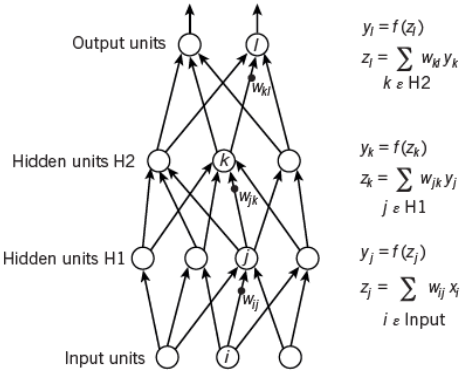
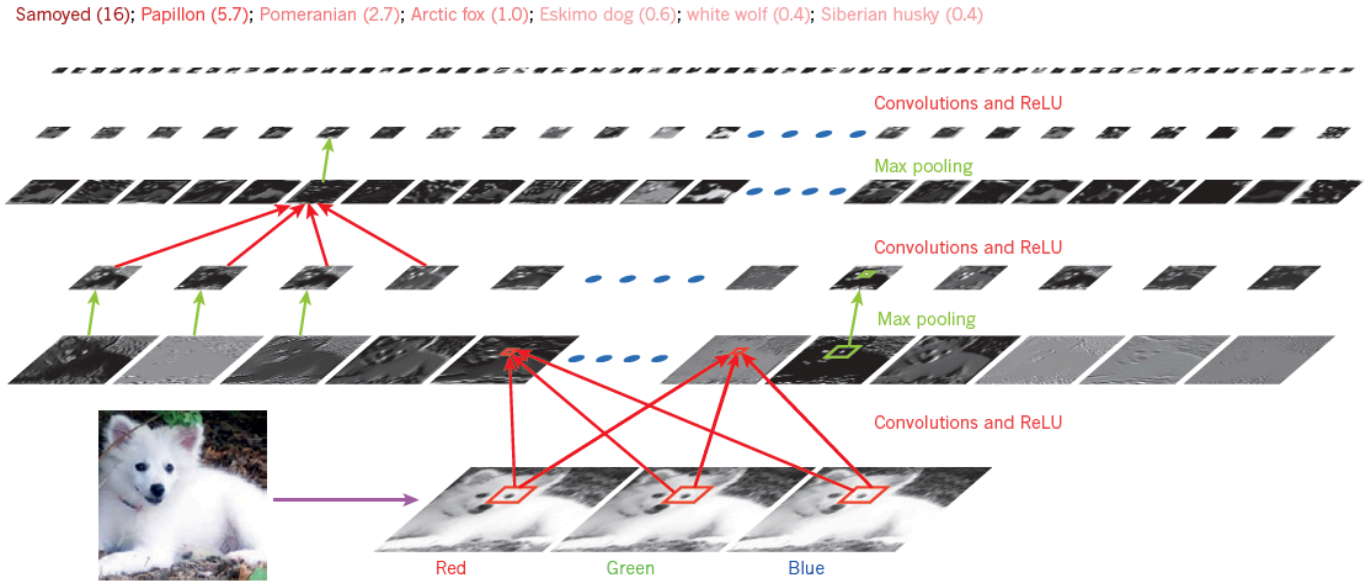
Topics in Deep Learning: Methods and Biomedical Applications

Lecture 2
Deep Supervised Learning: Feed-forward Neural Networks and Convolutional Neural Networks

Dr. Martin Renqiang Min
Prof. Mark Gerstein

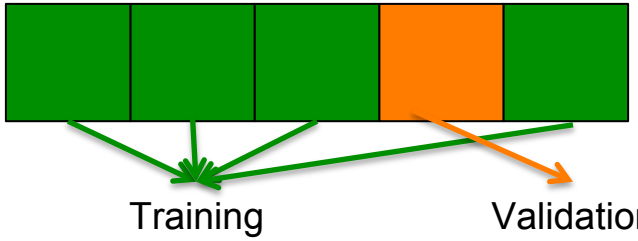
Supervised Deep Learning





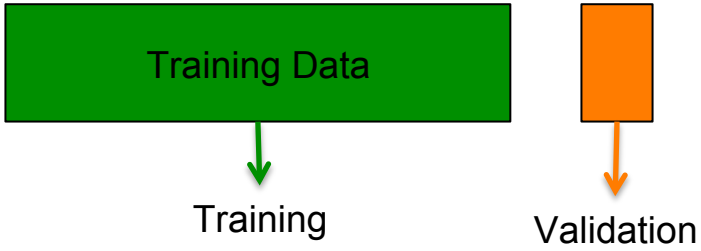
LeCun, Bengio, and Hinton, Deep Learning. Nature 2015

Supervised Deep Learning



Test Data

Supervised Machine Learning:
Feature Representation + Classification/ Regression Loss + Optimization (on training data)
→ Prediction (on test data)
(hyper-parameter tuning with n-fold CV, n=5)

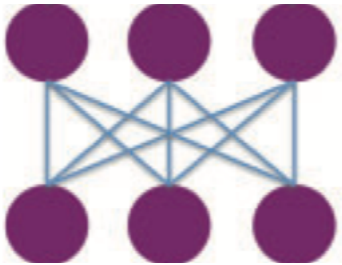


Test Data

Supervised Deep Learning:
Input features and adaptively learned features by hidden layers + Mean Squared Error/Hinge Loss/Cross-Entropy Loss + SGD with Momentum (on large-scale training data)
→ Good Prediction Performance (on test data)
(hyper-parameter tuning on a validation set)

Fully Connected Layer

y



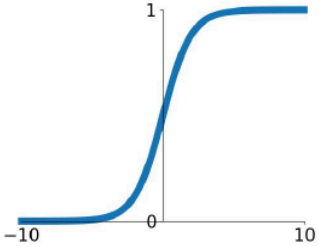
x

$$y = W x$$

Activation Functions

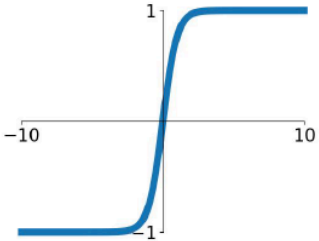
Sigmoid

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



tanh

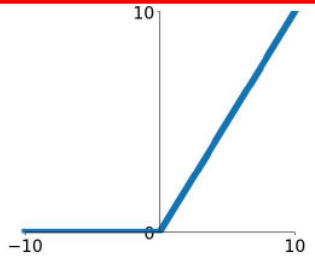
$$\tanh(x)$$



ReLU

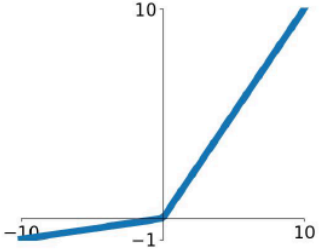
$$\max(0, x)$$

Good default choice



Leaky ReLU

$$\max(0.1x, x)$$

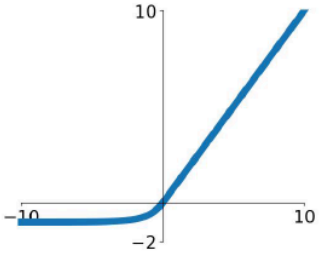


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

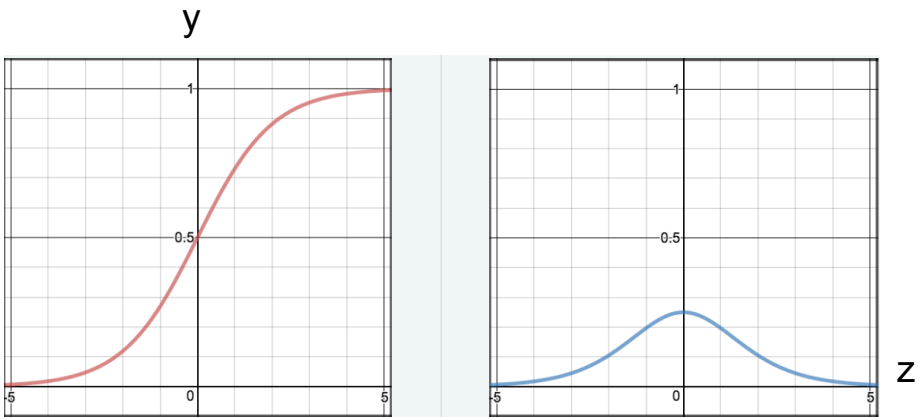
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



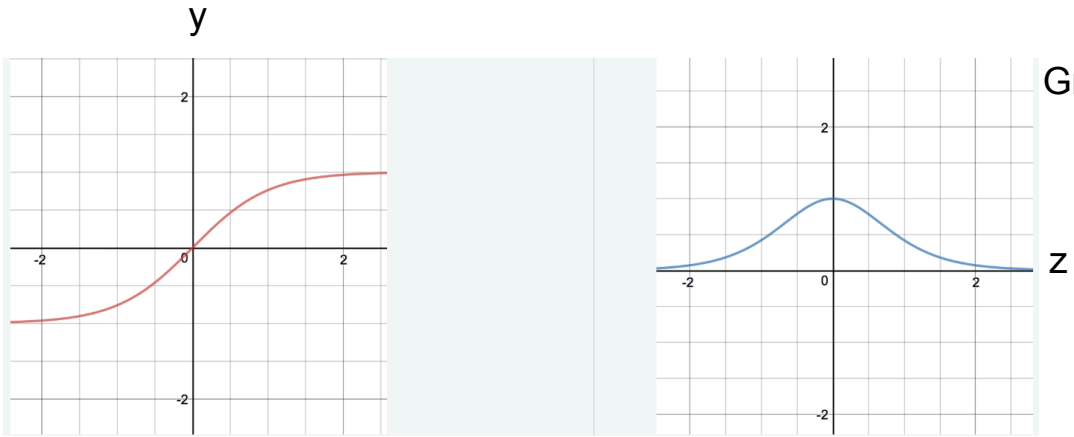
DNN with sigmoid and tanh activation functions has serious vanishing gradient and saturation issue

