CS559: Neural Network & Convolutional NN (CNN)

Lecture 10

Outline

- Review
- Neural Network
- Neural Network & Network Training
- Convolutional Neural Networks (CNN)
- Understanding Convolutional Networks

Review - Summary of Supervised Learning

$$y = \boldsymbol{w}^t \boldsymbol{x} + b$$

	Classification	Regression
Predictor	Sign	Score
Related to y	Margin (score, y)	Residual (score, y)
Loss Function	Zero-one, Hinge, Logistic	Squared, absolute deviation
Algorithm	Gradient Descent	Gradient Descent

$$\min_{w \in \mathbb{R}^d} TrainLoss(\boldsymbol{w}) = \frac{1}{N} \sum_{i=1}^{N} L(x_i, y_i, \boldsymbol{w})$$

Review - Optimization Problem

Gradient Descent (GD)

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla_{\boldsymbol{w}} TrainLoss(\boldsymbol{w})$$

$\nabla_w TrainLoss$ denotes the gradient of the (average) total training loss with respect to ***w.***

Stochastic Gradient Descent (SGD)

For each $(x, y) \in D_{train}$

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla_{\boldsymbol{w}} L(x, y, \boldsymbol{w})$$

$\nabla_w L$ denotes the gradient of one example loss with respect to ***w.***

Neural Network—What is it?

- Often associated with biological devices (brains), electronic devices, or network diagrams
- But the best conceptualization for this presentation is none of these: Think of neural network as a mathematical function.

Pros:

- Successfully used on a variety of domains: computer vision, speech recognition, gaming etc.
- Can provide solutions to very complex and nonlinear problems
- If provided with sufficient amount of data, can solve classification and forecasting problems accurately and easily
- Once trained, prediction is fast

Neural Network - Motivation

- Perceptron limitations
  - An algorithm for binary classifiers
    - $y = \begin{cases} 1 & if\ \boldsymbol{w}^T\boldsymbol{x} + b > 0 \\ 0 & Otherwise \end{cases}$
  - Does not terminate if the learning set is not linearly separable

- Feedforward networks and Backpropagation
  - A perceptron is an artificial neuron using the Heaviside step function as the activation function
  - Also termed the *single-layer* perceptron

- Allow to learn non linearly separable transformations from input to output;
- A single hidden layer allows to compute any input/output transformation;

Neural Network – Motivation Example

Predicting Car Collision

Input: position of two oncoming cars $\boldsymbol{x} = [x_1, x_2]^T$

Output: whether safe $(y = +1)$ or collide $(y = -1)$

- True function: safe if cars are far (at least 1): $y = sign(|x_1 - x_2| - 1)$
  - $\boldsymbol{x} = [1,3]^T, y = +1$
  - $\boldsymbol{x} = [3,1]^T, y = +1$
  - $\boldsymbol{x} = [1,0.5]^T, y = -1$

Neural Network – Motivation Example

Decomposition:

- Test if car 1 is far right of car 2: $h_1 = (x_1 - x_2 \geq 1)$
- Test if car 2 is far right of car 1: $h_2 = (x_2 - x_1 \geq 1)$
- Safe if at least one is true:

$$y = sign(h_1 + h_2)$$

$\boldsymbol{x}$	h_1	h_2	y
$[1,3]^T$	0	1	+1
$[3,1]^T$	1	0	+1
$[1,0.5]^T$	0	0	-1

Neural Network – Learning Strategy

- Define: $x = [1, x_1, x_2]^T$
- Intermediate hidden subproblems:

$$h_1 = (\boldsymbol{v}_1^T \boldsymbol{x} \geq 0), \boldsymbol{v_1} = [-1, +1, -1]^T$$
$$h_2 = (\boldsymbol{v}_2^T \boldsymbol{x} \geq 0), \boldsymbol{v_2} = [+1, -1, -1]^T$$

- Final prediction:

$$f_{\boldsymbol{V},\boldsymbol{w}}(\boldsymbol{x}) = sign(w_1 h_1 + w_2 h_2), \boldsymbol{w} = [w_1, w_2] = [1,1]$$

- Goal: Learn both hidden subproblems and combination weights
- Key Idea: joint learning

Neural Network – Learning Strategy

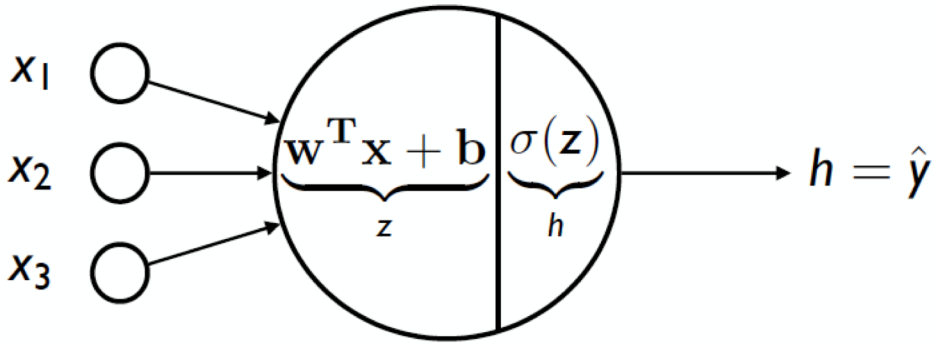
- Problem: gradient of h_1 w.r.t. $\boldsymbol{v_1}$ is 0: $h_1 = (\boldsymbol{v}_1^T \boldsymbol{x} \geq 0)$
- The logistic function maps $(-\infty, \infty)$ to (0,1):

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

- Derivative: $\sigma'(z) = \sigma(z)\big(1 - \sigma(z)\big)$
- Solution: $h_j = \sigma(\boldsymbol{v}_j^T \boldsymbol{x})$

Neural Network – No Hidden Units: Logistic Regression

Sigmoid activation function:



Output score: sigmoid function $\hat{y} = \sigma(\boldsymbol{w}^T \boldsymbol{x} + b) = \frac{1}{1+e^{-\boldsymbol{w}^T \boldsymbol{x}}}$

Logistic loss: $J(x, y, \boldsymbol{w}) = -y \log(\hat{y}) - (1-y) \log(1-\hat{y})$

Derivatives:

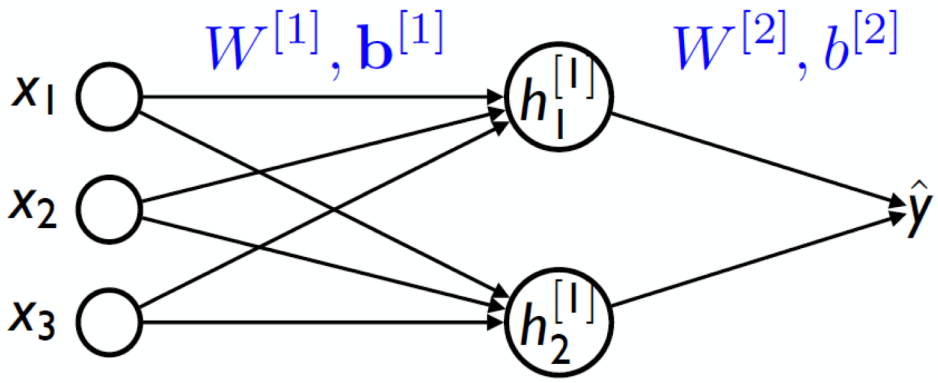
- $\frac{\partial \hat{y}}{\partial w_i} = \hat{y}(1-\hat{y})x_i$ & $\frac{\partial J}{\partial w_i} = \frac{\partial J}{\partial \hat{y}}\left(\frac{\partial \hat{y}}{\partial w_i}\right) = (\hat{y}-y)x_i$

Apply gradient descent (η is the learning rate):

- Element operation: $w_i^{t+1} = w_i^t - \eta\left(\frac{\partial J}{\partial w_i}\right)$
- Vector operation: $\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \eta\left(\frac{\partial J}{\partial \boldsymbol{w}}\right)$

Neural Network – One Hidden Layer $\mathbf{h}^{[l]}$

One hidden layer:



Hidden layer representation:

$$z_1^{[1]} = \boldsymbol{w}_1^{[1]^T}\boldsymbol{x} + b_1^{[1]}, h_1^{[1]} = \sigma\left(z_1^{[1]}\right)$$

$$z_2^{[1]} = \boldsymbol{w}_2^{[1]^T}\boldsymbol{x} + b_2^{[1]}, h_2^{[1]} = \sigma\left(z_2^{[1]}\right)$$

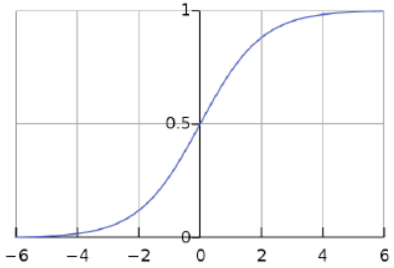
$$\boldsymbol{z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}$$
$$\boldsymbol{h}^{[1]} = \sigma\left(\boldsymbol{z}^{[1]}\right)$$

Output Layer:

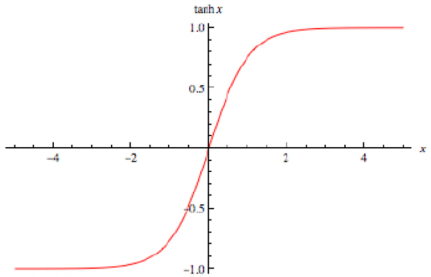
$$z^{[2]} = W^{[2]}\boldsymbol{h}^{[1]} + b^{[2]}$$
$$\hat{y} = \sigma\left(z^{[2]}\right)$$

