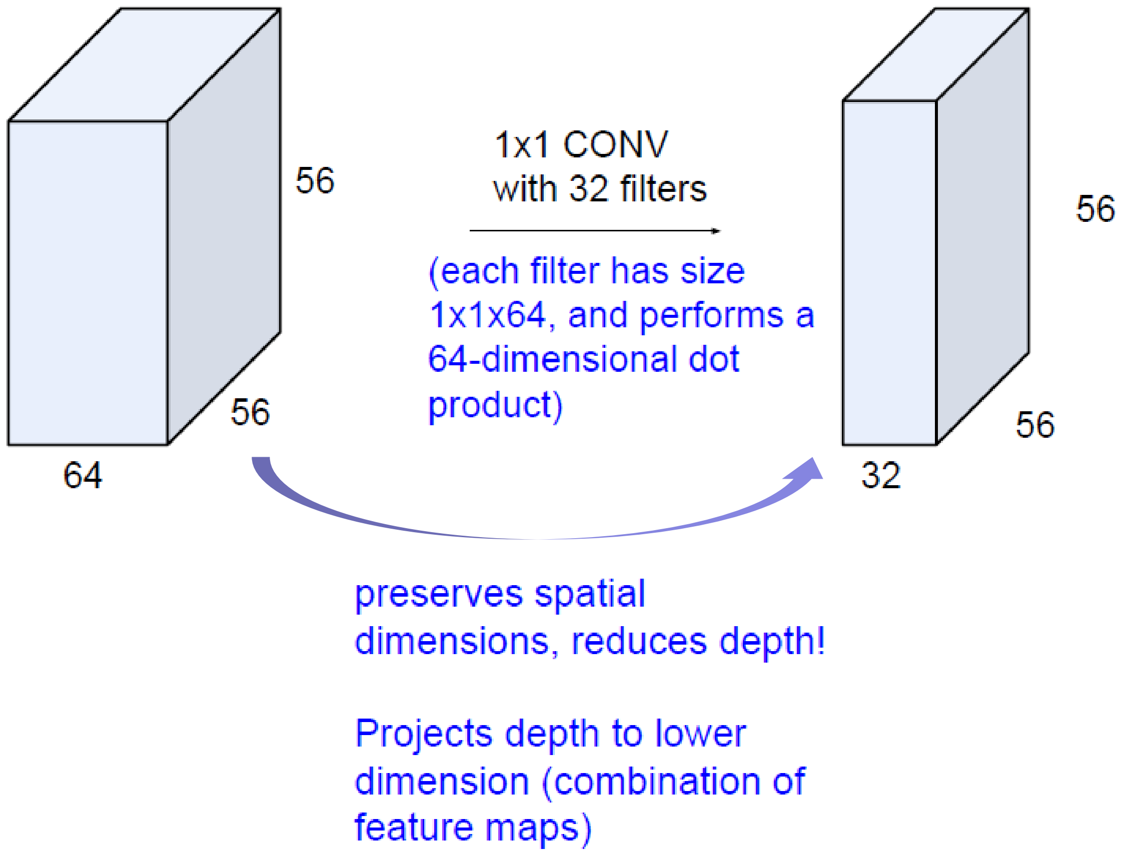


Naive Inception module

Solution: “bottleneck” layers that use 1x1 convolutions to reduce feature depth

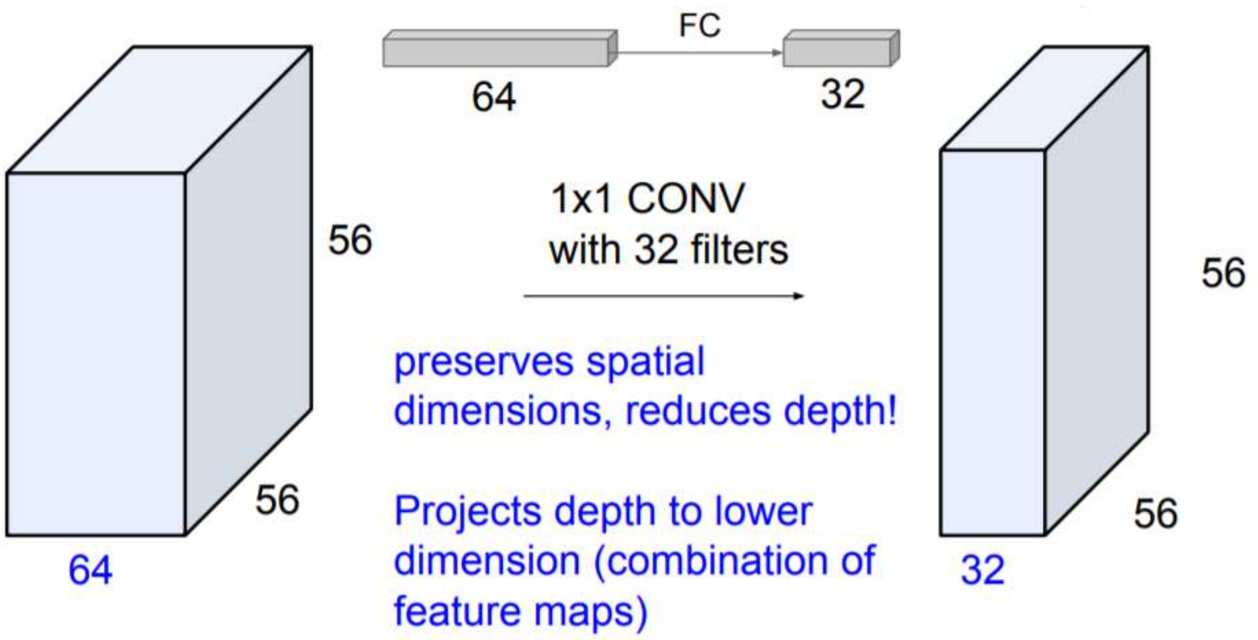
GoogLeNet

- Bottleneck layer



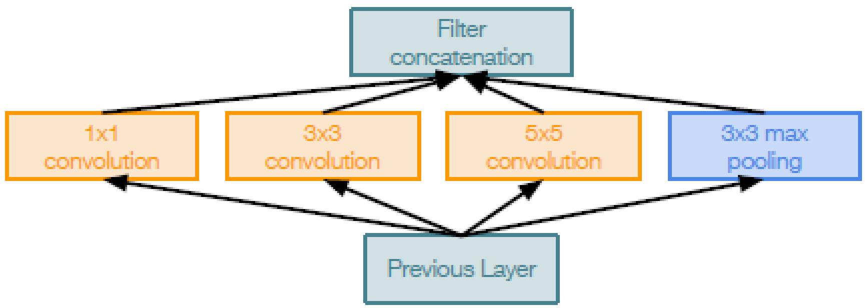
GoogLeNet

- 1x1 Convolutions
  - Alternatively, interpret it as applying the same FC layer on each input pixel

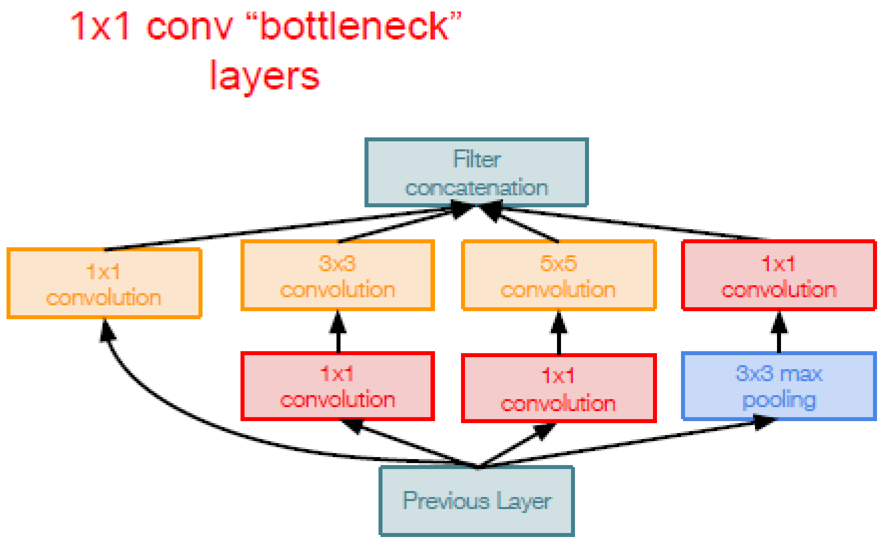


GoogLeNet

- Inception Module



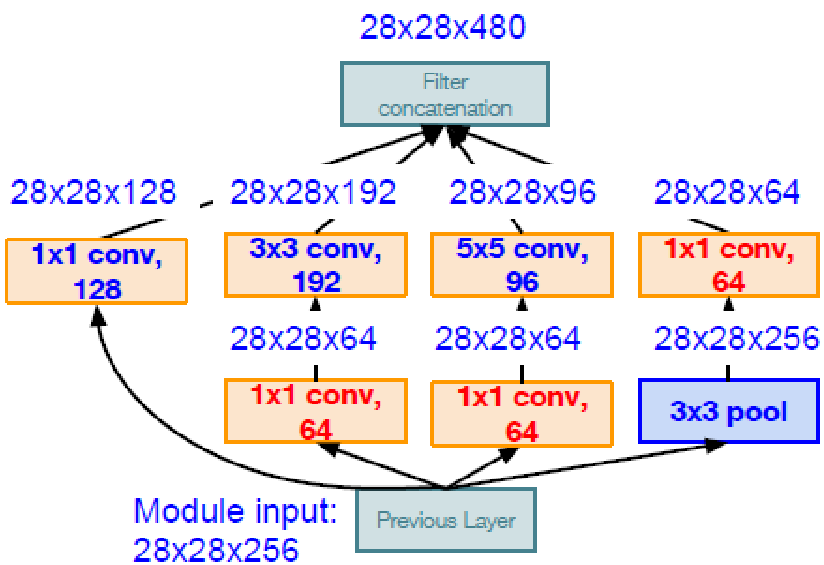
Naive Inception module



Inception module with dimension reduction

GoogLeNet

- Inception Module



Inception module with dimension reduction

Conv Ops:

