ECBM E6040 Neural Networks and Deep Learning

Lecture #10: Optimization for Training Deep Models (cont'd), and Convolutional Networks

Aurel A. Lazar

Columbia University
Department of Electrical Engineering

April 5, 2016

Outline of Part I

Outline of Part II

3. Approximate Second-Order Methods
   - Newton's Method
   - Conjugate Gradients
   - BFGS

4. Optimization Strategies and Meta-Algorithms
   - Coordinate Descent
   - Initialization Strategies

Outline of Part III

Part I

Review of Previous Lecture

Topics Covered

- Optimization for Model Training
- Challenges in Neural Network Optimization
- Basic Learning Algorithms
- Algorithms for Adaptive Learning Rates

Learning Objectives

- Major challenges: ill-conditioning, local minima, saddle points, just to name a few.
- Stochastic gradient descent and variants are the algorithms of choice in neural networks.
- The choice of which algorithm to use depends as much on the users familiarity with the algorithm as it does on any established notion of superior performance.

Part II

Optimization for Training Deep Models (cont'd)

Overview

We'll investigate second-order methods to the training of deep networks by examining the empirical risk:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\hat{p}} L(f(\mathbf{x}; \boldsymbol{\theta}), y) = \frac{1}{m} \sum_{i=1}^{m} L(f(\mathbf{x}^i; \boldsymbol{\theta}), y^i),$$

where m is the number of training examples and $\hat{p}(\mathbf{x}, y)$ is the empirical distribution.

Newton's Updating Algorithm

Given the objective function $J(\boldsymbol{\theta})$, the Hessian matrix of J with respect to $\boldsymbol{\theta}$ is defined by

$$[\mathbf{H}(J)(\boldsymbol{\theta})]_{ij} = \frac{\partial^2}{\partial\theta_i\partial\theta_j}J(\boldsymbol{\theta}).$$

Newton's method to be developed here is based on using a second-order Taylor series expansion to approximate $J(\boldsymbol{\theta})$ near some point $\boldsymbol{\theta}_0$, ignoring derivatives of higher order, i.e.,

$$J(\boldsymbol{\theta}) = J(\boldsymbol{\theta}_0)+(\boldsymbol{\theta}-\boldsymbol{\theta}_0)^T\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta}_0)+\frac{1}{2}(\boldsymbol{\theta}-\boldsymbol{\theta}_0)^T\mathbf{H}(J)(\boldsymbol{\theta}_0)(\boldsymbol{\theta}-\boldsymbol{\theta}_0).$$

At the critical point we obtain the Newton parameter update rule:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - [\mathbf{H}(J)(\boldsymbol{\theta}_0)]^{-1}\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta}_0).$$

If the objective function is convex but not quadratic, this update can be iterated (see Algorithm 1.)

Algorithm 1: Newton's method with objective $J(\boldsymbol{\theta})$

Algorithm 1 Newton's method with objective $J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\mathbf{x}^i;\boldsymbol{\theta}), y^i)$

Require: Initial parameter $\boldsymbol{\theta}_0$.

while Stopping criterion not met **do**
  Initialize the gradient $\mathbf{g} = \mathbf{0}$
  Initialize the Hessian $\mathbf{H} = \mathbf{0}$
  for i=1 to m **do**
    Compute gradient $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i;\boldsymbol{\theta}), y^i)/m$
    Compute Hessian $\mathbf{H} \leftarrow \mathbf{H} + \nabla^2_{\boldsymbol{\theta}} L(f(\mathbf{x}^i;\boldsymbol{\theta}), y^i)/m$
  end for
  Compute Hessian inverse: $\mathbf{H}^{-1}$
  Compute update: $\Delta\boldsymbol{\theta} = \mathbf{H}^{-1}\mathbf{g}$
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \Delta\boldsymbol{\theta}$
end while

Intuition Behind the Newton Algorithm

Change of Variables

The Hessian can be interpreted as a transformation from the Euclidean space where the problem is defined to a space where gradient optimization is simplified. Since $\mathbf{H}$ is a real symmetric matrix, it can be represented as $\mathbf{H} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T$, where $\mathbf{Q}$ is an orthogonal matrix.

We consider the alternative parametrization

$$\boldsymbol{\phi} = \mathbf{\Lambda}^{1/2}\mathbf{Q}^T\boldsymbol{\theta}$$

and note that by the chain rule

$$\nabla_{\boldsymbol{\phi}} J(\boldsymbol{\theta}_0) = \left[\frac{\partial \boldsymbol{\theta}}{\partial \boldsymbol{\phi}}\right]^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) = \mathbf{Q}\mathbf{\Lambda}^{-1/2}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0).$$

or equivalently, $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) = \mathbf{\Lambda}^{1/2}\mathbf{Q}^T\nabla_{\boldsymbol{\phi}} J(\boldsymbol{\theta}_0)$.

