Image and Video Processing

Convolutional Networks: General Concepts

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
  - Loss functions
- Neural network and training
  - From single perceptron to multi-layer perceptrons
  - Gradient descent for model training, Back propagation
- Convolutional network
  - Why using convolution
  - Multichannel convolution
  - Pooling
  - Receptive field
- Deep networks
  - Large dataset, deep models
  - Stochastic gradient descent: general concept
  - Data Preprocessing and Regularization
- Training, validation and testing and cross validation

Supervised Learning

- Given a dataset with many samples
  - Each sample has an input signal x_i (e.g. an image) and a ground truth output y_i (e.g. a class label or a segmentation map)
- Learning objective
  - Learn a function or model (parameterized by θ) that maps x to y:

    $\hat{y} = f(x; \theta)$
  - 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 parameter θ
    - $R(\theta)$: 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

Loss Function: Regression

- Regression case:
  - y_i = target variable for sample i
  - Typically continuous valued

- Output layer:
  - $\hat{y}_i$ = estimate of y_i

- Loss function: Use L2 loss

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

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

$$L(\theta) = \sum_{i=1}^{N} \|\boldsymbol{y}_i - \widehat{\boldsymbol{y}}_i\|_2^2 = \sum_{i=1}^{N} \sum_{j=1}^{K} (y_{ik} - \hat{y}_{i,k})^2$$

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 some intermediate values $z_{O,i}$: $\hat{y}_i = \frac{1}{1 + e^{-z_{O,i}}}$

- 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:
$$L(\theta) = -\sum_{i=1}^{N} \ln P(y_i | x_i, \theta)$$
$$= -\sum_{i=1}^{N} y_i \ln \hat{y}_i + (1 - y_i) \ln (1 - \hat{y}_i)$$

  activate when y_i=1   activate when y_i=0

  - Called the binary cross-entropy

Loss Function: Multi-Class Classification

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

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

  - Ex. 3 class, class 2: y=[0, 1, 0]

- Output: $\hat{y}_i = (\hat{y}_{i,1}, \ldots, \hat{y}_{i,K}); \hat{y}_{i,k} = P(y_i = k | x_i, \theta)$

  - Output given by softmax on $z_{O,i}$: $\hat{y}_{i,k} = \dfrac{e^{z_{O,ik}}}{\sum_\ell e^{z_{O,il}}}$

- 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 or just 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	$$\sum_{i,k} (y_{ik} - \hat{y}_{i,k})^2$$
Binary classification	$y_i = \{0,1\}$	$\hat{y}_i$ = 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	$$-\sum_i \sum_{k=1}^{K} y_{ik} \ln \hat{y}_{i,k}$$

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
  - Loss functions
- → Neural network and training
  - From single perceptron to multi-layer perceptrons
  - Gradient descent for model training, Back propagation
- Convolutional network
  - Why using convolution
  - Multichannel convolution
  - Pooling
- Deep networks
- Training of deep networks
  - Stochastic gradient descent: general concept
  - Data Preprocessing and Regularization
- Training, validation and testing and cross validation

General Structure of Neural Networks

- Input: $x = (x_1, \cdots, x_d)$

- Hidden layer:
  - Affine transform: $z_H = W_H x + b_H$
  - Activation function: $u_H = g_{act}(z_H)$
    - (element wise operation)

- Output layer:
  - Affine transform: $z_O = W_O u_H + b_O$
  - Output function: $\hat{y} = g_{out}(z_O)$

- This is known as Multilayer Perceptron (MLP)

- Can be used for classification or regression, with different output functions
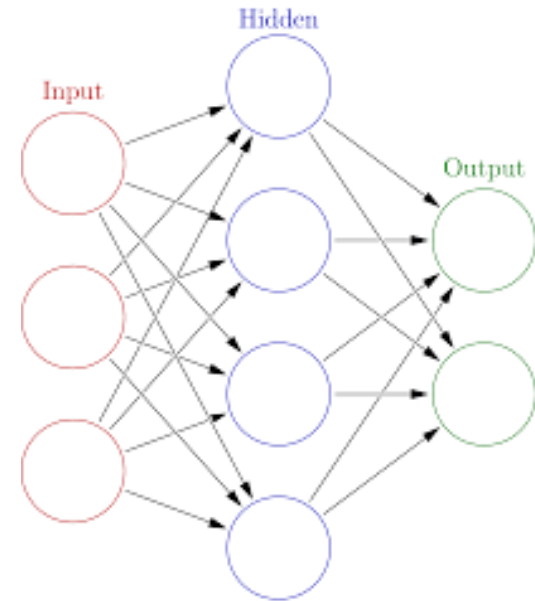
A Single Neuron (Perceptron)

- First combine input variables x_j using an affine transform
  - $z = \sum_j W_j x_j + b$,
  - W_j: Weights; b: Bias
- Then apply a element-wise nonlinear mapping (activation function $g(z)$)
 - $\hat{y} = g(z)$
- This is a simple model of a human neuron.

Figure from https://cs231n.github.io/neural-networks-1/
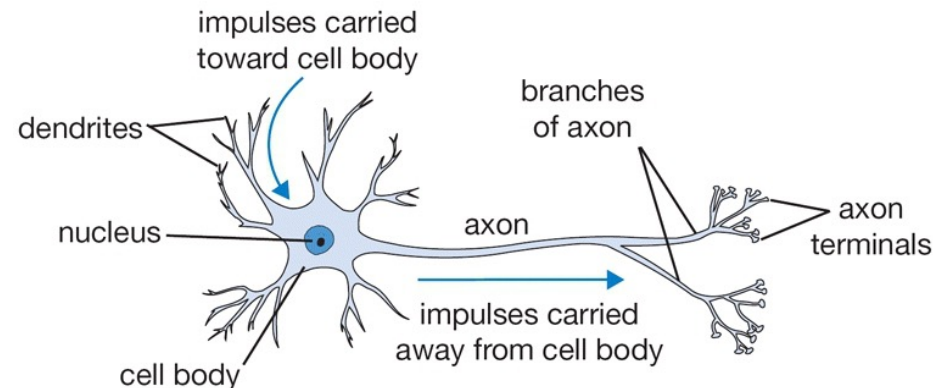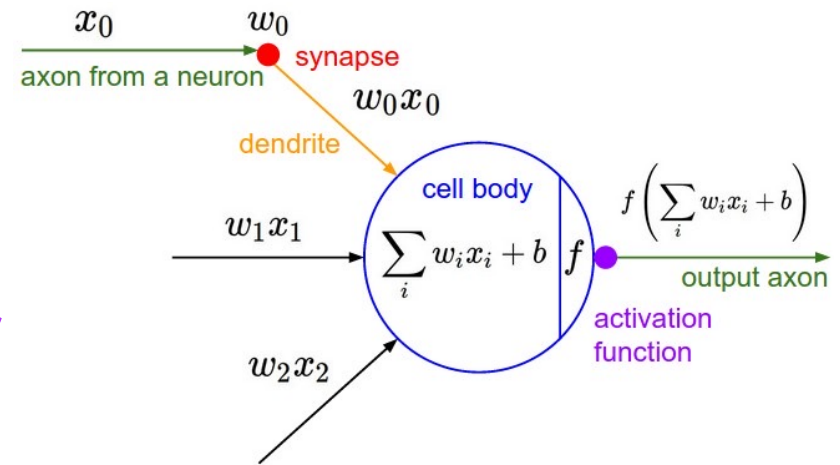
What can a perceptron do?

- For binary classification (= logistic regression)

  – $z = \sum_j W_j x_j + b$

  – $\hat{y} = \frac{1}{1+e^{-z}}$ (sigmoid activation)

  – $\hat{y}$ is probability of belonging to class 1

  – $\hat{y} = \frac{1}{2}$, when $z = 0$

  – $z = 0$ linearly separates all possible points x by a hyperplane defined by W and b

- Works great if the two classes are linearly separable!

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

$z > 0$

$z = 0$
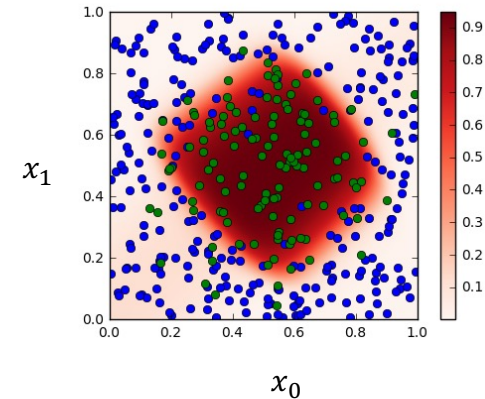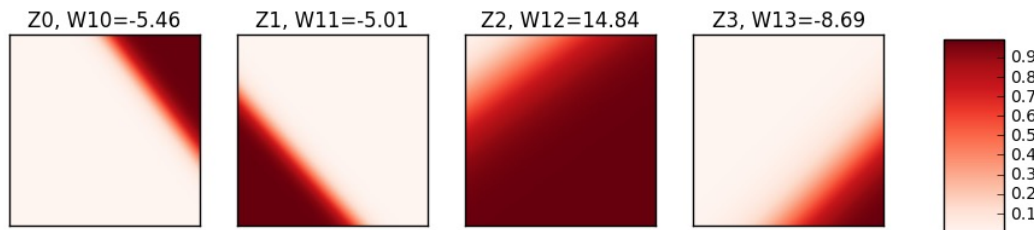
$z < 0$
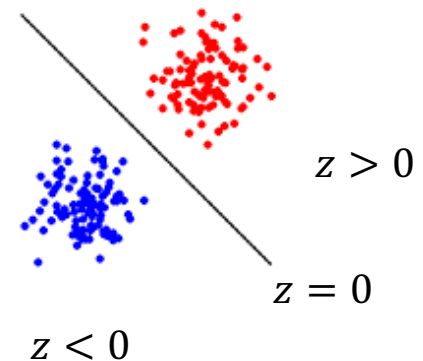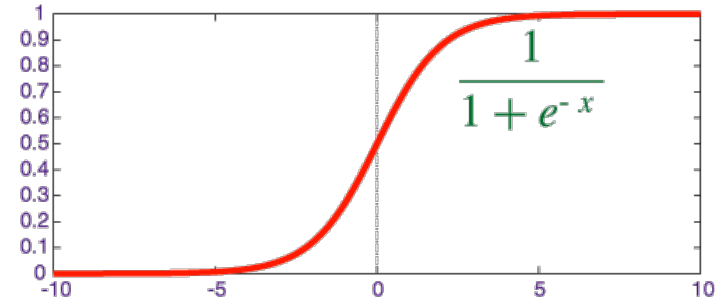
What if not linearly separable?
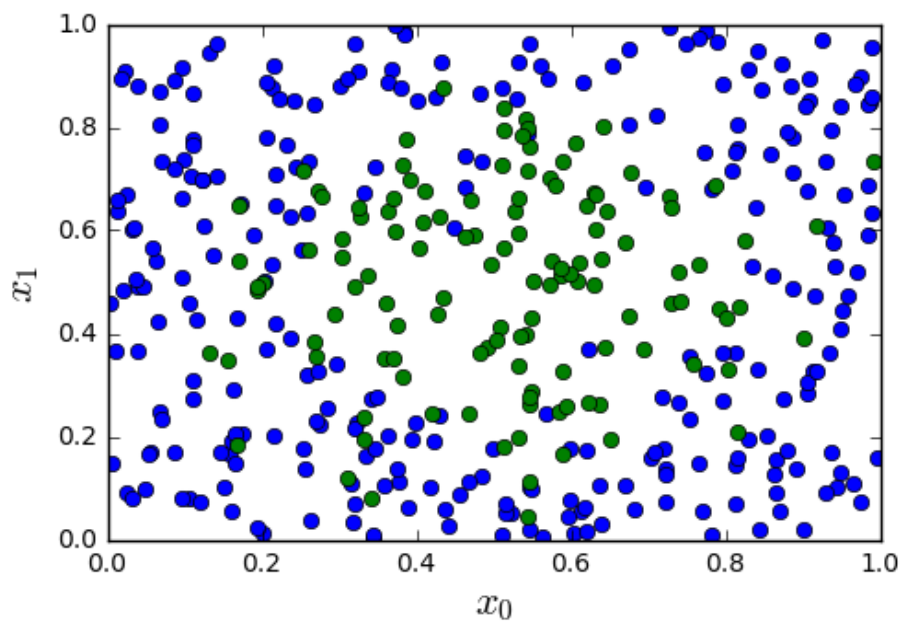
A Two-Stage Classifier
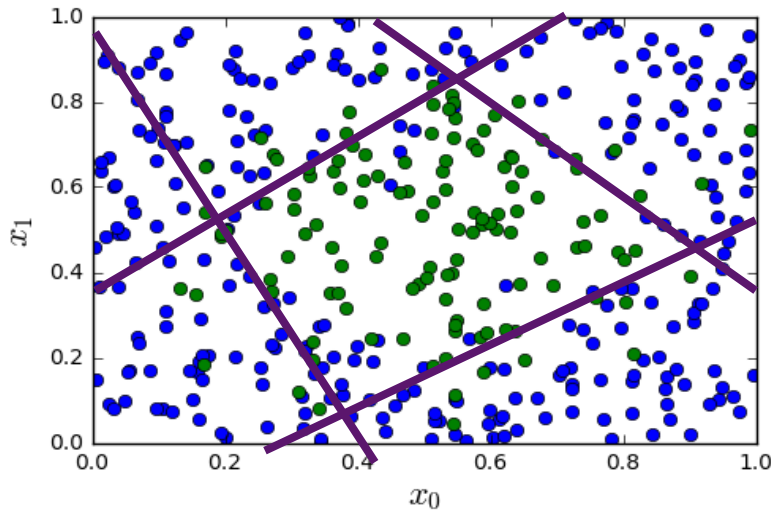
- Input sample: $\boldsymbol{x} = (x_1, x_2)^T$
- First step: Hidden layer
  - Take $N_H = 4$ linear discriminants
    $$z_{H,1} = \boldsymbol{w}_{H,1}^T x + b_{H,1}$$
    $$\vdots$$
    $$z_{H,N_H} = \boldsymbol{w}_{H,M}^T x + b_{H,M}$$
  - Make a soft decision on each linear region
    $$u_{H,m} = g(z_{H,m}) = 1/(1 + e^{-z_{H,m}})$$
- Second step: Output layer
  - Linear step $z_O = w_O^T u_H + b_O$
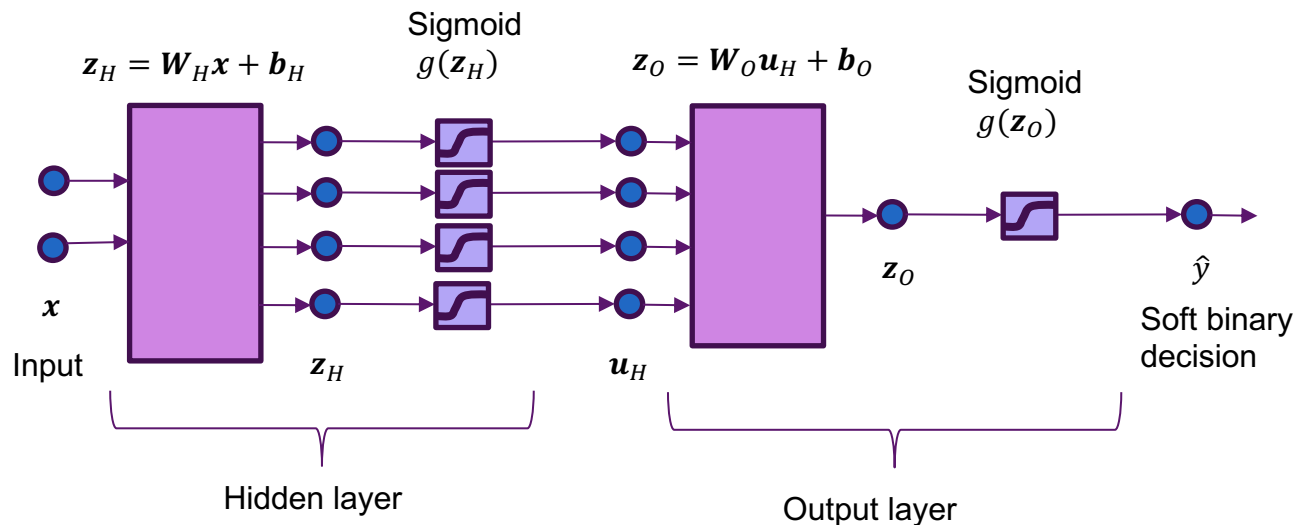 - Soft decision: $\hat{y} = g(z_O)$