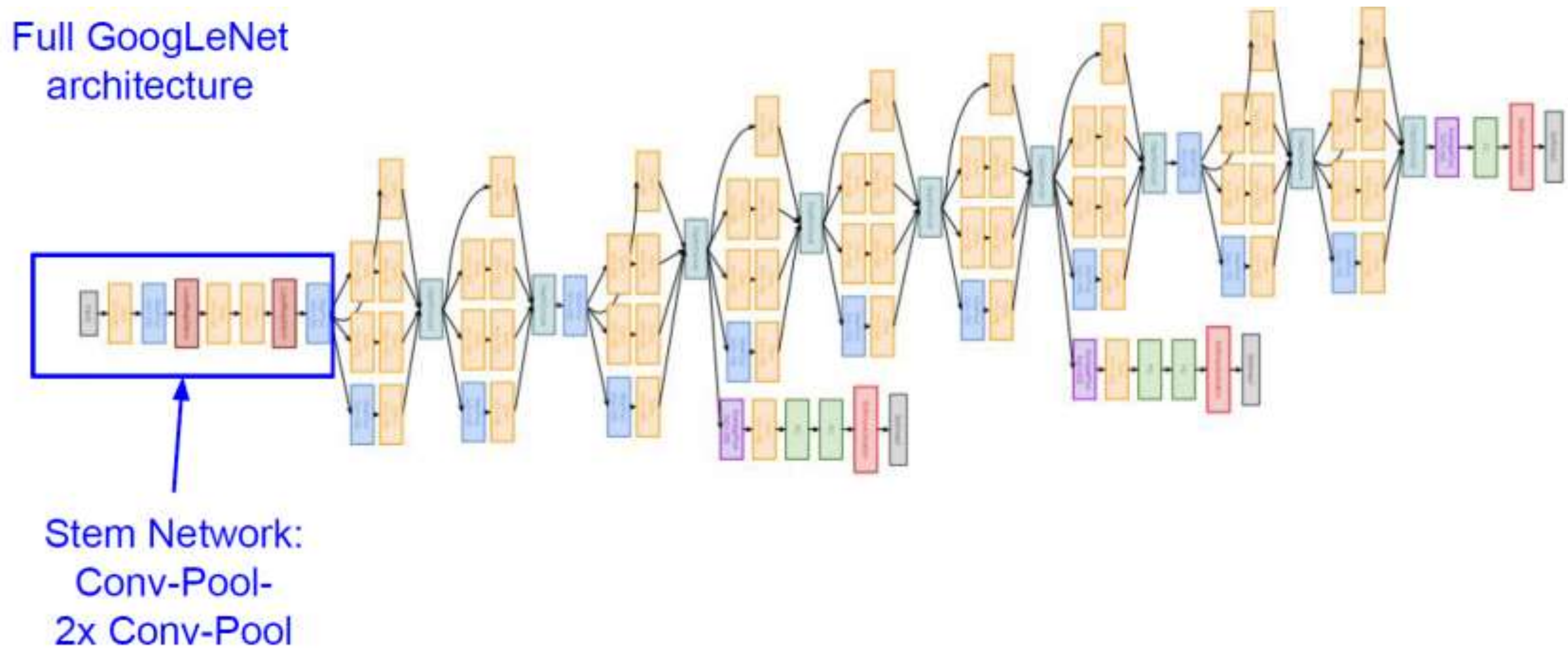
[1x1 conv, 64] 28x28x64x1x1x256
[1x1 conv, 64] 28x28x64x1x1x256
[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x64
[5x5 conv, 96] 28x28x96x5x5x64
[1x1 conv, 64] 28x28x64x1x1x256

