# Deep Learning

## A USP 2019 Summer School Course

Instructor: Dr. Zhangyang (Atlas) Wang

Email: atlaswang@tamu.edu

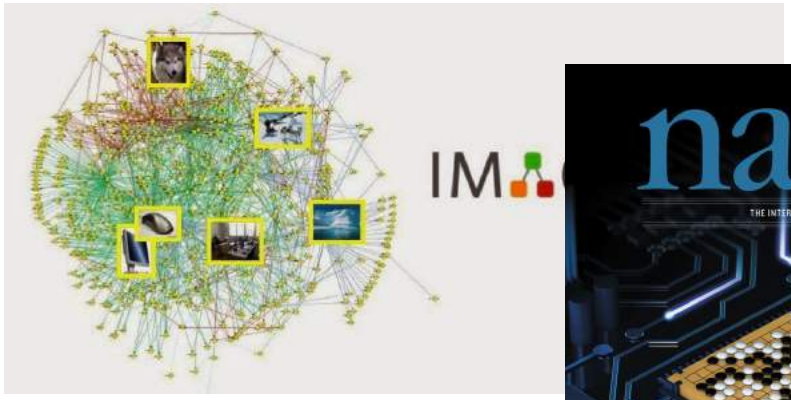
Website: www.atlaswang.com

COMPUTER SCIENCE & ENGINEERING
TEXAS A&M UNIVERSITY

2019 Turing Award winners, left to right, Yann LeCun, Geoff Hinton, and Yoshua Bengio, reoriented artificial intelligence around neural networks

# A Triumph of Deep Learning: 2012 - present

Top-performers in many tasks, over many domains



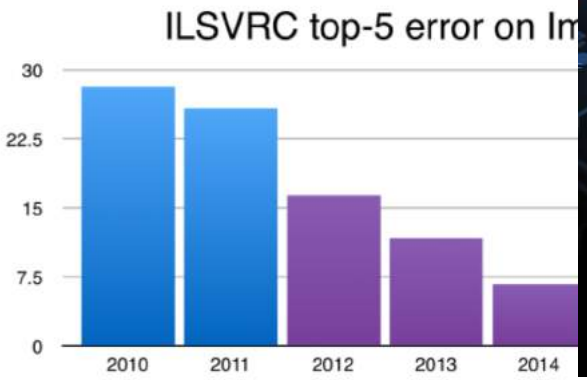
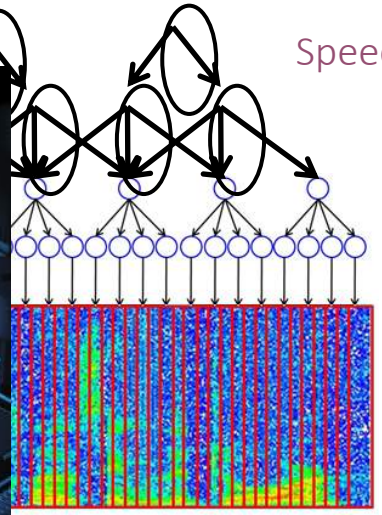


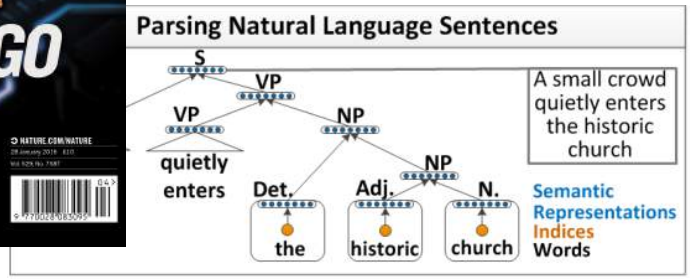

Image classification, detection, localization…


Speech Recognition

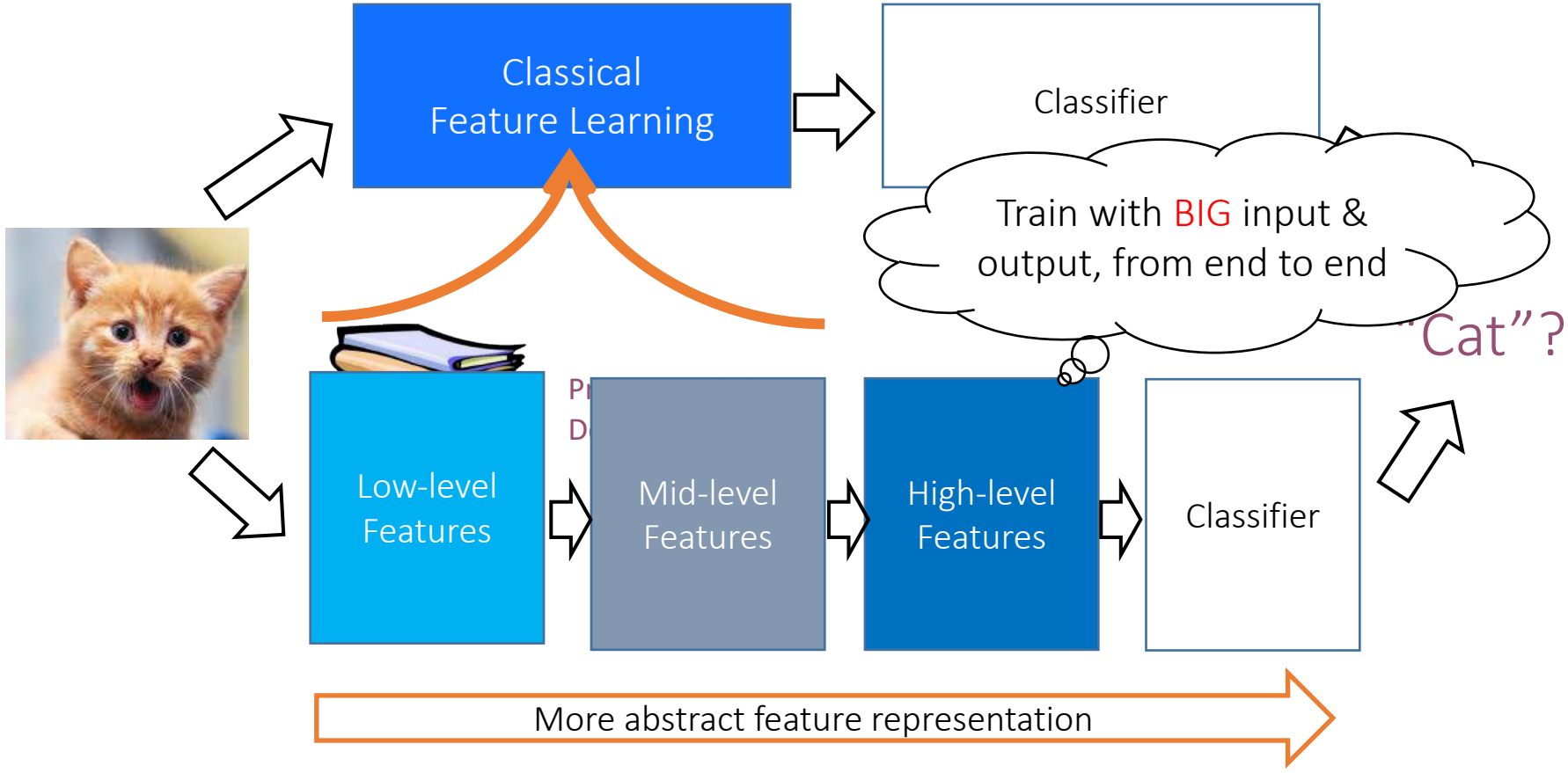


Spectrogram

NLP

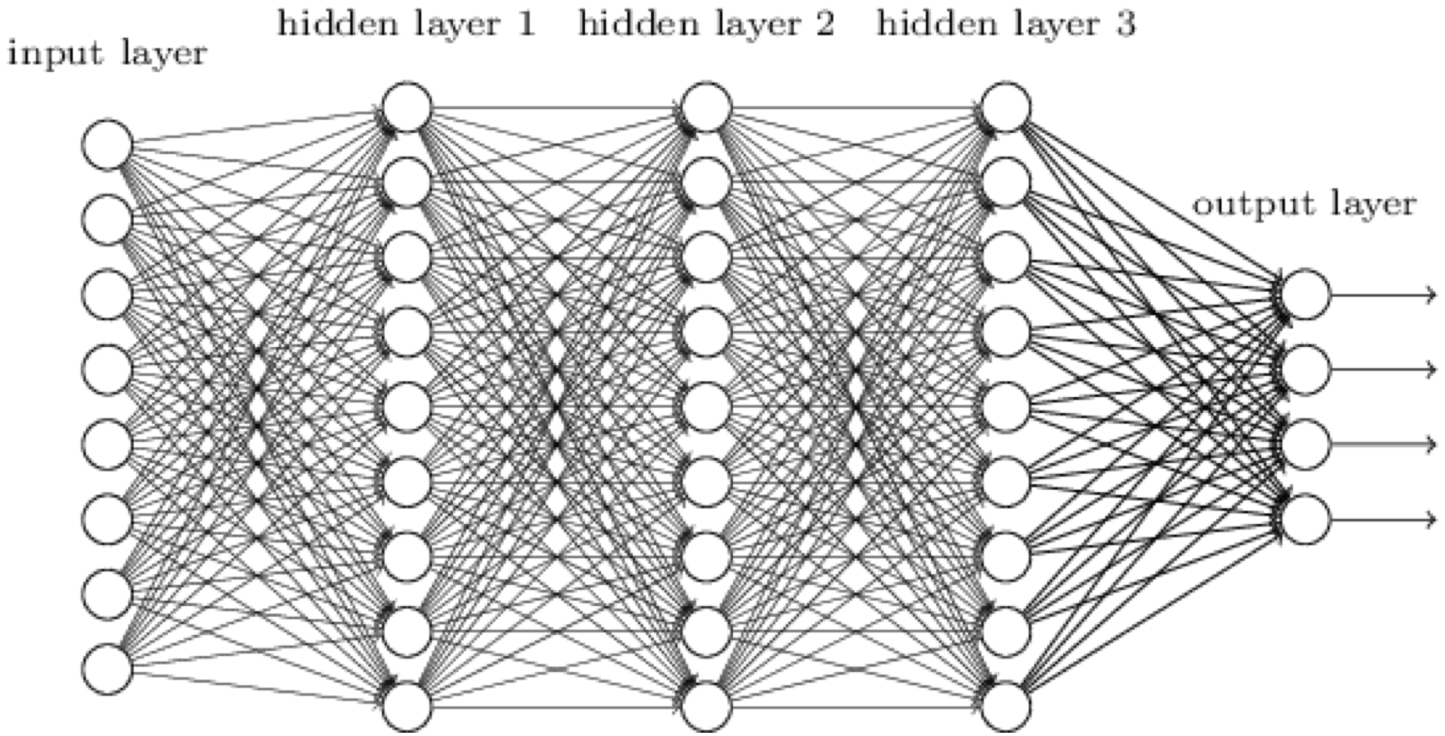


# Feature learning: Going Deep

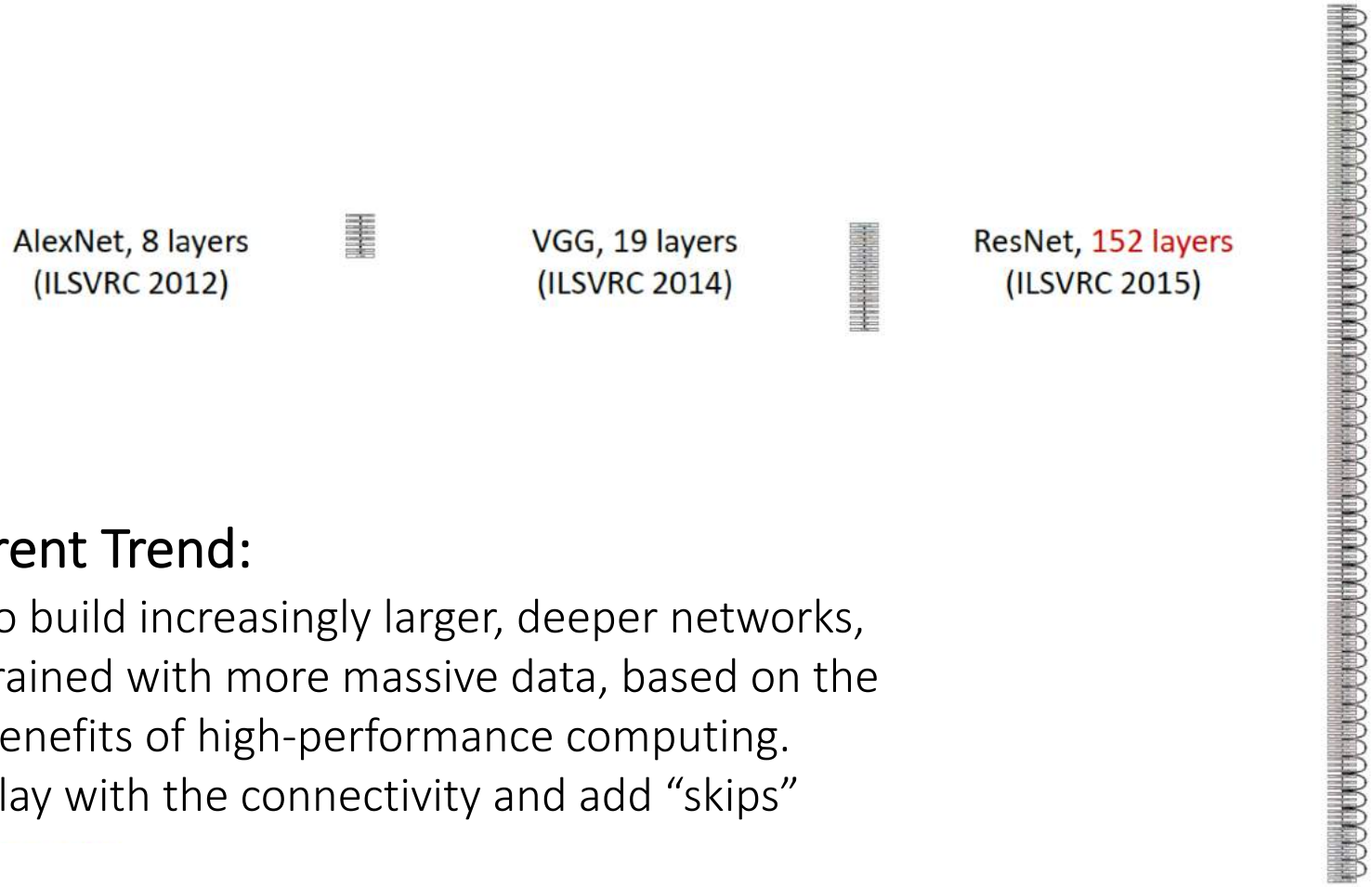


# Deep learning

- Learn a *feature hierarchy* all the way from raw inputs (e.g. pixels) to classifier
- Each layer extracts features from the output of previous layer
- Train all layers jointly



# Status Quo




## Current Trend:

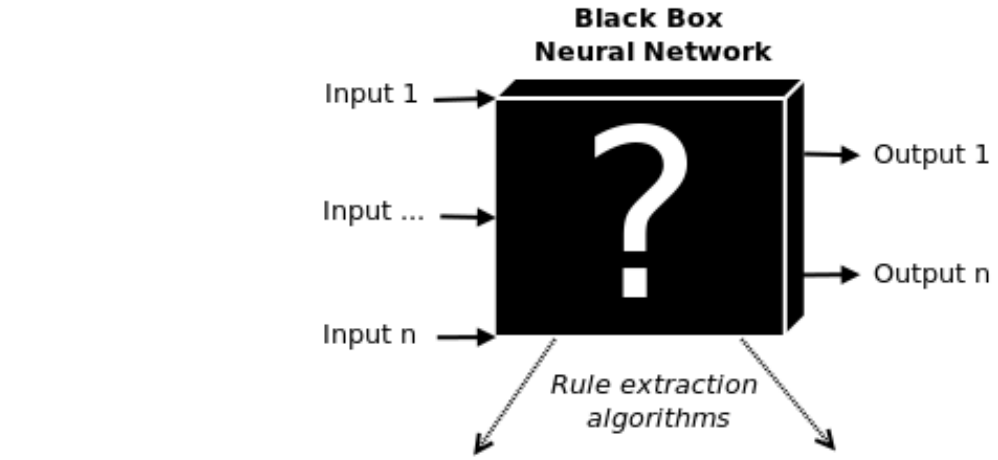
- To build increasingly larger, deeper networks, trained with more massive data, based on the benefits of high-performance computing.
- Play with the connectivity and add “skips”


I WAS WINNING IMAGENET
UNTIL A DEEPER MODEL CAME ALONG
imgflip.com

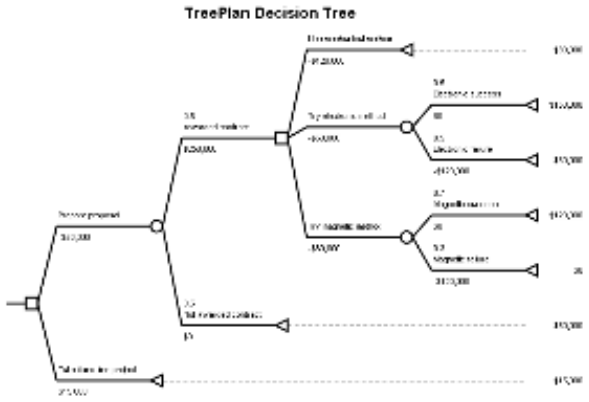

专业深度学习调试
caffe安装 10元
CNN 5元/层
RNN 8元/层
专业

# Grand Challenges

- Why/how deep learning works?
  - *In theory, many cases shouldn't even work…*
  - Gap between engineering (or art) and science: Lack of theoretical understandings & guarantees, and analytical tools
  - Training is computationally expensive and difficult,  relying on many "magics"
  - No principled way to incorporate domain expertise, or to interpret the model behaviors

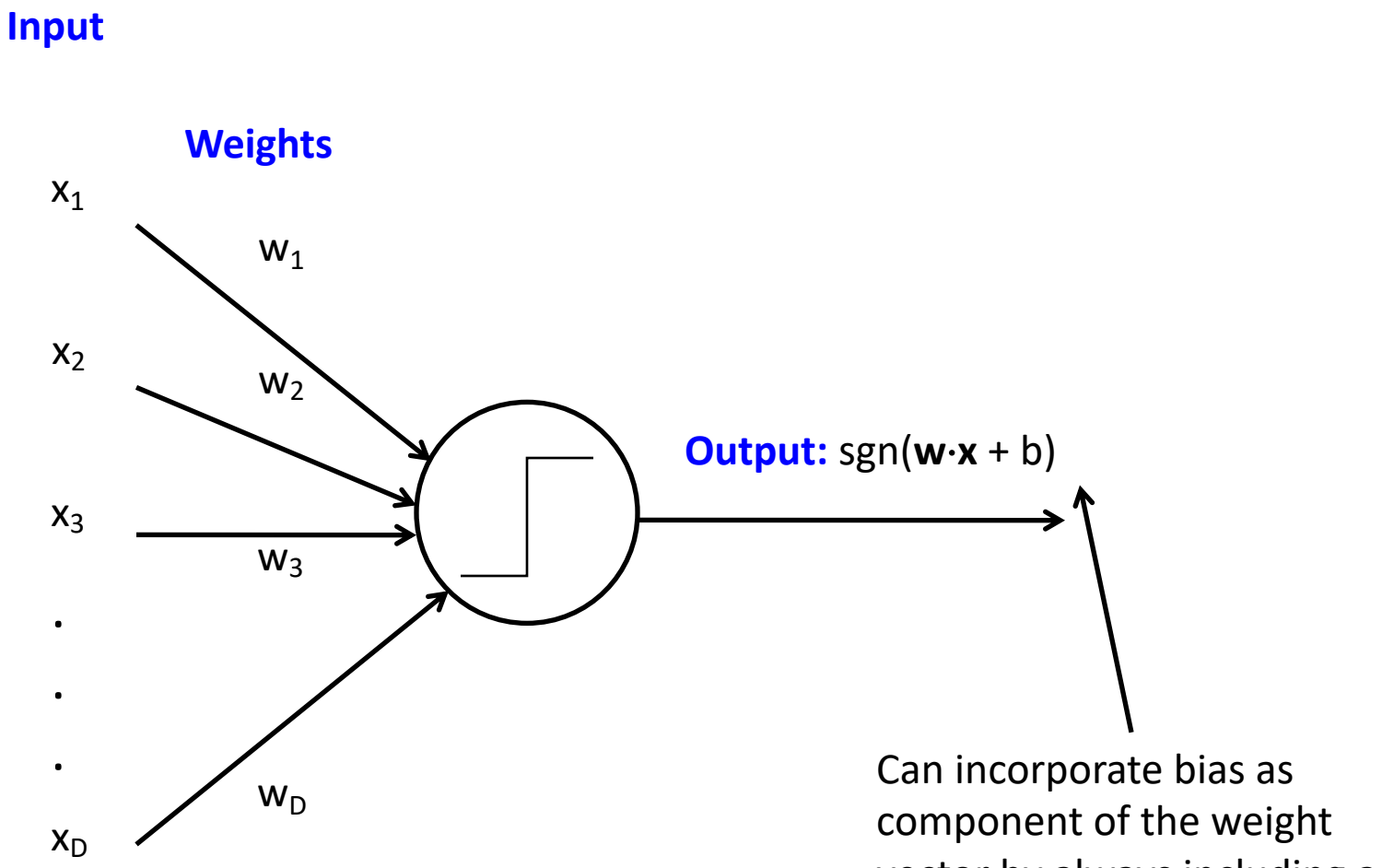



Extract Logic in either symbolic or fuzzy form

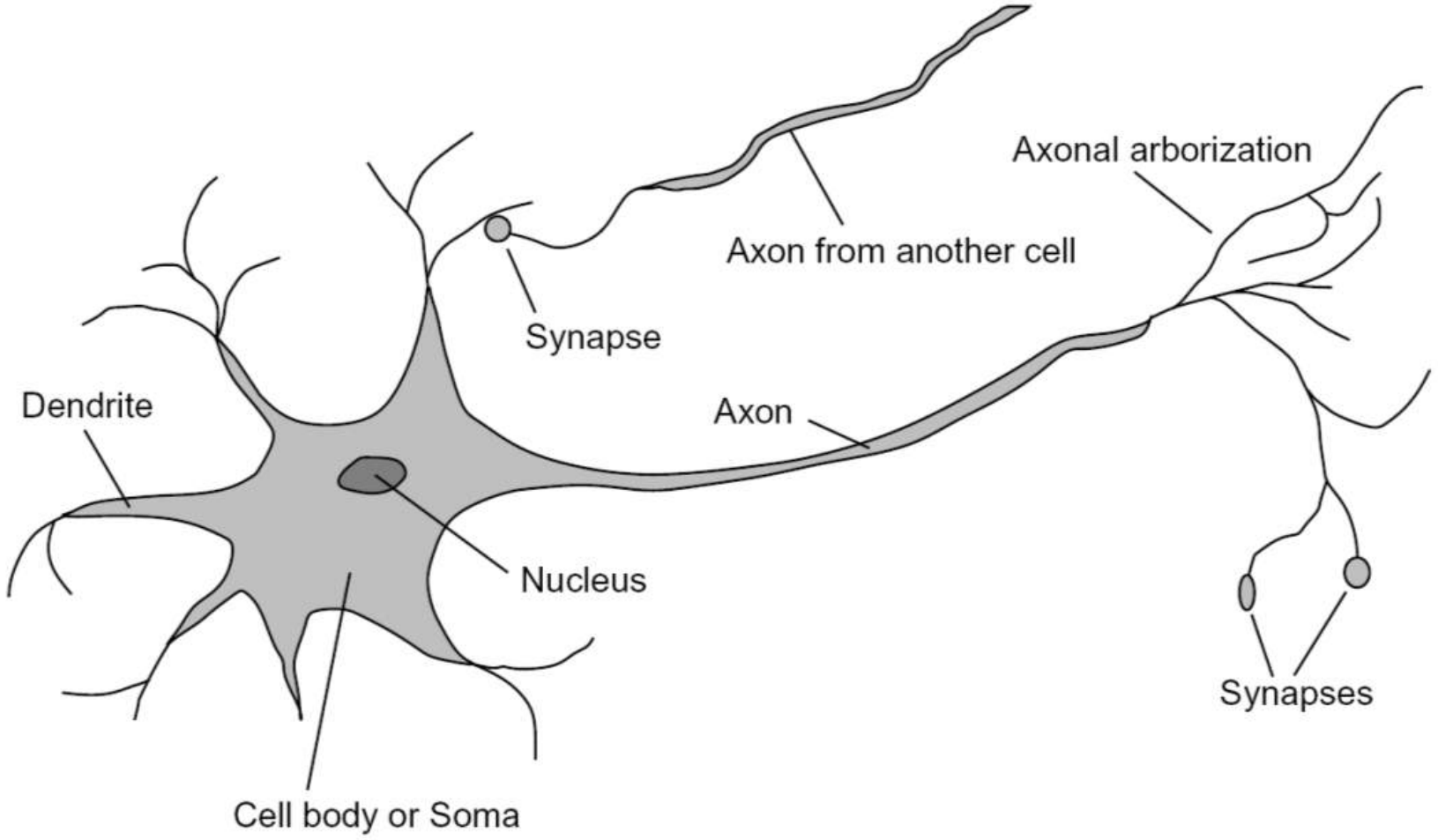



Extract Decision Trees contraining Logic

# Perceptron


Input
Weights
$x_1$
$w_1$
$x_2$
$w_2$
$x_3$
$w_3$
$w_D$
$x_D$
Output: sgn(**w**·**x** + b)
Can incorporate bias as component of the weight vector by always including a feature with value set to 1

# Loose inspiration: Human neurons



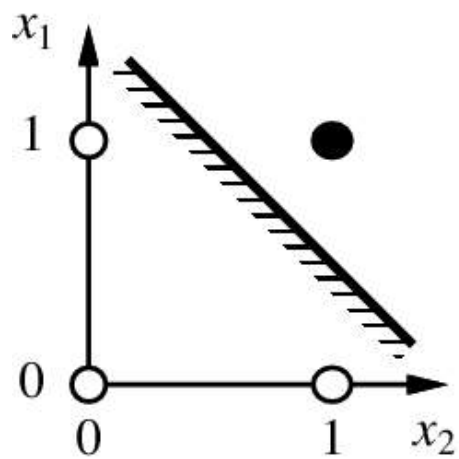
# Perceptron training algorithm

- Initialize weights
- Cycle through training examples in multiple passes (*epochs*)
- For each training example:
  - Classify with current weights: $y' = \mathrm{sgn}(\mathbf{w} \cdot \mathbf{x})$
  - If classified incorrectly, update weights:

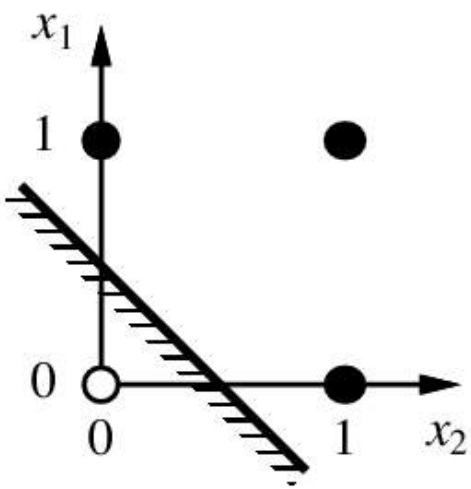
$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\left(y - y'\right)\mathbf{x}$$

  - α is a *learning rate* that should decay as a function of epoch t, e.g., 1000/(1000+t)