$$\frac{1}{1+e^{-z}}$$



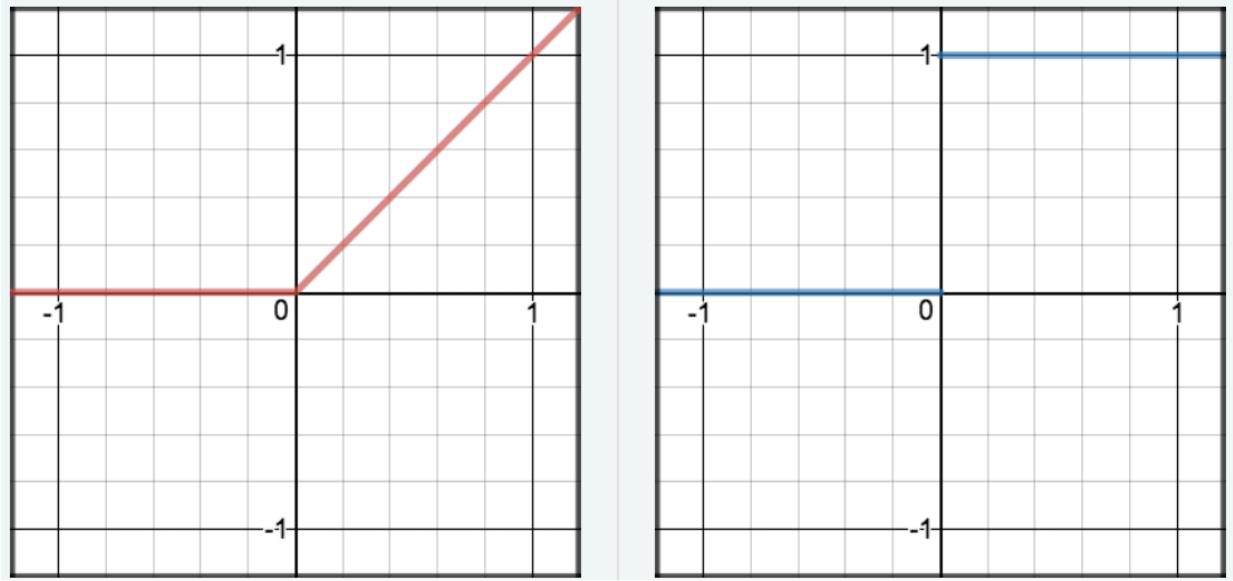
Grad = y(1-y)

$$\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$$



Grad = $1-y^2$

ReLU Activation Function

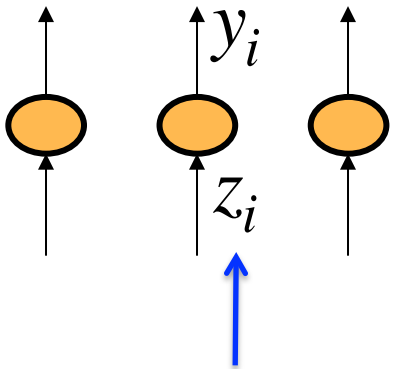


Avoid vanishing gradient and less computationally expensive than sigmoid and tanh

But it might cause dead neuron and the activity is not bounded above

Softmax Activation Function

The output units in a softmax group use a non-local non-linearity:



softmax group

this is called the "logit"

$$y_i = \frac{e^{z_i}}{\sum_{j \in group} e^{z_j}}$$

$$\frac{\partial y_i}{\partial z_i} = y_i \left(1 - y_i\right)$$

Often used on top of a fully connected layer, which transforms an activity vector **z** into probabilities of classifying **x** into K classes

Loss Function: Cross-Entropy Loss

The right cost function is the negative log probability of the target class.

C has a very big gradient when the target value is 1 and the output is almost zero.

A value of 0.001 is much better than 0.0000001

The steepness of dC/dy exactly balances the flatness of dy/dz

$$C = -\sum_j t_j \log y_j$$

Target Class

$$\frac{\partial C}{\partial z_i} = \sum_j \frac{\partial C}{\partial y_j} \frac{\partial y_j}{\partial z_i} = y_i - t_i$$

Loss Function: Mean Squared Error

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

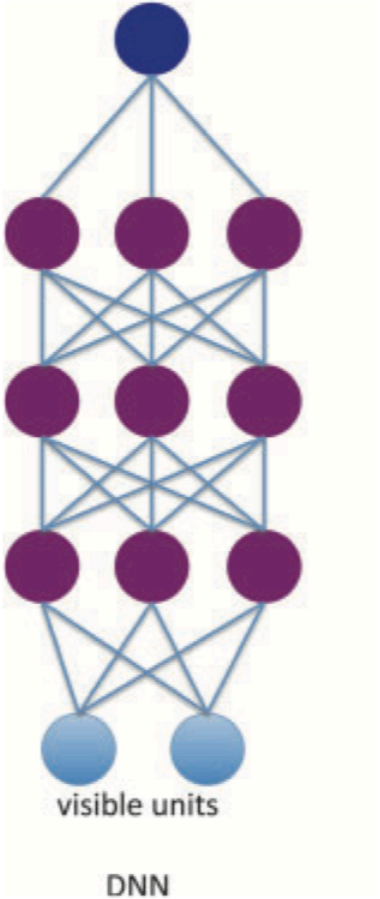
MSE is a very bad cost function for softmax output units.
Why?

Loss Function: Hinge Loss

$$\sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

The score for the wrong class must be 1 margin smaller than the score for the ground-truth class;
Otherwise, there is a loss incurred

Deep Feedforward Neural Network with Sigmoid Hidden Units



y

h3

w3

h2

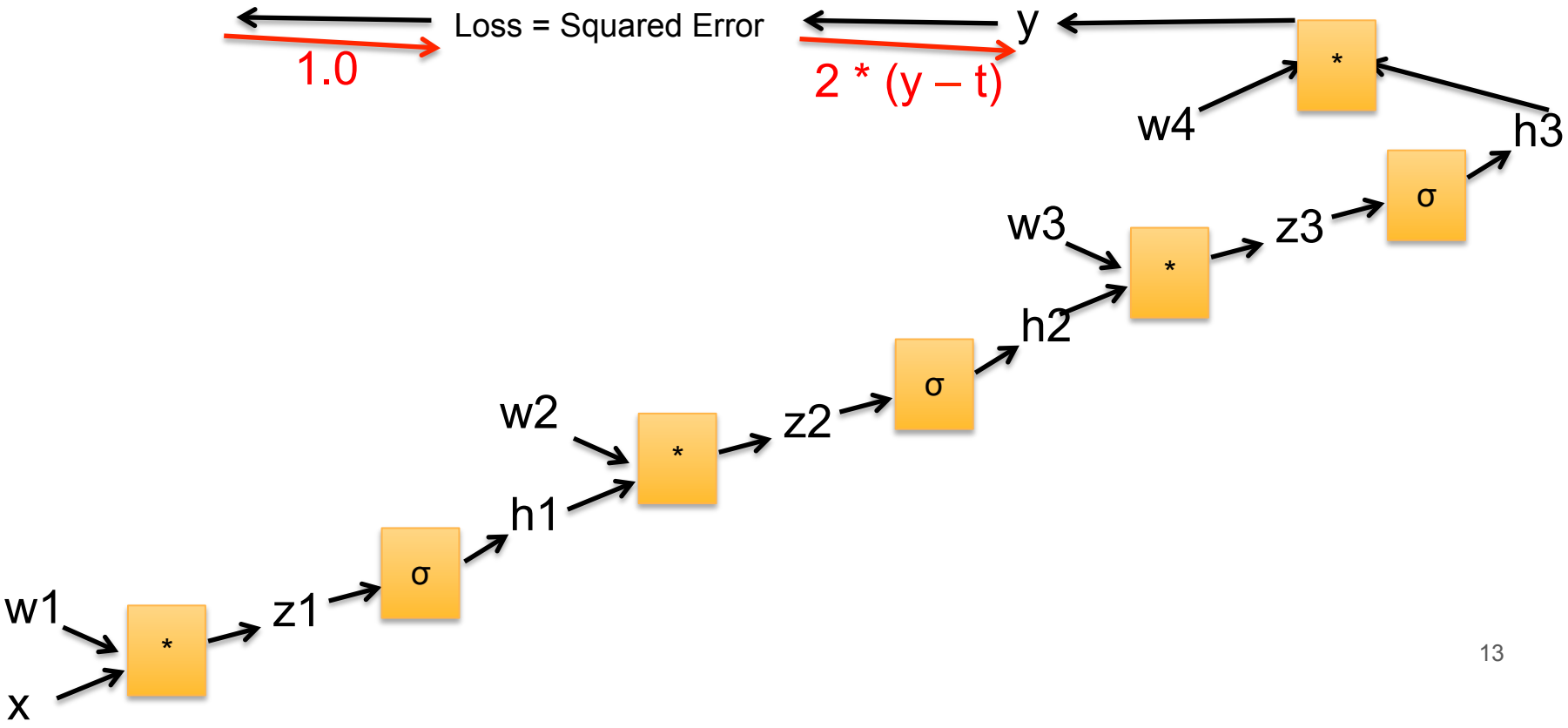
w2

h1

w1

x

Backpropagation with a Computational Graph



Train a Deep Neural Network with SGD

Split our training dataset into N mini-batches with batch size b

For Iteration = 1, …, Num_Max_Iterations

randomly choose a mini-batch D_i

$$v_{i+1} := 0.9 \cdot v_i - 0.0005 \cdot \epsilon \cdot w_i - \epsilon \cdot \left\langle \frac{\partial L}{\partial w}\Big|_{w_i} \right\rangle_{D_i}$$

$$w_{i+1} := w_i + v_{i+1}$$

where i is the iteration index, v is the momentum variable, ϵ is the learning rate, and $\left\langle \frac{\partial L}{\partial w}\Big|_{w_i} \right\rangle_{D_i}$ is the average over the ith batch D_i of the derivative of the objective with respect to w, evaluated at w_i.

(you can also have two loops: outer loop over iterations, inner loop over mini-batches)

DNN works much worse than a shallow CNN even on MNIST!

~1.0% vs. ~0.60%

Why?