Total: 358M ops

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

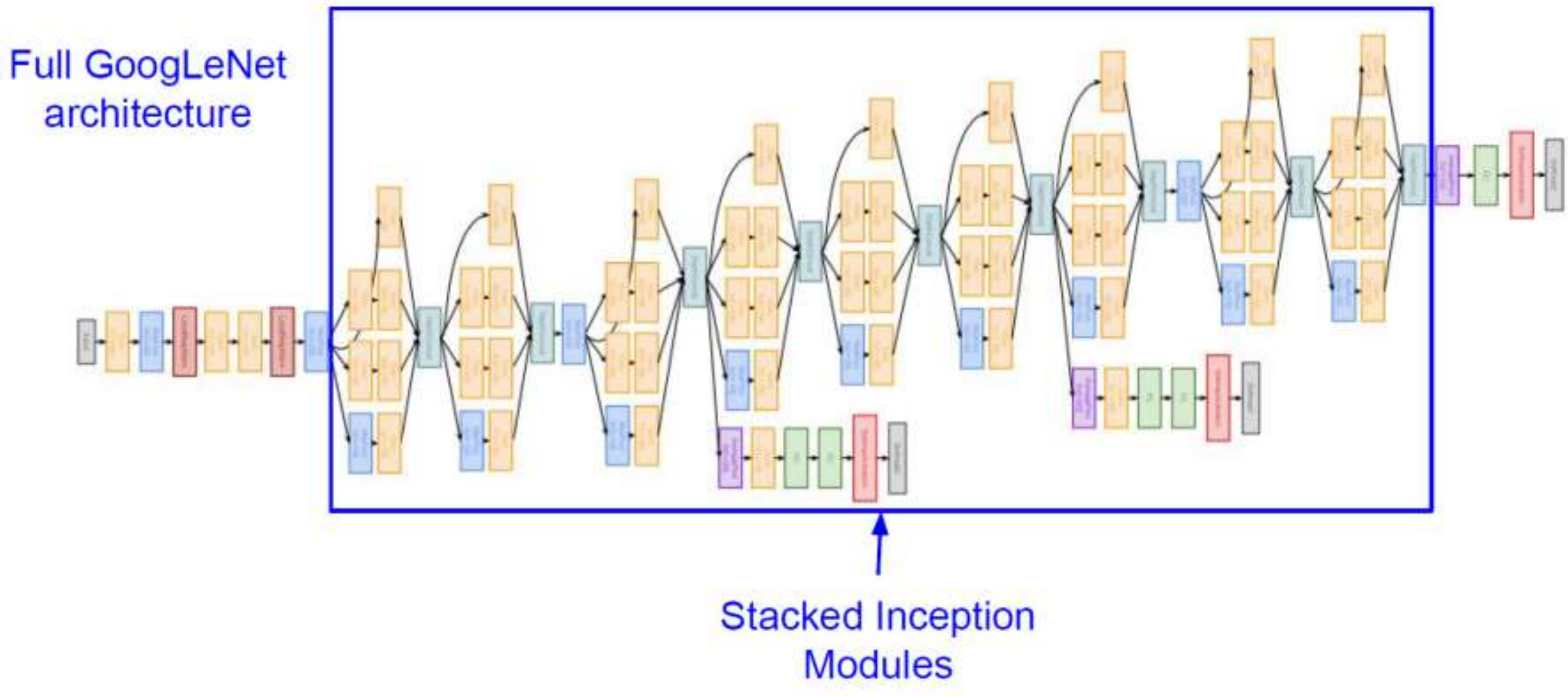
GoogLeNet

- Overall network structure



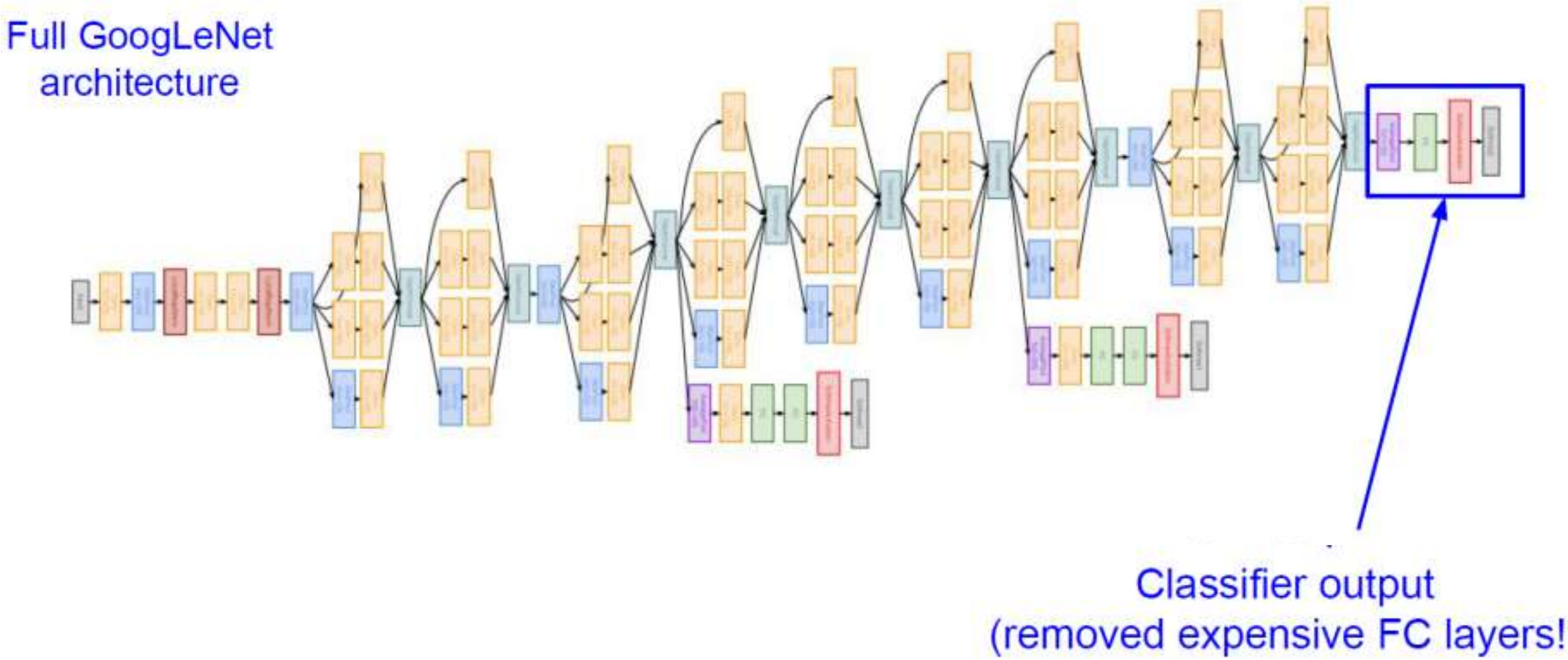
GoogLeNet

- Overall network structure



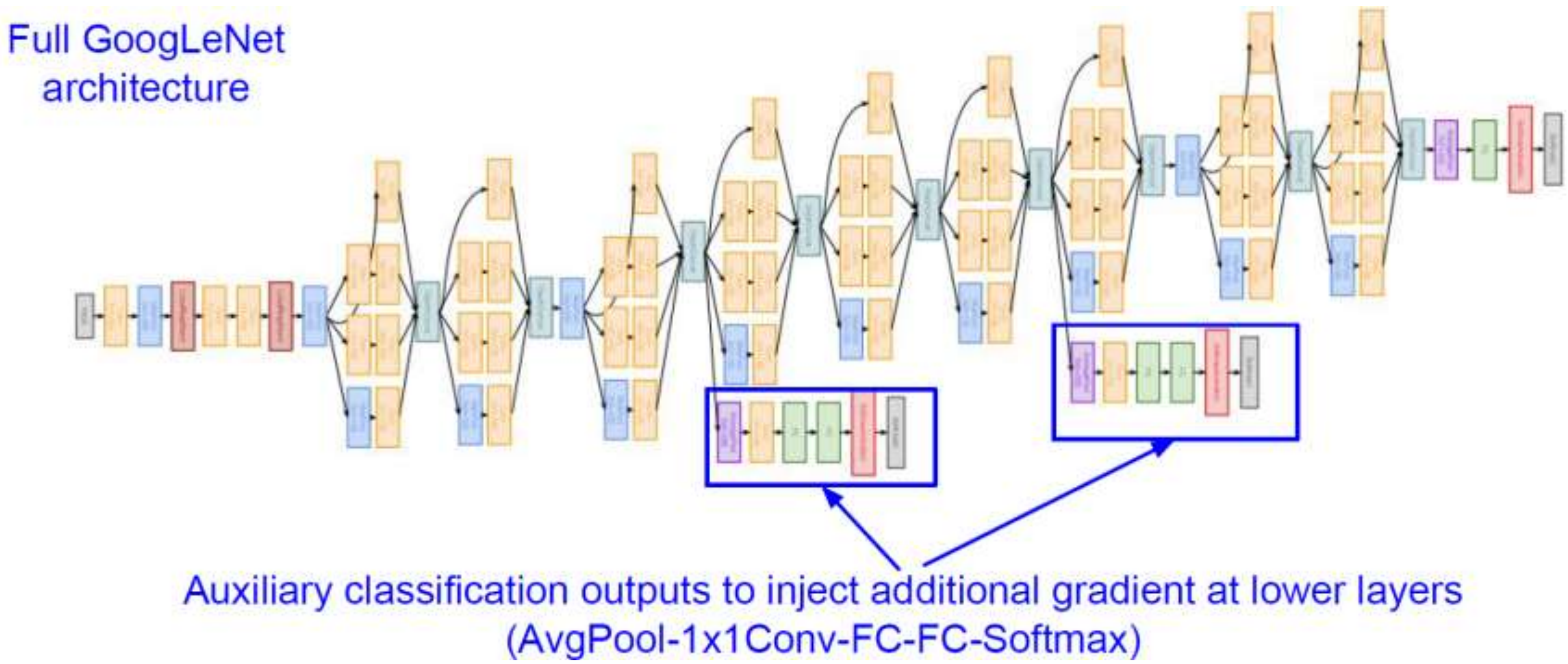
GoogLeNet

- Overall network structure



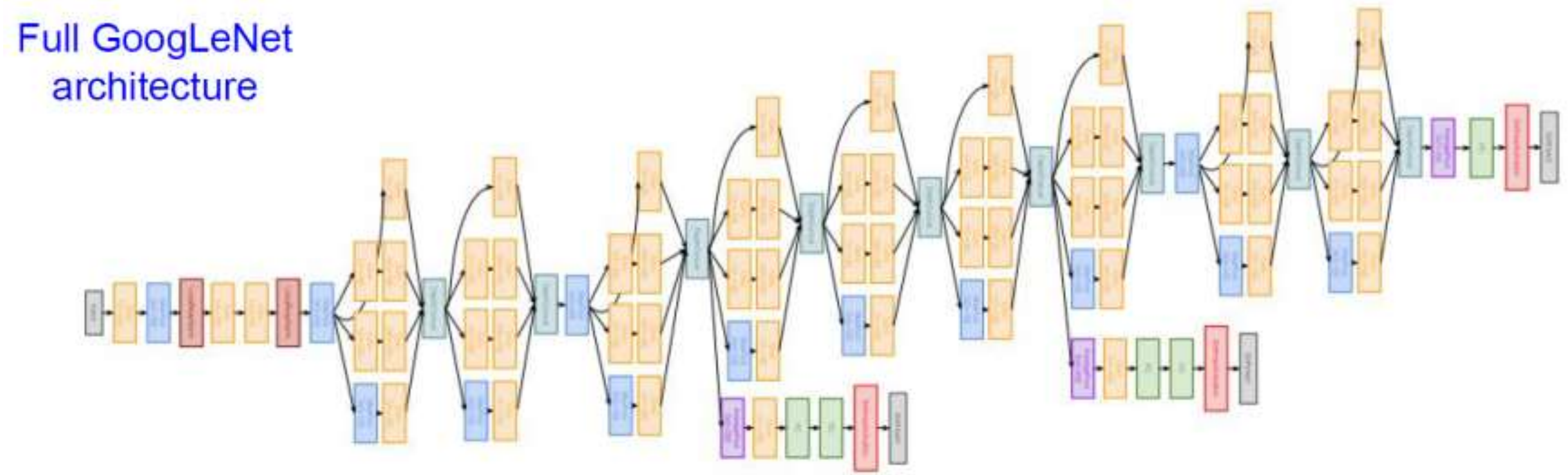
GoogLeNet

- Overall network structure



GoogLeNet

- Overall network structure



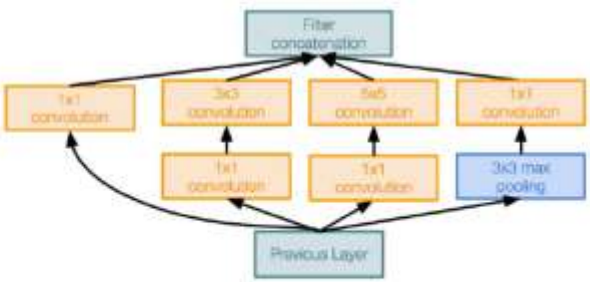
22 total layers with weights (including each parallel layer in an Inception module)

GoogLeNet

- Summary

Deeper networks, with computational efficiency

- 22 layers
- Efficient “Inception” module
- No FC layers
- 12x less params than AlexNet
- ILSVRC’14 classification winner (6.7% top 5 error)



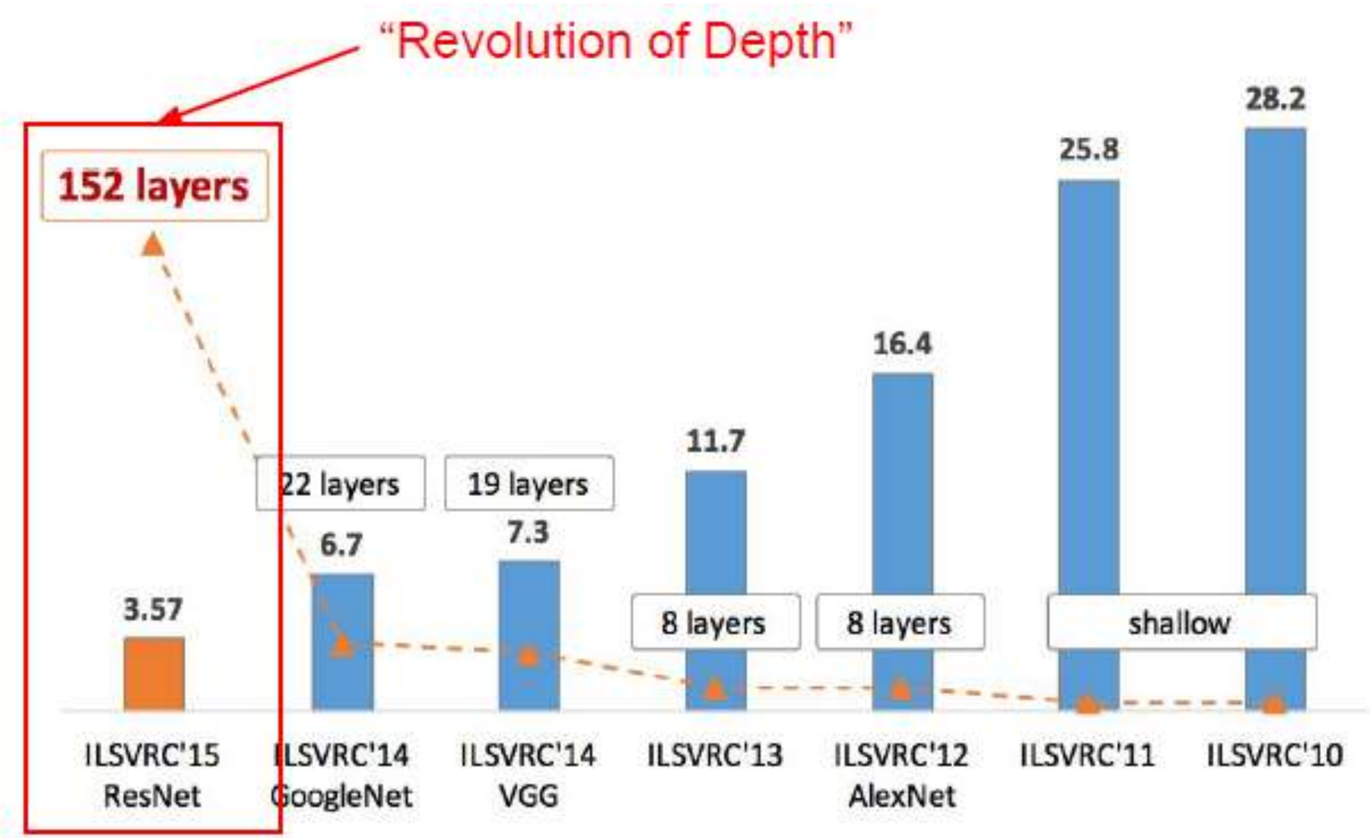
Inception module

Outline

- CNN architectures
  - Sequential structure: LeNet/AlexNet/VGGNet
  - Parallel branches: GoogLeNet
  - Residual structure: ResNet/DenseNet
  - Network Architecture Search

