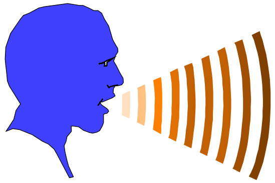


Image and Video Processing

Convolutional Networks for Image Processing (Part I)

Yao Wang
Tandon School of Engineering, New York University

Many contents from Sundeep Rangan:
https://github.com/sdrangan/introml/blob/master/sequence.md

Outline

- Supervised learning: General concepts
- Neural network architecture
  - From single perceptron to multi-layer perceptrons
- Convolutional network architecture
  - Why using convolution and many layers
  - Multichannel convolution
  - Pooling
- Deep networks
- Model training
  - Loss functions
  - Stochastic gradient descent: general concept
  - Data Preprocessing and Regularization
- Training, validation and testing and cross validation
- Demo: training a ConvNet classifier

Supervised Learning

- Given a dataset with many samples
  - Each sample has an input signal x_i (e.g. image) and a ground truth output y_i
- Learning objective
  - Learn a function or model (parameterized by θ) that maps x to y: f(x;θ)=y
  - The function may not be represented by a closed-form representation.
  - Ex: with a neural net, θ includes the weights and biases in all layers
- Formulate as an optimization problem
  - $\theta = \; argmin_\theta \sum_i L(\hat{y}_i, y_i) + \lambda R(\theta)$
    - Loss is the sum of losses for all training samples, all sharing the same parameterθ
    - R(θ): regularization term based on desirable properties of θ
- Generalization ability of a learnt model
  - The model should perform well on testing samples not used for training. Performance is measured on testing samples. More on this later.

Classification vs. Regression

- Classification
  - Each input x (e.g. an image or features of the image) is mapped to a class label $\hat{y}$ (e.g. a person, dog, etc.), and there are only a finite number of classes
  - Predicted output is the probability for each possible class (sum to 1)
  - Typical loss function
    - Binary classification: binary cross entropy
    - Multi-class: cross entropy
- Regression
  - Each input x is mapped to one or multiple continuous values $\hat{y}$
  - Typical loss: MSE

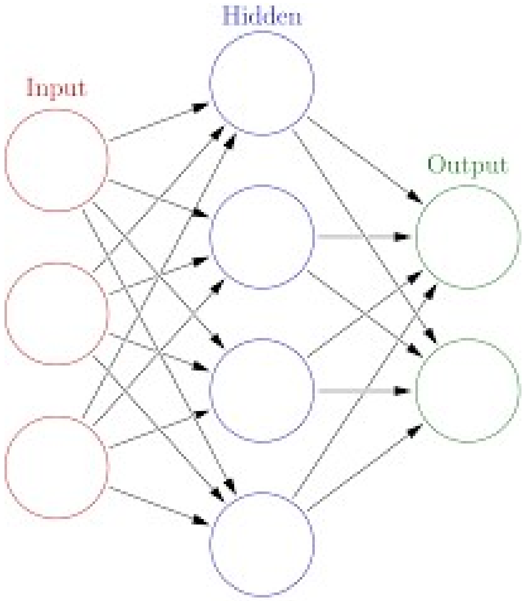
How to Approximate a Function?

- Many possibilities!
  - Lead to different types of models
- Linear regression
- Logistic regression (for classification): linear followed by a sigmoid function to convert to probability
- Support vector machine for classification/regression
- Decision tree for classification/regression
- Neural Networks (multi-layers of logistic regression)
  - A two layer network can approximate any function with sufficient number of hidden nodes
- Convolutional networks
  - Special neural nets that exploit spatial/temporal structure of data such as images and videos
  - Each layer uses multiple convolution filters
  - Needs many layers but each layer with small number of parameters

Outline

- Supervised learning: General concepts
- Neural network architecture
  - From single perceptron to multi-layer perceptrons
- Convolutional network architecture
  - Why using convolution and many layers
  - Multichannel convolution
  - Pooling
- Deep networks
- Model training
  - Loss functions
  - Stochastic gradient descent: general concept
  - Data Preprocessing and Regularization
- Training, validation and testing and cross validation
- Demo: training a ConvNet classifier

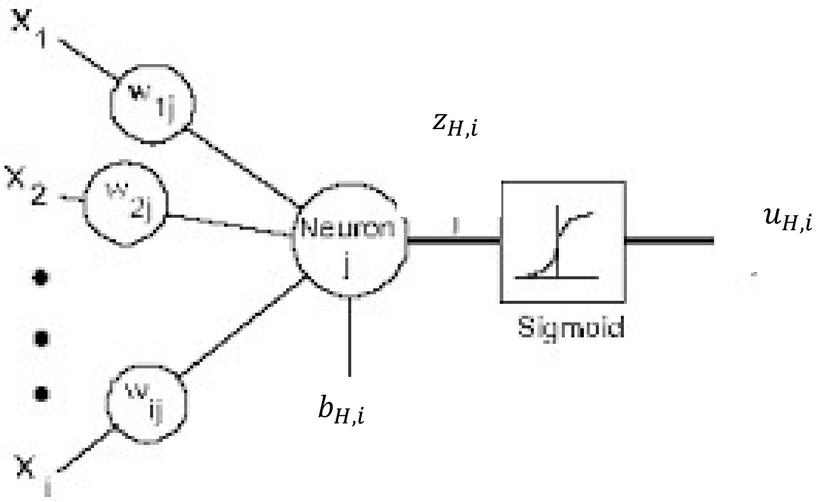
General Structure of Neural Networks

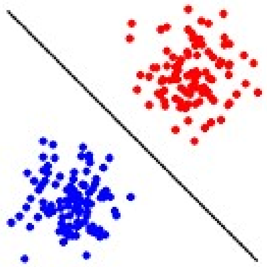


- Input: $\boldsymbol{x} = (x_1, \cdots, x_d)$
  - d = number of features
- Hidden layer:
  - Linear transform: $\boldsymbol{z}_H = \boldsymbol{W}_H \boldsymbol{x} + \boldsymbol{b}_H$
  - Activation function: $\boldsymbol{u}_H = g_{act}(\boldsymbol{z}_H)$
  - Dimension: M hidden units
- Output layer:
  - Linear transform: $\boldsymbol{z}_O = \boldsymbol{W}_O \boldsymbol{u}_H + \boldsymbol{b}_O$
  - Output function: $\boldsymbol{u}_O = g_{out}(\boldsymbol{z}_O)$
  - Dimension: K = number of classes / outputs
- Can be used for classification or regression, with different output functions

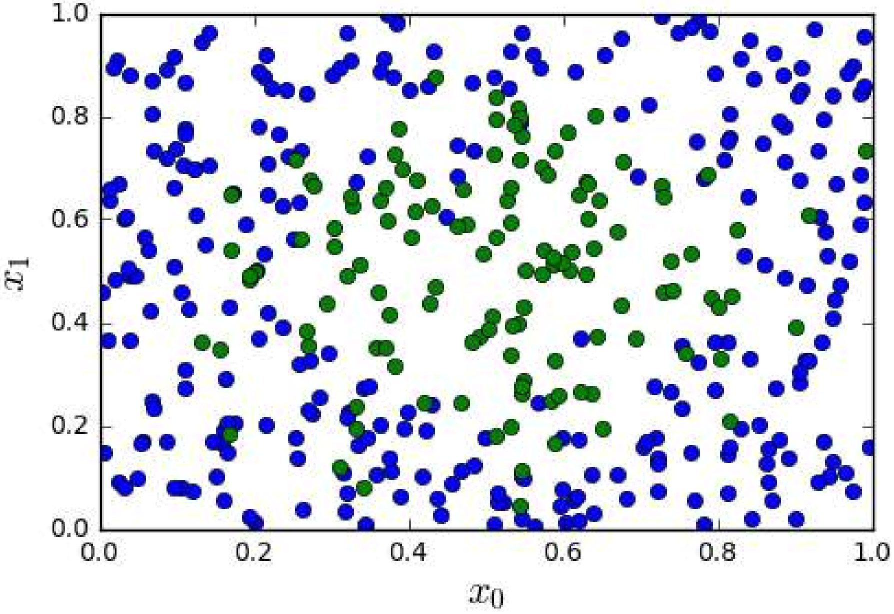
A Single Neuron (Perceptron)

- First combine input variables x_j using an affine transform
  - $z_{H,i} = \sum_j W_{H,ij} x_j + b_{H,i}\,, \quad i = 1,2, \dots ,$
  - $W_{H,ij}$: Weights; $b_{H,i}$: Bias
  - $z_{H,i} = 0$ linearly separates all possible points $\boldsymbol{x}$ by a hyperplane
- Then apply a element-wise nonlinear mapping (activation function $g(z)$)
  - $u_{H,i} = g\left(z_{H,i}\right),\ i = 1,2, \dots ,$
- Equivalent to logistic regression or classifier when the nonlinearity is sigmoidal
  - Works great if the two classes are linearly separable!

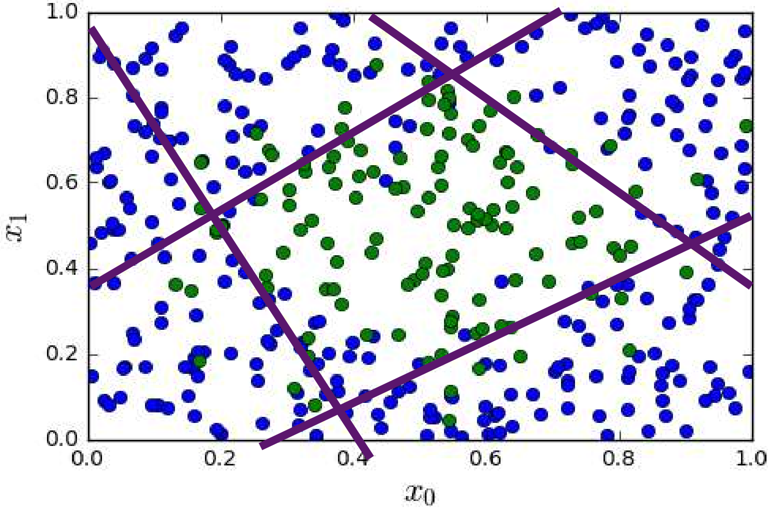


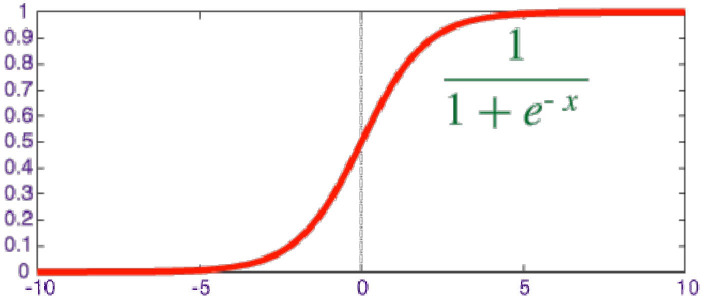


What if not linearly separable?



A Two-Stage Classifier

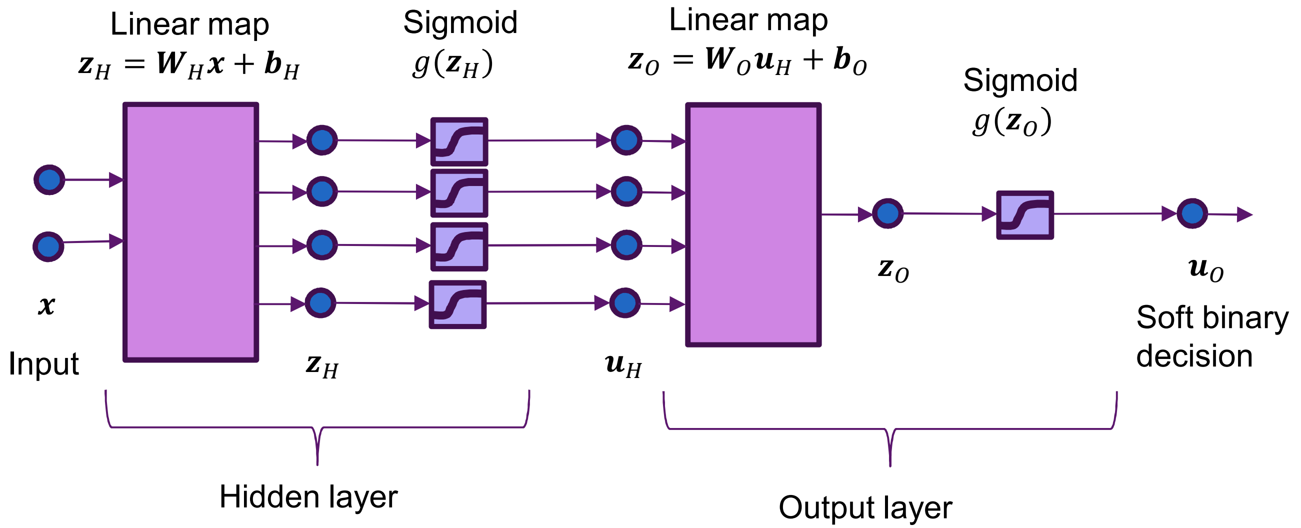




- Input sample: $\boldsymbol{x} = (x_1, x_2)^T$
- First step: Hidden layer
  - Take $N_H = 4$ linear discriminants
    $z_{H,1} = \boldsymbol{w}_{H,1}^T x + b_{H,1}$
    $\vdots$
    $z_{H,N_H} = \boldsymbol{w}_{H,M}^T x + b_{H,M}$
  - Make a soft decision on each linear region
    $u_{H,m} = g(z_{H,m}) = 1/(1 + e^{-z_{H,m}})$
- Second step: Output layer
  - Linear step $z_O = w_O^T u_H + b_O$
  - Soft decision: $u_O = g(z_O)$