Intuition Behind the Newton Algorithm (cont'd)

Change of Variables

$$
\begin{aligned}
J(\boldsymbol{\theta}) &= J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \mathbf{H}(J)(\boldsymbol{\theta}_0)(\boldsymbol{\theta} - \boldsymbol{\theta}_0) \\
&= J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \mathbf{Q}\boldsymbol{\Lambda}^{1/2} \nabla_{\boldsymbol{\phi}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^T(\boldsymbol{\theta} - \boldsymbol{\theta}_0) \\
&= J(\boldsymbol{\theta}_0) + (\boldsymbol{\Lambda}^{1/2}\mathbf{Q}^T\boldsymbol{\theta} - \boldsymbol{\Lambda}^{1/2}\mathbf{Q}^T\boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\phi}} J(\boldsymbol{\theta}_0) + \\
&\qquad + (\boldsymbol{\Lambda}^{1/2}\mathbf{Q}^T\boldsymbol{\theta} - \boldsymbol{\Lambda}^{1/2}\mathbf{Q}^T\boldsymbol{\theta}_0)^T (\boldsymbol{\Lambda}^{1/2}\mathbf{Q}^T\boldsymbol{\theta} - \boldsymbol{\Lambda}^{1/2}\mathbf{Q}^T\boldsymbol{\theta}_0) \\
&= J(\boldsymbol{\theta}_0) + (\boldsymbol{\phi} - \boldsymbol{\phi}_0)^T \nabla_{\boldsymbol{\phi}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\phi} - \boldsymbol{\phi}_0)^T (\boldsymbol{\phi} - \boldsymbol{\phi}_0).
\end{aligned}
$$

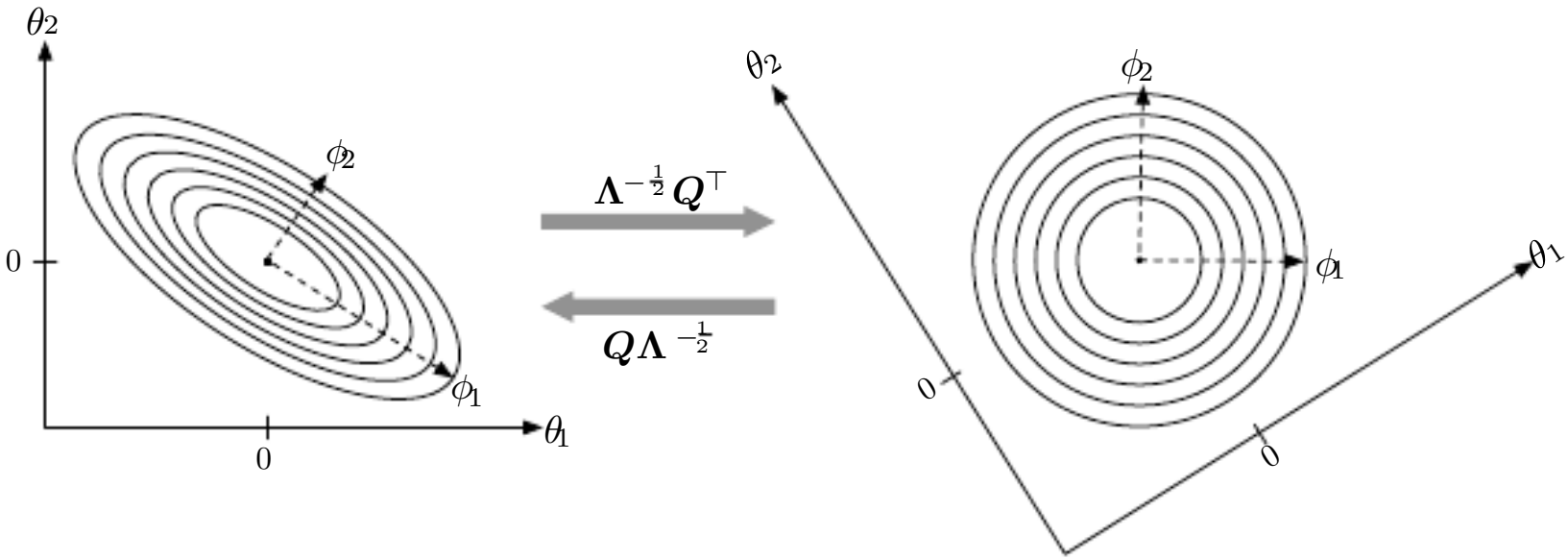
Computing the gradient with respect to $\boldsymbol{\phi}$ and setting this to zero yields the Newton update in $\boldsymbol{\phi}$-space:

$$\boldsymbol{\phi}^* = \boldsymbol{\phi}_0 - \nabla_{\boldsymbol{\phi}} J(\boldsymbol{\theta}_0).$$

That is, in $\boldsymbol{\phi}$-space, the Newton update is transformed into a standard gradient step with unit learning rate

Intuition Behind the Newton Algorithm (cont'd)

Change of Variables



Newton's method maps an arbitrary and possibly ill-conditioned quadratic objective function in parameter space $\boldsymbol{\theta}$ into an alternative parameter space $\boldsymbol{\phi}$ where the quadratic objective function is isometric.

