An Introduction to Convolutional Neural Networks

Yuan YAO

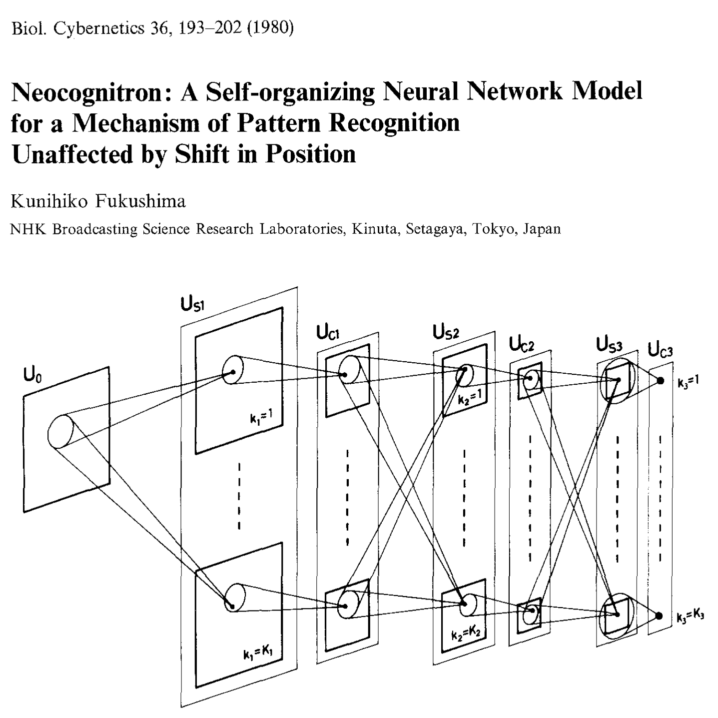
HKUST

Summary

- We had covered so far
  - Linear models (linear and logistic regression) – always a good start, simple yet powerful
  - Model Assessment and Selection – basics for all methods
  - Trees, Random Forests, and Boosting – good for high dim mixed-type heterogeneous features
  - Support Vector Machines – good for small amount of data but high dim geometric features
- Next, neural networks for unstructured data (image, language etc.):
  - **Convolutional Neural Networks** – image data
  - Recurrent Neural Networks, LSTM – sequence data
  - Transformer, BERT – machine translation etc.
  - Generative models and GANs – new unsupervised learning for image, etc.
  - Reinforcement Learning – Markov decision process, playing games, etc.

Convolutional Neural Networks: shift invariances and locality for images

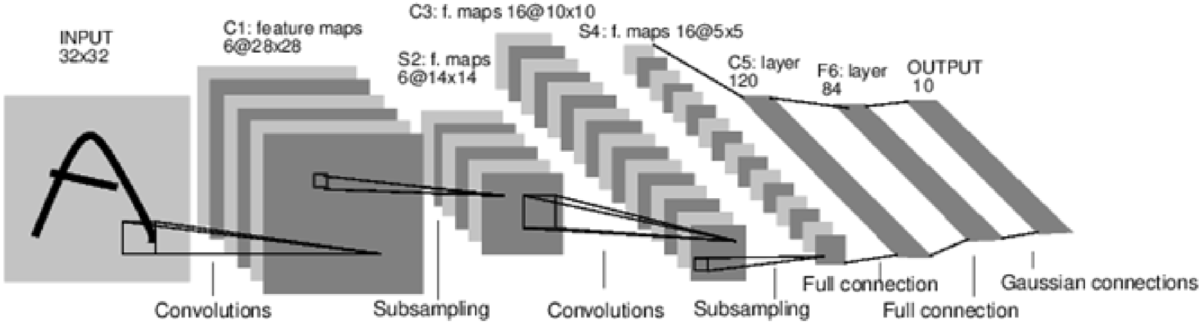
- Can be traced to *Neocognitron* of Kunihiko Fukushima (1979)
- Yann LeCun combined convolutional neural networks with back propagation (1989)
- Imposes **shift invariance** and **locality** on the weights
- Forward pass remains similar
- Backpropagation slightly changes – need to sum over the gradients from all spatial positions


Biol. Cybernetics 36, 193–202 (1980)

Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position

Kunihiko Fukushima

NHK Broadcasting Science Research Laboratories, Kinuta, Setagaya, Tokyo, Japan

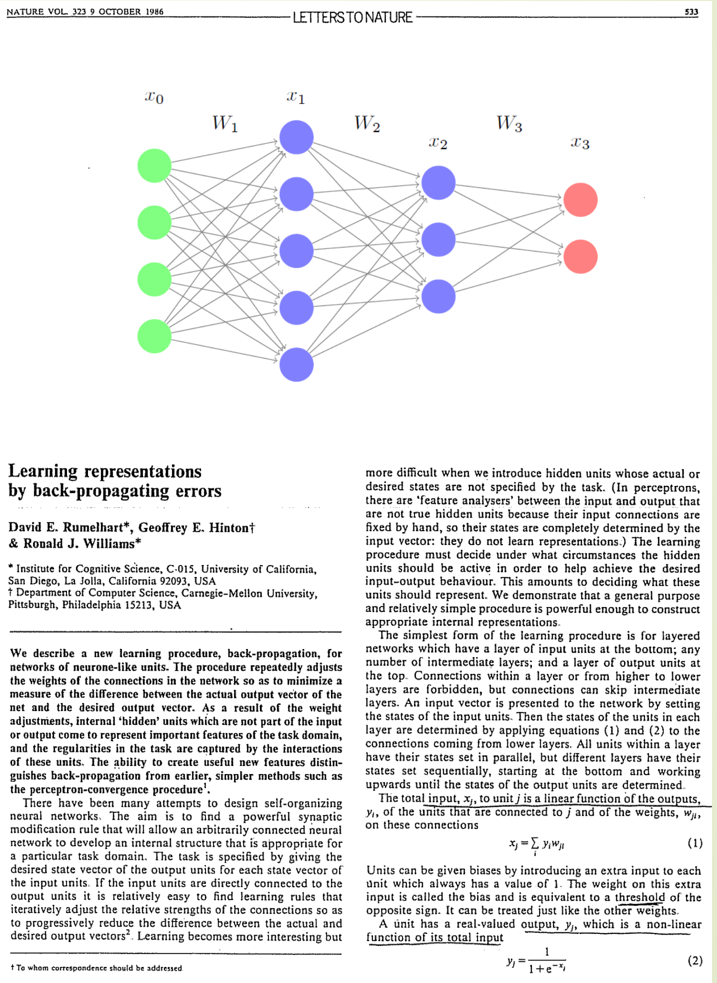


Multilayer Perceptrons (MLP) and Back-Propagation (BP) Algorithms

Rumelhart, Hinton, Williams (1986)
Learning representations by back-propagating errors, Nature, 323(9): 533-536

BP algorithms as **stochastic gradient descent** algorithms (**Robbins–Monro 1950; Kiefer-Wolfowitz 1951**) with Chain rules of Gradient maps

MLP classifies **XOR**, but the global hurdle on topology (connectivity) computation still exists



Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

There have been many attempts to design self-organizing neural networks. The aim is to find a powerful synaptic modification rule that will allow an arbitrarily connected neural network to develop an internal structure that is appropriate for a particular task domain. The task is specified by giving the desired state vector of the output units for each state vector of the input units. If the input units are directly connected to the output units it is relatively easy to find learning rules that iteratively adjust the relative strengths of the connections so as to progressively reduce the difference between the actual and desired output vectors[2]. Learning becomes more interesting but more difficult when we introduce hidden units whose actual or desired states are not specified by the task. (In perceptrons, there are 'feature analysers' between the input and output that are not true hidden units because their input connections are fixed by hand, so their states are completely determined by the input vector: they do not learn representations.) The learning procedure must decide under what circumstances the hidden units should be active in order to help achieve the desired input–output behaviour. This amounts to deciding what these units should represent. We demonstrate that a general purpose and relatively simple procedure is powerful enough to construct appropriate internal representations.

The simplest form of the learning procedure is for layered networks which have a layer of input units at the bottom; any number of intermediate layers; and a layer of output units at the top. Connections within a layer or from higher to lower layers are forbidden, but connections can skip intermediate layers. An input vector is presented to the network by setting the states of the input units. Then the states of the units in each layer are determined by applying equations (1) and (2) to the connections coming from lower layers. All units within a layer have their states set in parallel, but different layers have their states set sequentially, starting at the bottom and working upwards until the states of the output units are determined.

The total input, x_j, to unit j is a linear function of the outputs, y_i, of the units that are connected to j and of the weights, w_{ji}, on these connections

$$x_j = \sum_i y_i w_{ji} \qquad (1)$$

Units can be given biases by introducing an extra input to each unit which always has a value of 1. The weight on this extra input is called the bias and is equivalent to a threshold of the opposite sign. It can be treated just like the other weights.

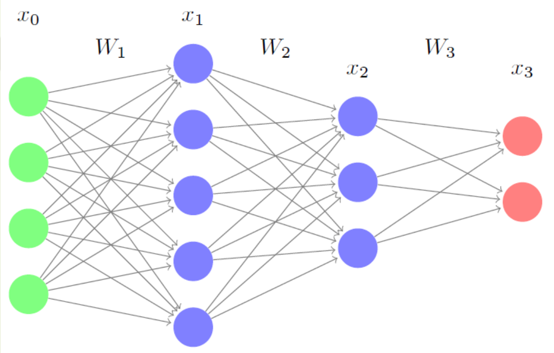
A unit has a real-valued output, y_j, which is a non-linear function of its total input

$$y_j = \frac{1}{1+e^{-x_j}} \qquad (2)$$

† To whom correspondence should be addressed.

BP Algorithm: Forward Pass

- Cascade of repeated [linear operation followed by coordinatewise nonlinearity]'s
- Nonlinearities: sigmoid, hyperbolic tangent, (recently) ReLU.



Algorithm 1 Forward pass

Input: x_0
Output: x_L

1: **for** $\ell = 1$ to L **do**
2: $\quad x_\ell = f_\ell(W_\ell x_{\ell-1} + b_\ell)$
3: **end for**

BP algorithm = Gradient Descent Method

- Training examples $\{x_0^i\}_{i=1}^n$ and labels $\{y^i\}_{i=1}^n$
- Output of the network $\{x_L^i\}_{i=1}^m$
- Objective

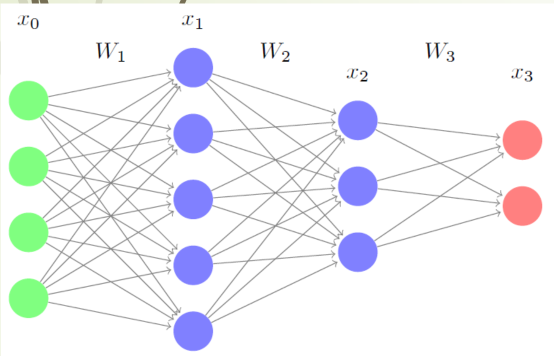
$$J(\{W_l\},\{b_l\}) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}\|y^i - x_L^i\|_2^2 \qquad (1)$$

Other losses include cross-entropy, logistic loss, exponential loss, etc.

- Gradient descent

$$W_l = W_l - \eta\frac{\partial J}{\partial W_l}$$

$$b_l = b_l - \eta\frac{\partial J}{\partial b_l}$$



In practice: use Stochastic Gradient Descent (SGD)

Derivation of BP: Lagrangian Multiplier

LeCun et al. 1988

Given n training examples $(I_i, y_i) \equiv$ (input,target) and L layers

- Constrained optimization

$$\min_{W,x} \quad \sum_{i=1}^{n} \|x_i(L) - y_i\|_2$$

subject to $x_i(\ell) = f_\ell\Big[W_\ell x_i\,(\ell-1)\Big],$

$$i = 1, \ldots, n, \quad \ell = 1, \ldots, L, \; x_i(0) = I_i$$

- Lagrangian formulation (Unconstrained)

$$\min_{W,x,B} \mathcal{L}(W, x, B)$$

$$\mathcal{L}(W, x, B) = \sum_{i=1}^{n} \Bigg\{ \|x_i(L) - y_i\|_2^2 + \sum_{\ell=1}^{L} B_i(\ell)^T \Big(x_i(\ell) - f_\ell\Big[W_\ell x_i\,(\ell-1)\Big]\Big) \Bigg\}$$

http://yann.lecun.com/exdb/publis/pdf/lecun-88.pdf

back-propagation – derivation

- $\frac{\partial \mathcal{L}}{\partial B}$

Forward pass

$$x_i(\ell) = f_\ell\Big[\underbrace{W_\ell x_i(\ell-1)}_{A_i(\ell)}\Big] \quad \ell = 1,\ldots,L, \quad i = 1,\ldots,n$$

- $\frac{\partial \mathcal{L}}{\partial x}$, $z_\ell = [\nabla f_\ell]B(\ell)$

Backward (adjoint) pass

$$z(L) = 2\nabla f_L\Big[A_i(L)\Big](y_i - x_i(L))$$

$$z_i(\ell) = \nabla f_\ell\Big[A_i(\ell)\Big]W_{\ell+1}^T z_i(\ell+1) \quad \ell = 0,\ldots,L-1$$

- $W \leftarrow W + \lambda \frac{\partial \mathcal{L}}{\partial W}$

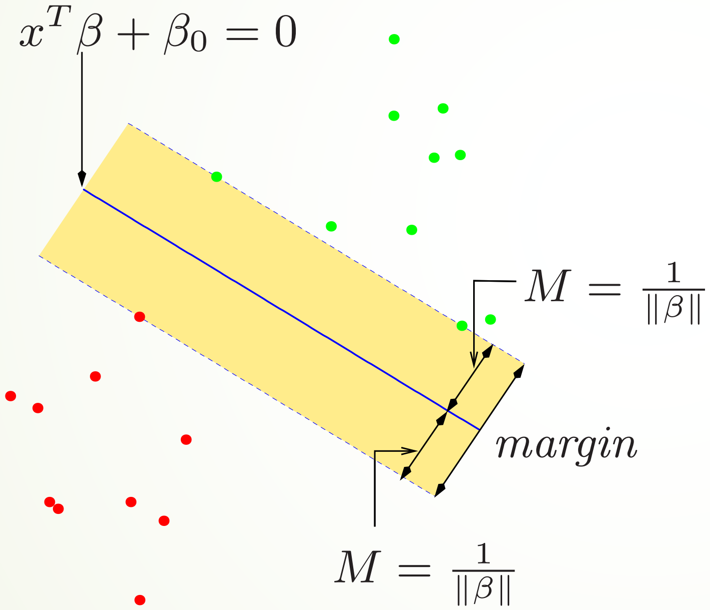
Weight update

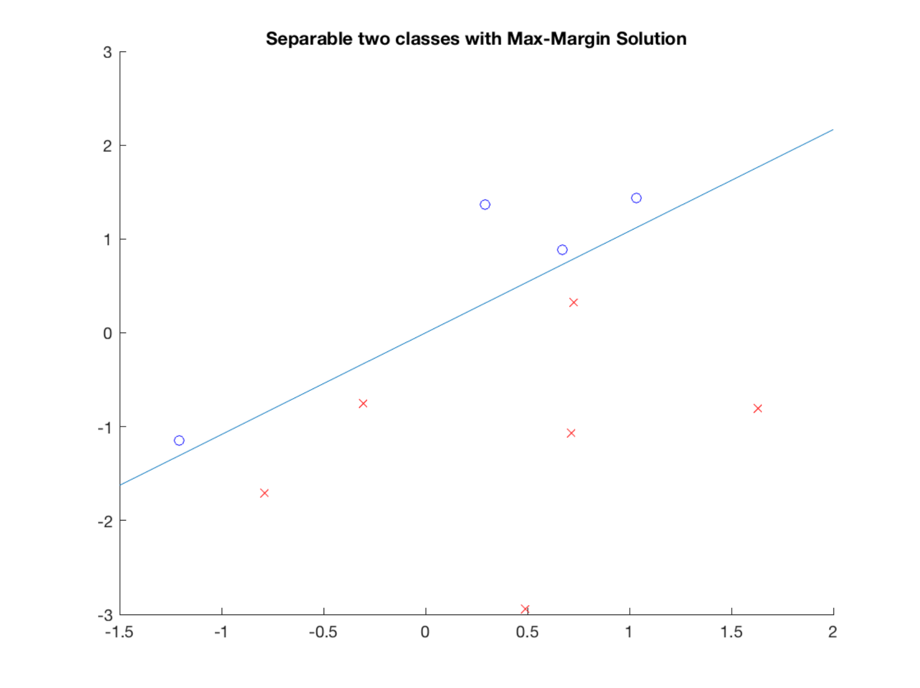
$$W_\ell \leftarrow W_\ell + \lambda \sum_{i=1}^{n} z_i(\ell) x_i^T(\ell-1)$$

Support Vector Machine (Max-Margin Classifier)

$$\text{minimize}_{\beta_0,\beta_1,\ldots,\beta_p} \|\beta\|^2 := \sum_j \beta_j^2$$

$$\text{subject to } y_i(\beta_0 + \beta_1 x_{i1} + \ldots + \beta_p x_{ip}) \geq 1 \text{ for all } i$$





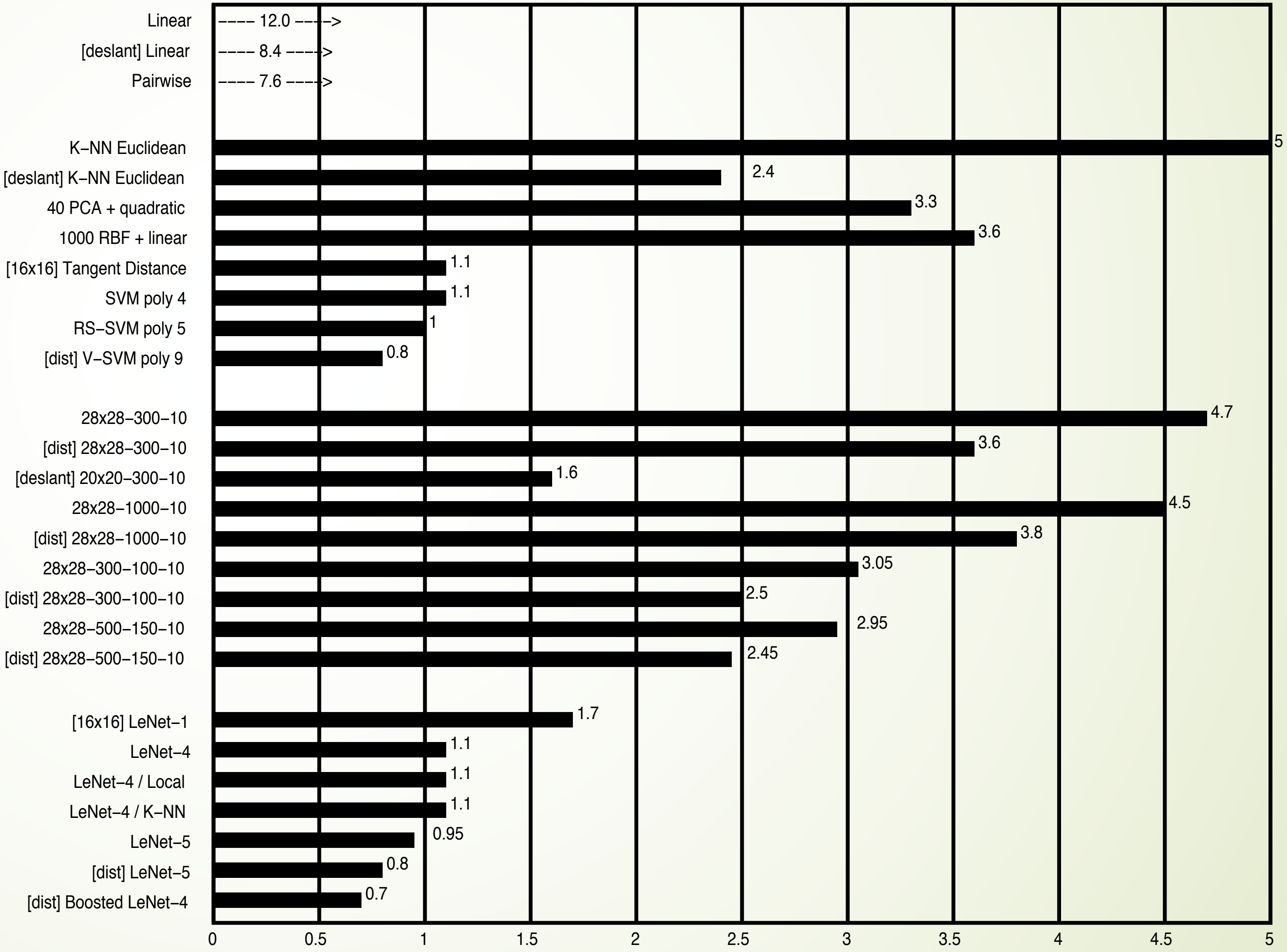
Vladmir Vapnik, 1994

Convex optimization + Reproducing Kernel Hilbert Spaces (Grace Wahba etc.)