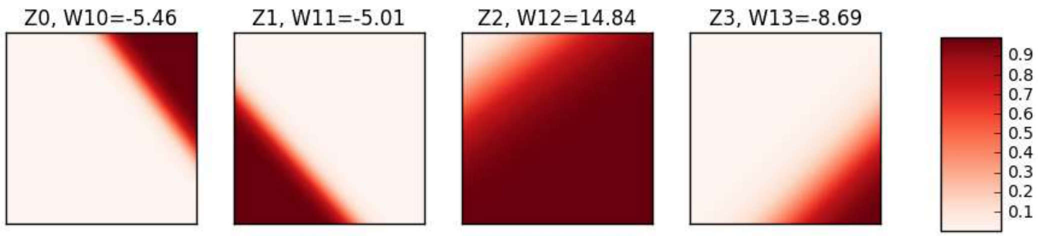
Two-Layer Neural Net for Binary Classification

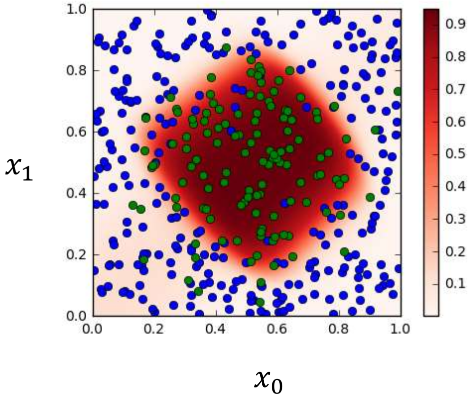
- Hidden layer: $\boldsymbol{z}_H = \boldsymbol{W}_H\boldsymbol{x} + \boldsymbol{b}_H, \quad \boldsymbol{u}_H = g(\boldsymbol{z}_H)$
- Output layer: $\boldsymbol{z}_O = \boldsymbol{W}_O\boldsymbol{u}_H + \boldsymbol{b}_O,\ u_O = g(\boldsymbol{z}_O)$



Hidden layer does not have to use sigmoidal. tanh( ) / ReLU is more often used.
Can have more than one hidden layers.
Also known as a “Multi-Layer Perceptron” (MLP)

Step 1 Outputs and Step 2 Outputs

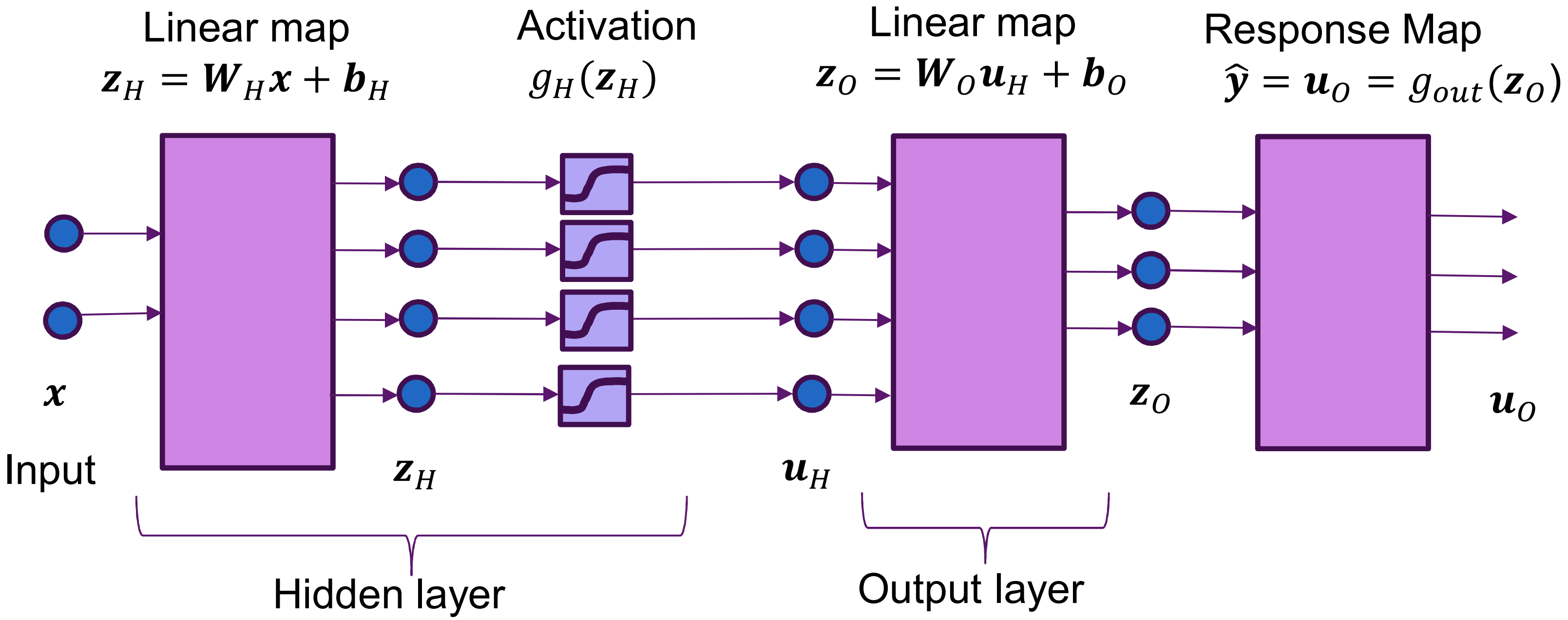




- Each output from step 1 is from a linear classifier with soft decision (Logistic regression)
- Final output is a weighted average of step 1 outputs using the weights indicated on top of the figures

Two-Layer Neural Net for Multiple Outputs

- Hidden layer: $\boldsymbol{z}_H = \boldsymbol{W}_H\boldsymbol{x} + \boldsymbol{b}_H, \quad \boldsymbol{u}_H = g_{act}(\boldsymbol{z}_H)$
- Output layer: $\boldsymbol{z}_O = \boldsymbol{W}_O\boldsymbol{u}_H + \boldsymbol{b}_O$
- Response map: $\hat{y} = u_O = g_{out}(\boldsymbol{z}_O)$

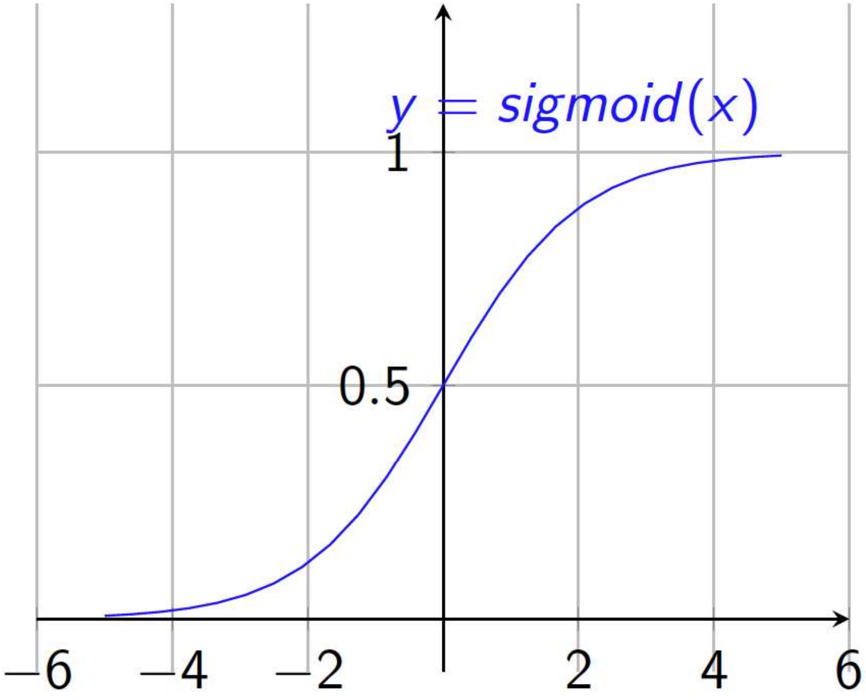

Linear map
$\boldsymbol{z}_H = \boldsymbol{W}_H\boldsymbol{x} + \boldsymbol{b}_H$
Activation
$g_H(\boldsymbol{z}_H)$
Linear map
$\boldsymbol{z}_O = \boldsymbol{W}_O\boldsymbol{u}_H + \boldsymbol{b}_O$
Response Map
$\hat{\boldsymbol{y}} = \boldsymbol{u}_O = g_{out}(\boldsymbol{z}_O)$
$\boldsymbol{x}$
Input
$\boldsymbol{z}_H$
$\boldsymbol{u}_H$
$\boldsymbol{z}_O$
$\boldsymbol{u}_O$
Hidden layer
Output layer

Response Map or Output Activation

- Last layer depends on type of response
- Binary classification: $y = \pm 1$
  - z_O is a scalar
  - Hard decision: $\hat{y} = \text{sign}(z_O)$
  - Soft decision: $\hat{y} = P(y = 1|x) = 1/(1 + e^{-z_O})$ (probability of class 1)
- Multi-class classification: $y = 1, \dots, K$
  - Ground truth label $\boldsymbol{y}$ is *K*-dimension (One-Hot Encoding)
  - $\boldsymbol{z}_O = \left[z_{O,1}, \cdots, z_{O,K}\right]^T$ is a vector
  - $u_{O,k} = P(y = k|x)$ (probability of class k)
  - Hard decision: $u_{O,k} = 1 \; if \; k = \arg\max_{l} z_{O,l} \,;\; u_{O,k} = 0, \text{otherwise}$
  - Soft decision: $u_{O,k} = S_k(\boldsymbol{z}_O) = \frac{e^{z_{O,k}}}{\sum_l e^{z_{O,l}}}$ (softmax)
- Regression: $\boldsymbol{y} \in R^d$
  - $\hat{\boldsymbol{y}} = \boldsymbol{z}_O$ (linear output layer)

Non-linearities: Sigmoid

- $\sigma(z) = \frac{1}{1+e^{-z}}$
- Interpretation as firing rate of neuron
- Bounded between [0,1]
- Saturation for large +ve,-ve inputs
- Gradients go to zero
- Outputs centered at 0.5 (poor conditioning)
- Not used in practice

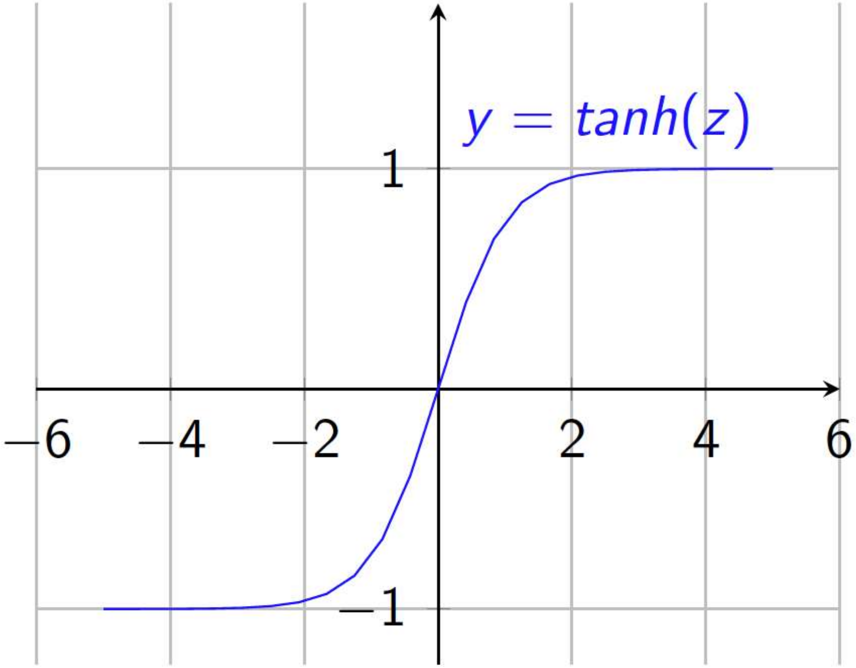


From Fergus: https://cs.nyu.edu/~fergus/teaching/vision/2_neural_nets.pdf

Sigmoid nonlinearity converts z to a probability of being one class, and is used for binary classification. Not used in intermediate layers.