Regularizing the Hessian

If the eigenvalues of the Hessian are not all positive near a saddle point, then Newton's method may cause updates to move in the wrong direction. This situation can be avoided by regularizing the Hessian. The regularized update becomes

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - [\mathbf{H}(J)(\boldsymbol{\theta}_0) + \alpha\mathbf{I}]^{-1}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0).$$

The algorithm works fairly well as long as the negative eigenvalues of the Hessian are still relatively close to zero. In cases where there are more extreme directions of curvature, the value of α would have to be sufficiently large to offset the negative eigenvalues. However, as α increases in size, the Hessian becomes dominated by the $\alpha\mathbf{I}$ diagonal and the direction chosen by Newton's method converges to the standard gradient.

Limitations of Newton's Method

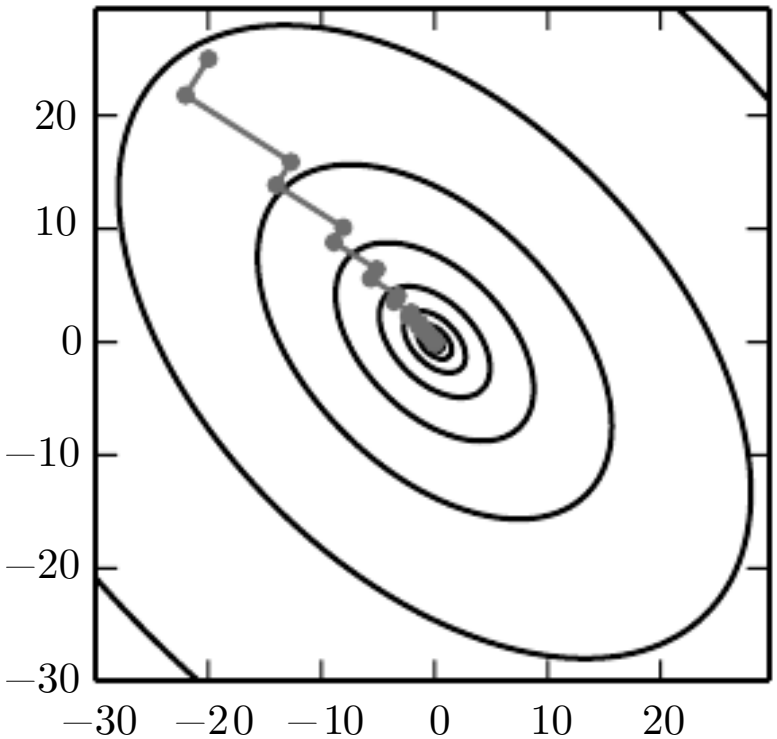
Newton's method for training large neural networks is limited by the significant computational burden it imposes as the inverse Hessian has to be computed at every training iteration.

Newton's method would require the inversion of a $K \times K$ matrix - with computational complexity of $\mathcal{O}(K^3)$.

As a consequence, only networks with very small number of parameters can be practically trained via Newton's method.

The Method of Conjugate Gradients

In the method of steepest descent, line searches are applied iteratively in the direction associated with the gradient.



When applied in a quadratic bowl, steepest descent progresses in a rather ineffective back-and-forth, zig-zag pattern as each line search direction (the gradient) is guaranteed to be orthogonal to the previous line search direction.

Conjugate gradients is a method to efficiently avoid the calculation of the inverse Hessian by iteratively descending conjugate directions.

The Method of Conjugate Gradients (cont'd)

In the method of conjugate gradients, we seek to find a search direction that is conjugate to the previous line search direction, i.e., it will not undo progress made in that direction.
At training iteration t, the next search direction $\mathbf{d}_t$ takes the form:

$$\mathbf{d}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \beta_t \mathbf{d}_{t-1}$$

were β_t is a coefficient whose magnitude controls how much of the direction, $\mathbf{d}_{t-1}$, we should add back to the current search direction.