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 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: $\boldsymbol{z}_O = \boldsymbol{W}_O \boldsymbol{u}_H + \boldsymbol{b}_O, \hat{y} = g(\boldsymbol{z}_O)$

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)

Two-Layer Neural Net for Multiple Outputs

- Hidden layer: $\boldsymbol{z}_H = \boldsymbol{W}_H \boldsymbol{x} + \boldsymbol{b}_H, \quad \boldsymbol{u}_H = g_{act}(\boldsymbol{z}_H) \ (element\ wise)$
- Output layer: $\boldsymbol{z}_O = \boldsymbol{W}_O \boldsymbol{u}_H + \boldsymbol{b}_O$
- Output: $\widehat{\boldsymbol{y}} = g_{out}(\boldsymbol{z}_O)$
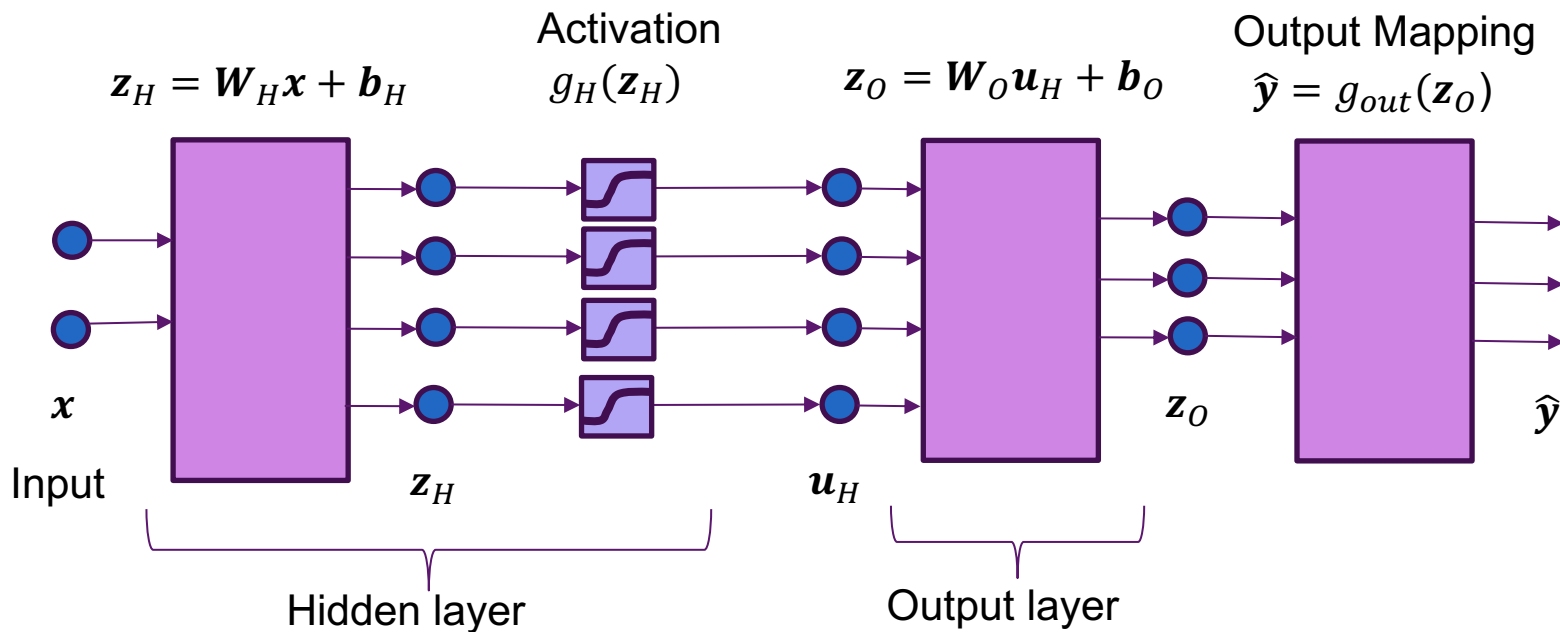
Output Activation

- Last layer depends on type of response
- Binary classification: $y = \pm 1$
  - z_O is a scalar
  - Hard decision: $\hat{y} = \text{sign}(z_O)$ (not differentiable)
  - Soft decision (sigmoid): $\hat{y} = P(y = 1|x) = 1/(1 + e^{-z_O})$ (probability of class 1)
- Multi-class classification: $y = 1, \dots, K$
  - Ground truth label **y** is *K*-dimension
    - One-Hot Encoding, Ex. 3 class, class 2: y=[0, 1, 0]
  - $\mathbf{z}_O = \begin{bmatrix} z_{O,1}, \cdots, z_{O,K} \end{bmatrix}^T$ is a vector
  - $\hat{y}_k = P(y = k|x)$ (probability of class k)

  - Softmax activation: $\hat{y}_k = S_k(\mathbf{z}_O) = \dfrac{e^{z_{O,k}}}{\sum_l e^{z_{O,l}}}$

- Regression: $\mathbf{y} \in R^d$
  - $\hat{\mathbf{y}} = \mathbf{z}_O$ (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
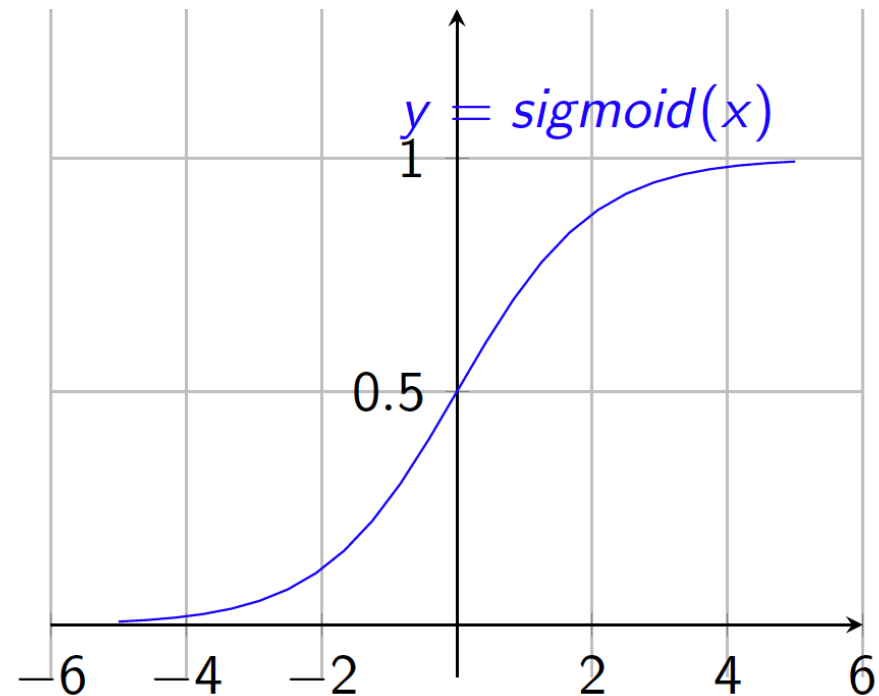
$y = sigmoid(x)$

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.
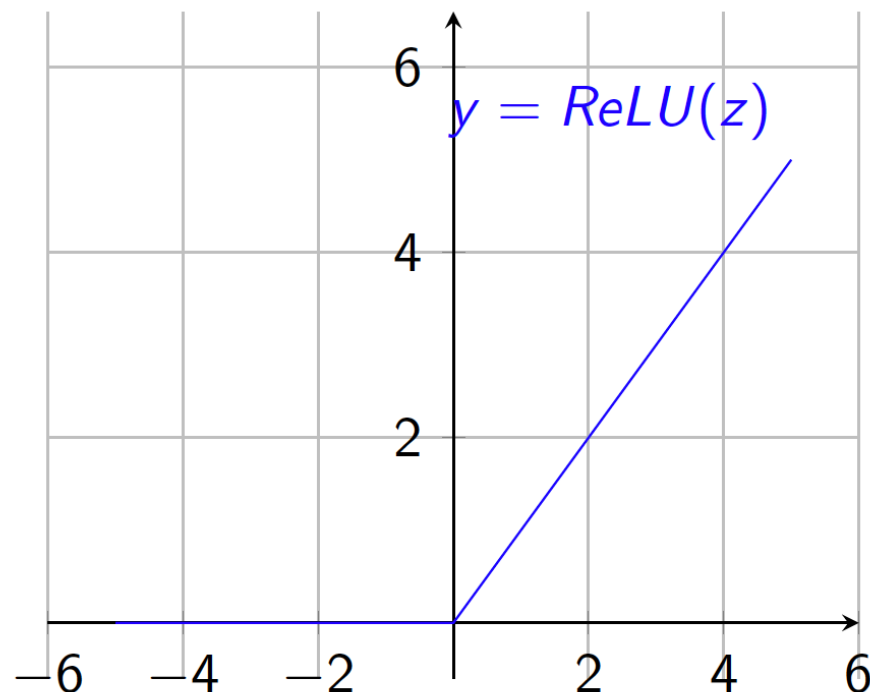
Non-linearities: Tanh

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

$y = tanh(z)$
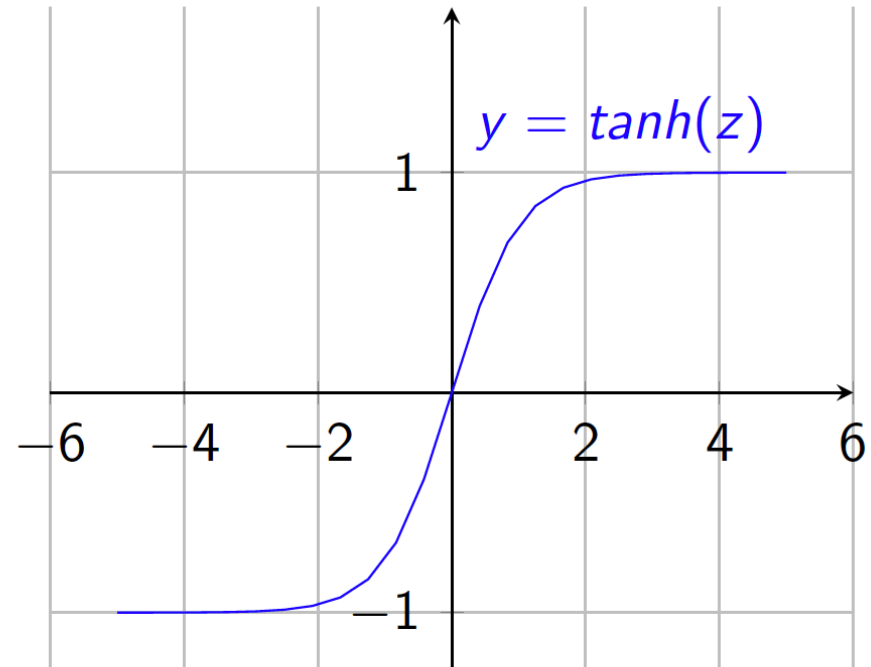
From Fergus: https://cs.nyu.edu/~fergus/teaching/vision/2_neural_nets.pdf

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.

$y = ReLU(z)$

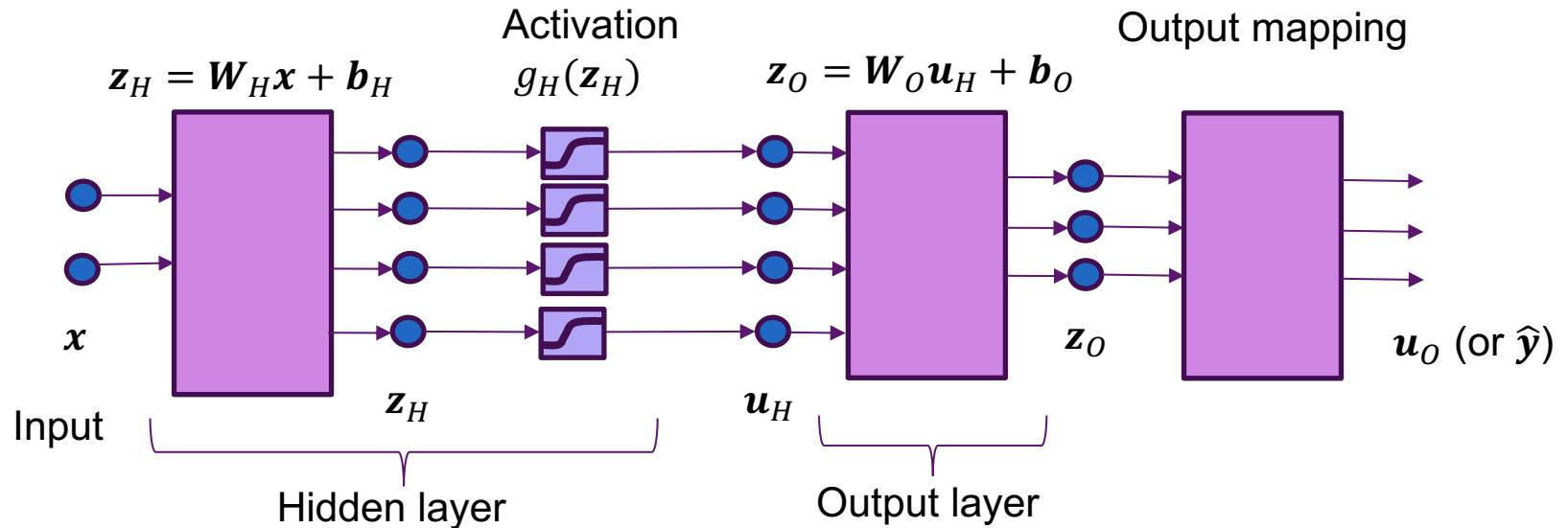From Fergus: https://cs.nyu.edu/~fergus/teaching/vision/2_neural_nets.pdf

Non-linearities: Leaky RELU

- Leaky Rectified Linear z

  $\sigma(z) = 1[z > 0]max(0, x) +$
  $1[z < 0]\underset{min}{\cancel{max}}(0, \alpha z)$

- 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).