Neural Network – Activation Functions

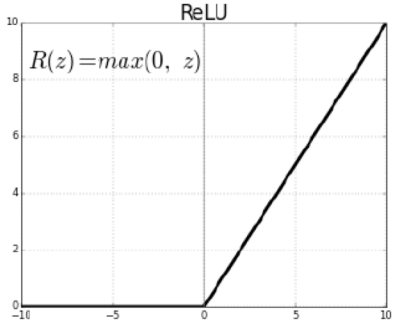
Sigmoid: $f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$



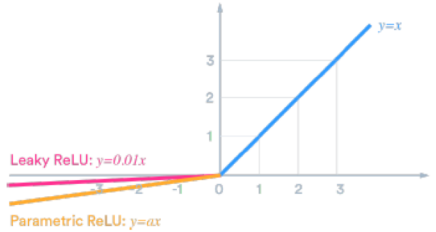
tanh: $f(x) = 2\sigma(2x) - 1$



ReLU: $f(x) = \max(0, x)$



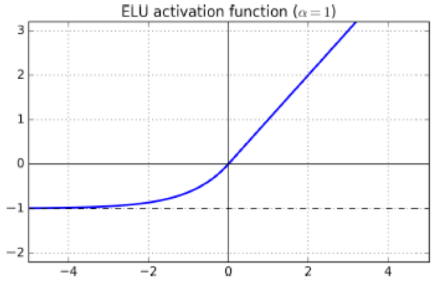
Leaky ReLU: $f(x) = \max(\alpha x, x)$



Maxout

$$\max(\mathbf{w}_1^T \mathbf{x} + b_1, \mathbf{w}_2^T \mathbf{x} + b_2)$$

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



Neural Network – One Hidden Layer

Think of intermediate hidden units as learned features of a linear predictor.
Feature learning: manually specified features: x
automatically learned features: $h(\boldsymbol{x}) = [h_1(x), \cdots, h_k(x)]$

Neural Network – Loss Minimization

Optimization problem:

$$Trainloss(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{|D_{train}|} \sum_{x,y \in D_{train}} Loss(x, y, \boldsymbol{W}, \boldsymbol{b})$$

$$J = Loss(x, y, W, b) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

$$\hat{y} = h^{[2]}$$

Goal: compute gradient

$$\nabla_{\boldsymbol{W},\boldsymbol{b}} J(\boldsymbol{W}, \boldsymbol{b})$$

Stochastic Gradient Descent:

Repeat, given a sample $\left(x^{(i)}, y^{(i)}\right)$:

$$dW^{[1]} = \frac{\partial J}{\partial W^{[1]}}, W^{[1]} = W^{[1]} - \eta dW^{[1]}$$

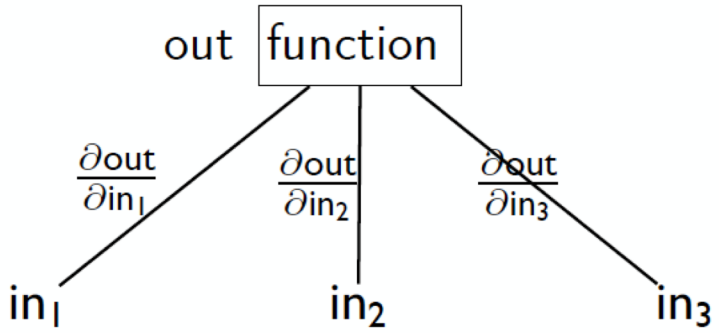
$$d\boldsymbol{b}^{[1]} = \frac{\partial J}{\partial \boldsymbol{b}^{[1]}}, \boldsymbol{b}^{[1]} = \boldsymbol{b}^{[1]} - \eta d\boldsymbol{b}^{[1]}$$

$$dW^{[2]} = \frac{\partial J}{\partial W^{[2]}}, W^{[2]} = W^{[2]} - \eta dW^{[2]}$$

$$db^{[1]} = \frac{\partial J}{\partial b^{[2]}}, b^{[2]} = b^{[2]} - \eta db^{[2]}$$

Neural Network – Approach

- Mathematical: just grind through the chain rule
- Next: visualize the computation using a computation graph
- Advantages:
  - Avoid long equations
  - Reveal structure of computations (modularity, efficiency, dependencies)



Neural Network – Back Propagation

- Goal: to learn the weights so that the loss (error) is minimized.
- It provides an efficient procedure to compute derivatives.
- For a fixed sample (**x**,y), we want to learn $\hat{y} = f(\boldsymbol{x})$
- Negative Log-likelihood over input $\boldsymbol{x}_n$ is $L_n = -y_n \log(\hat{y}_n) - (1 - y_n)\log(1 - \hat{y}_n)$
- Total error of training examples is $E = \sum_n L_n$

$$\underbrace{W^{[1]}x + b^{[1]}}_{\boldsymbol{z}^{[1]}} \underbrace{\sigma\left(z^{[1]}\right)}_{\boldsymbol{h}^{[1]}} \Longrightarrow \underbrace{W^{[2]}\boldsymbol{h}^{[1]} + b^{[2]}}_{\boldsymbol{z}^{[2]}} \underbrace{\sigma\left(z^{[2]}\right)}_{\boldsymbol{h}^{[2]} = \hat{y}} \Longrightarrow L(\hat{y}, y)$$

$$d\boldsymbol{z}^{[1]} = \frac{\partial L}{\partial z^{[2]}}\left(\frac{\partial z^{[2]}}{\partial \boldsymbol{z}^{[1]}}\right) = {W^{[2]}}^T dz^{[2]} \circ \boldsymbol{h}^{[1]} \circ \left(1 - \boldsymbol{h}^{[1]}\right)$$

$$dW^{[1]} = d\boldsymbol{z}^{[1]}\boldsymbol{x}^T$$

$$db^{[1]} = dz^{[1]}$$

$$dz^{[2]} = \frac{\partial L}{\partial \hat{y}}\left(\frac{\partial \hat{y}}{\partial z^{[2]}}\right) = \hat{y} - y$$

$$dW^{[2]} = dz^{[2]}{\boldsymbol{h}^{[1]}}^T$$

$$db^{[2]} = dz^{[2]}$$

Neural Network – Back Propagation

Forward Propagation

$$\boldsymbol{z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}$$
$$\boldsymbol{h}^{[1]} = \sigma\left(\boldsymbol{z}^{[1]}\right)$$
$$z^{[2]} = W^{[2]}\boldsymbol{h}^{[1]} + b^{[2]}$$
$$\hat{y} = \sigma\left(z^{[2]}\right)$$

Backpropagation

$$dz^{[2]} = \frac{\partial L}{\partial \hat{y}}\left(\frac{\partial \hat{y}}{\partial z^{[2]}}\right) = \hat{y} - y$$
$$dW^{[2]} = dz^{[2]}{\boldsymbol{h}^{[1]}}^{T}$$
$$db^{[2]} = dz^{[2]}$$
$$d\boldsymbol{z}^{[1]} = \frac{\partial L}{\partial z^{[2]}}\left(\frac{\partial z^{[2]}}{\partial \boldsymbol{z}^{[1]}}\right) = {W^{[2]}}^{T} dz^{[2]} \circ \boldsymbol{h}^{[1]} \circ \left(1 - \boldsymbol{h}^{[1]}\right)$$
$$dW^{[1]} = d\boldsymbol{z}^{[1]}\boldsymbol{x}^{T}$$
$$db^{[1]} = dz^{[1]}$$

Neural Network – Vectorizing across multiple examples

Forward Propagation

For $n = 1$ to N:

$$z_n^{[1]} = W^{[1]} x_n + b^{[1]}$$
$$\boldsymbol{h}_n^{[1]} = \sigma\left(\boldsymbol{z}_n^{[1]}\right)$$
$$z^{[2]} = W^{[2]} \boldsymbol{h}_n^{[1]} + b^{[2]}$$
$$h_n^{[2]} = \sigma\left(z_n^{[2]}\right)$$

$$Z^{[1]} = W^{[1]} X + \boldsymbol{b}^{[1]}$$
$$\boldsymbol{H}^{[1]} = \sigma\left(Z^{[1]}\right)$$
$$Z^{[2]} = W^{[2]} H^{[1]} + b^{[2]}$$
$$H^{[2]} = \sigma\left(Z^{[2]}\right)$$

Backpropagation

Given one data point

$$dz^{[2]} = h^{[2]} - y$$
$$dW^{[2]} = dz^{[2]} \boldsymbol{h}^{[1]^T}$$
$$db^{[2]} = dz^{[2]}$$
$$d\boldsymbol{z}^{[1]} = W^{[2]^T} dz^{[2]} \circ \boldsymbol{h}^{[1]} \circ \left(1 - \boldsymbol{h}^{[1]}\right)$$
$$dW^{[1]} = d\boldsymbol{z}^{[1]} \boldsymbol{x}^T$$
$$db^{[1]} = dz^{[1]}$$

$$dZ^{[2]} = H^{[2]} - \boldsymbol{y}$$
$$dW^{[2]} = \frac{1}{N} dZ^{[2]} H^{[1]^T}$$
$$db^{[2]} = \frac{1}{N} \sum dZ^{[2]}$$
$$dZ^{[1]} = W^{[2]^T} dZ^{[2]} \circ H^{[1]} \circ \left(1 - H^{[1]}\right)$$
$$dW^{[1]} = \frac{1}{N} dZ^{[1]} X^T$$
$$d\boldsymbol{b}^{[1]} = \frac{1}{N} \sum d\boldsymbol{z}^{[1]}$$

Neural Network – Adaptive learning rates

Popular and simple idea: reduce the learning rate by some factor every few epochs.