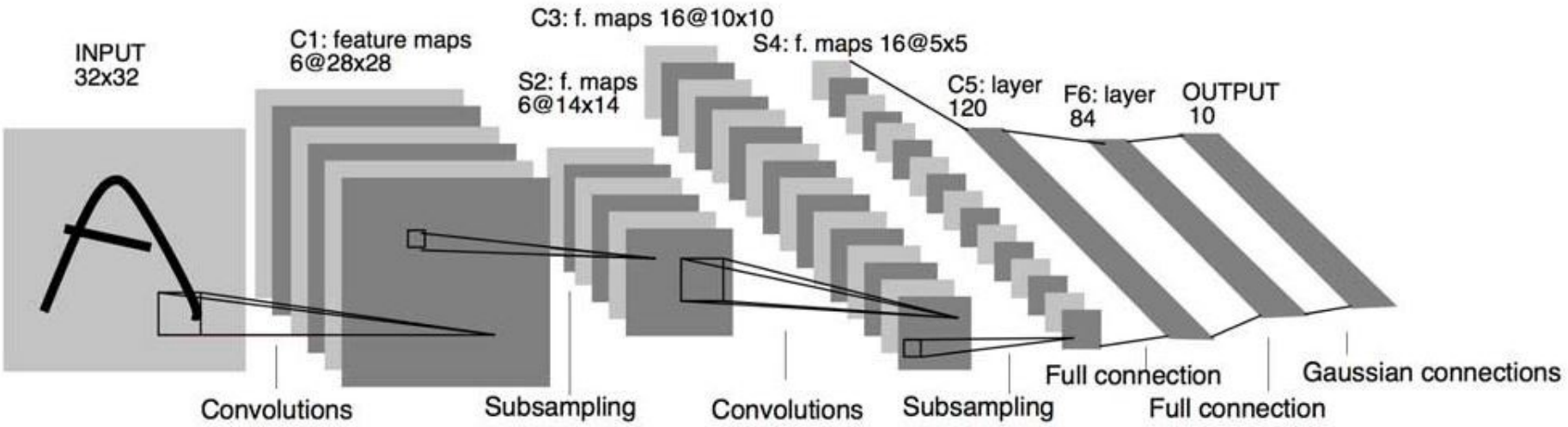
Hubel and Wiesel Experiment

https://www.youtube.com/watch?v=OGxVfKJqX5E

Message from Last Lecture

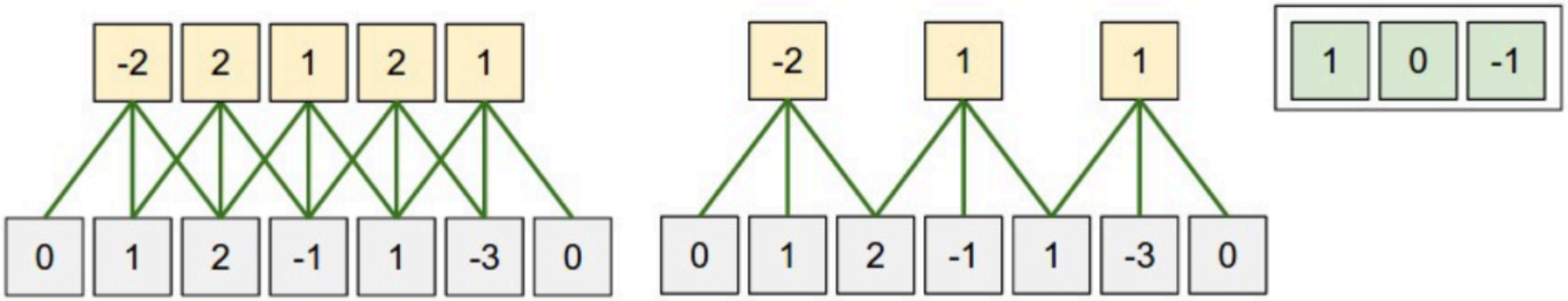
Deep learners should **combine** their **knowledge** with large-scale **data** to grow programs, **encode essential knowledge into network structures**, and let backpropagation and stochastic gradient descent do the heavy lifting.

Convolutional Neural Network: LeNet (1998)



LeCun et al., 1998

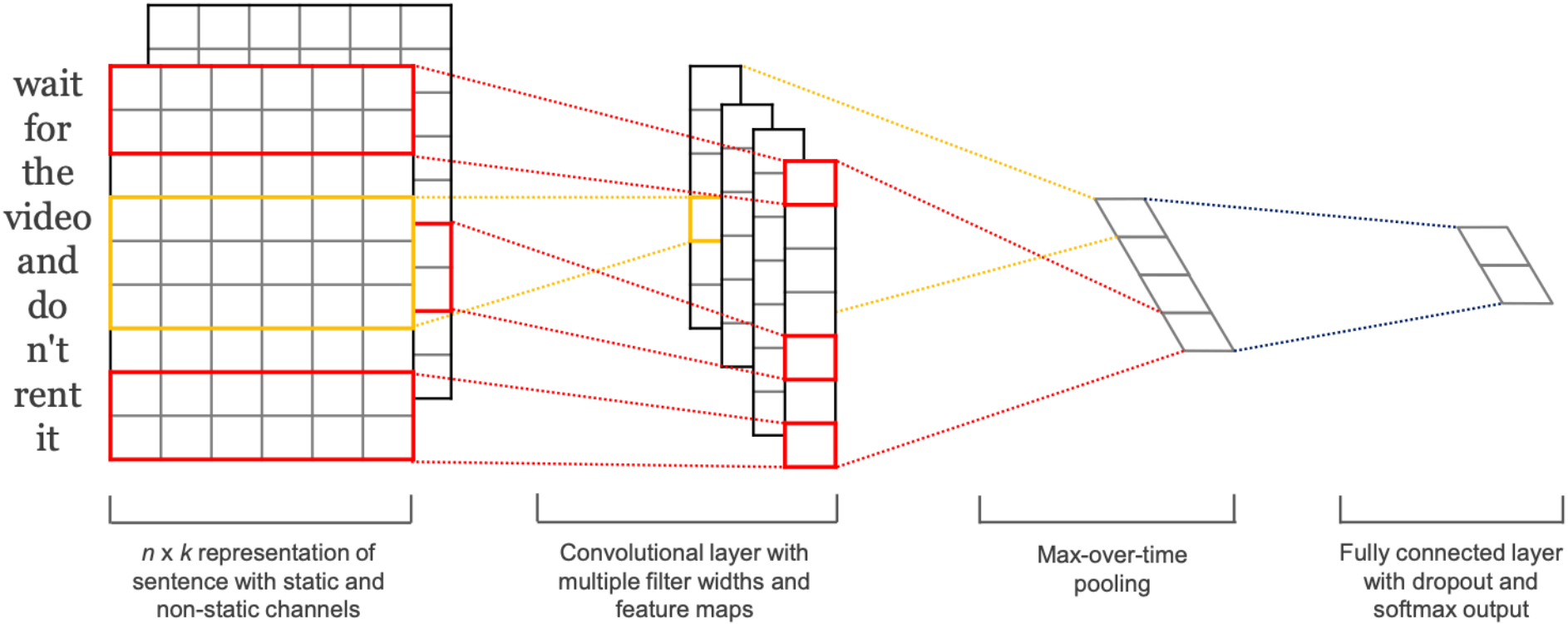
1D Convolution with W =5, F = 3, Stride = 1, Padding = 1



Output Size = $(W - F + 2P)/S + 1$

http://cs231n.github.io/convolutional-networks/

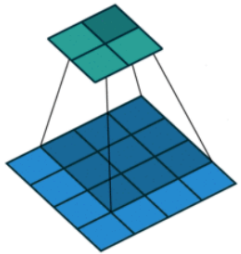
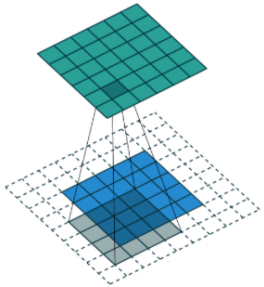
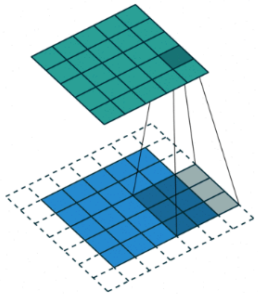
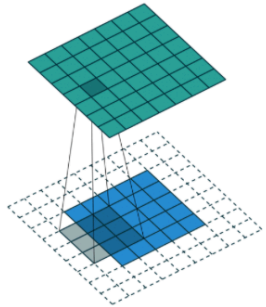
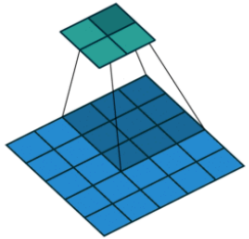
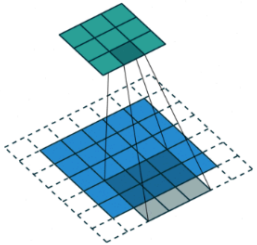
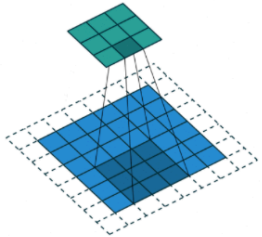
1D Convolution over Sentences



Yoon Kim, Convolutional Neural Networks for Sentence Classification. EMNLP 2014

2D Convolutions

N.B.: Blue maps are inputs, and cyan maps are outputs.

			
No padding, no strides	Arbitrary padding, no strides	Half padding, no strides	Full padding, no strides
			
No padding, strides	Padding, strides	Padding, strides (odd)	

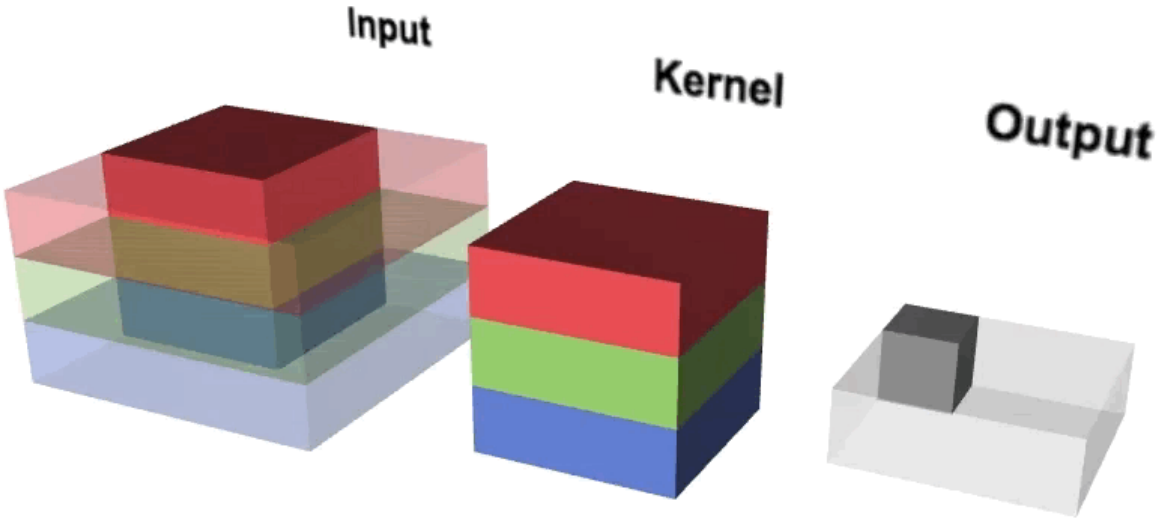
https://github.com/vdumoulin/conv_arithmetic