Two directions, $\mathbf{d}_t$ and $\mathbf{d}_{t-1}$, are defined as conjugate if

$$\mathbf{d}_t^T \mathbf{H}(J) \mathbf{d}_{t-1}.$$

The Method of Conjugate Gradients (cont'd)

We have

$$\begin{aligned}
\mathbf{d}_t^T \mathbf{H}\, \mathbf{d}_{t-1} &= 0 \\
\mathbf{d}_t^T \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T\, \mathbf{d}_{t-1} &= 0 \\
\left(\mathbf{\Lambda}^{1/2}\mathbf{Q}^T\, \mathbf{d}_t\right)^T \left(\mathbf{\Lambda}^{1/2}\mathbf{Q}^T\, \mathbf{d}_{t-1}\right) &= 0 \\
\left(\mathbf{d}_t^{\phi}\right)^T \mathbf{d}_{t-1}^{\phi} &= 0,
\end{aligned}$$

where $\mathbf{d}_t^{\phi} = \mathbf{\Lambda}^{1/2}\mathbf{Q}^T\, \mathbf{d}_t$ as the direction $\mathbf{d}_t$ transformed to ϕ-space. The method of conjugate gradients can be interpreted as the application of the method of steepest descent in ϕ-space. The conjugate directions can be computed using the eigen-decomposition of the Hessian, $\mathbf{H}$.

The Method of Conjugate Gradients (cont'd)

Can we calculate the conjugate directions without computing the Hessian, its inverse or its decomposition?

1. Fletcher-Reeves:

$$\beta_t = \frac{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})}$$

2. Polak-Ribiere:

$$\beta_t = \frac{\left(\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t) - \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})\right)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})}$$

Algorithm 2: Conjugate Gradient Method

Algorithm 2 Conjugate Gradient Method

Require: Initial parameter $\boldsymbol{\theta}_0$.

Initialize $\boldsymbol{\rho}_0 = \mathbf{0}$

while Stopping criterion not met **do**

Initialize the gradient $\mathbf{g}_t = \mathbf{0}$

for i=1 to m **do** % loop over the training set.

Compute gradient $\mathbf{g}_t \leftarrow \mathbf{g}_t + \frac{1}{m}\nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), y^i)$

end for

Compute: $\beta_t = \frac{(\mathbf{g}_t - \mathbf{g}_{t-1})^T \mathbf{g}_t}{\mathbf{g}_{t-1}^T \mathbf{g}_{t-1}}$ (Polak - Ribière)

Compute search directions: $\boldsymbol{\rho}_t = -\mathbf{g}_t + \beta_t \boldsymbol{\rho}_{t-1}$

Perform line search to find:

$\eta^* = \text{argmin}_\eta \frac{1}{m} \sum_{i=1}^m L(f(\mathbf{x}^i; \boldsymbol{\theta}), y^i)$

Apply update: $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \eta^* \boldsymbol{\rho}_t$

end while

The Broyden - Fletcher - Goldfarb - Shanno Algorithm

The Newton algorithm

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - [\mathbf{H}(J)(\boldsymbol{\theta}_0)]^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0).$$

can be re-written as

$$\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t = -\mathbf{H}^{-1} \left(\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t+1}) - \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t) \right)$$

The BFGS method is based on approximating the inverse of the Hessian with a matrix $\mathbf{M}$ that is updated as

$$\mathbf{M}_t = \mathbf{M}_{t-1} + (1 + \frac{\boldsymbol{\phi}^T \mathbf{M}_{t-1} \boldsymbol{\phi}}{\boldsymbol{\Delta}^T \boldsymbol{\phi}}) \frac{\boldsymbol{\phi} \boldsymbol{\phi}^T}{\boldsymbol{\Delta}^T \boldsymbol{\phi}} - \frac{\boldsymbol{\Delta} \boldsymbol{\phi}^T \mathbf{M}_{t-1} + \mathbf{M}_{t-1} \boldsymbol{\phi} \boldsymbol{\Delta}^T}{\boldsymbol{\Delta}^T \boldsymbol{\phi}},$$

where $\boldsymbol{\phi} = \mathbf{g}_t - \mathbf{g}_{t-1}$, $\mathbf{g}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$ and $\boldsymbol{\Delta} = \boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-1}$.

Algorithm 3: The BFGS Method

Algorithm 3 The BFGS Method

Require: Initial parameter $\boldsymbol{\theta}_0$.