- At the beginning, we are far from the destination, so we use larger learning rate
- After several epochs, we are close to the destination, so we reduce the learning rate
- E.g. 1/t decay: $\eta^t = \eta/\sqrt{t+1}$

Learning rate cannot be one-size-fits-all

- Giving different parameters different learning rates.

Divide the learning rate of each parameter by the RMS of its previous derivatives, σ^t.

$$\eta^t = \frac{\eta}{\sqrt{t+1}}, g^t = \frac{\partial L(w^t)}{\partial w}$$

- SGD: $w^{t+1} \leftarrow w^t - \eta^t g^t$
- Adagrad: $w^{(t+1)} \leftarrow w^{(t)} - \frac{\eta^{(t)}}{\sigma^{(t)}} g^{(t)}$

Neural Network – Adagrad

$$w^{(1)} \leftarrow w^{(0)} - \frac{\eta^{(0)}}{\sigma^{(0)}} g^{(0)} \qquad \sigma^{(0)} = \sqrt{(g^{(0)})^2 + \epsilon}$$

$$w^{(2)} \leftarrow w^{(1)} - \frac{\eta^{(1)}}{\sigma^{(1)}} g^{(1)} \qquad \sigma^{(1)} = \sqrt{\frac{1}{2}[(g^{(0)})^2 + (g^{(1)})^2] + \epsilon}$$

$$\vdots \qquad\qquad \vdots$$

$$w^{(t+1)} \leftarrow w^{(t)} - \frac{\eta^{(t)}}{\sigma^{(t)}} g^{(t)} \qquad \sigma^{(t)} = \sqrt{\frac{1}{t+1}\sum_{i=1}^{t}(g^{(i)})^2 + \epsilon}$$

ϵ is a smoothing term that avoids division by zero (usually on the order of 10^{-8})

$$w^{(t+1)} \leftarrow w^{(t)} - \left(\frac{\eta}{\sqrt{\sum_{i=1}^{t}(g^{(i)})^2}}\right) g^{(t)}$$

where $\eta^{(t)} = \frac{\eta}{\sqrt{t+1}}$

Larger gradient, larger step

Larger gradient, smaller step

Neural Network – Adagrad

g^0	g^1	g^2	g^3	g^4	...
0.001	0.001	0.003	0.002	0.1	...
g^0	g^1	g^2	g^3	g^4	...
10.8	20.9	31.7	12.1	0.1	...

Neural Networks – Multilayer Perception

A multilayer perceptron represents an adaptable model $y(\cdot, w)$ able to map D-dimensional input to C-dimensional output:

$$y(\cdot, w): \mathbb{R}^D \longrightarrow \mathbb{R}^C, x \longmapsto y(x, w) = \begin{pmatrix} y_1(x, w) \\ \vdots \\ y_C(x, w) \end{pmatrix}.$$

In general, a $(L + 1)$-layer perceptron consists of $(L + 1)$ layers, each layer l taking linear combinations of the previous layer $(l - 1)$ (or the input) as input.

Neural Networks – Multilayer Perception

On input $x \in \mathbb{R}^D$, layer $l = 1$ computes a vector $y^{(1)} := \left(y_1^{(1)}, \dots, y_{m^{(1)}}^{(1)}\right)$ where

$$y_i^{(1)} = f\left(z_i^{(1)}\right) \text{ with } z_i^{(1)} = \sum_{j=1}^{D} w_{i,j}^{(1)} x_j + w_{i,0}^{(1)} .$$

- i-th component is called "unit i"
- f is called activation function and $w_{i,j}^{(1)}$ are adjustable weights.

Layer $l = 1$ computes linear combinations of the input and applies an (non-linear) activation function…

The first layer can be interpreted as *generalized* linear model:

$$y_i^{(1)} = f\left(\left(w_i^{(1)}\right)^T x + w_{i,0}^{(1)}\right).$$

Idea: recursively apply L additional layers on the output $y^{(1)}$ of the first layer.

Neural Networks – Multilayer Perception

In general, layer l computes a vector $y^{(l)} := \left(y_1^{(l)}, \cdots, y_{m^{(l)}}^{(l)}\right)$ as follows:

$$y_i^{(l)} = f\left(z_i^{(l)}\right) \text{ with } z_i^{(l)} = \sum_{j=1}^{m^{(l-1)}} w_{i,j}^{(l)} y_j^{(l-1)} + w_{i,0}^{(l)} .$$

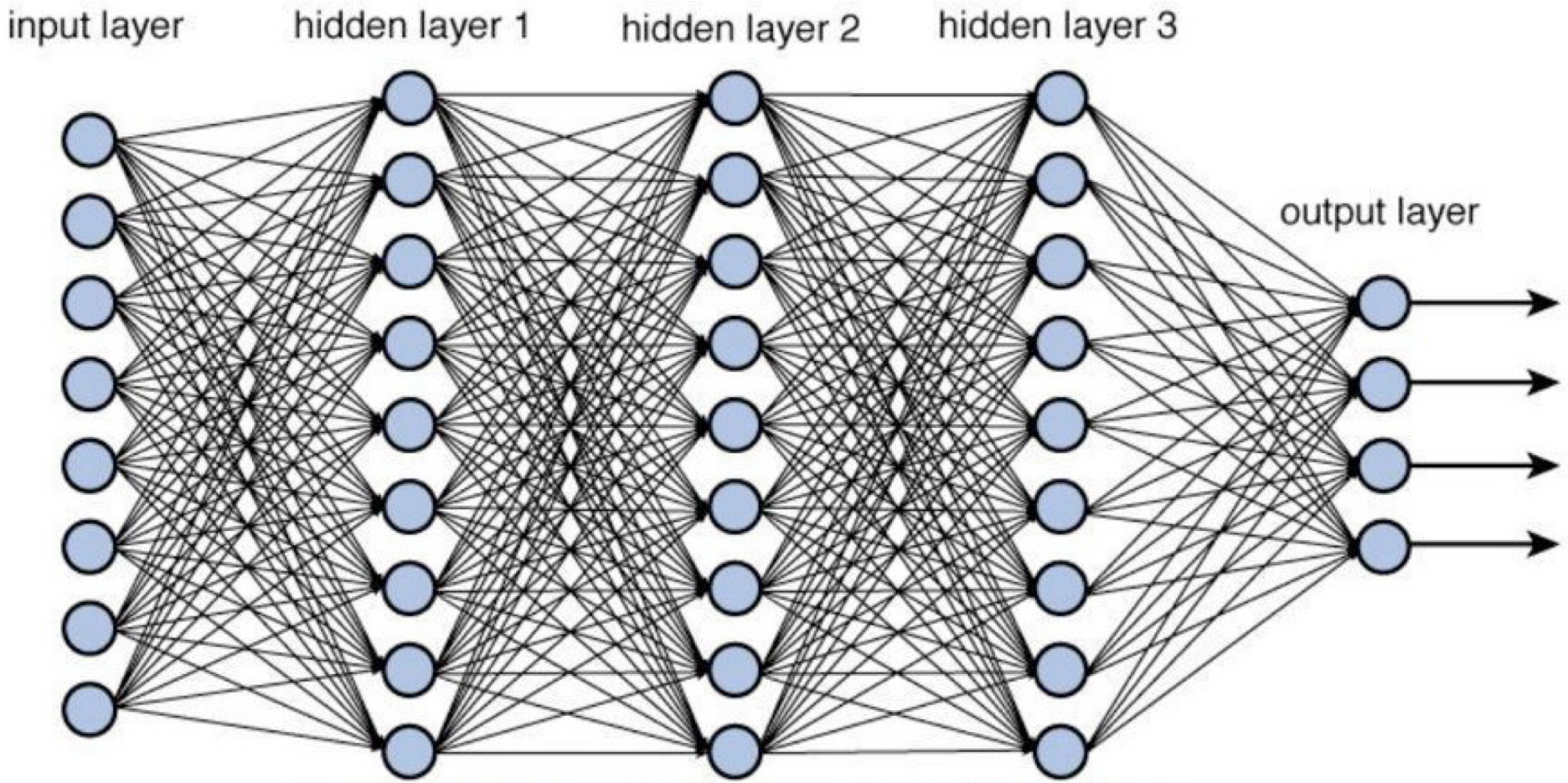
Thus, layer l computes linear combinations of layer $(l-1)$ and applies an activation function.

Layer (L+1) is called output layer because it computes the output of the multilayer perception:

$$y(x,w) = \begin{pmatrix} y_1(x,w) \\ \vdots \\ y_C(x,w) \end{pmatrix} := \begin{pmatrix} y_1^{(L+1)} \\ \vdots \\ y_C^{(L+1)} \end{pmatrix} = y^{(L+1)}$$

where $C = m^{(L+1)}$ is the number of output dimensions.

Neural Network – Deep Neural Network Graph



Neural Network – Activation Functions

How to choose the activation function f in each layer?

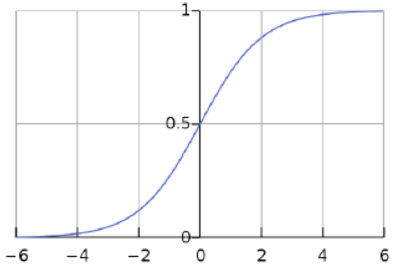
- Non-linear activation functions will increase the expressive power: Multilayer perceptron with $L + 1 \geq 2$ are universal approximators [1]!
- Depending on the applications: for classification we may want to interpret the output as posterior probabilities

$$y_i(x, w) = p(c = 1|x)$$

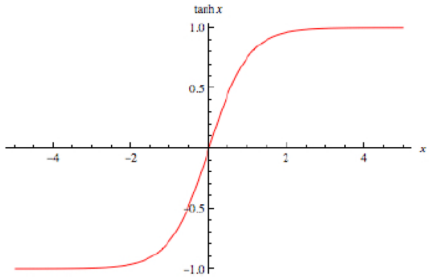
where c denotes the random variable for the class.