2D Convolution Animations

See the animation at

https://github.com/vdumoulin/conv_arithmetic

2D 3x3 Convolution Applied to RGB Input of Size 5x5

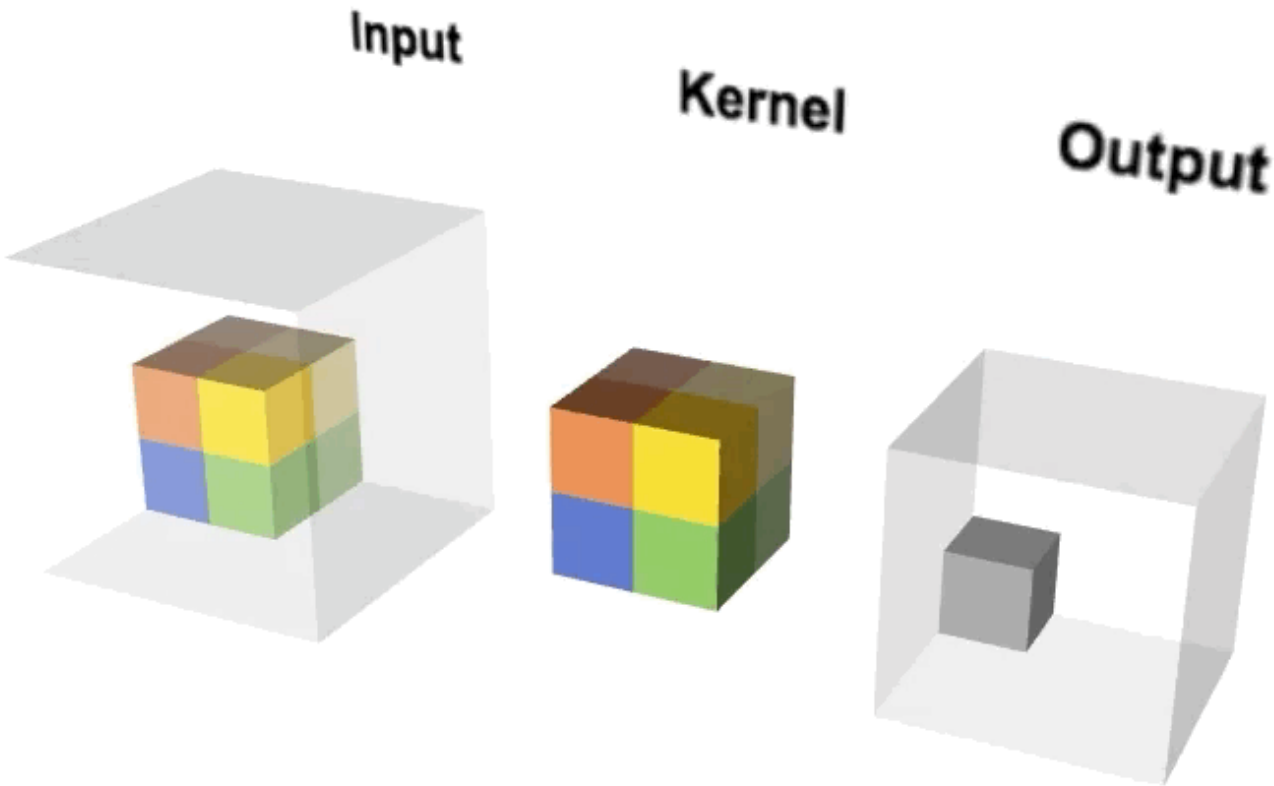


Picture credit: https://thomelane.github.io/convolutions/2DConvRGB.html

2D Convolutions in Numbers

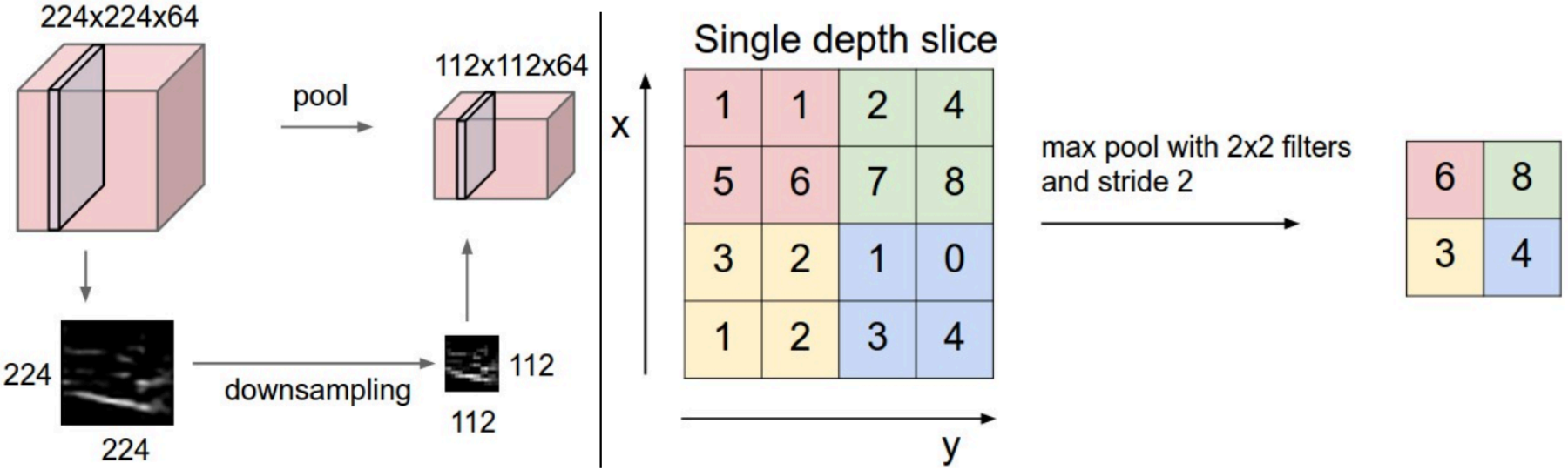
http://cs231n.github.io/convolutional-networks/

3D Convolution



Picture credit: https://thomelane.github.io/convolutions/3DConv.html

Max Pooling



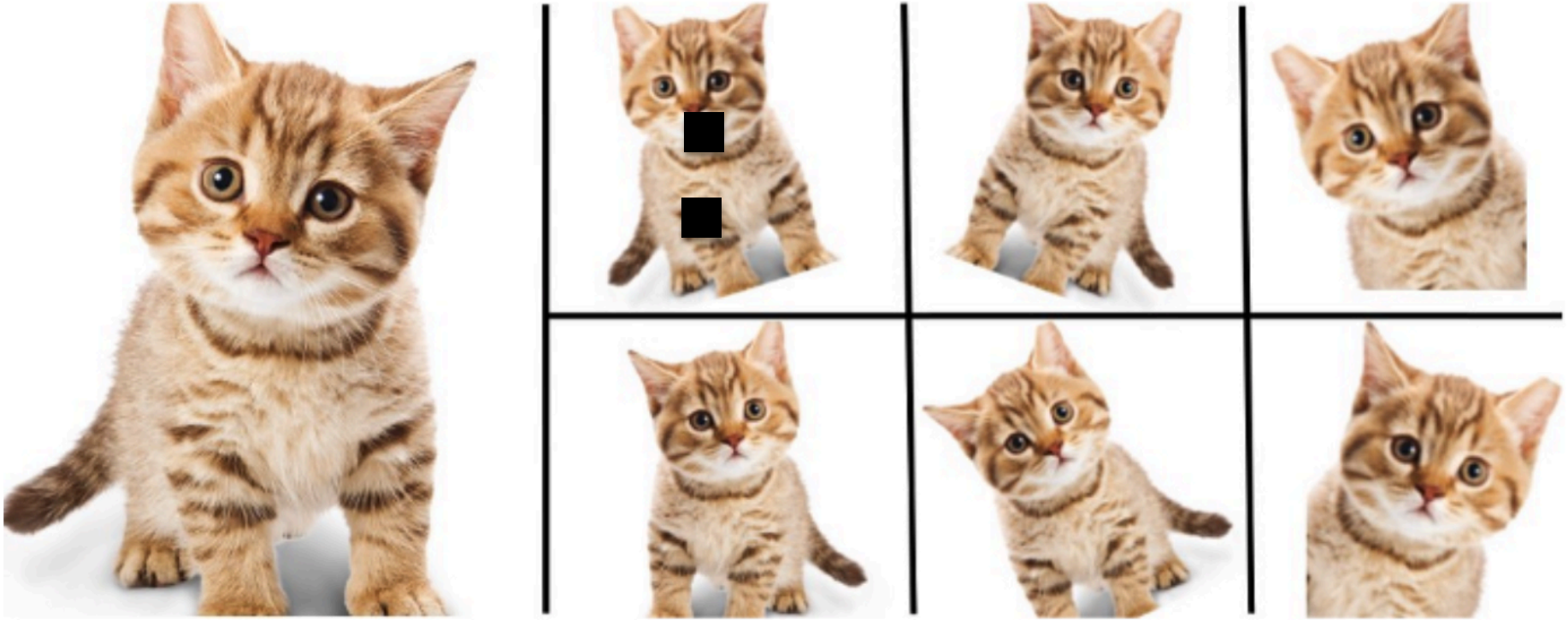
Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. **Left:** In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. **Right:** The most common downsampling operation is max, giving rise to **max pooling**, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).

http://cs231n.github.io/convolutional-networks/

Average Pooling is also widely used, especially in NLP

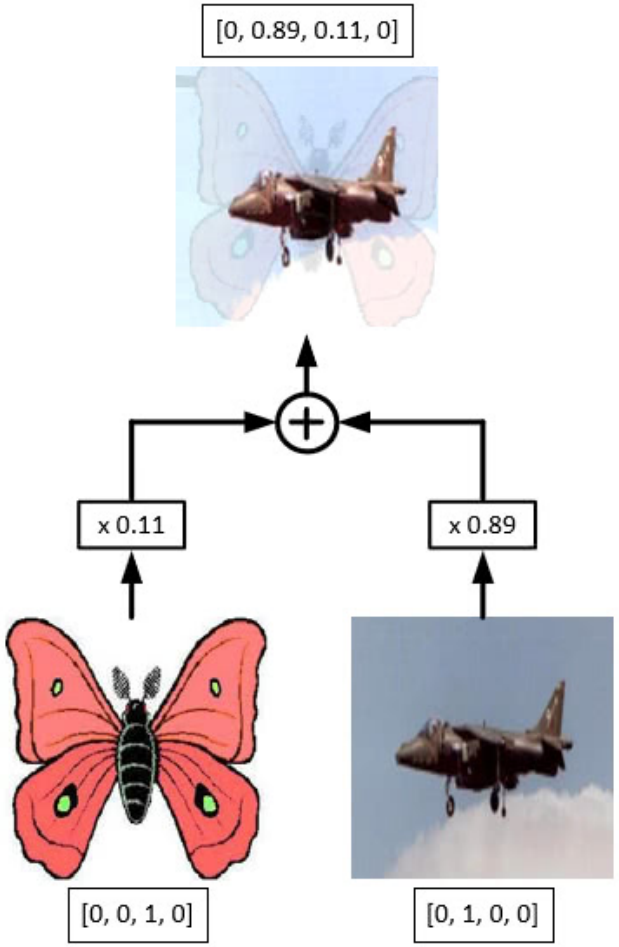
Data Augmentation

Random erasing, horizontal flipping, rotation, scaling (with cropping), cropping, contrast, color



Picture credit: https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

Mixup



$$\tilde{x} = \lambda x_i + (1 - \lambda) x_j, \quad \text{where } x_i, x_j \text{ are raw input vectors}$$
$$\tilde{y} = \lambda y_i + (1 - \lambda) y_j, \quad \text{where } y_i, y_j \text{ are one-hot label encodings}$$

Zhang *et al*., Mixup: beyond empirical risk minimization. ICLR 2018.