Non-linearities: Tanh

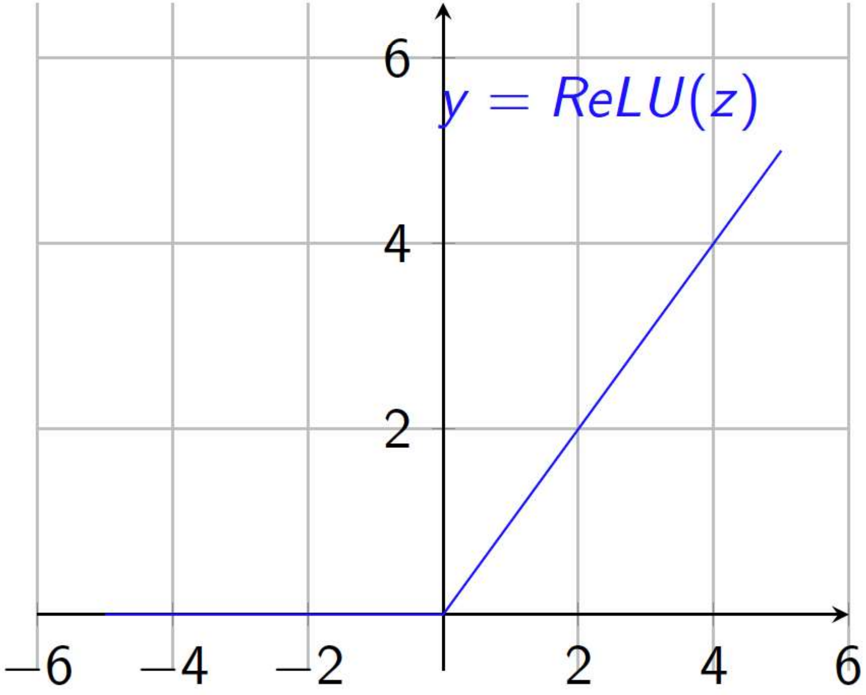
- $\sigma(z) = \tanh(z)$
- Bounded in [+1,-1] range
- Saturation for large +ve, -ve inputs
- Outputs centered at zero
- Preferable to sigmoid



From Fergus: https://cs.nyu.edu/~fergus/teaching/vision/2_neural_nets.pdf

Non-linearities: Rectified Linear (ReLU)

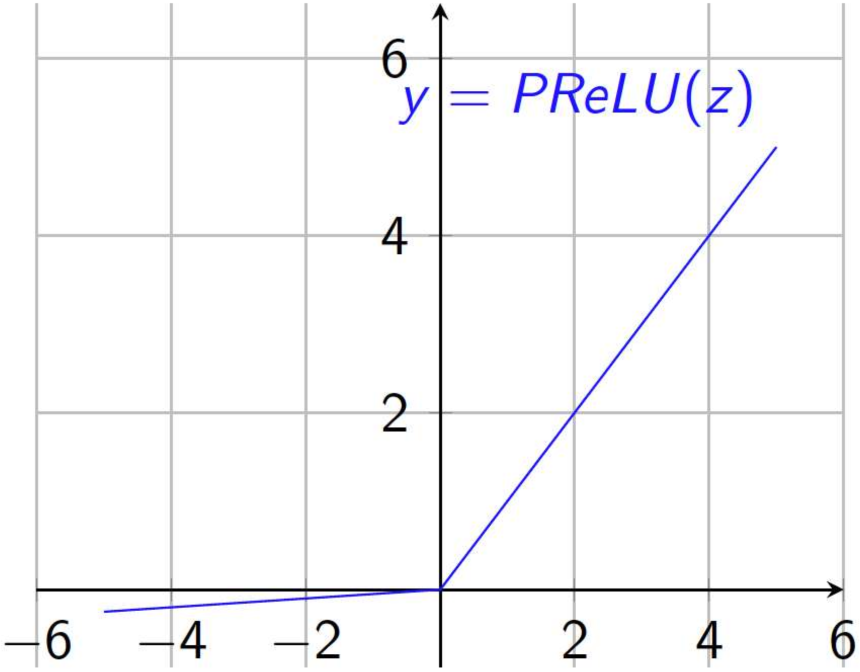
- $\sigma(z) = max(z, 0)$
- Unbounded output (on positive side)
- Efficient to implement: $\frac{d\sigma(z)}{dz} = \{0, 1\}$.
- Also seems to help convergence (see 6x speedup vs tanh in Krizhevsky et al.)
- Drawback: if strongly in negative region, unit is dead forever (no gradient).
- Default choice: widely used in current models.



From Fergus: https://cs.nyu.edu/~fergus/teaching/vision/2_neural_nets.pdf

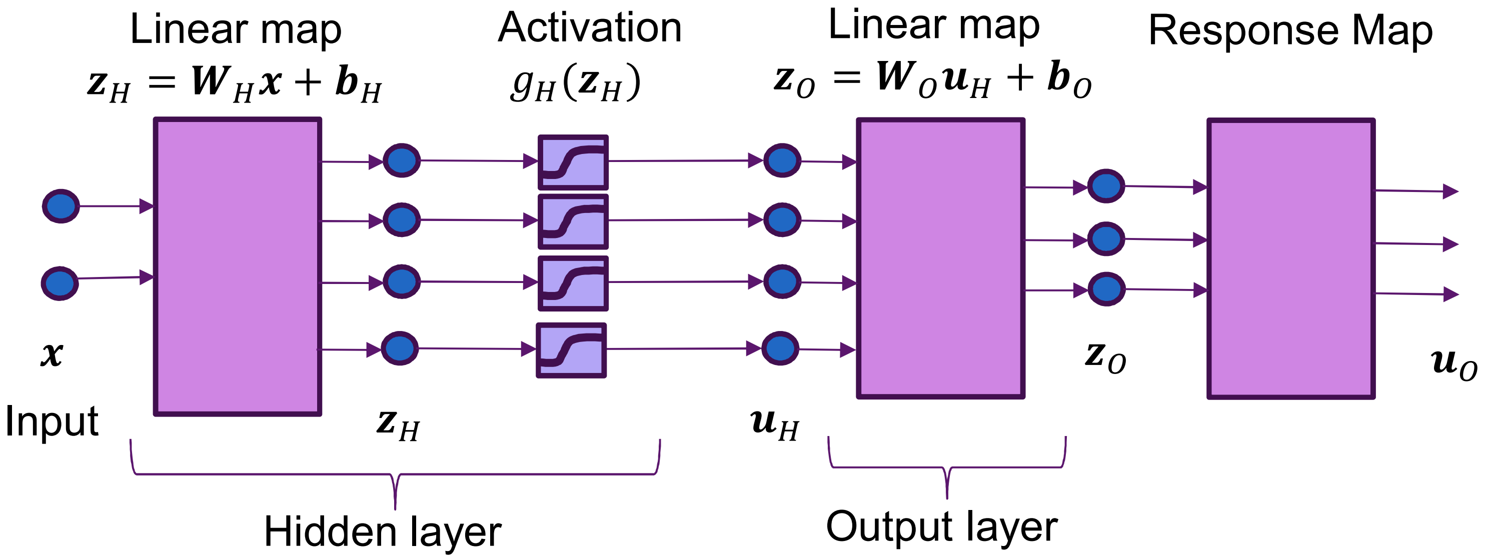
Non-linearities: Leaky RELU

- Leaky Rectified Linear $\sigma(z) = 1[z > 0]max(0, \overset{z}{x}) + 1[z < 0]max(0, \alpha z)$
- where α is small, e.g. 0.02
- Also known as probabilistic ReLU (PReLU)
- Has non-zero gradients everywhere (unlike ReLU)
- α can also be learned (see Kaiming He et al. 2015).



From Fergus: https://cs.nyu.edu/~fergus/teaching/vision/2_neural_nets.pdf

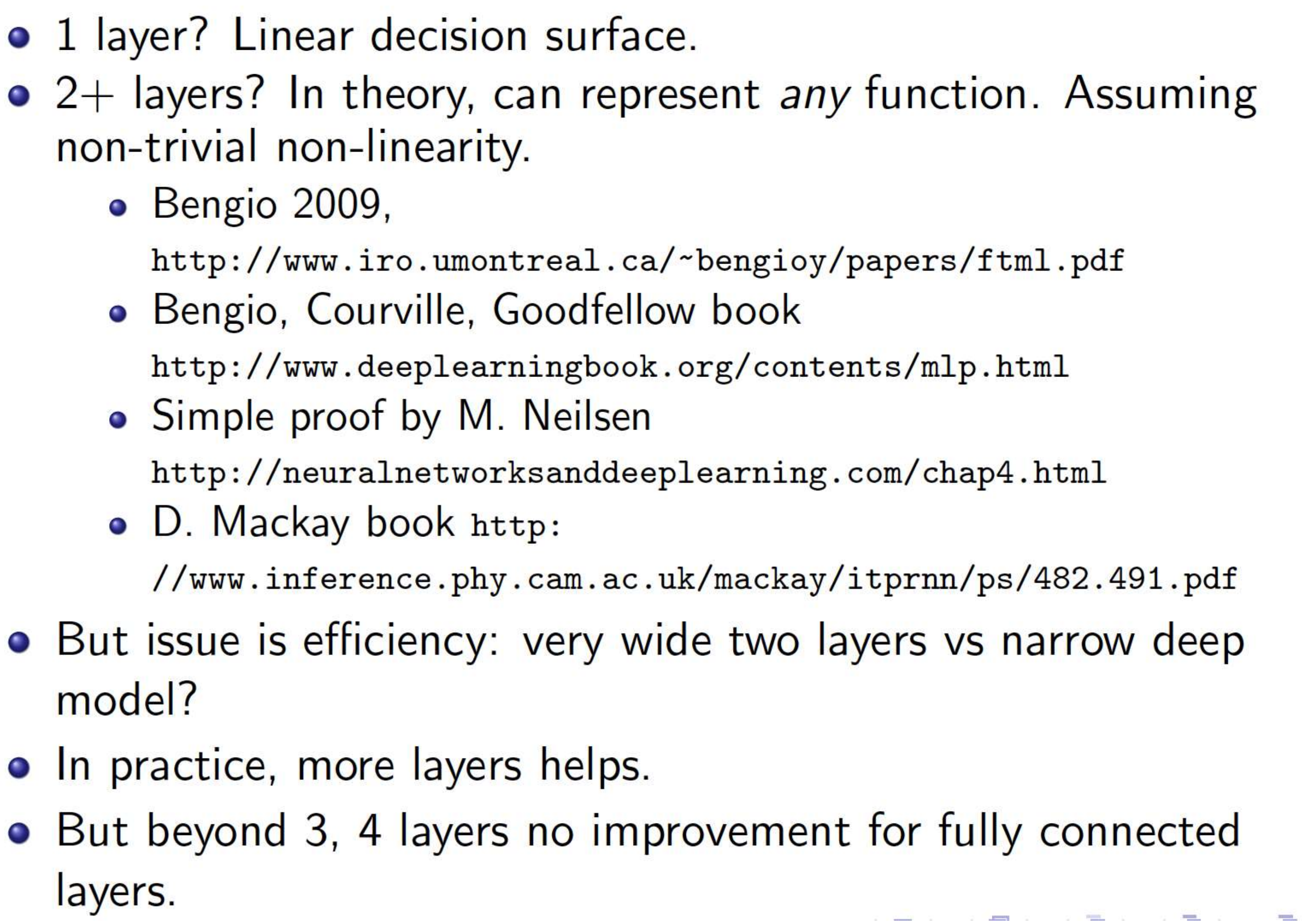
Number of Parameters of a Two Layer Network



Layer	Parameter	Symbol	Number parameters
Hidden layer	Bias	b_H	N_H
	Weights	W_H	$N_H d$
Output layer	Bias	b_O	K
	Weights	W_O	$K N_H$
Total			$N_H(d+1) + K(N_H + 1)$

- d = input dimension, N_H= number of hidden units, K=output dimension
- N_H is a free parameter. Should be chosen properly.

Representation Power: what function can an MLP represent?



- 1 layer? Linear decision surface.
- 2+ layers? In theory, can represent *any* function. Assuming non-trivial non-linearity.
  - Bengio 2009, `http://www.iro.umontreal.ca/~bengioy/papers/ftml.pdf`
  - Bengio, Courville, Goodfellow book `http://www.deeplearningbook.org/contents/mlp.html`
  - Simple proof by M. Neilsen `http://neuralnetworksanddeeplearning.com/chap4.html`
  - D. Mackay book `http://www.inference.phy.cam.ac.uk/mackay/itprnn/ps/482.491.pdf`
- But issue is efficiency: very wide two layers vs narrow deep model?
- In practice, more layers helps.
- But beyond 3, 4 layers no improvement for fully connected layers.

From Fergus: https://cs.nyu.edu/~fergus/teaching/vision/2_neural_nets.pdf

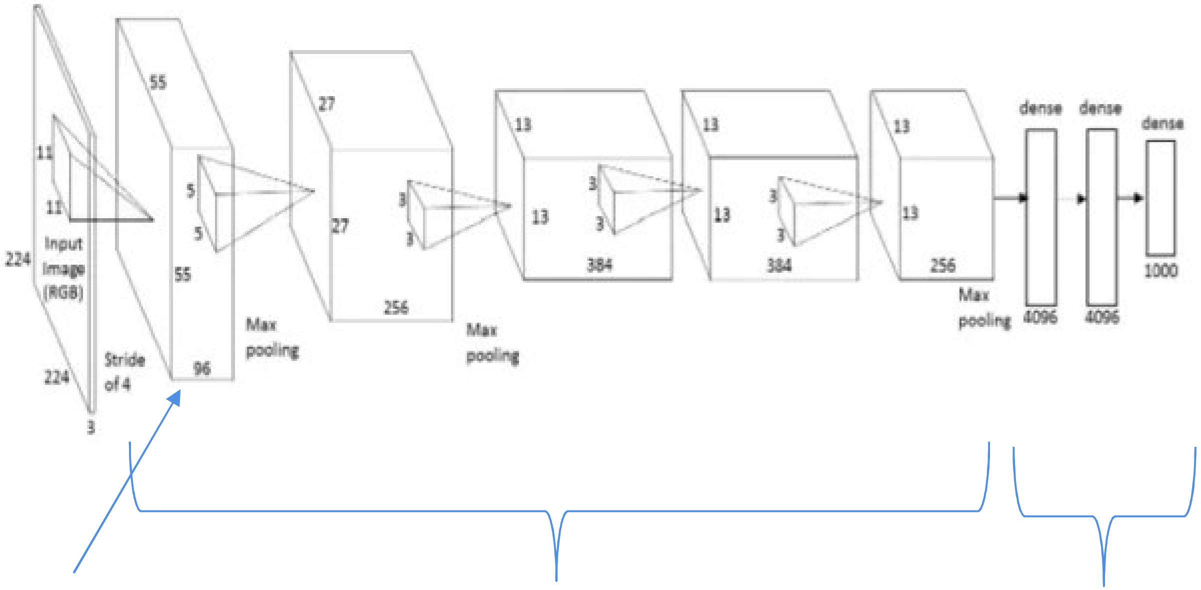
Outline

- Supervised learning: General concepts
- Neural network architecture
  - From single perceptron to multi-layer perceptrons
- Convolutional network architecture
  - Why using convolution and many layers
  - Multichannel convolution
  - Pooling
- Deep networks
- Model training
  - Loss functions
  - Stochastic gradient descent: general concept
  - Data Preprocessing and Regularization
- Training, validation and testing and cross validation
- Demo: training a ConvNet classifier