Neural Network – Activation Functions

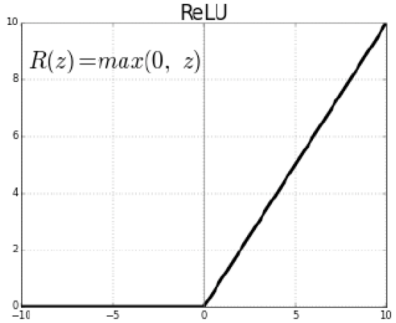
Sigmoid: $f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$



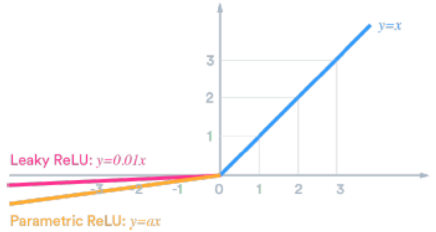
tanh: $f(x) = 2\sigma(2x) - 1$



ReLU: $f(x) = \max(0, x)$



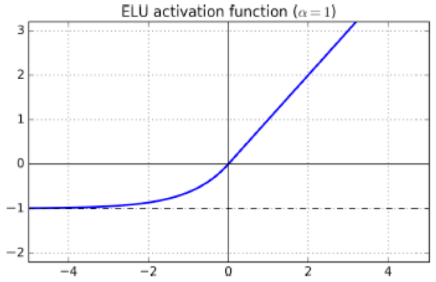
Leaky ReLU: $f(x) = \max(\alpha x, x)$



Maxout

$$\max(\mathbf{w}_1^T \mathbf{x} + b_1, \mathbf{w}_2^T \mathbf{x} + b_2)$$

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



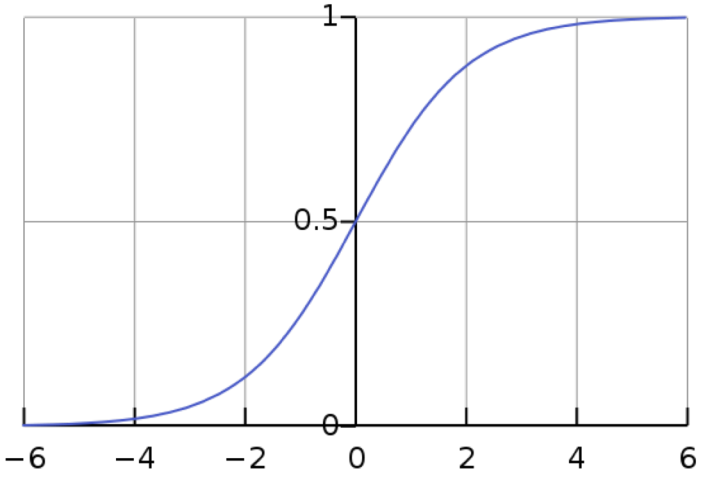
Neural Network – Activation Functions

Usually the activation function is chosen to be the logistic sigmoid:

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

which is non-linear, monotonic, and differentiable.

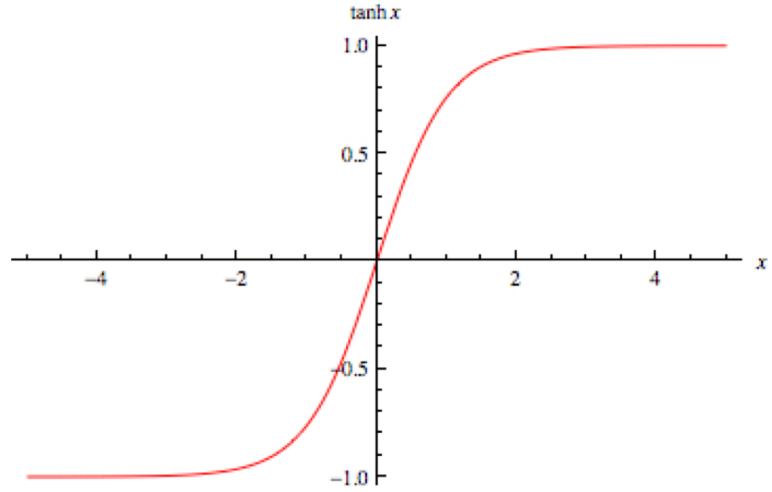
- Squashes numbers to range [0,1]
- Vanishing gradient problem
- The output of sigmoid function is not zero centered.



The hyperbolic tangent is used frequently:

$$\tanh(x) = 2\sigma(2x) - 1$$

- Simply a scaled sigmoid neuron
- Squashes numbers to range [-1,1]
- The output is zero centered
- Vanishing gradient problem
- Tanh non-linearly is always preferred to the sigmoid.

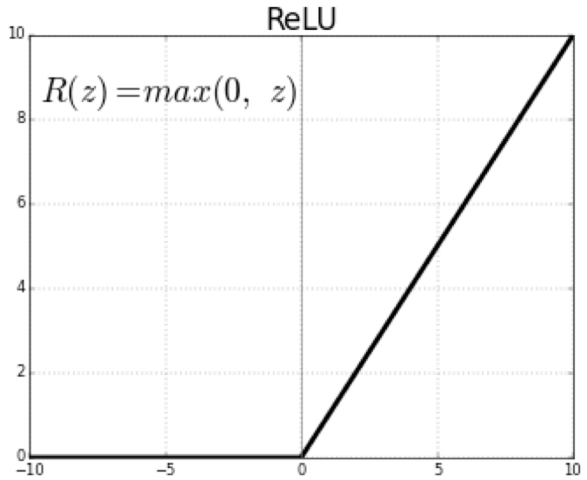


Neural Network – Activation Functions

ReLU: Reasons for success [Nair and Hinton ICML10]:

$$f(x) = \max(0, x)$$

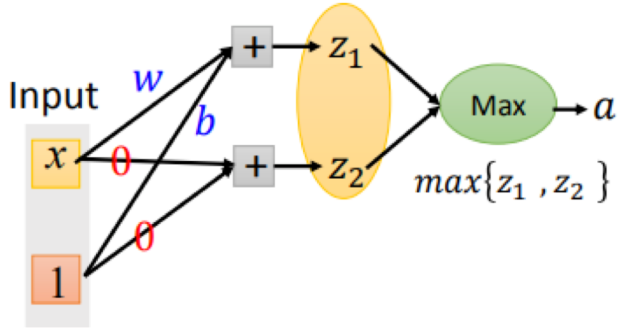
- Fast to compute – No exponential computations
- Solves the vanishing gradient problem
- Greatly accelerates SGD

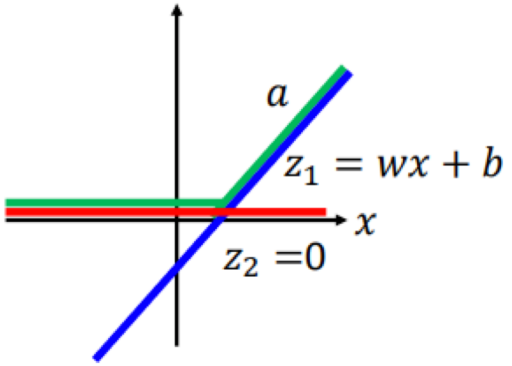


Maxout: Learnable activation function [Goodfellow et al. ICML13]:

$$f(x) = \max\left(w_1^T x + b_1, w_2^T x + b_2\right)$$

- Activation function in maxout network can be any piecewise linear convex function
- How many pieces depending on how many elements in a group
- Note that there is a significant increase in the number of parameters required for the model with maxout activation functions compared to the previous cases.





Neural Networks & Network Training

By now, we have a general model $y(\cdot, w)$ depending on W weights.
Idea: Learn the weights to perform regression or classification.

- We focus on classification.

Given a training set: $D_{train} = \{(x_n, y_n): 1 \leq n \leq N\}$

learn the mapping represented by D_{train} by minimizing the squared error using iterative optimization.

$$E(w) = \sum_{n=1}^{N} E_n(w) = \sum_{n=1}^{N} \sum_{k=1}^{C} \left(f_k(x_n, w) - y_{n,k}\right)^2$$

Neural Network & Network Training

We distinguish

- Stochastic Training A training sample (x_n, t_n) is chosen at random, and the weights w are updated to minimize $L_n(w)$.
- Batch and Mini-Batch Training A set $M \subseteq \{1, \dots, N\}$ of training samples is chosen and the weights w are updated based on the cumulative error $L_M(w) = \sum_{n \in M} L_n(w)$.
- And online training is possible.

Problem: How to minimize $L_n(w)$ (stochastic training)?

- $L_n(w)$ may be highly non-linear with many poor local minima.