Picture credit: https://www.dlology.com/blog/how-to-do-mixup-training-from-image-files-in-keras/

Case Study: AlexNet

[PDF] ImageNet Classification with Deep Convolutional Neural ...

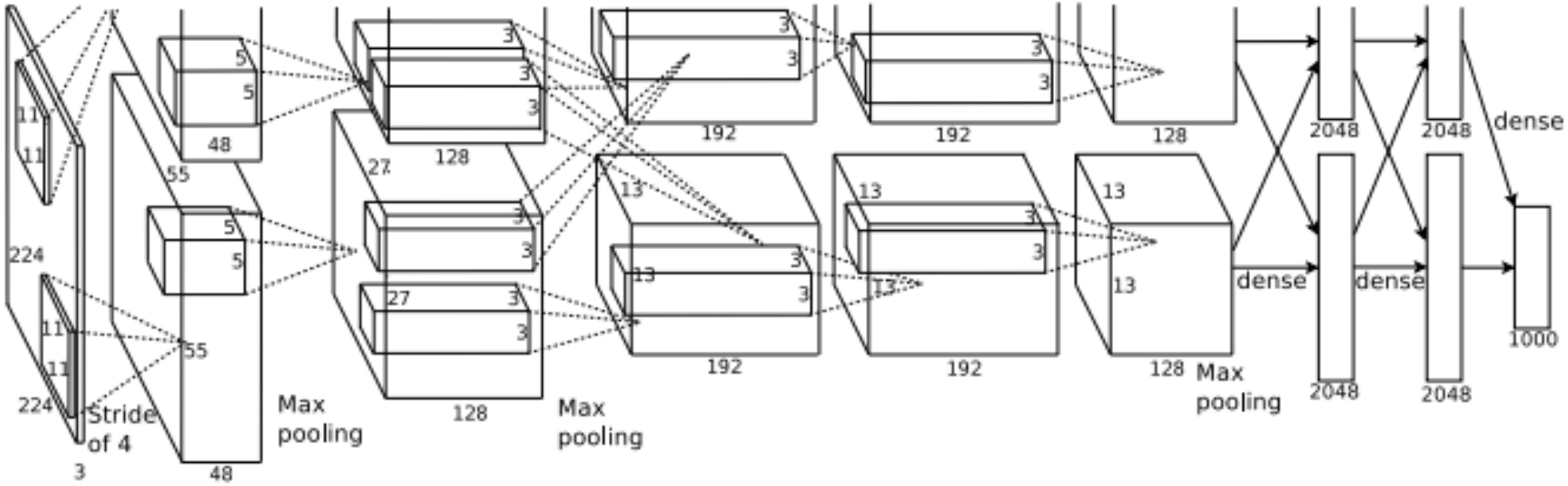
https://papers.nips.cc › paper › 4824-imagenet-classification-with-deep-co...

by A Krizhevsky - 2012 - Cited by 54415 - Related articles

We trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 dif-.

NIPS 2012

AlexNet Network Structure



Pay attention to the output Size and the number of parameters

Training AlexNet using SGD with Momentum and Weight Decay

$$
\begin{aligned}
v_{i+1} &:= 0.9 \cdot v_i - 0.0005 \cdot \epsilon \cdot w_i - \epsilon \cdot \left\langle \frac{\partial L}{\partial w}\Big|_{w_i} \right\rangle_{D_i} \\
w_{i+1} &:= w_i + v_{i+1}
\end{aligned}
$$

where i is the iteration index, v is the momentum variable, ϵ is the learning rate, and $\left\langle \frac{\partial L}{\partial w}\Big|_{w_i} \right\rangle_{D_i}$ is the average over the ith batch D_i of the derivative of the objective with respect to w, evaluated at w_i.

AlexNet with ReLU Converges Much Faster

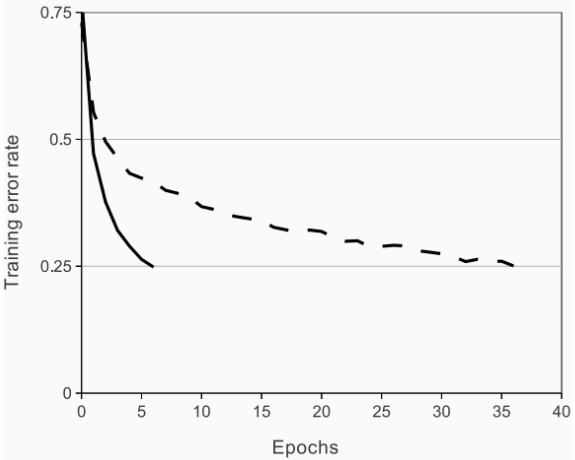


Figure 1: A four-layer convolutional neural network with ReLUs **(solid line)** reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with tanh neurons **(dashed line)**. The learning rates for each network were chosen independently to make training as fast as possible. No regularization of any kind was employed. The magnitude of the effect demonstrated here varies with network architecture, but networks with ReLUs consistently learn several times faster than equivalents with saturating neurons.

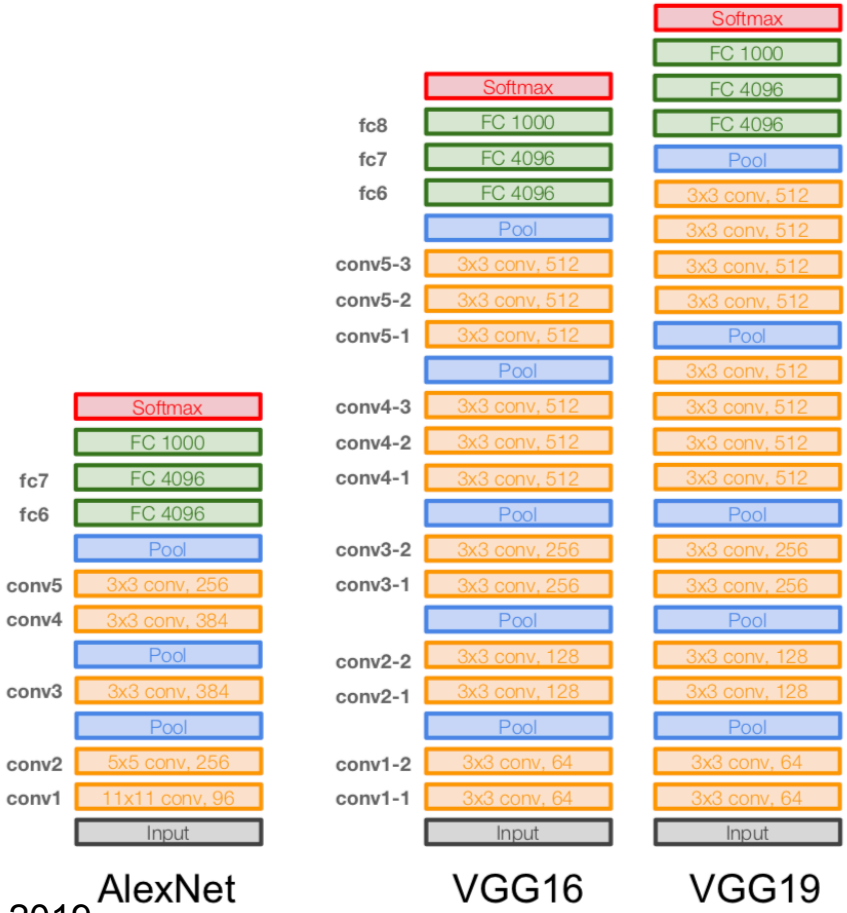
AlexNet vs. VGG

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

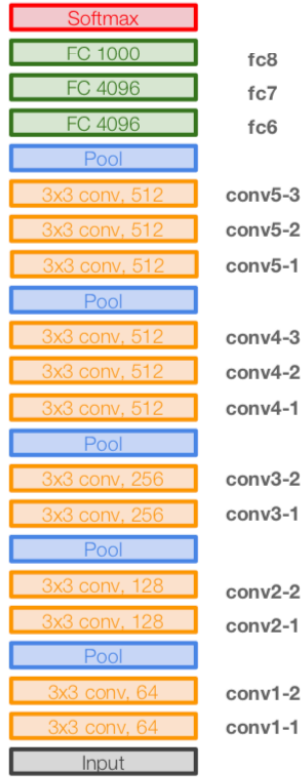


Picture Credit: Fei-Fei, Johnson, and Yeung, Stanford cs231n, 2019

(not counting biases)

INPUT: [224x224x3] memory: 224*224*3=150K params: 0
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



VGG16

Common names

The deeper, the better?