$y = PReLU(z)$
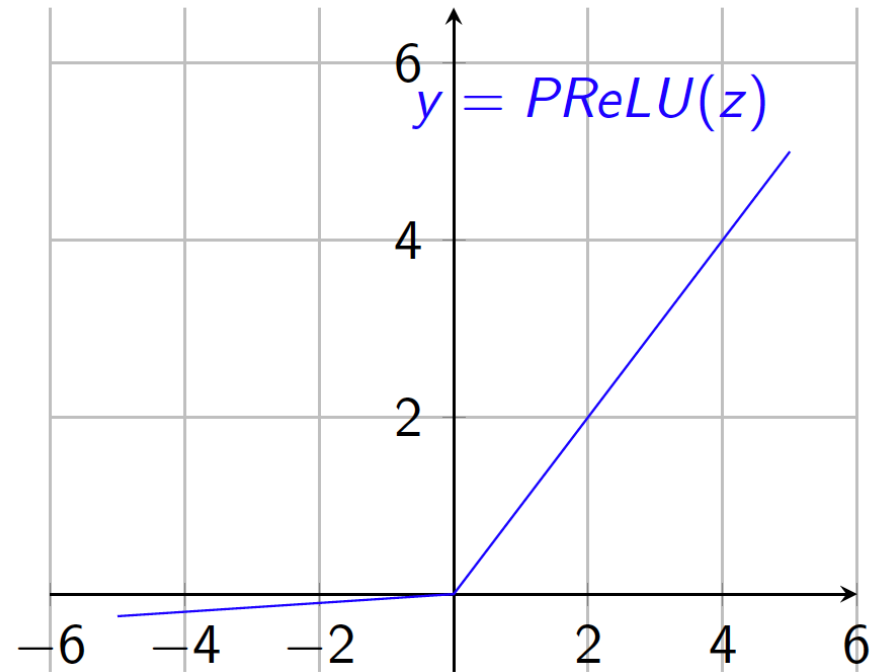
From Fergus: https://cs.nyu.edu/~fergus/teaching/vision/2_neural_nets.pdf

Number of Parameters of a Two Layer Network

Activation $g_H(z_H)$

Output mapping

$z_H = W_H x + b_H$

$z_O = W_O u_H + b_O$

Input

z_H

u_H

z_O

u_O (or $\hat{y}$)

Hidden layer

Output layer

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: a linear classifier (separating two classes by a hyperplane) = logistic regression

- 2 layer (1 hidden layer + 1 output)
  - Theoretically can represent any function with sigmoid activation and sufficient hidden nodes
  - *G. Cybenko, Approximation by Superpositions of Sigmoidal Function, 1989*
  - Michael Nielsen: A visual proof that neural nets can compute any function
  - However, very wide 2-layer MLP may not be better than narrow deep models in practice
  - Beyond 3 or 4 layers are not helpful with fully connected layers
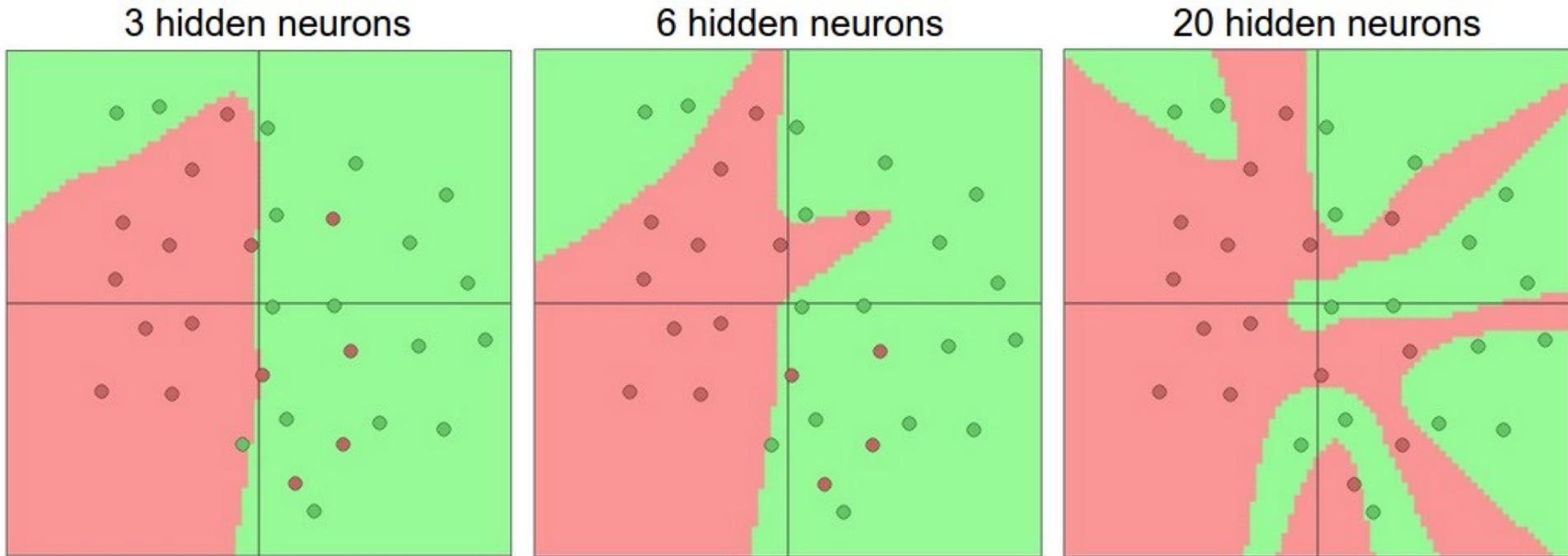   - For conv layers, deeper is better!

How many hidden nodes?

Figure from https://cs231n.github.io/neural-networks-1/

More hidden nodes allow the network to represent more complicated partition between two classes but are also more prone to overfitting (fitting to noise in the training data)

Generally high capacity network with appropriate regularization is preferred over smaller networks

Pop Quiz

- What is supervised learning?

- What is a perceptron and what can it do?

- What is a multi-layer perceptron (MLP) and what can it do?

- How many parameters in a layer with N input, M output?

- Why do we use MLP?

Pop Quiz (w/ answers)

- ## What is supervised learning?
  - Given samples (x_i, y_i), learn a mapping function $y = f(x, \theta)$ (or parameter θ) so that $\hat{y}_i = f(x_i, \theta)$ is close to y_i, $\forall i$. Different methods using different functional forms f( ).

- ## What is a perceptron and what can it do?
  - A perceptron computes a weighted sum of input components plus a bias, followed by a nonlinear activation.
  - It can separate samples by a hyperplane. Perfect for linearly separable classes

- ## What is a multi-layer perceptron (MLP) and what can it do?
  - Each layer contains multiple perceptrons
  - Can separate non-linearly separable clusters

- ## How many parameters in a layer with N input, M output?
  - (N+1)M

- ## Why do we use MLP
  - 2 layers can represent any function, depending on the number of nodes in each layer, but in practice we don't use more than 4 layers
  - MLP mimics how our brain works!

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 $(\boldsymbol{x}_i, y_i)$

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_{Oi}	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	$\boldsymbol{y}_i = (y_{i1}, \ldots, y_{iK})$	$\hat{y}_{ik}$ = Prediction of y_{ik} K outputs / sample	Squared / L2 loss	$\sum_{i,k} (y_{ik} - \hat{y}_{i,k})^2$
Binary classification	$y_i = \{0,1\}$	$\hat{y}_i$ = 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	$y_i = \{1, \ldots, K\}$	$\hat{y}_{ik}$ = Prob. for class k K outputs / sample	Categorical cross entropy	$-\sum_i \sum_{k=1}^{K} y_{ik} \ln \hat{y}_{i,k}$

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:

$$\theta^{k+1} = \theta^k - \alpha \nabla L(\theta^k) = \theta^k - \alpha \sum_{i=1}^{N} \nabla L_i(\theta^k, \boldsymbol{x}_i, y_i)$$
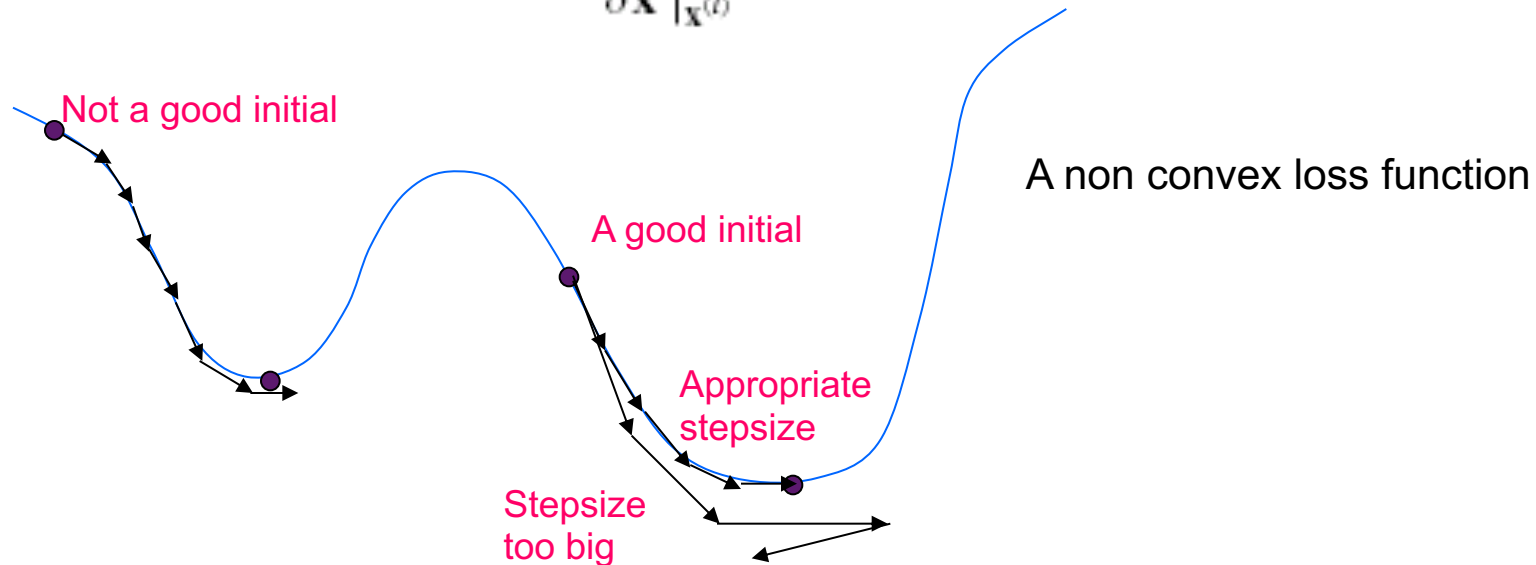
  - Each iteration requires
    - Compute N loss functions (forward pass)
    - Compute gradients (backward pass)
    - Update the network parameters

Gradient Descent Method

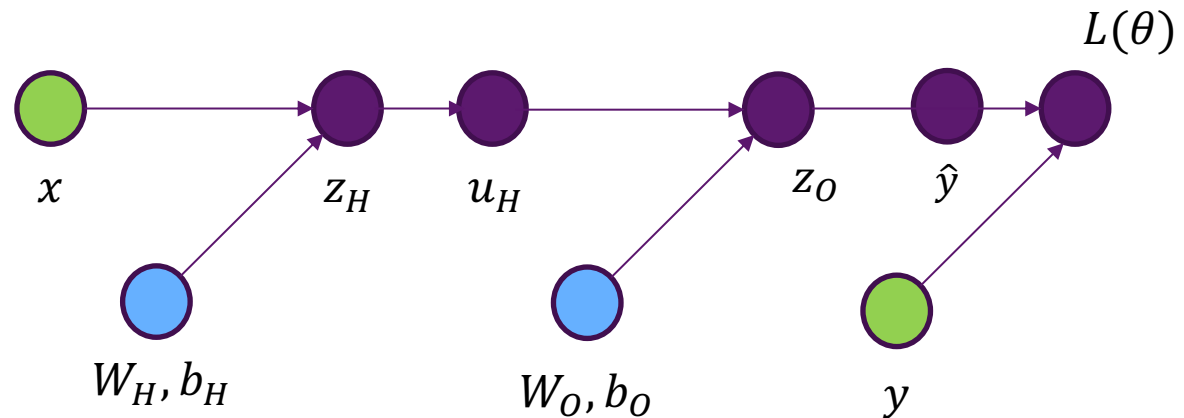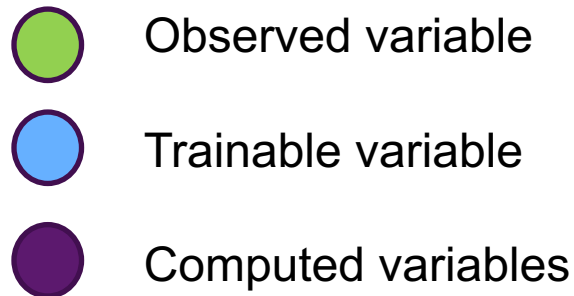- Iteratively update the current estimate in the direction opposite the gradient direction.

$$\mathbf{x}^{(l+1)} = \mathbf{x}^{(l)} - \alpha \left. \frac{\partial J}{\partial \mathbf{x}} \right|_{\mathbf{x}^{(l)}}$$

Not a good initial

A good initial

A non convex loss function

Appropriate stepsize

Stepsize too big

- The solution depends on the initial condition. Reaches the local minimum closest to the initial condition
- Yield optimal solution only if J is convex regardless initial solution

Computation Graph & Forward Pass

- Neural network loss function can be computed via a computation graph

- Sequence of operations starting from measured data and parameters

- Loss function computed via a forward pass in the computation graph

  - $z_{H,i} = W_H x_i + b_H$

  - $u_{H,i} = g_{act}(z_{H,i})$

  - $z_{O,i} = W_O u_{H,i} + b_O$

  - $\hat{y}_i = g_{out}(z_{O,i})$

  - $L = \sum_i L_i(\hat{y}_i, y_i)$

Observed variable

Trainable variable

Computed variables

Gradient Calculation through Back Propagation

- Backpropagation:
  - Compute gradients backwards
  - Work one node at a time using Chain Rule