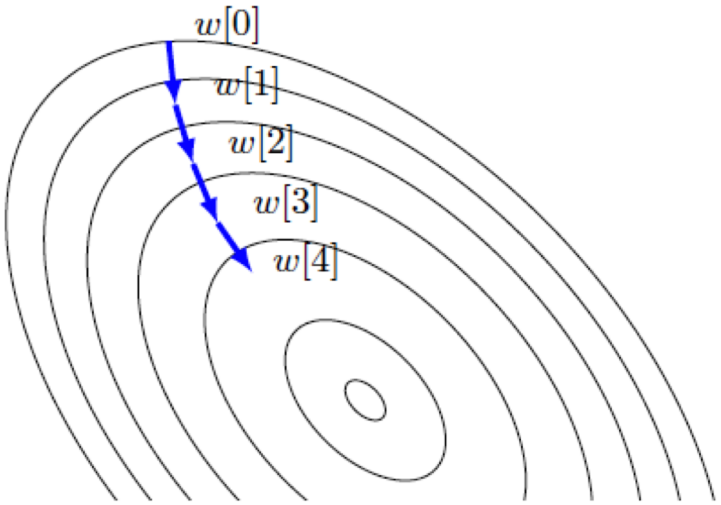
Framework for iterative optimization: Let

- $w[0]$ be an initial guess for the weights (several initialization techniques are available), and
- $w[t]$ be the weights at iteration t.

Neural Network & Network Training

In iteration $[t+1]$, choose a weight update $\Delta w[t]$ and set

$$w[t+1] = w[t] + \Delta \mathrm{w[t]}$$



Example: GD minimizes the error/loss $L_n(w)$ by taking steps in the direction of the negative gradient:

$$\Delta w[t] = -\eta \frac{\partial L_n}{\partial w[t]}$$

Problem: How to evaluate $\frac{\partial L_n}{\partial w[t]}$ in iteration $[t+1]$?

- "Error Backpropagation" allows to evaluate $\frac{\partial L_n}{\partial w[t]}$ in $\mathcal{O}(W)$!
- See the original paper "Learning Representations by Back Propagating Errors," by Rumelhart et al. [2] for further details.

Neural Network – Deep Learning

Multilayer perceptron are called deep if they have more than three layers: $L + 1 > 3$.

Motivation: Lower layers can automatically learn a hierarchy of features or a suitable dimensionality reduction.

- No hand-crafted features necessary anymore!

However, training deep neural networks is considered very difficult!

- Error measure represents a highly non-convex, “potentially intractable" [3] optimization problem.

Possible approaches:

- Different activation functions offer faster learning – e.g., ReLU or tanh(z)
- Unsupervised pre-training can be done layer-wise
- Further - See “Learning Deep Architectures for AI," by Y. Bengio [4] for a detailed discussion of state-of-the-art approaches to deep learning.

Neural Network – Summary

The multilayer perceptron represents a standard model of neural networks. They ...

- allow to tailor the architecture (layers, activation functions) to the problem;
- can be trained using gradient descent and error backpropagation;
- can be used for learning feature hierarchies (deep learning).

Deep learning is considered difficult.

Convolutional Neural Networks (CNN)

Convolutional networks represent specialized networks for application in computer vision:

- they accept images as raw input (preserving spatial information),
- and build up (learn) a hierarchy of features (no hand-crafted features necessary).

Problem: Internal workings of convolutional networks not well understood ...

- Unsatisfactory state for evaluation and research!

Idea: Visualize feature activations within the network ...

Convolutional Neural Networks

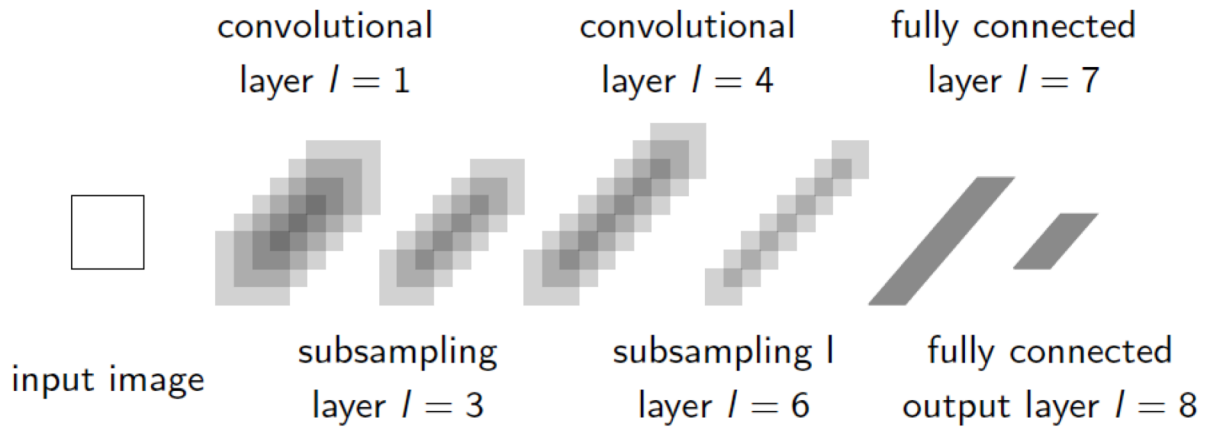


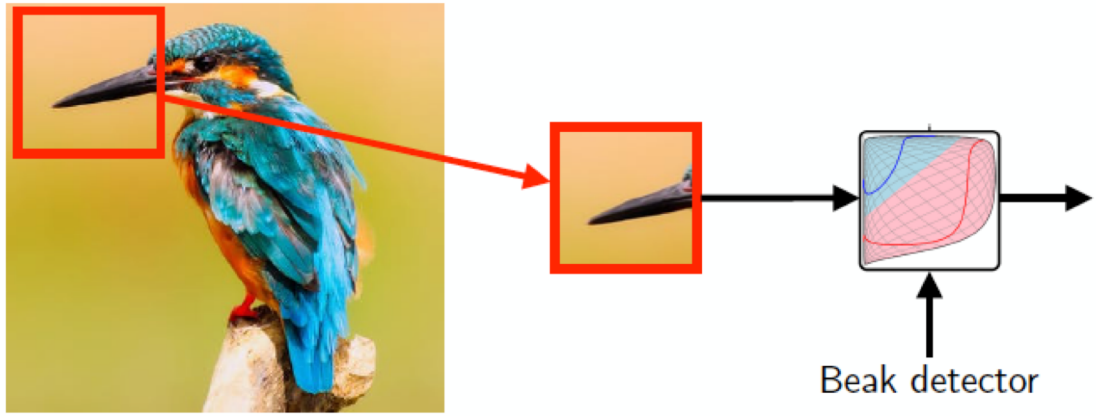
Figure: The architecture of the original convolution neural network, as introduced by LeCun et al. (1989)

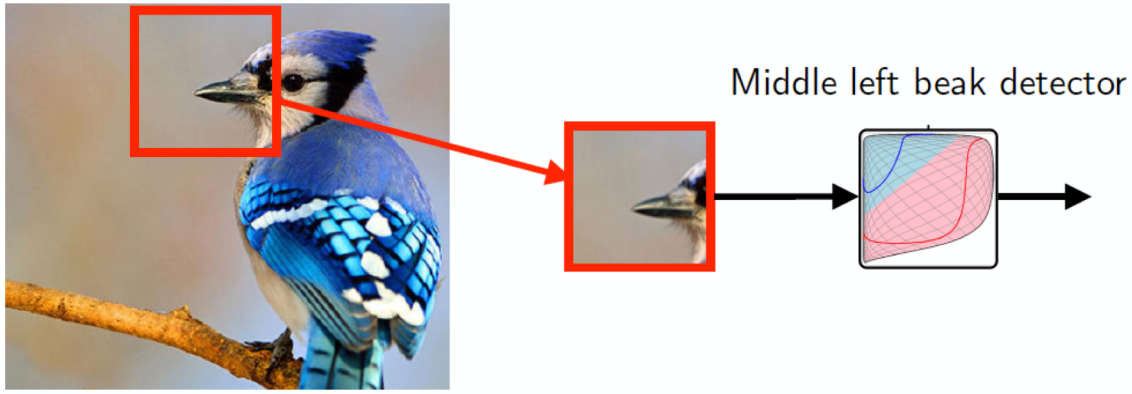
Original Convolutional Neural Network [5], [6] aims to build up a feature hierarchy by alternating

convolutional layer – non-linearity layer – subsampling layer

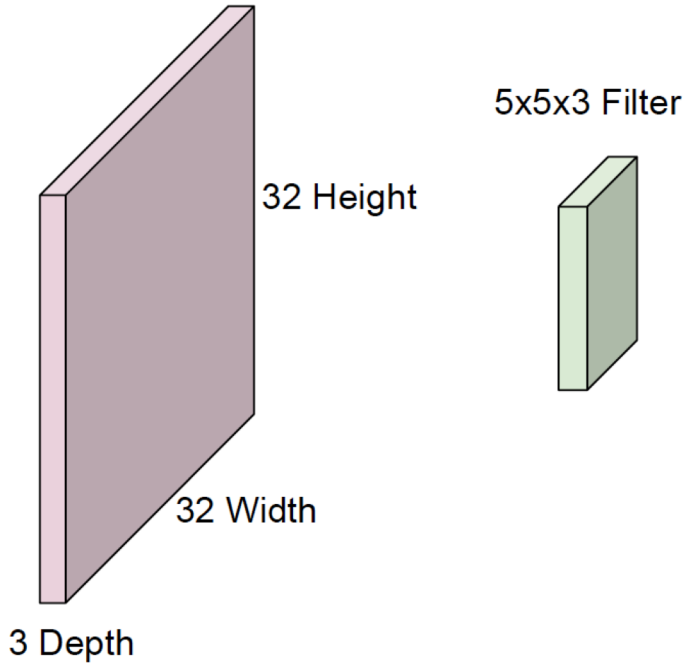
followed by a multilayer perceptron for classification.