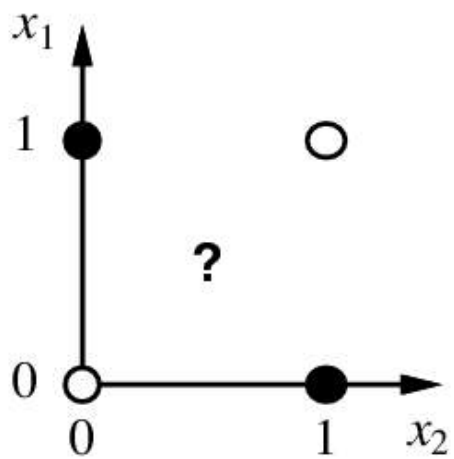
# Linear separability




$x_1$ **and** $x_2$



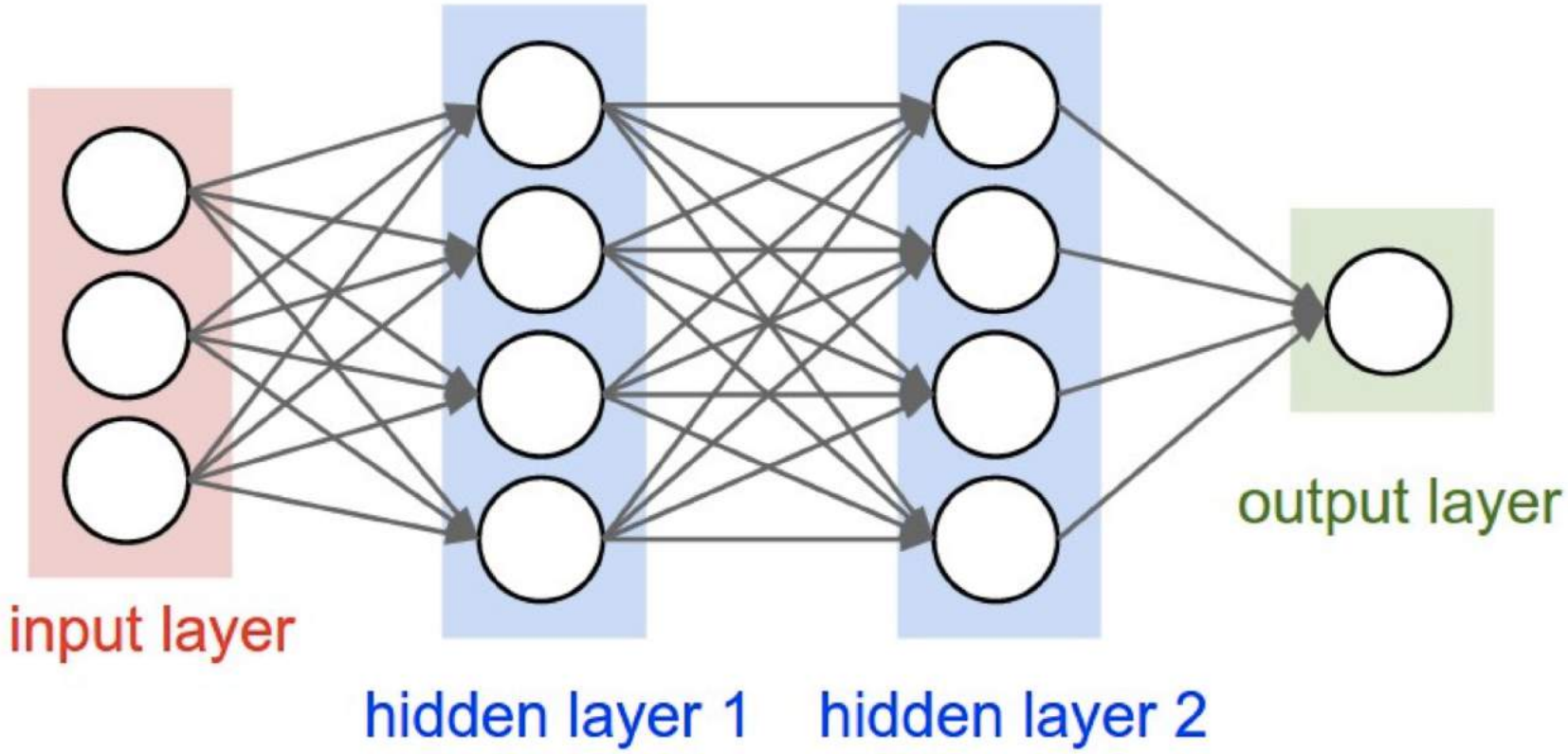

$x_1$ **or** $x_2$




$x_1$ **xor** $x_2$

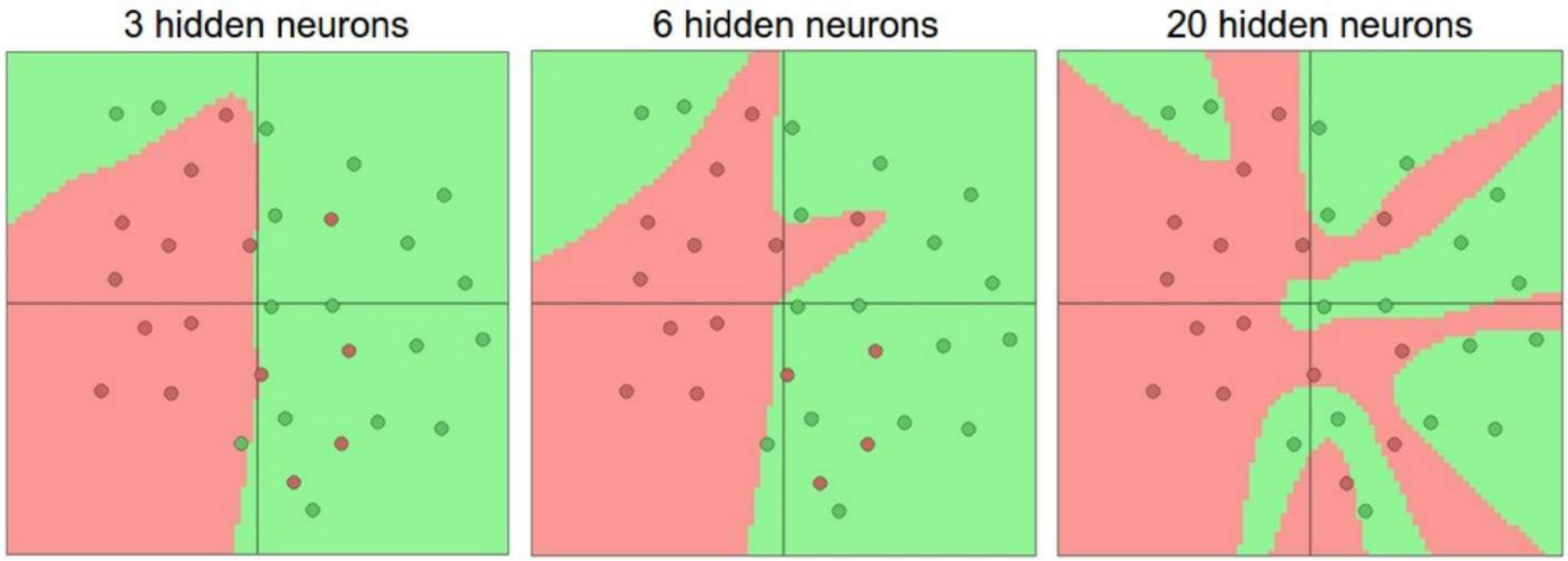
# How do we make nonlinear classifiers out of perceptrons?

- Build a multi-layer neural network!



# Network with a single hidden layer

- Hidden layer size and *network capacity*:




Source: http://cs231n.github.io/neural-networks-1/

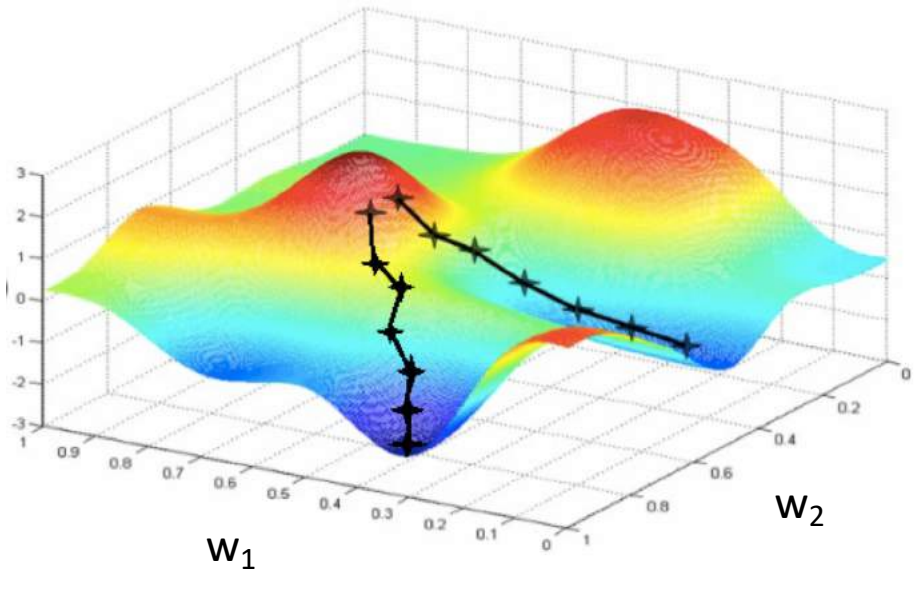
# Training of multi-layer networks

- Find network weights to minimize the error between true and estimated labels of training examples:

$$E(\mathbf{w}) = \sum_{j=1}^{N} \left( y_j - f_{\mathbf{w}}(\mathbf{x}_j) \right)^2$$

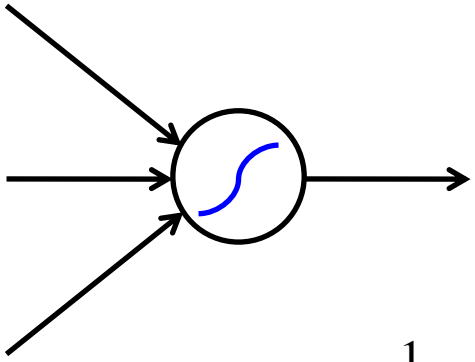
- Update weights by **gradient descent:**

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial E}{\partial \mathbf{w}}$$

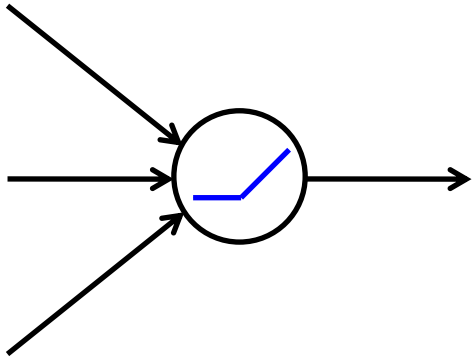


# Training of multi-layer networks

- **Gradient descent** requires neural networks to be equipped with a (nearly) differentiable nonlinearity function, called neuron

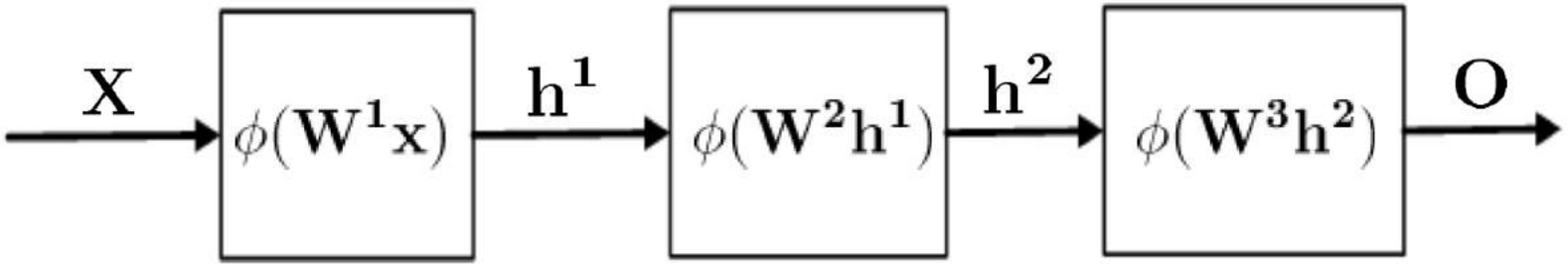


**Sigmoid:** $g(t) = \frac{1}{1+e^{-t}}$

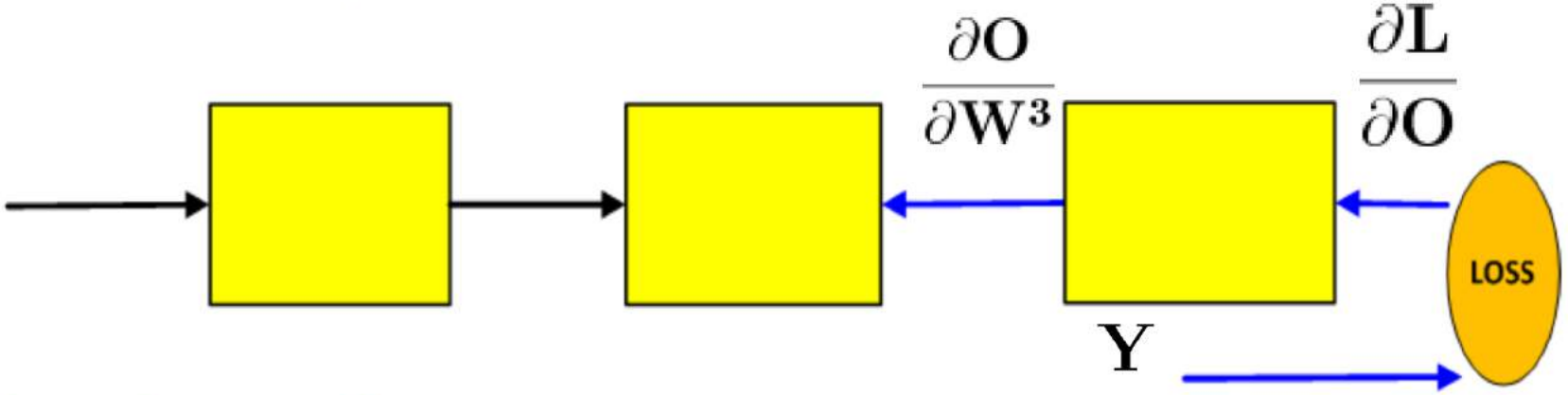


**Rectified linear unit (ReLU):** $g(t) = \max(0,t)$

# Forward-Backward Propagation




**Forward propagation:** $h(\mathbf{x}) = \phi(\mathbf{W}\mathbf{x})$




**Backward propagation:** $\frac{\partial \mathbf{L}}{\partial \mathbf{W^3}} = \frac{\partial \mathbf{L}}{\partial \mathbf{O}} \frac{\partial \mathbf{O}}{\partial \mathbf{W^3}}$ **(Chain Rule)**

# NNs are Universal Approximators (in theory)

Let $\varphi(\cdot)$ be a nonconstant, bounded, and monotonically-increasing continuous function. Let $I_m$ denote the $m$-dimensional unit hypercube $[0,1]^m$. The space of continuous functions on $I_m$ is denoted by $C(I_m)$. Then, given any $\varepsilon > 0$ and any function $f \in C(I_m)$, there exist an integer $N$, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$, where $i = 1, \cdots, N$, such that we may define:

$$F(x) = \sum_{i=1}^{N} v_i \varphi\left(w_i^T x + b_i\right)$$

as an approximate realization of the function $f$ where $f$ is independent of $\varphi$; that is,

$$|F(x) - f(x)| < \varepsilon$$

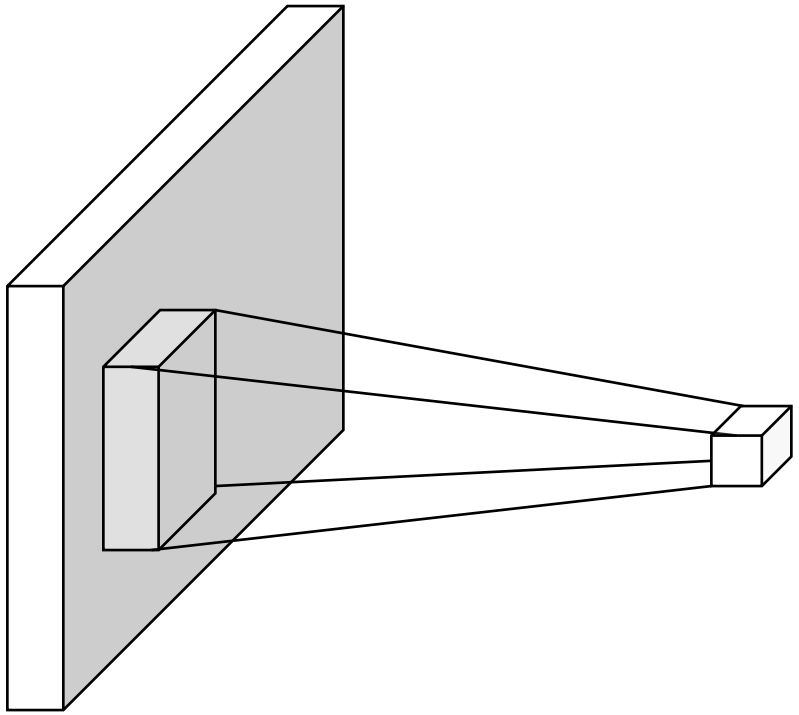
for all $x \in I_m$. In other words, functions of the form $F(x)$ are dense in $C(I_m)$.

- A feed-forward network with **a single hidden layer** containing **a finite number of *nonlinear* neurons**, can approximate **any continuous function on compact subsets of $R^n$**, under mild assumptions.
- It is not the specific choice of the activation function, but rather the **multilayer feedforward architecture** itself which gives neural networks the potential of being universal approximators.
- It does not touch upon the **algorithmic learnability** of those parameters.

# From NNs to Convolution NNs

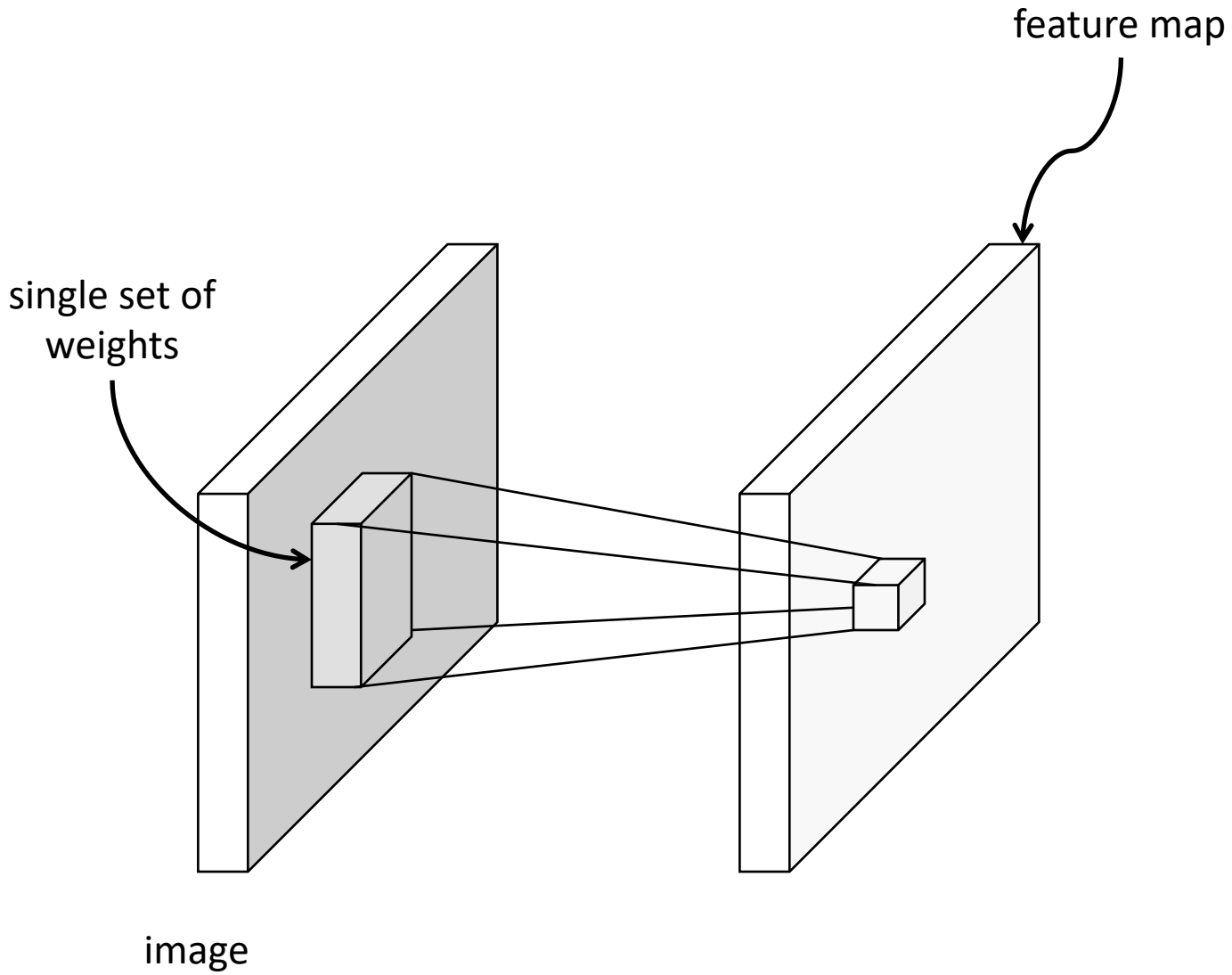
The most important building block in modern deep learning

# From fully connected to convolutional networks

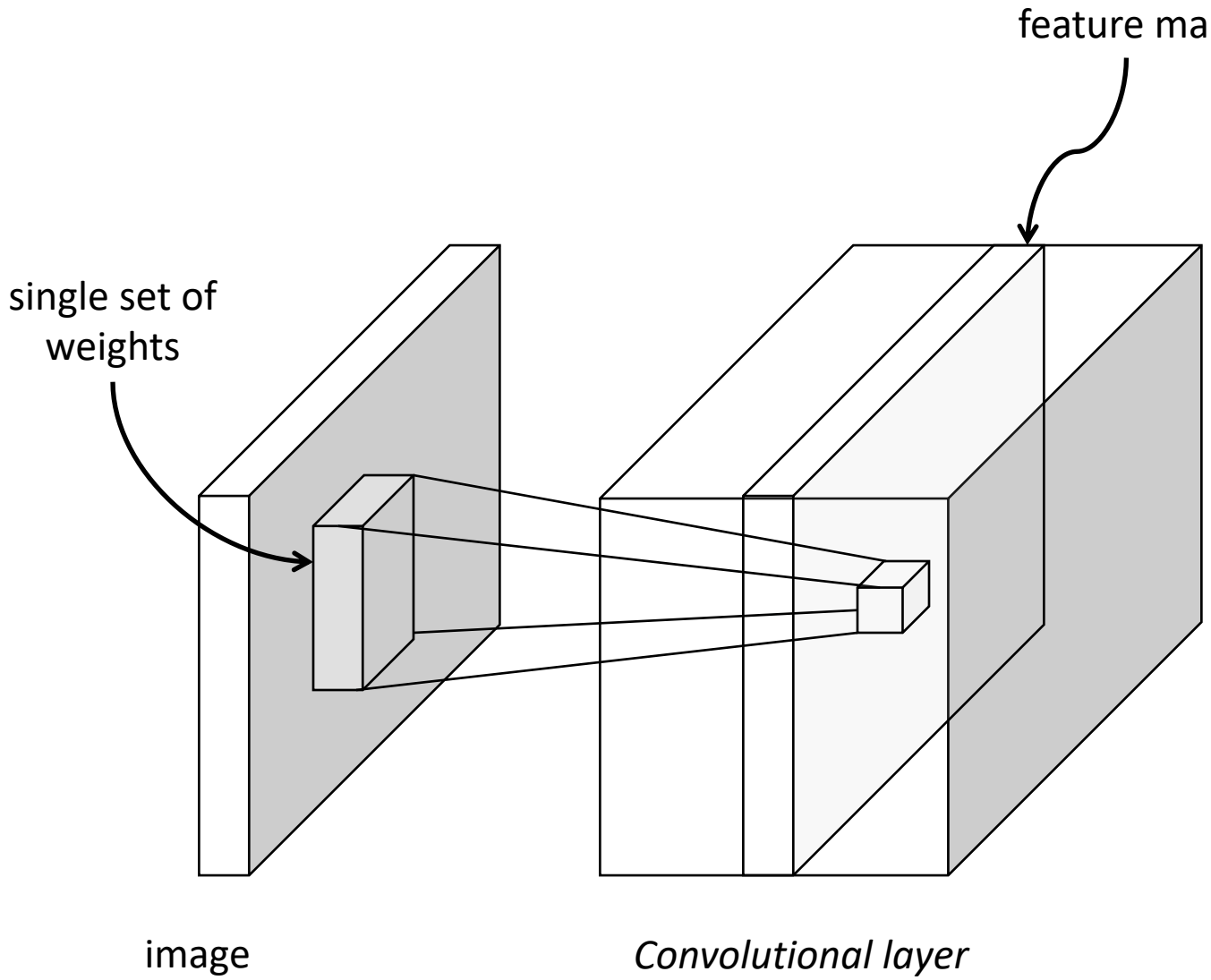


image

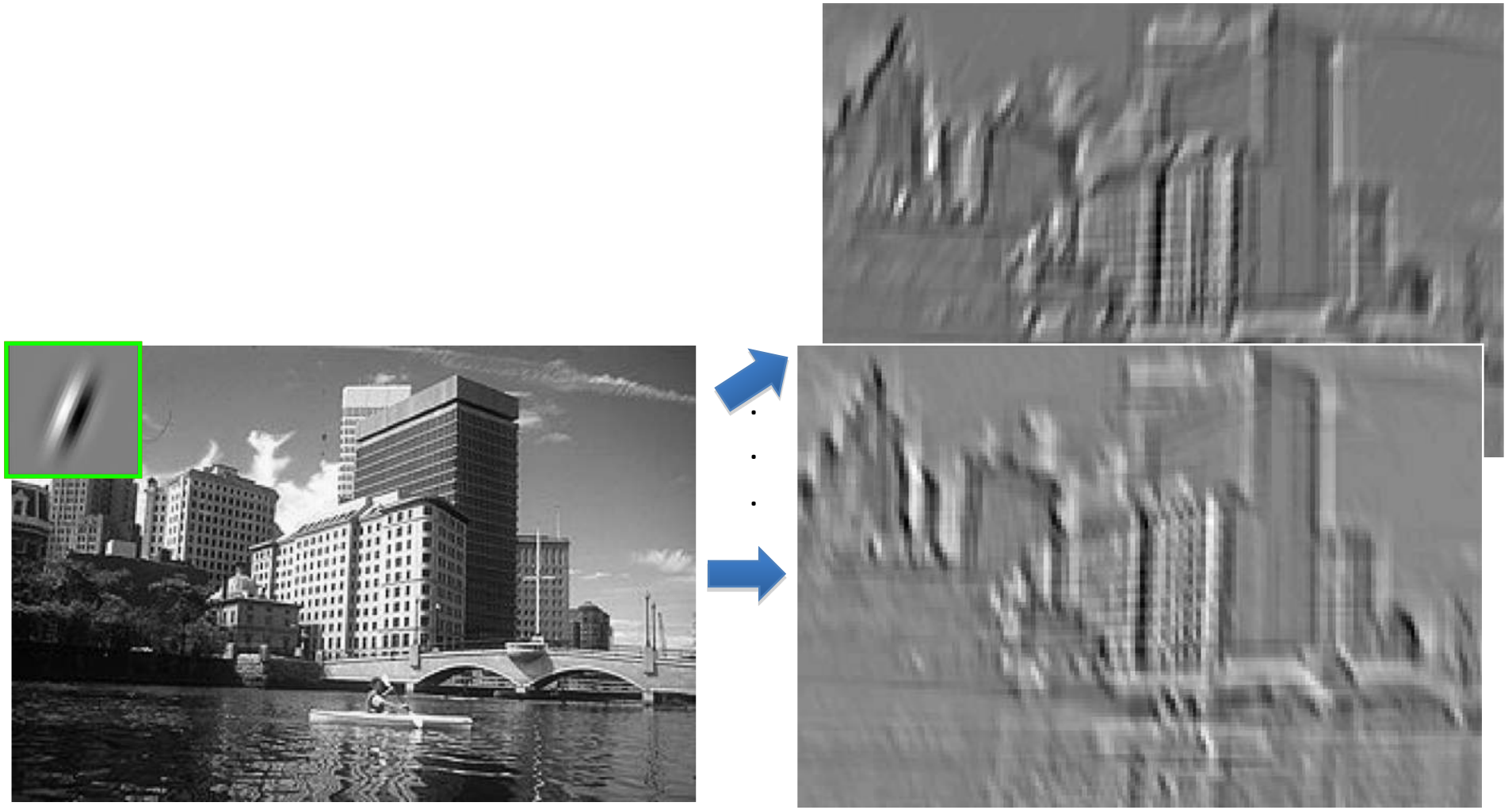
# From fully connected to convolutional networks