MNIST Challenge Test Error: SVM vs. CNN LeCun et al. 1998

Simple SVM performs as well as Multilayer Convolutional Neural Networks which need careful tuning (LeNets)

Second dark era for NN: 2000s



LeNet

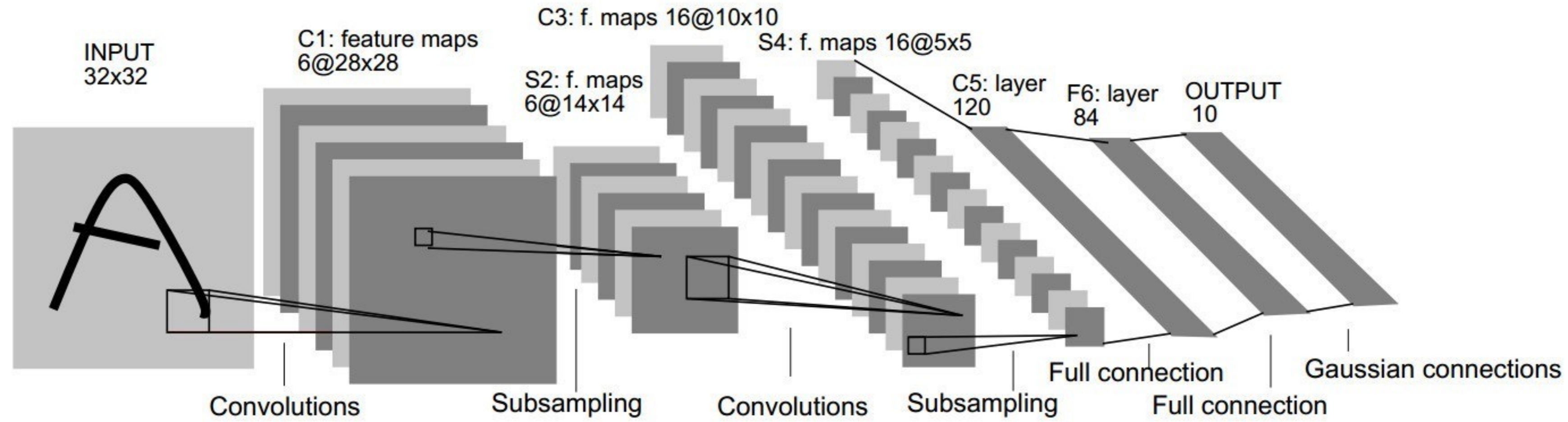
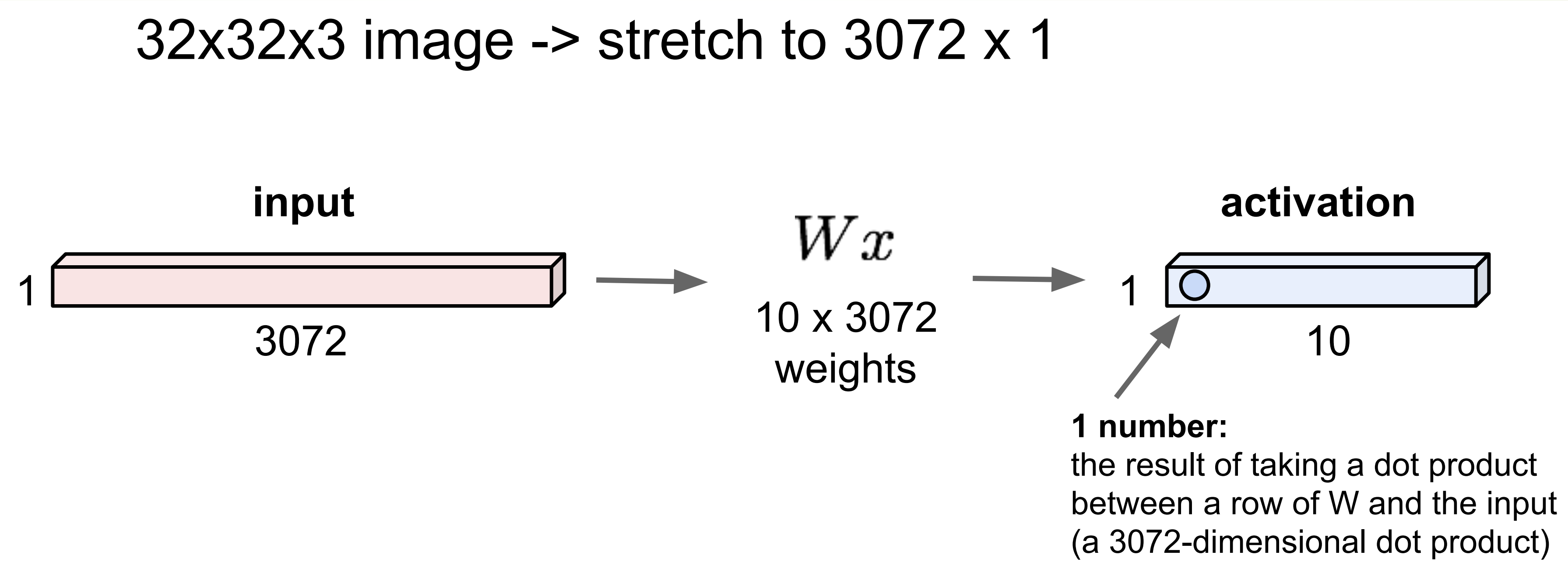


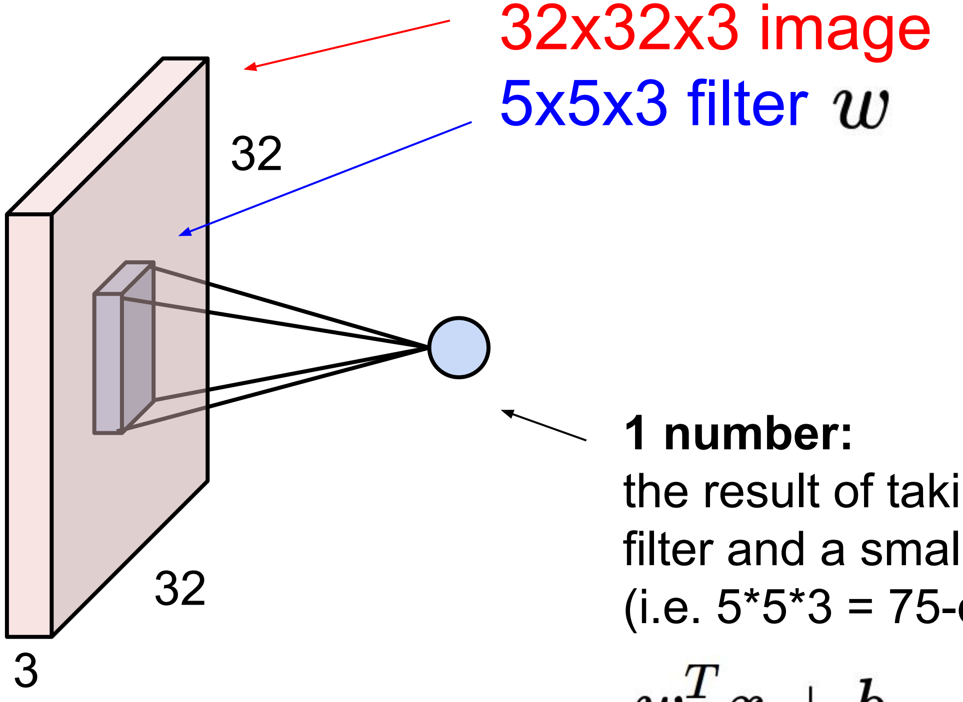
Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

- **Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner**. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, november 1998.

Fully Connected Layer



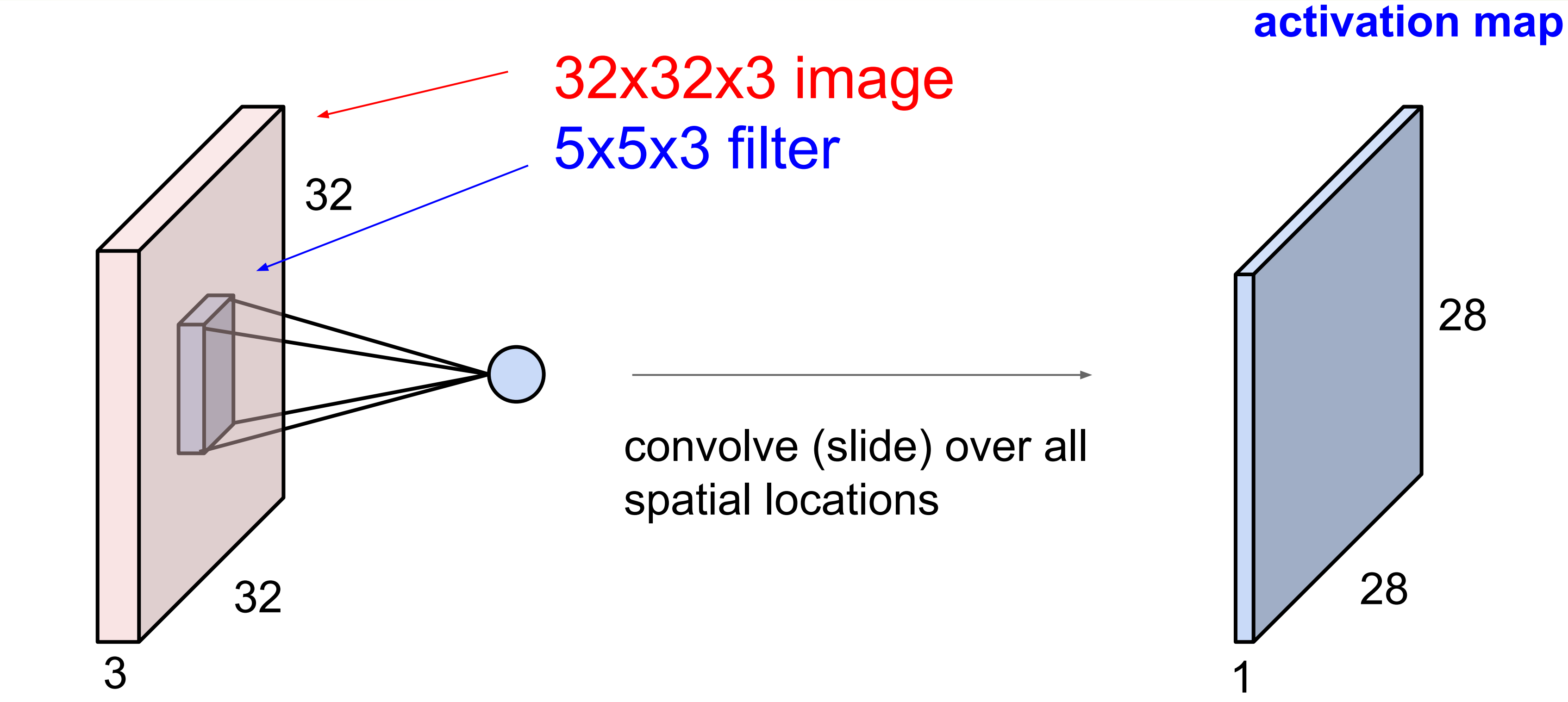
Convolution



1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

$$w^T x + b$$

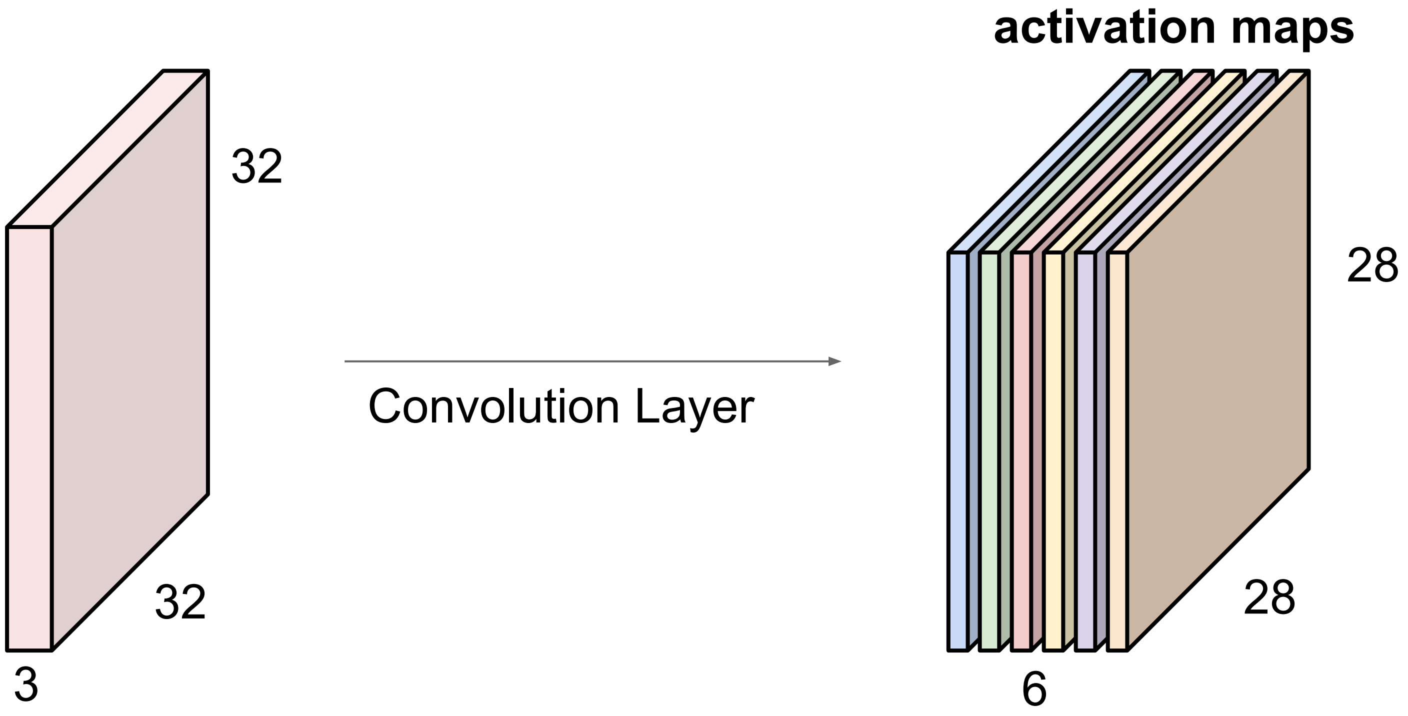
Convolution Layer: a first (blue) filter



Convolution Layer: a second (green) filter

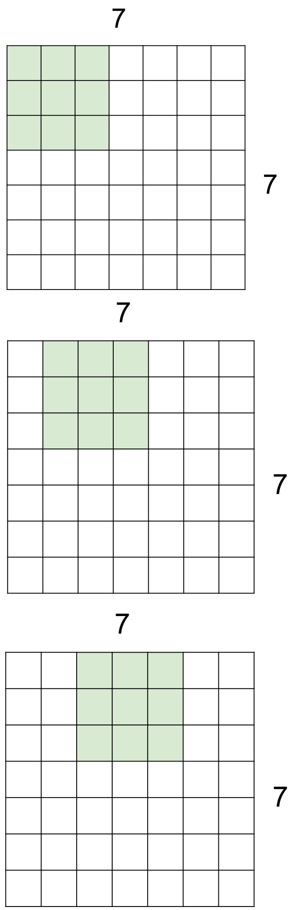
Convolution Layer

For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We stack these up to get a "new image" of size 28x28x6!

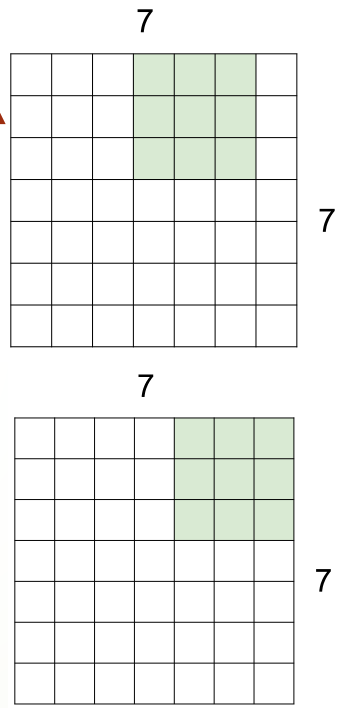
A Closer Look at Convolution: stride=1



7x7 input (spatially)
assume 3x3 filter

7x7 input (spatially)
assume 3x3 filter

7x7 input (spatially)
assume 3x3 filter

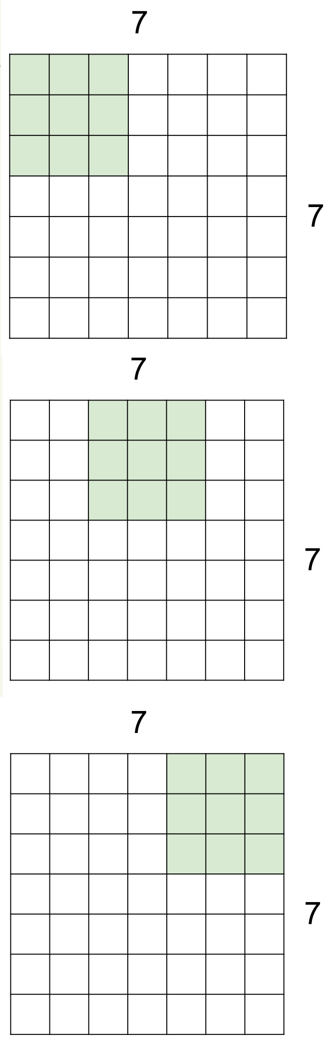


7x7 input (spatially)
assume 3x3 filter

7x7 input (spatially)
assume 3x3 filter

=> 5x5 output

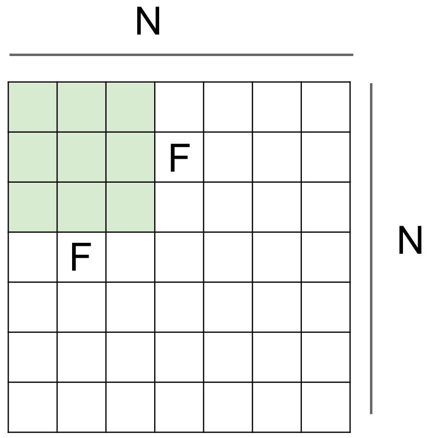
A Closer Look at Convolution: stride=2



7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

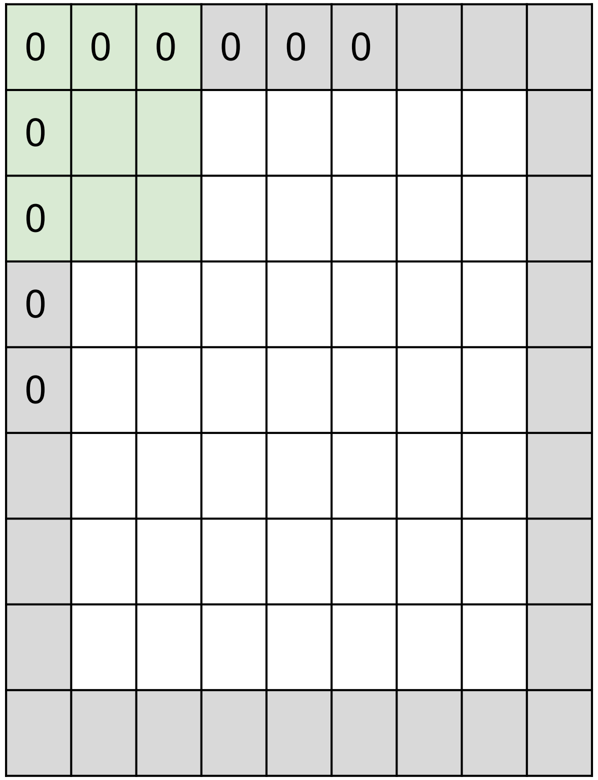
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output!