Convolutional Network

- MLP uses fully-connected layers:
  - In each layer, each output is a weighted sum of all the inputs followed by a non-linearity
  - If the input is an image, each output of the first layer will depends on all the pixels
  - In image processing, we benefit from local operations (convolution), to detect local patterns (motivated by visual system computation)
- Convolutional network uses convolutional layers
  - Each layer produces multiple output feature maps, each obtained by convolving each input feature map and sum all convolved feature maps (multi-channel convolution)
  - Each layer is specified by the filter corresponding to each output map. Multiple filters are used to produce multiple maps
  - Motivated by visual system processing using local computations
  - Significantly smaller number of parameters for the same number of output at each layer

Example network



- Alex Net
- Each convolutional layer has:
  - 2D convolution
  - Activation (eg. ReLU)
  - Pooling or sub-sampling

96 feature maps of size 55x55 each

Convolutional layers
For feature extraction

2D convolution with Activation and pooling / sub-sampling

Fully connected layers
For Classification task

Matrix multiplication & activation

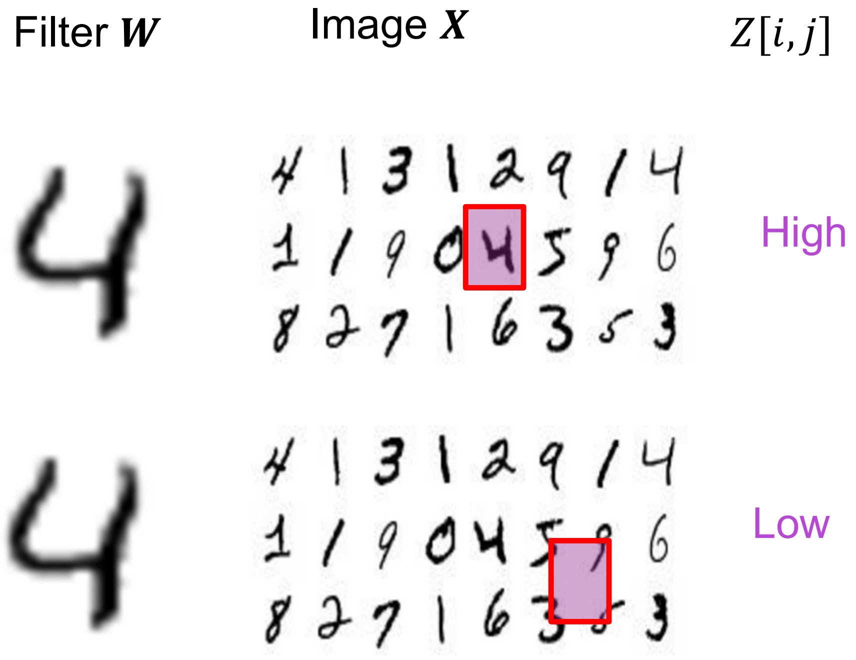
Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems*. 2012.

What does convolution do?

- Convolution: Find local feature by sliding a filter (convolution w/o reversal)
- Large image: $X \ \ N_1 \times N_2$ (e.g. 512 x 512)
- Small filter: $W \ \ K_1 \times K_2$ (e.g. 8 x 8)
- At each pixel (i, j) compute:

$$Z[i,j] = \sum_{k_1=0}^{K_1-1} \sum_{k_2=0}^{K_2-1} W[k_1,k_2] X[i+k_1, j+k_2]$$

  - Correlation of W with image box starting at (i, j)
  - $Z[i, j]$ is large if feature is present around (i, j)



Filter $\boldsymbol{W}$

Image $\boldsymbol{X}$

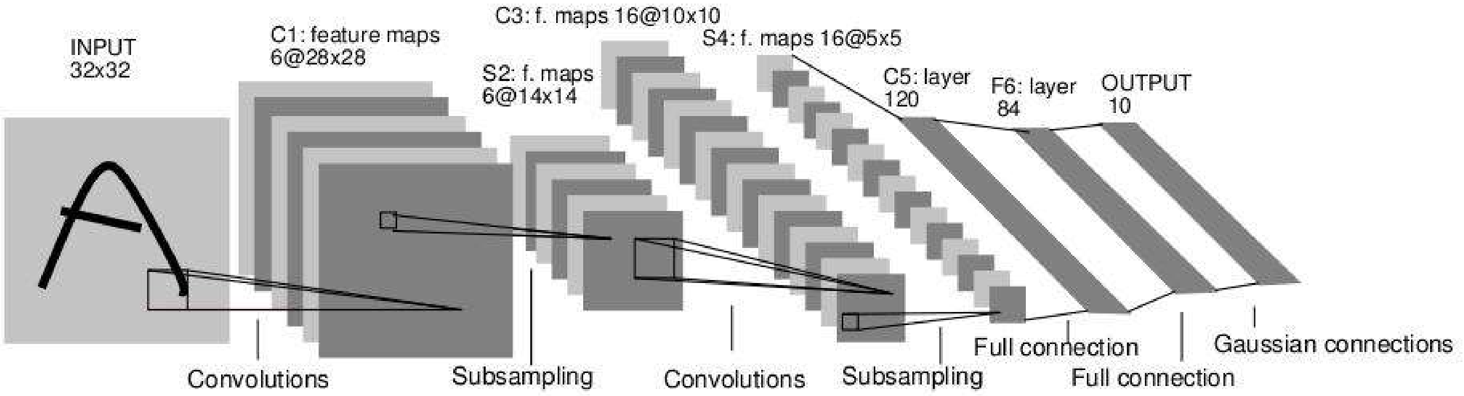
$Z[i,j]$

High

Low

Why Convolution Layers?

- Exploit two properties of images
  - Dependencies are local
    - No need to have each output unit connect to all pixles
  - Spatially stationary statistics
    - Translation invariant dependencies
    - Slide the same filter over all input pixels
    - Only approximately true
- LeCun et al. 1989 (LeNet)



From Fergus: https://cs.nyu.edu/~fergus/teaching/vision/3_convnets.pdf

Convolution with/without reversal

- In signal processing and math, convolution includes flipping:

$$z[n_1, n_2] = \sum_{k_2=0}^{K_2-1} \sum_{k_1=0}^{K_2-1} w[k_1, k_2] x[n_1 - k_1, n_2 - k_2]$$

  - For this class, we will call this convolution with reversal

- But, in many neural network packages, convolution does not include flipping:

$$z[n_1, n_2] = \sum_{k_2=0}^{K_2-1} \sum_{k_1=0}^{K_2-1} w[k_1, k_2] x[n_1 + k_1, n_2 + k_2]$$

  - Will call this convolution without reversal (= correlation)

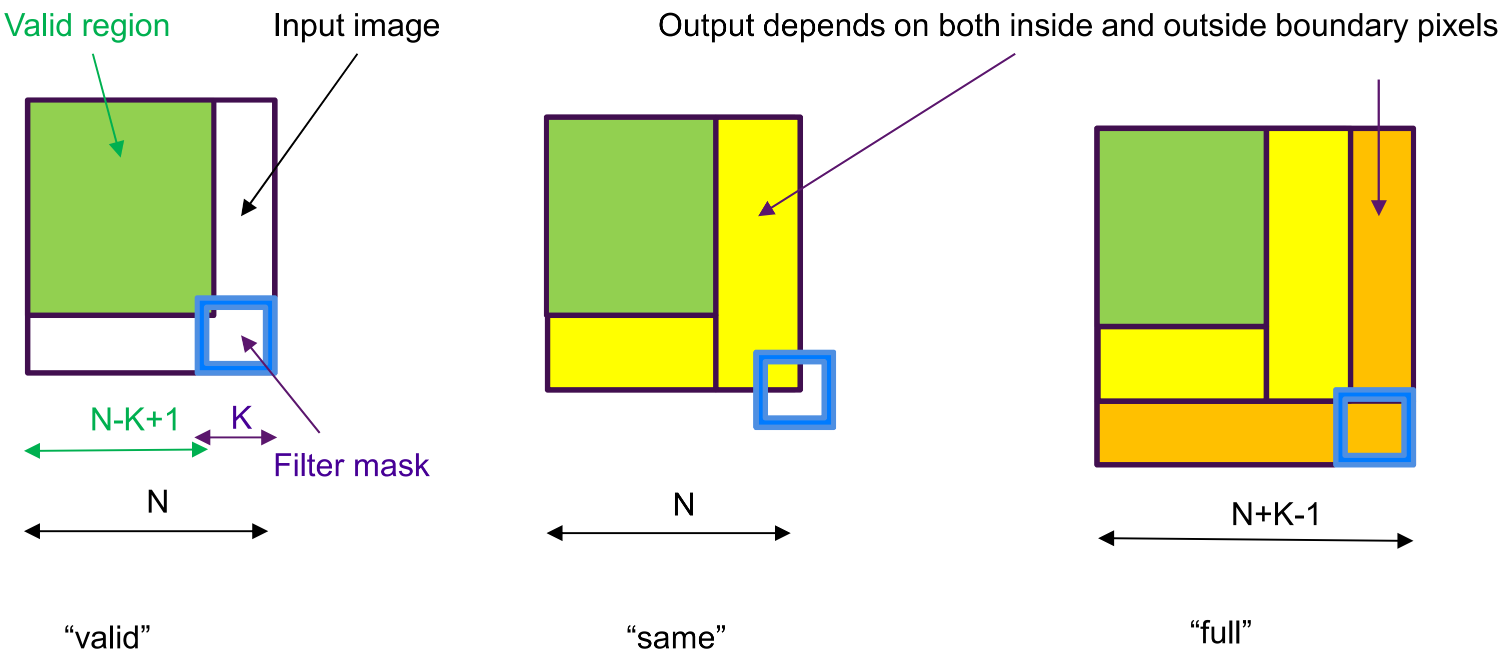
Boundary Conditions

- Suppose inputs are
  - $x,$ size $N_1 \times N_2,$ w: size $K_1 \times K_2,$ $K_1 \leq N_1, K_2 \leq N_2$
  - $z = x * w$ (without reversal)

$$z[n_1, n_2] = \sum_{k_2=0}^{K_2-1} \sum_{k_1=0}^{K_2-1} w[k_1, k_2] x[n_1 + k_1, n_2 + k_2]$$

- Different ways to define outputs
- Valid mode: $0 \leq n_1 < N_1 - K_1 + 1,\ 0 \leq n_2 < N_2 - K_2 + 1$
  - Requires no zero padding
- Same mode: Output size $N_1 \times N_2$
  - Usually use zero padding for neural networks
- Full mode: Output size $(N_1 + K_1 - 1) \times (N_2 + K_2 - 1)$
  - Not used often in neural networks

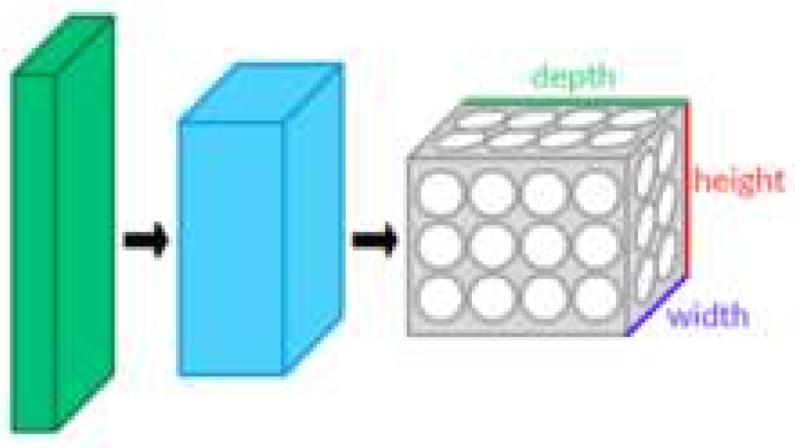
Boundary Effect



Note that with convolution with reversal, the boundary effect will be observed at the top and left sides.