Acknowledgement: Zemel et al's CSC411 and Feifei Li et al's cs231n notes

ImageNet (ILSVRC)



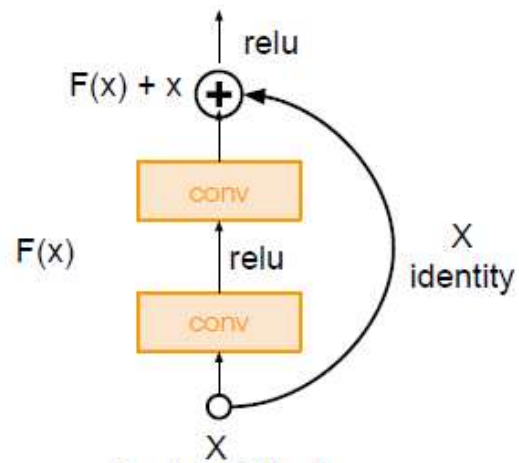
ResNet

Case Study: ResNet

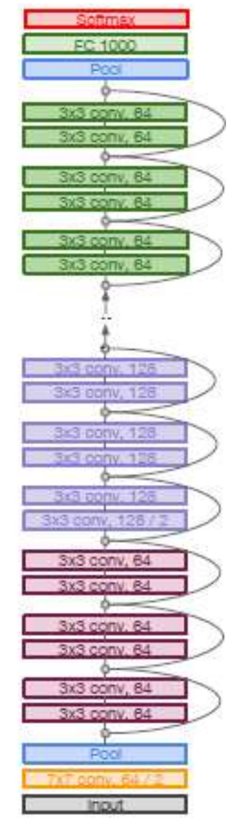
[He et al., 2015]

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!

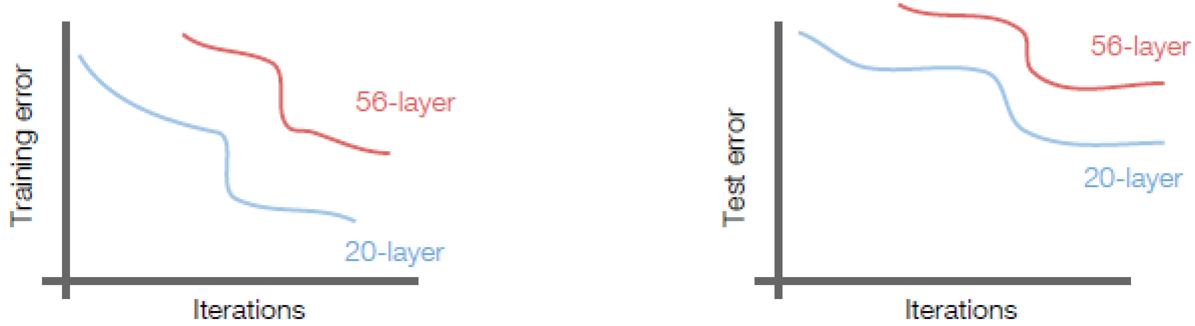


Residual block



ResNet

- What happens when stacking deeper plain conv layers?



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

ResNet

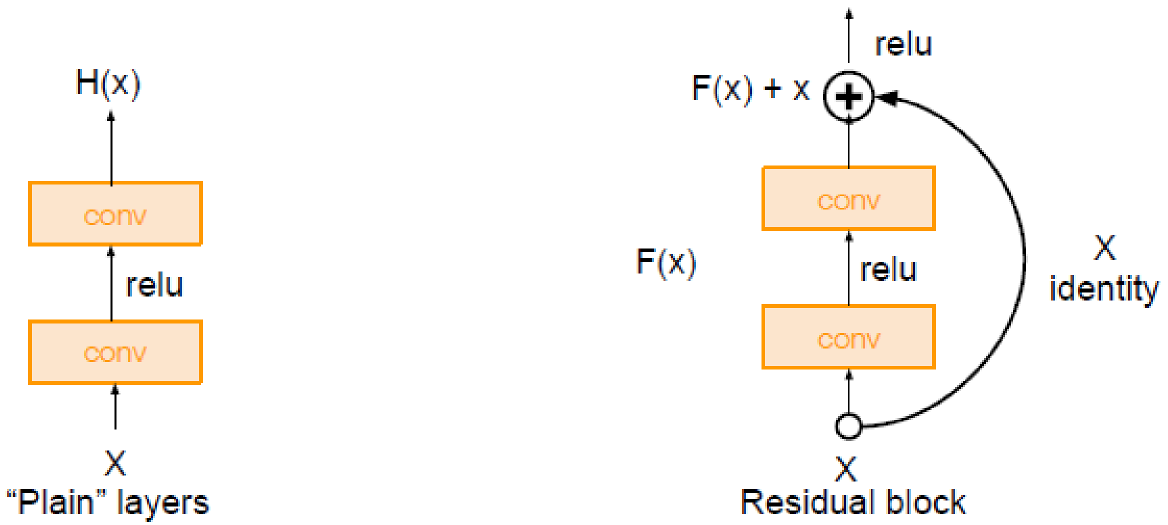
- Hypothesis:
  - The problem is an optimization problem, deeper models are harder to optimize

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

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

ResNet

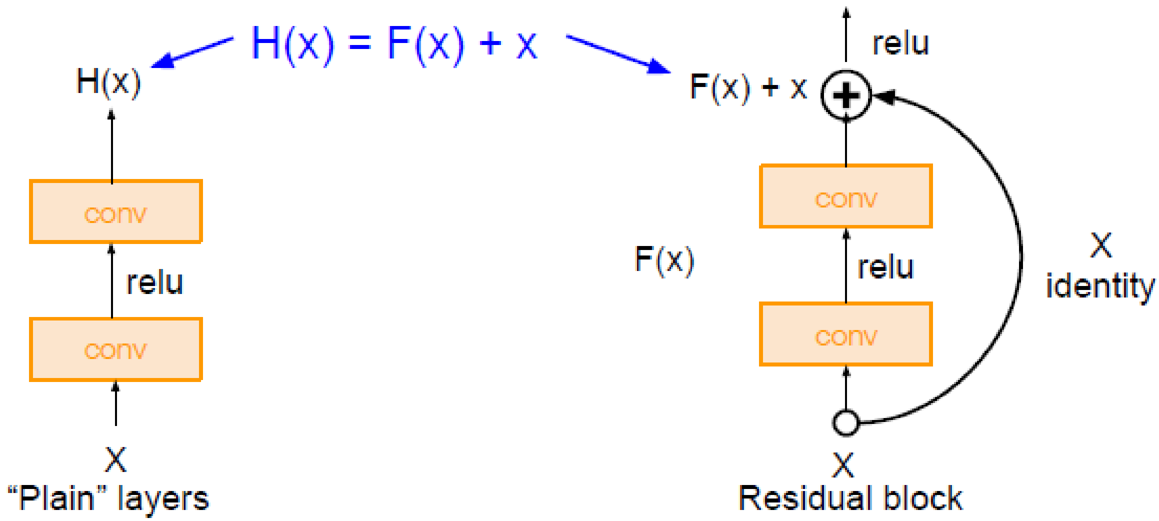
- Solution:
  - Use network layers to fit a residual mapping



He et al "Deep Residual Learning for Image Recognition", CVPR 2016

ResNet

- Solution:
  - Use network layers to fit a residual mapping



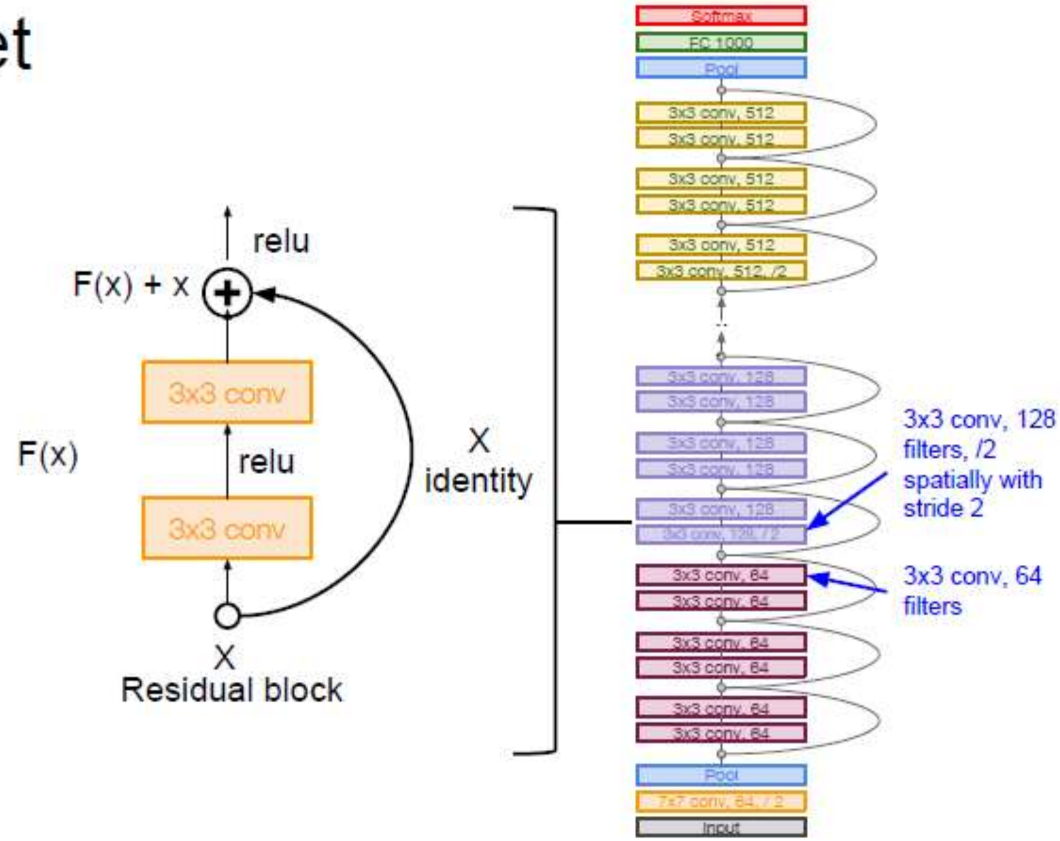
Use layers to fit residual $F(x) = H(x) - x$ instead of $H(x)$ directly