Convolutional Neural Networks – Convolutional layer





- Some patterns are much smaller than the whole image
- A neuron does not have to see the whole image to discover the pattern.
- Connecting to small region with less parameters.



Convolutional Neural Networks – Convolutional Operation

- Idea: Allow raw image input while preserving the spatial relationship between pixels.
- Tool: Discrete convolution of image I with filter $K \in \mathbb{R}^{(2h_1+1)\times(2h_2+1)}$ is defined as

$$(I * K)_{r,s} = \sum_{u=-h_1}^{h_1} \sum_{v=-h_2}^{h_2} K_{(u,v)} I_{(r+u,s+v)}$$

where the filter K is given by

$$K = \begin{pmatrix} K_{-h_1,-h_2} & \cdots & K_{-h_1,h_2} \\ \vdots & K_{0,0} & \vdots \\ K_{h_1,-h_2} & \cdots & K_{h_1,h_2} \end{pmatrix}.$$

Convolutional Neural Networks – Convolutional Layer

Central part of convolutional network: convolutional layer.

- Can handle raw image input.

Idea: Apply a set of learned filters to the image in order to obtain a set of feature maps.

Can be repeated: Apply a different set of filters to the obtained feature maps to get more complex features:

- Generate a hierarchy of feature maps.

Convolutional Neural Networks – Convolutional layer

Let layer l be a convolutional layer.

Input: $m_1^{(l-1)}$ feature maps $Y_i^{(l-1)}$ of size $m_2^{(l-1)} \times m_3^{(l-1)}$ from the previous layer.

Output: $m_1^{(l)}$ feature maps of size $m_2^{(l)} \times m_3^{(l)}$ given by

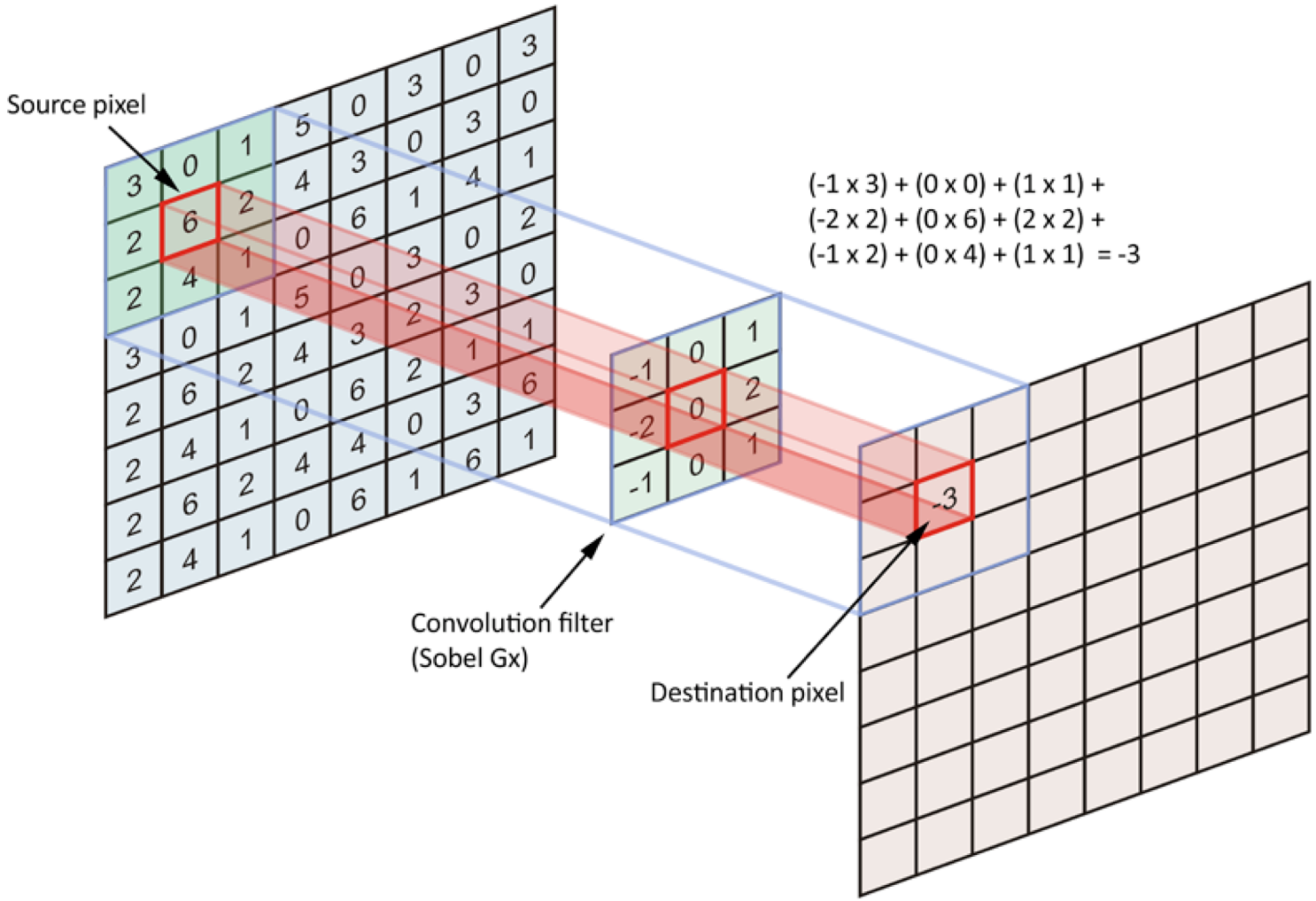
$$Y_i^{(l)} = B_i^{(l)} + \sum_{j=1}^{m_1^{(l)}} K_{i,j}^{(l)} * Y_j^{(l-1)}$$

where $B_i^{(l)}$ is called bias matrix and $K_{i,j}^{(l)}$ are the filters to be learned.

Notes:

- The size $m_2^{(l)} \times m_3^{(l)}$ of the output feature maps depends on the definition of discrete convolution (especially how borders are handled).
- The weights $w_{i,j}^{(l)}$ are hidden in the bias matrix $B_i^{(l)}$ and the filters $K_{i,j}^{(l)}$.

Convolutional Neural Networks – Convolutional Layer



Convolutional Neural Networks – Non-Linearity Layer

Let layer l be a non-linearity layer.

Given $m_1^{(l-1)}$ feature maps, a non-linearity layer applies an activation function to all these feature maps:

$$Y_i^{(l)} = f\left(Y_i^{(l-1)}\right)$$

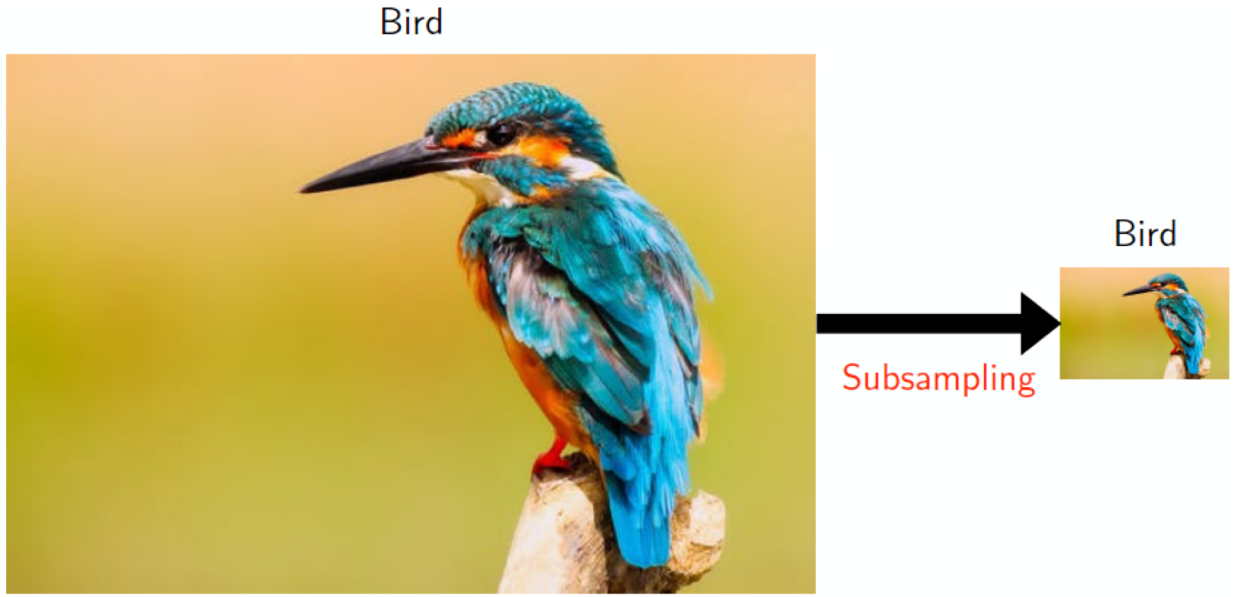
where f operates point-wise.

Usually, f is the hyperbolic tangent.

Layer l computes $m_1^{(l)} = m_1^{(l-1)}$ feature maps unchanged in size $\left(m_2^{(l)} = m_2^{(l-1)}, m_3^{(l)} = m_3^{(l-1)}\right)$.

Convolutional Neural Networks – Subsampling and Pooling Layer

Subsampling the pixels will not change the object



We can subsample the pixels to make image smaller: Less parameters for the network to process the image.