- First compute all derivatives of all the variables
  - $\partial L / \partial \hat{y}$
  - $\partial L / \partial z_O$ from $\partial L / \partial \hat{y}, \partial \hat{y} / \partial z_O$
  - $\partial L / \partial u_H$ from $\partial L / \partial z_O, \partial z_O / \partial u_H$
  - $\partial L / \partial z_H$ from $\partial L / \partial u_H, \partial u_H / \partial z_H$
- Then compute gradient of parameters:
  - $\partial L / \partial W_O$ from $\partial L / \partial z_O, \partial z_O / \partial W_O$
  - $\partial L / \partial b_O$ from $\partial L / \partial z_O, \partial z_O / \partial b_O$
  - $\partial L / \partial W_H$ from $\partial L / \partial z_H, \partial z_H / \partial W_H$
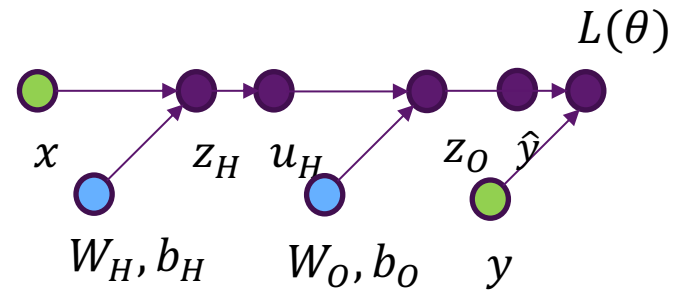 - $\partial L / \partial b_H$ from $\partial L / \partial z_H, \partial z_H / \partial b_H$
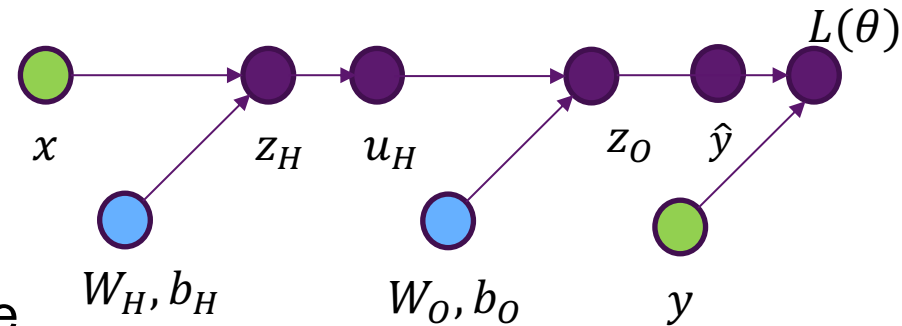
Back-Propagation Example

- 2-layer MLP with M hidden units, single output unit
  - Sigmoid activation, binary cross entropy loss
  - N samples, D input dimension
- Loss for sample i:
  - $L_i = -y_i \ln \hat{y}_i - (1 - y_i) \ln(1 - \hat{y}_i)$
  - $\hat{y}_i = \frac{1}{1+e^{-z_{oi}}}, z_{oi} = \sum_j W_{O,j} u_{H,j} + b_{O,j}$

$L(\theta)$

$x \qquad z_H \quad u_H \qquad z_O \quad \hat{y}$

$W_H, b_H \qquad W_O, b_O \qquad y$

- Gradient backprop:

  - $\frac{\partial L_i}{\partial \hat{y}_i} = -\frac{y_i}{\hat{y}_i} + \frac{1-y_i}{1-\hat{y}_i}, \frac{\partial \hat{y}_i}{\partial z_{O,i}} = -\frac{e^{-z_{oi}}}{1+e^{-z_{oi}}}, \frac{\partial z_{O,i}}{\partial w_{O,j}} = u_{H,j}, \frac{\partial z_{O,i}}{\partial b_{O,j}} = 1$

  - $\frac{\partial L_i}{\partial z_{O,i}} = \frac{\partial L_i}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial z_{O,i}}$

  - $\frac{\partial L_i}{\partial w_{O,j}} = \frac{\partial L_i}{\partial z_{O,i}} \frac{\partial z_{O,i}}{\partial w_{O,j}}$

  - …

Recap: 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:

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

  - Each iteration requires
    - Compute N loss functions (forward pass)
    - Compute gradients (backward pass)
    - Update the network parameters

- Luckily, with modern ML platforms, you can call built-in functions for gradient calculation and updates!

Pop Quizzes

- How do you train a MLP?

- Loss function for classification?

- Loss function for regression?

- How to determine the gradient?

Pop Quizzes

- ## How do you train a MLP?

  - Set up a loss function

  - Derive the gradient

  - Update the model parameters based on the gradient repeatedly until converge

- ## Loss function for classification?

  - Cross entropy

- ## Loss function for regression?

  - Typically Mean Square Error

- ## How to determine the gradient?

  - Using the Chain rule.

Outline

- Supervised learning: General concepts
  - Loss functions
- Neural network and training
  - From single perceptron to multi-layer perceptrons
  - Gradient descent for model training, Back propagation
- Convolutional network
  - Why using convolution
  - Multichannel convolution
  - Pooling
  - Receptive field
- Deep networks
- Training of deep networks
  - Stochastic gradient descent: general concept
  - Data Preprocessing and Regularization
- Training, validation and testing and cross validation

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 depend 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
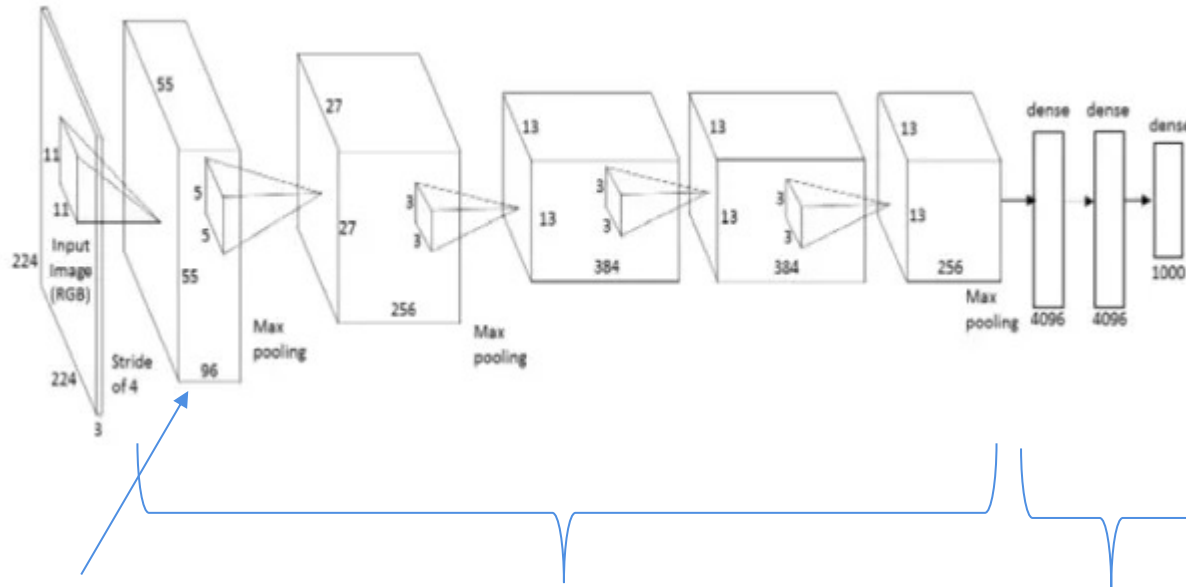
96 feature maps of size 55x55 each

Convolutional layers For feature extraction

Fully connected layers For Classification task

2D convolution with Activation and pooling / sub-sampling

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
- 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)
- This is convolution WITHOUT reversal!
- Should have been called Correlation!

Filter W Image 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 pixels
  - Spatially stationary statistics
    - Translation invariant dependencies
    - Slide the same filter over all input pixels
- LeCun et al. 1989 (LeNet)
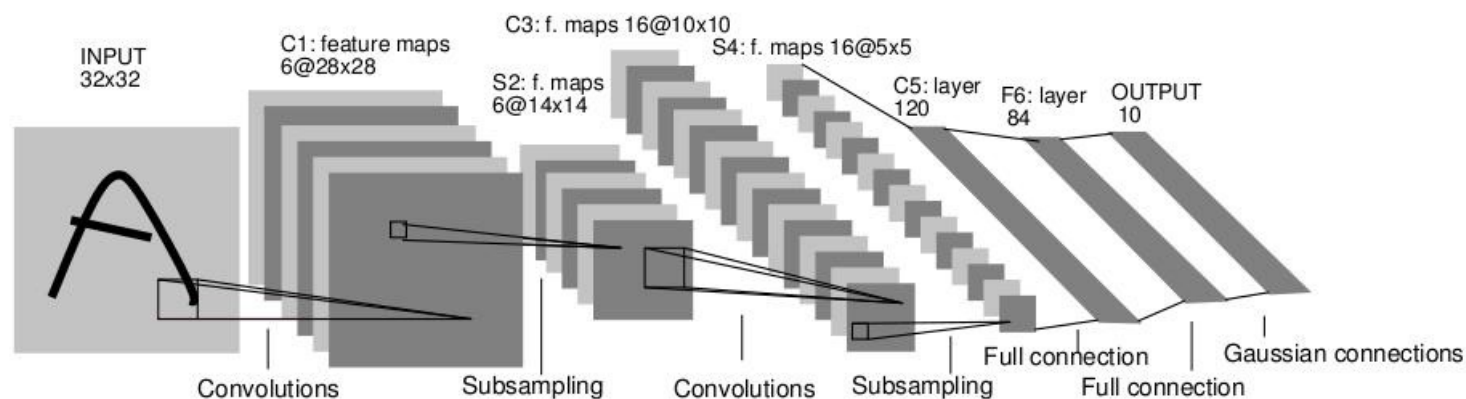
From Fergus: https://cs.nyu.edu/~fergus/teaching/vision/3_convnets.pdf

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 Effect (Conv. w/o reversal)

Valid region   Input image   Output depends on both inside and outside boundary pixels (not valid)

N-K+1   K

Filter mask

N   N   N+K-1

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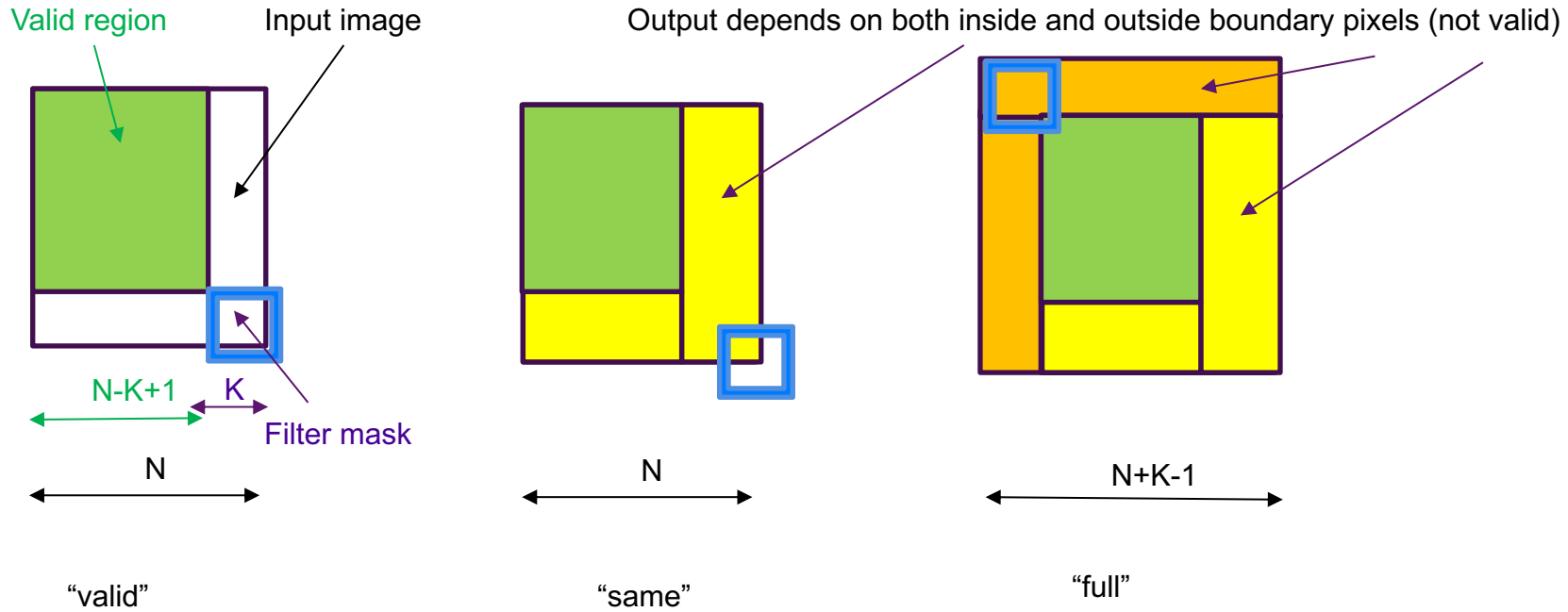
$$z[n_1, n_2] = \sum_{k_2=0}^{K_2-1} \sum_{k_1=0}^{K_2-1} w[k_1, k_2] x[n_1 + k_1, n_2 + k_2]$$

When the desired output size is "same" as input, zero pad before conv. to the right and bottom with a border width=[K-1]!
"Full" is rarely used. Needs padding on all four sides.

Boundary Handling

- Suppose inputs are
  - x, size $N_1 \times N_2$, w: size $K_1 \times K_2$, $K_1 \le N_1$, $K_2 \le N_2$
  - $z = x * w$ (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: $0 \le n_1 < N_1 - K_1 + 1$, $0 \le n_2 < N_2 - K_2 + 1$
  - Requires no zero padding
- Same mode: Output size $N_1 \times N_2$
  - Usually use zero padding of input images on the right and bottom
- Full mode: Output size $(N_1 + K_1 - 1) \times (N_2 + K_2 - 1)$
  - Not used often in neural networks
  - Need zero padding on all sides

Multi-Channel Convolution

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

Multi-Channel Convolution

N_{in} channels N_{out} channels
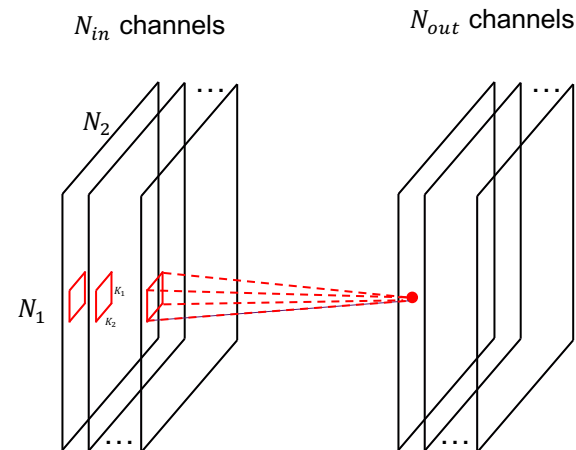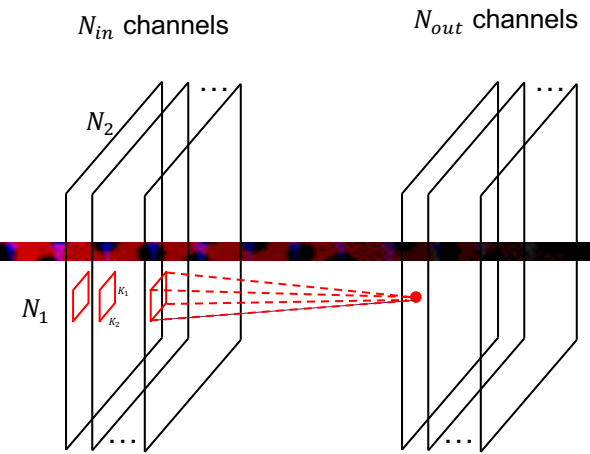
- For each output channel m, perform 2D convolution on each input channel and add results, then add bias

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

- 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
- # filter parameters in each layer
    - Filter coefficients: $K_1 K_2 N_{in} N_{out}$, Biases: N_{out}
    - Total: $(K_1 K_2 N_{in} + 1) N_{out}$
- Computation complexity:
    - $N_1 N_2 K_1 K_2 N_{in} N_{out}$ multiplications (without stride)
    - Significantly more than conventional conv. if you have many input and output channels!