Initialize inverse Hessian $\mathbf{M}_0 = \mathbf{I}$

while Stopping criterion not met **do**

Compute gradient $\mathbf{g}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$ (via batch back-prop)

Compute $\boldsymbol{\phi} = \mathbf{g}_t - \mathbf{g}_{t-1}$, $\boldsymbol{\Delta} = \boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-1}$

Approx $\mathbf{H}^{-1}$: $\mathbf{M}_t = \mathbf{M}_{t-1} + (1 + \frac{\boldsymbol{\phi}^T \mathbf{M}_{t-1} \boldsymbol{\phi}}{\boldsymbol{\Delta}^T \boldsymbol{\phi}}) \frac{\boldsymbol{\phi}\boldsymbol{\phi}^T}{\boldsymbol{\Delta}^T \boldsymbol{\phi}} - \frac{\boldsymbol{\Delta}\boldsymbol{\phi}^T \mathbf{M}_{t-1} + \mathbf{M}_{t-1}\boldsymbol{\phi}\boldsymbol{\Delta}^T}{\boldsymbol{\Delta}^T \boldsymbol{\phi}}$

Compute search directions: $\boldsymbol{\rho}_t = \mathbf{M}_t \mathbf{g}_t$

Perform line search to find: $\eta^* = \text{argmin}_\eta \, J(\boldsymbol{\theta}_t + \eta \boldsymbol{\rho}_t)$

Apply update: $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \eta^* \boldsymbol{\rho}_t$

end while

Limited Memory BFGS

The BFGS algorithm must store the inverse Hessian matrix, $\mathbf{M}$, that requires $\mathcal{O}(n^2)$ memory, making BFGS impractical for most modern deep learning models that typically have millions of parameters. We set $\mathbf{M}_{t-1} = \mathbf{I}$ to obtain:

$$\mathbf{M}_t = \mathbf{I} + (1 + \frac{\boldsymbol{\phi}^T \boldsymbol{\phi}}{\boldsymbol{\Delta}^T \boldsymbol{\phi}}) \frac{\boldsymbol{\phi} \boldsymbol{\phi}^T}{\boldsymbol{\Delta}^T \boldsymbol{\phi}} - \frac{\boldsymbol{\Delta} \boldsymbol{\phi}^T + \boldsymbol{\phi} \boldsymbol{\Delta}^T}{\boldsymbol{\Delta}^T \boldsymbol{\phi}}.$$

Therefore, the direction update formula becomes

$$\boldsymbol{\rho}_t = \mathbf{g}_t + a\boldsymbol{\phi} + b\boldsymbol{\Delta},$$

where the scalars a and b are given by

$$a = (1 + \frac{\boldsymbol{\phi}^T \boldsymbol{\phi}}{\boldsymbol{\Delta}^T \boldsymbol{\phi}}) \frac{\boldsymbol{\phi}^T \mathbf{g}_t}{\boldsymbol{\Delta}^T \boldsymbol{\phi}} - \frac{\boldsymbol{\Delta}^T \mathbf{g}_t}{\boldsymbol{\Delta}^T \boldsymbol{\phi}} \quad \text{and} \quad b = -\frac{\boldsymbol{\phi}^T \mathbf{g}_t}{\boldsymbol{\Delta}^T \boldsymbol{\phi}}$$

Coordinate Descent

Coordinate descent is a methodology of optimizing along one coordinate at a time. Block coordinate descent refers to minimizing with respect to a subset of the variables simultaneously.

Divide and conquer: different variables in the optimization problem are separated into groups that play relatively isolated roles. For example, consider the cost function

$$J(\mathbf{X}, \mathbf{W}) = \sum_{ij} |H_{ij}| + \sum_{ij} \left(\mathbf{X} - \mathbf{W}^T \mathbf{H}\right)^2_{ij}.$$

This function describes a learning problem called sparse coding, where the goal is to find a weight matrix $\mathbf{W}$ that can linearly decode a matrix of activation values $\mathbf{H}$ to reconstruct the training set $\mathbf{X}$.

Coordinate Descent (cont'd)

In full generality finding $\min J$ is hard because J is not convex.

However, we can divide the inputs to the training algorithm into two sets: the dictionary parameters $\mathbf{W}$ and the code representations $\mathbf{H}$. Minimizing the objective function with respect to either one of these sets of variables is a convex problem.

Block coordinate descent thus gives us an optimization strategy that allows us to use efficient convex optimization algorithms.

Initialization Strategies

Training deep models is a sufficiently difficult task that most algorithms are strongly affected by the choice of initialization. The initial point can determine whether

- the algorithm converges at all,
- they are unstable and, thereby, the algorithm encounters numerical difficulties,
- the algorithm fails altogether.

When learning does converge, the initial point can determine how quickly learning converges and whether it converges to a point with high or low cost. Also, points of comparable cost can have wildly varying generalization error, and the initial point can affect the generalization as well.

Initialization Strategies (cont'd)

The only property known with complete certainty is that the initial parameters need to break symmetry between different units:

- if two units have the same initial parameters, then a deterministic learning algorithm applied to a deterministic cost and model will constantly update both of these units in the same way;
- if the model or training algorithm is capable of using stochasticity to compute different updates for different units, it is usually best to initialize each unit to compute a different function from all of the other units.