Output size:
(N - F) / stride + 1

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33 :\

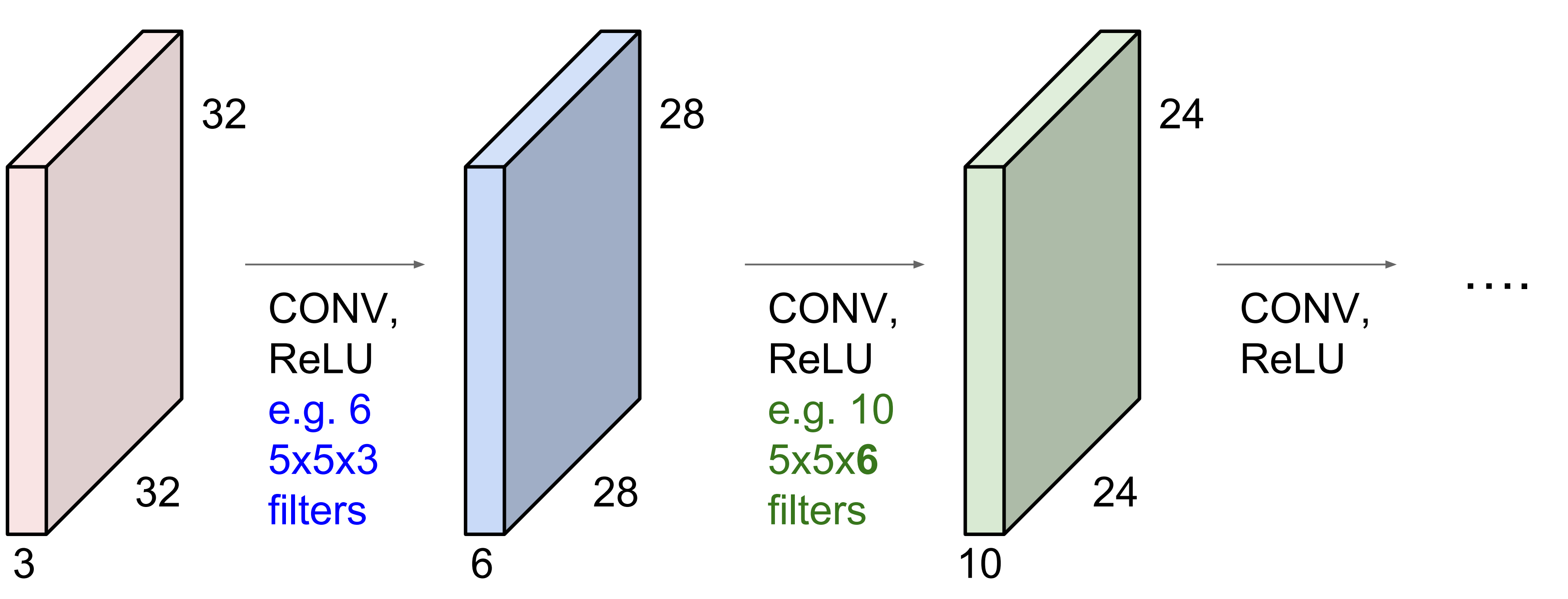
A Closer Look at Convolution: Padding



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!
in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

ConvNet:



Stride = 1
Padding = 0

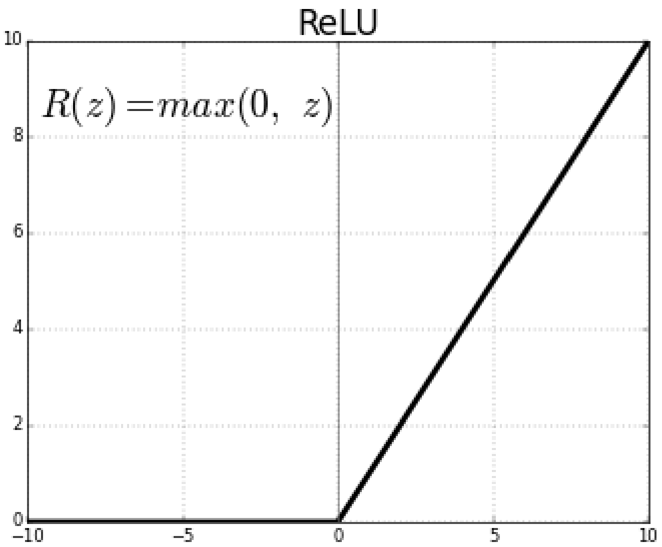
Formula: NewImageSize = floor((ImageSize – Filter + 2*Padding)/Stride + 1)

Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

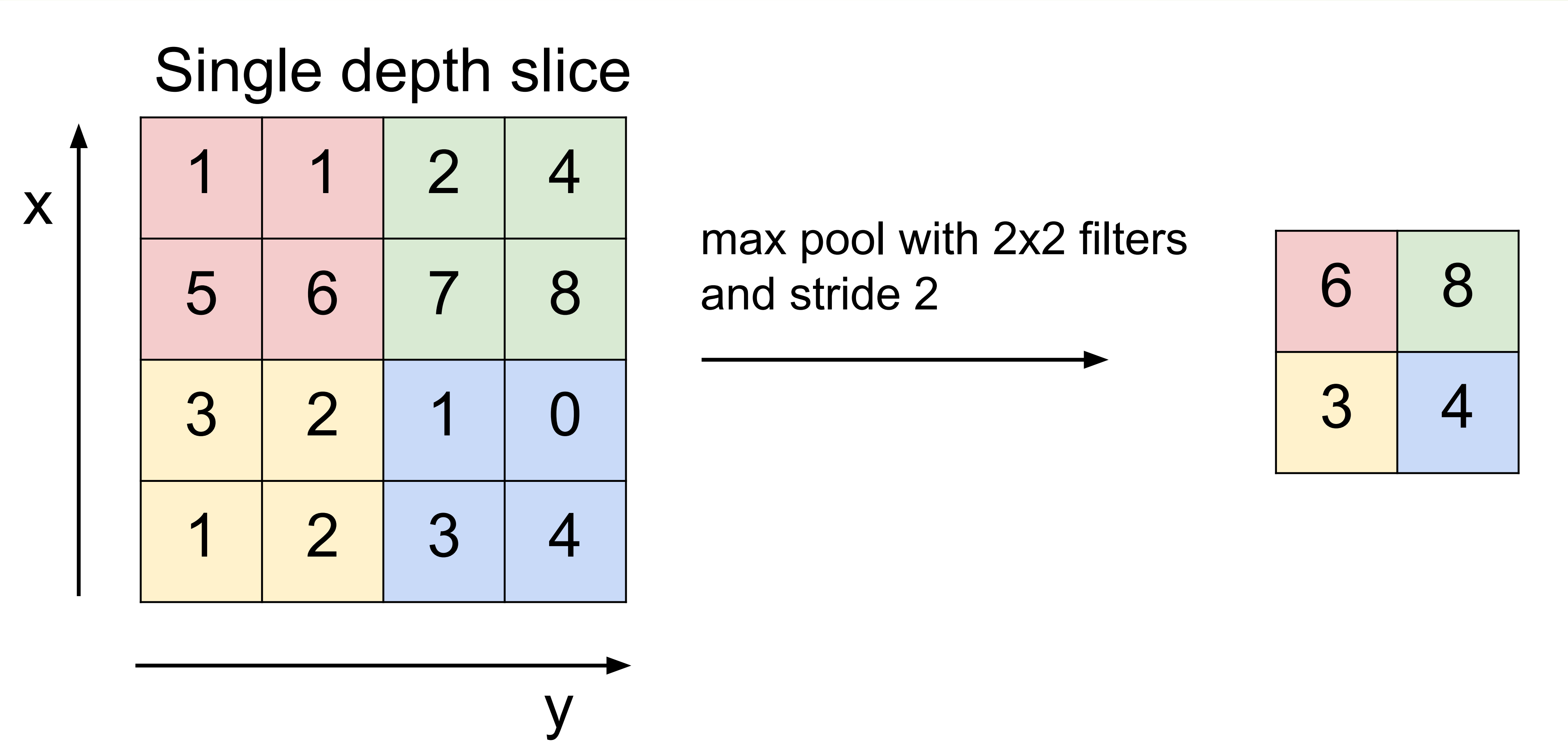
ReLU

- Non-saturating function and therefore faster convergence when compared to other nonlinearities
- Problem of dying neurons



Source: `https://ml4a.github.io/ml4a/neural_networks/`

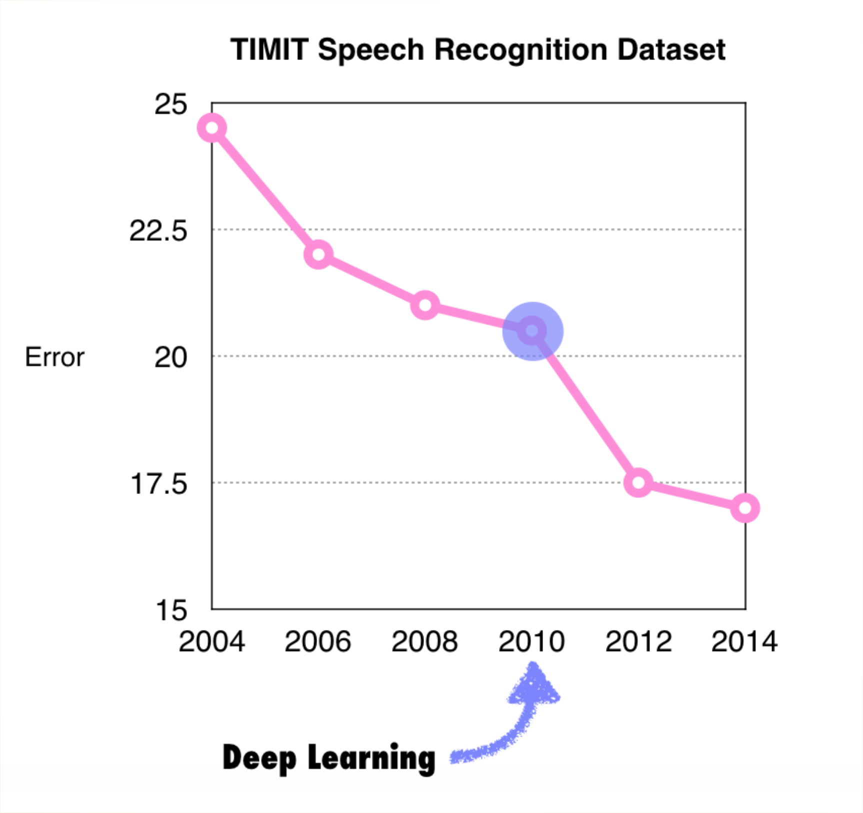
Max Pooling


Single depth slice
x
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
y
max pool with 2x2 filters
and stride 2
6 8
3 4

2000-2010: The Era of SVM, Boosting, … as nights of Neural Networks

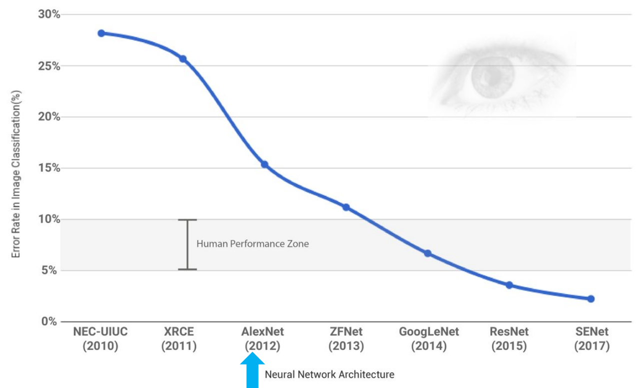
Around the year of 2012...

Speech Recognition: TIMIT



Computer Vision: ImageNet

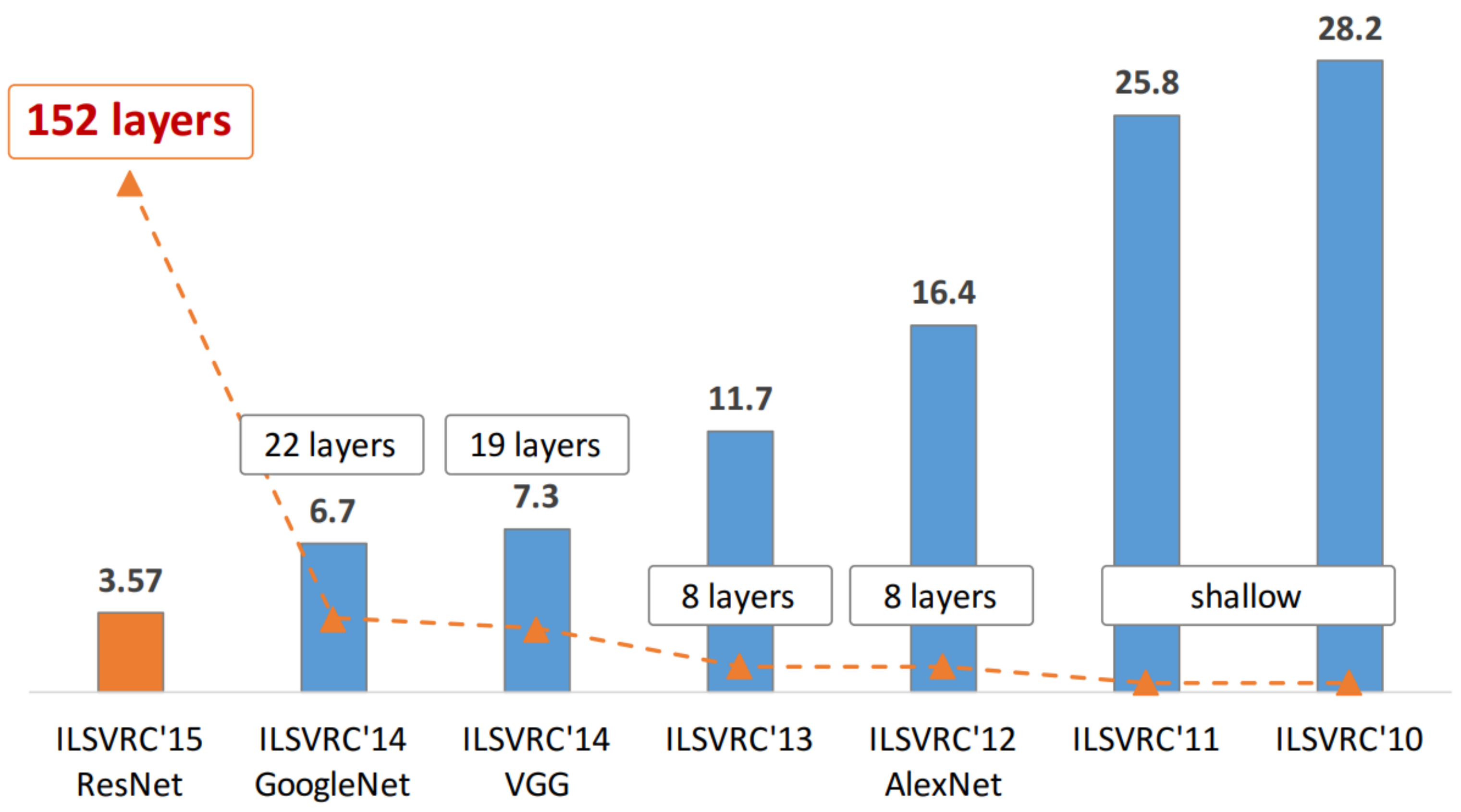
- ImageNet (subset):
  - 1.2 million training images
  - 100,000 test images
  - 1000 classes
- ImageNet large-scale visual recognition Challenge



source: https://www.linkedin.com/pulse/must-read-path-breaking-papers-image-classification-muktabh-mayank

Deep Learning

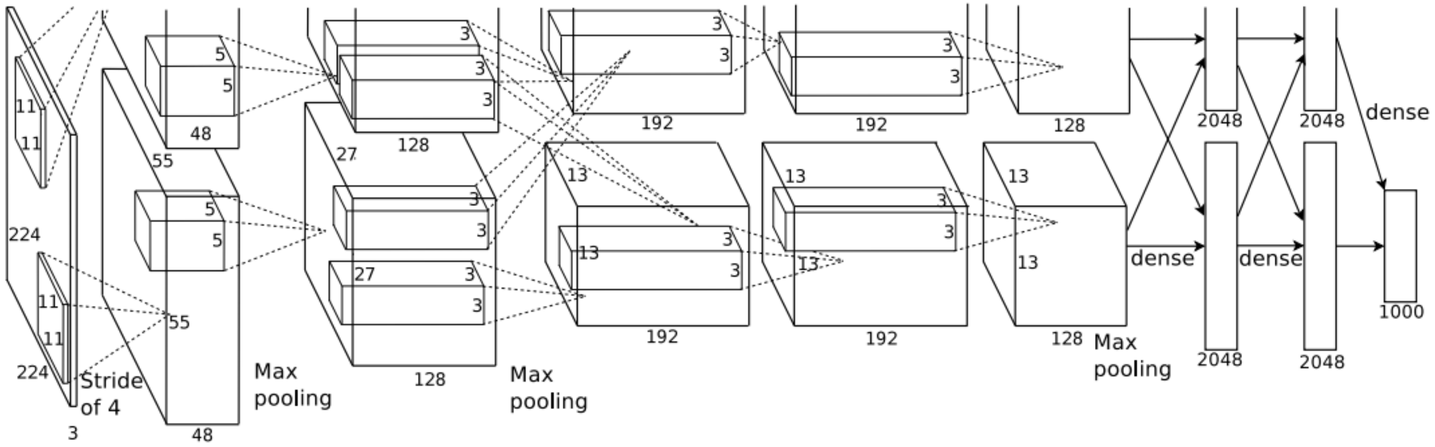
Depth as function of year



[He et al., 2016]

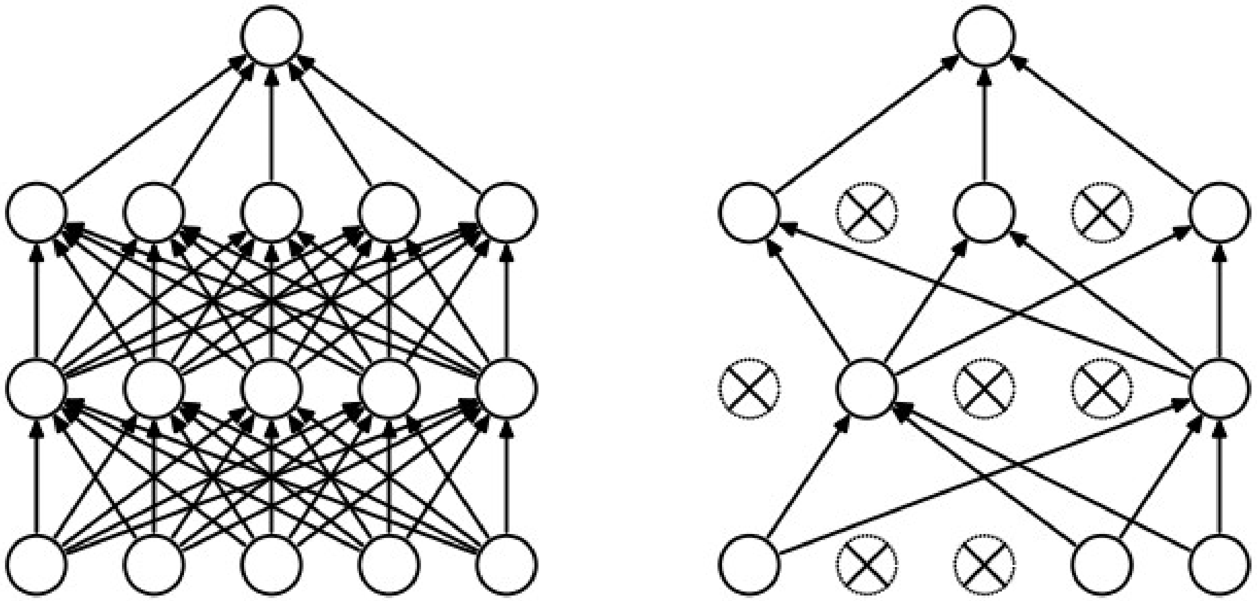
AlexNet (2012): Architecture

- 8 layers: first 5 convolutional, rest fully connected
- ReLU nonlinearity
- Local response normalization
- Max-pooling
- Dropout



Source: [Krizhevsky et al., 2012]

AlexNet (2012): Dropout



(a) Standard Neural Net

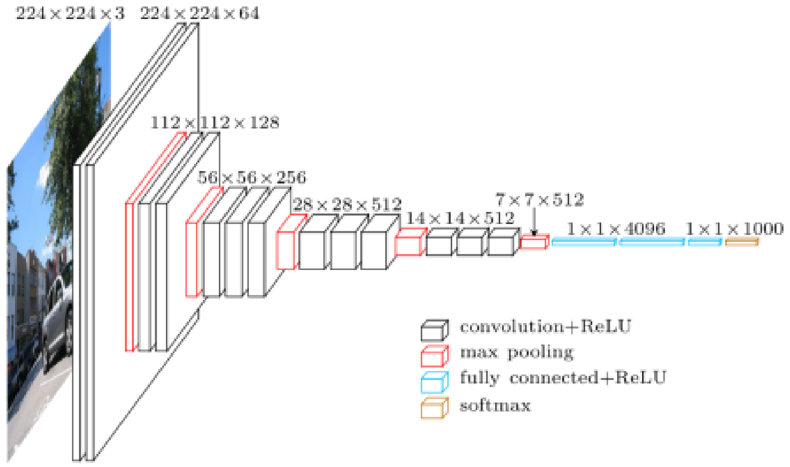
(b) After applying dropout.

Source: [Srivastava et al., 2014]

- Zero every neuron with probability $1 - p$
- At test time, multiply every neuron by p

VGG (2014) [Simonyan-Zisserman'14]

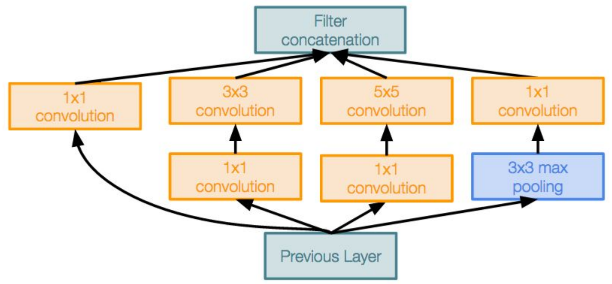
- Deeper than AlexNet: 11-19 layers versus 8
- No local response normalization
- Number of filters multiplied by two every few layers
- Spatial extent of filters 3×3 in all layers
- Instead of 7×7 filters, use three layers of 3×3 filters
  - Gain intermediate nonlinearity
  - Impose a regularization on the 7×7 filters



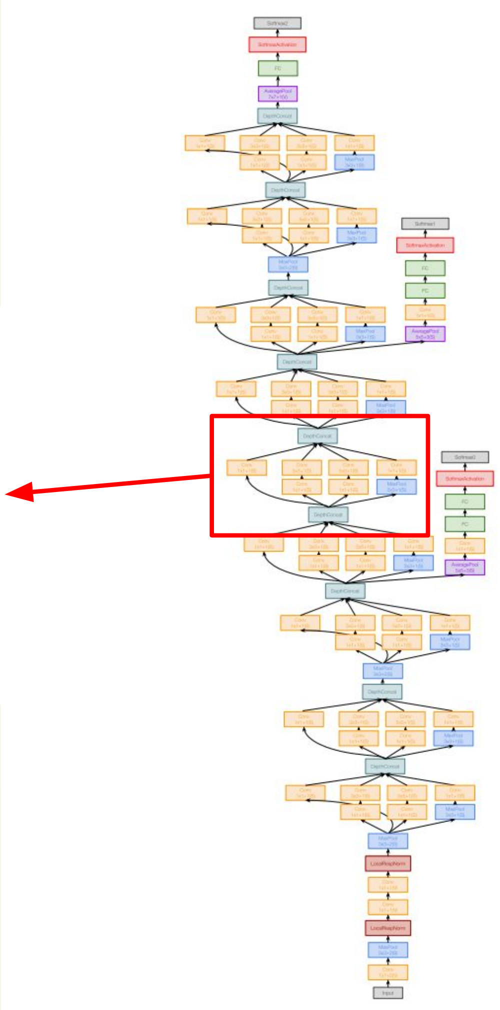
Source: `https://blog.heuritech.com/2016/02/29/`

GoogLeNet *[Szegedy et al., 2014]*

- 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)