Pooling

- Pooling
  - Aggregate $S \times S$ pixels in each output channel to 1 pixel
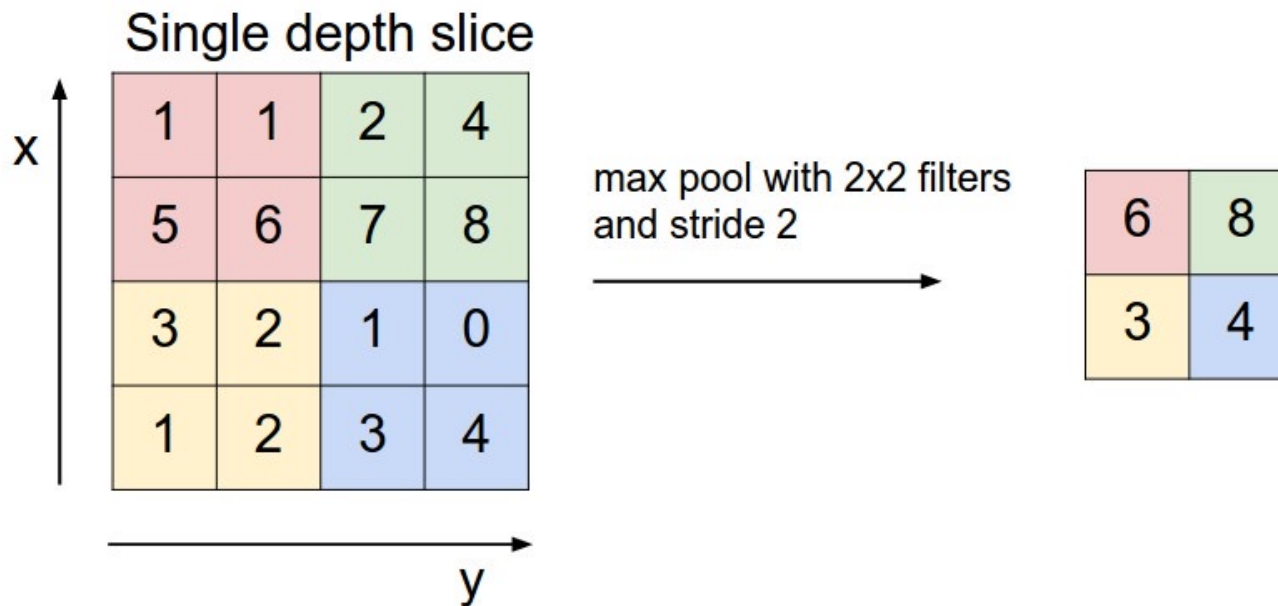 - Different methods (max, average, down-sampling)

Single depth slice

1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

max pool with 2x2 filters and stride 2

6	8
3	4

Figure from https://cs231n.github.io/convolutional-networks/

Strided Convolution

- If we are going to down sample by a factor of 2 after convolution, we don't have to compute the convolution results for every pixel. Instead we could compute convolution for every other sample (both horizontally and vertically)

- Stride-s means to skip (s-1) samples

- For 2D: skip in both directions

- Input feature map size: N_1, N_2

- Output feature map size: M_1, M_2, $M_1 = \frac{N_1}{s}$, $M_2 = \frac{N_2}{s}$

- Computations:
    - $N_1 N_2 K_1 K_2 N_{in} N_{out}$ multiplications (without stride)
    - $M_1 M_2 K_1 K_2 N_{in} N_{out}$ multiplications (with stride), reduce by a factor of s^2

- Number of parameters?
    - Does not change!

- In practice, it may be better to use non-strided conv, followed by max pooling

Receptive Field

3x3 Conv without stride

Input

R=3x3

R=5x5
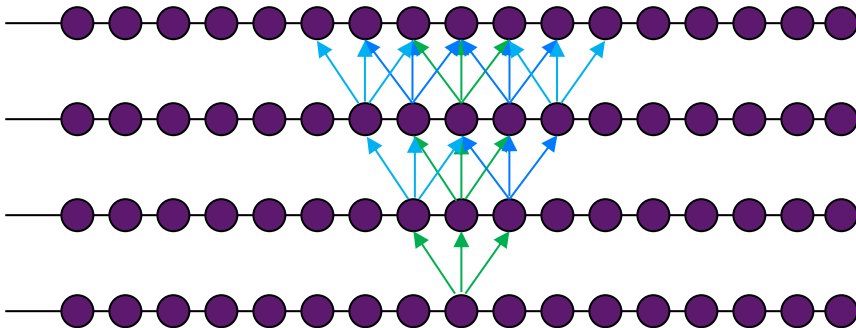
R=7x7

Receptive field =
The number of pixels in the input image that affects one pixel at the output at (also called perceptive field)

A large receptive field is desired to use global information for decision making.

3x3 Conv with stride 2

R=3x3

R=7x7
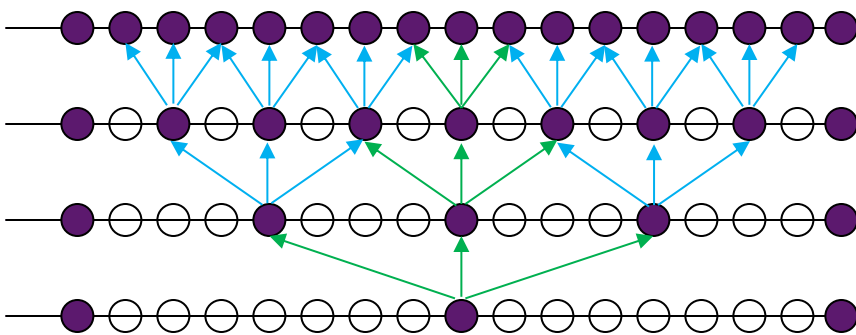
R=15x15

With stride 2 (or stride 1 followed by max pool), we can reach a receptive field of 15 with 3 layers. Without stride, it will take 7 Layers! The two networks have the same number of parameters

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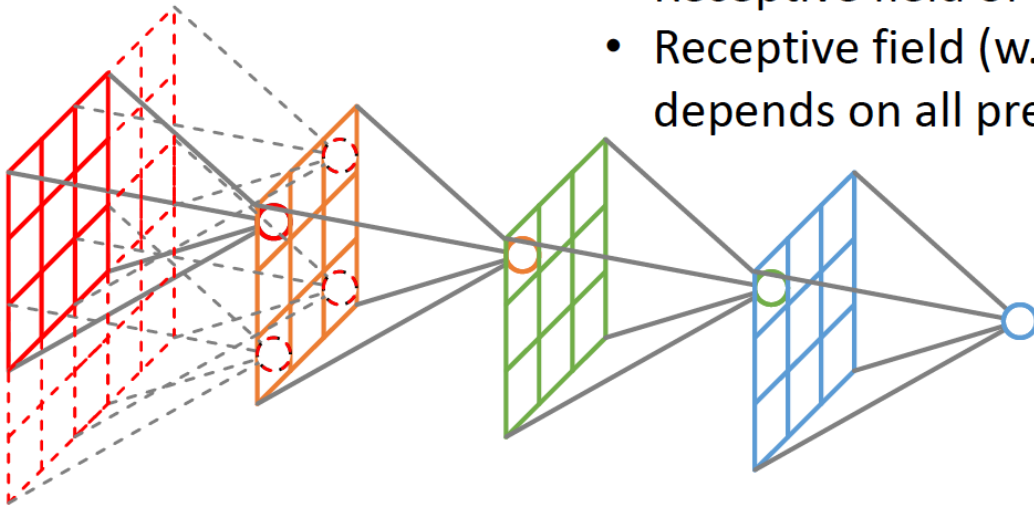
- Correspondence between a feature map pixel and an image pixel is not unique
- Map a feature map pixel to the center of the receptive field on the image in the SPP-net paper

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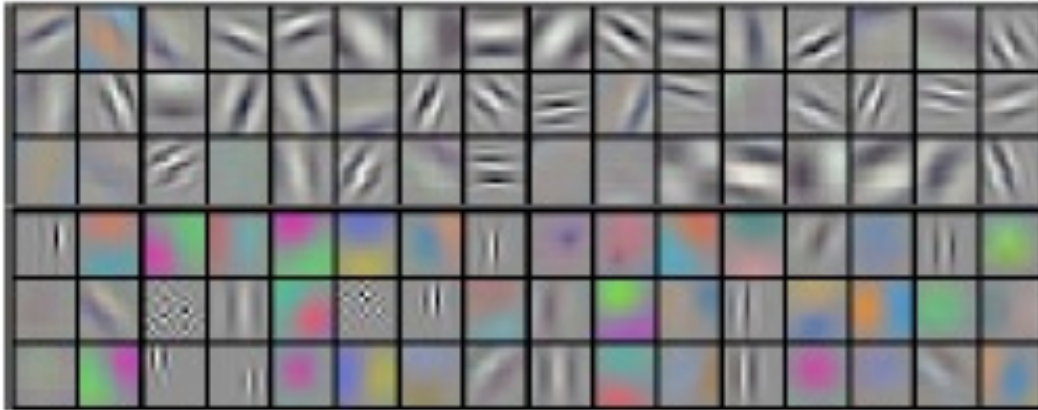
- AlexNet first layer
  - 96 filters
  - Size 11 x 11 x 3
  - Applied to image of 224 x 224 x 3
- What do these learned features look like?
- Selective to basic low-level features
  - Curves, edges, color transitions, …

What filter size to use?

- Compare 1 layer with 7x7 filters vs. 3 layers with each with 3x3 filters plus nonlinear activation

- Same receptive field

- Which one is better? Assuming input and output both have 16 channels

- 1 layer with 16 7x7 filters, feature map size 256x256
  - 16x16x7x7+16=12560 parameters and only 1 non-linear activation
  - 16x16x7x7x256x256=822 million multiplications

- 3 layers with 3x3 filters
  - 3* (16x16x3x3+16)= 6960 parameters and 3 non-linear activations
  - 3* (16x16x3x3x256x256)=452 million multiplications

- Using more layers each with small filters enjoys fewer parameters, less computations and higher presentation power (more nonlinearity)!
  - Benefit of deeper networks with small filters

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 N1xN2, Nout output channels with size M1xM2
  - Fully connected network: Nin*Nout*N1*N2*M1*M2+Nout*M1*M2
  - 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).

Pop Quizzes

- Why do we use convolutional networks
- What is the # of parameters for a layer with Nin input channel, Nout output channels with filter size of KxK
- What is the receptive field of a conv. layer ?
- Why do we want a large receptive field?
- Why do we do pooling?
- What is strided convolution?

Pop Quizzes (w/ Answers)

- Why do we use convolutional networks
  - Can extract local features/patterns that may appear any where
  - Follow the translational invariance operation of the eye/brain
  - Can significantly reduce the network parameters for images or other spatio-temporal signals
- What is the # of parameters for a layer with Nin input channel, Nout output channels with filter size of KxK
  - Nin * Nout * K*K (filters) + Nout (bias)
- What is the receptive field of a conv. network ?
  - The number of pixels in the input image that affects one pixel in the output image
- Why do we want a large receptive field?
  - To be able to see more global information
- Why do we do pulling?
  - To aggregate information spatially and to reduce the spatial size of subsequent layers
  - Require fewer layers to reach the same receptive field
- What is strided convolution?
  - Equivalent to convolution followed by down-sampling, but avoiding unnecessary computations

How do we train conv net?

- Can use the same approach for training the neural net, but the network parameters include the filters and biases, in addition to any fully connected layers
- Need to take gradients with respect to the filter coefficients
- Nontrivial if we have many many layers (i.e. deep networks)
- Computationally expansive if we have many many training samples
- The success of deep networks is propelled by
  - Large dataset (images from the internet and cloud sourcing annotation)
  - "smart" gradient descent algorithms (Stochastic Gradient Descent)
  - advances in computing (GPU and parallel computing)
  - Open source development platforms
 - Open source trained models and codes

Outline

- Supervised learning: General concepts
  - Loss functions
- Neural network and training
  - From single perceptron to multi-layer perceptrons
  - Gradient descent for model training, Back propagation
- Convolutional network
  - Why using convolution
  - Multichannel convolution
  - Pooling
  - Receptive field
- Deep networks
  - Large dataset, deep models
  - Stochastic gradient descent: general concept
  - Data Preprocessing and Regularization
- Training, validation and testing and cross validation

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)

- Performance saturated
  - Difficult to make significant advancements

https://www.cs.toronto.edu/~kriz/cifar.html
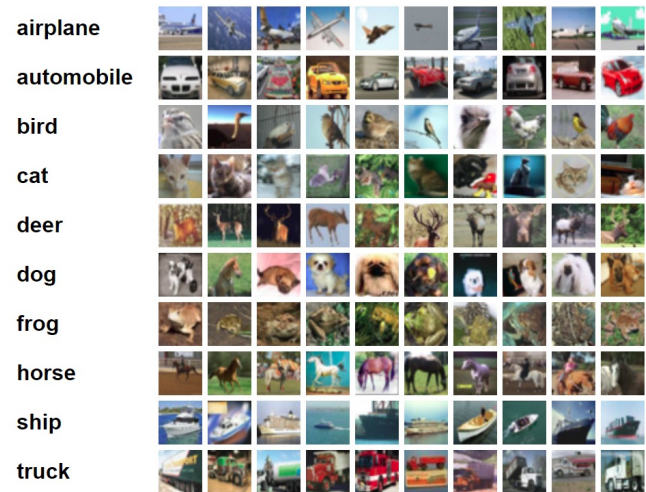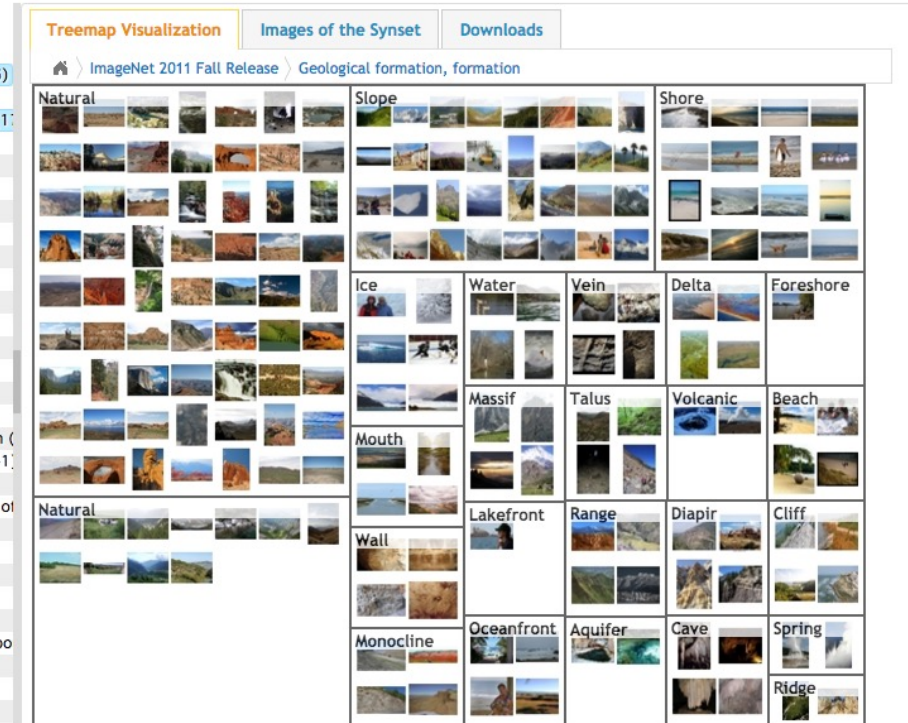
ImageNet (2009)