Convolutional Inputs & Outputs

- Inputs and outputs are images with multiple channels
  - Number of channels also called the depth
- Can be described as tensors
- Input tensor, X shape (N_1, N_2, N_{in})
  - $N_1, N_2 = \text{input image size}$
  - N_{in} = number of input channels
- Output tensor, Z shape (M_1, M_2, N_{out})
  - $M_1, M_2 = \text{output image size}$
  - N_{out} = number of output channels



Multi-Channel Convolution

- Weight and bias:
  - W: Weight tensor, size $(K_1, K_2, N_{in}, N_{out})$
  - b: Bias vector, size N_{out}
- Convolutions performed over space and added over channels

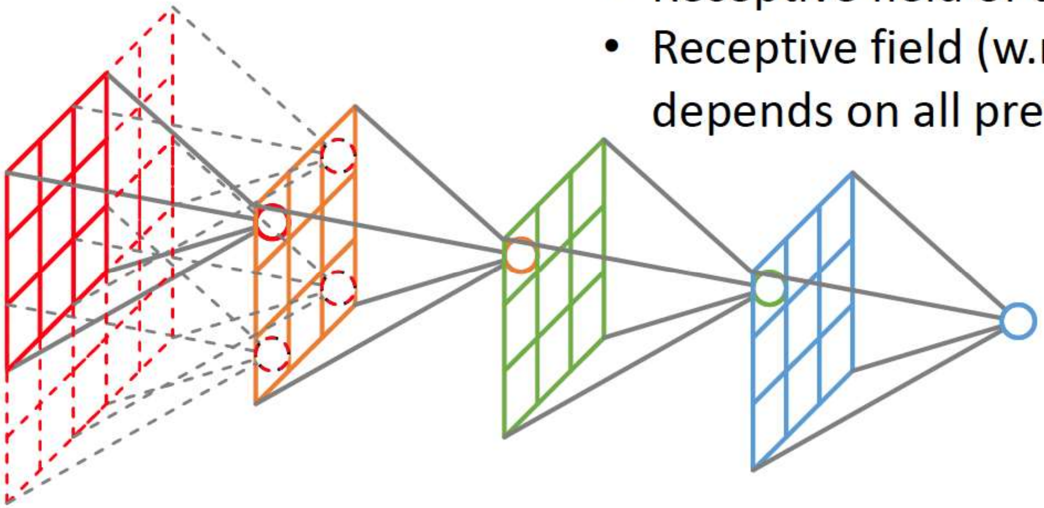
$$Z[i_1, i_2, m] = \sum_{k_1=0}^{K_1-1} \sum_{k_2=0}^{K_2-1} \sum_{n=0}^{N_{in}-1} W[k_1, k_2, n, m] X[i_1 + k_1, i_2 + k_2, n] + b[m]$$

- For each output channel m, input channel n
  - Computes 2D convolution with $W[:, :, n, m]$ (2D filters of size $K_1 \times K_2$)
  - Sums results over n
  - Different 2D filter for each input channel and output channel pair

Activation and Pooling

- Convolution typically followed by activation and pooling
- Activation, typically ReLU or PReLu
  - Zeros out negative values
- Pooling
  - Downsample output after activation
  - Different methods (max, sum, sub-sampling)
  - Output combines local features from adjacent regions
  - Creates more complex features over wider areas

Receptive Field



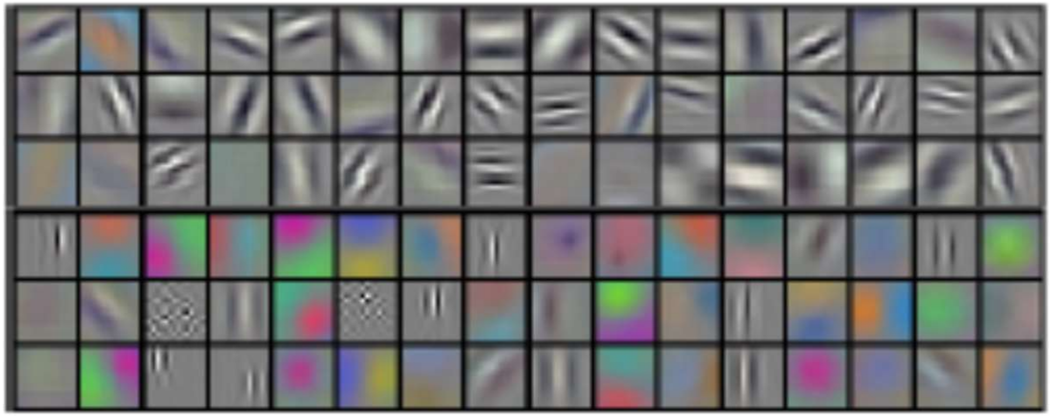
- Receptive field of the first layer is the filter size
- Receptive field (w.r.t. input image) of a deeper layer depends on all previous layers' filter size and strides

- Correspondence between a feature map pixel and an image pixel is not unique
- Map a feature map pixel to the center of the receptive field on the image in the SPP-net paper

Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition". ECCV 2014.

From Fergus: https://cs.nyu.edu/~fergus/teaching/vision/3_convnets.pdf

What do convnet learn?



- AlexNet first layer
  - 96 filters
  - Size 11 x 11 x 3
  - Applied to image of 224 x 224 x 3
- What do these learned features look like?
- Selective to basic low-level features
  - Curves, edges, color transitions, …

Convolution vs Fully Connected

- Convolution exploits translational invariance
  - Same features is scanned over whole image
- Greatly reduces number of parameters
  - Nin input channels of size M1xN1, Nout output channels with size M2xN2
  - Fully connected network: Nin*Nout*M1*N1*M2*N2+Nout*M2*N2
  - Convolutional network with K1xK2 filter: Nin*Nout*K1*K2+Nout
- Example: Consider first layer in LeNet
  - 32 x 32 image (1 channel) to 6 channels using 5 x 5 filters
  - Creates 6 x 28 x 28 outputs (keeping only the valid region)
  - Fully connected would require 32 x 32 x 6 x 28 x 28 + 6 x 28 x 28 = 4.9 million parameters!
  - Convolutional layer requires only 6 x 5 x 5 + 6 = 156 parameters
  - Reserve fully connected layers for last few layers (for non-image output such as classification).

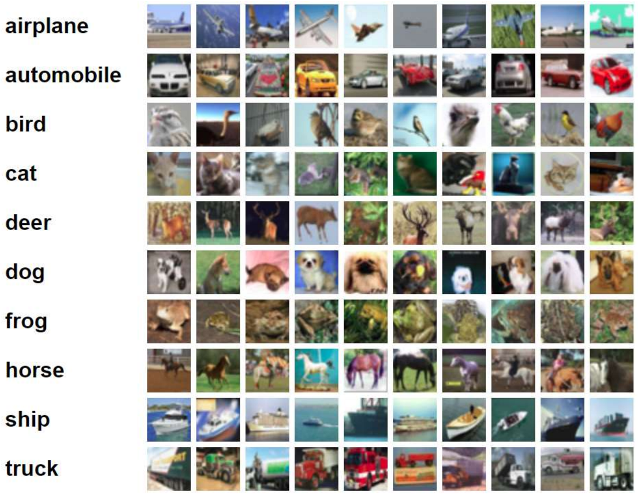
Outline

- Supervised learning: General concepts
- Neural network architecture
  - From single perceptron to multi-layer perceptrons
- Convolutional network architecture
  - Why using convolution and many layers
  - Multichannel convolution
  - Pooling
- Deep networks
- Model training
  - Loss functions
  - Stochastic gradient descent: general concept
  - Data Preprocessing and Regularization
- Training, validation and testing and cross validation
- Demo: training a ConvNet classifier

Large-Scale Image Classification

- Pre-2009, many image recognition systems worked on relatively small datasets
  - MNIST: 10 digits
  - CIFAR 10 (right)
  - CIFAR 100
  - …
- Small number of classes (10-100)
- Low resolution (eg. 32 x 32 x 3)

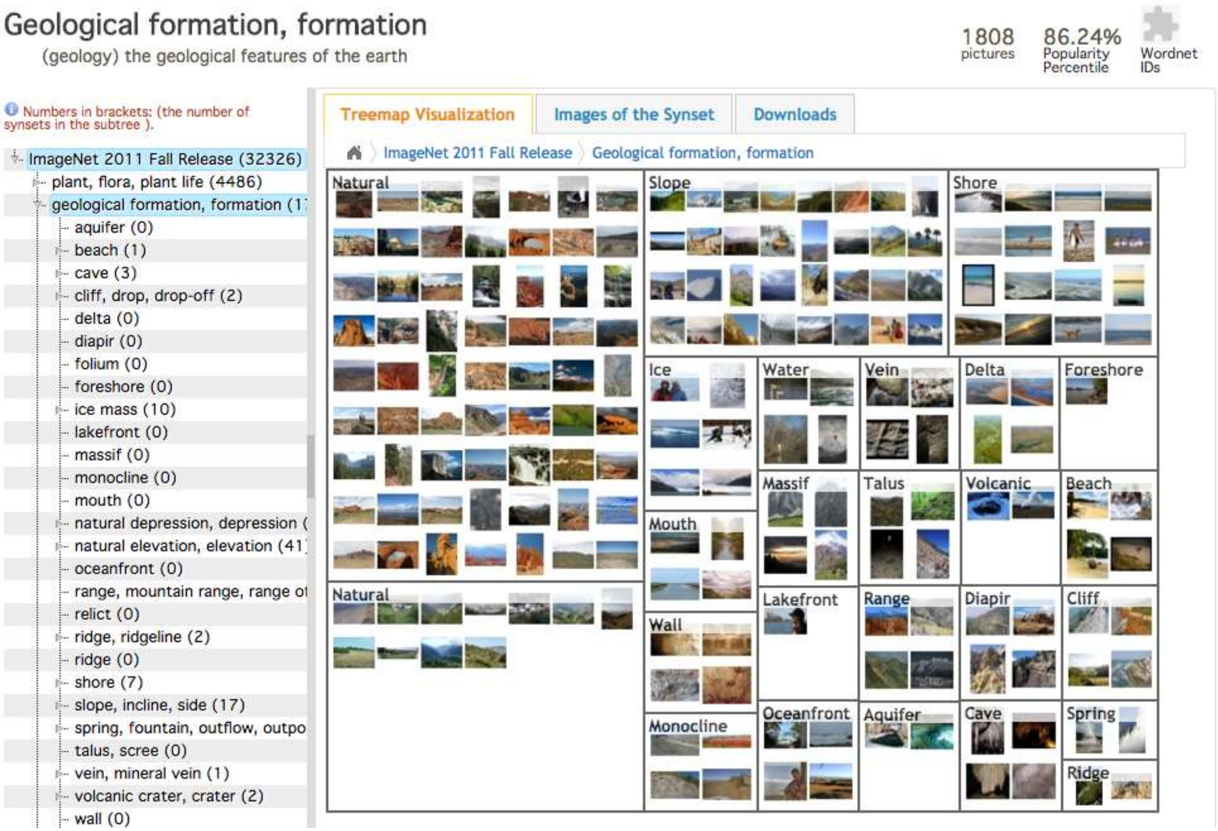
https://www.cs.toronto.edu/~kriz/cifar.html



- Performance saturated
  - Difficult to make significant advancements

ImageNet (2009)

- Better algorithms need better data
- Build a large-scale image dataset
- 2009 CVPR paper:
  - 3.2 million images
  - Annotated by mechanical turk
  - Much larger scale than any previous
- Hierarchical categories



Geological formation, formation

(geology) the geological features of the earth

1808 pictures

86.24% Popularity Percentile

Wordnet IDs

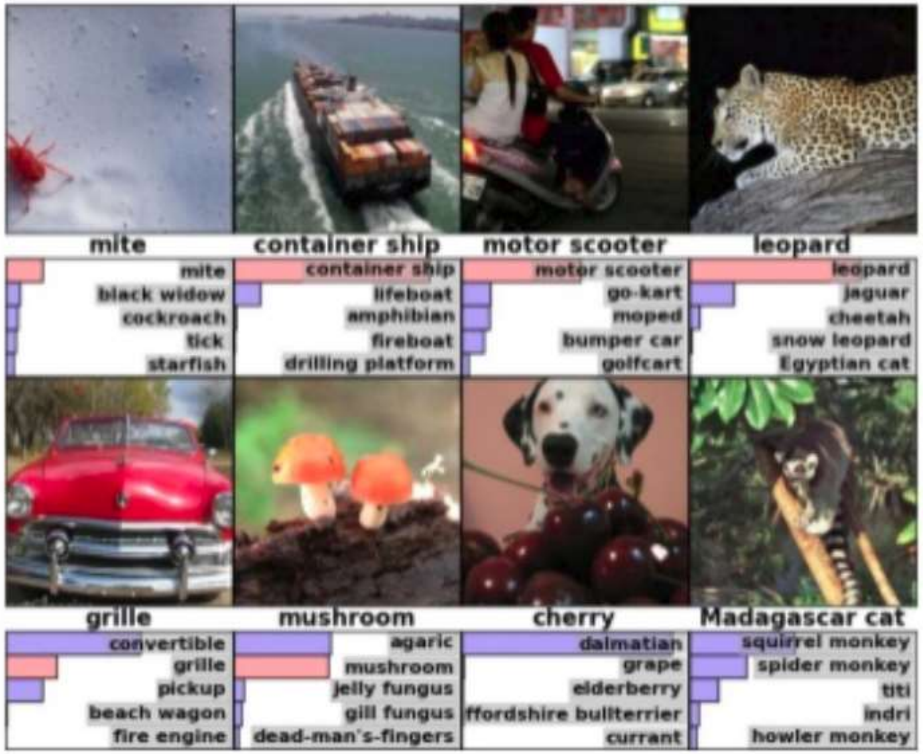
Numbers in brackets: (the number of synsets in the subtree).

- ImageNet 2011 Fall Release (32326)
  - plant, flora, plant life (4486)
  - geological formation, formation (1
    - aquifer (0)
    - beach (1)
    - cave (3)
    - cliff, drop, drop-off (2)
    - delta (0)
    - diapir (0)
    - folium (0)
    - foreshore (0)
    - ice mass (10)
    - lakefront (0)
    - massif (0)
    - monocline (0)
    - mouth (0)
    - natural depression, depression (
    - natural elevation, elevation (41
    - oceanfront (0)
    - range, mountain range, range of
    - relict (0)
    - ridge, ridgeline (2)
    - ridge (0)
    - shore (7)
    - slope, incline, side (17)
    - spring, fountain, outflow, outpo
    - talus, scree (0)
    - vein, mineral vein (1)
    - volcanic crater, crater (2)
    - wall (0)

Deng, J., Dong, W., Socher, R., Li, L. J., Li, K., & Fei-Fei, L. (2009, June). Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on* (pp. 248-255). IEEE.