Convolutional Neural Networks – MaxPooling Layer

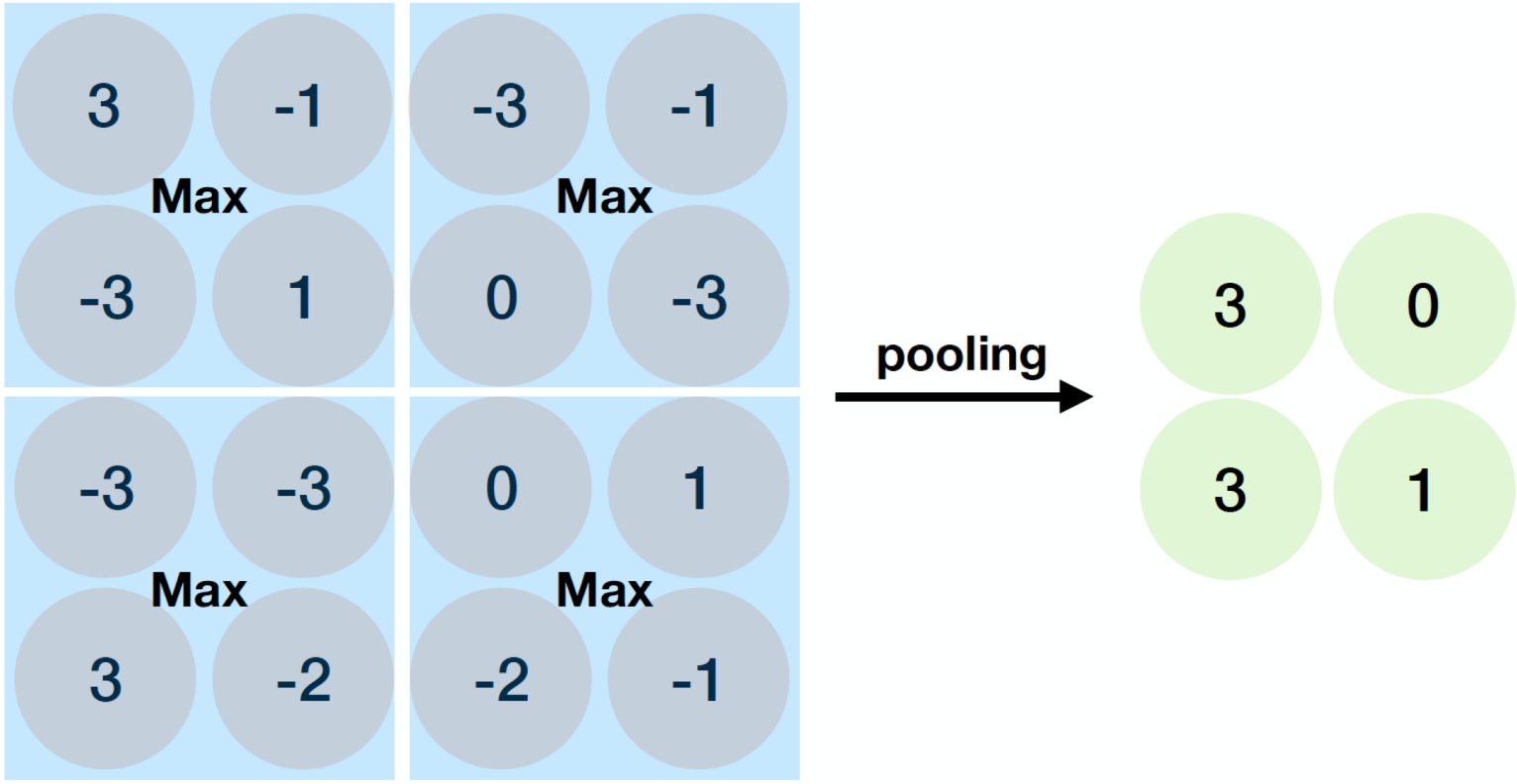


Figure: Max-pooling layer: 2×2 filter, stride 2

Convolutional Neural Networks – Putting it All Together

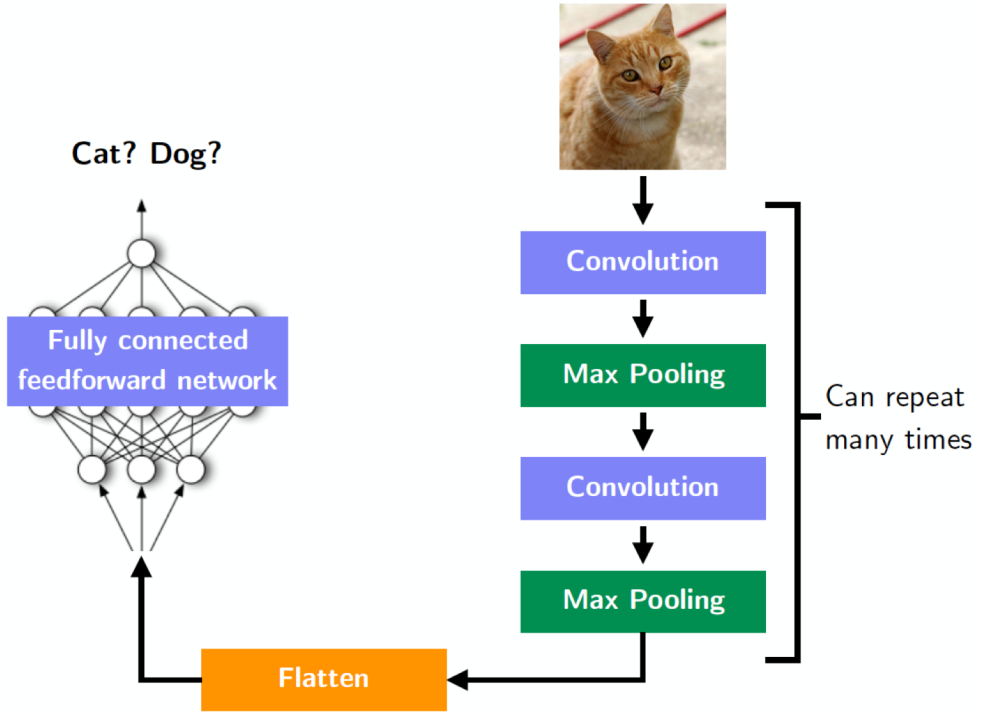
Remember: A convolutional network alternates

convolutional layer – non-linearity layer – subsampling layer

to build up a hierarchy of feature maps and uses a multilayer perceptron for classification.

Further details ...

- LeCun et al. [7] and Jarrett et al. [8] give a review of recent architectures.



Convolutional Neural Networks – Additional Layers

Researchers are constantly coming up with additional types of layers ...

- A rectification layer computes $Y_i^{(l)} = \left|Y_i^{(l-1)}\right|$ where the absolute value is computed point—wise.
- Local contrast normalization layers aim to enforce local competitiveness between adjacent feature maps (see Krizhevsky et al. [9] or LeCun et al. [7]).

Convolutional Neural Networks – Summary

A basic convolutional network consists of different types of layers:

- convolutional layers;
- non-linearity layers;
- subsampling layers.

Researchers are constantly thinking about additional types of layers to improve learning and performance.

Understanding Convolutional Networks

State: Convolutional networks perform well without requiring hand-crafted features.

- But: Learned feature hierarchy not well understood.

Idea: Visualize feature activations of higher convolutional layers ...

- Feature activations after first convolutional layer can be backprojected onto the image plane.

Zeiler et al. [10] propose a visualization technique based on deconvolutional networks.

Understanding Convolutional Networks – Deconvolutional Layer

Deconvolutional networks aim to build up a feature hierarchy ...

- by convolving the input image by a set of filters – like convolutional networks;
- however, they are fully unsupervised.

Idea: Given an input image (or a set of feature maps), try to reconstruct the input given the filters and their activations.

Let layer l be a deconvolutional layer.

Given feature maps $Y_i^{(l-1)}$ of the previous layer, try to reconstruct the input using the filters and their activations:

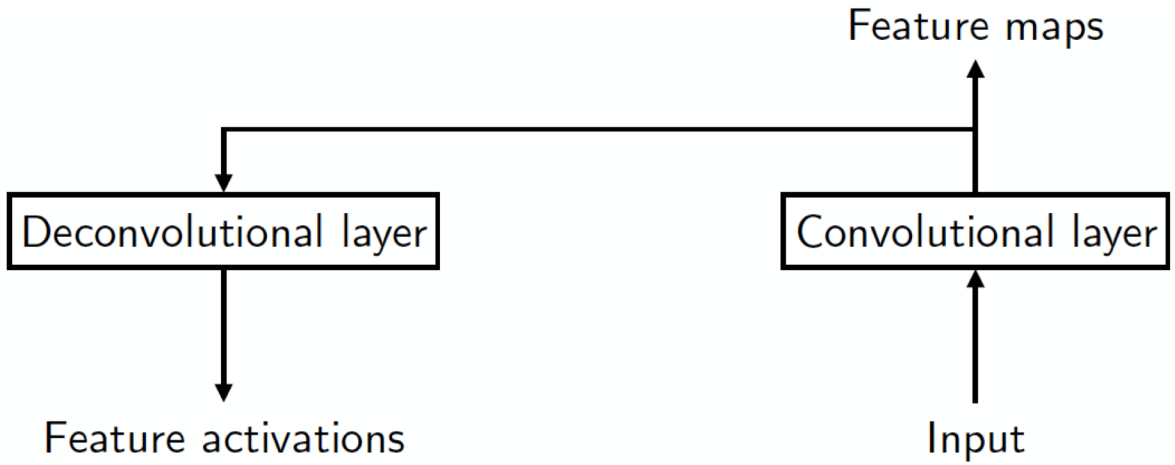
$$Y_i^{(l-1)} = \sum_{j=1}^{m_1^{(l)}} \left(K_{j,i}^{(l)}\right)^T * Y_j^{(l)}$$

Deconvolutional layers ...