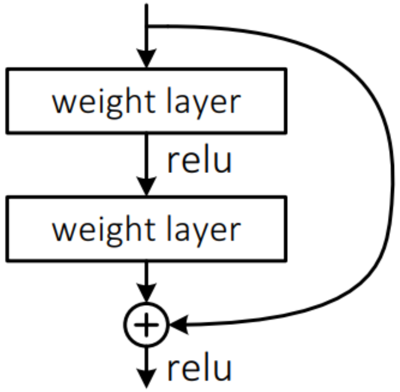
Inception module

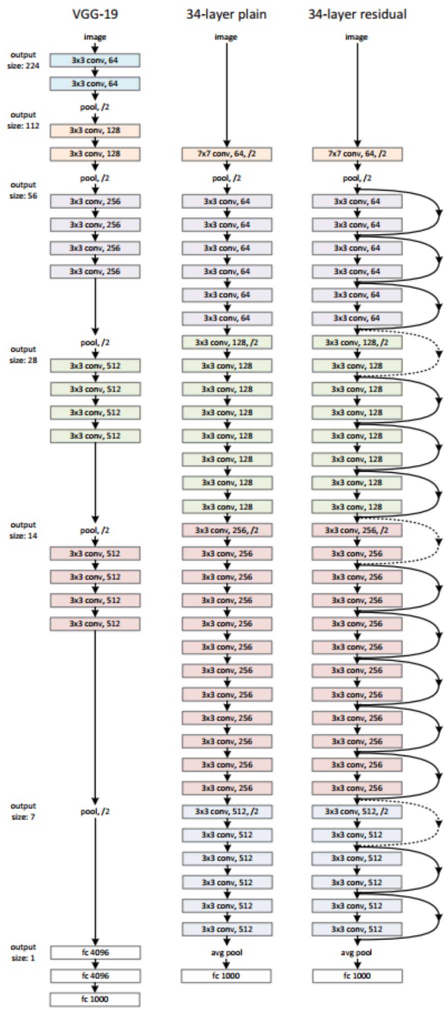


ResNet (2015) [HGRS-15]

ILSVRC'15 classification winner (3.57% top 5 error)

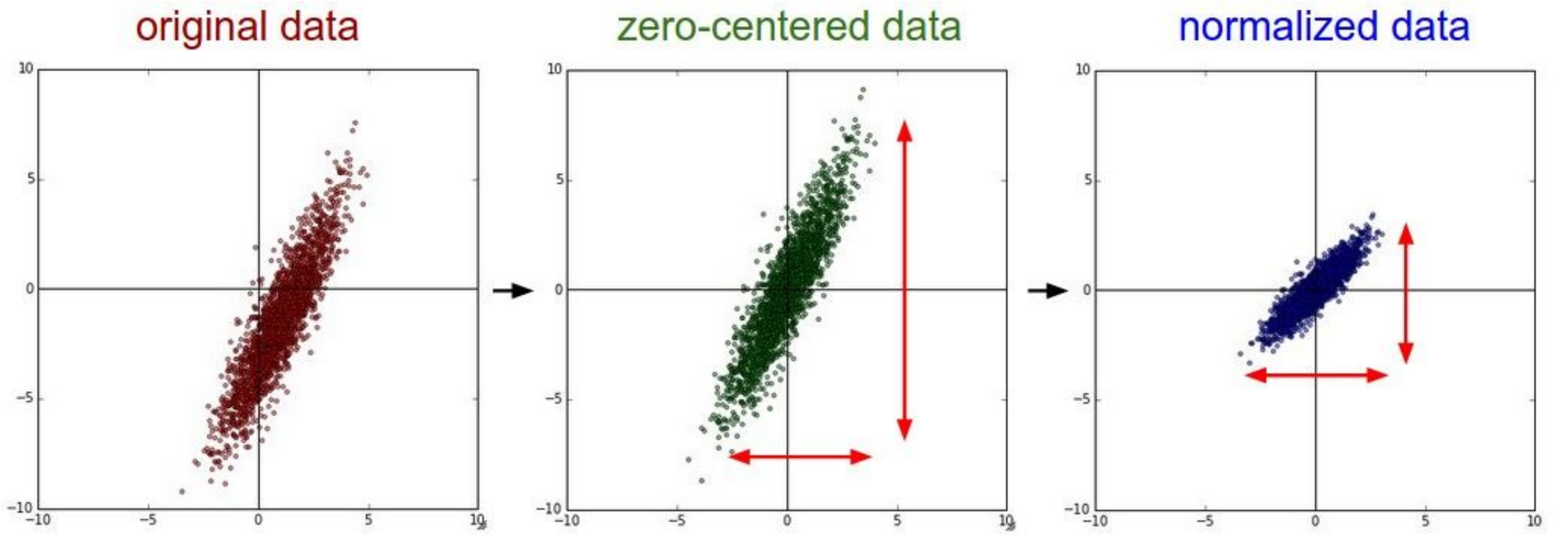
- Solves problem by adding skip connections
- Very deep: 152 layers
- No dropout
- Stride
- Batch normalization