Weights are usually initialized to values drawn from a Gaussian or uniform distribution. The choice of the distribution does not seem to matter; the scale of the initial distribution, however, does have a large effect on both the outcome of the optimization procedure and on the ability of the network to generalize.

Initialization Strategies (cont'd)

Regularization and optimization can give very different insights into how we should initialize a network:

- optimization suggests weights that are large enough to propagate information successfully,
- regularization encourages making weigths smaller.

Initializing the parameters $\boldsymbol{\theta}$ to $\boldsymbol{\theta}_0$ is similar to imposing a Gaussian prior $p(\boldsymbol{\theta})$ with mean $\boldsymbol{\theta}_0$. The prior implies that it is more likely that units do not interact with each other than that they do interact.

Units interact only if the likelihood term of the objective function expresses a strong preference for them to interact. If we initialize $\boldsymbol{\theta}_0$ to large values, then our prior specifies which units should interact with each other, and in pre-specified ways.

Part III

Convolutional Networks

The Convolution Operation

The convolution in a simplified form is defined by

$$s(t) = \int_{\mathbb{D}} x(a)w(t-a)da$$

and it is usually denoted by

$$s(t) = (x * w)(t).$$

Here x denotes the input, w denotes the kernel and s is the output, also referred to as the feature map. In discrete time notation

$$s[t] = (x * w)[t] = \sum_{a \in \mathbb{N}} x[a]w[t-a].$$

The Convolution Operation (cont'd)

The input is usually a multidimensional array of data and the kernel is usually a multidimensional array of learn-able parameters referred to as tensors. If the input is is a two-dimensional image I and K is a kernel is a two-dimensional kernel

$$s[i,j] = (I * K)[i,j] = \sum_m \sum_n I[m,n]K[i-m, j-n].$$

The convolution operation is commutative, that is,

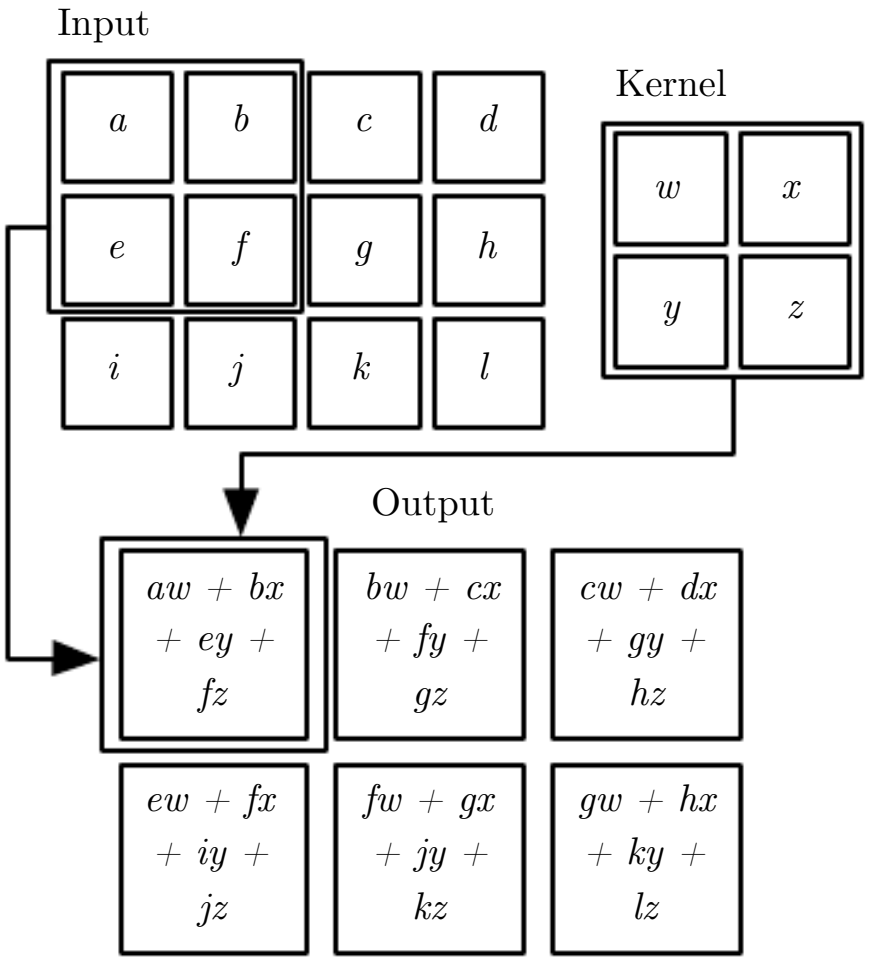
$$s[i,j] = (I * K)[i,j] = \sum_m \sum_n I[i-m, j-n]K[m,n].$$

Many neural network libraries implement the cross-correlation defined by (same as the convolution but without flipping the kernel)

$$s[i,j] = (I * K)[i,j] = \sum_m \sum_n I[i+m, j+n]K[m,n].$$

The Convolution Operation (cont'd)

An Example



2-D convolution without kernel flipping.