- are unsupervised by definition; (Deconvolutional Networks," by Zeiler et al. [11] for details on how to train deconvolutional networks.)
- need to learn feature activations and filters.

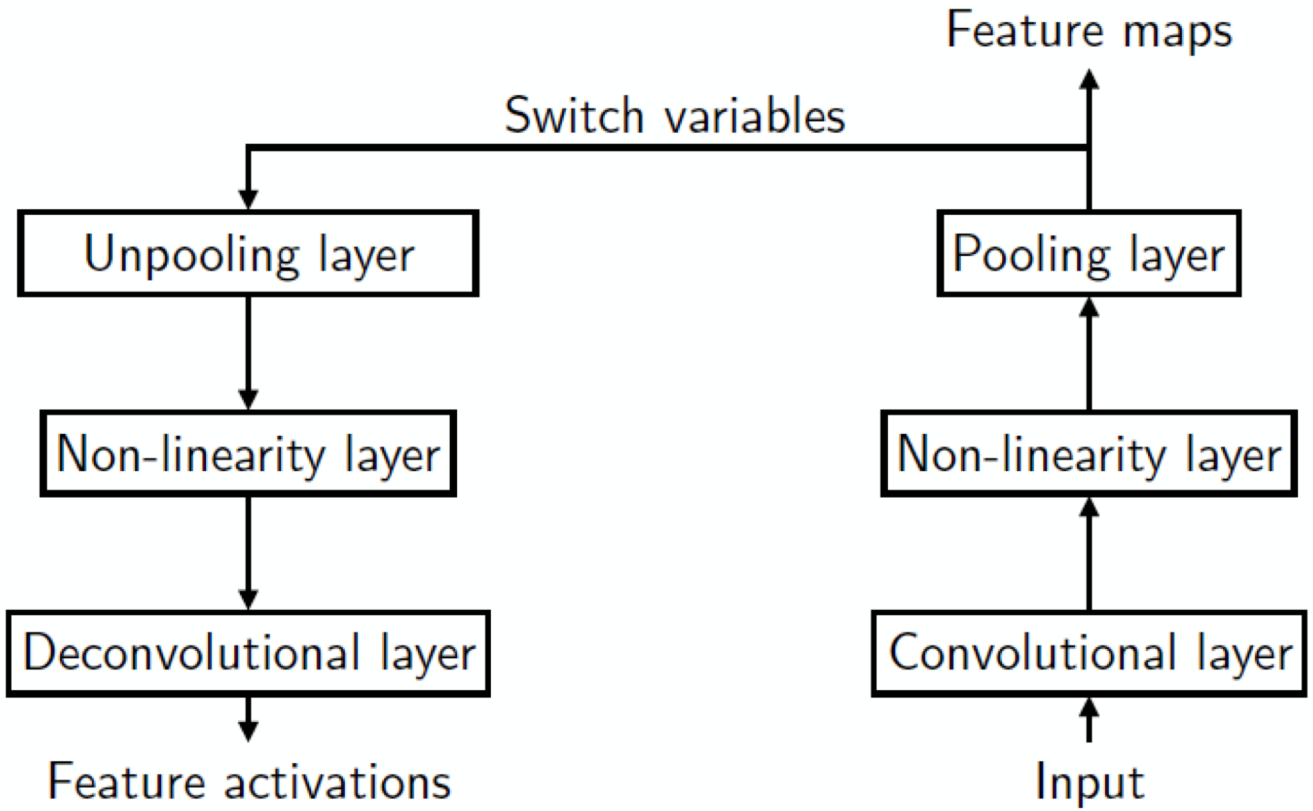
Understanding Convolutional Networks – Deconvolutional Layer

- Here: Deconvolutional layer used for visualization of trained convolutional network ...
  - filters are already learned – no training necessary.

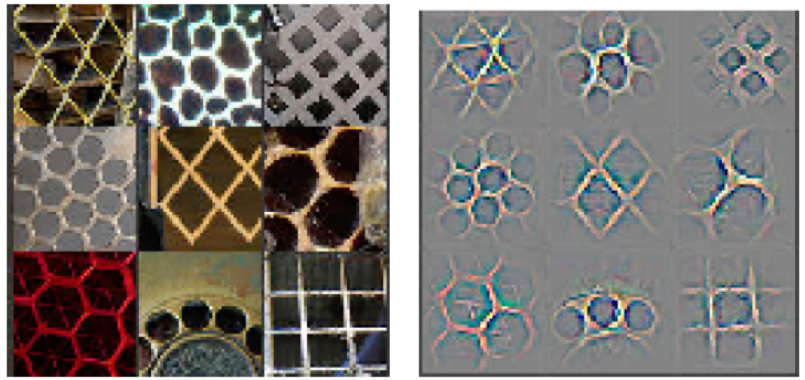


- Problem: Subsampling and pooling in higher layers.
- Remember: Placing windows at non-overlapping positions within the feature maps, pooling is accomplished by keeping one activation per window.
- Solution: Remember which pixels of a feature map were kept using so called “switch variables".

Understanding Convolutional Networks – Deconvolutional Layer



Understanding Convolutional Networks – Deconvolutional Layer



(a) Images. (b) Activations.

Figure: Activations of **layer 3** backprojected to pixel level [10].

(a) Images. (b) Activations.

Figure: Activations of **layer 4** backprojected to pixel level [10].

CNN – TensorFlow Implementation (Assignment 4)

- Weight Initialization
- Convolution and Pooling
- Convolution layer
- Fully connected layer
- Readout layer
- Reference: https://www.tensorow.org/get started/mnist/pros (See section: Build a Multilayer Convolutional Network)

CNN – TensorFlow Implementation

```
x = tf.placeholder(tf.float32, shape=[None, input_size])
y = tf.placeholder(tf.float32, shape=[None, classes_num])
```

```
def get_weights(shape):
    data = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(data)

def get_biases(shape):
    data = tf.constant(0.1, shape=shape)
    return tf.Variable(data)

def create_layer(shape):
    # Get the weights and biases
    W = get_weights(shape)
    b = get_biases([shape[-1]])

    return W, b

def convolution_2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1],
            padding='SAME')

def max_pooling(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
            strides=[1, 2, 2, 1], padding='SAME')
```

Input (placeholder)

- x is placeholder for input image
- y is label with one-hot representation, so second dimension of y is the number of classes
- None: the first dimension is the batch size, which can be any size.

Weight Initialization

- These variable will be initialized when user run `tf.global variables initializer'. Now they are just nodes in a graph without any value.

Convolution and Pooling

- Strides is 4-d: (num_samples, height, width, channels)
- padding: “same" means apply padding to keep output size as same as input size
- Conv2d pads with zeros and max pool pads with -inf.

CNN – TensorFlow Implementation

Reshape

- For example, tensor t = [[1,2],[3,4],[5,6],[7, 8]] has shape [4, 2]
- Reshape to [2, 4] will become [[1, 2, 3, 4], [5, 6, 7, 8]]
- Reshape to [-1, 4] will become [[1, 2, 3, 4], [5, 6, 7, 8]]
- -1 will be computed and becomes 2.

Fully Connected Layer

- Flatten all the maps and connect them with fully connected layer
- pay attention to the shape

Readout Layer

Conclusion

Convolutional networks perform well in computer vision tasks as they learn a feature hierarchy.

Internal workings of convolutional networks are not well understood.

- Zeiler and Fergus [10] use deconvolutional networks to visualize feature activations;
- this allows to analyze the feature hierarchy and to increase performance.
    - For example by adjusting the filter size and subsampling scheme.

Reference

[1] K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators," Neural Networks, vol. 2, no. 5, pp. 359-366, 1989.

[2] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Parallel distributed processing: Explorations in the microstructure of cognition," in, Cambridge: MIT Press, 1986, ch. Learning Representations by Back-Propagating Errors, pp. 318-362.

[3] D. Erhan, P.-A. Manzagol, Y. Bengio, S. Bengio, and P. Vincent, “The difficulty of training deep architectures and the effect of unsupervised pre-training," in Artificial Intelligence and Statistics, International Conference on, 2009, pp. 153-160.

[4] Y. Bengio, “Learning deep architectures for AI,” Foundations and Trends in Machine Learning, no. 1, pp. 1-127, 2009.

[5] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," Neural Comput., vol. 1, no. 4, pp. 541-551, Dec. 1989, issn: 0899-7667. doi/10.1162/neco.1989.1.4.541. [Online]. Available: http://dx.doi.org/10.1162/neco.1989.1.4.541.

[6] Y. Lecun, L. Bottou, Y. Bengio, and P. Haner, "Gradient-based learning applied to document recognition,” in Proceedings of the IEEE, 1998, pp. 2278-2324.

[7] Y. LeCun, K. Kavukvuoglu, and C. Farabet, "Convolutional networks and applications in vision," in Circuits and Systems, International Symposium on, 2010, pp. 253-256.

[8] K. Jarrett, K. Kavukcuogl, M. Ranzato, and Y. LeCun, "What is the best multi-stage architecture for object recognition?," in Computer Vision, International Conference on, 2009, pp. 2146-2153.

Reference

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in Advances in Neural Information Processing Systems, 2012, pp. 1097-1105.
[10] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," Computing Research Repository, vol. abs/1311.2901, 2013.
[11] M. D. Zeiler, D. Krishnan, G. W. Taylor, and R. Fergus, "Deconvolutional networks," in Computer Vision and Pattern Recognition, Conference on, 2010, pp. 2528-2535.

Next Lecture

Long short term memory
NLP