- 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
  - 1000 classes
    - Hierarchical categories

- Continuously growing
  - 14 million images as of 3/2021, 21,841 classes

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)
  - …
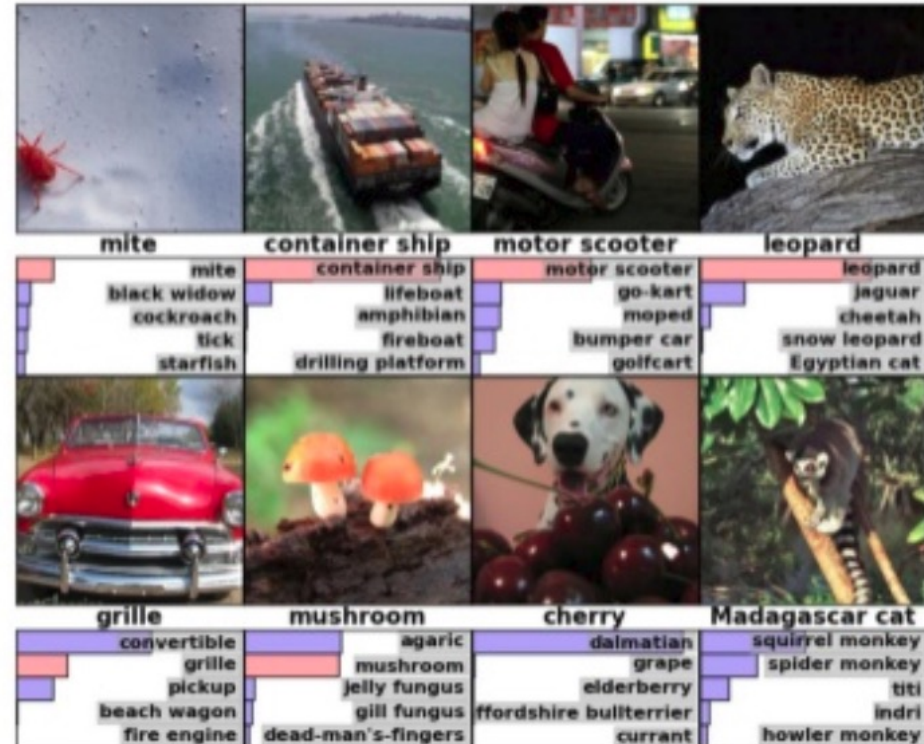
Deep Networks Enter 2012

- 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

IMAGENET
Accuracy Rate

• Traditional CV • Deep Learning
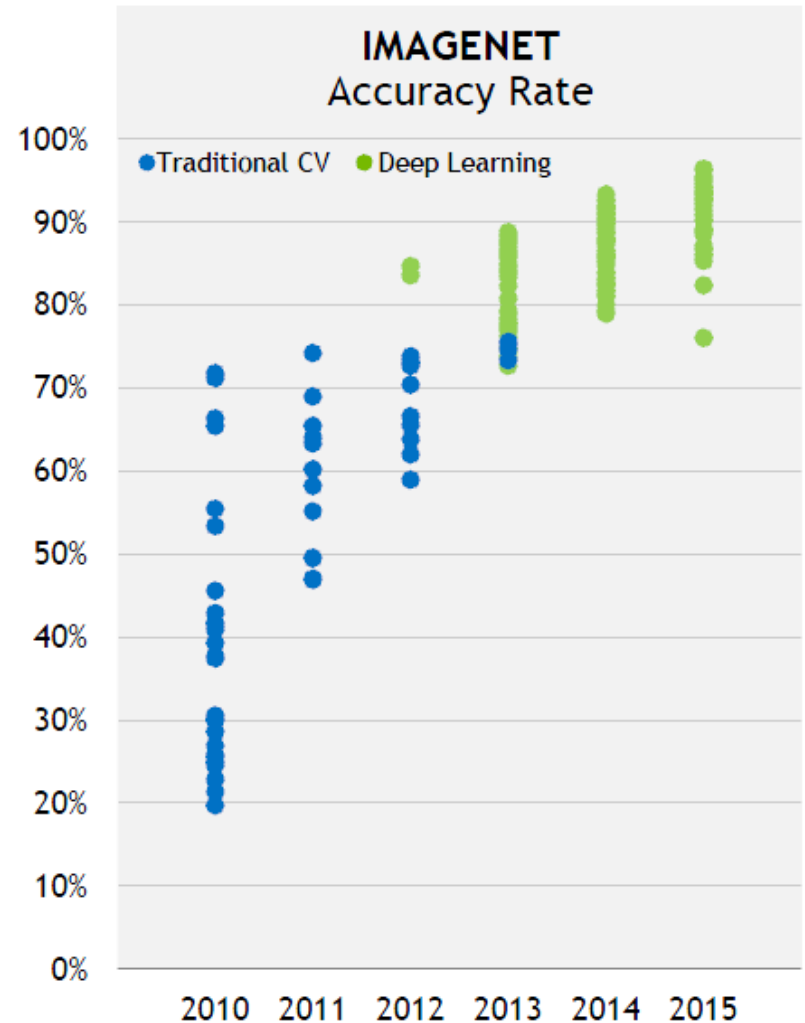
Alex Net

- 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 layer is 1000-way softmax
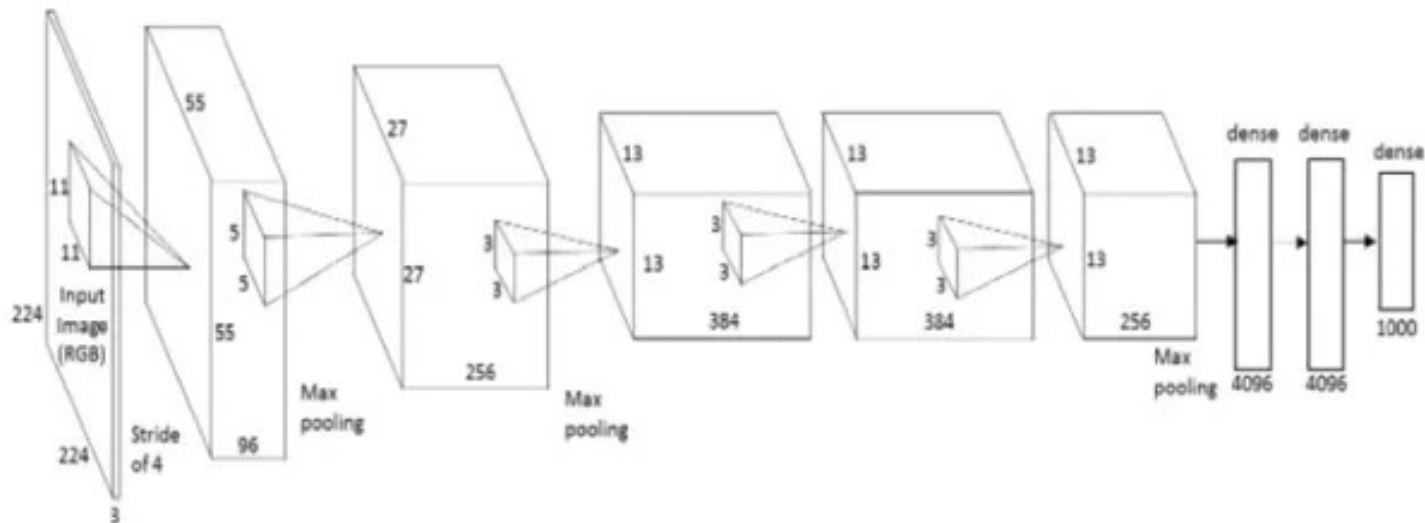
Why using many layers?
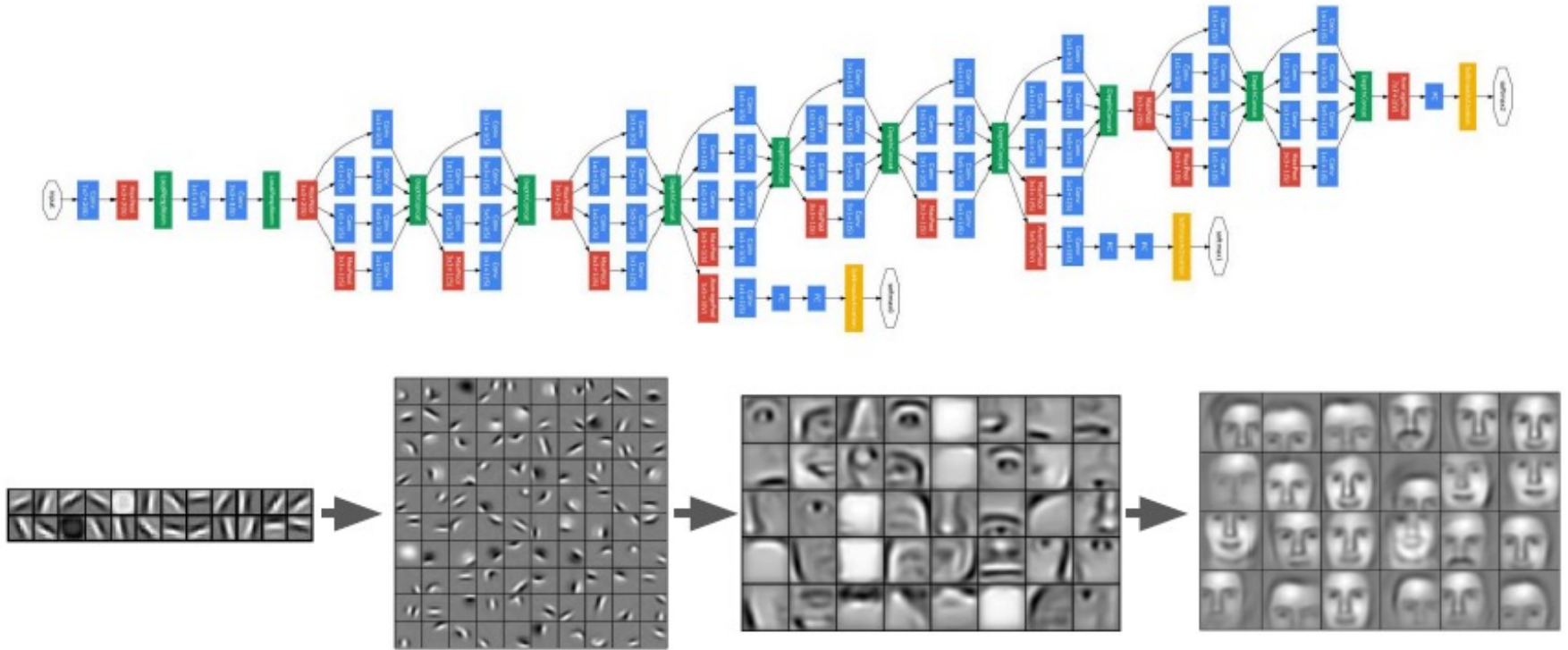
From: Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations, Honglak Lee et al.

Biological Inspiration

- Processing in the brain uses multi-layer processing
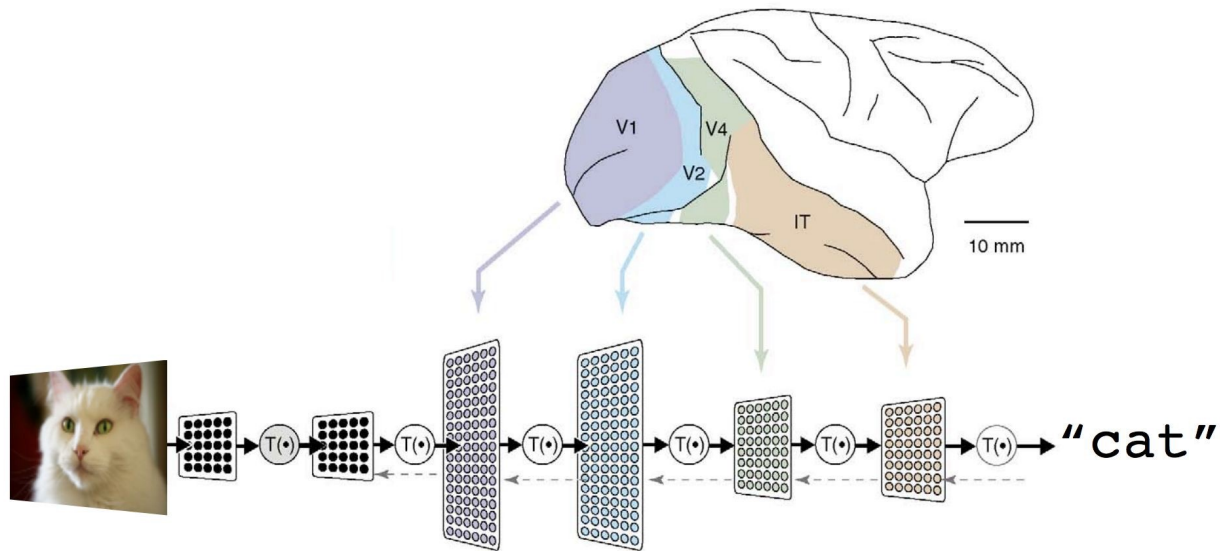
Outline

- 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
  - Recent history
  - Review of model training pipeline
  - Stochastic gradient descent: general concept
  - Data Preprocessing and Regularization
- Training, validation and testing and cross validation

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

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_{Oi}	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	$\sum_{i,k} (y_{ik} - \hat{y}_{i,k})^2$
Binary classification	$y_i = \{0,1\}$	$\hat{y}_i$ = 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	$-\sum_i \sum_{k=1}^{K} y_{ik} \ln \hat{y}_{i,k}$

Training with Gradient Descent

- Neural network training: Minimize loss function

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

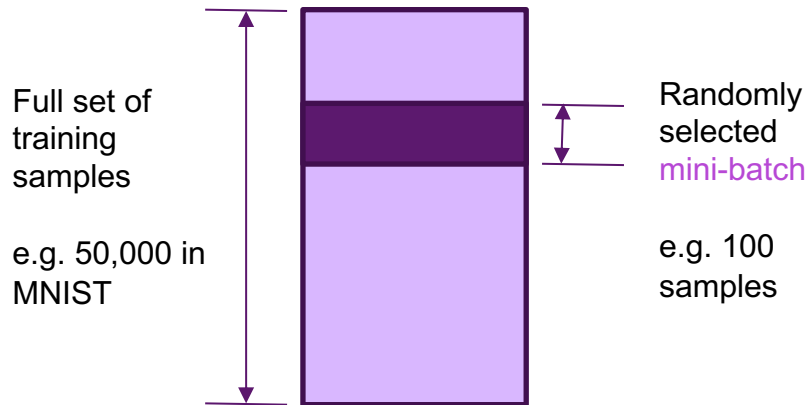 - $L_i(\theta, \boldsymbol{x}_i, y_i)$ = loss on sample i for parameter θ

- Standard gradient descent:

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

  - Each iteration requires computing N loss functions and gradients
  - But, gradient computation is expensive when data size N large

Stochastic Gradient Descent

Full set of
training
samples

e.g. 50,000 in
MNIST

Randomly
selected
mini-batch

e.g. 100
samples

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

- 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:
  $$\theta^{t+1} = \theta^t - \alpha^t g^t$$

Learning rate

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 $\alpha^t \to 0$ and $\sum_t \alpha^t = \infty$. Let $s_t = \sum_{k=0}^{t} \alpha^k$

  - Then $\theta^t \to \theta(s_t)$ 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
  - 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 use the minibatch gradient?
  - 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: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_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 θ_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 θ_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

Learning Rate Scheduling

- Constant learning rate typically not optimum

- The learning rate should be gradually reduced as the loss reduces

- Typically start with a relatively large initial learning rate (e.g. 10^-3), reduce by a factor (say 0.9) after every T epochs

- Initial learning rate and decay factor may be determined by trial and error (through validation data)

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

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
  - Norm constraint: $\|w_H\|_2 < c$
  - Should only turn on this regularization after a few epochs

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 after each conv/fully connected layer before nonlinear activation!
- 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: γ, β

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.