# From fully connected to convolutional networks



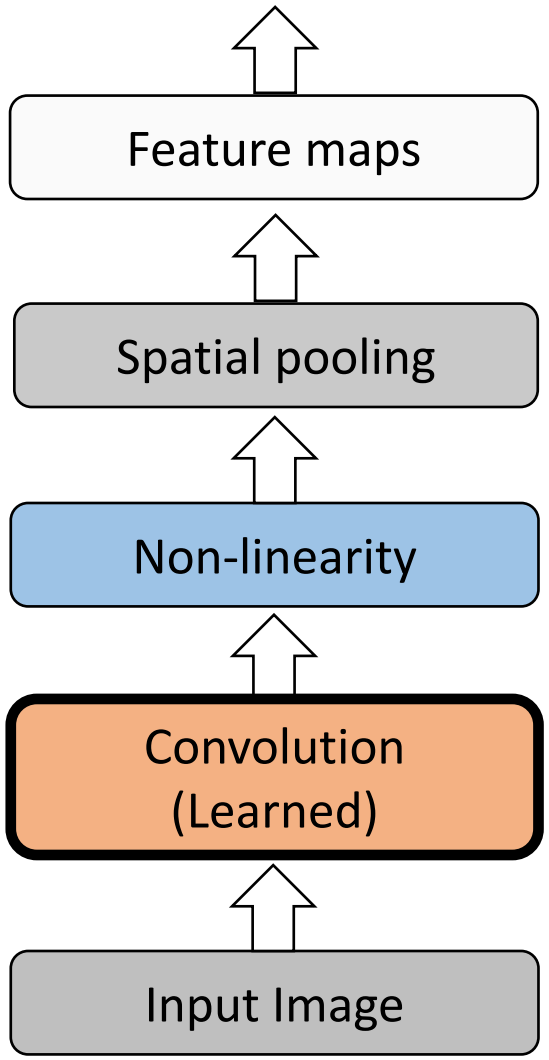
# Convolution as feature extraction

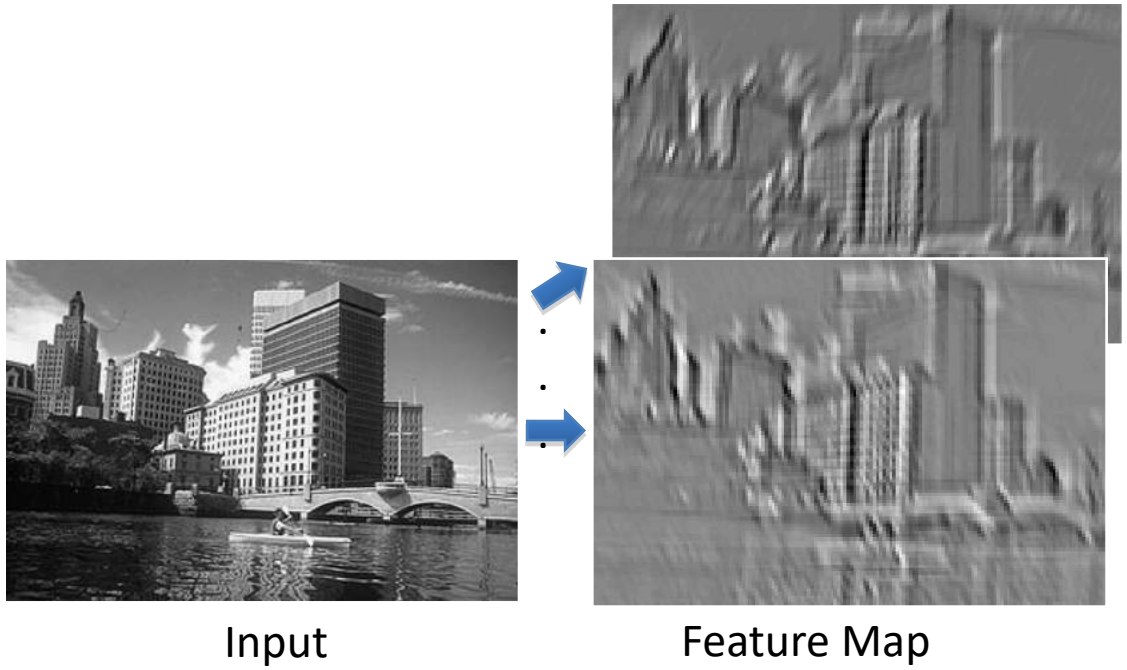


Input

Feature Map

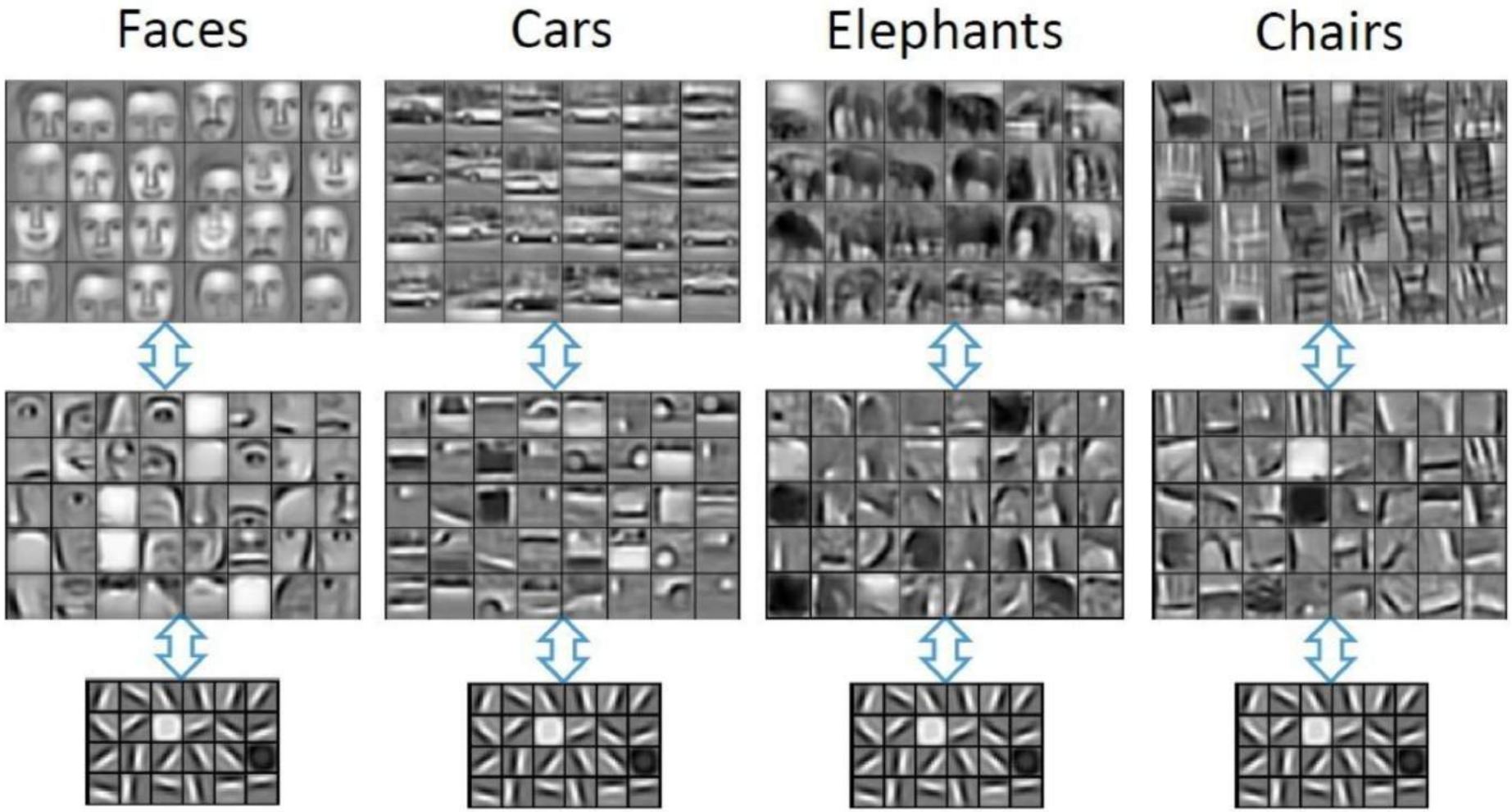
# Key operations in a CNN



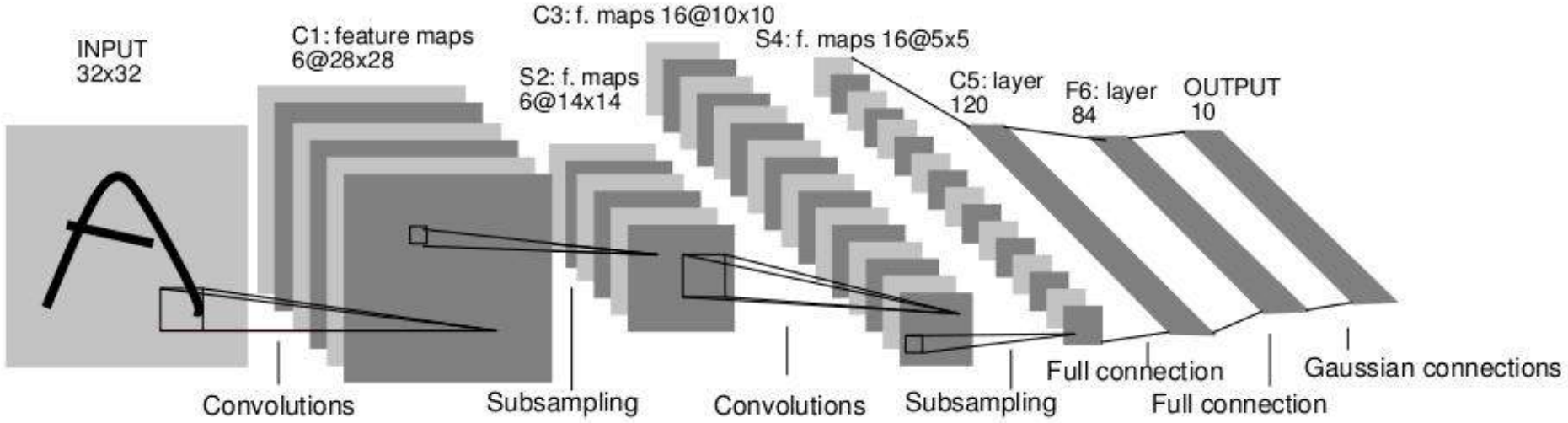




Source: R. Fergus, Y. LeCun

# Deep Features (May) Learn Semantic Hierarchy



# LeNet-5




- Average pooling
- Sigmoid or tanh nonlinearity
- Fully connected layers at the end
- Trained on MNIST digit dataset with 60K training examples

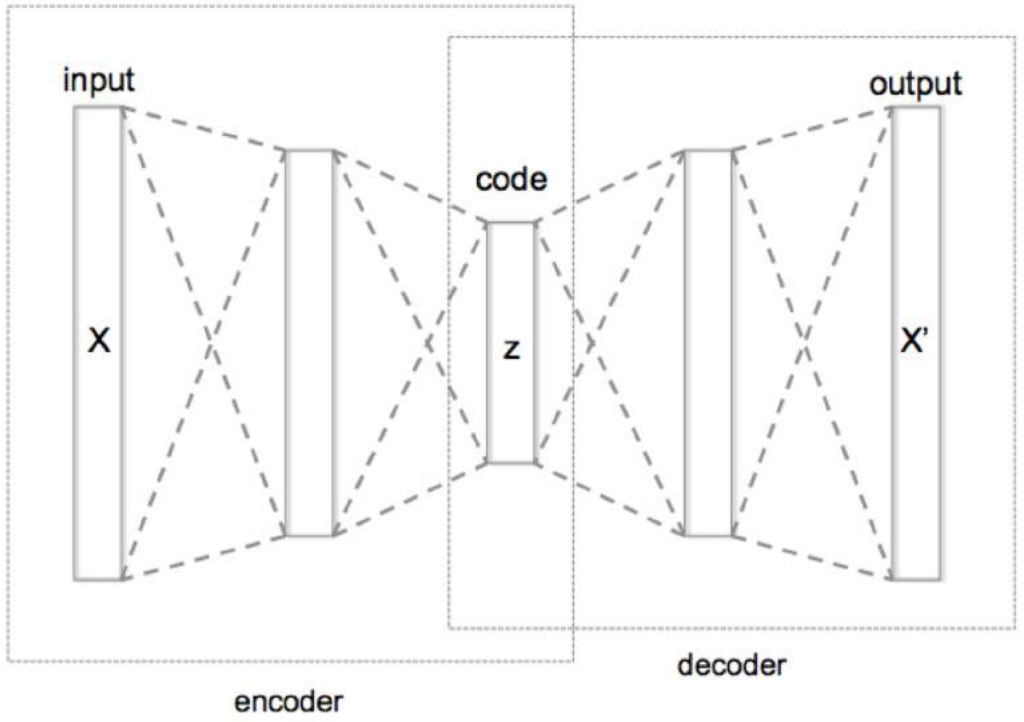
Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, Gradient-based learning applied to document recognition, Proc. IEEE 86(11): 2278–2324, 1998.

# Popular Backbones: From LeNet to DenseNet

A Remarkable Odyssey to Artificial Intelligence by Human Intelligence

# Auto-Encoder

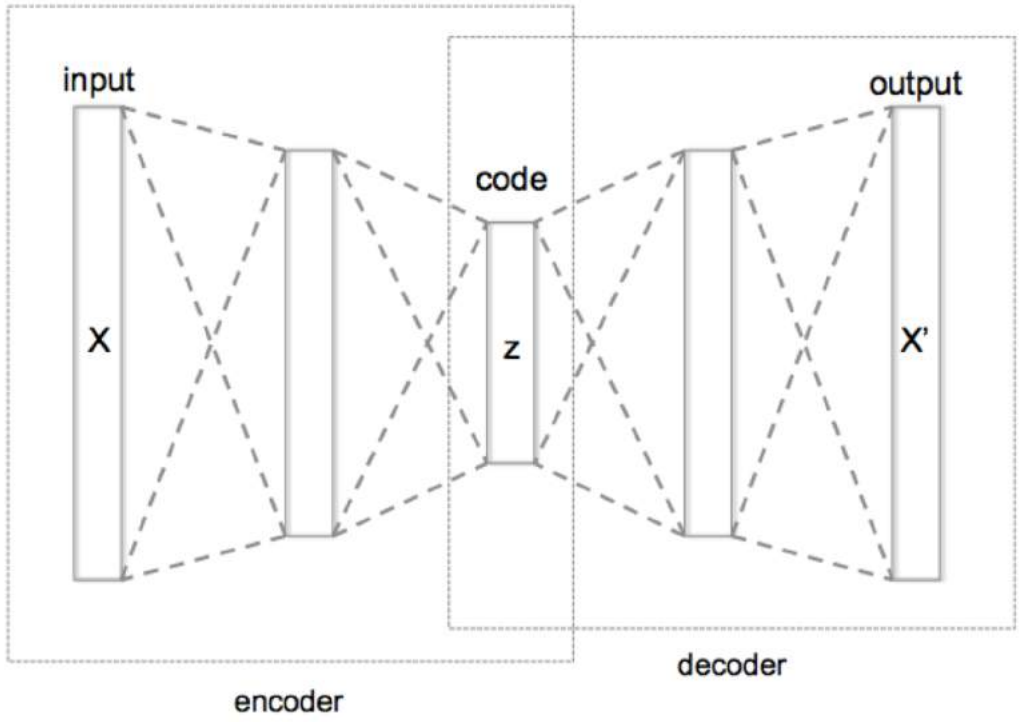
- Unsupervised feature extraction
- Reconstruct the input from itself via using “bottleneck”




**X = X’**

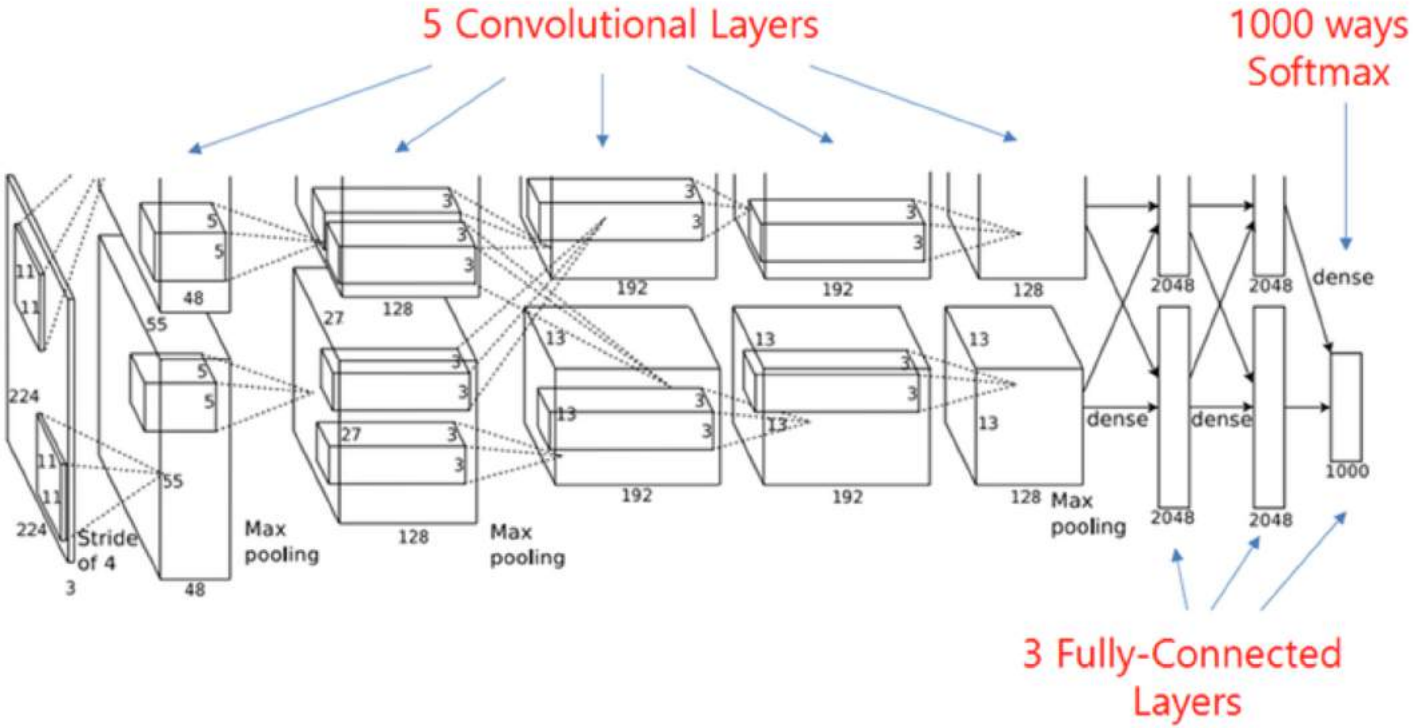
# Denoising Auto-Encoder

- Reconstruct the input from a slightly corrupted “noisy” version
- Purpose: learning robust features for better generalization




**X = X’ + noise**

# AlexNet, 2012




- The **FIRST** winner deep model in computer vision, and one of the most classical choices for domain experts to adapt for their applications
- 5 convolutional layers + 3 fully-connected layers + softmax classifier
- **Key Technical Features**: ReLU, dropout, data augmentation

# VGG-Net, 2014

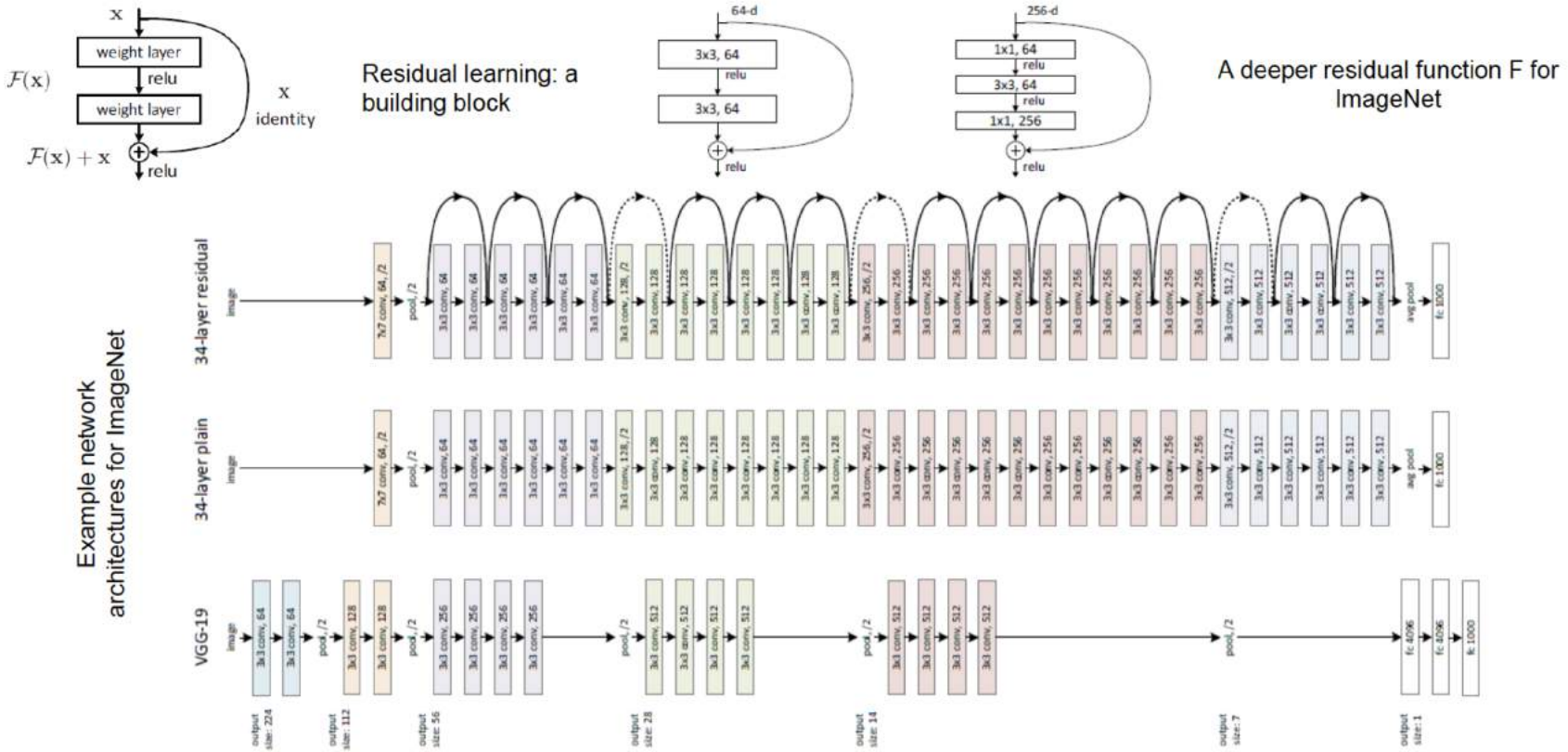
| ConvNet Configuration | | | | | |
|---|---|---|---|---|---|
| A | A-LRN | B | C | D | E |
| 11 weight layers | 11 weight layers | 13 weight layers | 16 weight layers | 16 weight layers | 19 weight layers |
| input (224 × 224 RGB image) | | | | | |
| conv3-64 | conv3-64<br>**LRN** | conv3-64<br>**conv3-64** | conv3-64<br>conv3-64 | conv3-64<br>conv3-64 | conv3-64<br>conv3-64 |
| maxpool | | | | | |
| conv3-128 | conv3-128 | conv3-128<br>**conv3-128** | conv3-128<br>conv3-128 | conv3-128<br>conv3-128 | conv3-128<br>conv3-128 |
| maxpool | | | | | |
| conv3-256<br>conv3-256 | conv3-256<br>conv3-256 | conv3-256<br>conv3-256 | conv3-256<br>conv3-256<br>**conv1-256** | conv3-256<br>conv3-256<br>**conv3-256** | conv3-256<br>conv3-256<br>conv3-256<br>**conv3-256** |
| maxpool | | | | | |
| conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512<br>**conv1-512** | conv3-512<br>conv3-512<br>**conv3-512** | conv3-512<br>conv3-512<br>conv3-512<br>**conv3-512** |
| maxpool | | | | | |
| conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512<br>**conv1-512** | conv3-512<br>conv3-512<br>**conv3-512** | conv3-512<br>conv3-512<br>conv3-512<br>**conv3-512** |
| maxpool | | | | | |
| FC-4096 | | | | | |
| FC-4096 | | | | | |
| FC-1000 | | | | | |
| soft-max | | | | | |

Key Technical Features:

- Increase depth (up to 19)
- Smaller filter size (3)

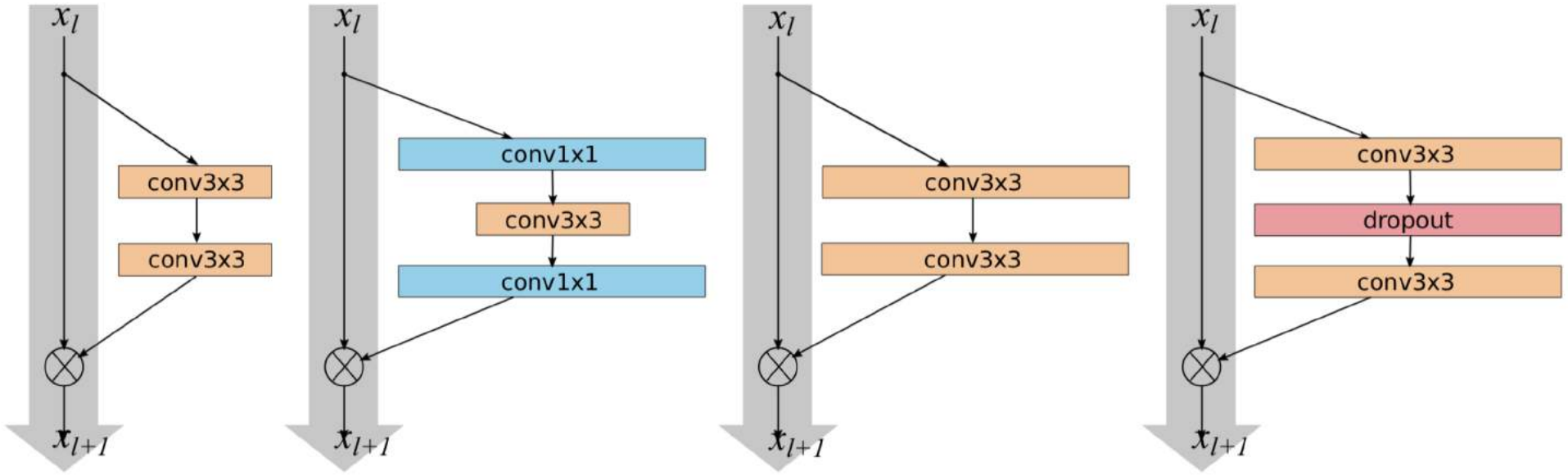
Configurations D and E are widely used for various tasks, called *VGG-16* and *VGG-19*

# Deep Residual Network (ResNet), 2015




**Key Technical Features**: skip connections for residual mapping, up to > 1000 layers

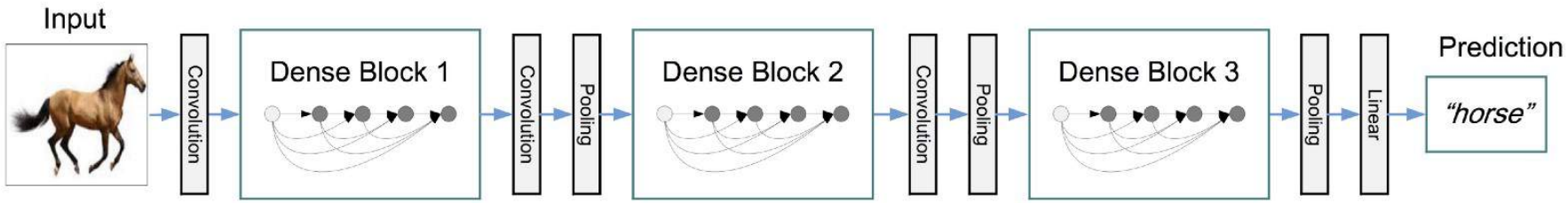
# Wide ResNet, 2016

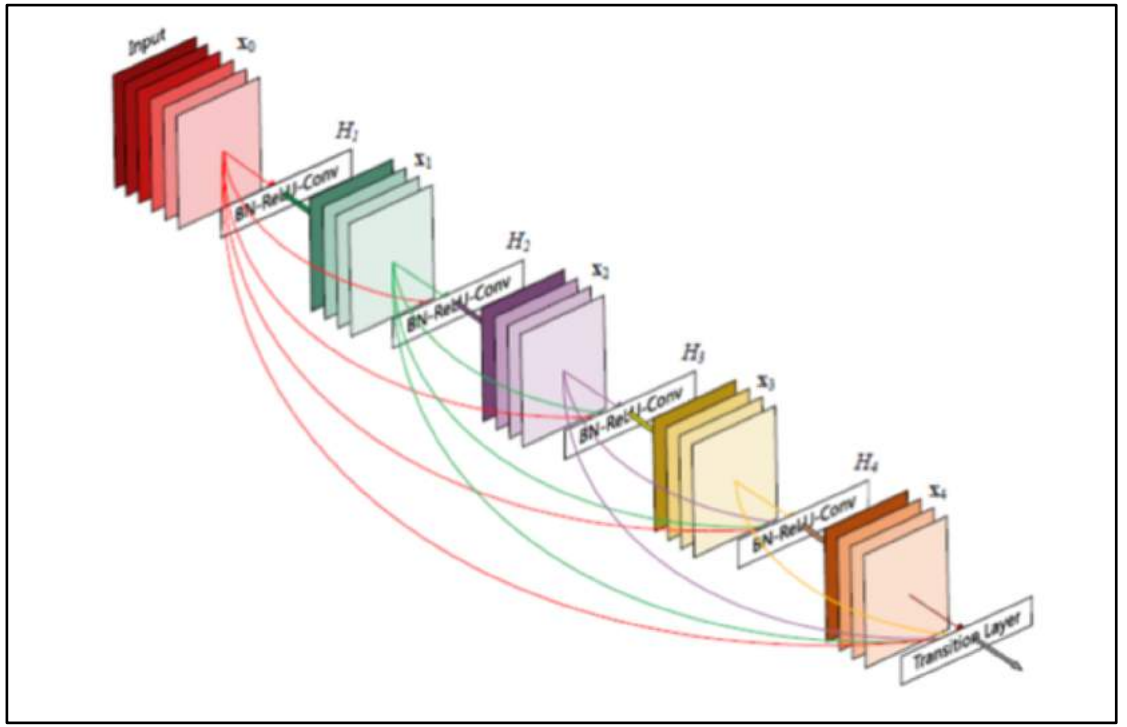



(a) basic (b) bottleneck (c) basic-wide (d) wide-dropout

- Widening of ResNet blocks (if done properly) provides a more effective way of improving performance of residual networks compared to increasing their depth.
- A wide 16-layer deep network has the same accuracy as a 1000-layer thin deep network and a comparable number of parameters, although being several times faster to train.