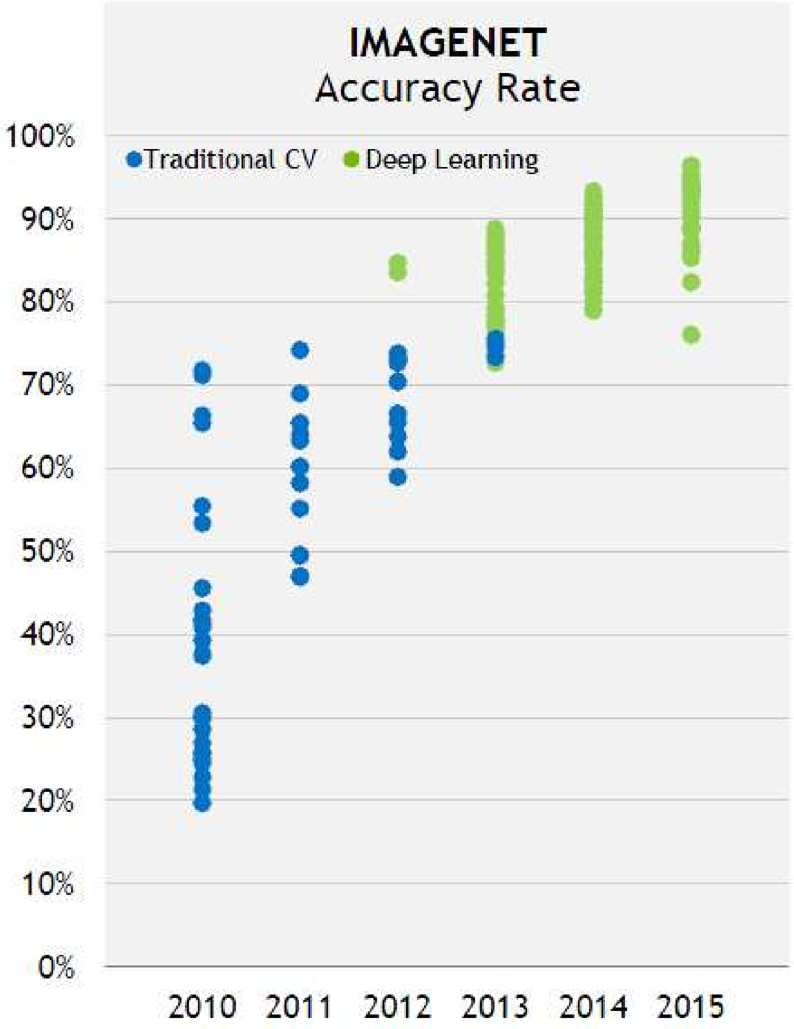
ILSVRC

- ImageNet Large-Scale Visual Recognition Challenge
- First year of competition in 2010
- Many developers tried their algorithms
- Many challenges:
  - Objects in variety of positions, lighting
  - Occlusions
  - Fine-grained categories (e.g. African elephants vs. Indian elephants)
  - …



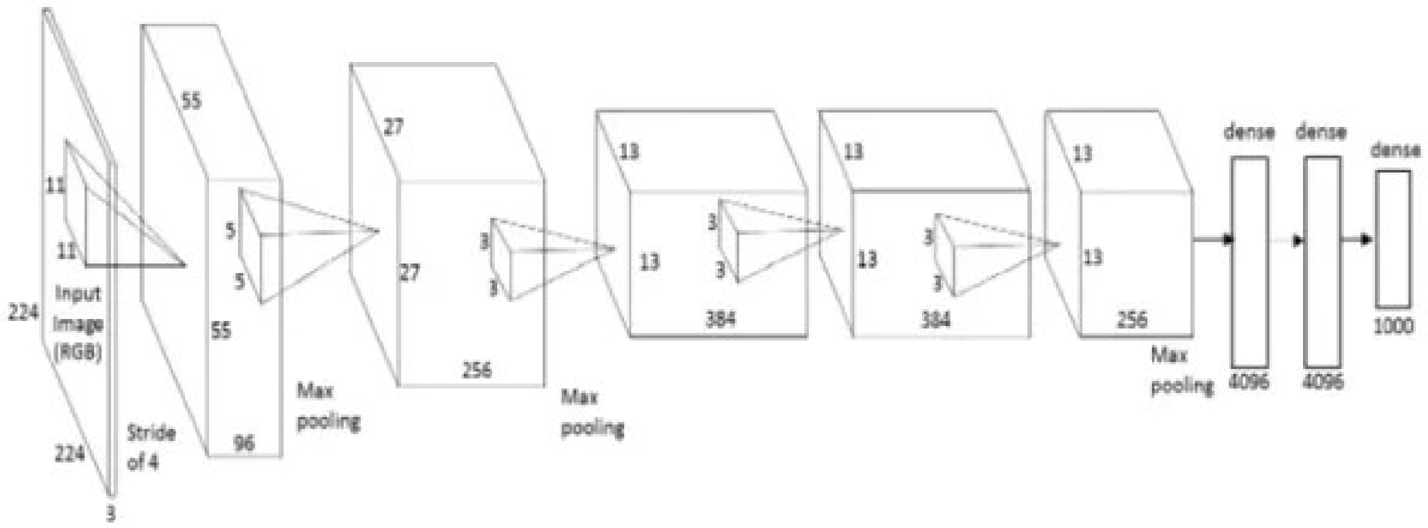
Deep Networks Enter 2012

- 2012: Stunning breakthrough by the first deep network
- “AlexNet” from U Toronto
- Easily won ILSVRC competition
  - Top-5 error rate: 15.3%, second place: 25.6%
- Soon, all competitive methods are deep networks

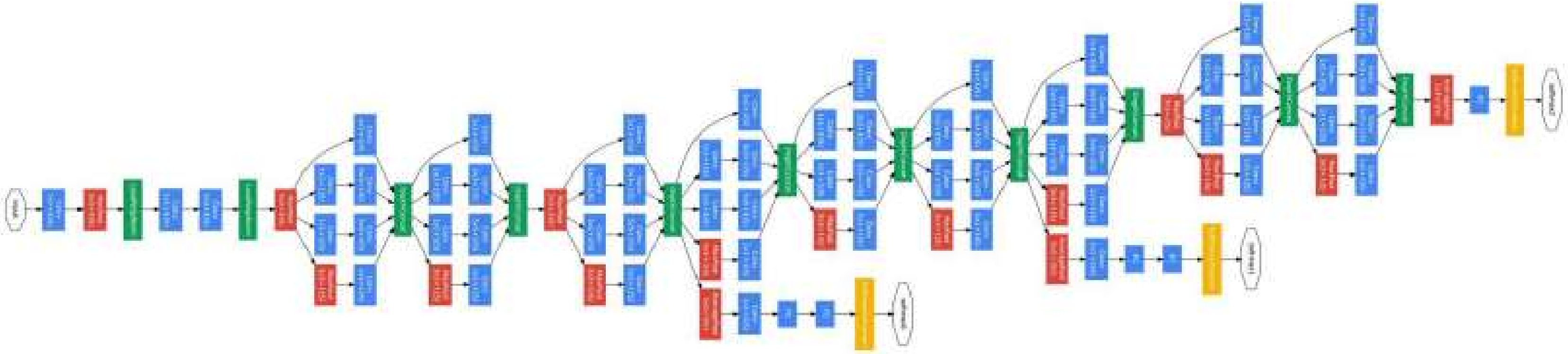


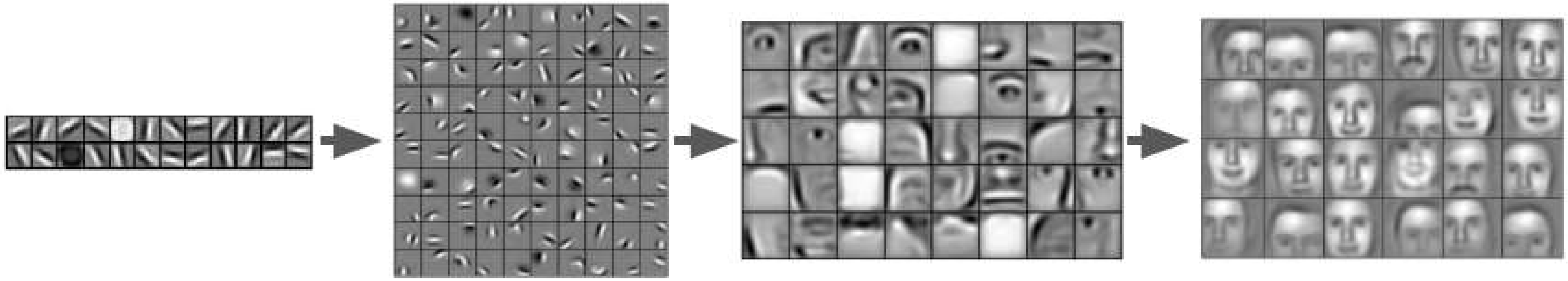
Alex Net

- Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, University of Toronto, 2012
- Key idea: Build a deep neural network
- 60 million parameters, 650000 neurons
- 5 conv layers + 3 FC layers
- Final is 1000-way softmax



Why using many layers?

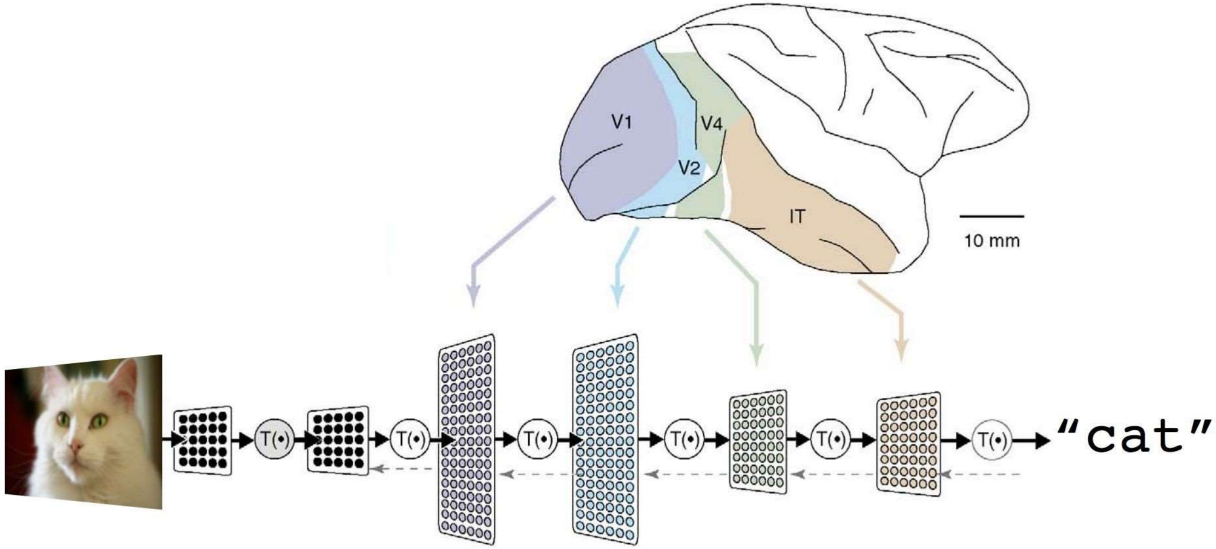




From: Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations, Honglak Lee et al.

Biological Inspiration

- Processing in the brain uses multi-layer processing



Outline

- Supervised learning: General concepts
- Neural network architecture
  - From single perceptron to multi-layer perceptrons
- Convolutional network architecture
  - Why using convolution and many layers
  - Multichannel convolution
  - Pooling
- Deep networks
- Model training
  - Loss functions
  - Stochastic gradient descent: general concept
  - Data Preprocessing and Regularization
- Training, validation and testing and cross validation
- Demo: training a ConvNet classifier

Model Training

- Given a network architecture, how to determine the weights/filters?
- Set up a loss function based on the given task
- Update the network parameters to minimize the loss using gradient descent
  - Stochastic gradient descent (SGD) for large training dataset

Training a Neural Network

- Given data: $(\boldsymbol{x}_i, y_i), i = 1, \ldots, N$
- Learn parameters: $\theta = (W_H, b_H, W_o, b_o)$
  - Weights/filters and biases for hidden and output layers
- Will minimize a loss function: $L(\theta)$

$$\hat{\theta} = \arg\min_{\theta} L(\theta)$$

  - $L(\theta)$ = measures how well parameters θ fit training data $(\boldsymbol{x}_i, y_i)$

Loss Function: Regression

- Regression case:
  - y_i = target variable for sample i
  - Typically continuous valued
- Output layer:
  - $\hat{y}_i = z_{Oi}$ = estimate of y_i
- Loss function: Use L2 loss

$$L(\theta) = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

- For vector $\boldsymbol{y}_i = (y_{i1}, \ldots, y_{iK})$, use vector L2 loss

$$L(\theta) = \sum_{i=1}^{N} \sum_{j=1}^{K} \left(y_{ik} - \hat{y}_{i,k}\right)^2$$

Loss Function: Binary Classification

- Binary classification:
  - Sample: x_i with label $y_i = \{0,1\}$ = class label,
  - Predicted output: $\hat{y}_i = P(y_i = 1|x_i, \theta)$; $1 - \hat{y}_i = P(y_i = 0|x_i, \theta)$
  - Output given by sigmoid on $z_{O,i}$: $\hat{y}_i = \frac{1}{1+e^{-z_{O,i}}}$

- Objective: maximize the likelihood (probability of y_i given x_i for all samples, assuming independence among samples)
  - $P(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta}) = \prod_{i=1}^{N} P(y_i|x_i, \theta)$

- Maximizing the likelihood = minimizing negative log likelihood:

$$\begin{aligned} L(\theta) &= -\sum_{i=1}^{N} \ln P(y_i|x_i, \theta) \\ &= -\sum_{i=1}^{N} \underset{\substack{\uparrow \\ \text{activate when } y_i=1}}{y_i \ln \hat{y}_i} + \underset{\substack{\uparrow \\ \text{activate when } y_i=0}}{(1 - y_i) \ln (1 - \hat{y}_i)} \end{aligned}$$

  - Called the binary cross-entropy

Loss Function: Multi-Class Classification

- Use one-hot-encoding to describe the label y_i

$$y_i = (y_{i1}, \dots, y_{iK}), \quad y_{ik} = \begin{cases} 1 & y_i = k \\ 0 & y_i \neq k \end{cases} \quad k = 1, \dots, K$$