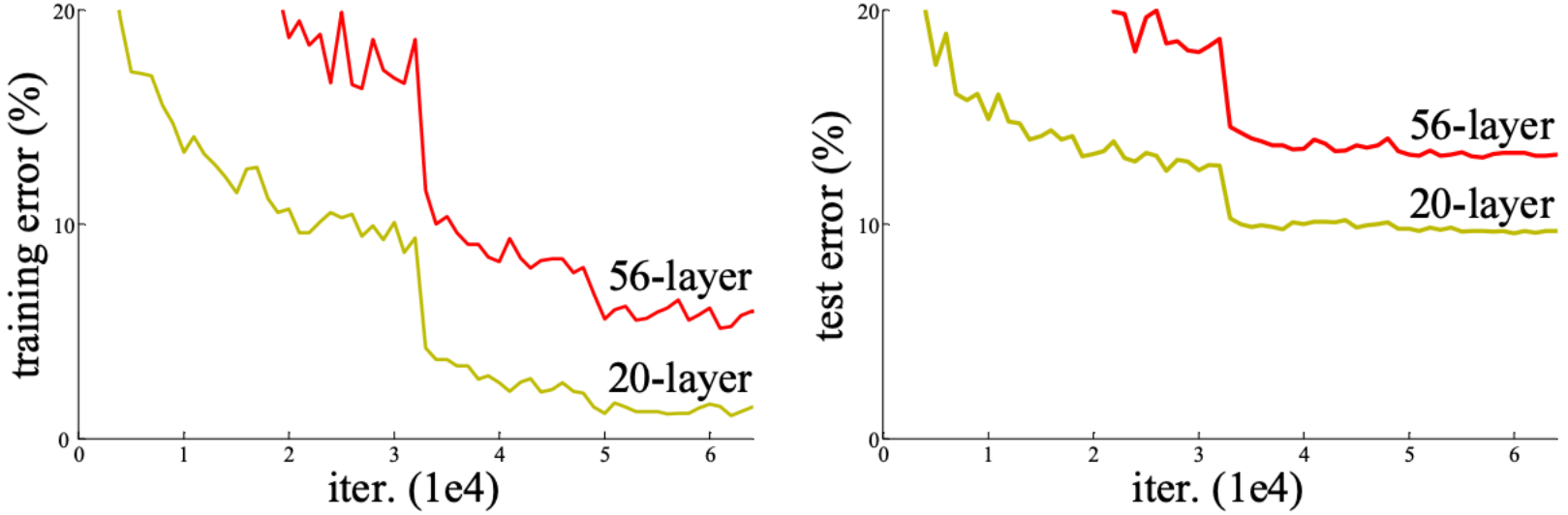


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet

He et al., CVPR 2015

Learning Residual Feature Maps is Easier

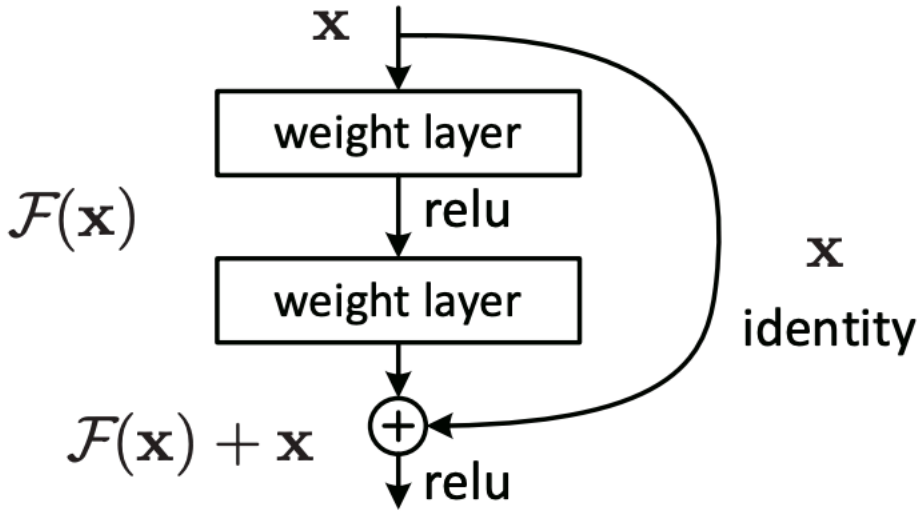
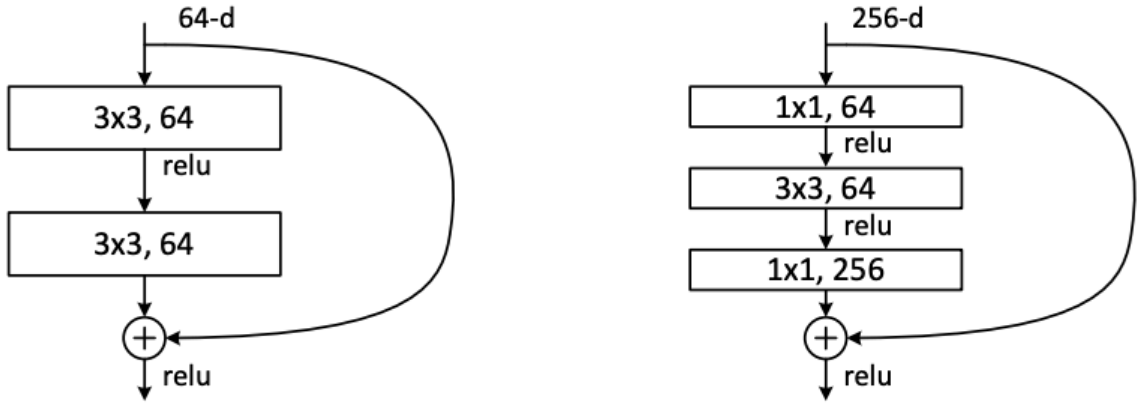


Figure 2. Residual learning: a building block.

He et al., Deep Residual Learning for Image Recognition. CVPR 2015

Learning Residual is Easier



He et al., Deep Residual Learning for Image Recognition. CVPR 2015

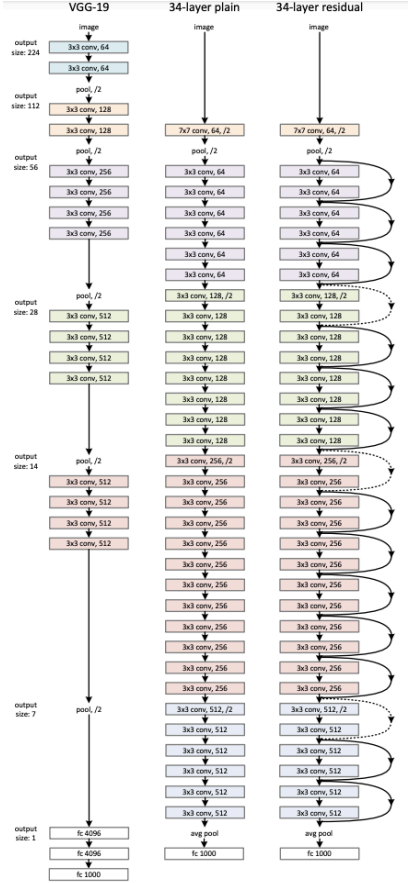


Figure 3. Example network architectures for ImageNet. **Left**: the VGG-19 model [41] (19.6 billion FLOPs) as a reference. **Middle**: a plain network with 34 parameter layers (3.6 billion FLOPs). **Right**: a residual network with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase dimensions. **Table 1** shows more details and other variants.

VGG vs. ResNet

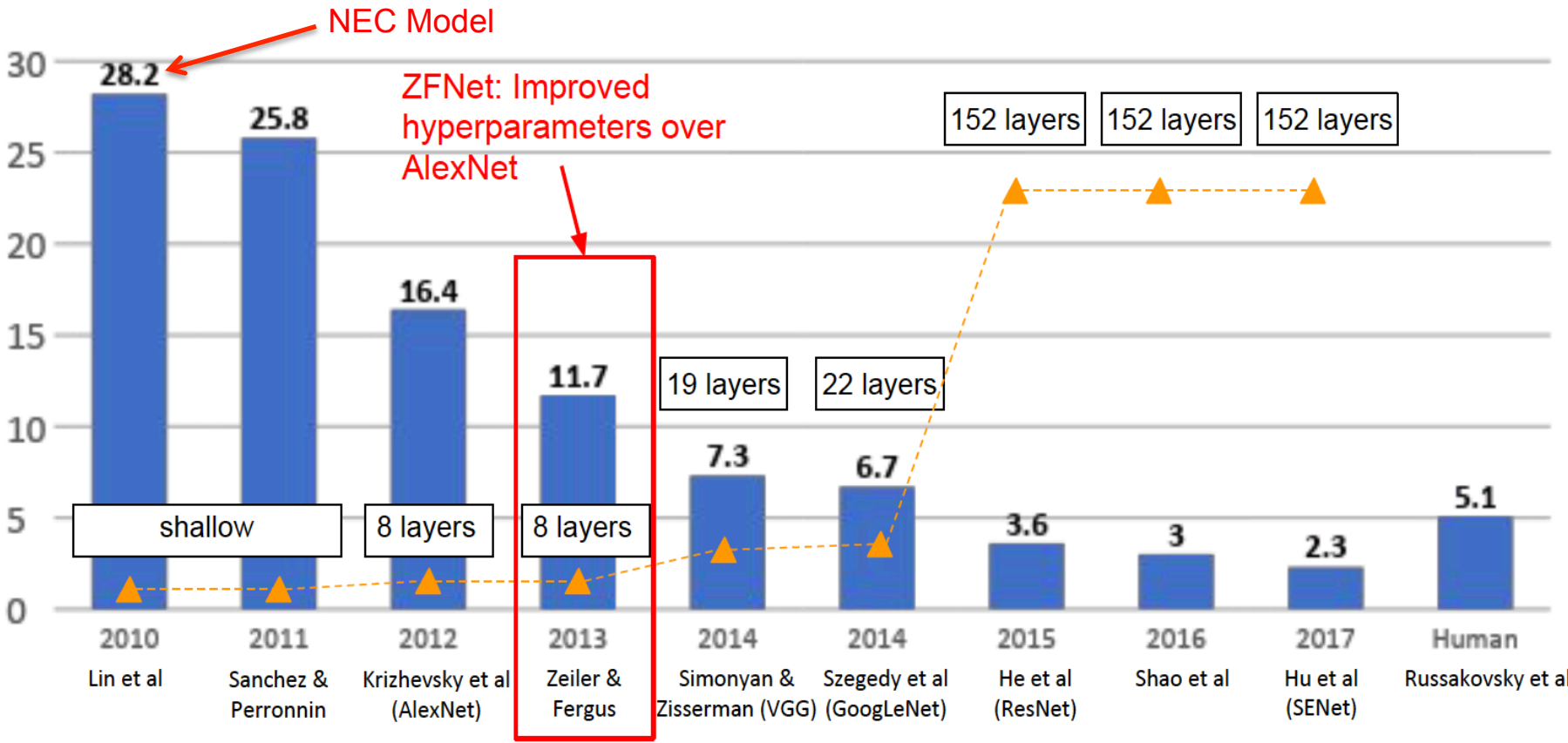
He et al., CVPR 2015

ResNet Details

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Architectures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of blocks stacked. Down-sampling is performed by conv3_1, conv4_1, and conv5_1 with a stride of 2.

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



Picture Credit: Fei-Fei, Johnson, and Yeung, Stanford cs231n, 2019

Conv2d in PyTorch

Conv2d

CLASS `torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')` [SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

where $\star$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

- `stride` controls the stride for the cross-correlation, a single number or a tuple.
- `padding` controls the amount of implicit zero-paddings on both sides for `padding` number of points for each dimension.
- `dilation` controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what `dilation` does.
- `groups` controls the connections between inputs and outputs. `in_channels` and `out_channels` must both be divisible by `groups`. For example,

Demonstration of training a simple CNN Classifier on CIFAR10 using PyTorch in Jupyter Notebook

Implement Your Own Forward and Backforward in PyTorch

```python
import torch

class MyReLU(torch.autograd.Function):
    """
    We can implement our own custom autograd Functions by subclassing
    torch.autograd.Function and implementing the forward and backward passes
    which operate on Tensors.
    """

    @staticmethod
    def forward(ctx, input):
        """
        In the forward pass we receive a Tensor containing the input and return
        a Tensor containing the output. ctx is a context object that can be used
        to stash information for backward computation. You can cache arbitrary
        objects for use in the backward pass using the ctx.save_for_backward method.
        """
        ctx.save_for_backward(input)
        return input.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_output):
        """
        In the backward pass we receive a Tensor containing the gradient of the loss
        with respect to the output, and we need to compute the gradient of the loss
        with respect to the input.
        """
        input, = ctx.saved_tensors
        grad_input = grad_output.clone()
        grad_input[input < 0] = 0
        return grad_input
```