Source: Deep Residual Learning for Image Recognition

Batch Normalization




```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

(Assume X [NxD] is data matrix,
each example in a row)

Batch Normalization

Algorithm 2 Batch normalization [Ioffe and Szegedy, 2015]

Input: Values of x over minibatch $x_1 \ldots x_B$, where x is a certain channel in a certain feature vector

Output: Normalized, scaled and shifted values $y_1 \ldots y_B$

1: $\mu = \frac{1}{B} \sum_{b=1}^{B} x_b$

2: $\sigma^2 = \frac{1}{B} \sum_{b=1}^{B} (x_b - \mu)^2$

3: $\hat{x}_b = \frac{x_b - \mu}{\sqrt{\sigma^2 + \epsilon}}$

4: $y_b = \gamma \hat{x}_b + \beta$

- Accelerates training and makes initialization less sensitive
- Zero mean and unit variance feature vectors

BatchNorm at Test

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \qquad \text{// normalize}$$

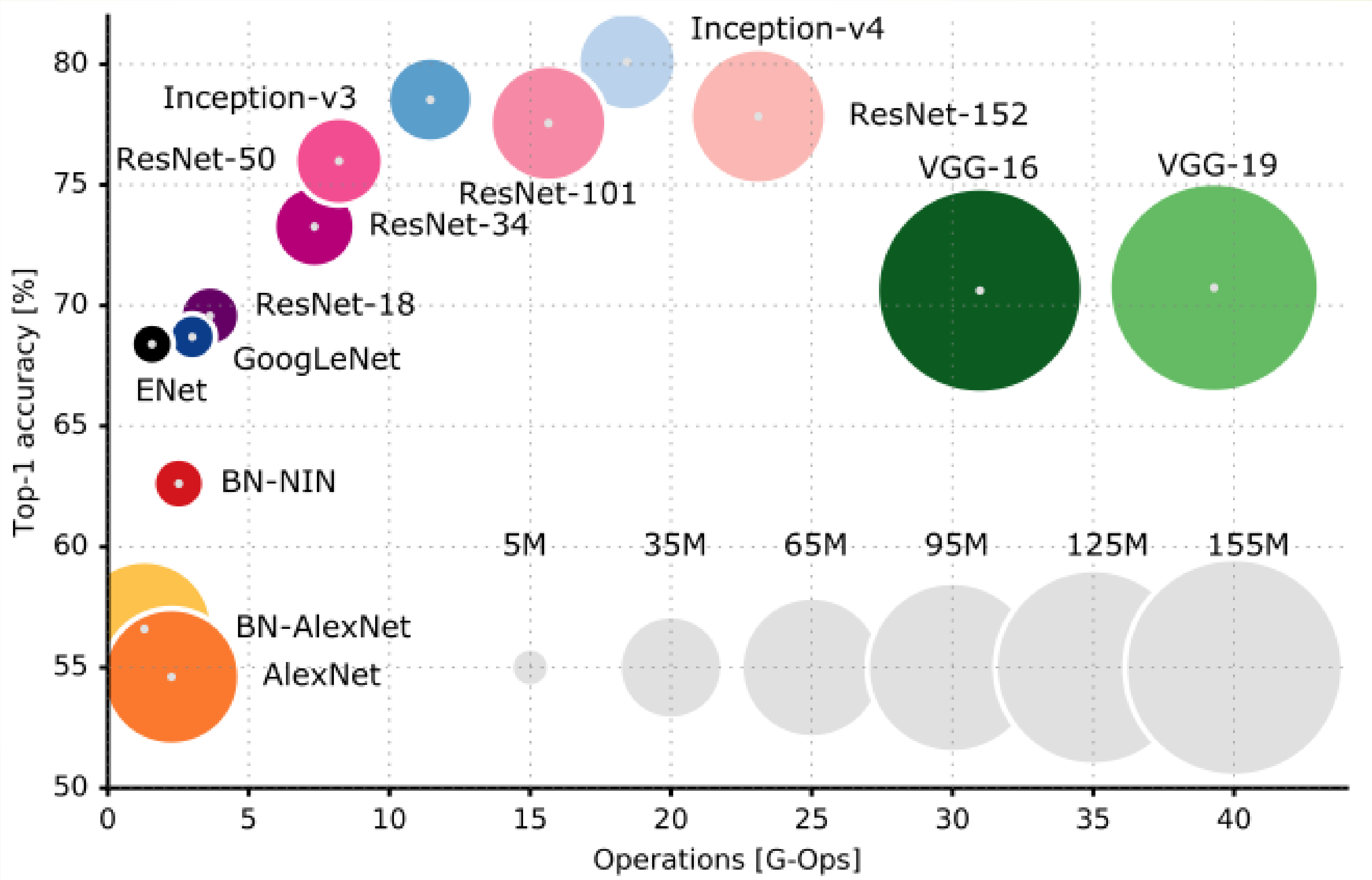
$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Note: at test time BatchNorm layer functions differently:

The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.

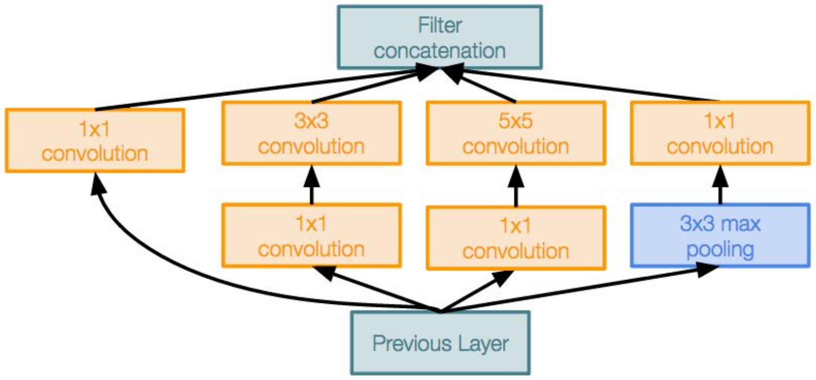
(e.g. can be estimated during training with running averages)

Complexity vs. Accuracy of Different Networks

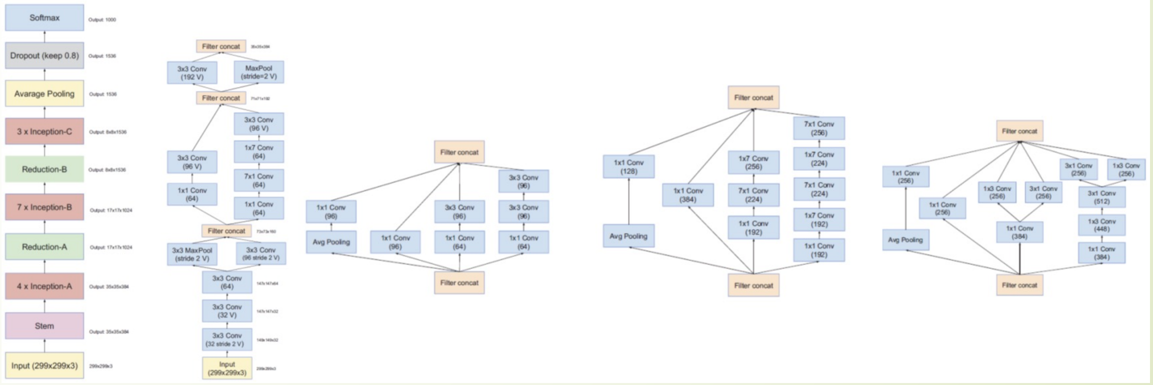


Inception-v4 = ResNet + Inception

- "**Inception**" module:
  - Introduced by *Szegedy et al., 2014* in ***GoogLeNet***
  - ILSVRC'14 classification winner (6.7% top 5 error)
- Apply parallel filter operations on the input from previous layer:
  - Dimensionality reduction (1x1 conv)
  - Multiple receptive field sizes for convolution (1x1, 3x3, 5x5)
  - Pooling operation (3x3)
- Concatenate all filter outputs together depth-wise

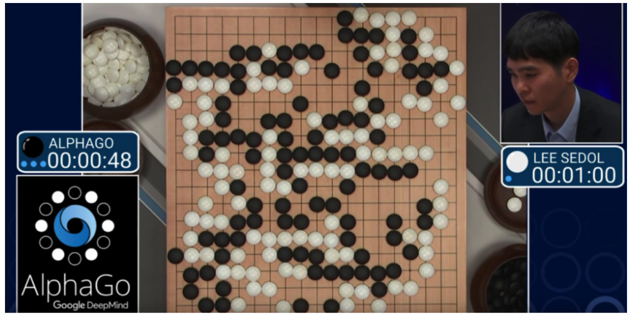


Inception module

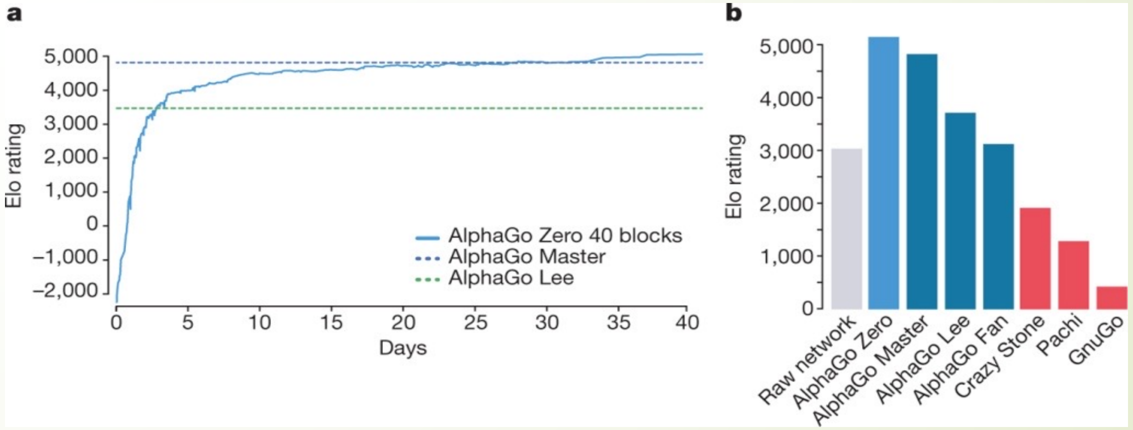


Reaching Human Performance Level in Games

Deep Blue in 1997



AlphaGo “LEE” 2016: Monte-Carlo Tree Pruning Search+CNN



AlphaGo ”ZERO” D Silver *et al. Nature* **550,** 354–359 (2017) doi:10.1038/nature24270

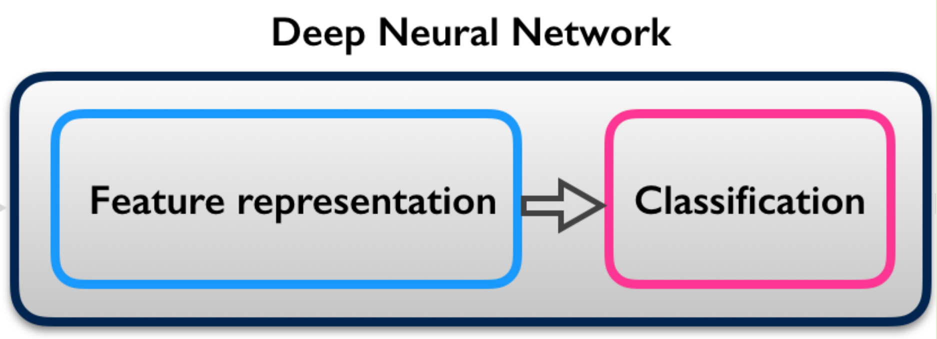
Deep Learning Softwares

- **Pytorch** (developed by Yann LeCun and Facebook):
  - http://pytorch.org/tutorials/
- Tensorflow (developed by Google based on Caffe)
  - https://www.tensorflow.org/tutorials/
- Theano (developed by Yoshua Bengio)
  - http://deeplearning.net/software/theano/tutorial/
- **Keras (based on Tensorflow or Pytorch)**
  - https://www.manning.com/books/deep-learning-with-python?a_aid=keras&a_bid=76564dff

Show some examples by jupyter notebooks

Transfer Learning: Feature Extraction and Fine Tuning

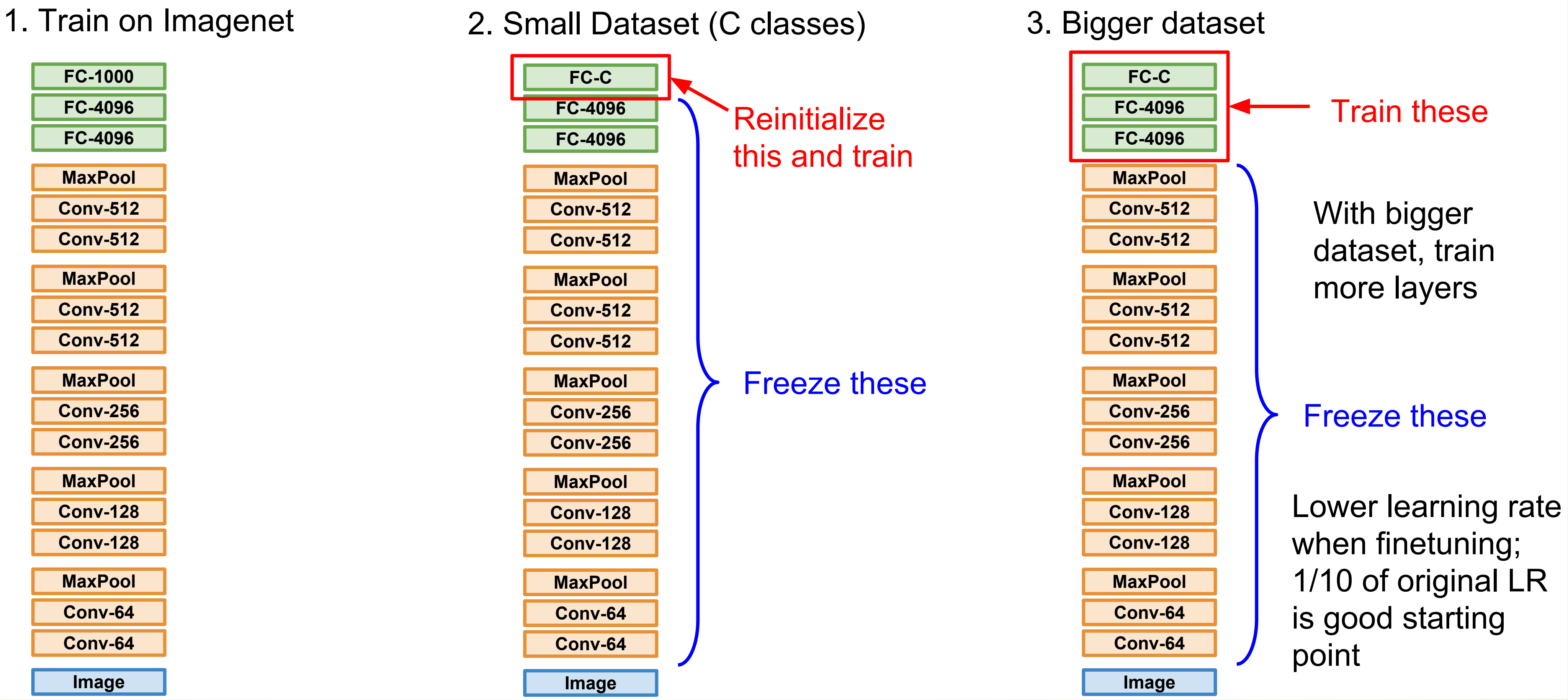
Transfer Learning?

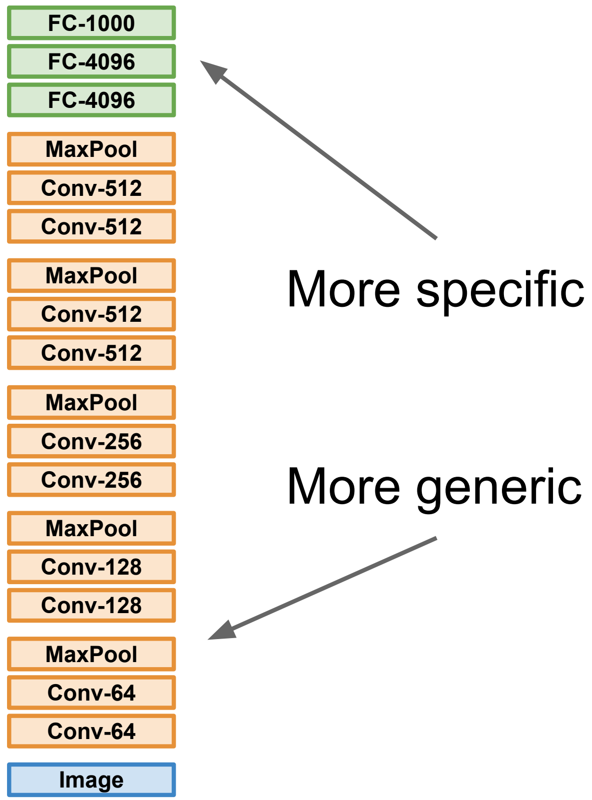


- Filters learned in first layers of a network are transferable from one task to another
- When solving another problem, no need to retrain the lower layers, just fine tune upper ones
- Is this simply due to the large amount of images in ImageNet?
- Does solving many classification problems simultaneously result in features that are more easily transferable?
- Does this imply filters can be learned in unsupervised manner?
- Can we characterize filters mathematically?

Transfer Learning with CNNs

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





	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

Summary

- Feature Extraction vs. Fine-Tuning:
  - Feature extraction usually refers to freeze the bottom (early layers) and retrain the top (last) layer
  - Fine-Tuning usually refers to retrain the last few layers or the whole network ninialized from pretrained parameters
  - They are both called transfer learning

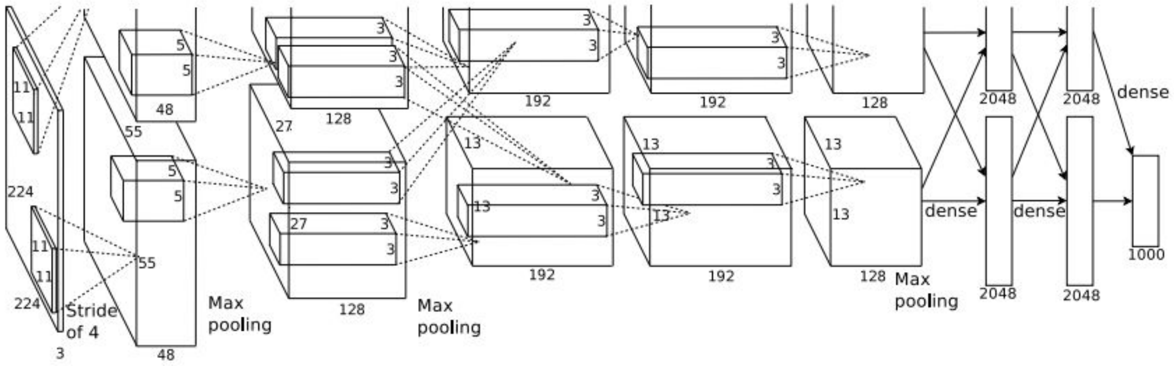
- Jupyter notebook examples with pytorch:
  - https://github.com/aifin-hkust/aifin-hkust.github.io/blob/master/2020/notebook/finetuning_resnet.ipynb

Visualizing Convolutional Networks

Understanding intermediate neurons?

This image is CC0 public domain

Input Image:
3 x 224 x 224



Class Scores:
1000 numbers

What are the intermediate features looking for?

Visualizing CNN Features: Gradient Ascent

- **Gradient ascent:** Generate a synthetic image that maximally activates a neuron

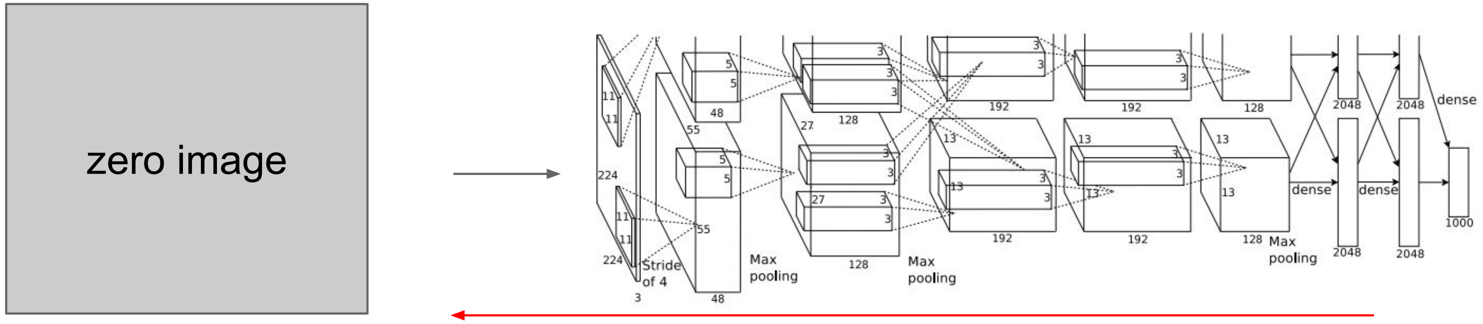
$$I^* = \arg\max_I \underbrace{\boxed{f(I)}}_{\text{Neuron value}} + \underbrace{\boxed{R(I)}}_{\text{Natural image regularizer}}$$

Visualizing CNN Features: Gradient Ascent

$$\arg\max_I \boxed{S_c(I)} - \lambda \|I\|_2^2$$

score for class c (before Softmax)

1. Initialize image to zeros



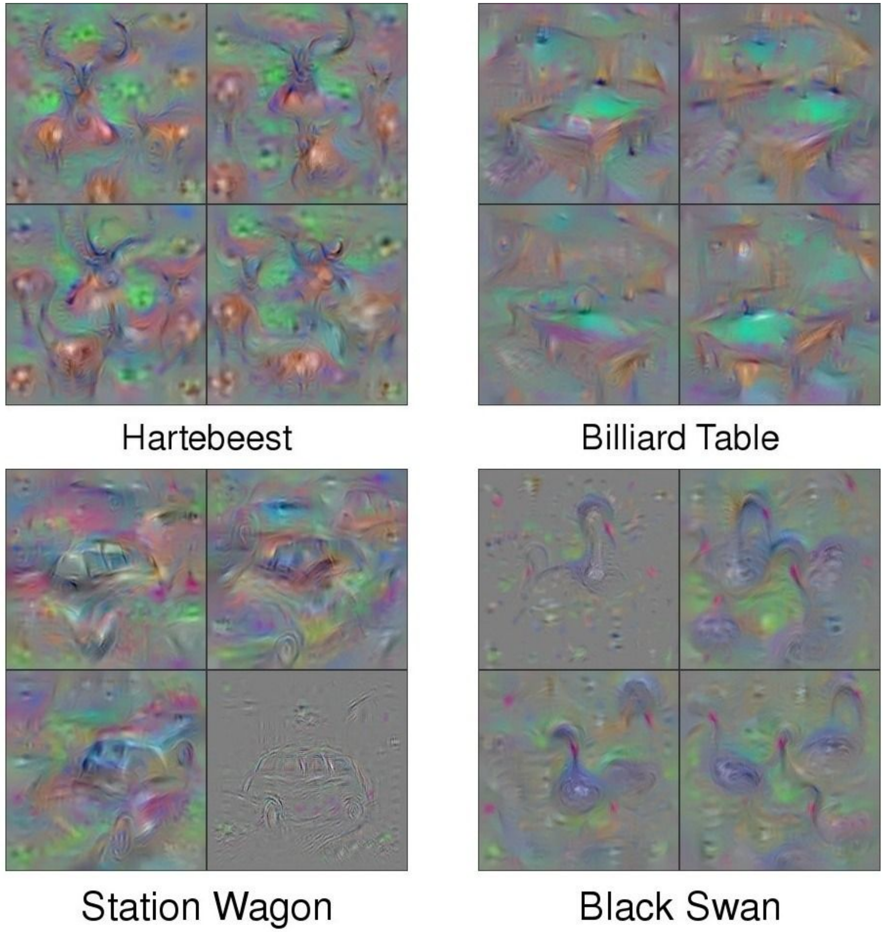
Repeat:
2. Forward image to compute current scores
3. Backprop to get gradient of neuron value with respect to image pixels
4. Make a small update to the image

Visualizing CNN Features: Gradient Ascent

$$\arg\max_I S_c(I) - \lambda \|I\|_2^2$$

Better regularizer: Penalize L2 norm of image; also during optimization periodically

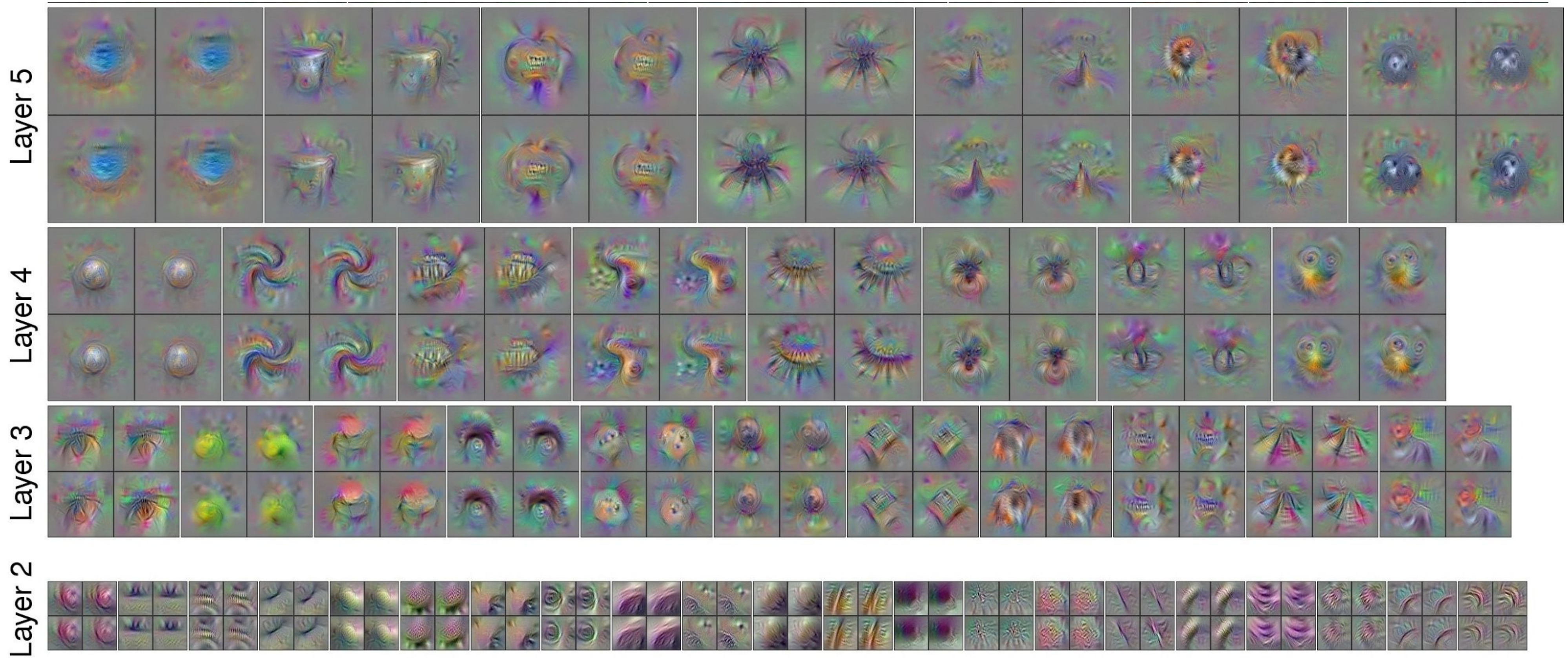
(1) Gaussian blur image
(2) Clip pixels with small values to 0
(3) Clip pixels with small gradients to 0



Yosinski et al, “Understanding Neural Networks Through Deep Visualization”, ICML DL Workshop 2014.

Visualizing CNN Features: Gradient Ascent

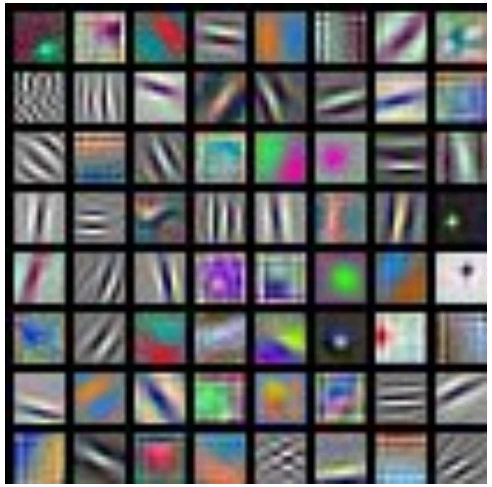
Use the same approach to visualize intermediate features



Yosinski et al, "Understanding Neural Networks Through Deep Visualization", ICML DL Workshop 2014.

It’s easy to visualize early layers

First Layer: Visualize Filters

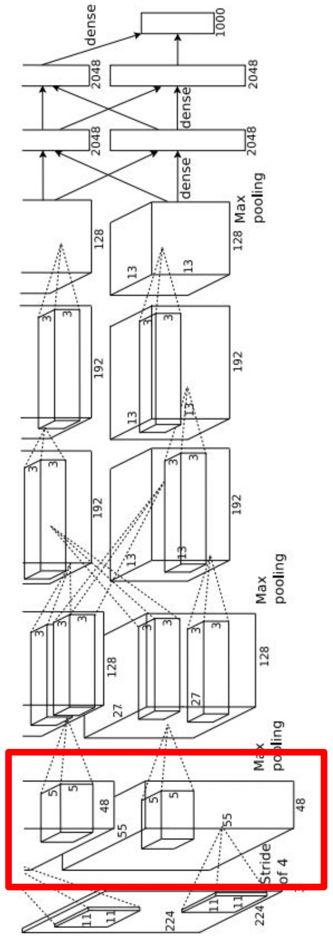


AlexNet:
64 x 3 x 11 x 11

ResNet-18:
64 x 3 x 7 x 7

ResNet-101:
64 x 3 x 7 x 7

DenseNet-121:
64 x 3 x 7 x 7



Krizhevsky, “One weird trick for parallelizing convolutional neural networks”, arXiv 2014
He et al, “Deep Residual Learning for Image Recognition”, CVPR 2016