ResNet

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)

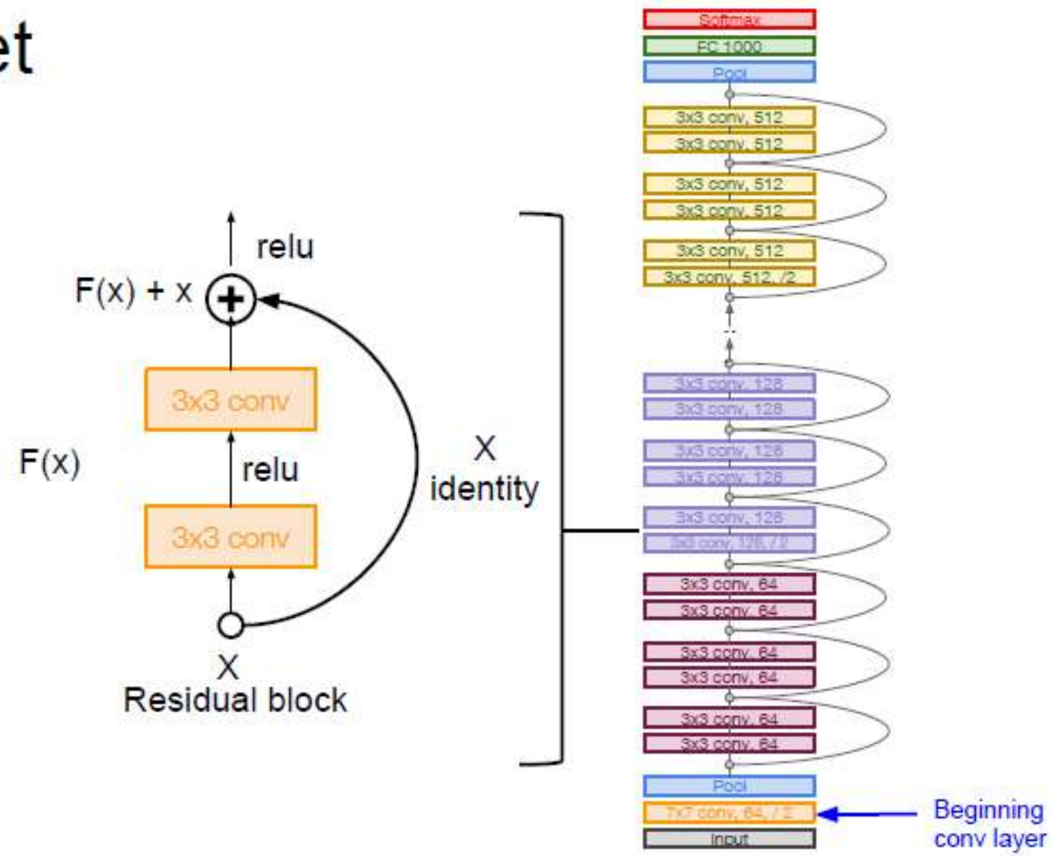


ResNet

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



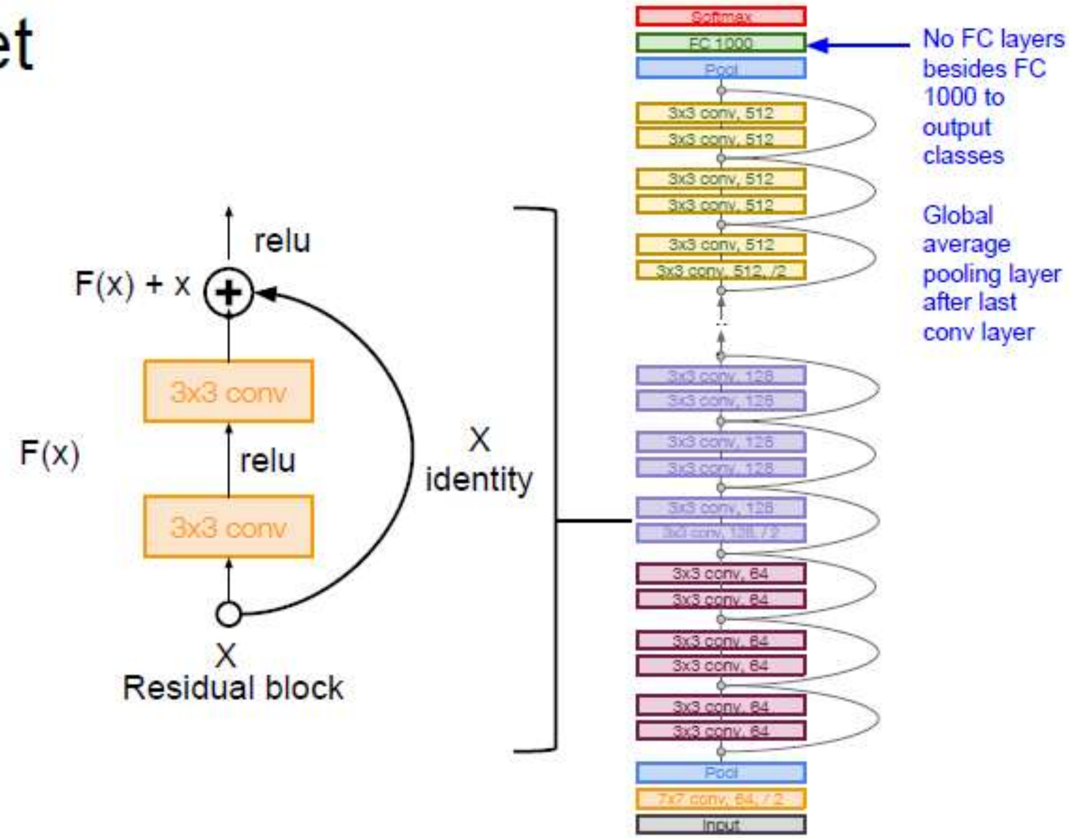
ResNet

Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:

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

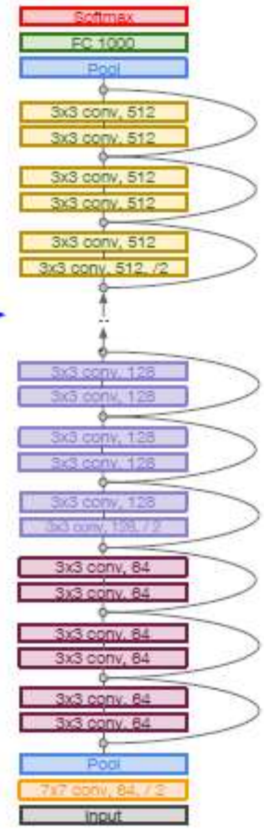


ResNet

Case Study: ResNet

[He et al., 2015]

Total depths of 34, 50, 101, or
152 layers for ImageNet



Case Study: ResNet

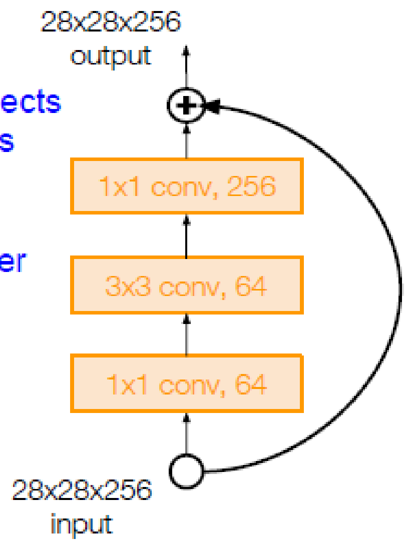
[He et al., 2015]

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

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



ResNet

- Training details

Training ResNet in practice:

- Batch Normalization after every CONV layer
- Xavier/2 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

ResNet

- Results

Experimental Results

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

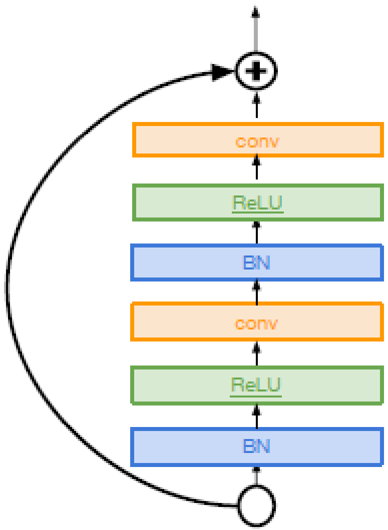
MSRA @ ILSVRC & COCO 2015 Competitions

- **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)

Other: Identity Mappings in ResNet

- Improved ResNet block design from creators of ResNet
- Creates a more direct path for propagating information throughout network (moves activation to residual mapping pathway)
- Gives better performance