- Output: $\hat{y}_i = \left(\hat{y}_{i,1}, \dots, \hat{y}_{i,K} \right); \ \hat{y}_{i,k} = P(y_i = k | x_i, \theta)$
  - Output given by softmax on $z_{O,i}$: $\hat{y}_{i,k} = \frac{e^{z_{O,ik}}}{\sum_\ell e^{z_{O,il}}}$

- Negative log-likelihood given by:

$$L(\theta) = -\sum_i \ln P(y_i = k | x_i, \theta) = -\sum_i \sum_{k=1}^{K} y_{ik} \ln \hat{y}_{i,k}$$

  - Called the categorical cross-entropy

Selecting the Right Loss Function

- Depends on the problem type
- Always compare final output $\hat{y}_i$ with target y_i

Problem	Target y_i	Output z_{0i}	Loss function	Formula
Regression	y_i = Scalar real	$\hat{y}_i$ = Prediction of y_i Scalar output / sample	Squared / L2 loss	$\sum_i (y_i - \hat{y}_i)^2$
Regression with vector samples	$\boldsymbol{y}_i = (y_{i1}, \dots, y_{iK})$	$\hat{y}_{ik}$ = Prediction of y_{ik} K outputs / sample	Squared / L2 loss	$\sum_{ik} \left(y_{ik} - \hat{y}_{i,k}\right)^2$
Binary classification	$y_\mathrm{i} = \{0,1\}$	$\hat{y}_i$ = Prob. for class 1 Scalar output / sample	Binary cross entropy	$-\sum_i y_i \ln \hat{y}_i + (1 - y_i) \ln (1 - \hat{y}_i)$
Multi-class classification	$y_i = \{1, \dots, K\}$	$\hat{y}_{ik}$ = Prob. for class k K outputs / sample	Categorical cross entropy	$-\sum_i \sum_{k=1}^{K} y_{ik} \ln \hat{y}_{i,k}$

Outline

- Supervised learning: General concepts
- Neural network architecture
  - From single perceptron to multi-layer perceptrons
- Convolutional network architecture
  - Why using convolution and many layers
  - Multichannel convolution
  - Pooling
- Deep networks
- Model training
  - Loss functions
  - Stochastic gradient descent: general concept
  - Data Preprocessing and Regularization
- Training, validation and testing and cross validation
- Demo: training a ConvNet classifier

Training with Gradient Descent

- Neural network training: Minimize loss function

$$\hat{\theta} = \arg\min_{\theta} L(\theta), \qquad L(\theta) = \sum_{i=1}^{N} L_i(\theta, \boldsymbol{x}_i, y_i)$$

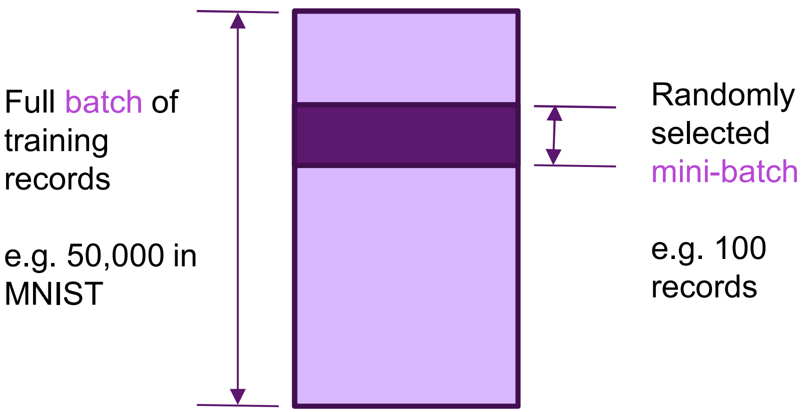
  - $L_i(\theta, \boldsymbol{x}_i, y_i)$ = loss on sample i for parameter θ

- Standard gradient descent:

$$\theta^{k+1} = \theta^{k} - \alpha \nabla L\left(\theta^{k}\right) = \theta^{k} - \alpha \sum_{i=1}^{N} \nabla L_i(\theta^{k}, \boldsymbol{x}_i, y_i)$$

  - Each iteration requires computing N loss functions and gradients
  - Will discuss how to compute later
  - But, gradient computation is expensive when data size N large

Stochastic Gradient Descent



- In each step:
  - Select random small "mini-batch"
  - Evaluate gradient on mini-batch

- For $t = 1$ to N_{steps}
  - Select random mini-batch $I \subset \{1, \ldots, N\}$
  - Compute gradient approximation:
    $$g^t = \frac{1}{|I|} \sum_{i \in I} \nabla L(x_i, y_i, \theta)$$
  - Update parameters:
    $$\theta^{t+1} = \theta^t - \underset{\uparrow \text{ Learning rate}}{\alpha^t} g^t$$

SGD Theory (Advanced, Optional)

- Expectation of Mini-batch gradient = true gradient :

$$E(g^t) = \frac{1}{N}\sum_{i=1}^{N} \nabla L(x_i, y_i, \theta) = \nabla L(\theta^t)$$

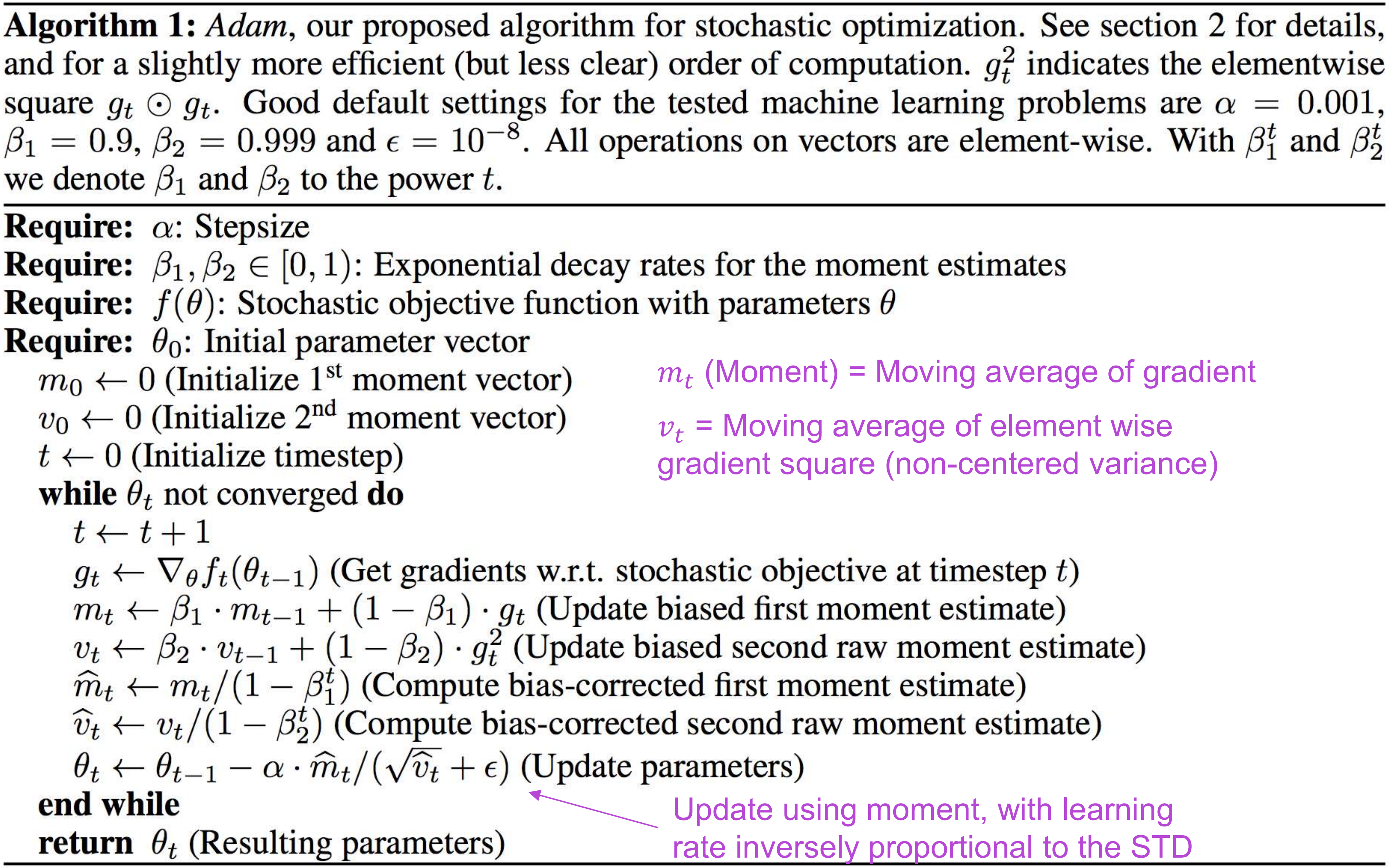
- Hence can write $g^t = \nabla L(\theta^t) + \xi^t$,
  - ξ^t= random error in gradient calculation, $E(\xi^t) = 0$
  - SGD update: $\theta^{t+1} = \theta^t - \alpha^t g^t$, $\theta^{t+1} = \theta^t - \alpha^t \nabla L(\theta^t) - \alpha^t \xi^t$
- Robins-Munro: Suppose that $\alpha^t \rightarrow 0$ and $\sum_t \alpha^t = \infty$. Let $s_t = \sum_{k=0}^{t} \alpha^k$
  - Then $\theta^t \rightarrow \theta(s_t)$ where $\theta(s)$ is the continuous solution to the differential equation:

$$\frac{d\theta(s)}{ds} = -\nabla L(\theta)$$

- High-level take away:
  - If step size is decreased, random errors in sub-sampling are averaged out

SGD Practical Issues

- Terminology:
  - Suppose minibatch size is B. Training size is N
  - Each training epoch includes updates going through all non-overlapping minibatches
  - There are $\frac{N}{B}$ steps per training epoch
- Data shuffling
  - Generally do not randomly pick a mini-batch
  - In each epoch, randomly shuffle training samples
  - Then, select mini-batches in order through the shuffled training samples.
  - It is critical to reshuffle in each epoch!
- How to adapt the learning rate?
  - Many optimization algorithms
  - ADAM is widely used
  - https://moodle2.cs.huji.ac.il/nu15/pluginfile.php/316969/mod_resource/content/1/adam_pres.pdf



Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return $\theta_t$ (Resulting parameters)
```

m_t (Moment) = Moving average of gradient

v_t = Moving average of element wise gradient square (non-centered variance)

Update using moment, with learning rate inversely proportional to the STD

[Adam: A Method for Stochastic Optimization, Kingma & Ba, arXiv:1412.6980]
https://arxiv.org/pdf/1412.6980.pdf

Outline

- Supervised learning: General concepts
- Neural network architecture
  - From single perceptron to multi-layer perceptrons
- Convolutional network architecture
  - Why using convolution and many layers
  - Multichannel convolution
  - Pooling
- Deep networks
- Model training
  - Loss functions
  - Stochastic gradient descent: general concept
  - ➡ Data Preprocessing and Regularization
- Training, validation and testing and cross validation
- Demo: training a ConvNet classifier

Initialization and Data Normalization

- When the loss function is not convex, solution by gradient descent algorithm depends on the initial solution
- Typically weights are initialized to random values near zero.
- Starting with large weights often lead to poor results.
- Normalizing data to zero mean and unit variance allows all input dimensions be treated equally and facilitate better convergence.
- With normalized data, it is typical to initialize the weights to be uniform in [-0.7, 0.7] [ESL]

Regularization: Penalizing large weights

- To avoid the weights get too large, can add a penalty term explicitly, with regularization level λ
- Ridge penalty

$$R(\theta) = \sum_{d,m} w_{H,d,m}^2 + \sum_{m,k} w_{O,m,k}^2 = \|w_H\|^2 + \|w_O\|^2$$

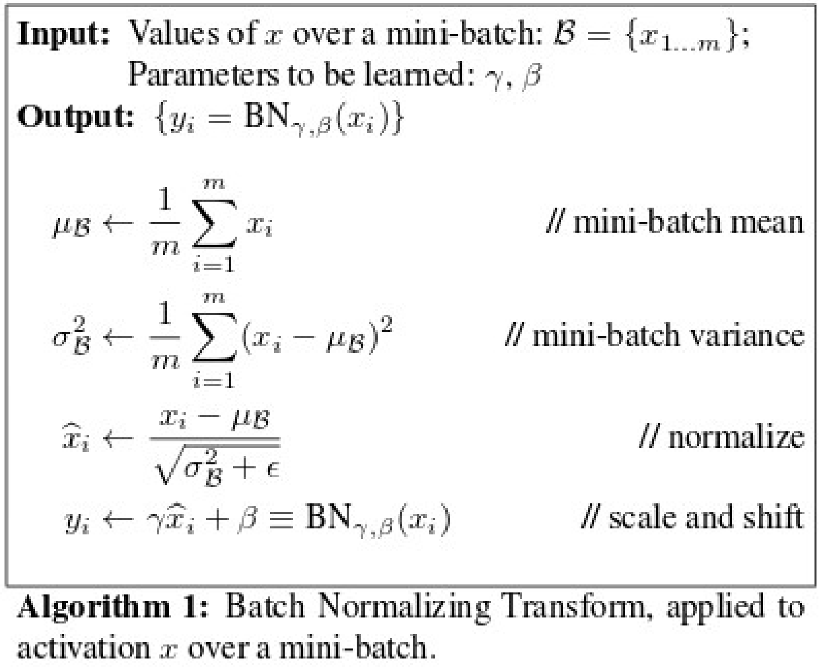
- Total loss

$$L_{reg}(\theta) = L(\theta) + \lambda R(\theta)$$

- Change in gradient calculation
- Typically used regularization
  - L2 = Ridge: Shrink the sizes of weights
  - L1: Prefer sparse set of weights
  - L1-L2: use a combination of both

Regularization: Batch normalization

- In addition to normalize the input data, also normalize the input to each intermediate layer within each batch
  - Invariant to intensity shift
- Then rescale the data using two parameters (to be learnt)
- For each output in a fully connected layer or a feature map in a conv layer, save the training data mean μ and STD σ as well
  - K feature maps: 4K parameters
- Add a Batch Normalization layer before each conv/fully connected layer!
- Can use a higher learning rate and hence converge faster