# Densely Connected Convolutional Networks (DenseNet), 2017

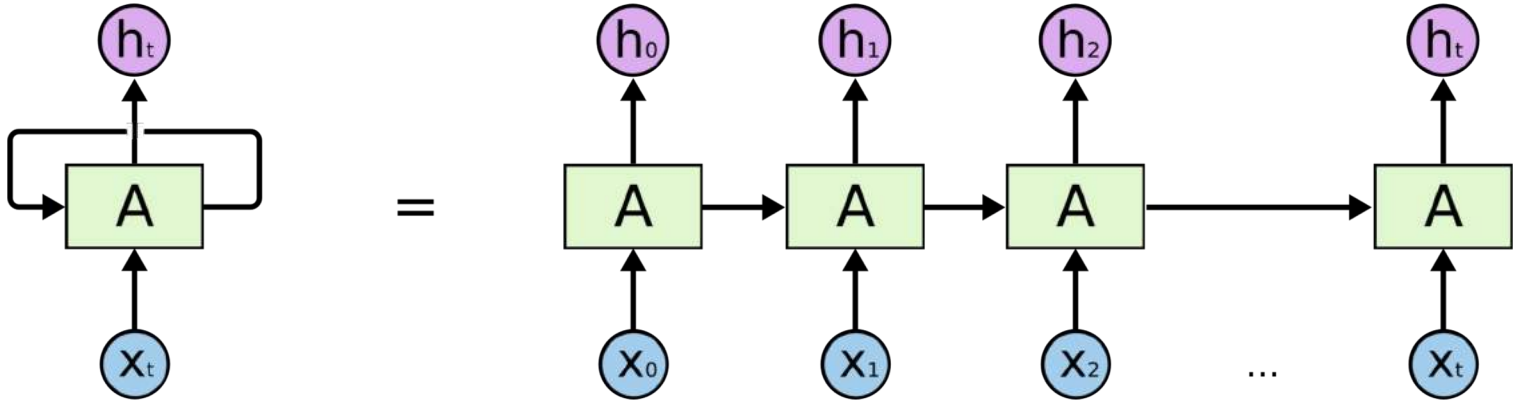



Key Technical Features:

- Finer combination of multi-scale features (or whatever…)

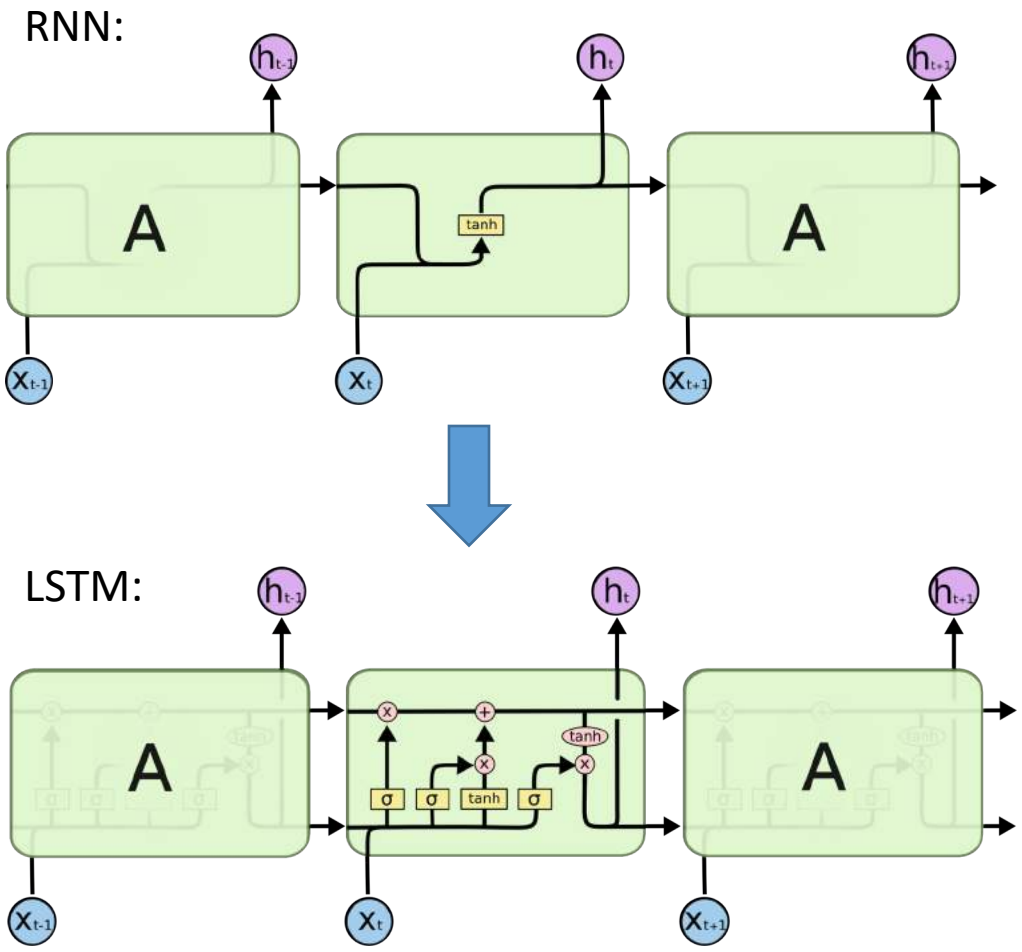
# RNN and LSTM




- A RNN is **unfolded** its forward and backward computations.
- Backpropagation Through Time (**BPTT**): Because the parameters are shared by all time steps in the network, the gradient at each output depends not only on the calculations of the current time step, but also the previous time steps
- **Vanishing/Exploding Gradients**: Difficulty in learning long-term dependency

*An intro article for RNN/LSTM*: "Understanding LSTM Networks": http://colah.github.io/posts/2015-08-Understanding-LSTMs/

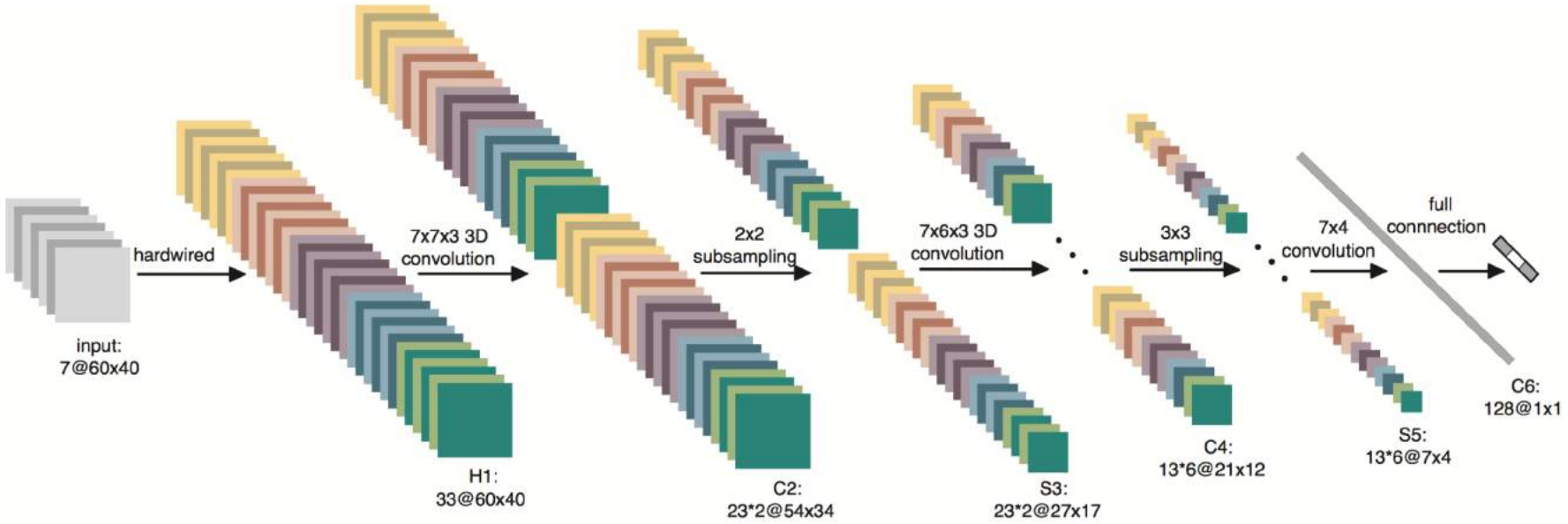
# RNN and LSTM




- A Long Short Term Memory (**LSTM**) combats vanishing gradients through a **gating** mechanism, thus capturing **long-term dependency** better.
  - Similar “shortcut” idea to ResNet
- A LSTM does the exact same thing as a RNN, just in a different way!
- **Key Idea:** the gating functions are learned together with weights, and determine how much information we would like keep from last state and current computation, etc.

# (More) Art of Convolutions

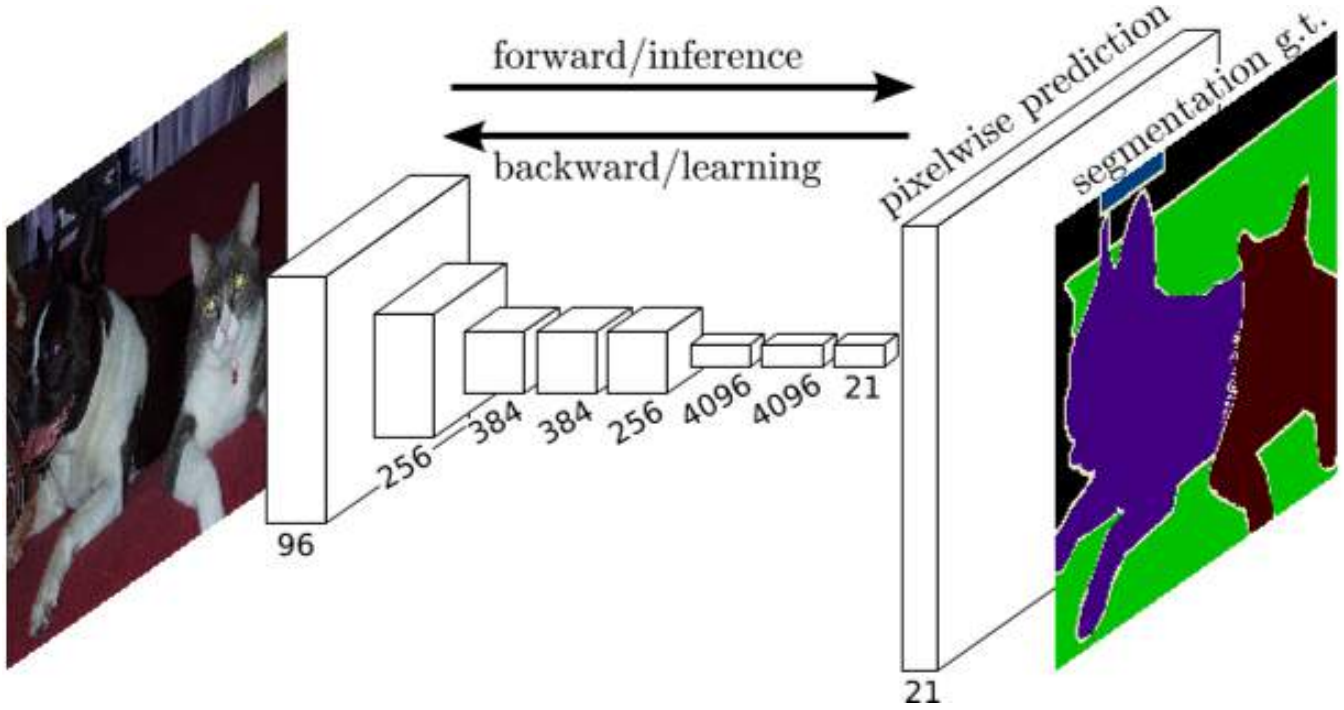
# 3D Convolutional Network (3D CNN), 2011




Key Technical Features:

- Going from 2D convolutional filters to 3D filters, to take temporal coherence into consideration

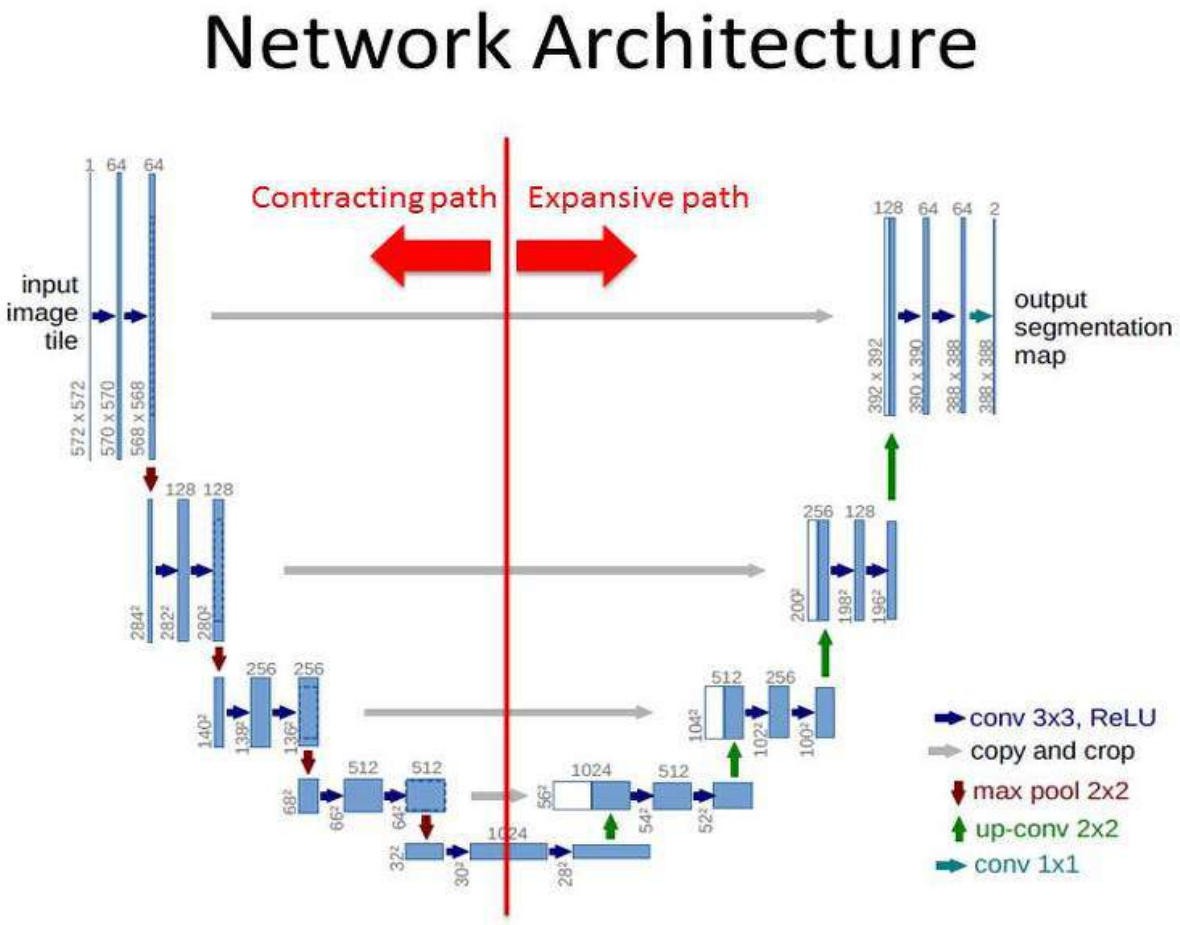
# Fully Convolutional Network (FCN), 2014




Key Technical Features:

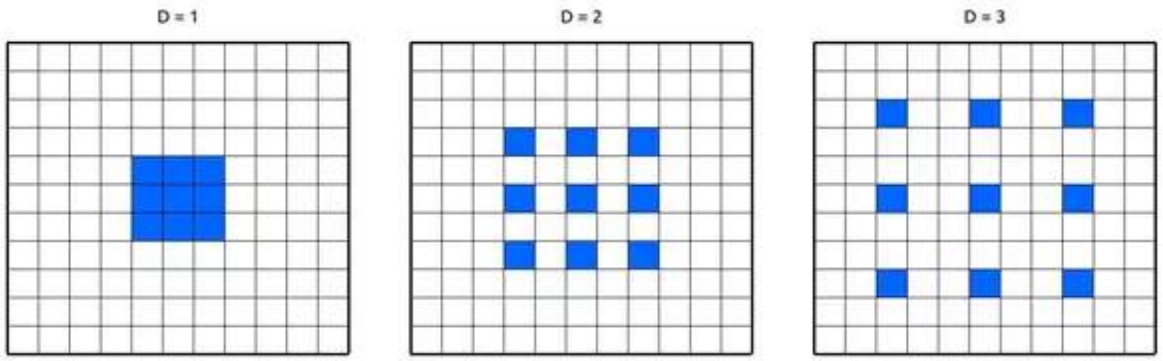
- No fully-connected layer -> No fixed requirement on input size
- Widely adopted in pixel-to-pixel prediction tasks, e.g., image segmentation

# U-Net, 2015




- The architecture consists of a contracting path to capture context
- …and a symmetric expanding path to enable precise localization.
- Also fully convolutional
- Very popular backbone for dense prediction (image segmentation, restoration…)

# Dilated Convolutions, 2015



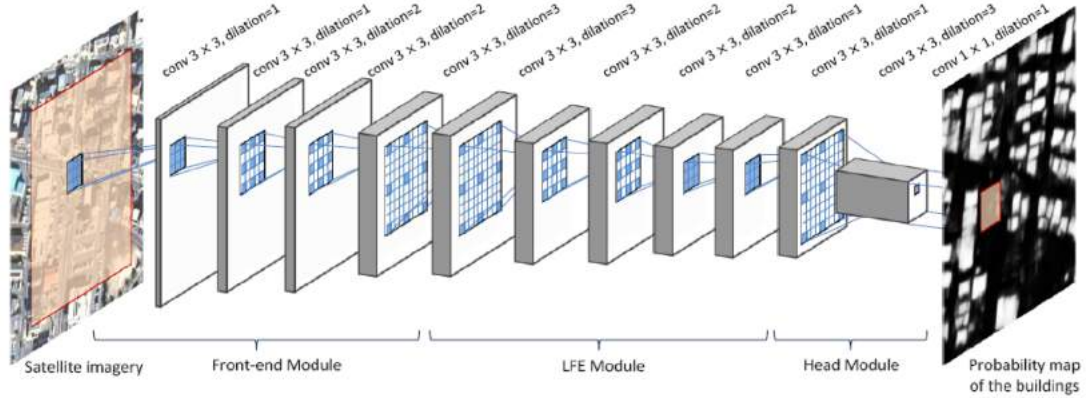
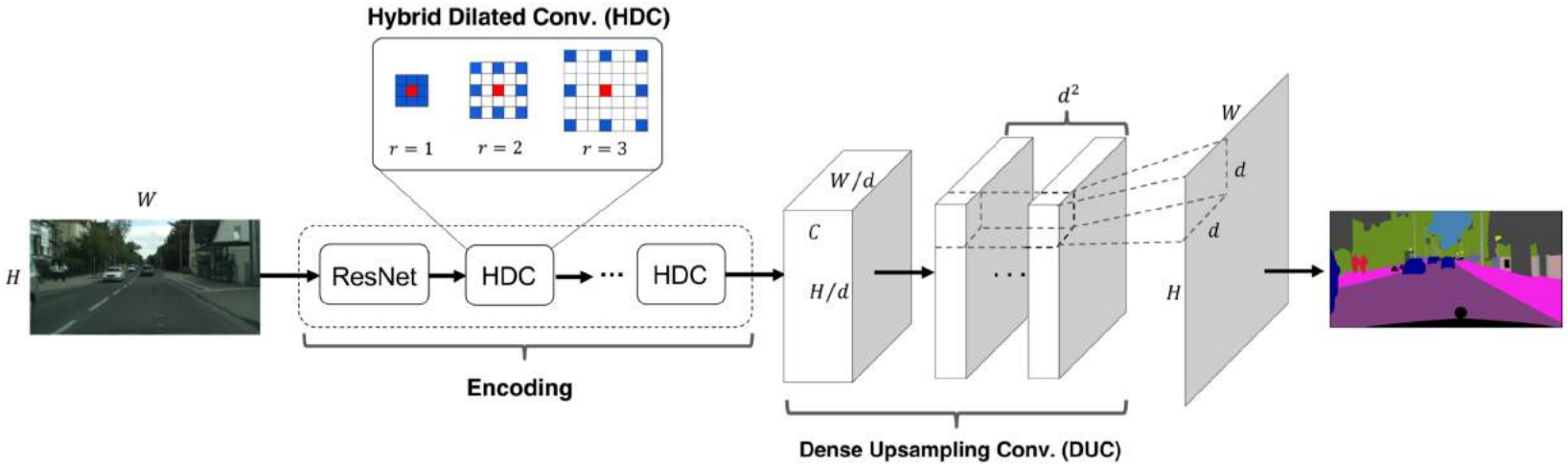
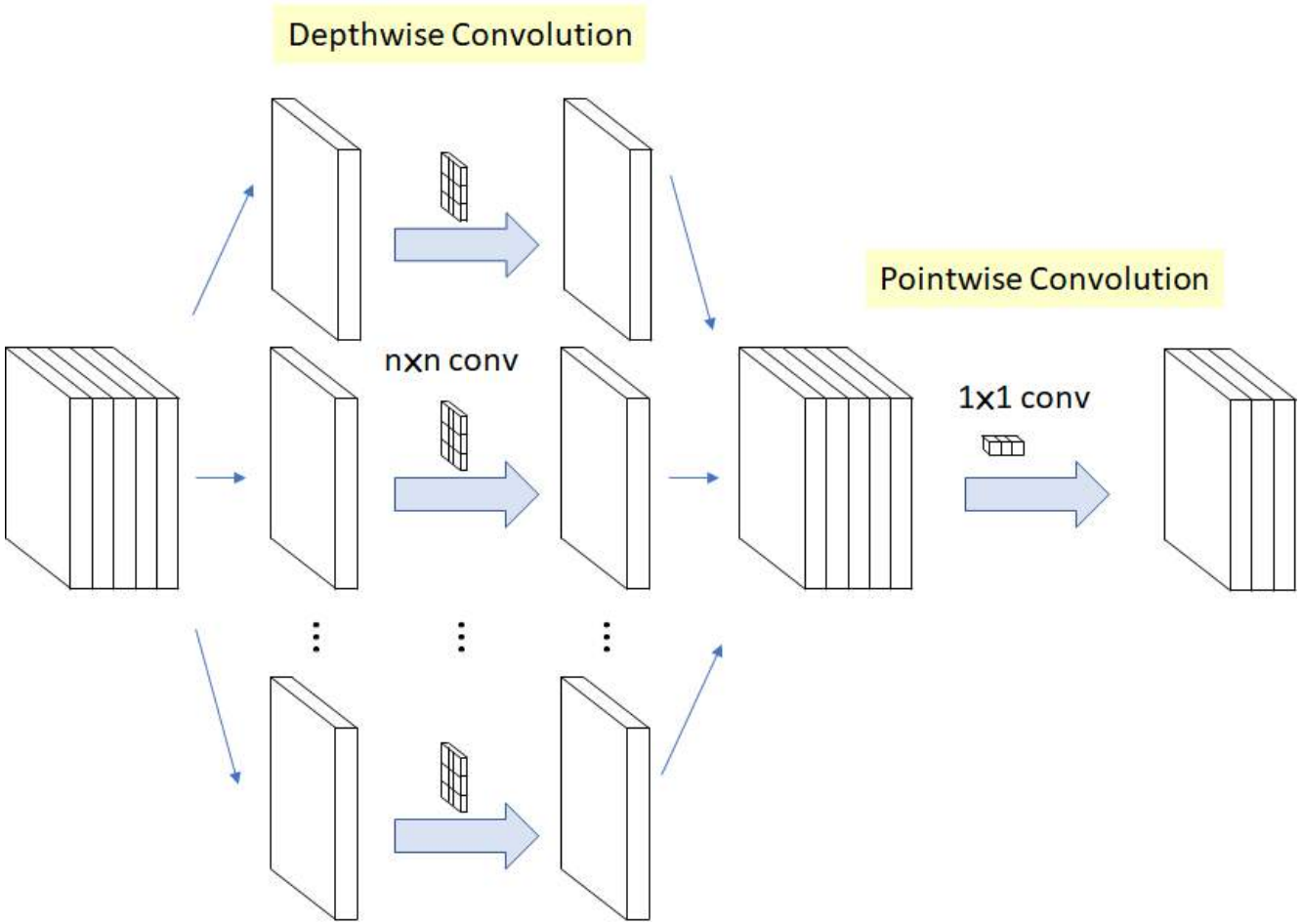
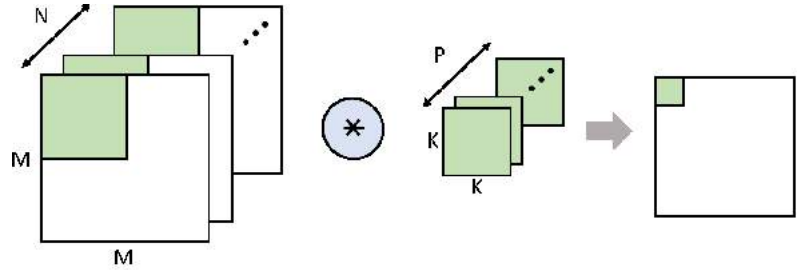




Figure 2. Overview of the proposed network architecture. In each layer, filter kernels are depicted in square with grid pattern. The blue cells in dilated kernel represent valid weights and blank cells represent invalid region. As shown, in case of kernels with dilation factor of 2, valid weights align by interval of 1 and in case of 3, they align by interval of 3.