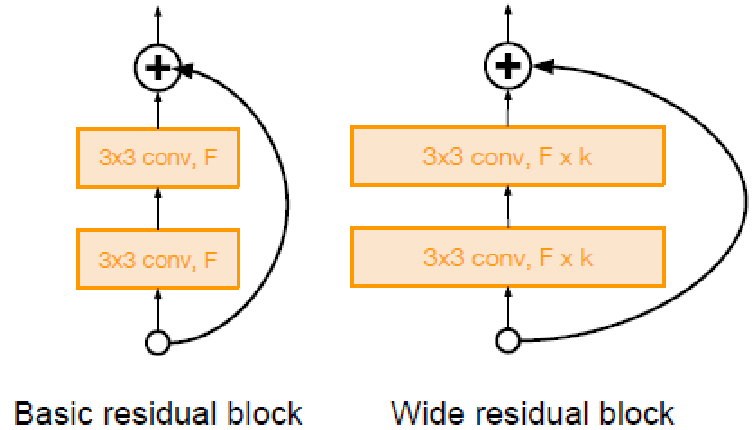


Other: Wide ResNets

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)

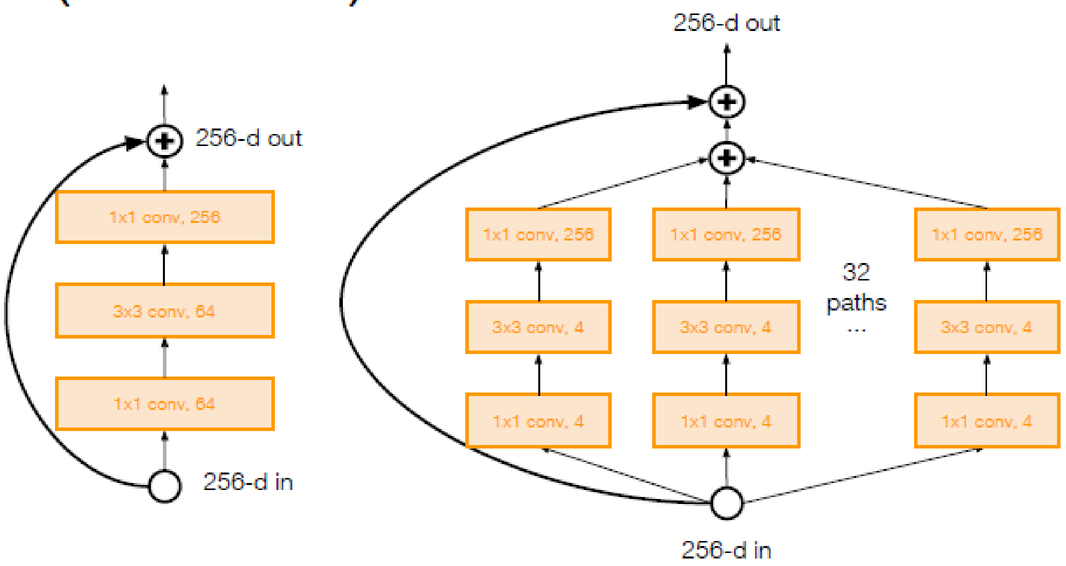


Other: ResNeXt

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

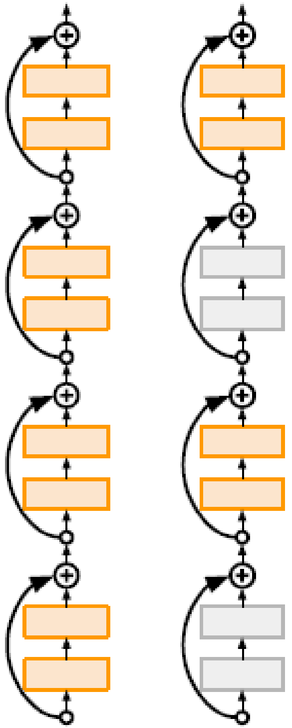


Other:ResNet with Stochastic Depth

Deep Networks with Stochastic Depth

[Huang et al. 2016]

- Motivation: reduce vanishing gradients and training time through short networks during training
- Randomly drop a subset of layers during each training pass
- Bypass with identity function
- Use full deep network at test time

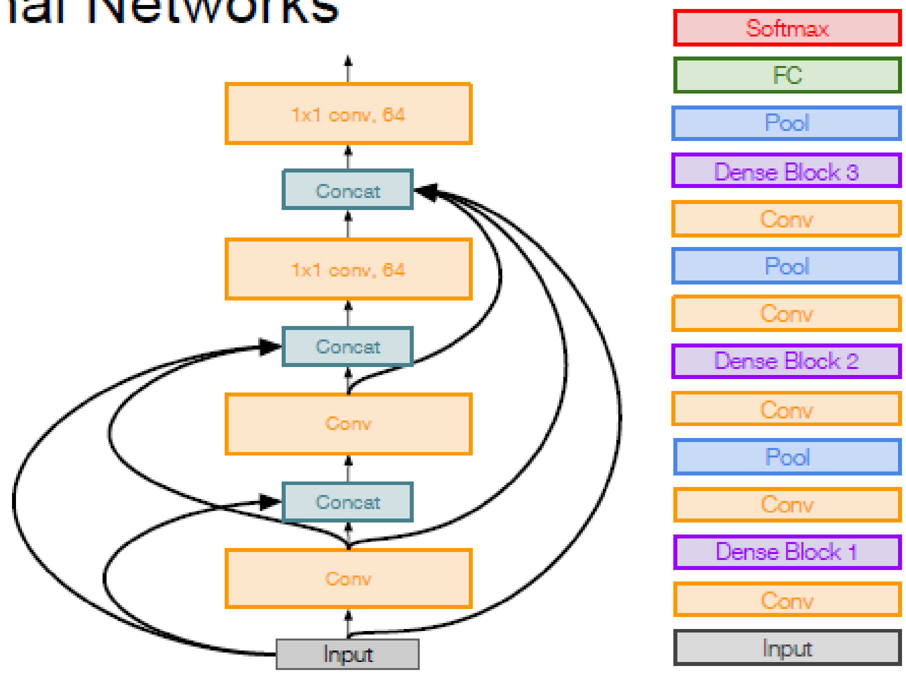


DenseNet

Densely Connected Convolutional Networks

[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

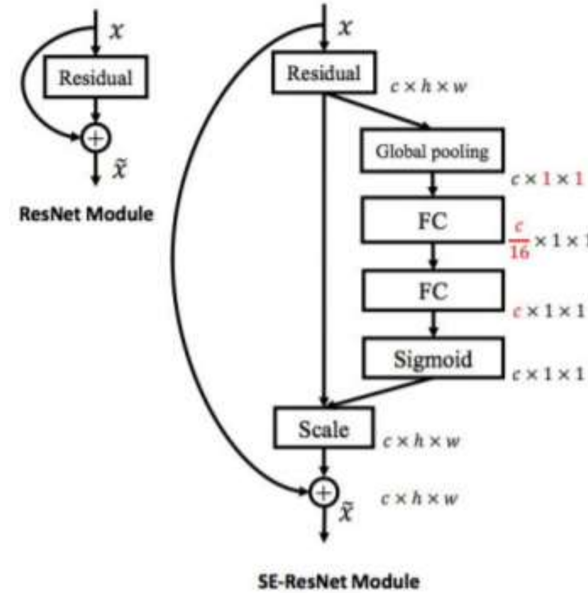


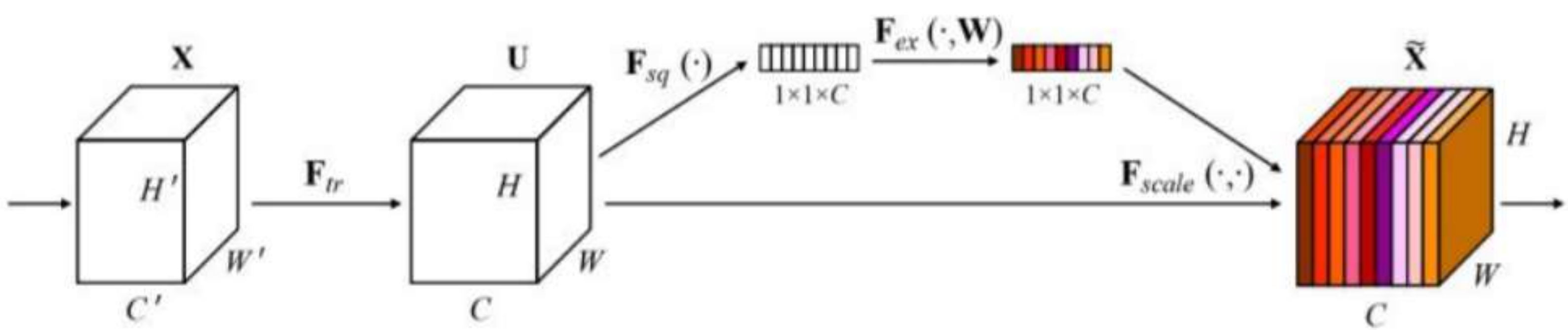
Dense Block

Squeeze-and-Excitation Networks (SENet)

[Hu et al. 2017]