```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
       Parameters to be learned: $\gamma$, $\beta$
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

    $\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                                  // mini-batch mean
    $\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$           // mini-batch variance
    $\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$                   // normalize
    $y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$               // scale and shift
```

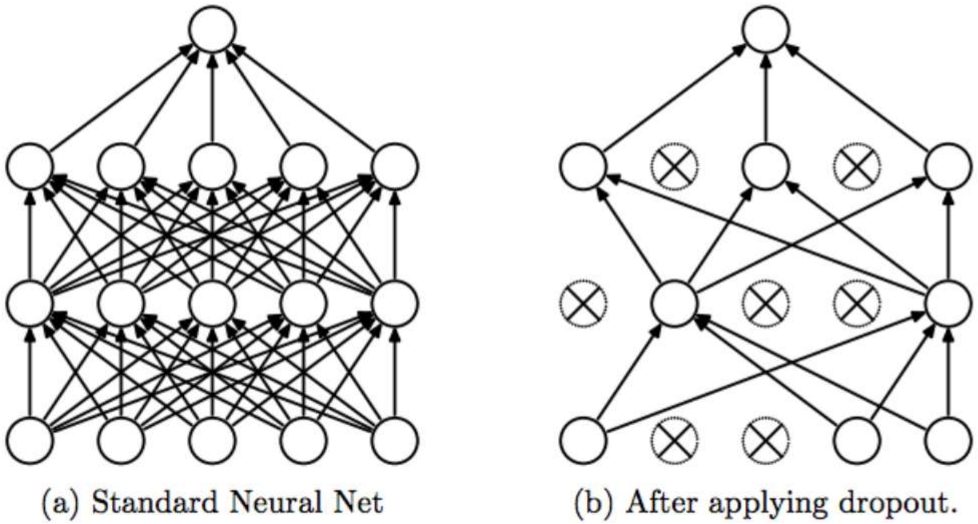
Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Sergey Ioffe, Christian Szegedy: **Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.**
https://arxiv.org/pdf/1502.03167v3.pdf

https://www.youtube.com/watch?v=nUUqwaxLnWs
https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c

Regularization: Dropout

- Drop some percentage (Dropout Rate) of nodes in each layer both in forward and backward pass in each training epoch
- Implemented by setting a certain input elements to this layer to zero
- Dropout forces a neural network to learn more robust features that are useful in conjunction with many different random subsets of the other neurons.
- Reduces overfitting
- Need more epochs to converge but each epoch takes less time



Srivastava, Nitish, et al. ” Dropout: a simple way to prevent neural networks from overfitting” , JMLR 2014

Data Augmentation

- When the training data are limited, can generate additional samples based on the anticipated diversity in the input data
- Image augmentation: by shifting, scaling, rotating the original training images

```
from keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(
    featurewise_center=False,  # set input mean to 0 over the dataset
    samplewise_center=False,  # set each sample mean to 0
    featurewise_std_normalization=False,  # divide inputs by std of the dataset
    samplewise_std_normalization=False,  # divide each input by its std
    zca_whitening=False,  # apply ZCA whitening
    rotation_range=0,  # randomly rotate images in the range (degrees, 0 to 180)
    width_shift_range=0.1,  # randomly shift images horizontally (fraction of total width)
    height_shift_range=0.1,  # randomly shift images vertically (fraction of total height)
    horizontal_flip=True,  # randomly flip images
    vertical_flip=False)  # randomly flip images
```

Practical Tips for Backprop

[from M. Ranzato and Y. LeCun]

- Use ReLU non-linearities (tanh and logistic are falling out of favor).
- Use cross-entropy loss for classification.
- Use Stochastic Gradient Descent on minibatches.
- Shuffle the training samples.
- Normalize the input variables (zero mean, unit variance). More on this later.
- Schedule to decrease the learning rate
- Use a bit of L1 or L2 regularization on the weights (or a combination) But it's best to turn it on after a couple of epochs
- Use dropout for regularization (Hinton et al 2012 http://arxiv.org/abs/1207.0580)
- See also [LeCun et al. Efficient Backprop 1998]
- And also Neural Networks, Tricks of the Trade (2012 edition) edited by G. Montavon, G. B. Orr, and K-R Muller (Springer)

From Fergus: https://cs.nyu.edu/~fergus/teaching/vision/2_neural_nets.pdf

Outline (Part I)

- Supervised learning:
  - General concepts
  - Classification vs. regression
- Neural network architecture
  - From single perceptron to multi-layer perceptrons
- Convolutional network architecture
  - Why using convolution and many layers
  - Multichannel convolution
  - Pooling
- Model training
  - Loss functions
  - Stochastic gradient descent: general concept
  - Data Preprocessing and Regularization
- Training, validation and testing and cross validation
- Demo: training a ConvNet classifier

Training and Testing

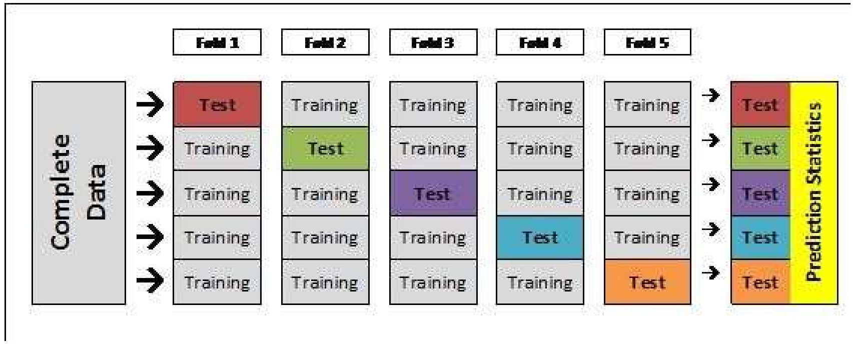
- Goal: use training data to learn a model that works well on unseen data!
- Randomly split the data set to training and testing subsets
  - Training and testing sets should contain the same percentages of different classes as the entire dataset
- Train (using SGD) on the training set and compute both training loss and validation loss (on the testing set) in successive epochs and plot loss curves
  - The training loss should decrease in successive epochs
  - But the validation loss may not!
  - Stop when validation loss starts to increase
  - Use the trained network on the testing set to evaluate performance
- When the training error at convergence is still large, the network architecture does not have enough representation power.
  - Need to modify network architecture.
- When the training error is very small but the validation error is large, the network is overfit.
  - Stop earlier, and if necessary modify network architecture.

Training/Validation/Testing Pipeline

- To evaluate multiple model structures (including different structures and multiple hyperparameters of the same structure, e.g. #layers, # filters, filter sizes)
- Split data to training/validation/testing
  - For each candidate model structure
    - Train on the training set, evaluate on the validation set
  - Determine the structure with best validation performance
  - Retrain the network using training and validation set together using the best structure
  - Evaluate the performance of the trained model on the test set

Cross Validation with Small Dataset

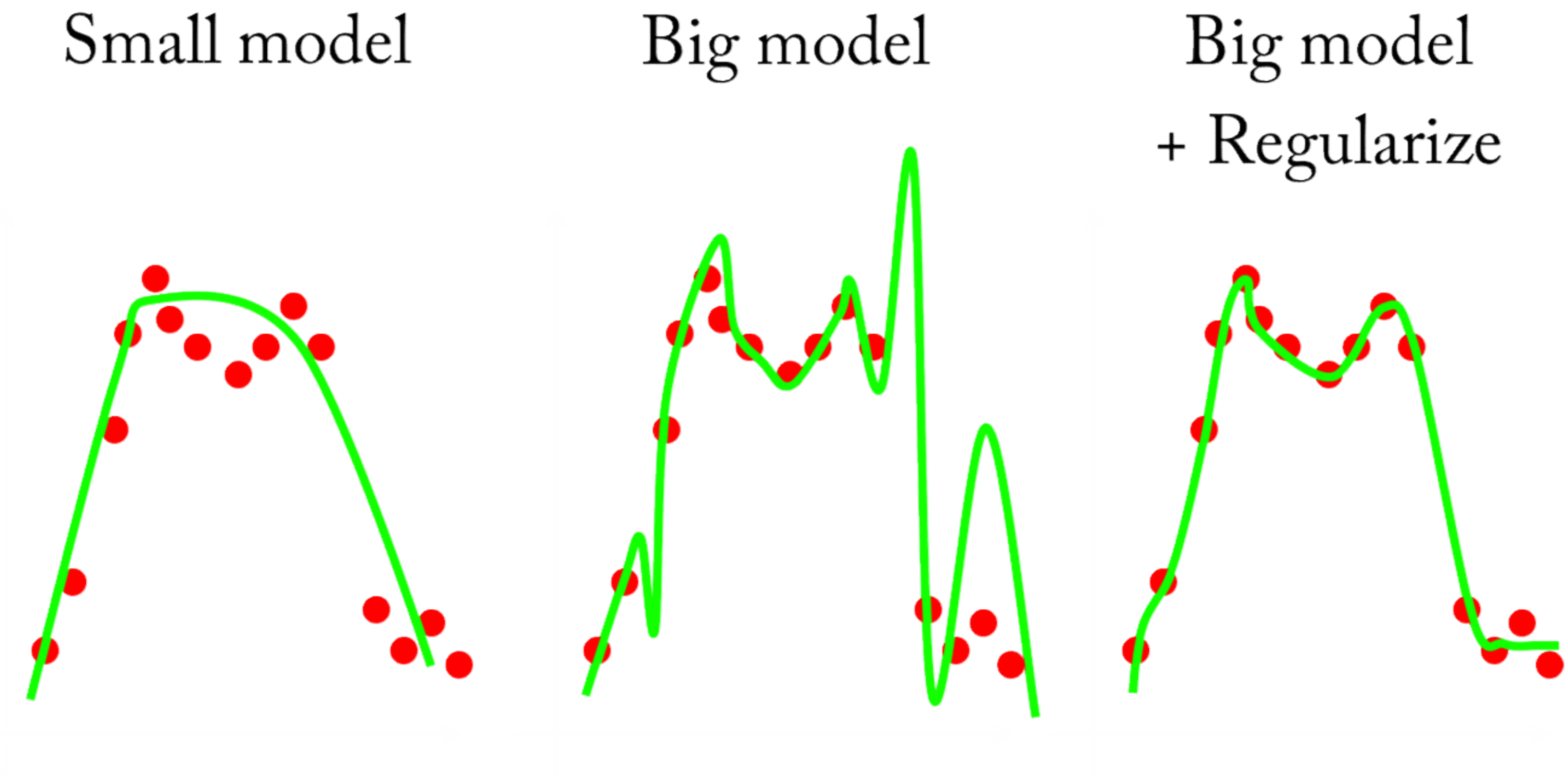
- When the available data set is small
- Partition to training and testing
- Within the training set
  - Divide to K-folds
  - For each candidate models structure
    - Using (K-1) fold for training, and 1 fold for testing;
    - Repeat K times
    - Average performance for all testing folds
  - Determine the best structure with the best average validation performance
  - Train the chosen structure using the entire training set
  - Instead of dividing to K-folds, can randomly draw 1/K percent for validation and use remaining (K-1)/K percent for training, and average validation performance over many random drawings.
- Evaluate the trained model on the testing set (held-out set)
- Training and testing set and each fold/draw within the training set should contain the same percentages of different classes as the entire dataset



Complete Data

Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Prediction Statistics
Test	Training	Training	Training	Training	Test
Training	Test	Training	Training	Training	Test
Training	Training	Test	Training	Training	Test
Training	Training	Training	Test	Training	Test
Training	Training	Training	Training	Test	Test

Model Structure Selection



Small model

Big model

Big model + Regularize

Better to have big model and regularize, than unfit with small model.

From Fergus: https://cs.nyu.edu/~fergus/teaching/vision/2_neural_nets.pdf

Summary: Building a Conv Net

- Define a network structure
  - Conv layer + fully connected layers
  - Add batch normalization and drop out
- Set up a loss function based on the given task
  - Need to add proper regularization on weights
- Partition data to training and testing
  - Proprocess data (zero-mean, unit variance)
  - Augment training data
- Perform stochastic gradient descent on training set
  - Calculate gradient for each batch (to be discussed later)
  - Update the parameters (ADAM optimizer preferred)
  - Evaluate the loss for training and testing set after each epoch
- Observe both training loss and validation loss curves
  - Decide when to stop
  - If training or validation loss is still very large, try to alter network structure

Outline

- Supervised learning: General concepts
- Neural network architecture
  - From single perceptron to multi-layer perceptrons
- Convolutional network architecture
  - Why using convolution and many layers
  - Multichannel convolution
  - Pooling
- Deep networks
- Model training
  - Loss functions
  - Stochastic gradient descent: general concept
  - Data Preprocessing and Regularization
- Training, validation and testing and cross validation
- Demo: training a ConvNet classifier using PyTorch

Deep Learning Zoo

- Torch
- Caffe
- Theano (Keras, Lasagne)
- CuDNN
- Tensorflow
- Mxnet
- Etc.

Recommended Readings

- Material for the machine learning class developed by Sundeep Rangan:
  - https://github.com/sdrangan/introml/blob/master/sequence.md
- Online course by Andrew Ng
  - https://www.coursera.org/learn/neural-networks-deep-learning?specialization=deep-learning
- Many online tutorials
- https://pytorch.org/tutorials/