Sergey Ioffe, Christian Szegedy: **Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.**
https://arxiv.org/pdf/1502.03167v3.pdf
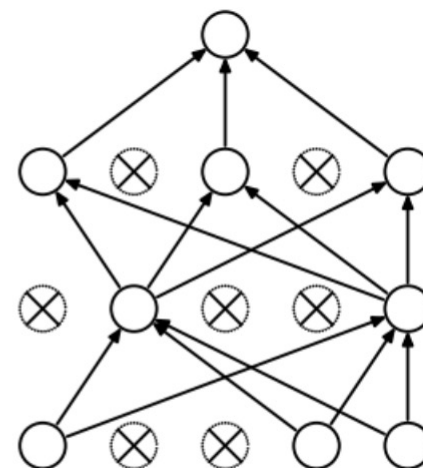
https://www.youtube.com/watch?v=nUUqwaxLnWs
https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c

Regularization: Dropout

- Drop some percentage (Dropout Rate, e.g. p=50%) 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, but scaling the output of each node by 1/p

- 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
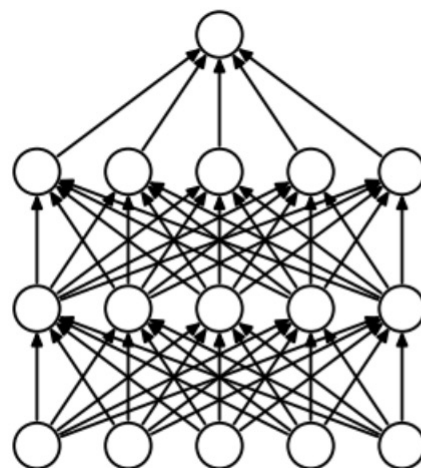
(a) Standard Neural Net   (b) After applying dropout.

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

Should only be applied to fully connected layers!

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
- Can provide significant performance boost when you have small training data

```
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
```

Useful Augmentations
(Beyond Geometric Manipulations)
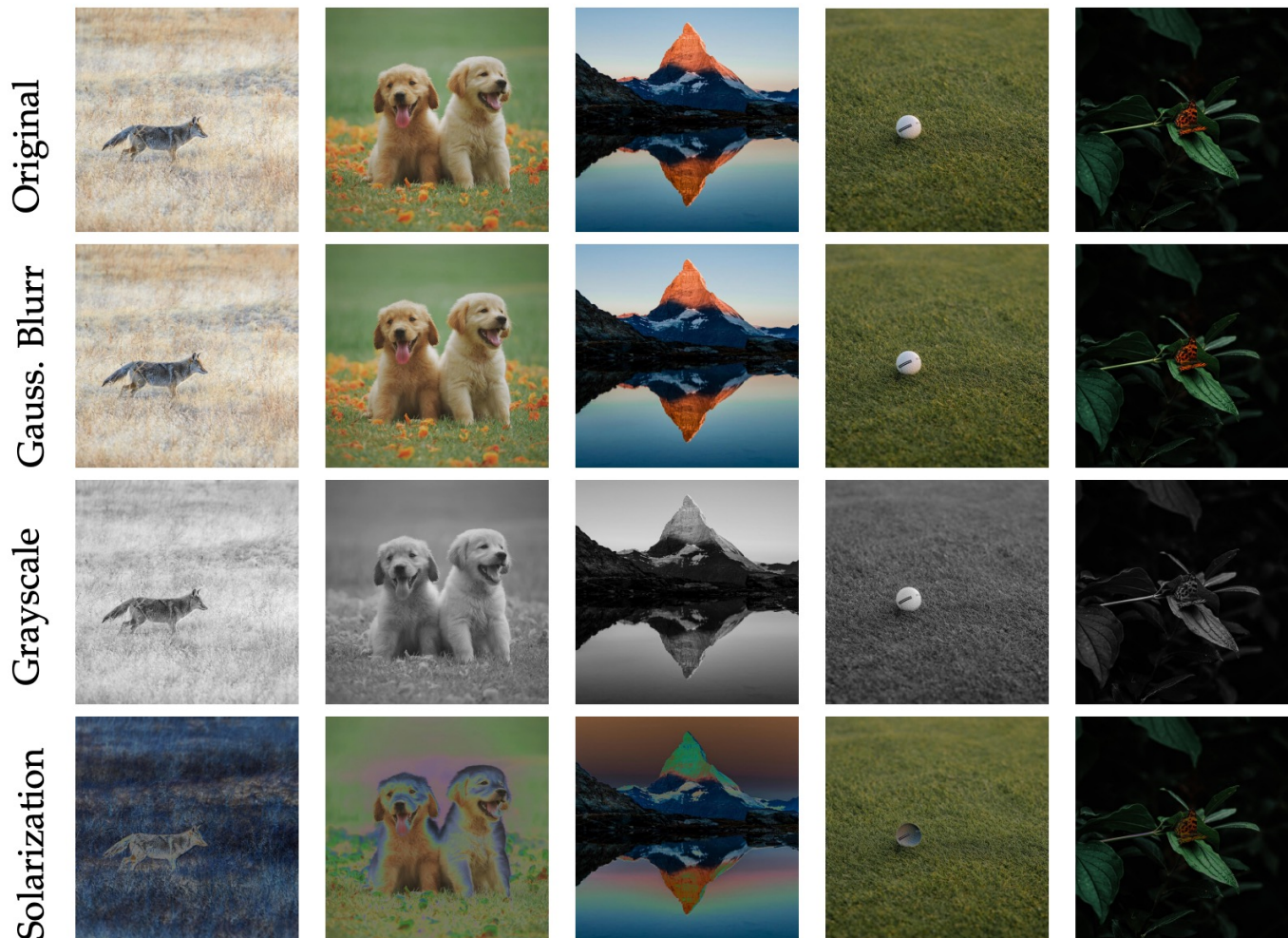
3-augment + random cropping: Ref. DeiT-III, https://arxiv.org/pdf/2204.07118.pdf

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

Pop Quizzes

- How is SGD different from standard gradient descent?

- Why is SGD helpful?

- How does ADAM optimizer differ from SGD?

- How should you set and adjust the learning rate?

- What is batch normalization and why?

- What is drop-out and why?

- What are some tricks to help the training?

Pop Quizzes (w/ Answers)

- How is SGD different from standard gradient descent?
  - Gradient descent: computing the loss and gradient for all training samples and then update the model
  - SGD: computing the loss and gradient for one minibatch of samples at a time and update the model based on the batch gradient, iterate among batches and then repeat after random shuffling of the total training set.

- Why is SGD helpful?
  - Reduce the memory cost, and allow multiple updates being done within each run of the entire training set, critical for large dataset

- How does ADAM optimizer differ from SGD?
  - Gradient descent: update using the gradient of the current batch
  - Adam: update using the "moment" = moving average of the batch gradient, which is less noisy, loss function less oscillating.

- How should you set and adjust the learning rate?
  - Relatively large learning rate in the beginning and gradually reduce, to help the network converge

Pop Quizzes (w/ Answers) (Cnt'd)

- What is batch normalization and why?
  - Normalize the input to the next layer to be zero mean and unit variance
  - Help to reduce the arbitrary scaling of the filter coefficients/weights
- What is drop-out and why?
  - Randomly drop out some nodes to make the training more robust
  - Only use in fully connected layers
- What are some tricks to help the training?
  - Data augmentation
  - L2 or L1 penalty on the weights of fully connected layers
  - Normalized filter coefficients
  - Batch norm and drop out
  - Using a good optimizer, e.g. Adam
 - Learning rate scheduling

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

Training, Validation, and Testing

- Goal: use training data to learn a model that works well on unseen data!
- Randomly split the data set to training, validation, and testing subsets
    - Each set 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 validation set) in successive epochs and plot loss curves
- Save the model at the epoch where the validation loss is minimal
- Apply the model to the testing subset
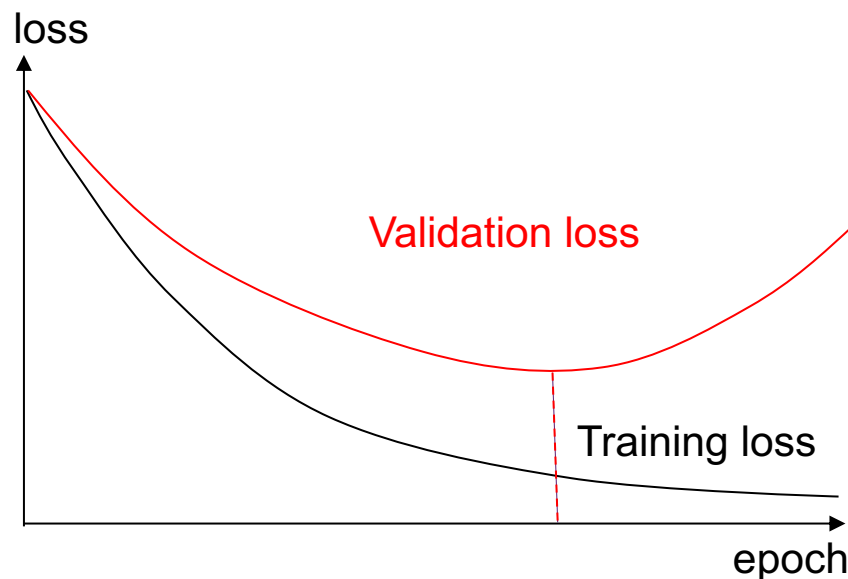
Checking the loss curves!
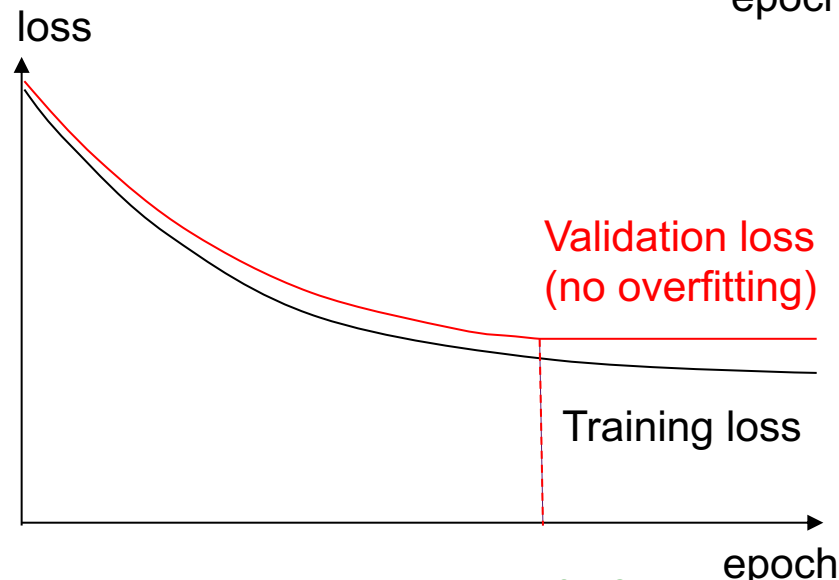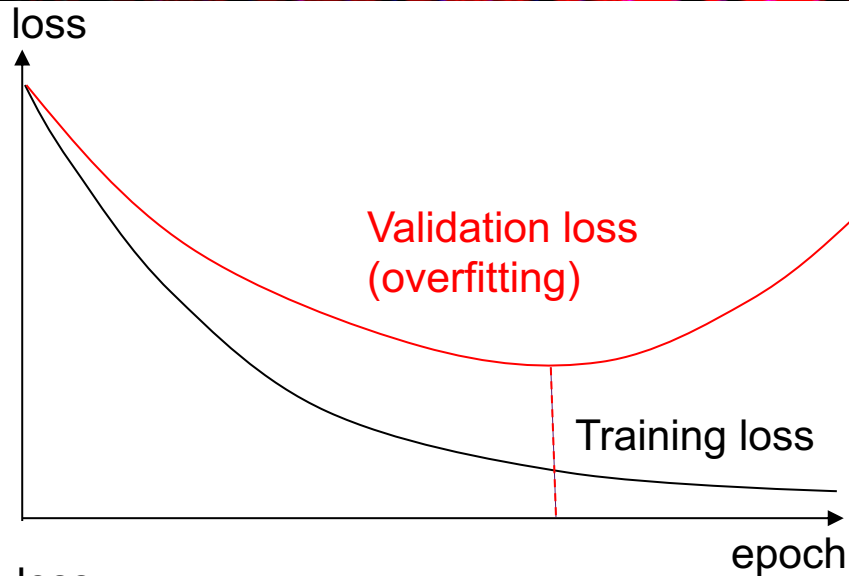
- The training loss should decrease (accuracy increase) in successive epochs
- But the validation loss may not!
- **Stop when validation loss starts to increase**
- If validation loss is significantly higher than training loss:
  - Use more regularization (stronger weight penalty, Batch norm, drop out)
  - Use more data augmentation
  - May need to increase data or reduce network complexity!