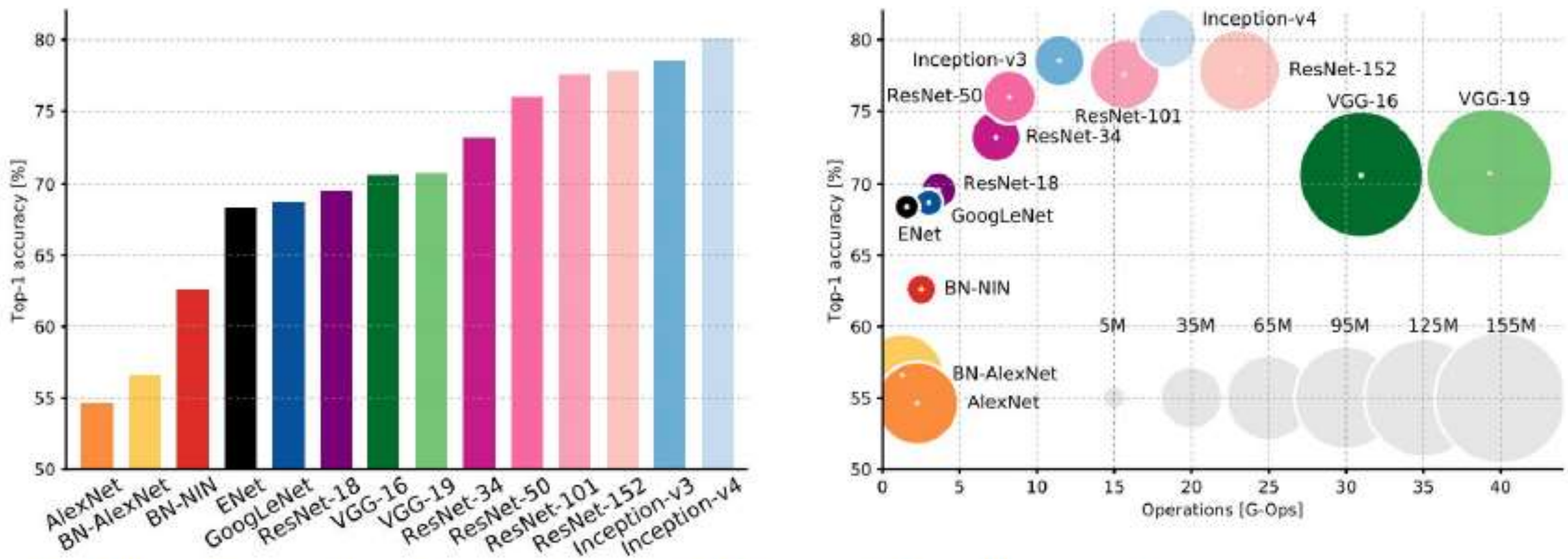
- Add a “feature recalibration” module that learns to adaptively reweight feature maps
- Global information (global avg. pooling layer) + 2 FC layers used to determine feature map weights
- ILSVRC’17 classification winner (using ResNeXt-152 as a base architecture)





Model complexity

Comparing complexity...



An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Outline

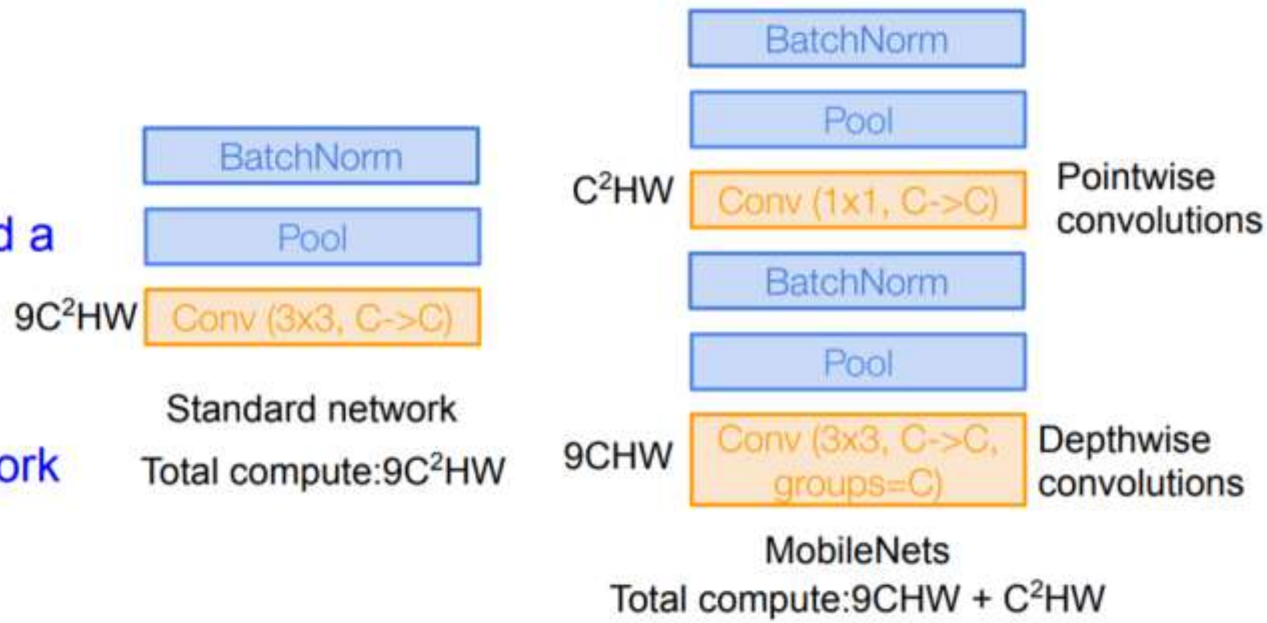
- CNN architectures
  - Sequential structure: LeNet/AlexNet/VGGNet
  - Parallel branches: GoogLeNet
  - Residual structure: ResNet/DenseNet
  - Network Architecture Search

Acknowledgement: Zemel et al's CSC411 and Feifei Li et al's cs231n notes

Efficient networks

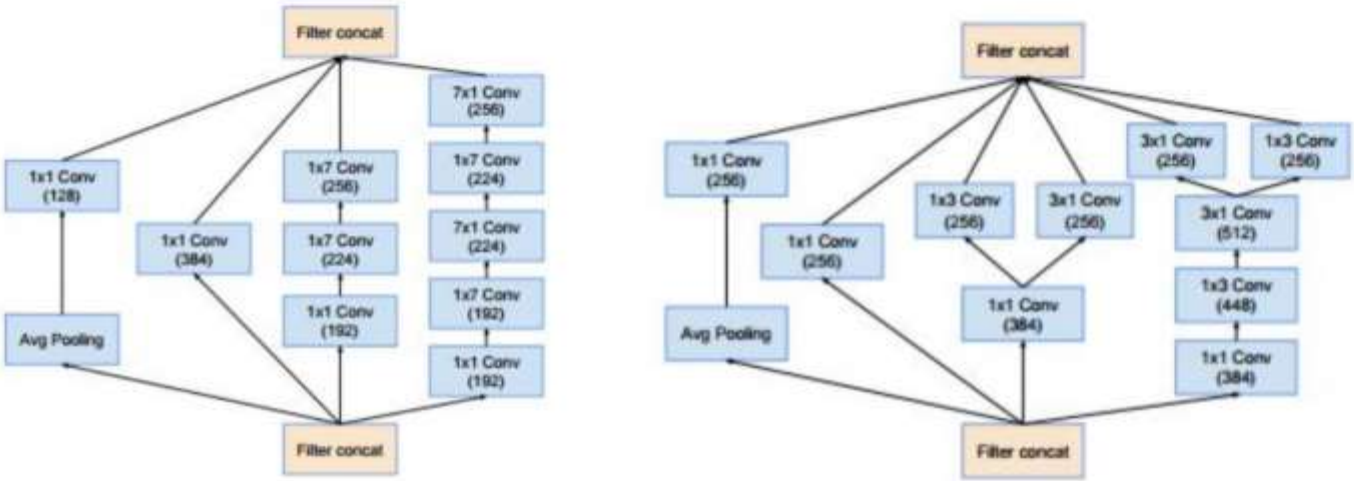
- MobileNets: Efficient Convolutional Neural Networks for Mobile Applications [Howard et al. 2017]

- 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.)
- ShuffleNet: Zhang et al, CVPR 2018



Network Architecture

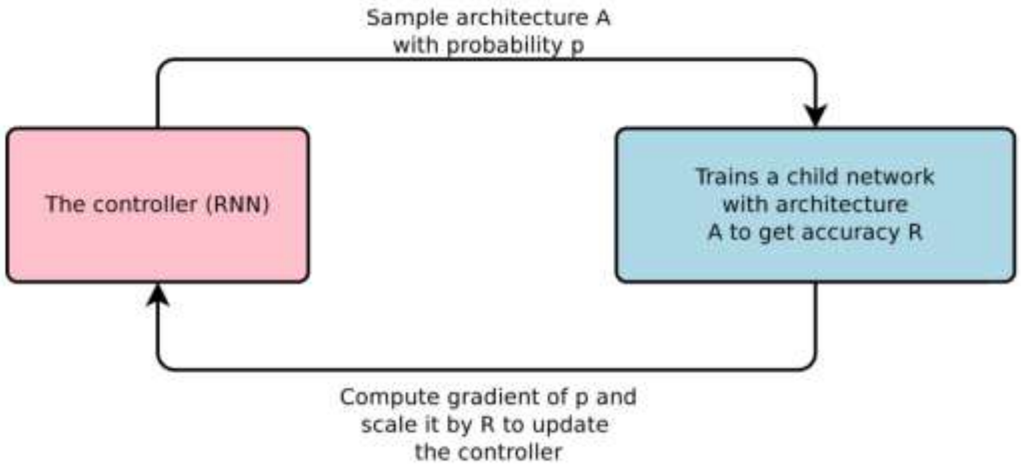
- Problems with network architecture
  - Designing NA is hard
  - Lots of human efforts go into tuning them
  - Not a lot of intuition into how to design them well
  - Can we learn good architectures automatically?