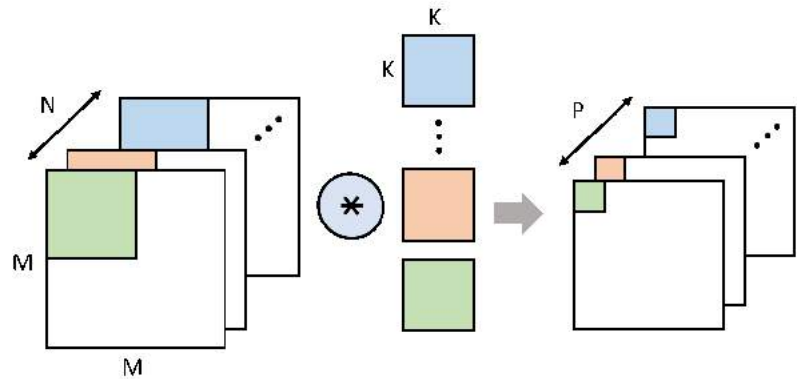


# Separate Convolutions, 2017

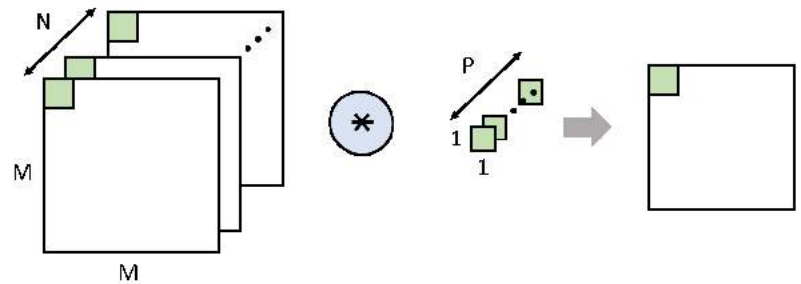



(a) standard convolution

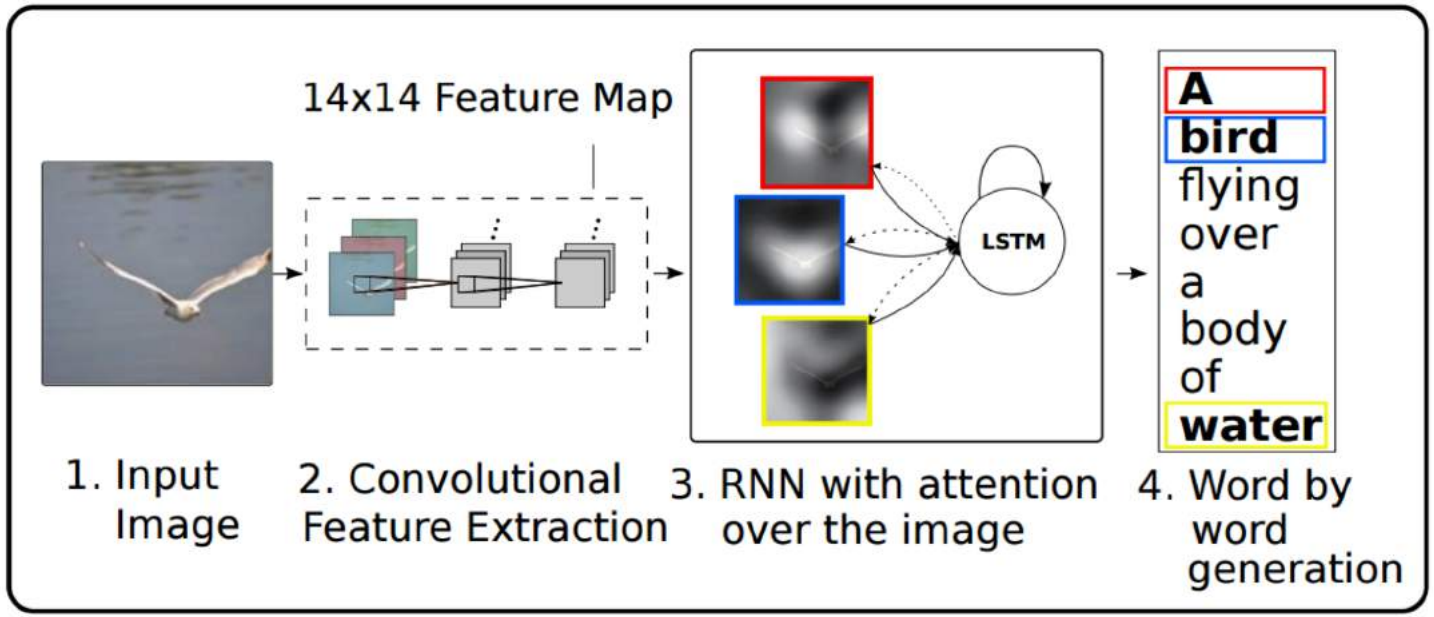



(b) depthwise convolution




(c) pointwise convolution

# Attention Mechanism, 2015




*"Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", 2015*

- **Idea is simple:** add a (learned) weighted mask to feature (feature selection)
- Use a feed-forward deep network to extract L feature vectors
- Use a recurrent network to iteratively update the attention (shown as bright regions) for each output word
- Obtain meaningful correspondences between words and attentions

# Examples of Visual Attention

A woman is throwing a frisbee in a park.

A dog is standing on a hardwood floor.

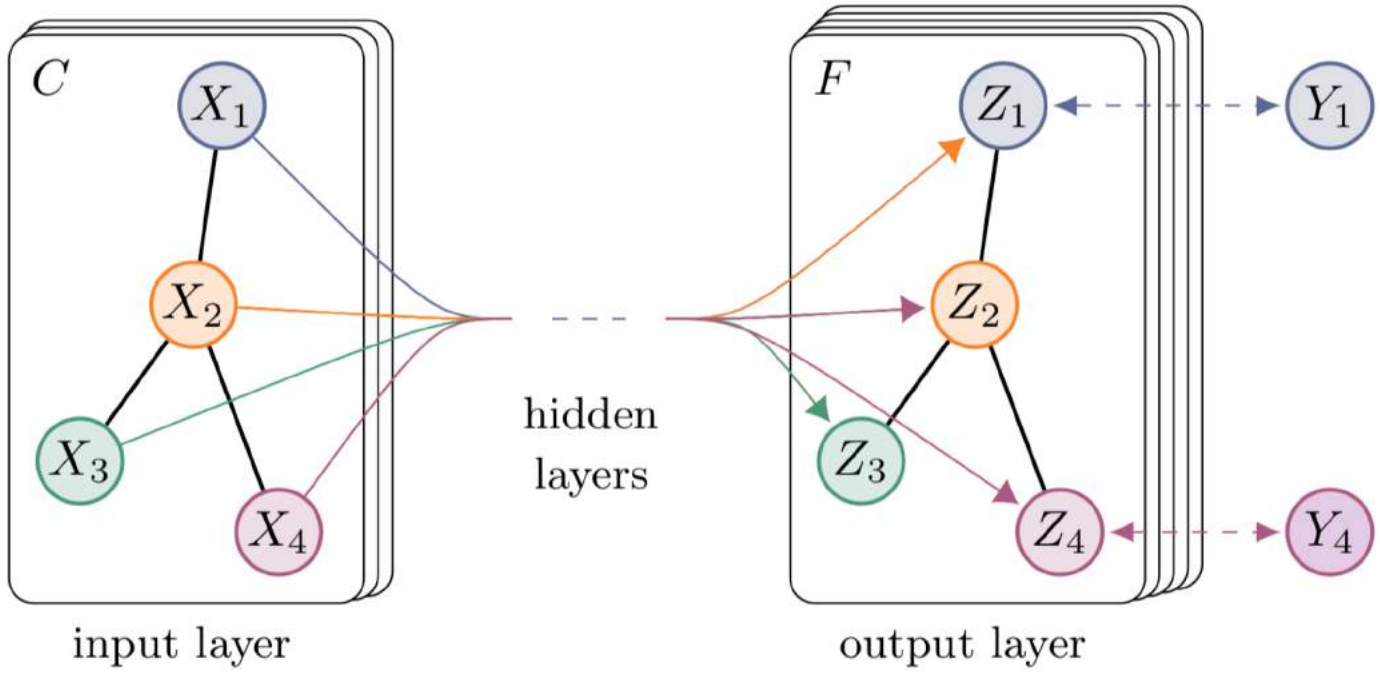

A stop sign is on a road with a mountain in the background.

A little girl sitting on a bed with a teddy bear.

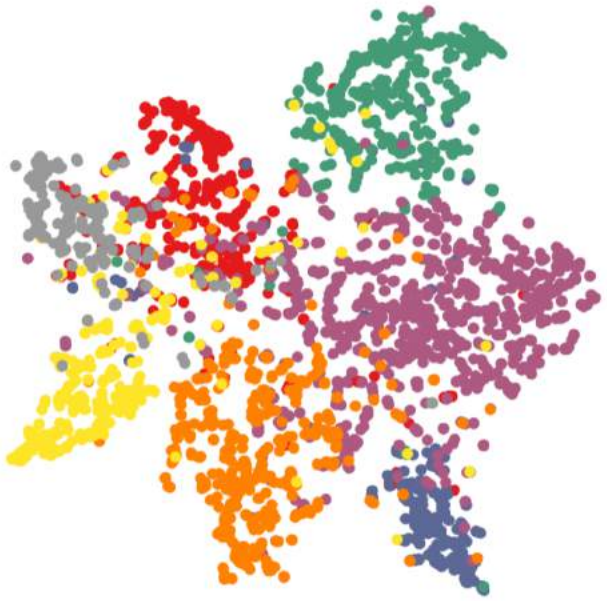
A group of people sitting on a boat in the water.

A giraffe standing in a forest with trees in the background.

# Graph Convolution Network (GCN), 2014




(a) Graph Convolutional Network



(b) Hidden layer activations

# Generative Models

# Deep Generative Models

- **Idea:** learn to understand data through generation

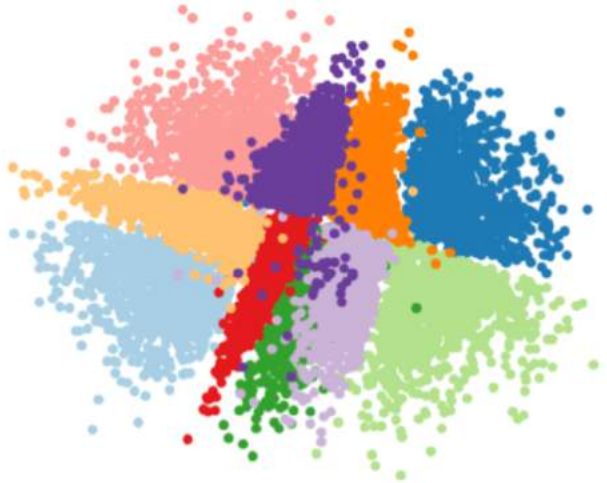
(c) Varying $c_2$ from −2 to 2 on InfoGAN (Rotation)

(a) Varying $c_1$ on InfoGAN (Digit type)

(d) Varying $c_3$ from −2 to 2 on InfoGAN (Width)

Chen et al. 2016

We have many orders of magnitude more data than labels. Hence **unsupervised learning** is important.



Sønderby et al. 2016

# Setup of Generative Models

**Discriminative model**: given *n* examples $(x^{(i)}, y^{(i)})$
learn $h : X \to Y$

**Generative model**: given *n* examples $x^{(i)}$, recover $p(x)$

**Maximum-likelihood objective**: $\prod_i p_\theta(x) = \sum_i \log p_\theta(x)$

**Generation:** Sampling from $p_\theta(x)$

# Autoregressive Models

**Factorize** dimension-wise:

$$p(x) = p(x_1)p(x_2|x_1)\dots p(x_n|x_1,\dots,x_{n-1})$$

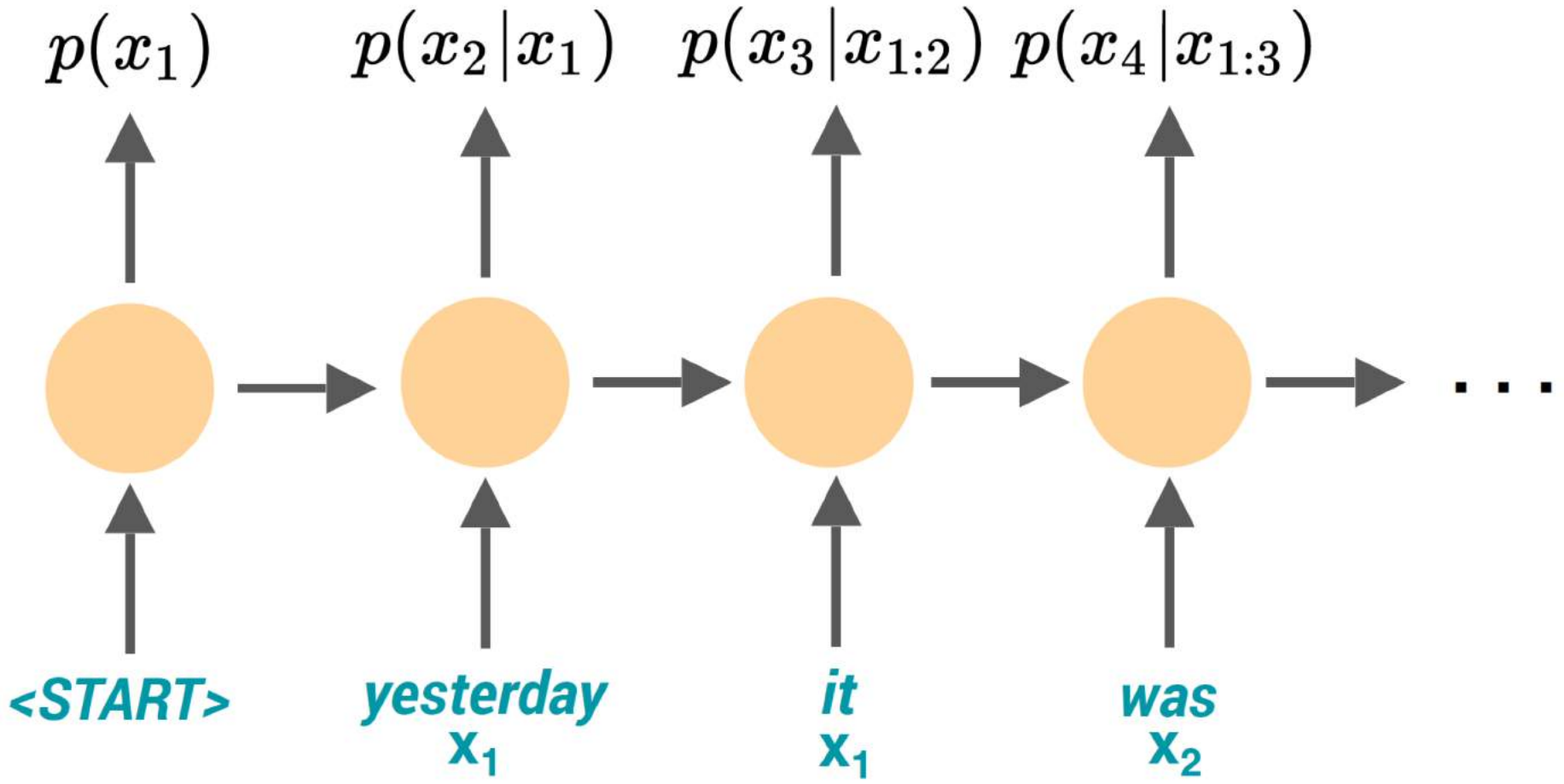
Build a "next-step prediction" model $p(x_n|x_1,\dots,x_{n-1})$

If x is **discrete**, network outputs a probability for each possible value

If x is **continuous**, network outputs parameters of a simple distribution (e.g. Gaussian mean and variance)… *or just discretize!*

**Generation:** sample one step at a time, conditioned on all previous steps

# RNNs for Autoregressive Language Modeling

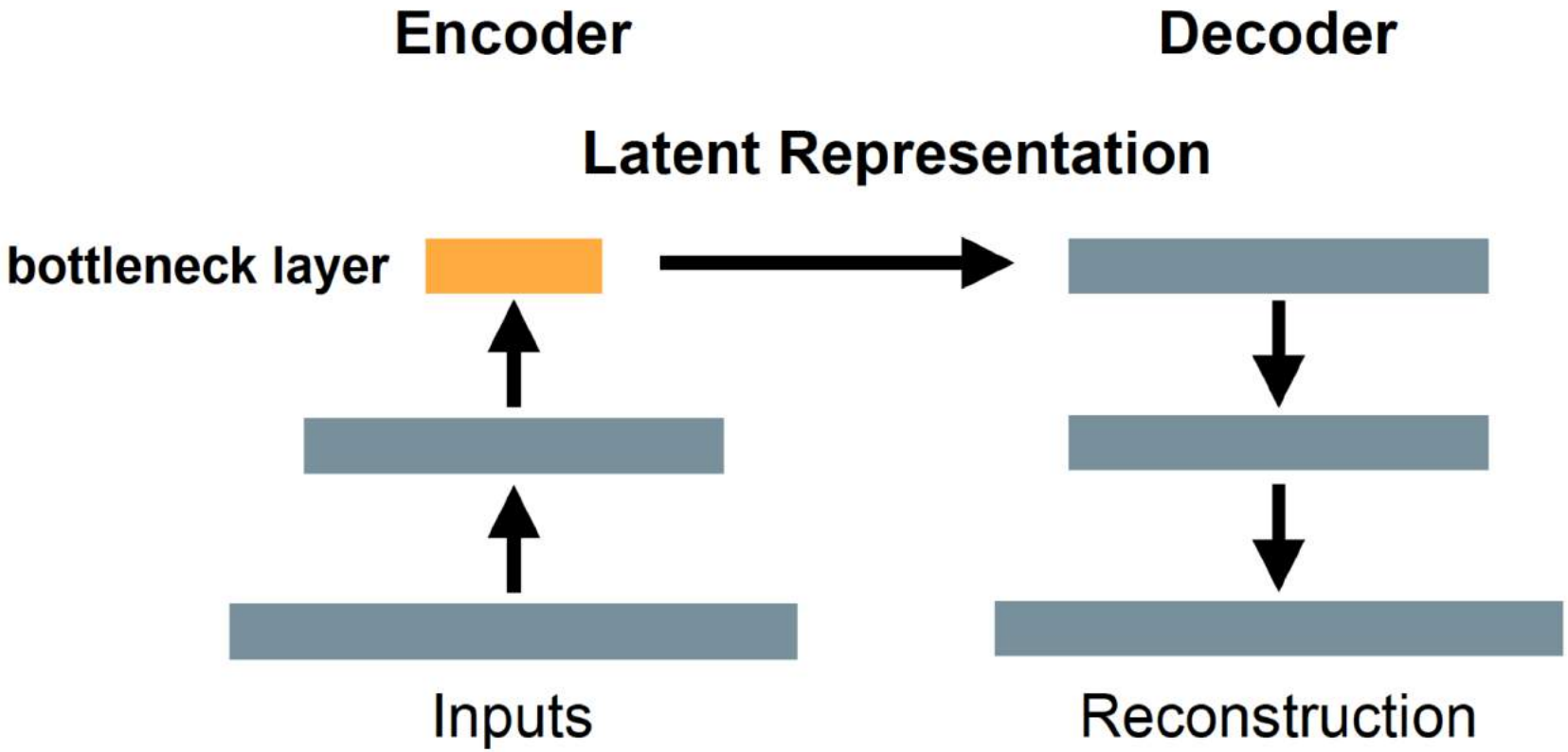


# PixelRNN (van der Oord et al. 2016)

- Autoregressive RNN over pixels in an image
- Models pixels as discrete-valued (256-way softmax at each step)

- **Problems:**
  - Sequential generation can be very slow
  - Not even close to the “true” image generating process

# Autoencoders for Representation Learning




$$L = (x - \hat{x})^2$$

**Idea:** compression as implicit generative modeling

# Variational Autoencoders (VAEs)

Generative extension of autoencoders which allow sampling and estimating probabilities

"Latent variables" with fixed prior distribution $p(z)$

Probabilistic encoder and decoder: $q(z|x), p(x|z)$

Trained to maximize a lower bound on log-probability:

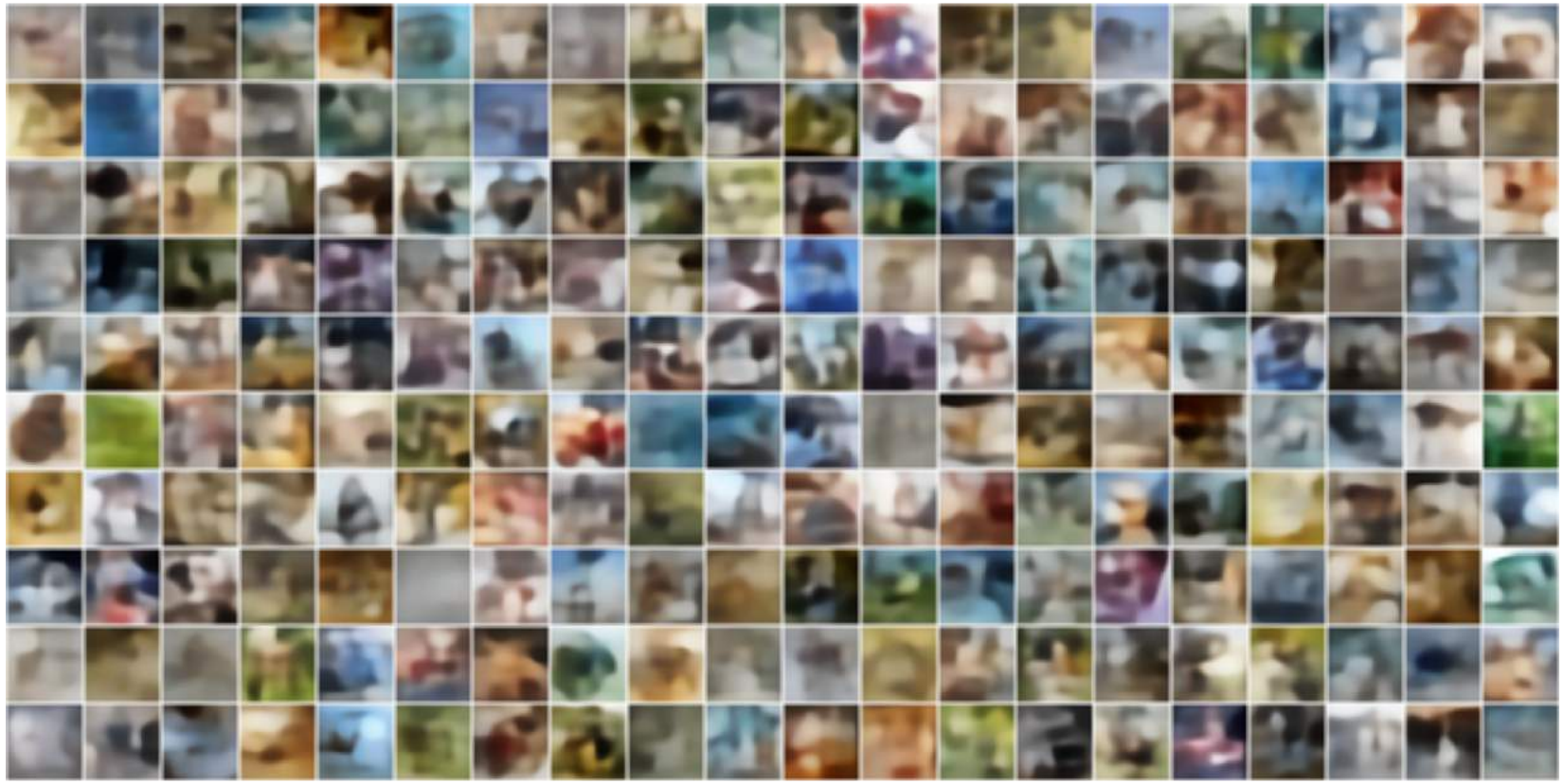
$$\log p(x) \geq \mathbb{E}_{z \sim q(z|x)} [\log p(x|z) + \log p(z) - \log q(z)]$$

# Variational Autoencoders (VAEs)

# Variational Autoencoders (VAEs)

Problems:

- Encoder and decoder's output distributions are typically limited (diagonal-covariance Gaussian or similar)
- This prevents the model from capturing fine details and leads to blurry generations

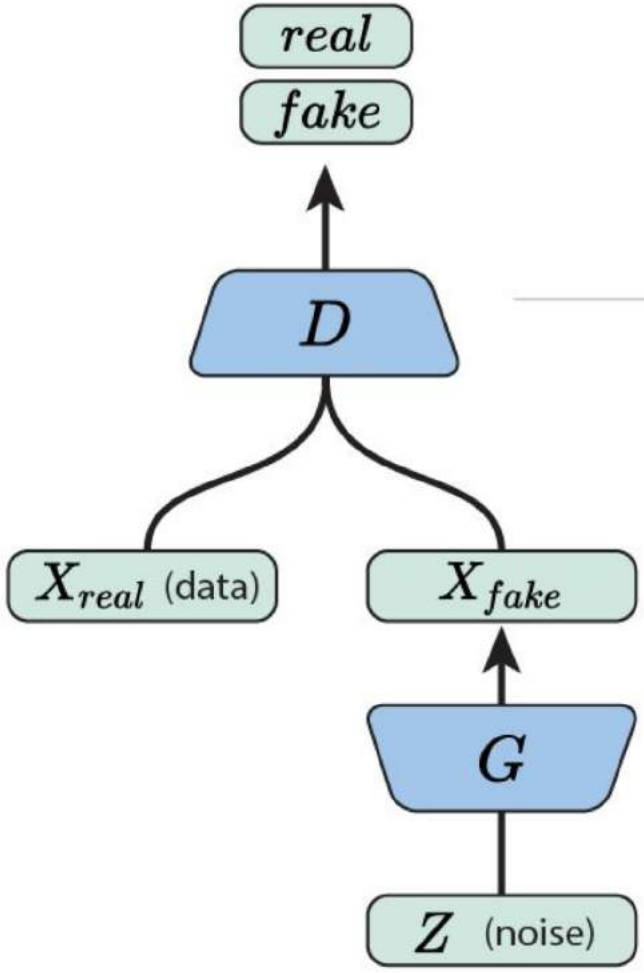


# The “New World”: GAN

**Generative Adversarial Networks** (GANs) are a way to make a generative model by having two neural networks compete with each other.