Huang et al, “Densely Connected Convolutional Networks”, CVPR 2017

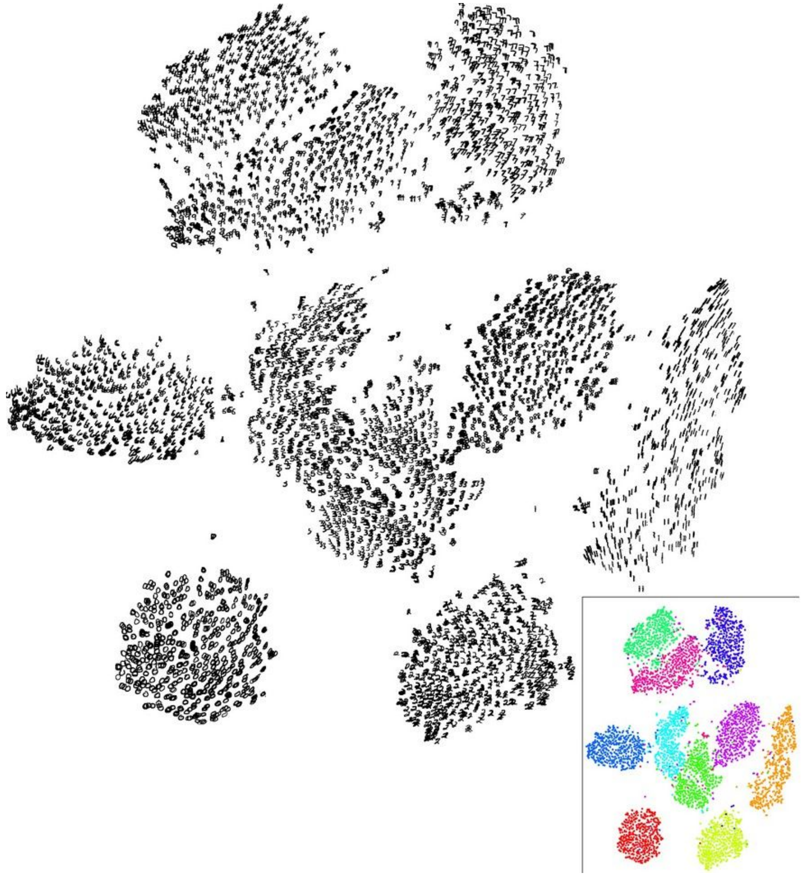
Last layers are hard to visualize

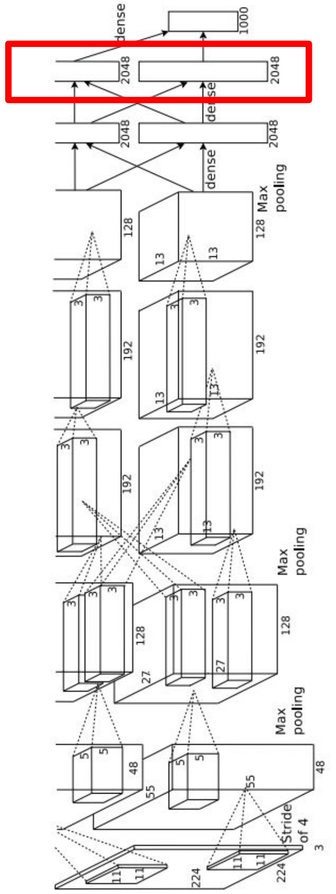
Last Layer: Dimensionality Reduction

Visualize the “space” of FC7 feature vectors by reducing dimensionality of vectors from 4096 to 2 dimensions

Simple algorithm: Principle Component Analysis (PCA)

More complex: **t-SNE**

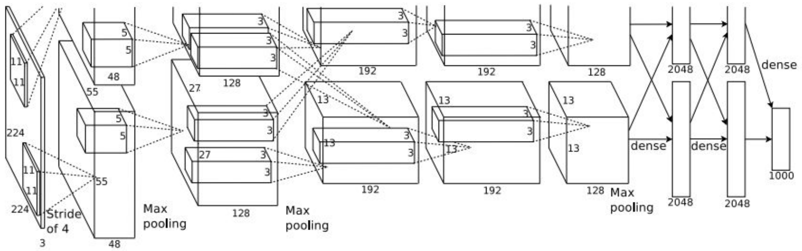




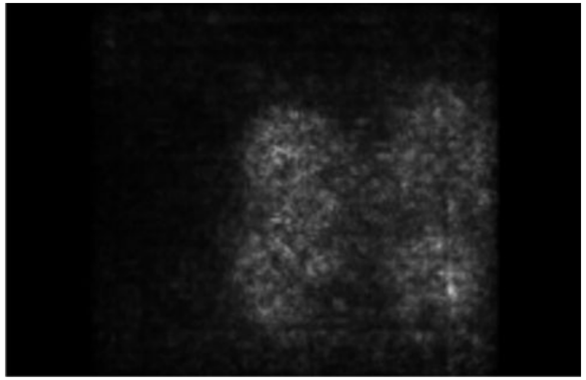
Van der Maaten and Hinton, “Visualizing Data using t-SNE”, JMLR 2008

Saliency Maps

How to tell which pixels matter for classification?



Dog

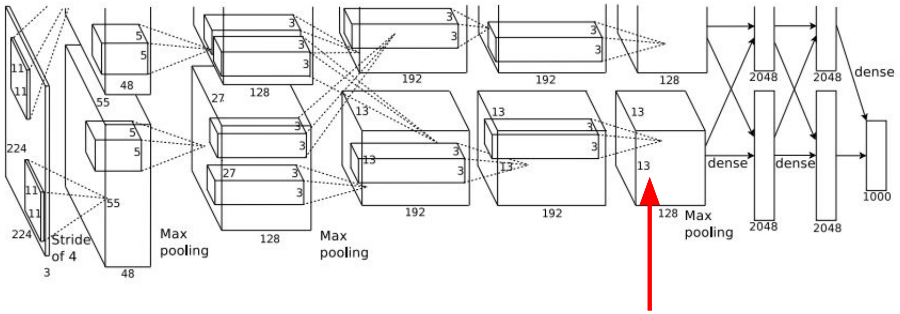


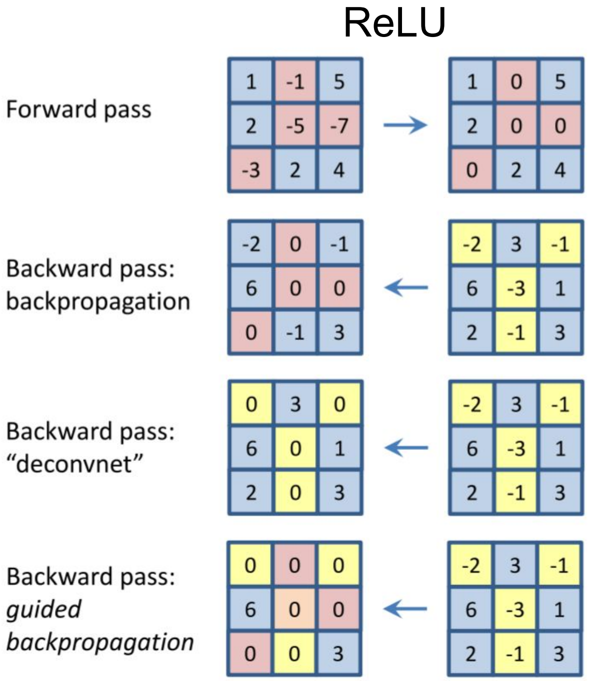
Compute gradient of (unnormalized) class score with respect to image pixels, take absolute value and max over RGB channels

Simonyan, Vedaldi, and Zisserman, "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR Workshop 2014.

Guided BP

Intermediate features via (guided) backprop





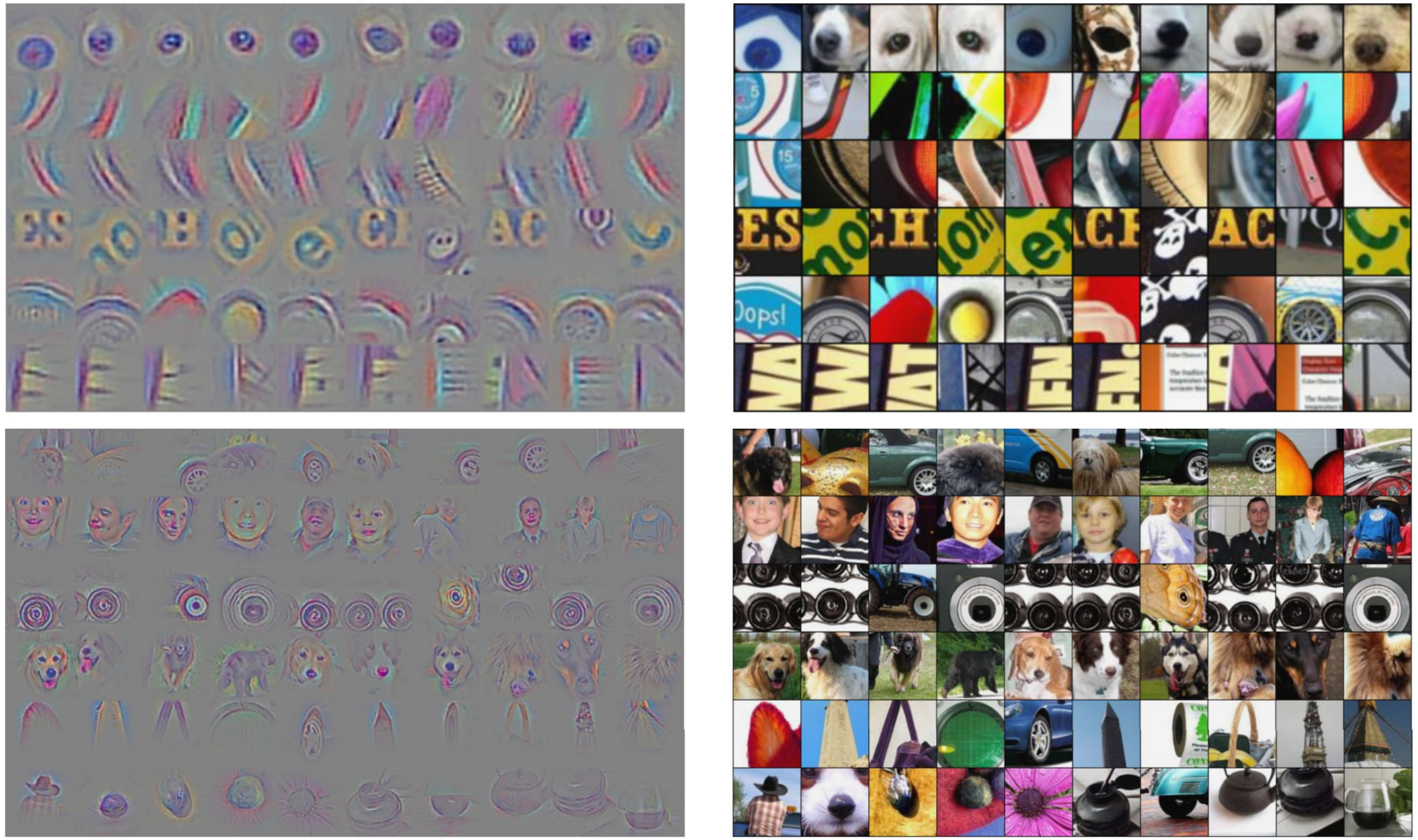
Pick a single intermediate neuron, e.g. one value in 128 x 13 x 13 conv5 feature map

Compute gradient of neuron value with respect to image pixels

Images come out nicer if you only backprop positive gradients through each ReLU (guided backprop)

Zeiler and Fergus, "Visualizing and Understanding Convolutional Networks", ECCV 2014
Springenberg et al, "Striving for Simplicity: The All Convolutional Net", ICLR Workshop 2015

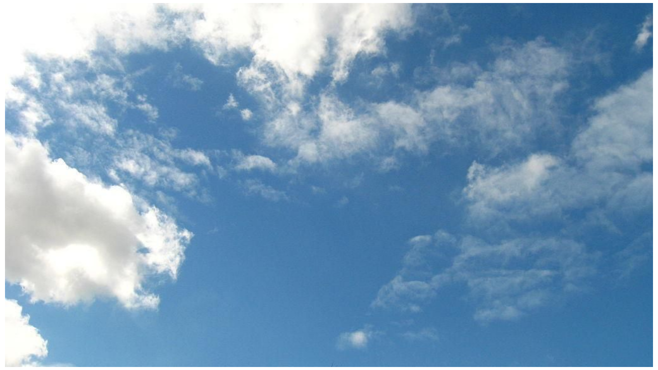
Intermediate features via Guided BP

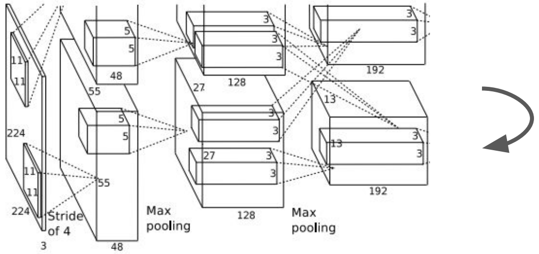


Zeiler and Fergus, “Visualizing and Understanding Convolutional Networks”, ECCV 2014
Springenberg et al, “Striving for Simplicity: The All Convolutional Net”, ICLR Workshop 2015

DeepDream: amplifying features

Rather than synthesizing an image to maximize a specific neuron, instead try to **amplify** the neuron activations at some layer in the network





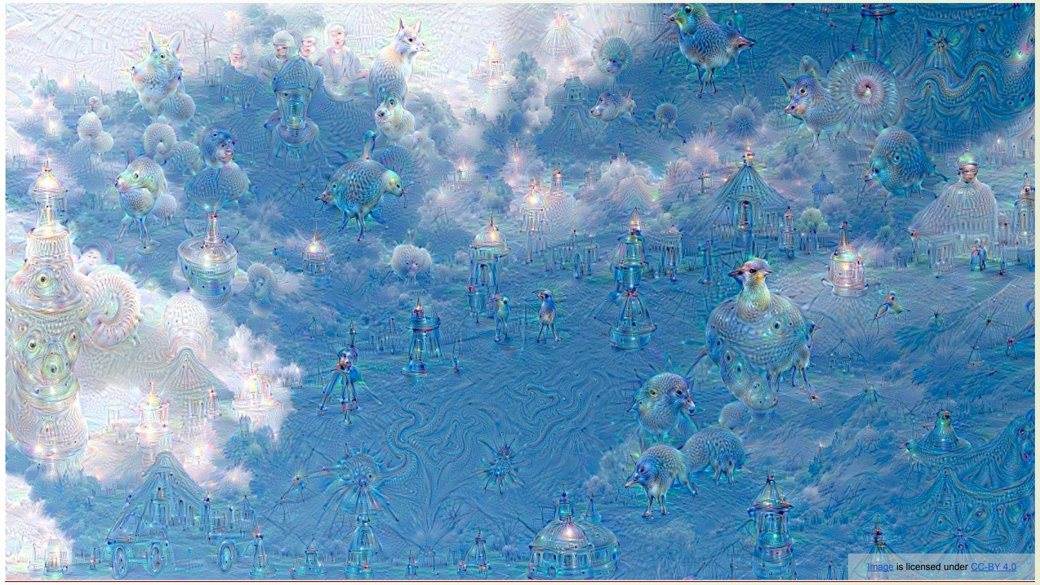
Choose an image and a layer in a CNN; repeat:

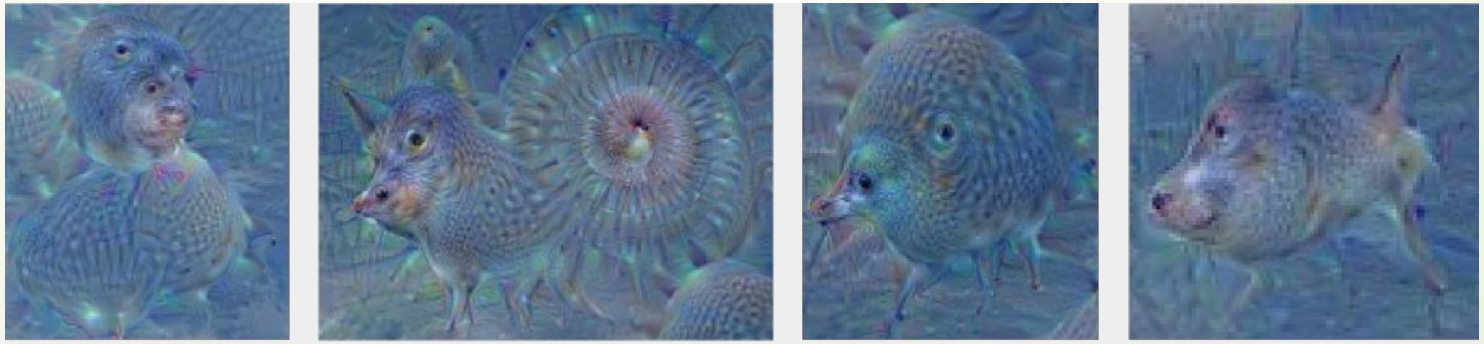
1. Forward: compute activations at chosen layer
2. Set gradient of chosen layer *equal to its activation*
3. Backward: Compute gradient on image
4. Update image

Equivalent to:

$$I^* = \arg\max_I \sum_i f_i(I)^2$$

Example: DeepDream of Sky





"Admiral Dog!" "The Pig-Snail" "The Camel-Bird" "The Dog-Fish"

More Examples

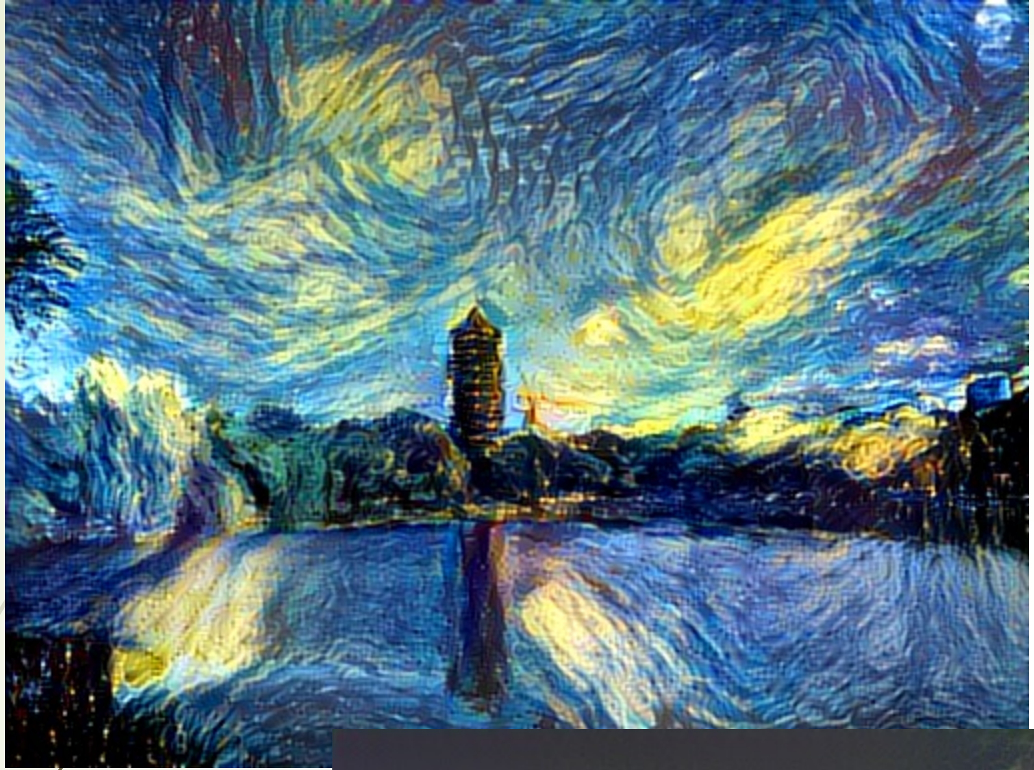
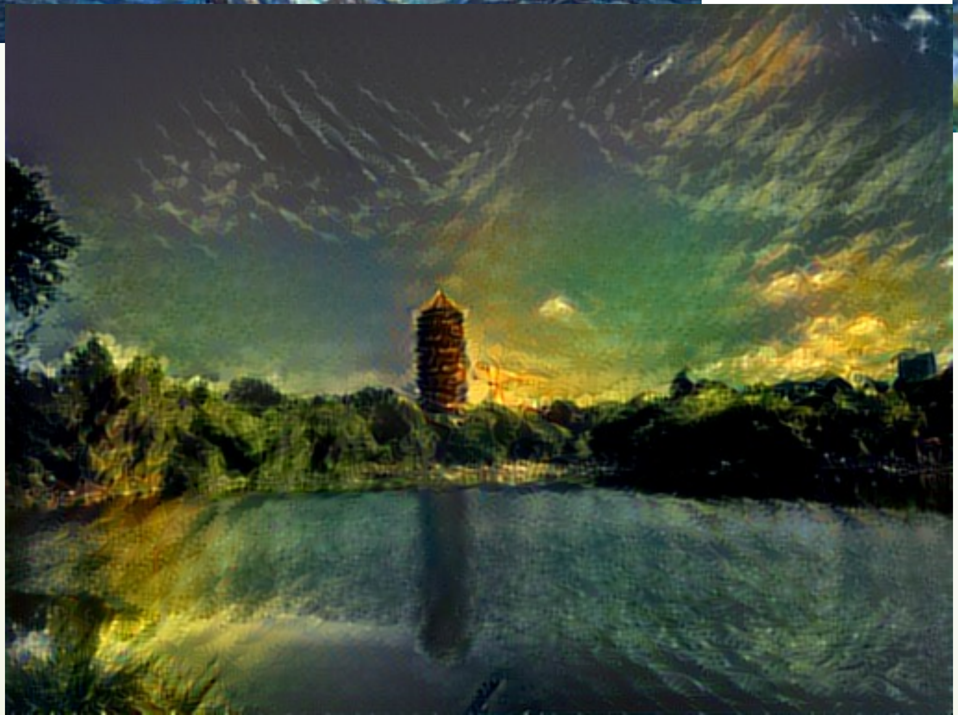
Python Notebooks

- An interesting Pytorch Implementation of these visualizatoin methods
  - https://github.com/utkuozbulak/pytorch-cnn-visualizations
- Some examples demo:
  - https://github.com/aifin-hkust/aifin-hkust.github.io/blob/master/2020/notebook/vgg16-visualization.ipynb
  - https://github.com/aifin-hkust/aifin-hkust.github.io/blob/master/2020/notebook/vgg16-heatmap.ipynb

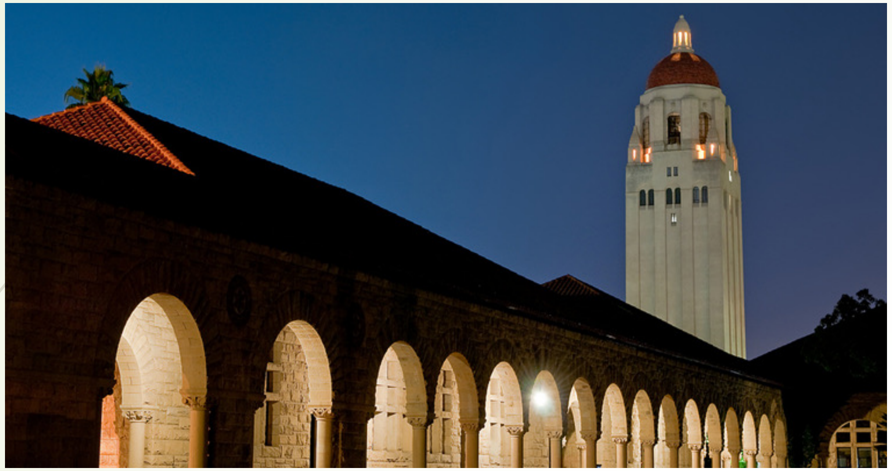
Neural Style

Example: The Noname Lake in PKU

Left: *Vincent Van Gogh*, Starry Night
Right: *Claude Monet*, Twilight Venice
Bottom: *William Turner*, Ship Wreck

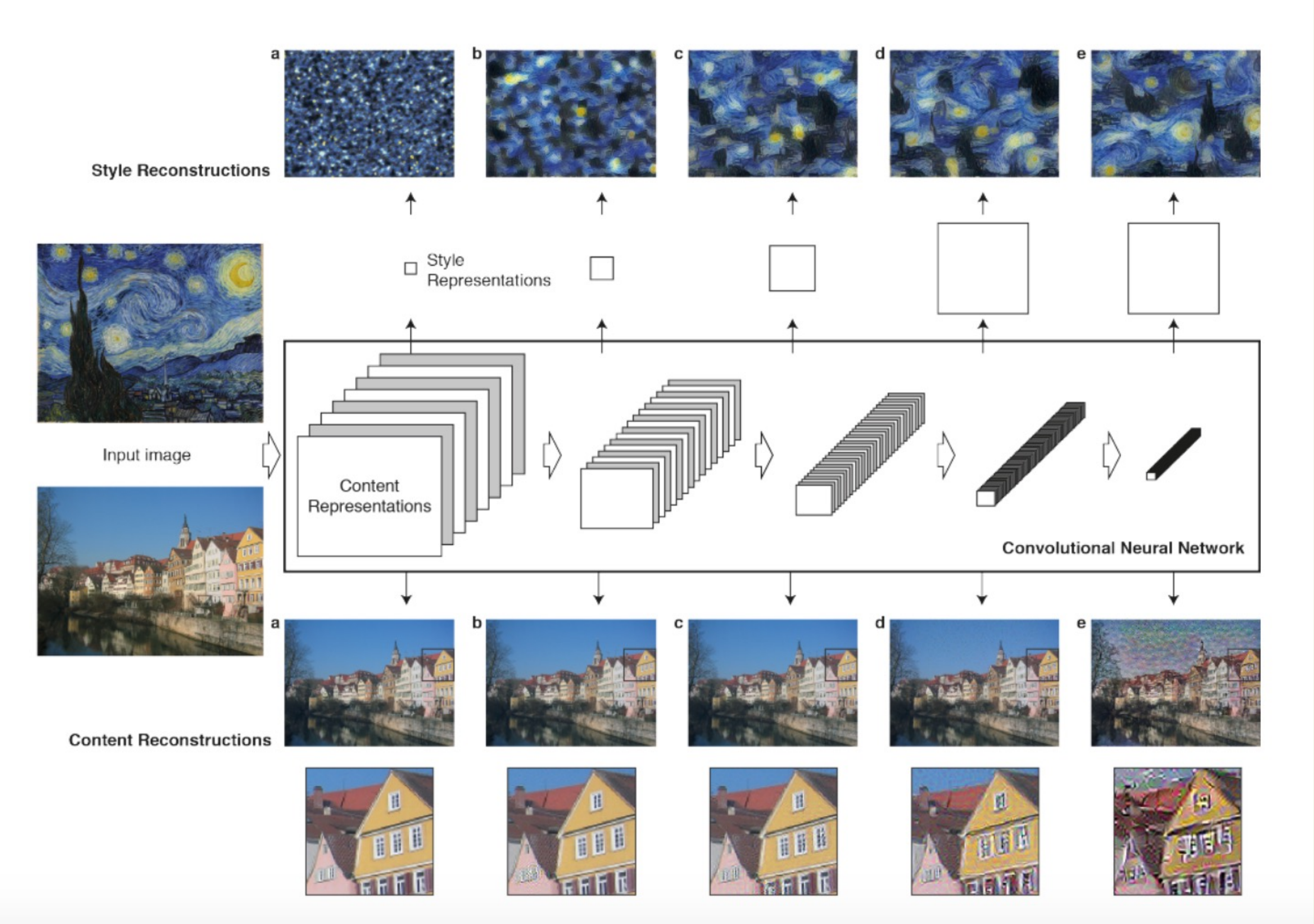

Application of Deep Learning:
Content-Style synthetic pictures
By “neural-style”



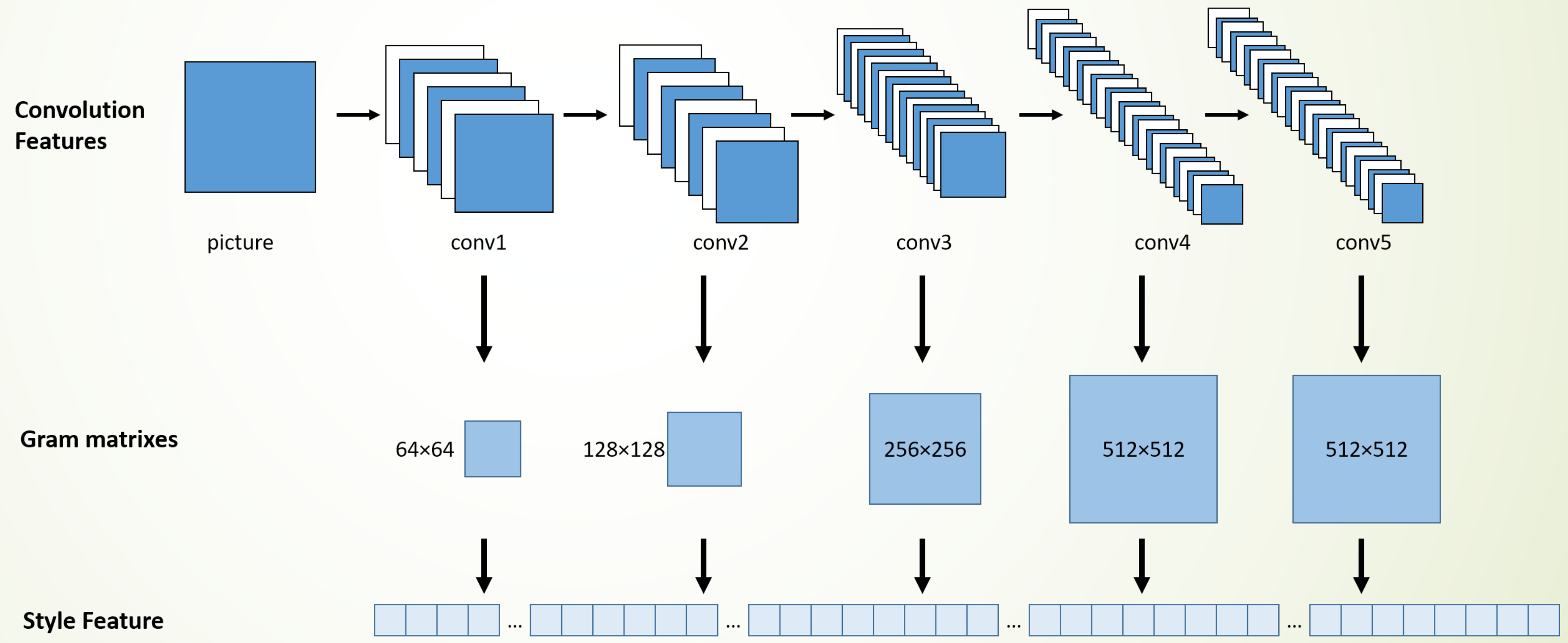

Neural Style

- J C Johnson's Website: https://github.com/jcjohnson/neural-style
- A torch implementation of the paper
  - *A Neural Algorithm of Artistic Style*,
  - by Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge.
  - http://arxiv.org/abs/1508.06576

Style-Content Feature Extraction

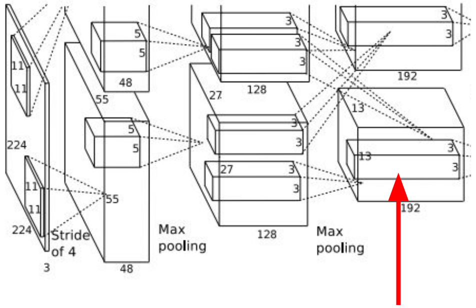


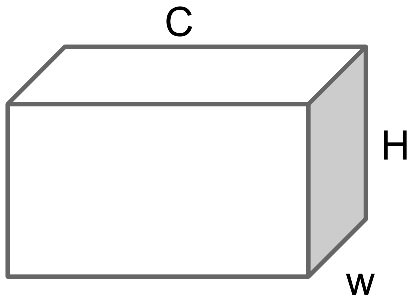
Style Features as Second Order Statistics

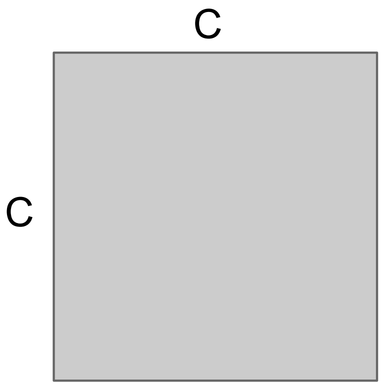


Neural Texture Synthesis

This image is in the public domain.







Each layer of CNN gives C x H x W tensor of features; H x W grid of C-dimensional vectors

Outer product of two C-dimensional vectors gives C x C matrix measuring co-occurrence

Average over all HW pairs of vectors, giving **Gram matrix** of shape C x C

Efficient to compute; reshape features from

C x H x W to =C x HW

then compute G = FF^T

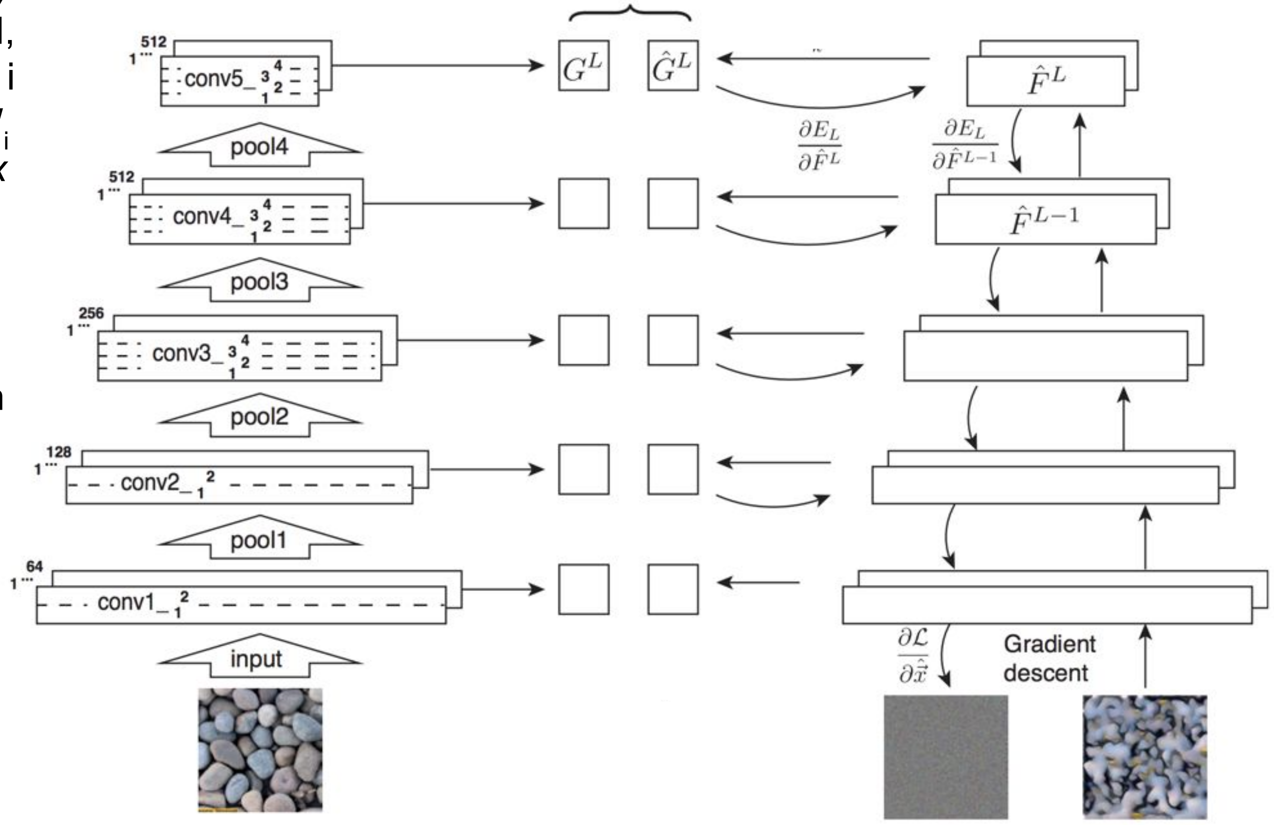
Neural Texture Synthesis

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} \left(G_{ij}^l - \hat{G}_{ij}^l\right)^2 \qquad \mathcal{L}(\vec{x}, \hat{\vec{x}}) = \sum_{l=0}^{L} w_l E_l$$

1. Pretrain a CNN on ImageNet (VGG-19)
2. Run input texture forward through CNN, record activations on every layer; layer i gives feature map of shape $C_i \times H_i \times W_i$
3. At each layer compute the *Gram matrix* giving outer product of features:

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l \text{ (shape } C_i \times C_i\text{)}$$

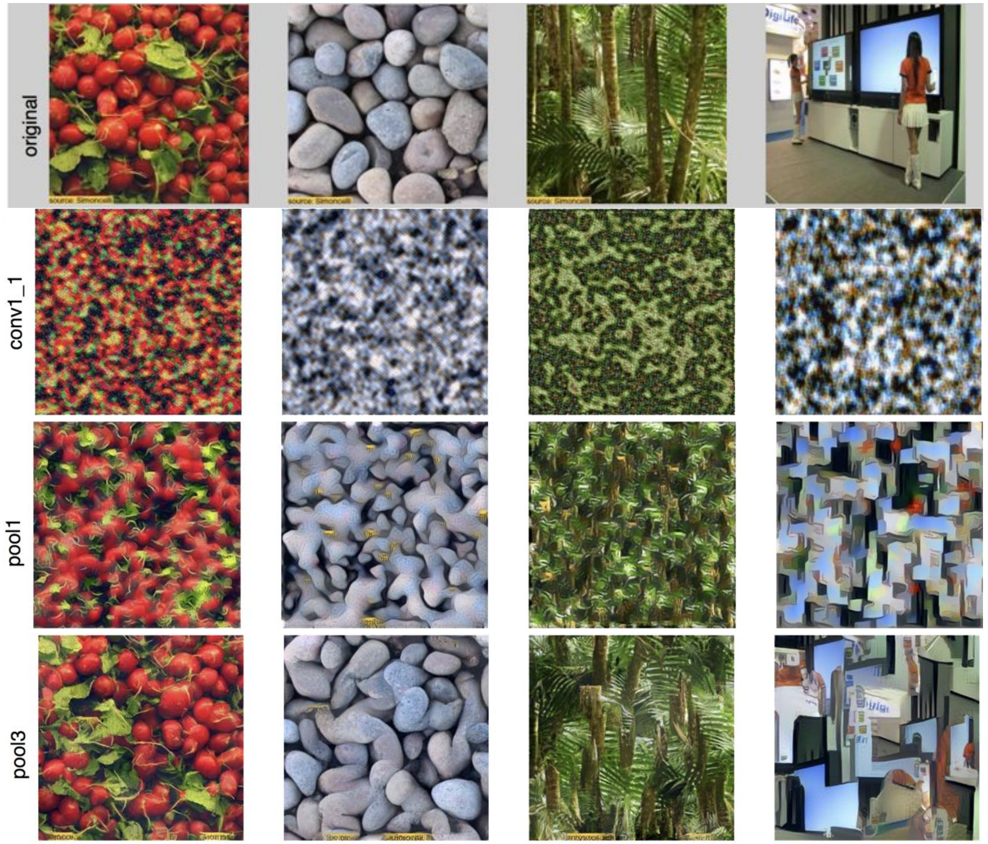
4. Initialize generated image from random noise
5. Pass generated image through CNN, compute Gram matrix on each layer
6. Compute loss: weighted sum of L2 distance between Gram matrices
7. Backprop to get gradient on image
8. Make gradient step on image
9. GOTO 5



Gatys, Ecker, and Bethge, "Texture Synthesis Using Convolutional Neural Networks", NIPS 2015

Neural Texture Synthesis

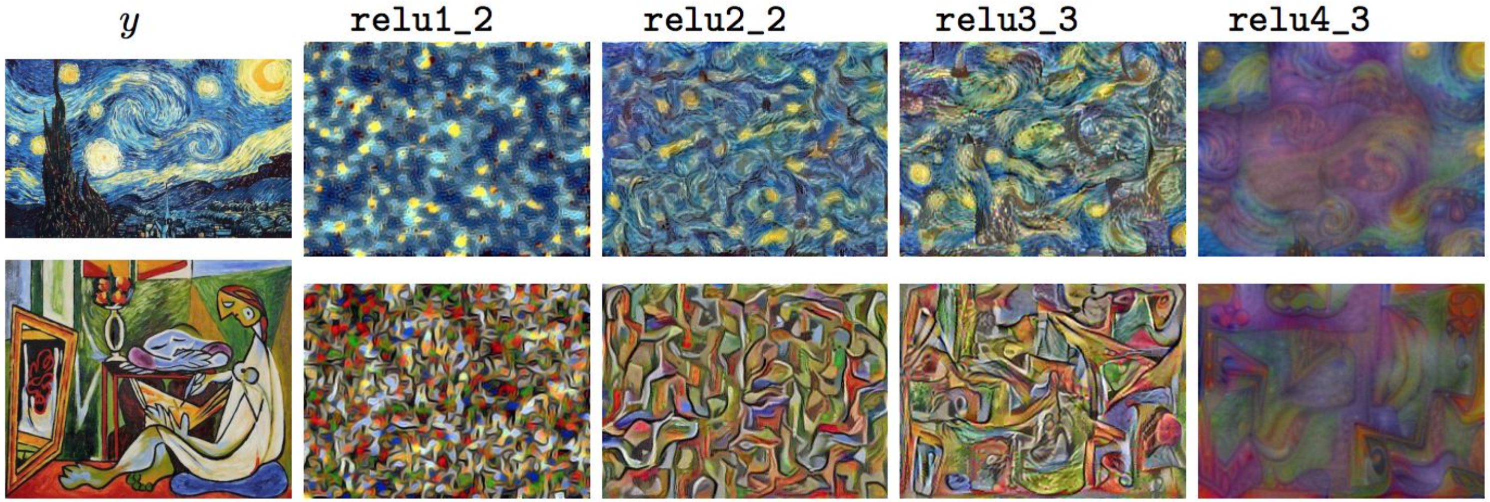
Reconstructing texture from higher layers recovers larger features from the input texture



Gatys, Ecker, and Bethge, “Texture Synthesis Using Convolutional Neural Networks”, NIPS 2015

Neural Texture Synthesis: Gram Reconstruction

Texture synthesis (Gram reconstruction)



Feature Inversion

Given a CNN feature vector for an image, find a new image that:

- Matches the given feature vector
- “looks natural” (image prior regularization)