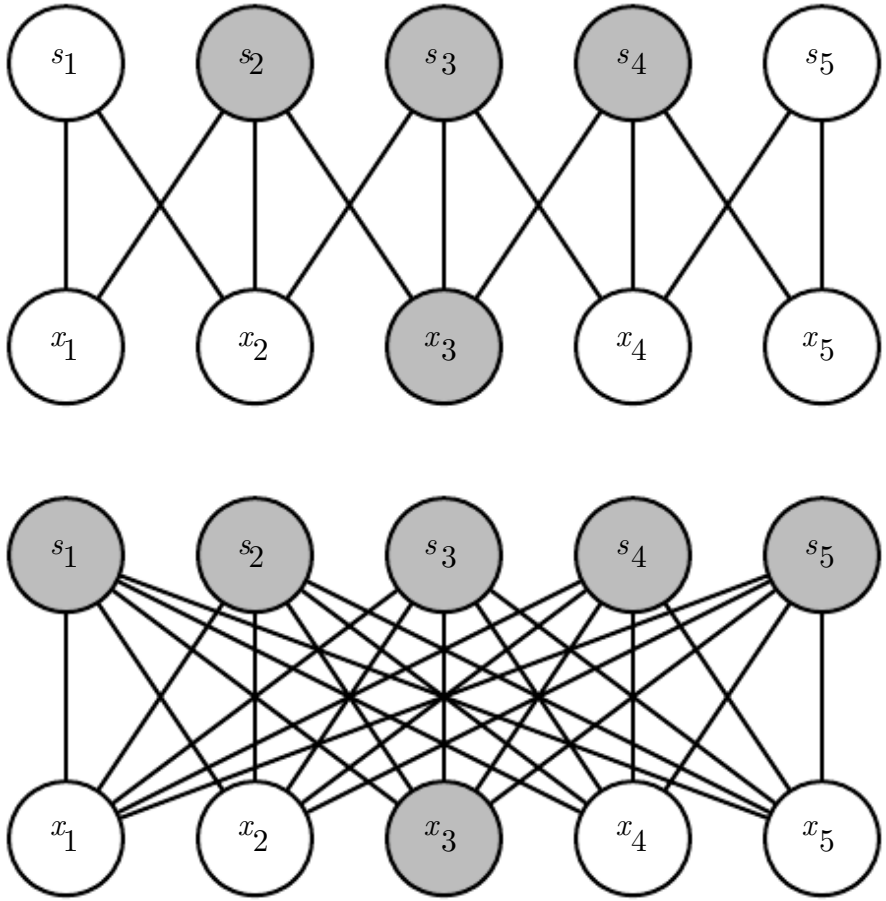
Motivation

Convolution leverages three important concepts that can help improve a machine learning system:

- sparse connectivity,
- parameter sharing,
- equivariant representations.

Moreover, convolution provides a means for working with inputs of variable size.

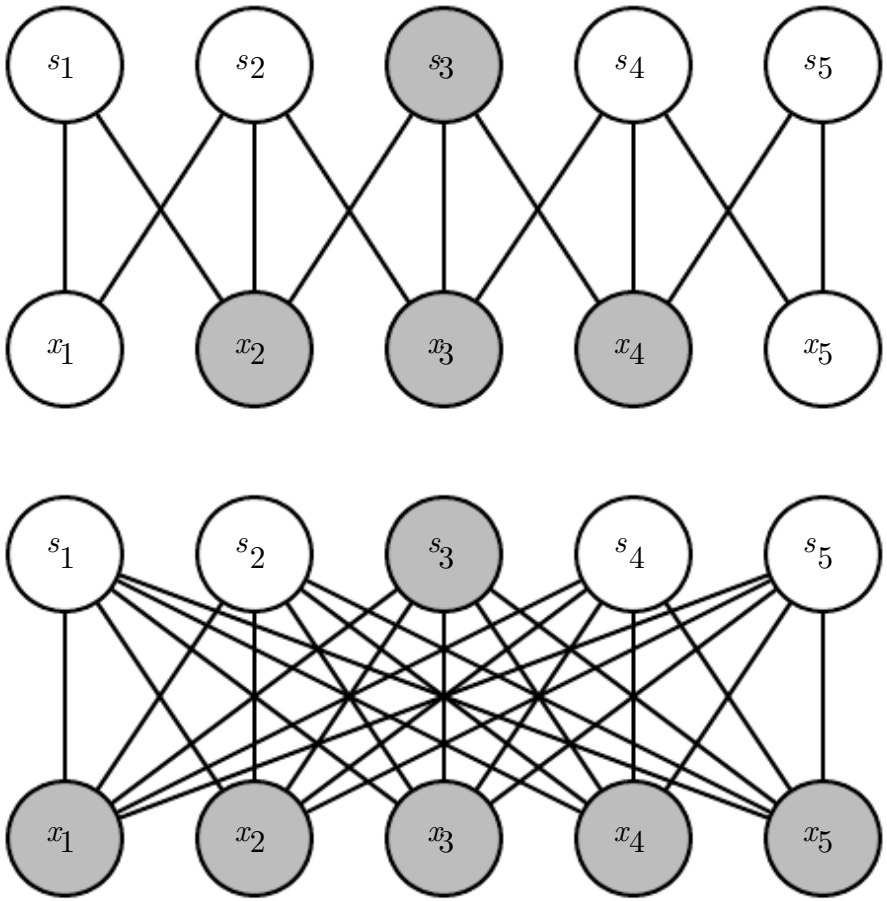
Convolutional Networks Have Sparse Connectivity