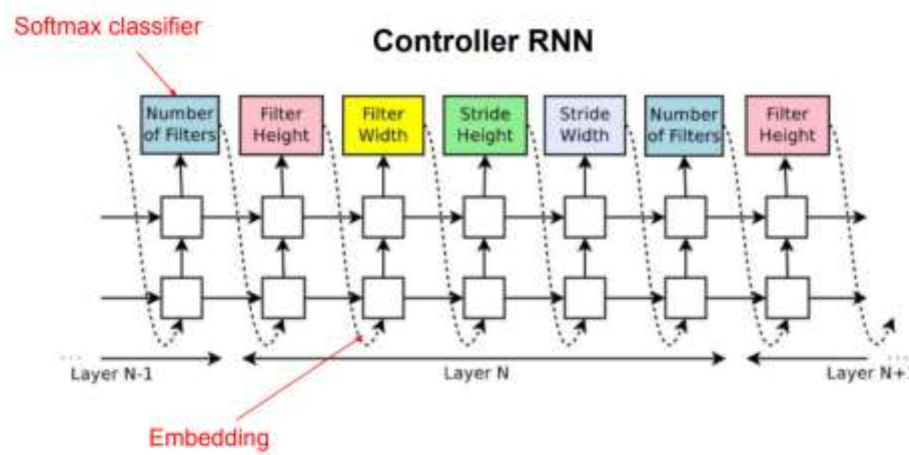
Two layers from the famous Inception V4 computer vision model.
Szegedy et al, 2017

Network Architecture

- Neural architecture search (Zoph and Le, ICLR 2016)

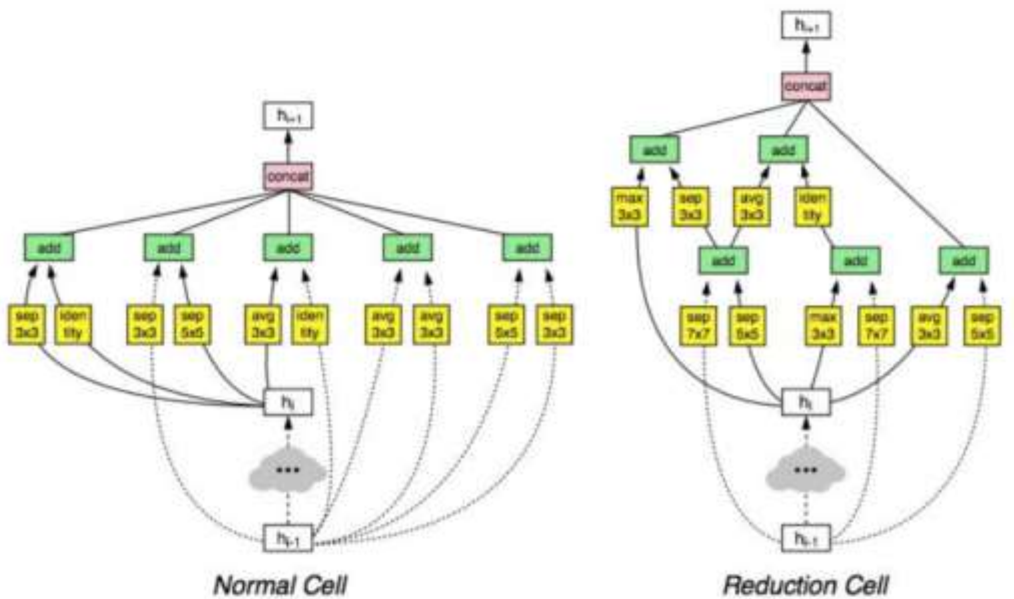


- "Controller" network that learns to design a good network architecture (output a string corresponding to network design)
- Iterate:
  1) Sample an architecture from search space
  2) Train the architecture to get a "reward" R corresponding to accuracy
  3) Compute gradient of sample probability, and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



Network Architecture

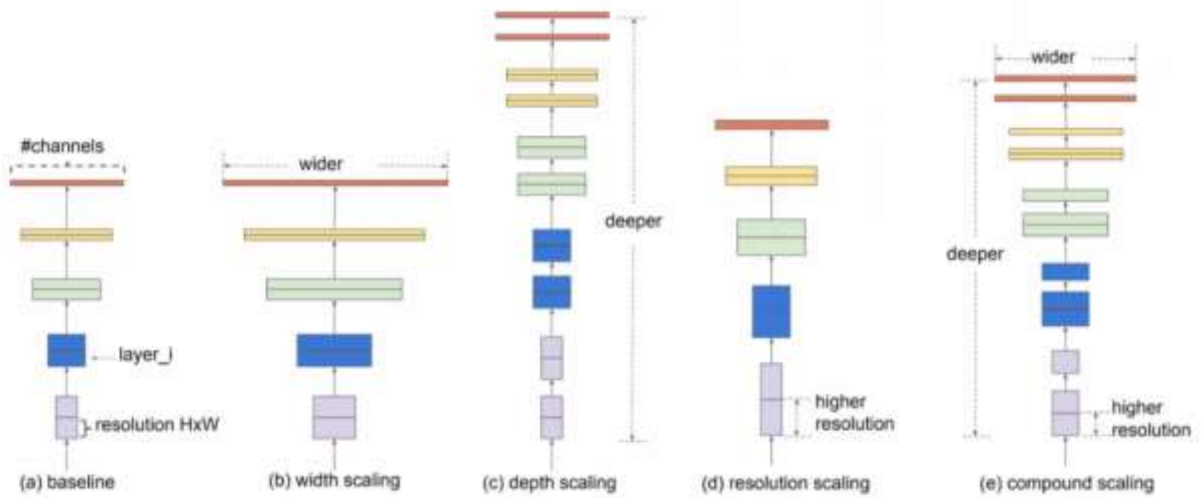
- Neural architecture search (Zoph et al. 2017)
  - Design a search space of building blocks ("cells") that can be flexibly stacked
  - NASNet: Use NAS to find best cell structure on smaller CIFAR-10 dataset, then transfer architecture to ImageNet
  - Many follow-up works in this space e.g. AmoebaNet (Real et al. 2019) and ENAS (Pham, Guan et al. 2018)



Network Architecture

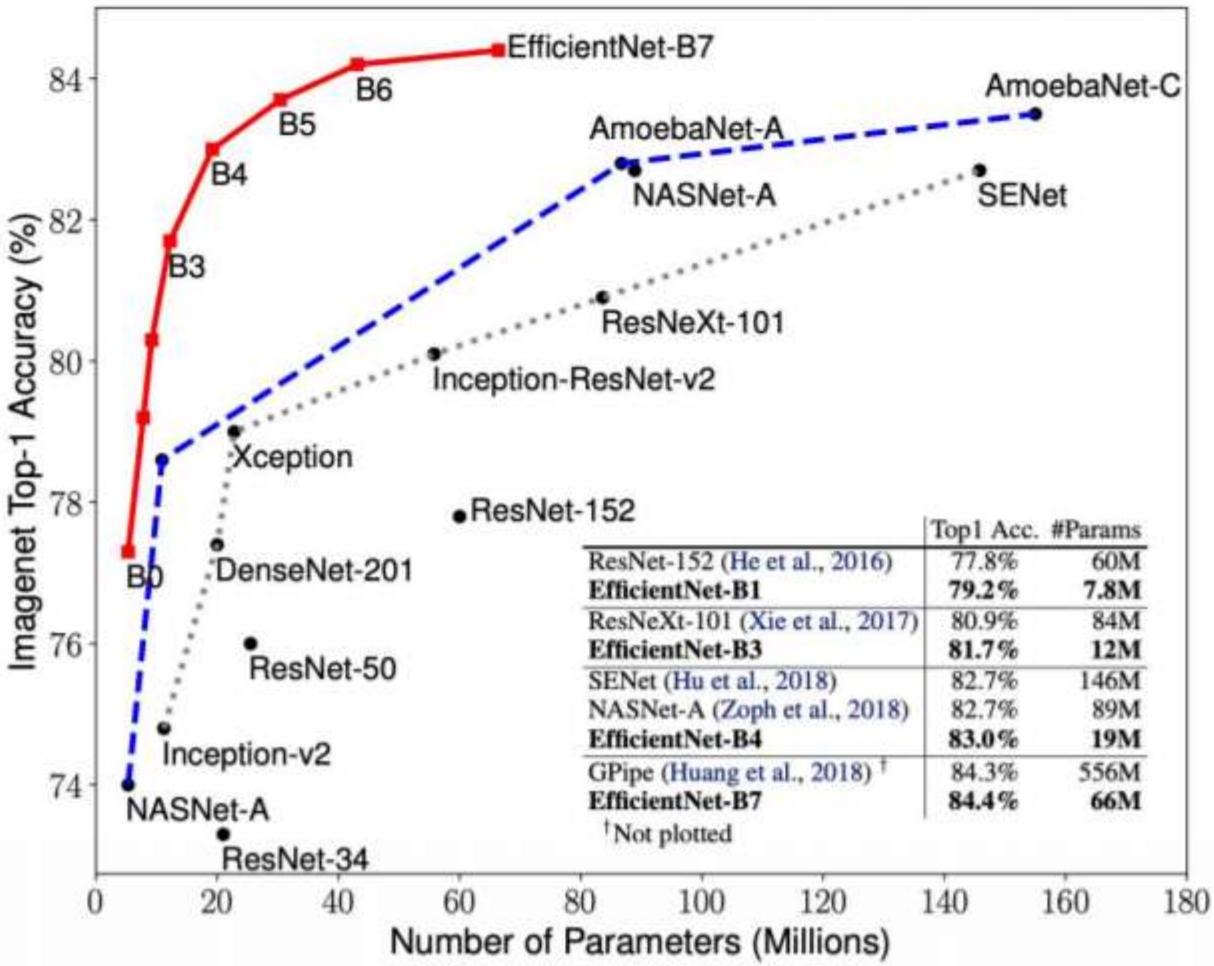
- EfficientNet: Smart Compound Scaling [Tan and Le. 2019]
  - Increase network capacity by scaling width, depth, and resolution, while balancing accuracy and efficiency.
  - Search for optimal set of compound scaling factors given a compute budget (target memory & flops).
  - Scale up using smart heuristic rules

$$\text{depth: } d = \alpha^{\phi}$$
$$\text{width: } w = \beta^{\phi}$$
$$\text{resolution: } r = \gamma^{\phi}$$
$$\text{s.t. } \alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$$
$$\alpha \geq 1, \beta \geq 1, \gamma \geq 1$$



Network Architecture

- EfficientNet: Smart Compound Scaling [Tan and Le. 2019]



	Top1 Acc.	#Params
ResNet-152 (He et al., 2016)	77.8%	60M
EfficientNet-B1	**79.2%**	**7.8M**
ResNeXt-101 (Xie et al., 2017)	80.9%	84M
EfficientNet-B3	**81.7%**	**12M**
SENet (Hu et al., 2018)	82.7%	146M
NASNet-A (Zoph et al., 2018)	82.7%	89M
EfficientNet-B4	**83.0%**	**19M**
GPipe (Huang et al., 2018) †	84.3%	556M
EfficientNet-B7	**84.4%**	**66M**

†Not plotted

Network structure summary

- 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

- Bag of tricks for improving generalization
  - Pros: you have a toolbox to use
  - Cons: many trial and error, tedious process

- Seeking fully automatic approaches to model selection
  - Bayesian optimization
  - Reinforcement learning

- Next time
  - CNN in Vision, RNN

- Reference
  - CS231n course notes