$$\mathbf{x}^* = \underset{\mathbf{x}\in\mathbb{R}^{H\times W\times C}}{\operatorname{argmin}} \ \ell(\Phi(\mathbf{x}), \Phi_0) + \lambda\mathcal{R}(\mathbf{x})$$

Given feature vector

Features of new image

$$\ell(\Phi(\mathbf{x}), \Phi_0) = \|\Phi(\mathbf{x}) - \Phi_0\|^2$$

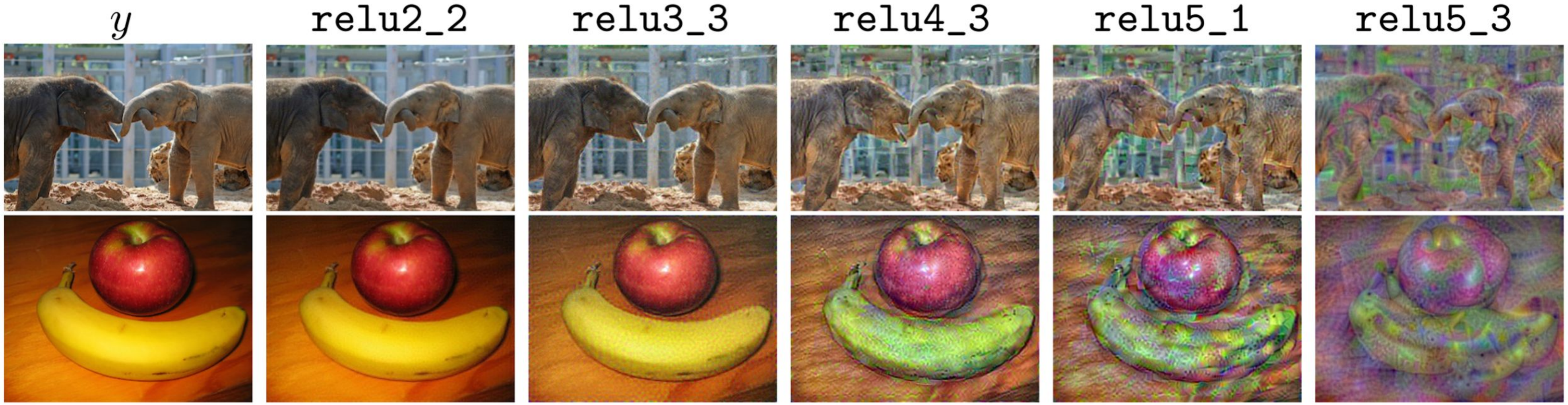
$$\mathcal{R}_{V^\beta}(\mathbf{x}) = \sum_{i,j}\left((x_{i,j+1} - x_{ij})^2 + (x_{i+1,j} - x_{ij})^2\right)^{\frac{\beta}{2}}$$

Total Variation regularizer (encourages spatial smoothness)

Mahendran and Vedaldi, “Understanding Deep Image Representations by Inverting Them”, CVPR 2015

Feature Inversion

Reconstructing from different layers of VGG-16



Mahendran and Vedaldi, “Understanding Deep Image Representations by Inverting Them”, CVPR 2015
Figure from Johnson, Alahi, and Fei-Fei, “Perceptual Losses for Real-Time Style Transfer and Super-Resolution”, ECCV 2016. Copyright Springer, 2016.
Reproduced for educational purposes.

Neural Style Transfer: Feature + Gram Reconstruction

Texture synthesis (Gram reconstruction)

Feature reconstruction

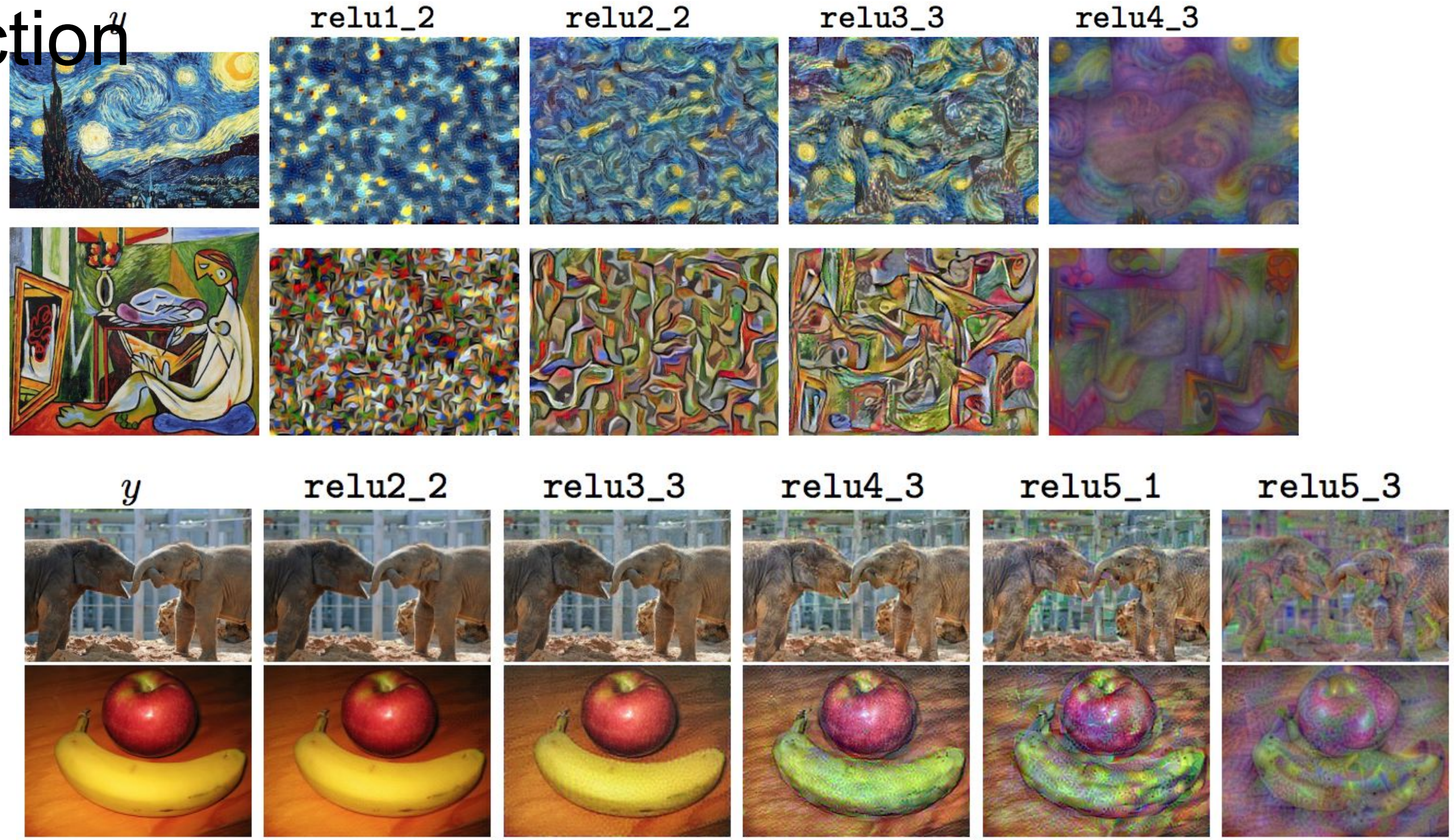


Figure from Johnson, Alahi, and Fei-Fei, “Perceptual Losses for Real-Time Style Transfer and Super-Resolution”, ECCV 2016. Copyright Springer, 2016. Reproduced for educational purposes.

Combined Loss for both Content (1st order statistics) and Style (2nd order statistics: Gram)

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

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

where

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} \left(G_{ij}^l - A_{ij}^l\right)^2 \qquad G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l .$$

Neural Style Transfer

Content Image

This image is licensed under CC-BY 3.0

+

Style Image

Starry Night by Van Gogh is in the public domain

=

Style Transfer!

This image copyright Justin Johnson, 2015. Reproduced with permission.

CNN learns texture features, not shapes!

(a) Texture image		(b) Content image		(c) Texture-shape cue conflict	
81.4%	**Indian elephant**	71.1%	**tabby cat**	63.9%	**Indian elephant**
10.3%	indri	17.3%	grey fox	26.4%	indri
8.2%	black swan	3.3%	Siamese cat	9.6%	black swan

Geirhos et al. ICLR 2019

https://videoken.com/embed/W2HvLBMhCJQ?tocitem=46

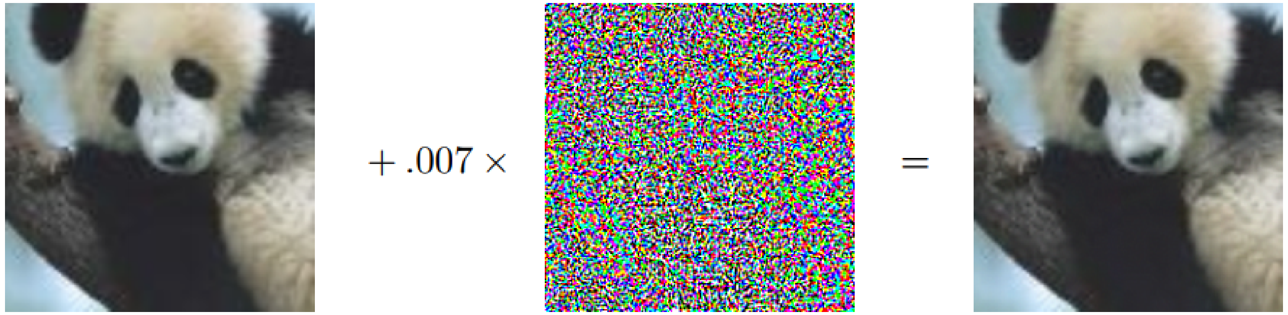
1:16:47

Examples

- Jupyter Notebook Demo

Adversarial Examples and Robustness

Deep Learning may be fragile: adversarial examples



$\boldsymbol{x}$	$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$	$\boldsymbol{x} + \epsilon\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"panda" 57.7% confidence	"nematode" 8.2% confidence	"gibbon" 99.3 % confidence

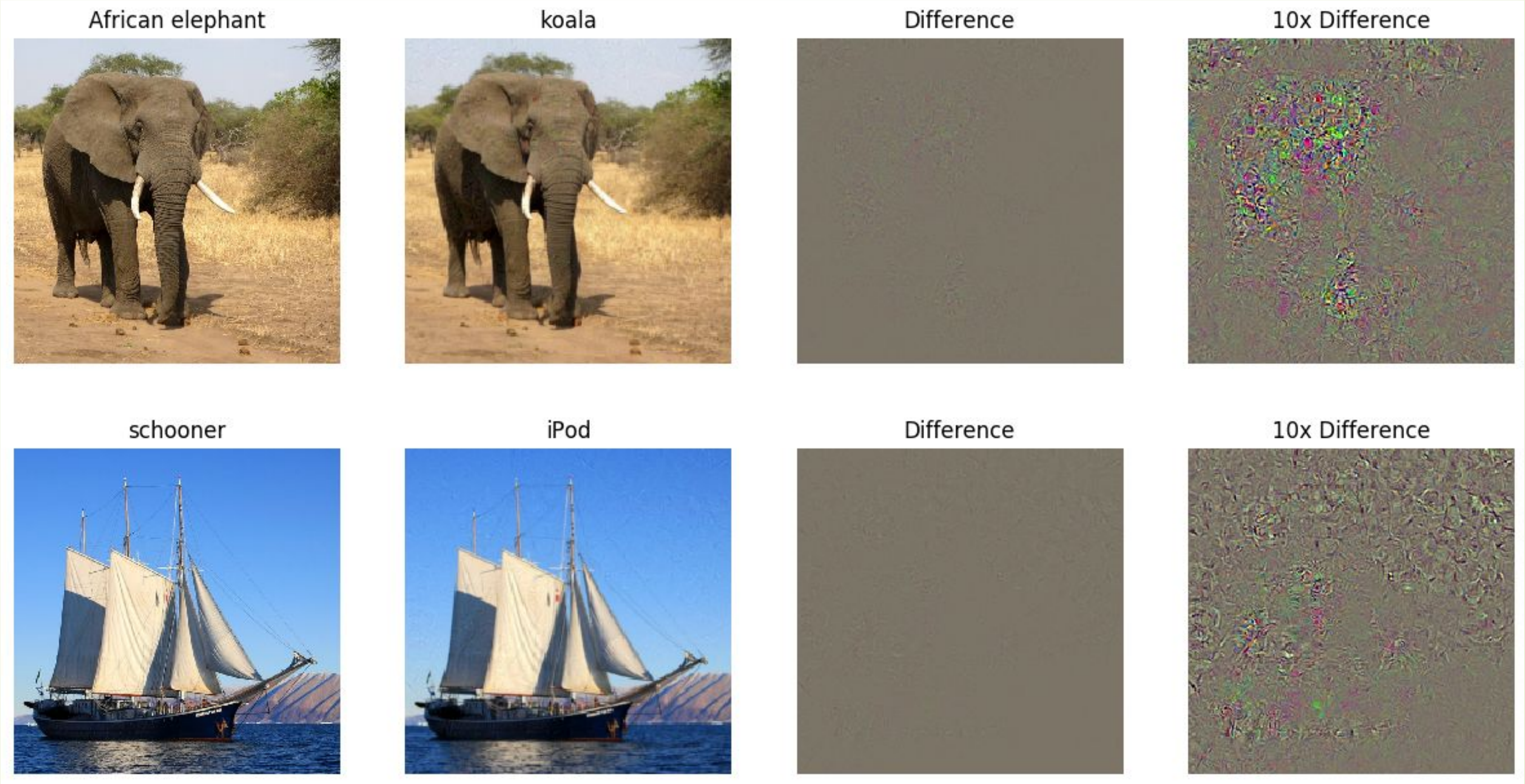
[Goodfellow et al., 2014]

- Small but malicious perturbations can result in severe misclassification
- Malicious examples generalize across different architectures
- What is source of instability?
- Can we robustify network?

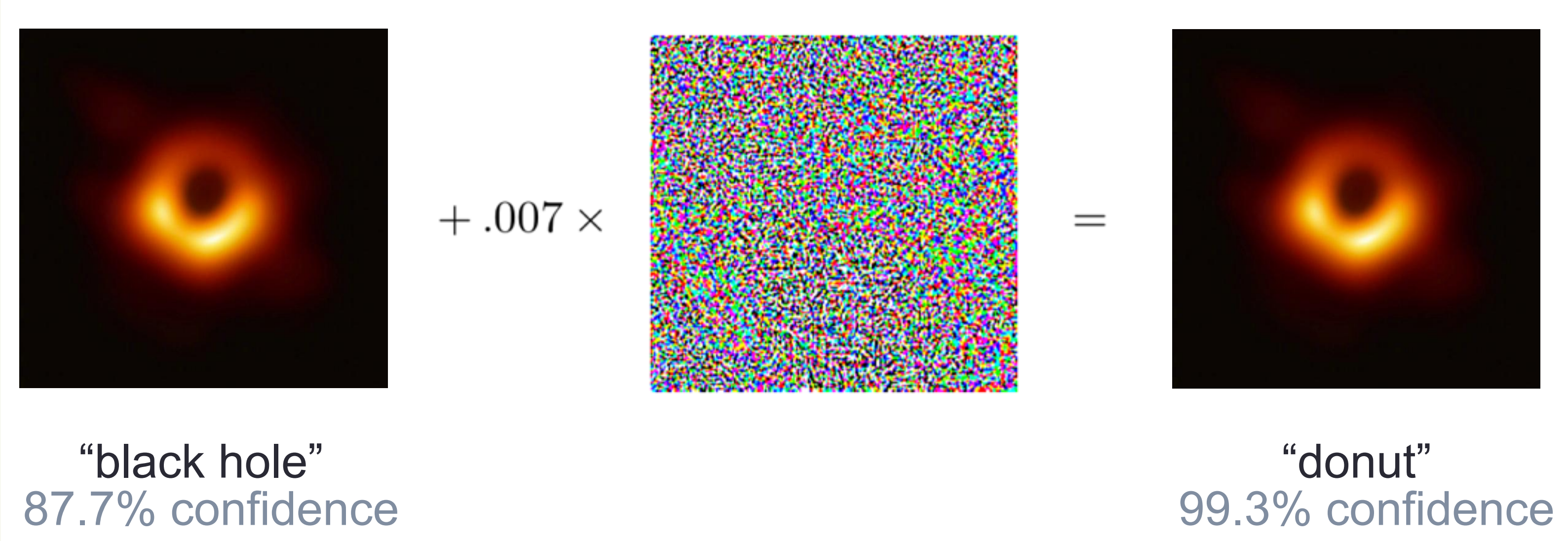
Adversarial Examples: Fooling Images

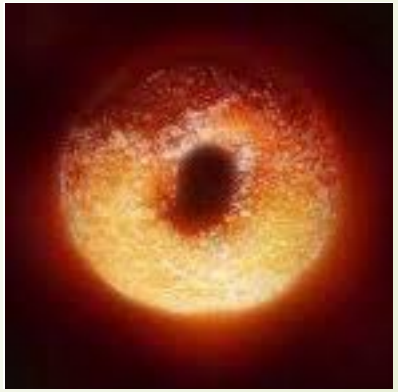
- Start from an arbitrary image
- Pick an arbitrary class
- Modify the image to maximize the class
- Repeat until network is fooled

Fooling Images/Adversarial Examples



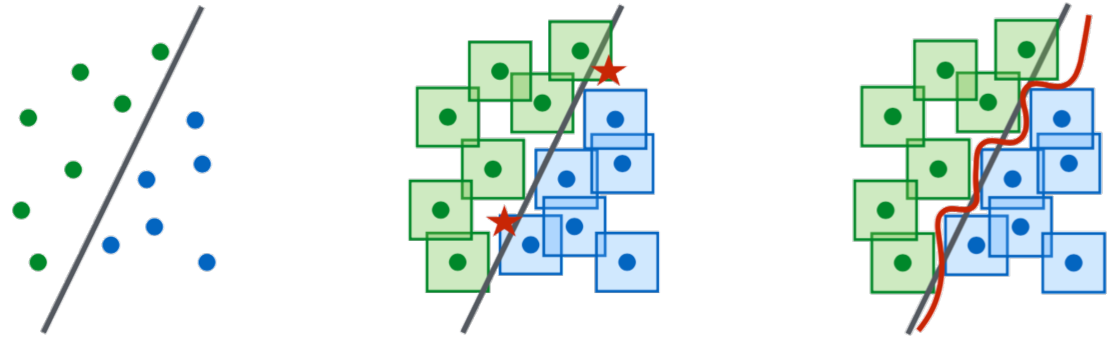
Convolutional Networks lack Robustness





Courtesy of Dr. Hongyang ZHANG.

Adversarial Robust Training



- Traditional training:

$$\min_{\theta} J_n(\theta, \mathbf{z} = (x_i, y_i)_{i=1}^n)$$

  - e.g. square or cross-entropy loss as negative log-likelihood of logit models

- Robust optimization (Madry et al. ICLR'2018):

$$\min_{\theta} \max_{\|\epsilon_i\| \leq \delta} J_n(\theta, \mathbf{z} = (x_i + \epsilon_i, y_i)_{i=1}^n)$$

  - robust to any distributions, yet computationally hard

Extended by **Hongyang ZHANG** et al. by TRADES, 2019.

Introduction to “(Re-)Imag(in)ing Price Trends”

By

Jingwen Jiang
University of Chicago

Bryan Kelly
Yale University, AQR Capital Management, and NBER

Dacheng Xiu
University of Chicago Booth School of Business

Brief Intro

In the empirical designs, they first embed 1D time-series data into 2D images depicting price and volumes.

Then they feed each training sample into CNN to estimate the probability of a positive subsequent return over short(5-day), medium(20-day), and long (60day) horizons.

Afterward, they use CNN-based out-of-sample predictions as signals in several asset pricing analyses.

Finally, they attempt to interpret the predictive patterns identified by the CNN.

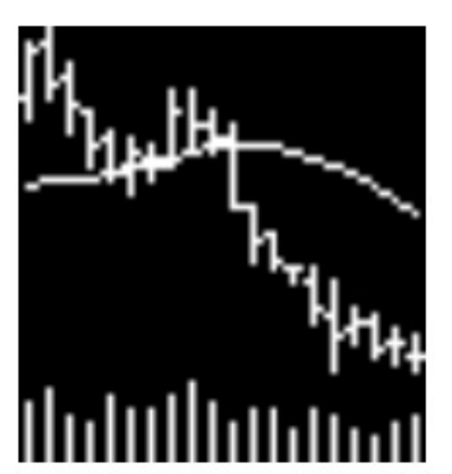
Replication task

Mainly focus on:

- Data Preparation
- Model Design
- Workflow Design
- Performance Evaluation
- Interpretation

Data

- The sample runs from 1993-2019 shows daily opening, high, low prices. The original paper constructs datasets consisting of three scales of horizons (5-day, 20-day, 60-day). Here we just collect the **20-day** version. The total size of data is 8.6G.
- We already transferred the OHLC charts into images following the same procedures. Current images have the same resolution (64 * 60) and added with moving average lines(MA) and volume bars(VB).



Data

- Images labels take value 1 for positive returns ('up') and 0 for non-positive returns ('down'). In addition, we use 2 to mark the NaN value.

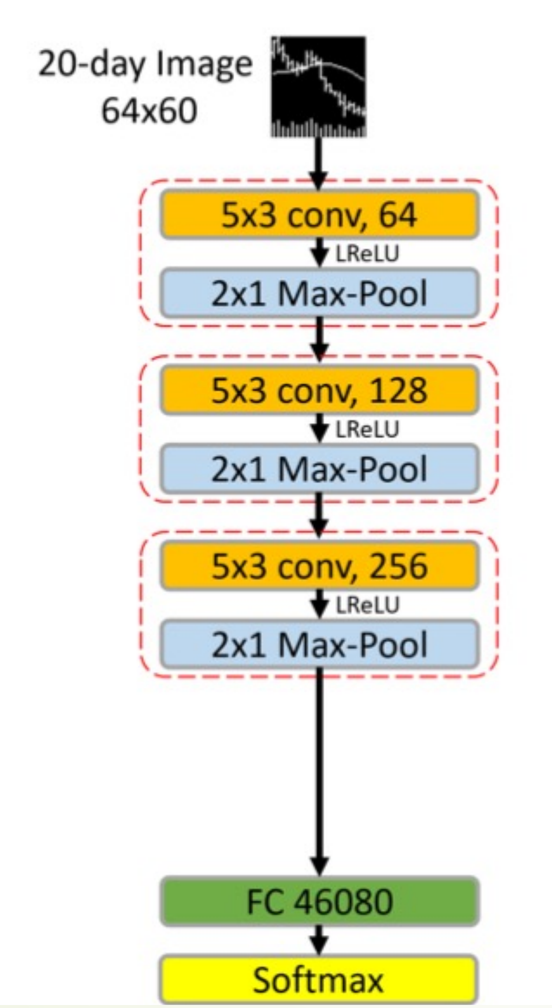
	Date	StockID	EWMA_vol	Retx	Retx_5d	Retx_20d	Retx_60d	Retx_week	Retx_month	Retx_quarter	...	Ret_week	Ret_month	Ret_quarter
0	2017-01-31	10001	0.000450	0.000000	4.370390e-07	-0.000002	-0.011860	NaN	-0.000002	NaN	...	NaN	-0.000002	NaN
1	2017-02-28	10001	0.000180	-0.003937	3.951997e-03	-0.003162	0.003953	NaN	0.003953	NaN	...	NaN	0.009953	NaN
2	2017-03-31	10001	0.000064	0.007936	-7.874612e-03	-0.015749	0.015748	-0.007875	-0.015749	0.017717	...	-0.007875	-0.015749	0.023704