Implement Your Own Forward and Backforward in PyTorch

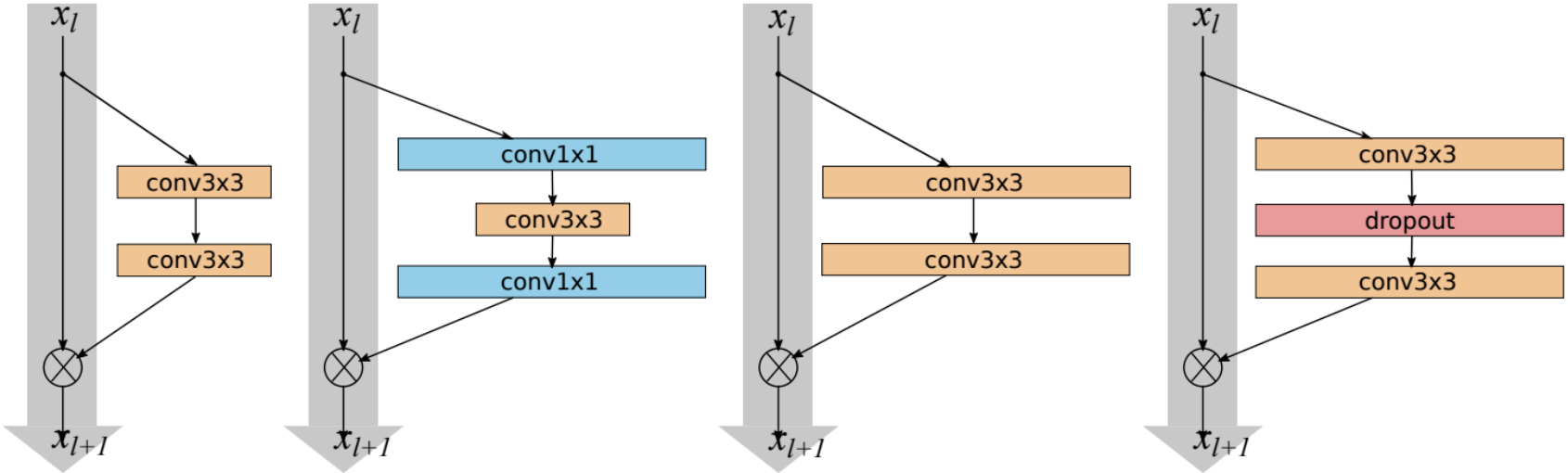
```
dtype = torch.float
device = torch.device("cpu")
# device = torch.device("cuda:0") # Uncomment this to run on GPU

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold input and outputs.
x = torch.randn(N, D_in, device=device, dtype=dtype)
y = torch.randn(N, D_out, device=device, dtype=dtype)

# Create random Tensors for weights.
w1 = torch.randn(D_in, H, device=device, dtype=dtype, requires_grad=True)
w2 = torch.randn(H, D_out, device=device, dtype=dtype, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    # To apply our Function, we use Function.apply method. We alias this as 'relu'.
    relu = MyReLU.apply
```

Implement Your Own Forward and Backforward in PyTorch

```
learning_rate = 1e-6
max_iter = 500
for t in range(max_iter):
    # To apply our Function, we use Function.apply method. We alias this as 'relu'.
    relu = MyReLU.apply

    # Forward pass: compute predicted y using operations; we compute
    # ReLU using our custom autograd operation.
    y_pred = relu(x.mm(w1)).mm(w2)

    # Compute and print loss
    loss = (y_pred - y).pow(2).sum()
    if t % 100 == 99:
        print(t, loss.item())

    # Use autograd to compute the backward pass.
    loss.backward()

    # Update weights using gradient descent
    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad

        # Manually zero the gradients after updating weights
        w1.grad.zero_()
        w2.grad.zero_()
```

The Remaining Slides Are Optional Materials for Your Interest

Wider ResNet



(a) basic (b) bottleneck (c) basic-wide (d) wide-dropout

Various residual blocks used in the paper. Batch normalization and ReLU precede each convolution (omitted for clarity)

Wider ResNet

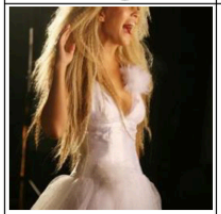
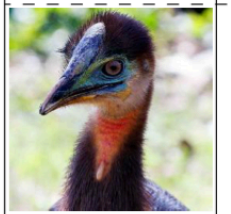
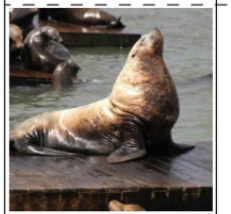
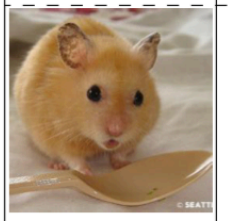
group name	output size	block type = $B(3,3)$
conv1	32×32	$[3\times3,\ 16]$
conv2	32×32	$\begin{bmatrix} 3\times3,\ 16\times k \\ 3\times3,\ 16\times k \end{bmatrix} \times N$
conv3	16×16	$\begin{bmatrix} 3\times3,\ 32\times k \\ 3\times3,\ 32\times k \end{bmatrix} \times N$
conv4	8×8	$\begin{bmatrix} 3\times3,\ 64\times k \\ 3\times3,\ 64\times k \end{bmatrix} \times N$
avg-pool	1×1	$[8 \times 8]$

Structure of wide residual networks. Network width is determined by factor k.

What can we do with a pre-trained Deep CNN on ImageNet?

- Simple Transfer learning
  - We transfer our learned model on the ImageNet to a different domain, for e.g., fine-grained flower category classification
  - It only works when the transferred domain is closely related to the source domain of ImageNet
- Few-shot learning
  - In this task, for each class, we only have a few labeled training examples
  - We can use the learned feature embeddings or their (weighted) mean as prototype(s)
- Zero-shot learning
  - In this task, we don’t have any training example for some classes, but we have semantic descriptions about them
  - A simple idea: Output a 1000-class probabilities of a test image and use a convex combination of the semantic descriptions of the top k known classes to construct semantic features of the test image

Zero-shot Learning Example

Test Image	Softmax Baseline [7]	DeViSE [6]	ConSE (10)
	wig fur coat Saluki, gazelle hound Afghan hound, Afghan stole	water spaniel tea gown bridal gown, wedding gown spaniel tights, leotards	business suit **dress, frock** hairpiece, false hair, postiche swimsuit, swimwear, bathing suit kit, outfit
	ostrich, Struthio camelus black stork, Ciconia nigra vulture crane peacock	heron owl, bird of Minerva, bird of night hawk bird of prey, raptor, raptorial bird finch	**ratite, ratite bird, flightless bird** peafowl, bird of Juno common spoonbill New World vulture, cathartid Greek partridge, rock partridge
	sea lion plane, carpenter's plane cowboy boot loggerhead, loggerhead turtle goose	elephant turtle turtleneck, turtle, polo-neck flip-flop, thong handcart, pushcart, cart, go-cart	California sea lion **Steller sea lion** Australian sea lion South American sea lion eared seal
	hamster broccoli Pomeranian capuchin, ringtail weasel	**golden hamster, Syrian hamster** rhesus, rhesus monkey pipe shaker American mink, Mustela vison	**golden hamster, Syrian hamster** rodent, gnawer Eurasian hamster rhesus, rhesus monkey rabbit, coney, cony

https://arxiv.org/pdf/1312.5650.pdf

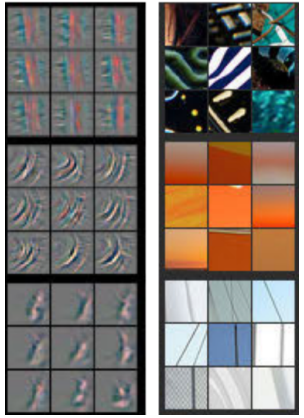
What do CNN (AlexNet-like) filters look like?

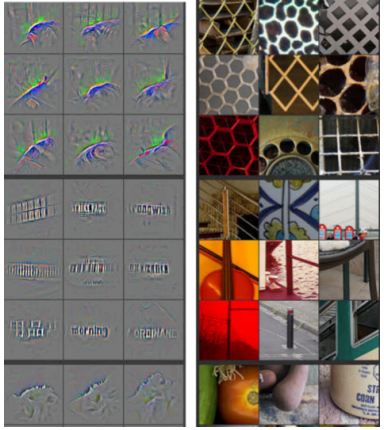
Zeiler and Fergus, 2013:
Visualizing and Understanding Convolutional Networks

An important convolutional operation called Transposed Convolution was invented in this paper, which will be discussed in Lec 5.

Layer 1

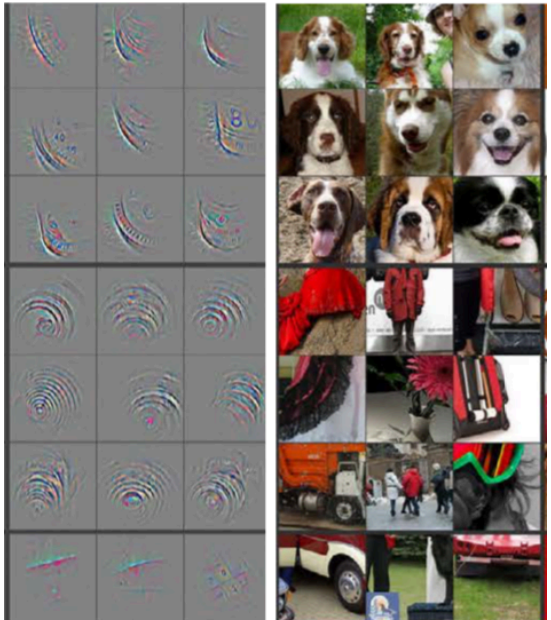
Layer 2

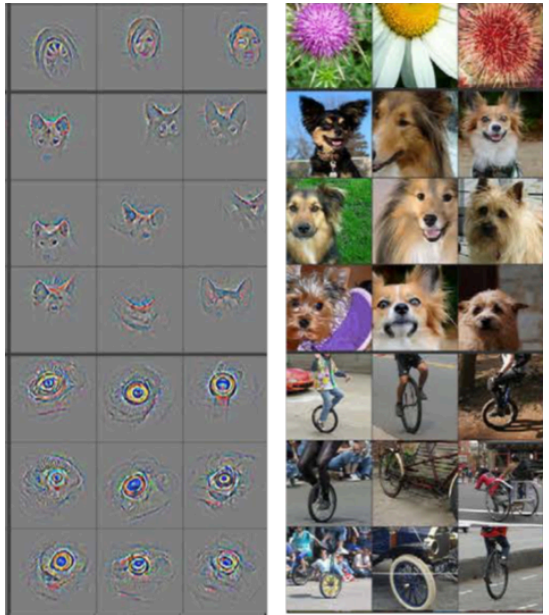




Layer 3

Layer 4





Layer 5

Figure 2. Visualization of features in a fully trained model. For layers 2-5 we show the top 9 activations in a random subset of feature maps across the validation data, projected down to pixel space using our deconvolutional network approach. Our reconstructions are *not* samples from the model: they are reconstructed patterns from the validation set that cause high activations in a given feature map. For each feature map we also show the corresponding image patches. Note: (i) the the strong grouping within each feature map, (ii) greater invariance at higher layers and (iii) exaggeration of

Deep Dream

Image credit and source:
https://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html

Deep Dream

Neural net "dreams"— generated purely from random noise, using a network trained on places by MIT Computer Science and AI Laboratory. See our Inceptionism gallery for hi-res versions of the images above and more (Images marked "Places205-GoogLeNet" were made using this network).