Model Architecture / Hyperparameter Tuning

- To compare among multiple model structures (e.g. #layers, # filters, filter sizes) and hyperparameters of the same structure (e.g. initial learning rate, learning rate decay schedule, regularization strength)

- Split data to training/validation/testing
  - For each candidate model structure / hyperparameter
    - 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 model structure / hyperparameter setting
    - Using (K-1) fold for training, and 1 fold for validation;
    - Repeat K times
    - Average performance for all validation folds
  - Determine the best structure / hyperparameter 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
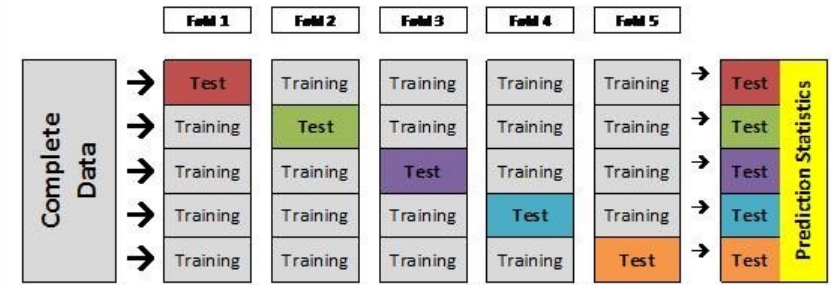
Underfit vs. Overfit

- When the training error at convergence is still large, the network is underfit
  - The chosen architecture does not have enough representation power (i.e. network does not have enough capacity).
  - Need to modify network architecture to be more complex (more layers or more feature maps or change the 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.
  - Regularization to reduce overfit (weight regularization, batch norm, drop out)
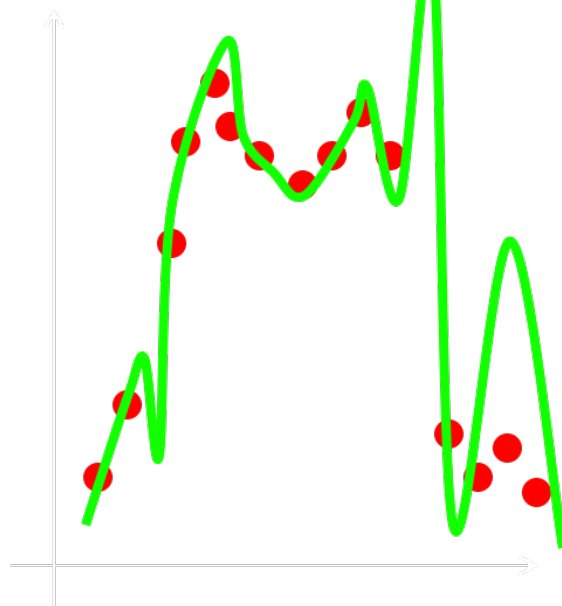 - More data augmentation, get more data
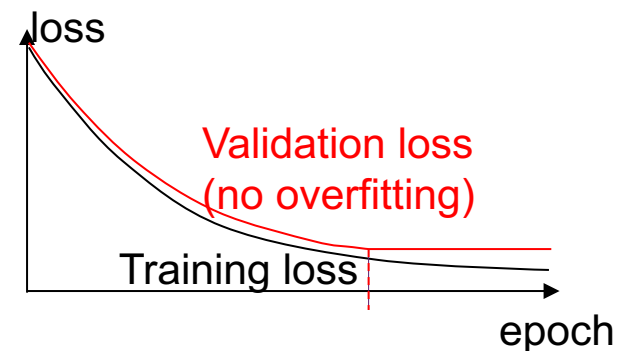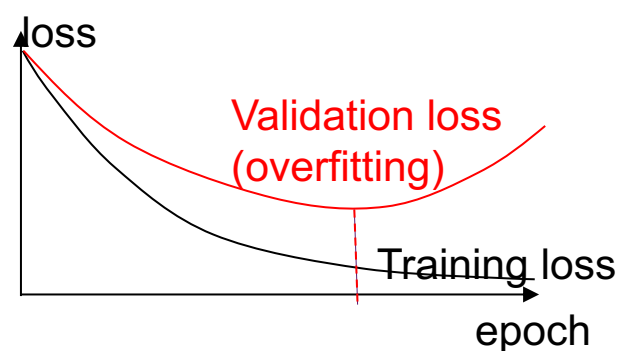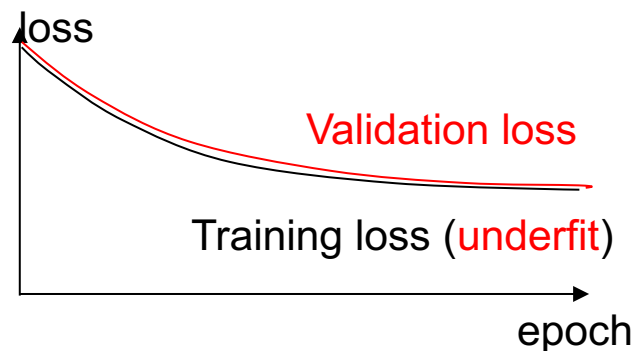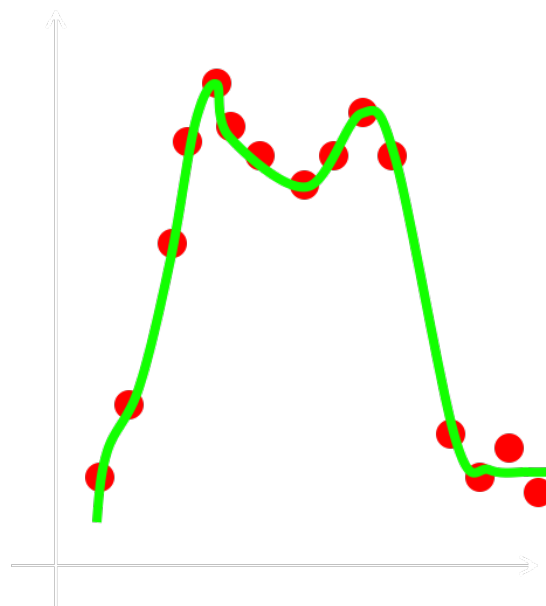
Model Structure Selection
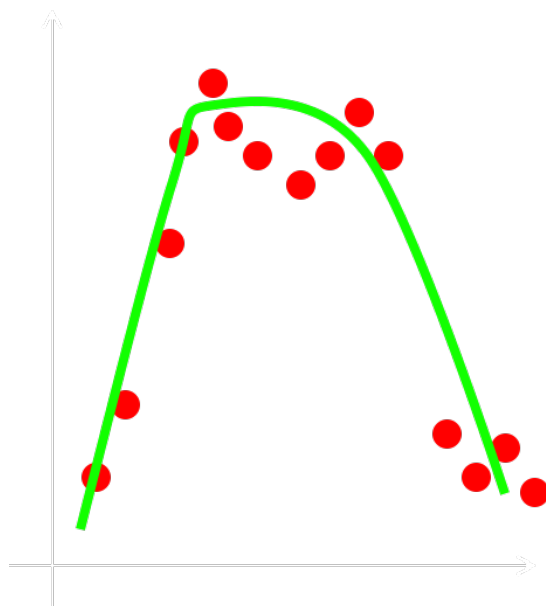
Small model   Big model   Big model + Regularize

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

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

Pop Quizzes

- How do you make your learnt model generalizable to unseen data?
- How do you tell if your model is underfit or overfit?
- What to do if it is underfit?
- What to do if it is overfit?

Pop Quizzes (w/ answers)

- How do you make your learnt model generalizable to unseen data?
    - Training/validation/testing split
    - Cross validation (necessary when the dataset is small)
    - Data augmentation (introduce more variance into the data, increase training samples)
- How do you tell if your model is underfit or overfit?
    - Underfit: training loss is still large
    - Overfit: training loss is small, validation loss is large
- What to do if it is underfit?
    - Increase model complexity
- What to do if it is overfit?
    - Introduce more data augmentation, regularization
   - Reduce network complexity or modify network architecture

Summary: Building a Conv Net

- Define a network structure
  - Conv layer + fully connected layers
  - Add batch normalization and drop out (in fully connected layers only)
- Set up a loss function based on the given task
  - Need to add proper regularization on weights
- Partition data to training, validation and testing
  - Preprocess data (zero-mean, unit variance)
  - Augment training data
- Perform stochastic gradient descent on training set
  - Calculate gradient for each minibatch
  - Update the model parameters (ADAM optimizer preferred)
  - Evaluate the loss for training and validation 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
- Fortunately, you don't have to write the code from scratch!
  - Many deep learning frameworks allow us to build/train/test a model relatively easily

Deep Learning Frameworks

Deep-Learning Package Zoo

- Torch
- Caffe
- Theano (Keras, Lasagne)
- CuDNN
- Tensorflow
- Mxnet
- Etc.
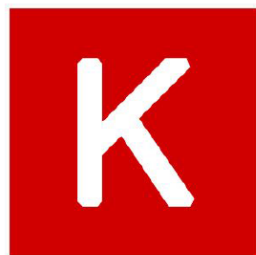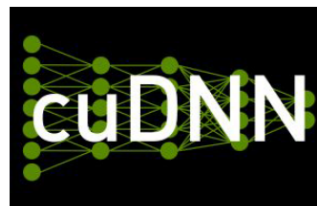
- Enable quick development of models and test ideas!
- Automatically compute gradients!

PyTorch

- We will use PyTorch for this class

- Many built-in utilities for
  - Defining the network
  - Defining the loss function
  - Perform SGD

- See PyTorch tutorial
  - https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
  - https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#sphx-glr-beginner-blitz-cifar10-tutorial-py

- Attend tutorials by TAs

Summary

- Supervised learning: General concepts
  - Loss functions
- Neural network and training
  - From single perceptron to multi-layer perceptrons
  - Gradient descent for model training, Back propagation
- Convolutional network
  - Why using convolution
  - Multichannel convolution
  - Pooling
  - Receptive field
- Deep networks
  - Large dataset, deep models
  - Stochastic gradient descent: general concept
  - Data Preprocessing and Regularization
  - Data augmentation
- Training, validation and testing and cross validation

What should you know
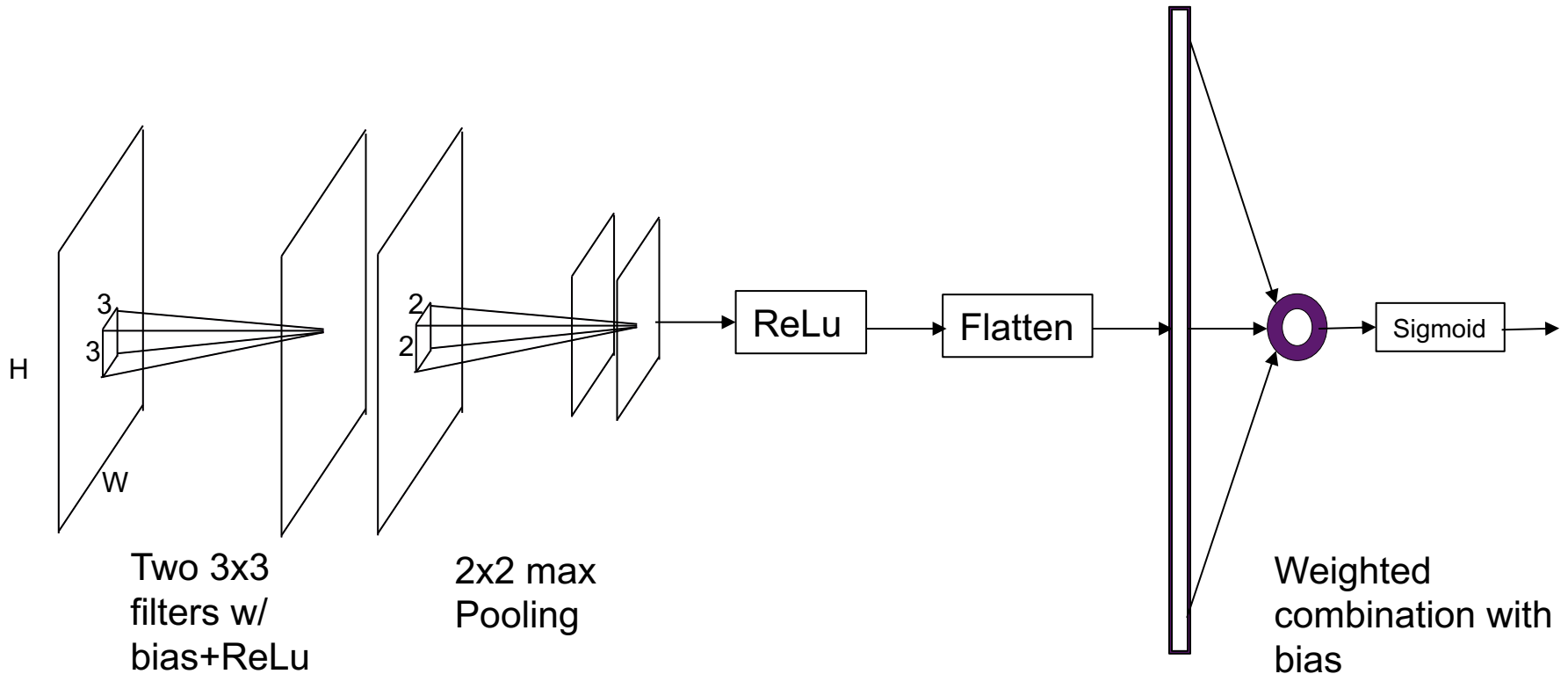
- Be able to answer all the pop quizzes!

Recommended Readings

- Material for the machine learning class developed by Sundeep Rangan:

  - https://github.com/sdrangan/introml/blob/master/sequence.md

- Material from Standard Univ. class CS231n Convolutional Neural Networks for Visual Recognition

  - lecture note https://cs231n.github.io/

- https://pytorch.org/tutorials/

Written Assignment (1)

1. For the simple network structure shown below (1 conv layer followed by downsampling and 1 fully connected layer)

   1) Determine the number of trainable parameters. Assume the input image is gray scale (one channel, with height=H, width=W).

   2) What is the receptive field (relative to the input image) of the output at layer 2 (before ReLu Box)?

3
3

H

W

Two 3x3 filters w/ bias+ReLu

2
2

2x2 max Pooling

ReLu

Flatten

Sigmoid

Weighted combination with bias

Written Assignment (2)

2. Consider the following simple fully convolutional network architecture used for image denoising

1) Determine the number of trainable parameters. Write down clearly the number of parameters in each layer, so that you can get partial credits.

2) What is the receptive field sizes (relative to the input image) of the filters in each layer?

3) A simple loss function for denoising is the mean squares error between the ground-truth (noise-free) image and the denoised image. Define the total loss function for all samples in a batch. Use $\hat{y}_{i,k}(m,n)$ to represent the k-th color component of the i-th denoised image, and $y_{i,k}(m,n)$ the corresponding noise-free image.

4) Let the j-th feature map for the i-th input image after Layer 2 be described by $z_{i,j}(m,n)$. The filters in Layer 3 be denoted by $h_{j,k}(m,n)$ and the biases by b_k, where j is the index of the input feature map, and k is the index of the output color channels. Express the output $\hat{y}_{i,k}(m,n)$ as a function of $z_{i,j}(m,n), h_{j,k}(m,n), b_k$.

5) Define the gradient of the loss function with respect to the parameters $h_{j,k}(m,n), b_k$.

Noisy image 3 color channels · 48 feature channels · 48 feature channels · Denoised image 3 color channels

Layer 1
5x5 filters w/ bias
+ ReLU

Layer 2
3x3 filters w/ bias
+ ReLU

Layer 3
3x3 filters w/ bias