3	2017-04-28	10001	0.000030	0.000000	9.999881e-03	0.016001	0.032002	0.010000	0.016001	NaN	...	0.010000	0.016001	NaN
4	2017-05-31	10001	0.000015	0.000000	4.370390e-07	0.015748	NaN	NaN	0.017717	NaN	...	NaN	0.023703	NaN

Data: Label Format

Retx_20d: < 0
Retx_20d_label: 0

Date	2017-01-31 00:00:00
StockID	10001
EWMA_vol	0.00045
Retx	0.0
Retx_5d	0.0
Retx_20d	-0.000002
Retx_60d	-0.01186
Retx_week	NaN
Retx_month	-0.000002
Retx_quarter	NaN
Retx_tstat	0.0
Retx_5d_tstat	0.00097
Retx_20d_tstat	-0.003558
Retx_60d_tstat	-26.333479
MarketCap	133078.0
Retx_label	0
Retx_5d_label	1
Retx_20d_label	0
Retx_60d_label	0
window_size	20
next_month_ret_0delay	-0.000002
next_month_ret_1delay	-0.000002
Ret	0.0
log_ret	0.0
cum_log_ret	2.075332
Ret_week	NaN
Ret_month	-0.000002
Ret_quarter	NaN
Ret_5d	0.0
Ret_20d	-0.000002
Ret_60d	-0.005954
Ret_65d	0.001998
Ret_180d	NaN
Ret_250d	NaN
Ret_260d	NaN

CNN Architecture Design



- A core building block consists of three operations:
  - convolution
  - activation
  - pooling
- In the paper, for 20-day images, they build a baseline CNN architecture with 3 conv blocks and connected with a fully connected layer as a classifier head.
- See the original paper for details (including the selection of the size of the convolution kernel, the selection of the convolution method, the design of the pooling layer and the selection of the activation function, etc.)

Data Split: Training, Validation, Testing

- Consider dividing the entire sample into training, validation and testing samples.
- In the original paper, they use the first seven-year sample (1993-1999) to train and validate the model, in which 70% of the sample are randomly selected for training and the remaining 30% for validation. The remaining twenty years of data comprise the out-of-sample test dataset.

Loss and evaluation

- You can simply treat the prediction analysis as a classification problem. Use Cross Entropy Loss

$$L_{CE}(y,\hat{y}) = -y\log(\hat{y}) - (1-y)log(1-\hat{y})$$

- To measure the classification accuracy, a true positive (TP) or true negative (TN) occurs when a predicted "up" probability of greater than 50% coincides with a positive realized return and a probability less than 50% coincides with a negative return. False positives and negatives (FP and FN) are the complementary outcomes.

$$Accuracy = (TP+TN)/(TP+TN+FP+FN)$$

  - For more evaluation metrics or methods, like ***Sharpe Ratio***, please refer to the original paper.

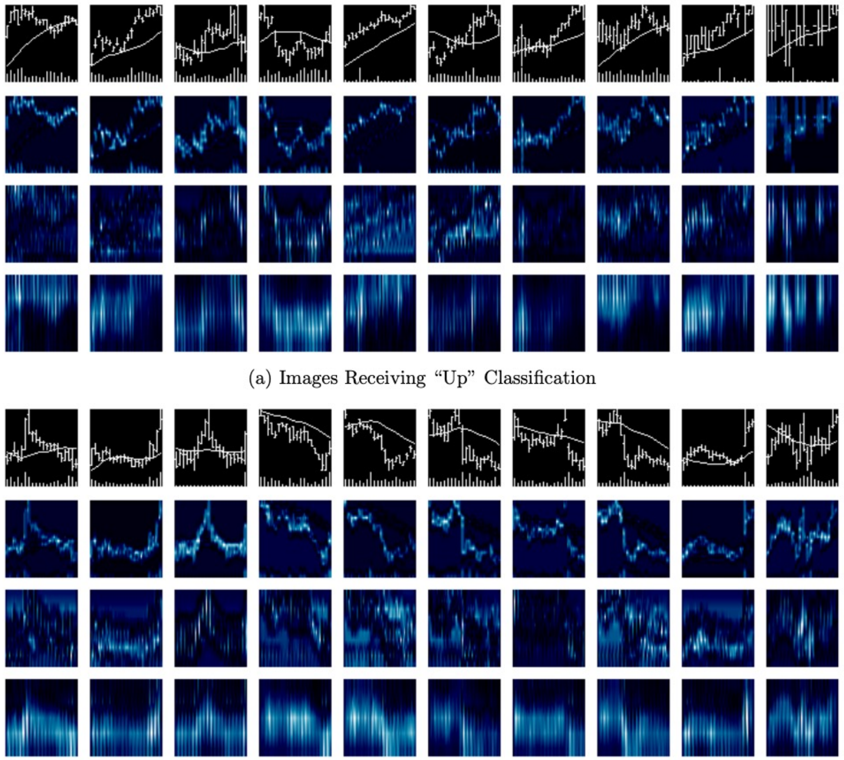
Training Process

- The author adopts several ways to combat over-fitting issue and aid efficient computation.
- They applied the Xavier initialization for weights in each layer, which guarantees faster convergence by scaling the initial weights.
- Other techniques like applying dropout, using batch normalization and early stopping also assists better performance.
- We recommend referring to the training details mentioned in the paper 3.3 when training the baseline model.

Exploring Interpretability

- Using a visualization method (Grad-CAM) to understand how different image examples activate the regions of the CNN to trigger 'up' or 'down' return predictions.

Figure 12: I20R20 Grad-CAM for 20 Images from 2019



(a) Images Receiving "Up" Classification

(b) Images Receiving "Down" Classification

Note: Brighter regions of the heatmap correspond to regions with the higher activation. For each panel, the first row is the original images followed by the grad-CAM for each layer in the CNN.

Extensions

- Ablation studies and test robustness,
  - For example, you can perform the sensitivity analysis of the CNN prediction model to alternate choices in model architecture (e.g., varying the number of filters in each layer or varying the number of layers, like the paper shows in Table 18)

Table 18: Sensitivity to Model Structure and Estimation, I20R20

		Loss		Acc.		Correlation		Sharpe Ratio	
		V	T	V	T	Spearman	Pearson	EW	VW
Baseline		**0.688**	**0.692**	**0.540**	**0.525**	**0.052**	**0.032**	**2.18**	**0.56**
Filters (64)	32	0.688	0.691	0.541	0.526	0.053	0.032	1.91	0.43
	128	0.692	0.692	0.535	0.527	0.050	0.030	2.01	0.45
Layers (3)	2	0.688	0.692	0.540	0.525	0.047	0.028	1.61	0.19
	4	0.689	0.692	0.538	0.524	0.051	0.031	2.00	0.38
Dropout (0.50)	0.00	0.698	0.695	0.532	0.521	0.044	0.027	2.32	0.54
	0.25	0.691	0.693	0.536	0.525	0.050	0.030	2.11	0.56
	0.75	0.691	0.692	0.530	0.525	0.041	0.024	1.30	0.05
BN (yes)	no	0.685	0.691	0.549	0.528	0.059	0.037	2.44	0.58
Xavier (yes)	no	0.688	0.692	0.540	0.526	0.053	0.033	2.10	0.39
Activation (LReLU)	ReLU	0.689	0.693	0.538	0.520	0.052	0.031	1.71	0.35
Max Pool Size (2×1)	2×2	0.687	0.692	0.545	0.526	0.056	0.034	2.09	0.41
Filter Size (5×3)	3×3	0.689	0.693	0.538	0.522	0.050	0.028	1.39	0.28
	7×3	0.688	0.691	0.541	0.527	0.055	0.033	2.01	0.41
Dilation/Stride (2,1)/(3,1)	(2,1)/(1,1)	0.689	0.692	0.537	0.526	0.052	0.033	2.04	0.53
	(1,1)/(3,1)	0.689	0.692	0.538	0.526	0.049	0.030	1.66	0.17
	(1,1)/(1,1)	0.687	0.692	0.545	0.527	0.055	0.035	2.09	0.52

- What's more
  - We encourage you not limited to simple binary classification tasks, since the label files we provided consist of more meaningful attributes, containing both categorical and numerical values.
    - For example, you can use the same 20-day horizon images to train your model to predict the return trend of different subsequent y-days even the detailed return values. (y can be 5, 20 even larger).
  - Stock prices might be of high noise, so you may consider other financial markets, e.g. cryptocurrency market
    - For example, Kaggle Contest on G-Research Crypto Forecasting: https://www.kaggle.com/c/g-research-crypto-forecasting

Thank you!