Image credit and source:
https://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html

Deep Dream Python Notebook Code

https://github.com/google/deepdream/blob/master/dream.ipynb

Style Transfer

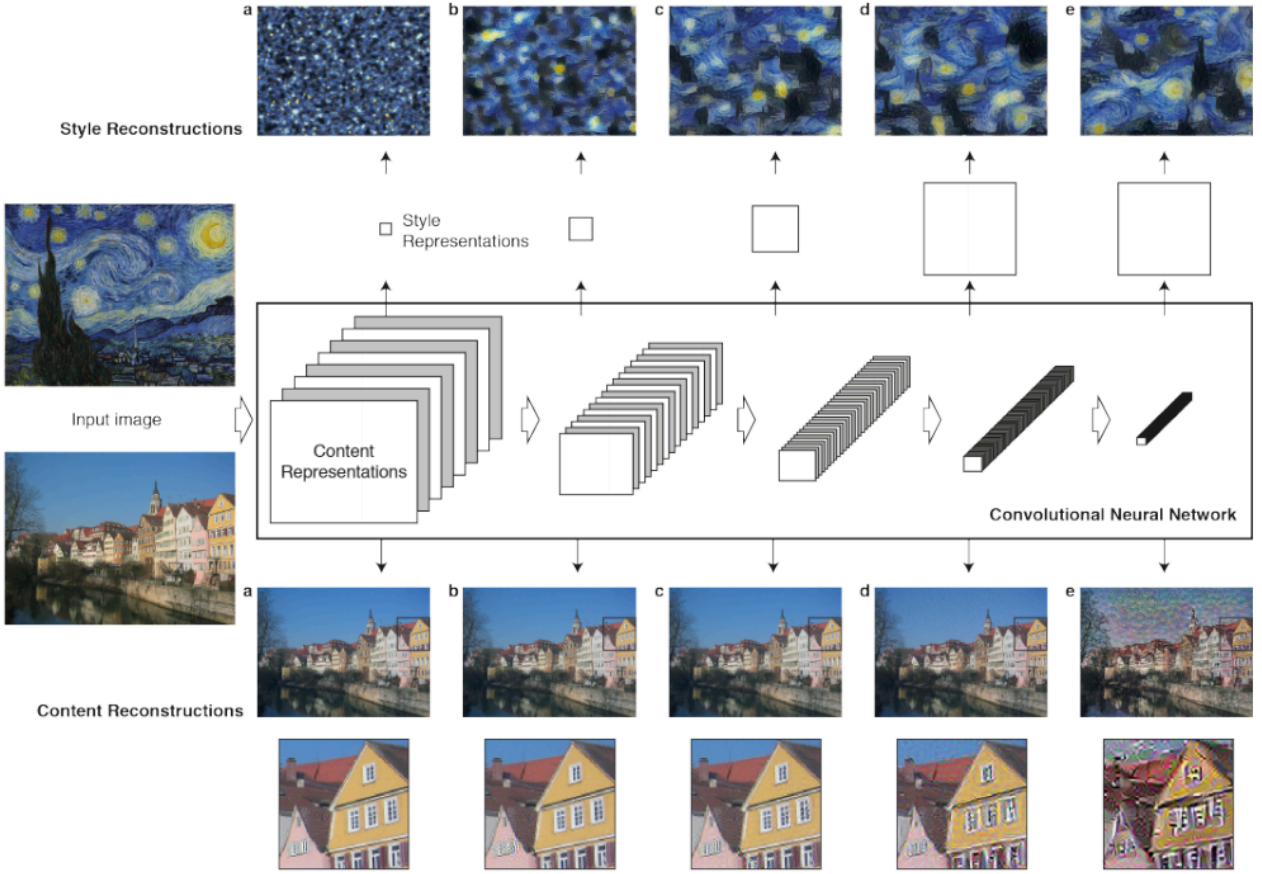
- We can also utilize a pre-trained deep CNN classifier such as VGG to calculate the feature maps in some specified layers for input images

- By matching the content feature maps of a generated image to an original input image and matching the Gram matrix of feature maps of the generated image to that of a style image, we can perform backpropagation to the pixel space to generate an artistic image similar to the input image

Gatys et al. 2015: A Neural Algorithm of Artistic Style

https://arxiv.org/pdf/1508.06576.pdf

Style Transfer



Style Transfer Method

$$\mathcal{L}_{content}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i,j} \left(F_{ij}^l - P_{ij}^l\right)^2 . \quad (1)$$

The derivative of this loss with respect to the activations in layer l equals

$$\frac{\partial \mathcal{L}_{content}}{\partial F_{ij}^l} = \begin{cases} \left(F^l - P^l\right)_{ij} & \text{if } F_{ij}^l > 0 \\ 0 & \text{if } F_{ij}^l < 0 . \end{cases} \quad (2)$$

$$\mathcal{L}_{total}(\vec{p}, \vec{a}, \vec{x}) = \alpha \mathcal{L}_{content}(\vec{p}, \vec{x}) + \beta \mathcal{L}_{style}(\vec{a}, \vec{x})$$

where α and β are the weighting factors for content and style reconstruction respectively.

A^l and G^l their respective style representations in layer l. The contribution of that layer to the total loss is then

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} \left(G_{ij}^l - A_{ij}^l\right)^2 \quad (4)$$

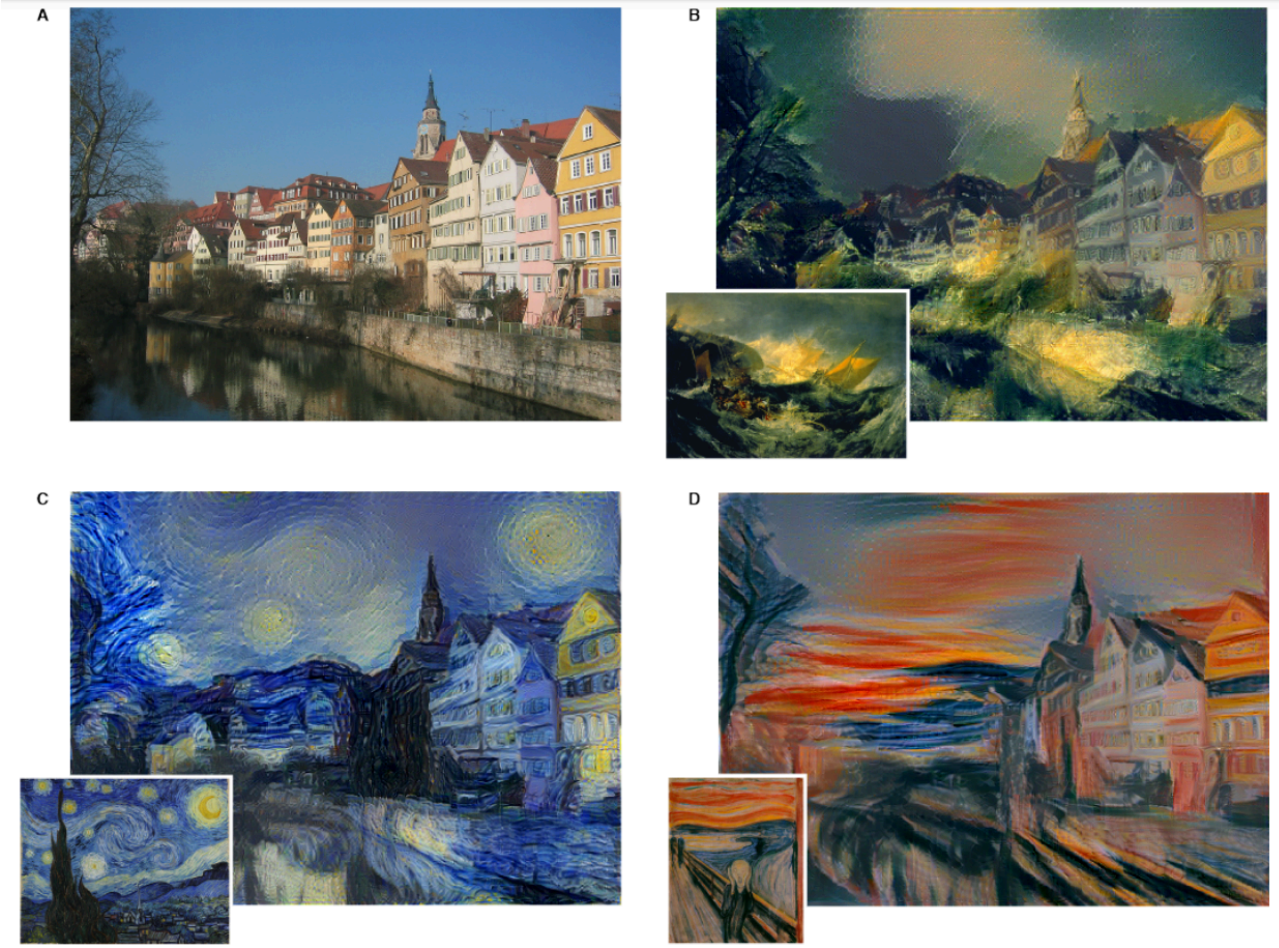
and the total loss is

$$\mathcal{L}_{style}(\vec{a}, \vec{x}) = \sum_{l=0}^{L} w_l E_l \quad (5)$$

where w_l are weighting factors of the contribution of each layer to the total loss (see below for specific values of w_l in our results). The derivative of E_l with respect to the activations in layer l can be computed analytically:

$$\frac{\partial E_l}{\partial F_{ij}^l} = \begin{cases} \frac{1}{N_l^2 M_l^2} \left((F^l)^{\mathrm{T}} \left(G^l - A^l\right)\right)_{ji} & \text{if } F_{ij}^l > 0 \\ 0 & \text{if } F_{ij}^l < 0 . \end{cases} \quad (6)$$

Style Transfer



Style Transfer Examples with a Fast Implementation

https://youtu.be/Khuj4ASldmU?t=6

Summary of Topics Discussed

- Activation Functions
- Loss Functions
- Training deep feedforward neural networks with backpropagation and mini-batch SGD
- Convolution and pooling operations in CNN
- Network architectures such as AlexNet, VGG, ResNet, and WideResNet
- Applications of supervised pre-trained CNNs
- Visualization of pre-trained CNN filters and receptive fields
- Style transfer
- Geoff Hinton, “Never stop coding.” Great discoveries are from practice.

The End

Next lecture:

Optimization, Regularization, Understanding Batch Normalization, and Robustness of Deep Neural Networks