Problems:

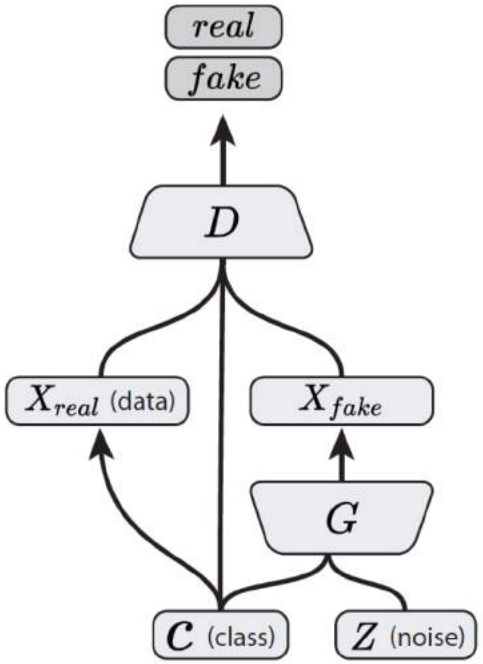
- Training is notoriously difficult and unstable (minimax optimization)
- Easily biased towards either G or D
- Still hard to generate real complicated images; good progress made though



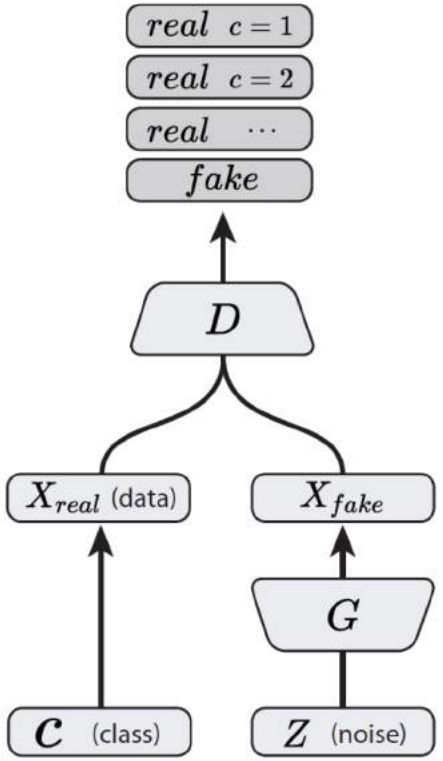

The **discriminator** tries to distinguish genuine data from forgeries created by the generator.

The **generator** turns random noise into immitations of the data, in an attempt to fool the discriminator.

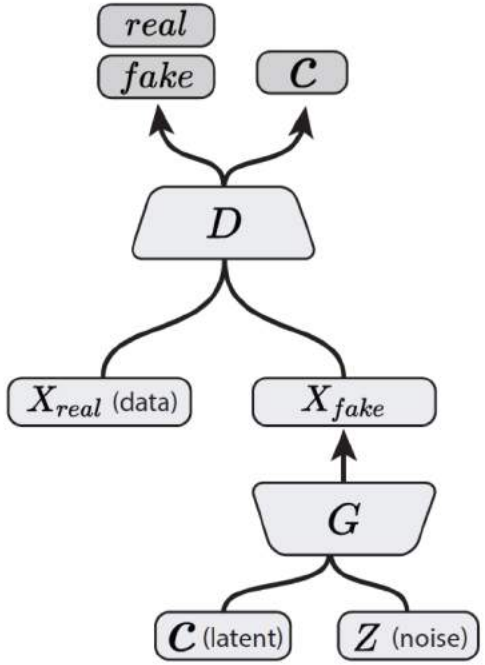
# From Unsupervised to Supervised: Conditional GAN



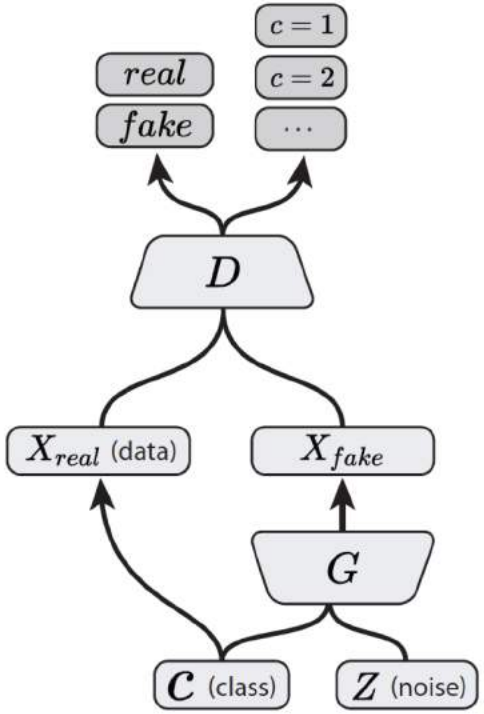

Conditional GAN
(Mirza & Osindero, 2014)




Semi-Supervised GAN
(Odena, 2016; Salimans, et al., 2016)




InfoGAN
(Chen, et al., 2016)




AC-GAN
(Present Work)

# How to Evaluate GANs?

- Quality of generated images: **Inception score (IS)**
- Diversity of generated images: **Fréchet Inception Distance (FID)**
- More criteria untouched yet:
  - Novelty of generated images?
  - Detection of co-variate shifts (privacy, fairness, etc.)?

# Spectral Normalization GAN (ICLR'2018)

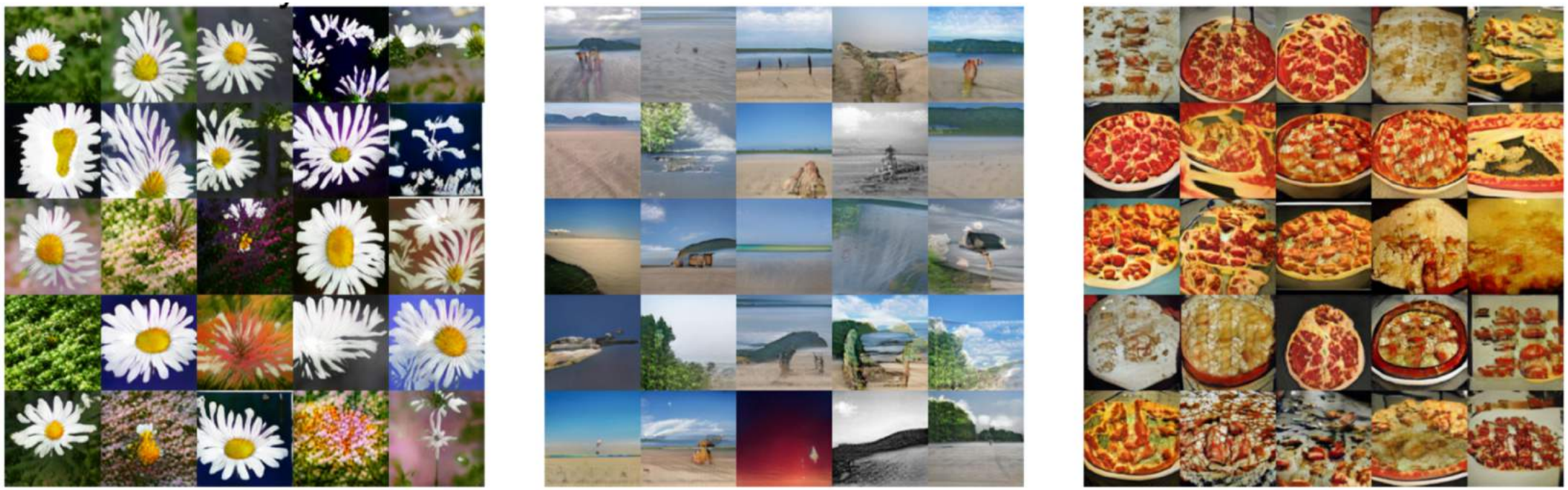


Figure 7: 128x128 pixel images generated by SN-GANs trained on ILSVRC2012 dataset. The inception score is 21.1±.35.

# More SOTA GANs

- Progressive GAN, ICLR 2018
- Self-Attention GAN, NeurIPS 2018
- StyleGAN, CVPR 2019
- BigGAN, ICLR 2019
- AutoGAN, ICCV, 2019
- …. ***But understand it's quite limited!!***


*The research of GAN remains to be a very active frontier, at both theoretical understanding and empirical design sides.*

# Optimization Algorithms

Where the magic happens

# Gradient Descent (GD)

**Algorithm 1** Batch Gradient Descent at Iteration $k$

**Require:** Learning rate $\epsilon_k$

**Require:** Initial Parameter $\theta$

1: **while** stopping criteria not met **do**

2: Compute gradient estimate over $N$ examples:

3: $\hat{\mathbf{g}} \leftarrow +\frac{1}{N}\nabla_\theta \sum_i L(f(\mathbf{x}^{(i)};\theta), \mathbf{y}^{(i)})$

4: Apply Update: $\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$

5: **end while**

- Positive: Gradient estimates are stable
- Negative: Need to compute gradients over the entire training for one update

# Stochastic Gradient Descent (SGD)

**Algorithm 2** Stochastic Gradient Descent at Iteration $k$

**Require:** Learning rate $\epsilon_k$

**Require:** Initial Parameter $\theta$

1: **while** stopping criteria not met **do**

2: Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set

3: Compute gradient estimate:

4: $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

5: Apply Update: $\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$

6: **end while**

- $\epsilon_k$ is learning rate at step $k$
- Sufficient condition to guarantee convergence:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty \text{ and } \sum_{k=1}^{\infty} \epsilon_k^2 < \infty$$

# GD versus SGD

- Batch Gradient Descent:

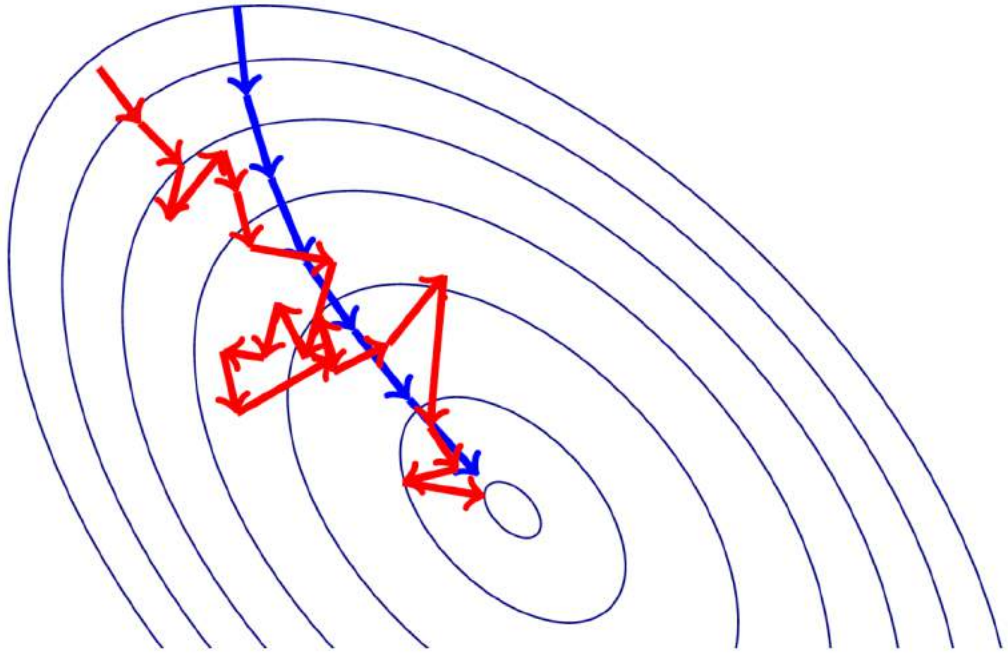
$$\hat{\mathbf{g}} \leftarrow +\frac{1}{N}\nabla_\theta \sum_i L(f(\mathbf{x}^{(i)};\theta),\mathbf{y}^{(i)})$$

$$\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$$

- SGD:

$$\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)};\theta),\mathbf{y}^{(i)})$$

$$\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$$



# Minibatch

- Potential Problem: Gradient estimates can be very noisy
- Obvious Solution: Use larger mini-batches (In theory, growingly larger)

- Advantage: Computation time per update does not depend on number of training examples.
- This allows convergence on extremely large datasets

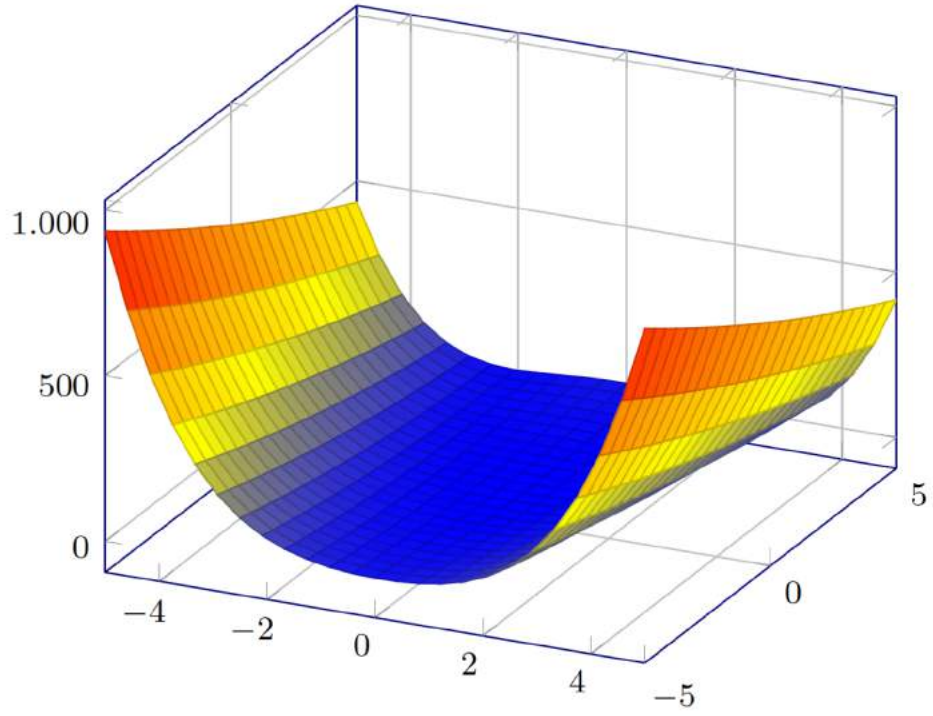
- **Empirically/theoretically, the larger MB size the better!!**

*"Large Scale Learning with Stochastic Gradient Descent", Leon Bottou.*

# Momentum

- The Momentum method is a method to accelerate learning using SGD

- In particular SGD suffers in the following scenarios:
  - Error surface has high curvature
  - Small but consistent gradients
  - Noisy gradients




- Gradient Descent would move quickly down the walls, but very slowly through the valley floor

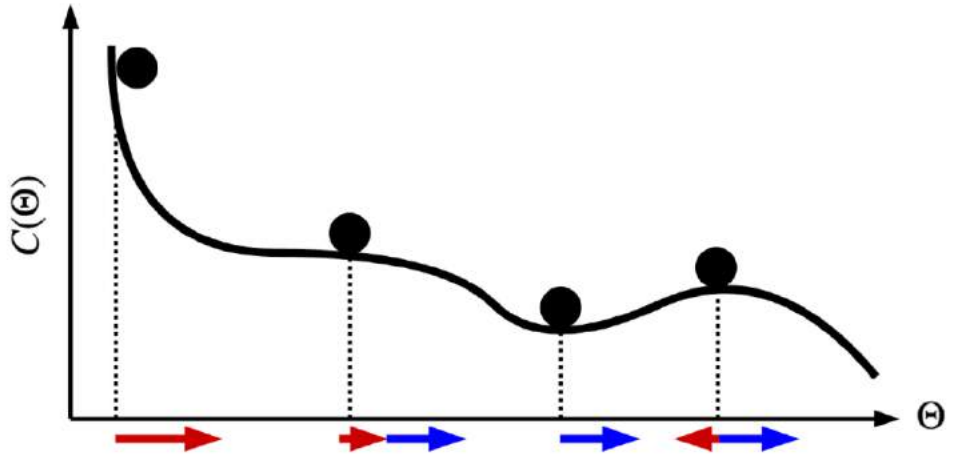
# Momentum

- Update rule in SGD:

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta \boldsymbol{g}^{(t)}$$

where $\boldsymbol{g}^{(t)} = \nabla_{\Theta} C(\Theta^{(t)})$

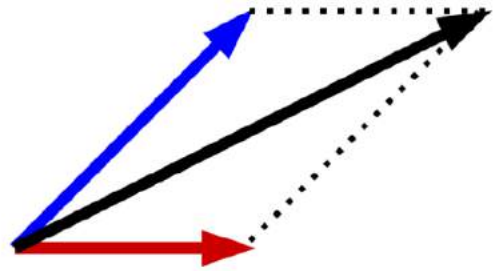
  - Gets stuck in local minima or saddle points




- Momentum: make the same movement $\boldsymbol{v}^{(t)}$ in the last iteration, corrected by negative gradient:

$$\boldsymbol{v}^{(t+1)} \leftarrow \lambda \boldsymbol{v}^{(t)} - (1-\lambda)\boldsymbol{g}^{(t)}$$

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} + \eta \boldsymbol{v}^{(t+1)}$$




- $\boldsymbol{v}^{(t)}$ is a moving average of $-\boldsymbol{g}^{(t)}$

# Adaptive Learning Rate Optimization

- Popular Solver Examples: AdGrad, RMSProp, Adam

SGD: $\theta \leftarrow \theta - \epsilon \hat{\mathbf{g}}$

Momentum: $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \hat{\mathbf{g}}$ then $\theta \leftarrow \theta + \mathbf{v}$

Nesterov: $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_\theta \left( L(f(\mathbf{x}^{(i)}; \theta + \alpha \mathbf{v}), \mathbf{y}^{(i)}) \right)$ then $\theta \leftarrow \theta + \mathbf{v}$

AdaGrad: $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{g} \odot \mathbf{g}$ then $\Delta\theta - \leftarrow \frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g}$ then $\theta \leftarrow \theta + \Delta\theta$

RMSProp: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho) \hat{\mathbf{g}} \odot \hat{\mathbf{g}}$ then $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$ then $\theta \leftarrow \theta + \Delta\theta$

Adam: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \rho_1^t}, \hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1 - \rho_2^t}$ then $\Delta\theta = -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}} + \delta}$ then $\theta \leftarrow \theta + \Delta\theta$

# AdaGrad

- **Idea:** Downscale a model parameter by square-root of sum of squares of all its historical values
- Parameters that have large partial derivative of the loss -> learning rates for them are rapidly declined
- Some interesting theoretical properties

**Algorithm 4** AdaGrad

**Require:** Global Learning rate $\epsilon$, Initial Parameter $\theta$, $\delta$

Initialize $\mathbf{r} = 0$

1: **while** stopping criteria not met **do**
2: Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set
3: Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$
4: Accumulate: $\mathbf{r} \leftarrow \mathbf{r} + \hat{\mathbf{g}} \odot \hat{\mathbf{g}}$
5: Compute update: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$
6: Apply Update: $\theta \leftarrow \theta + \Delta\theta$
7: **end while**

# RMSProp

- AdaGrad might shrink the learning rate too aggressively, we can adapt it to perform better by accumulating an exponentially decaying average of the gradient

**Algorithm 5** RMSProp

**Require:** Global Learning rate $\epsilon$, decay parameter $\rho$, $\delta$

Initialize $\mathbf{r} = 0$

1: **while** stopping criteria not met **do**
2: Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set
3: Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$
4: Accumulate: $\mathbf{r} \leftarrow \rho\mathbf{r} + (1-\rho)\hat{\mathbf{g}} \odot \hat{\mathbf{g}}$
5: Compute update: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta+\sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$
6: Apply Update: $\theta \leftarrow \theta + \Delta\theta$
7: **end while**

# Adam

- Adam is like RMSProp with Momentum but with bias correction terms for the first and second moments

**Algorithm 7** RMSProp with Nesterov

**Require:** $\epsilon$ (set to 0.0001), decay rates $\rho_1$ (set to 0.9), $\rho_2$ (set to 0.9), $\theta$, $\delta$

Initialize moments variables $\mathbf{s} = 0$ and $\mathbf{r} = 0$, time step t $= 0$

1: **while** stopping criteria not met **do**
2: Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set
3: Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$
4: $t \leftarrow t + 1$
5: Update: $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1)\hat{\mathbf{g}}$
6: Update: $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2)\hat{\mathbf{g}} \odot \hat{\mathbf{g}}$
7: Correct Biases: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1-\rho_1^t}, \hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1-\rho_2^t}$
8: Compute Update: $\Delta\theta = -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}}+\delta}$
9: Apply Update: $\theta \leftarrow \theta + \Delta\theta$
10: **end while**

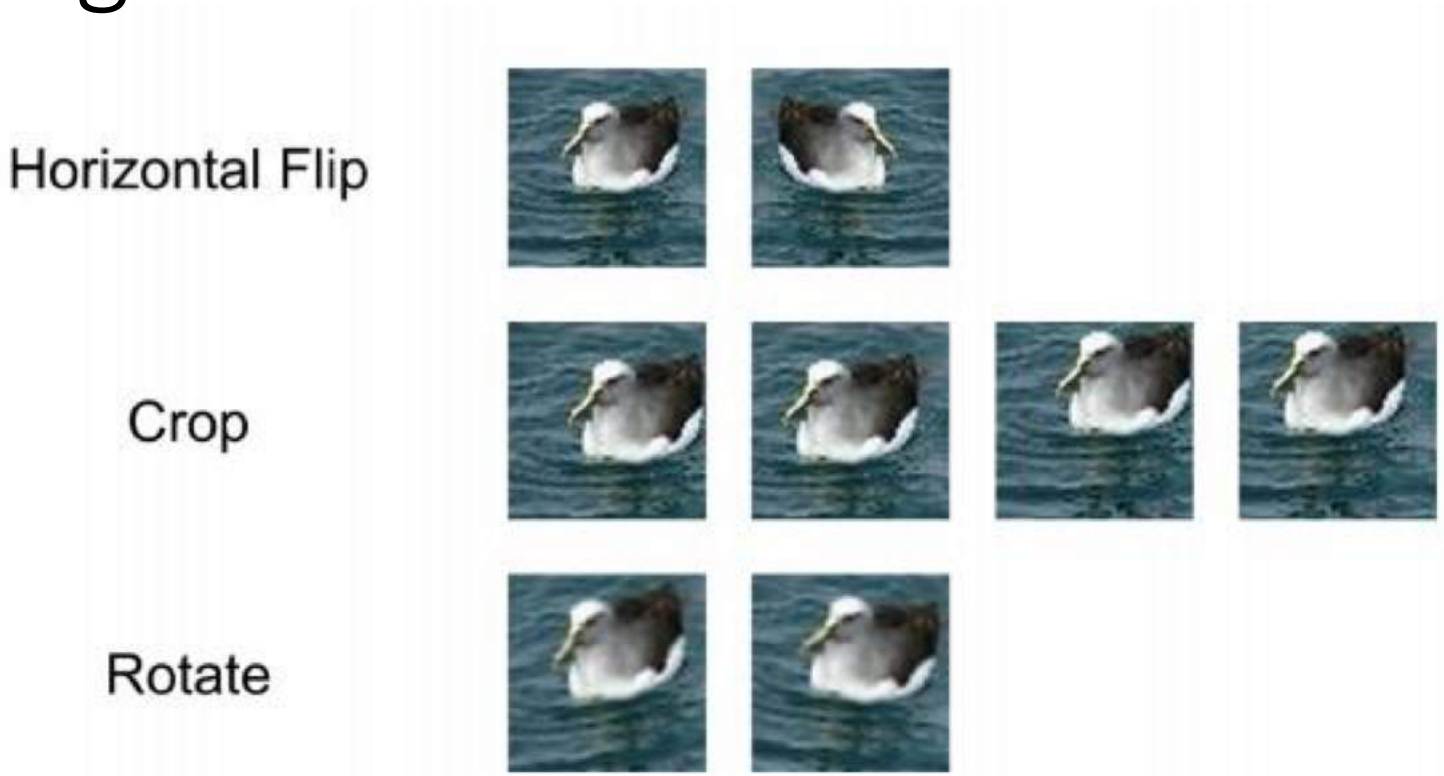
# Training Techniques and ”tricks”

Where the “black magic” happens

# Regularization in Deep Learning

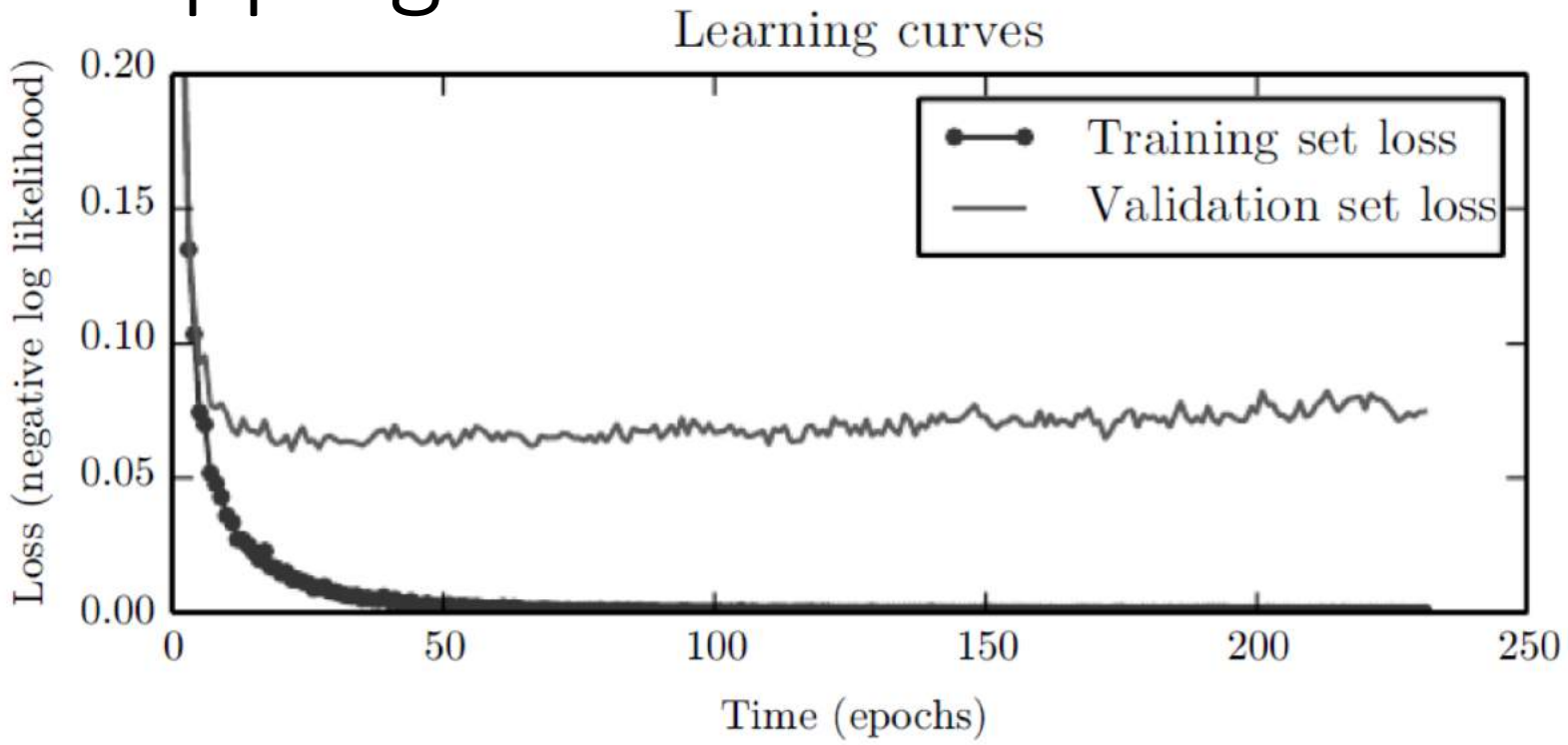
- Norm penalty/weight decay: $l_2$, $l_1$
- Data Augmentation
- Early Stopping
- Random Pruning: Dropout, Dropconnect, etc.
- Batch Normalization
- Structured Weights
- ……

# Data Augmentation




- Adding noise to the input: a special kind of augmentation
- Be careful about the transformation applied -> label preserving
  - **Example:** classifying ‘b’ and ‘d’; ‘6’ and ‘9’

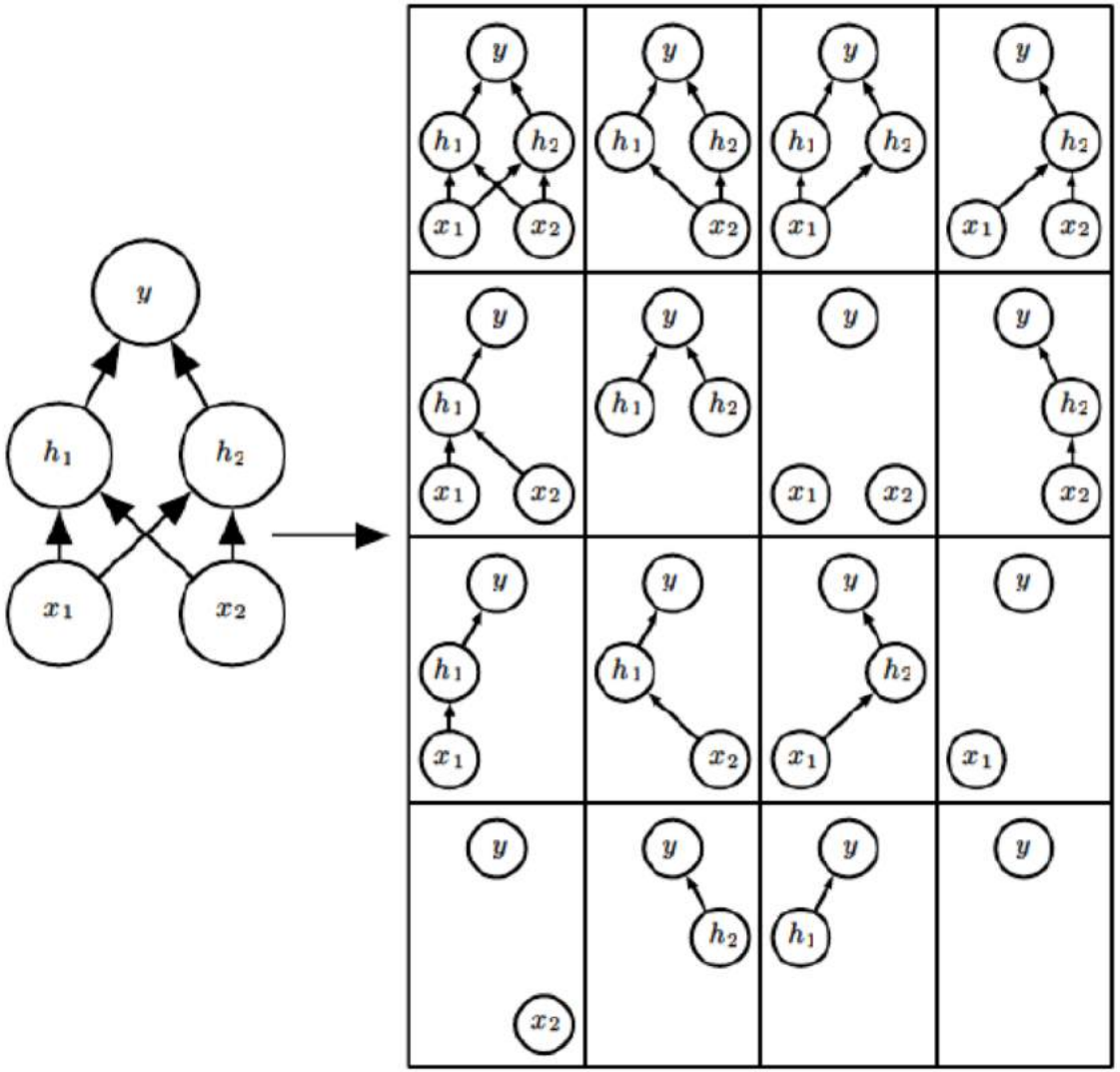
# Early Stopping




- Idea: don't train the network to too small training error
- Recall overfitting: Larger the hypothesis class, easier to find a hypothesis that fits the difference between the two
- Prevent overfitting: use validation error to decide when to stop

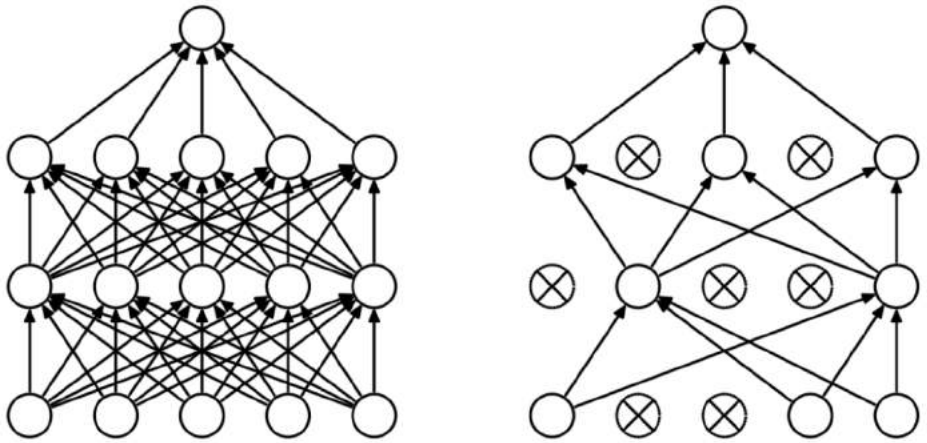
# Dropout

- Randomly select weights to update
  - In each update step, randomly sample a different binary mask to all the input and hidden units
  - Multiple the mask bits with the units and do the update as usual
  - Typical dropout probability: 0.2 for input and 0.5 for hidden units
  - Very useful for FC layers, less for conv layers, not useful in RNNs



# Dropout: A Stochastic Ensemble

- ***Dropout***: a feature-based bagging
  - Resamples input as well as ***latent*** features
  - With ***parameter sharing*** among voters



- SGD training: each time loading a minibatch, randomly sample a binary mask to apply to all input and hidden units
  - Each unit has probability $\alpha$ to be included (a hyperparameter)
  - Typically, $0.8$ for input units and $0.5$ for hidden units
- Different minibatches are used to train different parts of the NN
  - Similar to bagging, but much more efficient
  - No need to retrain unmasked units
  - Exponential number of voters

# Batch Normalization

- In ML, we assume future data will be drawn from same probability distribution as training data
- For a hidden layer, after training, the earlier layers have new weights and hence may generate a new distribution for the next hidden layer
- We want to reduce this internal covariate shift for the benefit of later layers

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

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

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

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

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

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

**Algorithm 1:** Batch Normalizing Transform, applied to activation $x$ over a mini-batch.

# Batch Normalization

- First three steps are just like standardization of input data, but with respect to only the data in mini-batch.
- We can take derivative and incorporate the learning of last step parameters into backpropagation.
- Note last step can completely un-do previous 3 steps
- But even if so, this un-doing is driven by the later layers, not the earlier layers; later layers get to “choose” whether they want standard normal inputs or not

# Data Pre-processing

- **Simplest: zero-center** the data, and then **normalize** them
  - It only makes sense to apply this pre-processing if you have a reason to believe that different input features have different scales (or units), but they should be of approximately equal importance to the learning algorithm.
  - In case of images, the relative scales of pixels are already approximately equal (and in range from 0 to 255), so it is not strictly necessary to perform this additional pre-processing step.

- **PCA Whitening**: the data is first centered, and then projected into the eigen basis to decorrelate the data, followed by dividing every dimension by the corresponding eigenvalue to normalize the scale.
- Never forget to **shuffle** your data each epoch for SGD input!
- Modern CNNs for image/video data usually do not need them, or need just zero-centered data. However, for other modalities it's case by case.

# Weight Initialization

- All Zero Initialization: Terribly Wrong!
  - If every neuron in the network computes the same output, then they will also all compute the same gradients during back-propagation and undergo the exact same parameter updates.

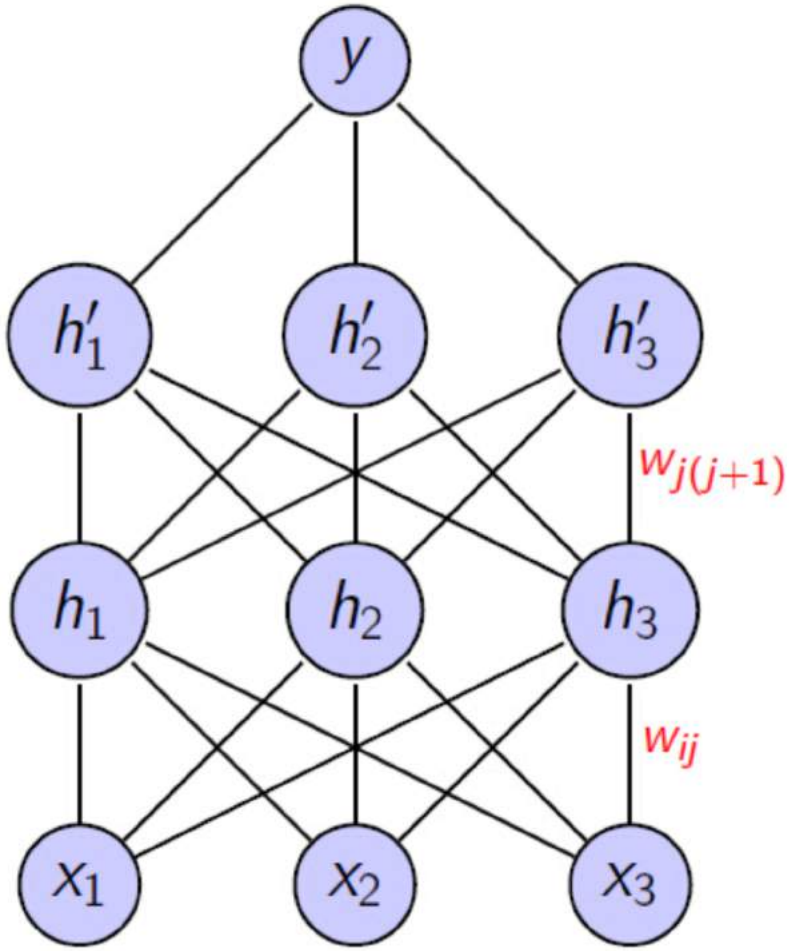
- Small Random Initialization
- Current recommendation for initializing CNNs with RELU:

$$w = np.random.randn(n) * sqrt(2.0/n)$$

- "randn": Gaussian; "n": the number of inputs for current layer.

- For reconstruction-type CNNs, layer-wise pre-training is safe.
- Even safer: start from a pre-trained model

# Why are deep architectures hard to train?




- Vanishing gradient problem in backward propagation

$$\frac{\partial Loss}{\partial w_{ij}} = \frac{\partial Loss}{\partial in_j}\frac{\partial in_j}{\partial w_{ij}} = \delta_j x_i$$

$$\delta_j = \left[\sum_{j+1} \delta_{j+1} w_{j(j+1)}\right] \sigma'(in_j)$$

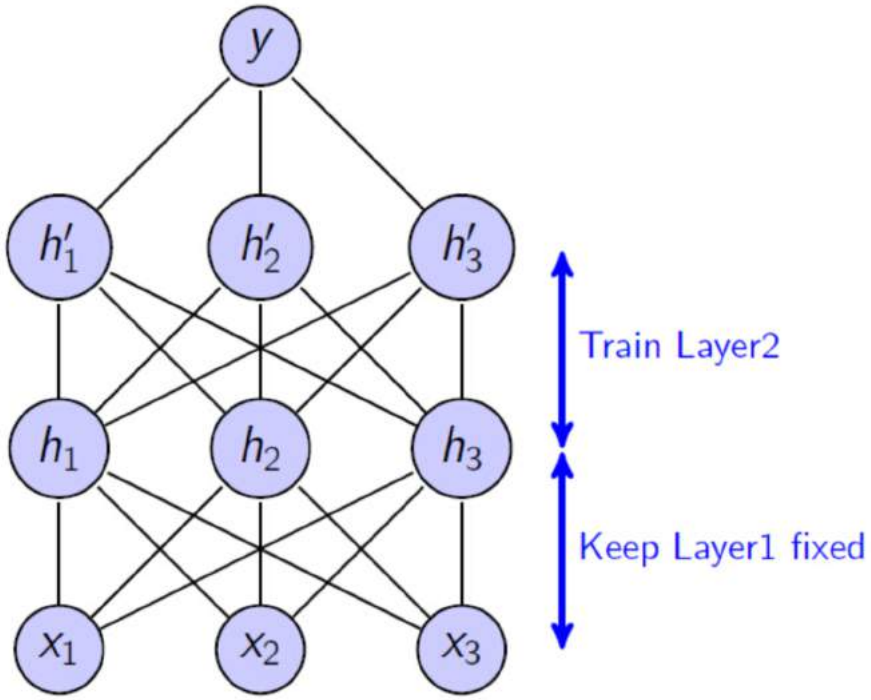
$\delta_j$ may vanish after repeated multiplication

- Insufficient training data

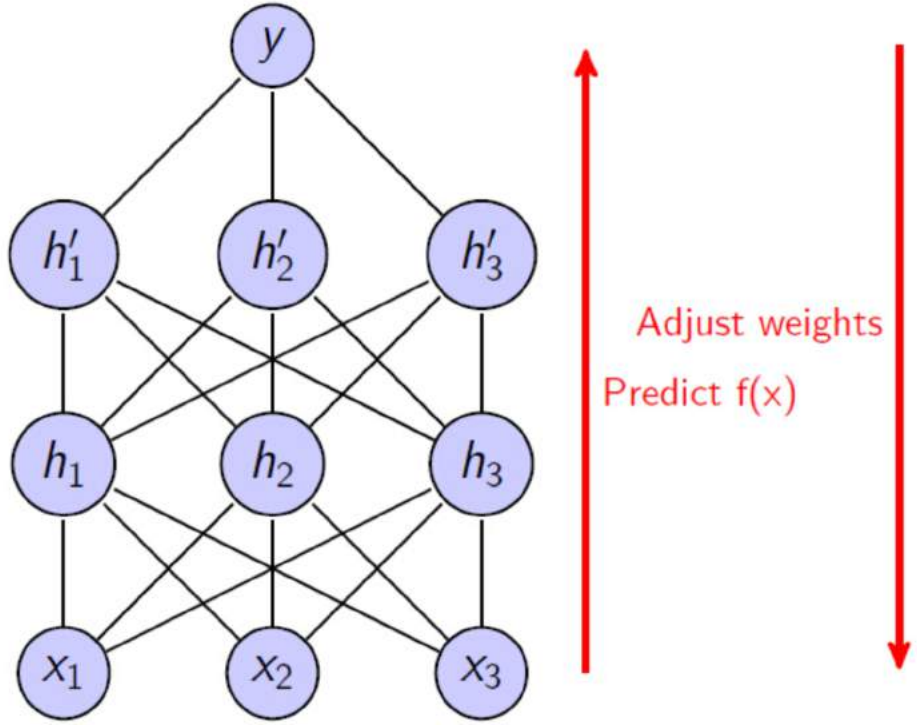
**BP may be too simple an optimization way …**

# Layer-wise Pre-training [Hinton et al. 2006]

First, train one layer at a time, optimizing data-likelihood objective $P(x)$




Finally, fine-tune labeled objective $P(y|x)$ by Backpropagation



# Fine-tuning Pre-trained Model

- Deep features are fairly transferable, and open-source pre-trained models are now everywhere.

| | **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 large number of layers |

- For datasets, *Caltech-101* is similar to *ImageNet*, where both two are object-centric image data sets; while *Place Database* is different from *ImageNet*, where one is scene-centric and the other is object-centric.

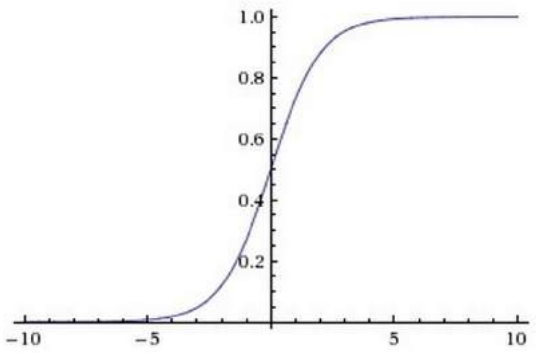
# Hyperparameter Choice

- **Mini-batch size:** preferably to be power of 2

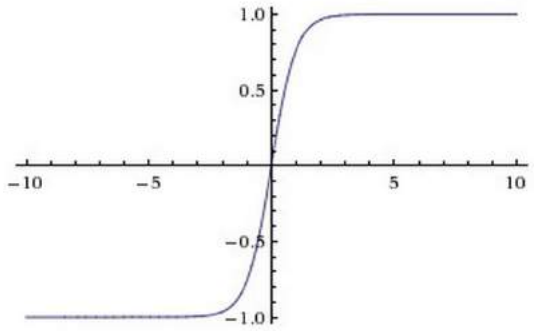
- **Filter size, pooling size, padding**
  - **Recommended** (but no universal best setting exists: always try some different!): small filter (e.g., 3×3, 1×1), small stride (e.g., 1), small max pooling (e.g., 2×2), zero-padding
  - Important to preserve the spatial size of feature maps when going deep

- **Learning rate**
  - **Popular Learning rate scheduling:** if you see that you stopped making progress on the validation set, divide the LR by 2 (or by 5), and keep going, which might give you a surprise.
  - More research: SGD re-start, cyclical learning rate, etc.

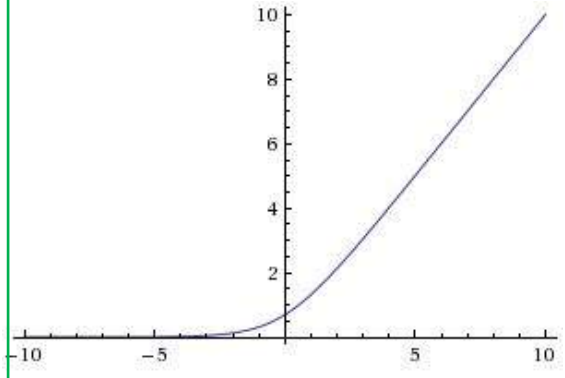
# Choice of Activation Functions



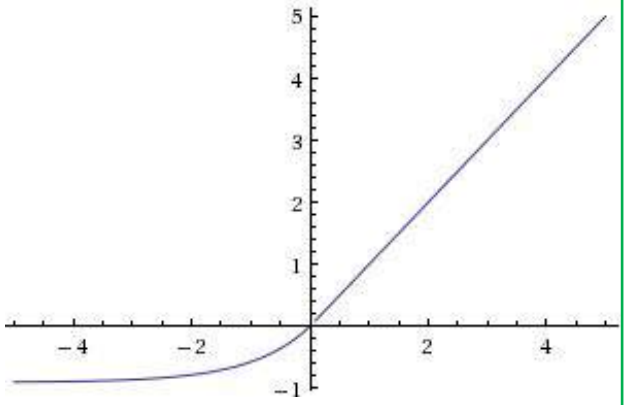

Sigmoid



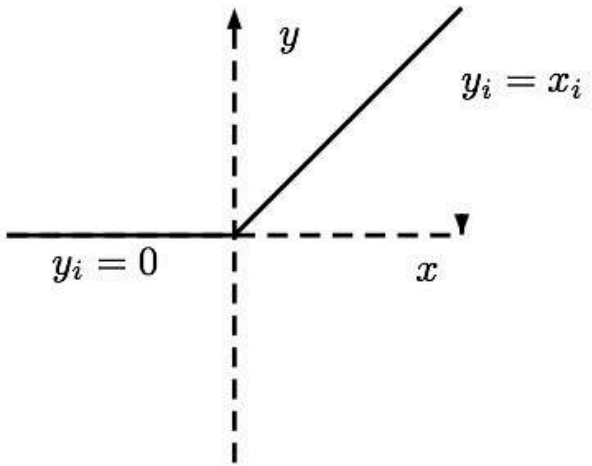

Tanh



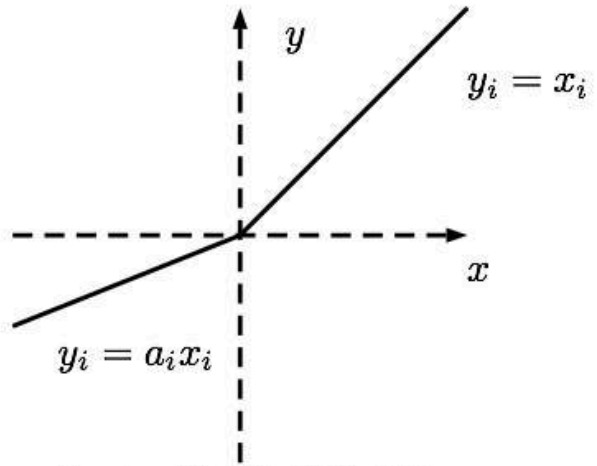

Softplus



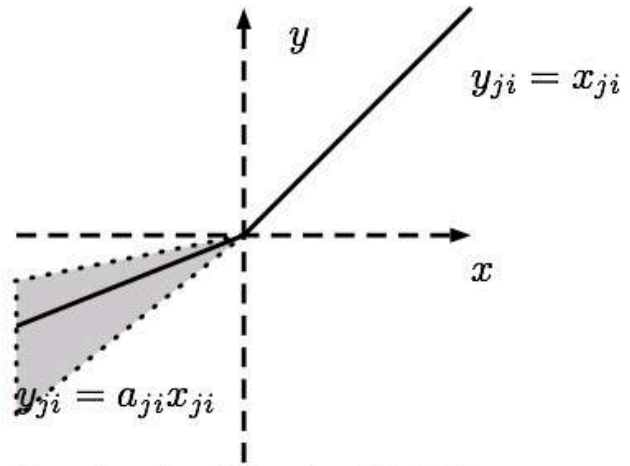

ELU




ReLU




Leaky ReLU/PReLU




Randomized Leaky ReLU

# Monitor Your Training Curve

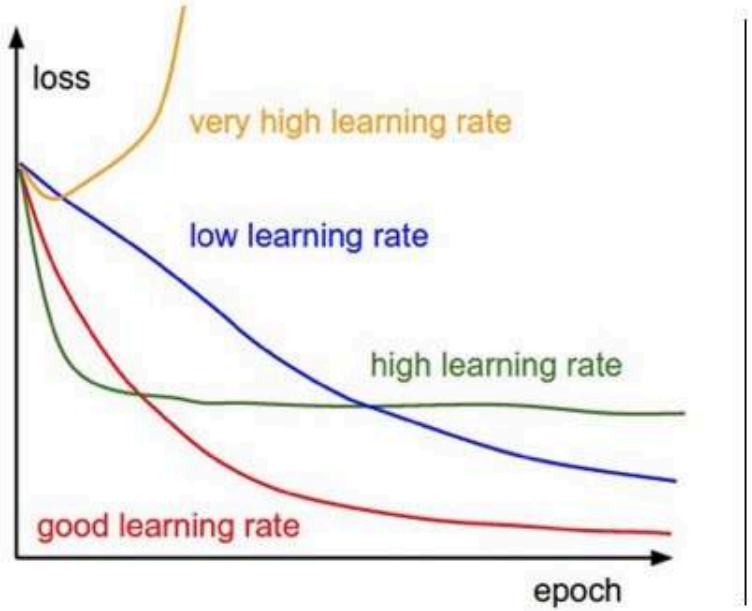
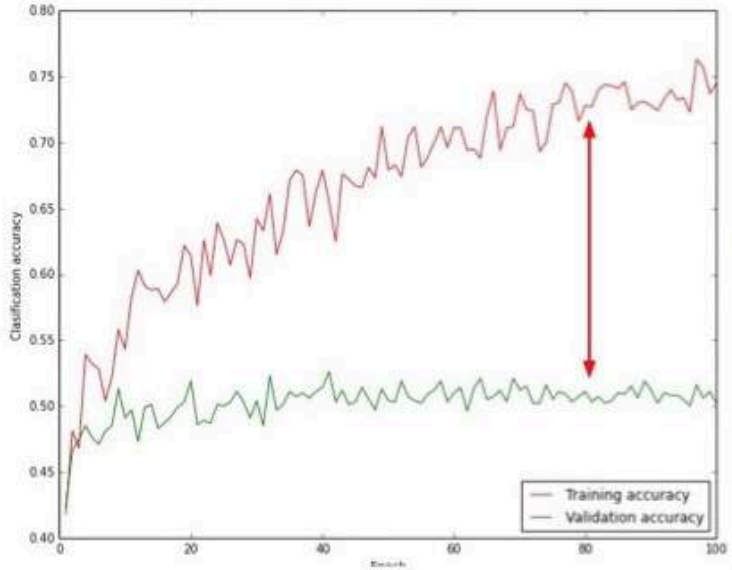



Figure 1




big gap = overfitting
=> increase regularization strength

no gap
=> increase model capacity

Figure 3

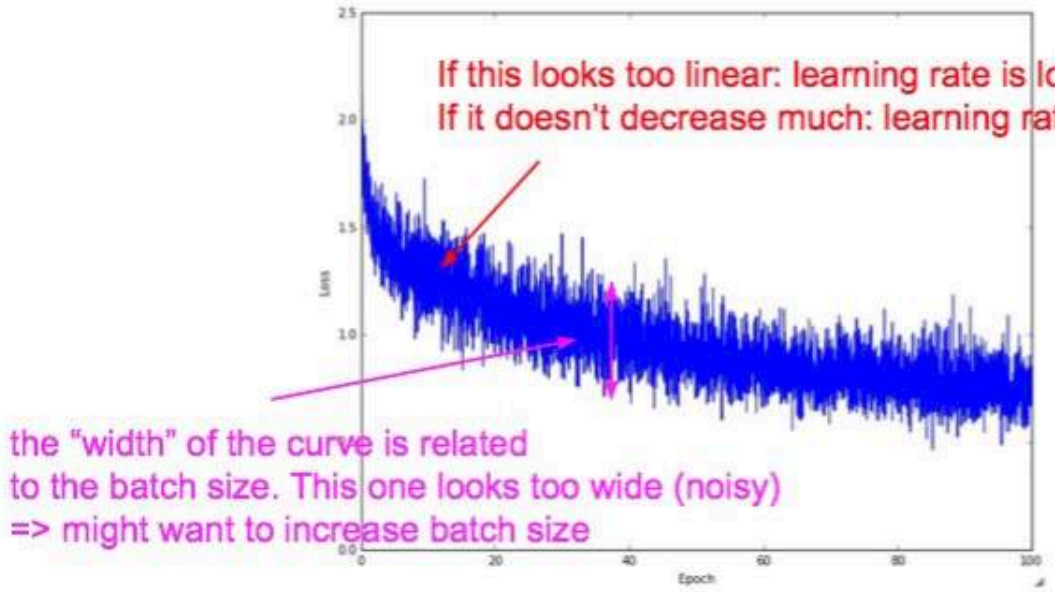
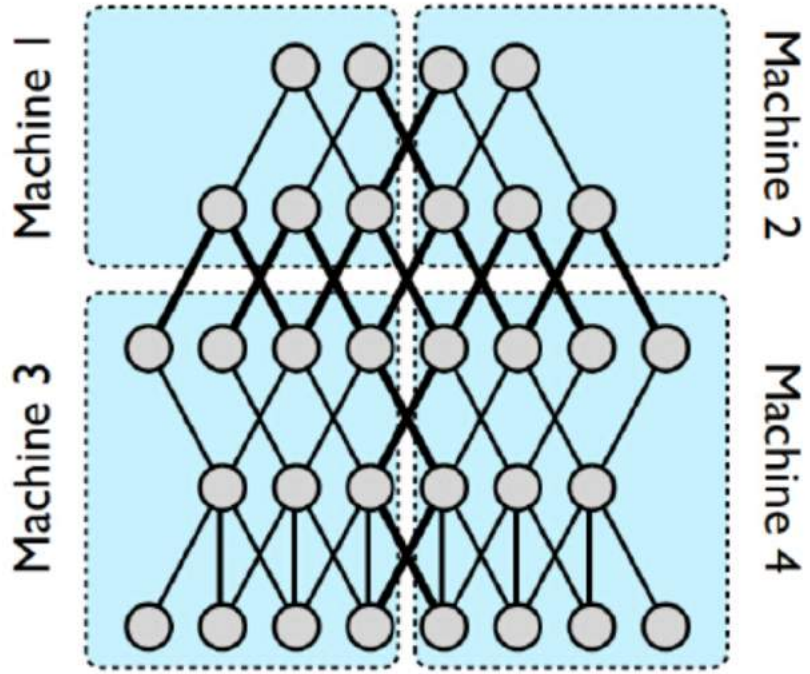



Figure 2

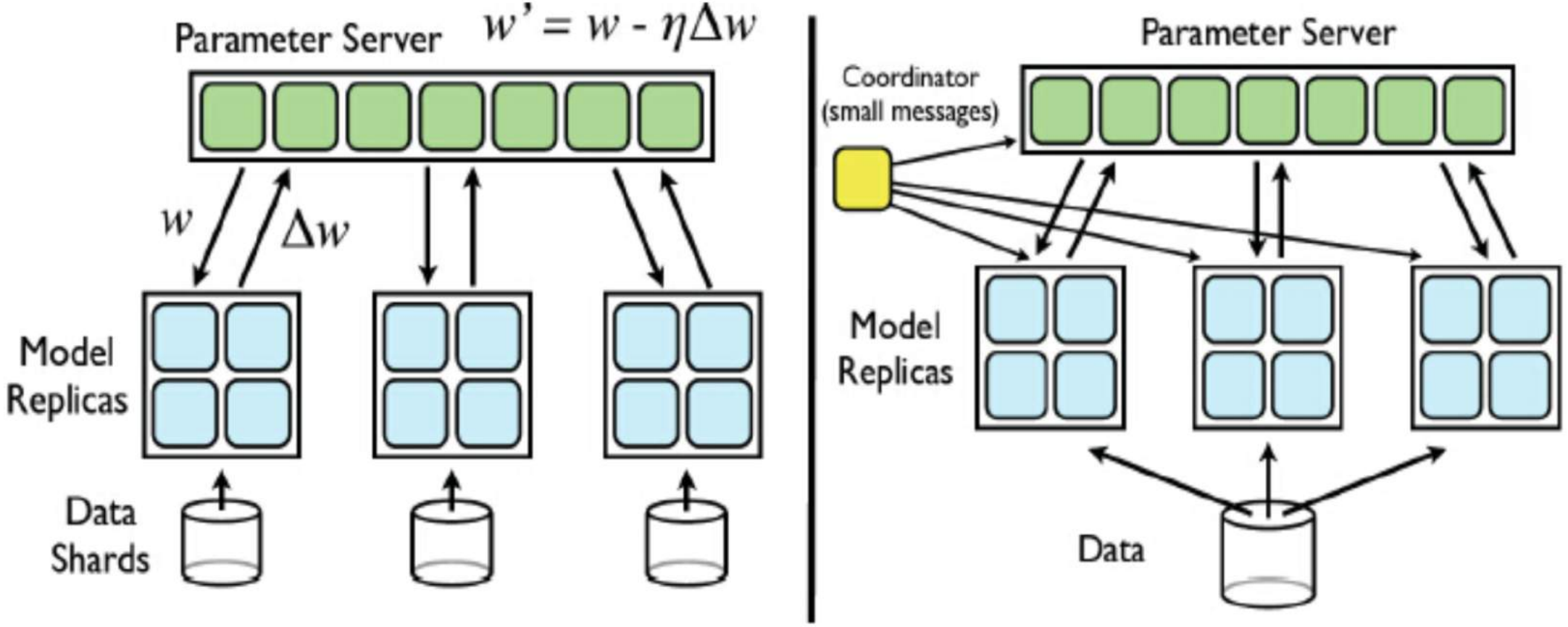
# Efficient Deep Learning

# Model Parallelism




- Deep net is stored and processed on multiple cores (multi-thread) or machines (message passing)
- Performance benefit depends on connectivity structure vs. computational demand

# Data Parallelism



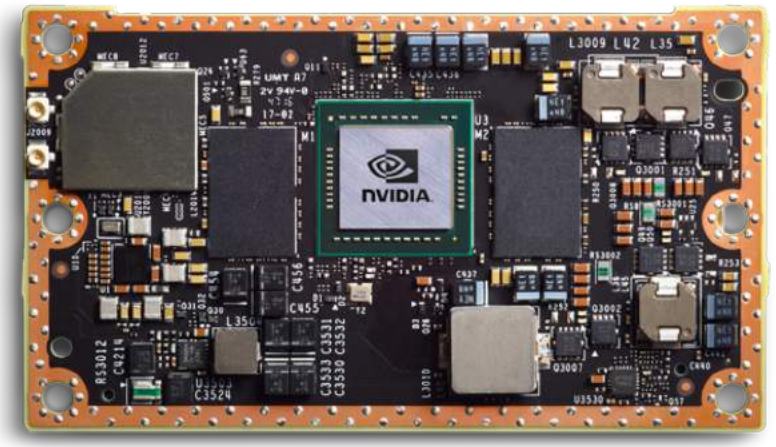

Asynchronous SGD

Distributed L-BFGS

# Efficient Deep Learning on the Edge

- Three top concerns:
  - **Storage and Memory**
  - **Speed or Latency**
  - **Energy Efficiency**


- The three goals all pursue “light weight”
- … but they are often **not aligned***
- … so need to **consider all** in implementation
- … and for both **Inference** and **Training**






* Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks, IEEE ISSCC 2016

# Model Compression

- Training Phase:
  - The easiest way to extract a lot of knowledge from the training data is to learn many different models in parallel.
  - 3B: Big Data, Big Model, Big Ensemble
  - Imagenet:  1.2 million pictures in 1,000 categories.
  - AlexNet: ~ 240Mb, VGG16: ~550Mb
- Testing Phase:
  - Want small and specialist models.
  - Minimize the amount of computation and the memory footprint.
  - Real time prediction
  - Even able to run on mobile devices.

# Two Main Streams

- **“Transfer”:** How to transfer knowledge from big general model (teacher) to small specialist models (student)?
  - Example: “Distilling the Knowledge in a Neural Network”, G. Hinton et. al., 2015

- **“Compress”:** How to reduce the size of the same model, during or after training, without losing much accuracy.
  - Example: “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding”, S. Han et. al., 2016

- **Comparison:** Knowledge Transfer provides a way to train a new small model inheriting from big general models, while Deep Compression Directly does the surgery on big models, using a pipeline: pruning, quantization & Huffman coding.

# Knowledge Transfer/“Distilling”: Main Idea

- Introduce “Soft targets” as a way to transfer the knowledge from big models.
  - Classifiers built from a softmax function have a great deal more information contained in them than just a classifier;
  - The correlations in the softmax outputs are very informative.

- Hard Target: the ground truth label (one-hot vector)
- Soft Target: $q_i = \frac{exp(z_i/T)}{\sum_j exp(z_j/T)}$ T is “temperature”, z is logit

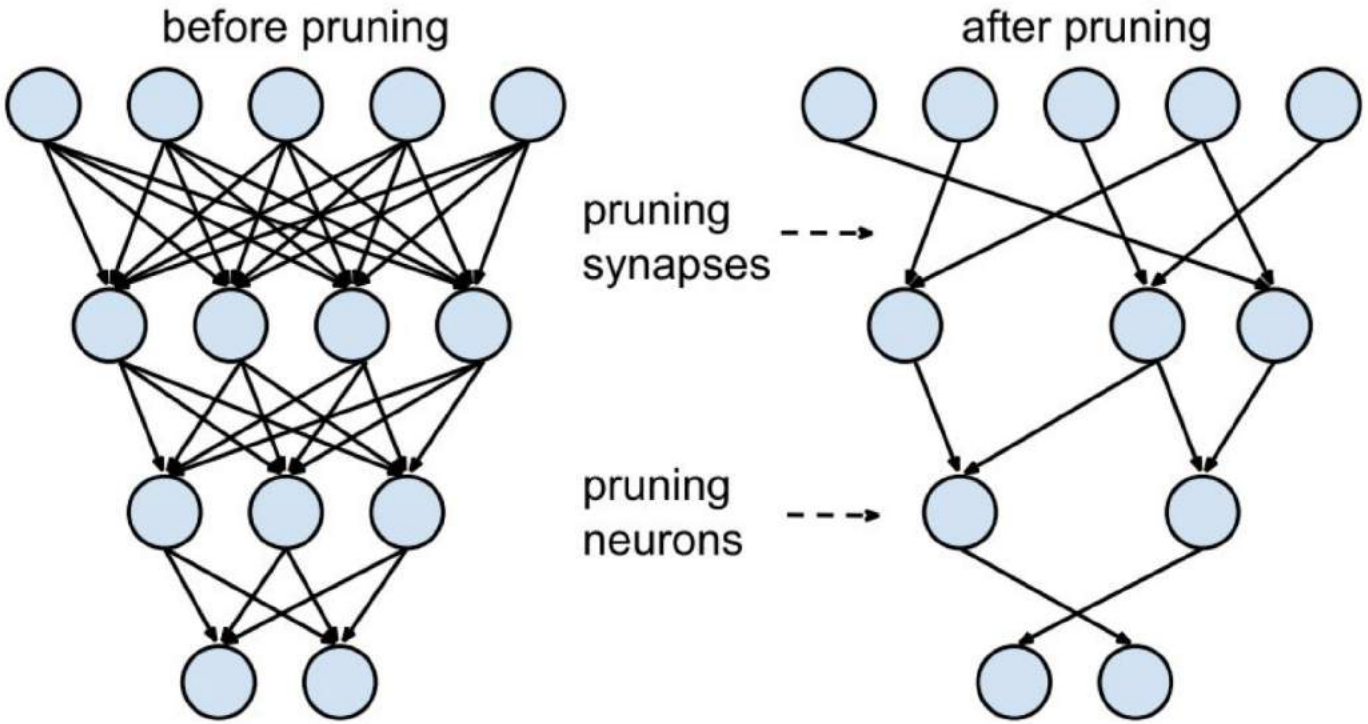
- More information in soft targets

| cow | dog | cat | car | |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | original hard targets |
| .05 | .3 | .2 | .005 | softened output of ensemble |

**Hinton’s Observation:** If we can extract the knowledge from the data using very big models or ensembles of models, it is quite easy to distill most of it into a much smaller model for deployment.

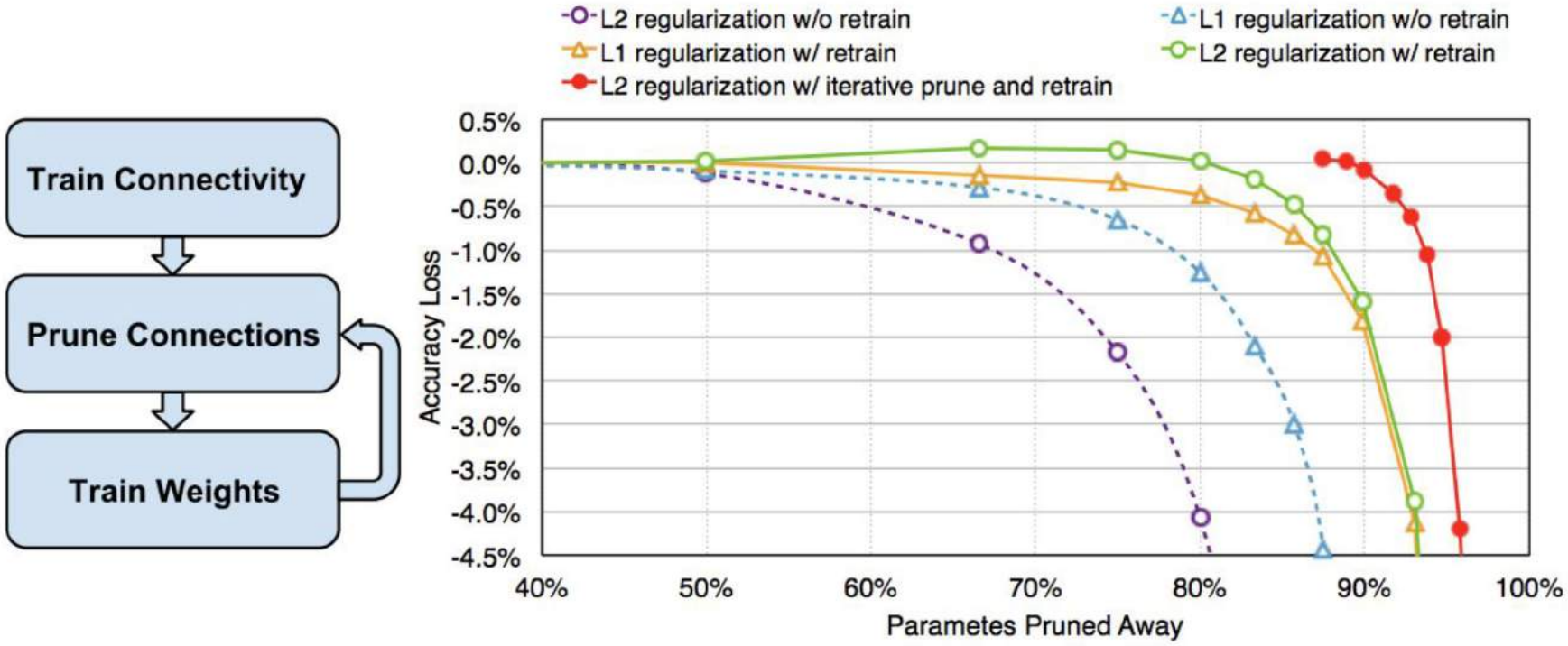
# Deep Compression: Main Idea (i)

## Pruning



# Deep Compression: Main Idea (ii)

## Retrain to Recover Accuracy




Network pruning can save 9x to 13x parameters without drop in accuracy

# Deep Compression: Main Idea (iii)

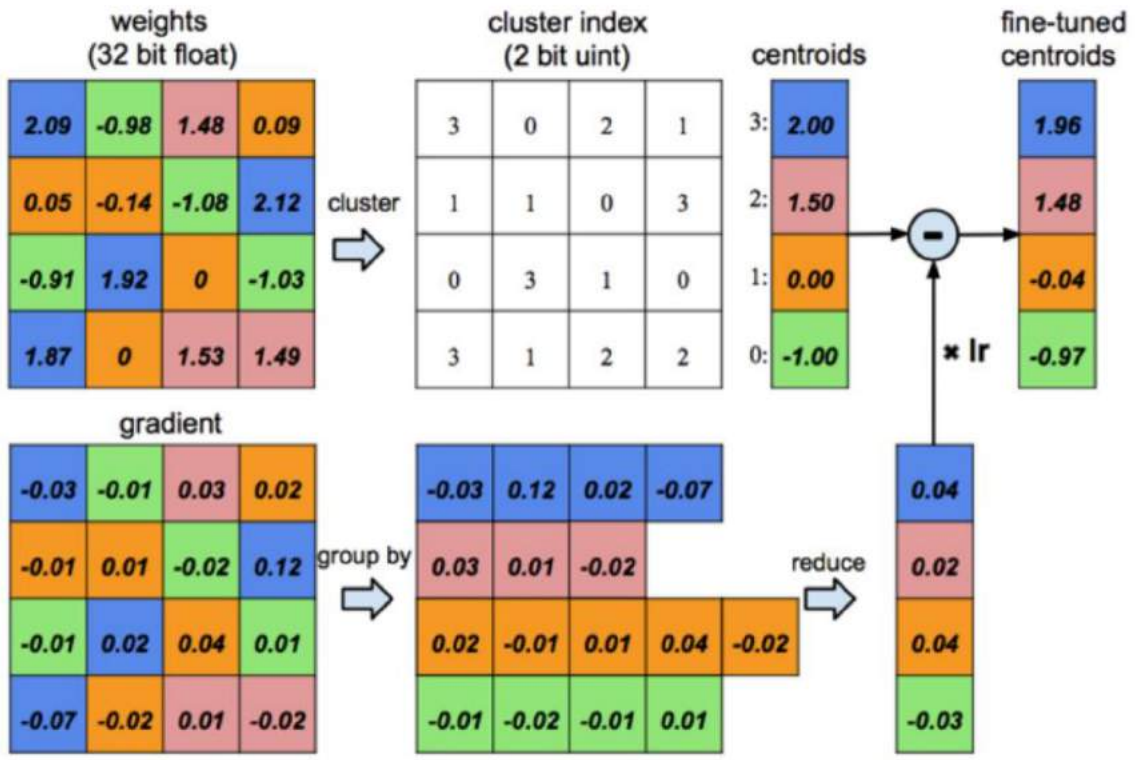
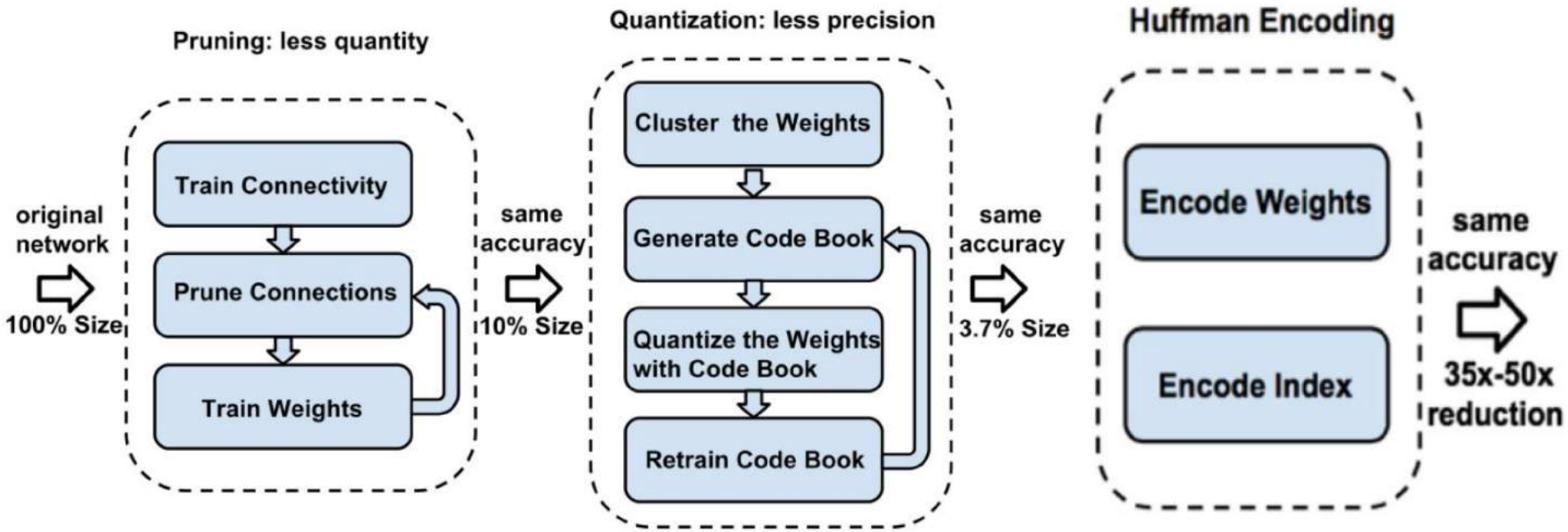



Figure 3: Weight sharing by scalar quantization (top) and centroids fine-tuning (bottom)

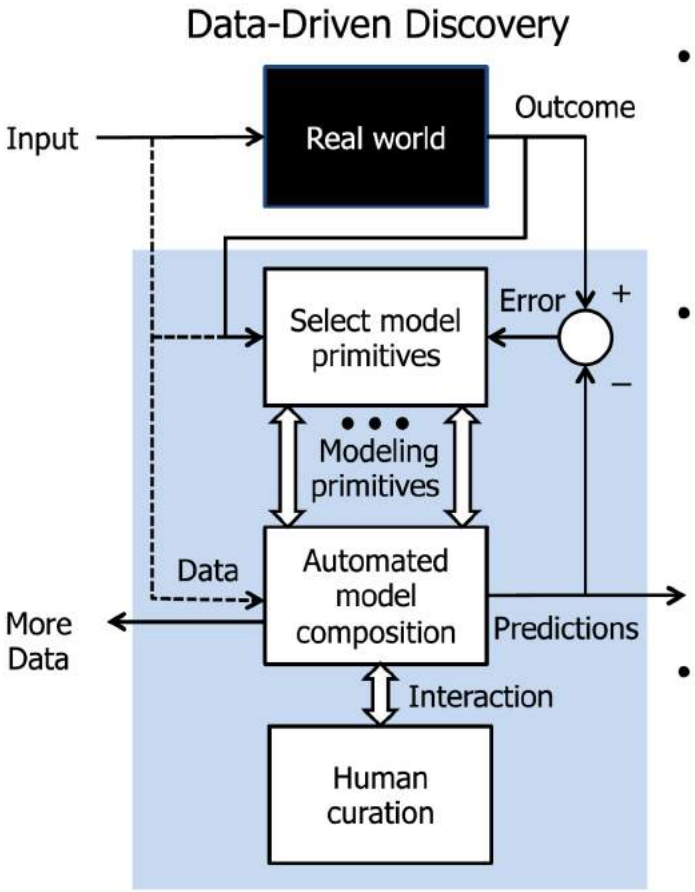
# Deep Compression: Main Idea (iv)

## Huffman Coding



# Meta-Learning

# Automatic Composition of Deep Architectures


- TA1: A library of selectable primitives
  - Create a "vocabulary" of modeling primitives
  - Make primitives automatically selectable
- TA2: Automatically compose complex models
  - Mine corpora of complex models to learn the "syntax" of primitive composition
  - Find optimal compositions
  - Predict additional data requirements
- TA3: Curation of models by non-experts
  - Decompose and formalize questions
  - Explain data and models to enable selection and editing

Today: Hand-crafting, Human engineering, trial and error, trick or magic …

Ideal: automating the architecture design & hyperparameter choice, together with weight learning

# Nerual Architecture Search (NAS) using RL (2016, 2017)

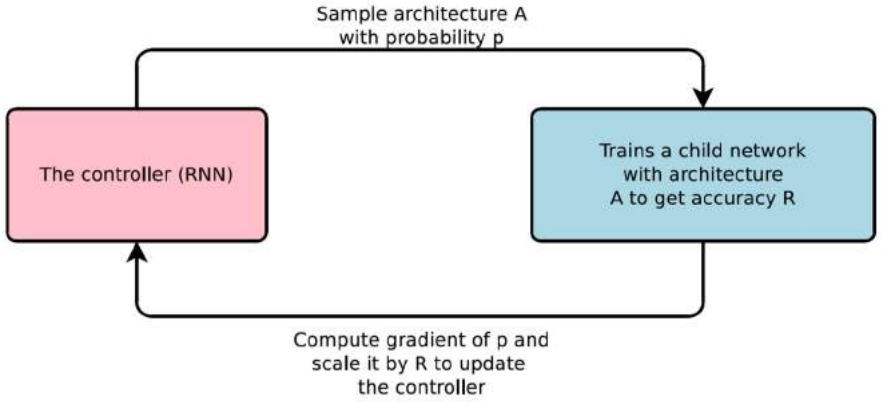



Figure 1: An overview of Neural Architecture Search.

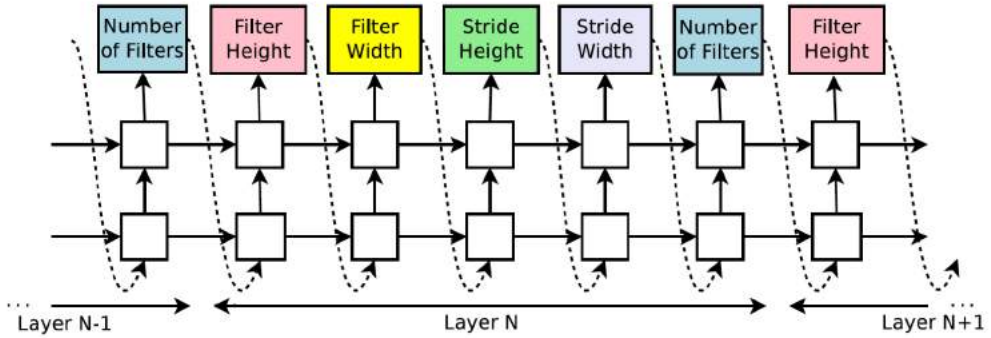
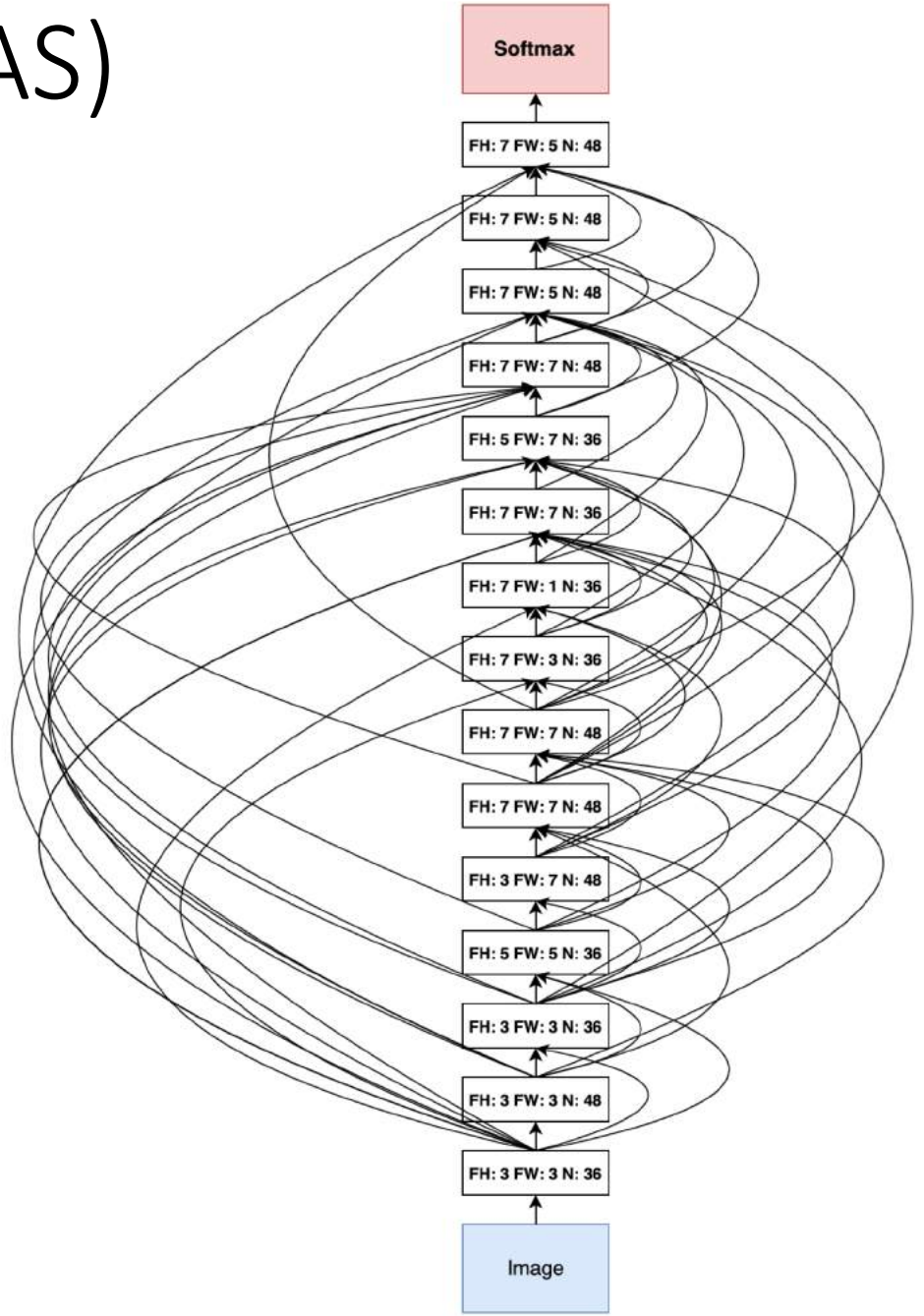



Figure 2: How our controller recurrent neural network samples a simple convolutional netwo predicts filter height, filter width, stride height, stride width, and number of filters for one laye repeats. Every prediction is carried out by a softmax classifier and then fed into the next time as input.



# What it looks like now… (SMASH, ICLR'18)



NAS is a hot new field, indeed adopted by many leading companies, with lots of future potential!

- **Trends:** search better, search faster/cheaper, search for more tasks, search for efficient models …

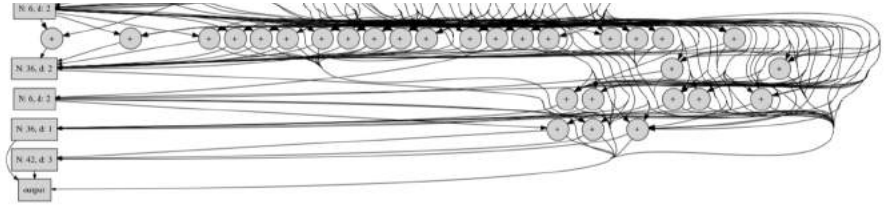


Figure 10: An expanded (though still partially simplified) view of the final block of our best SMASHv2 net. Floating paths are an artifact of the graph generation process, and are actually attached to the nearest rectangular node.

Figure 13: An expanded (though still partially simplified) view of the final block of our best SMASHv1 net. Floating paths are an artifact of the graph generation process, and are actually attached to the nearest rectangular node.