Matrix multiplication (bottom) with full connectivity. Convolution with a kernel highlights sparse connectivity.

Convolutional Networks Have Sparse Connectivity (cont'd)

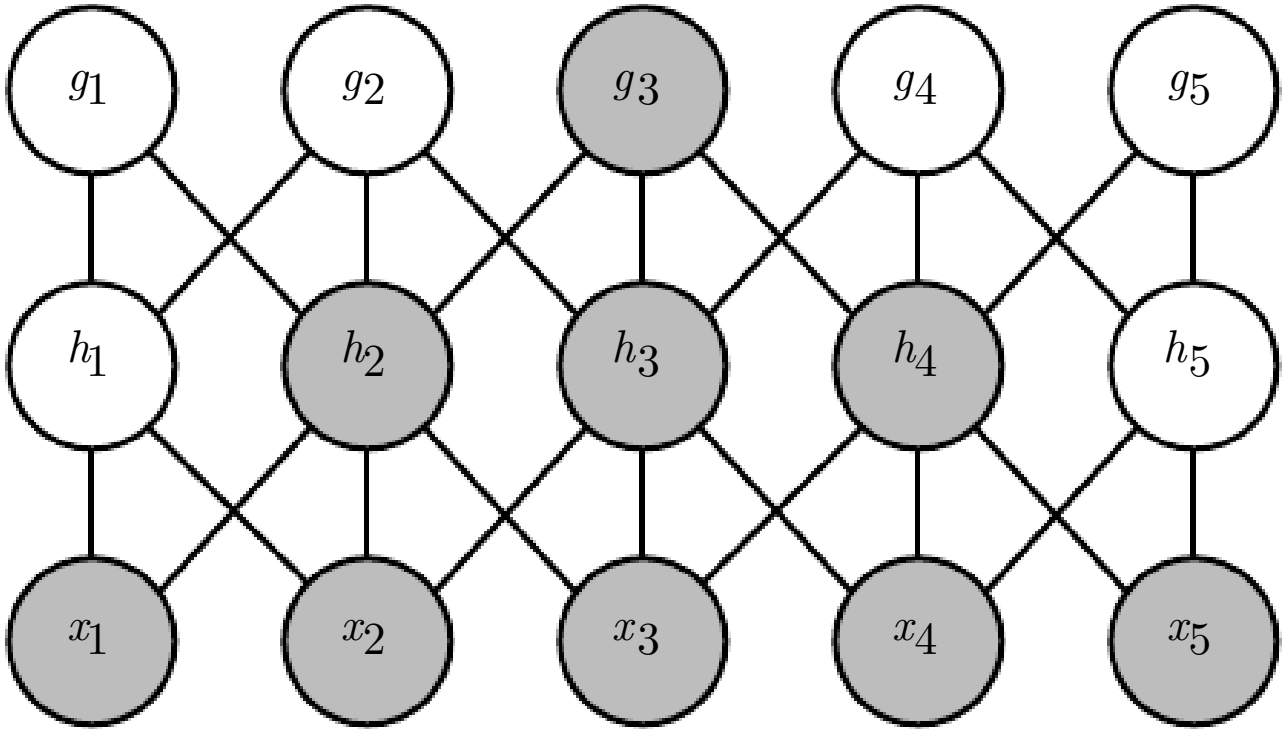
Characterization via Receptive Fields



Top: s_3 receives 3 inputs. Bottom: s_3 receives full input.

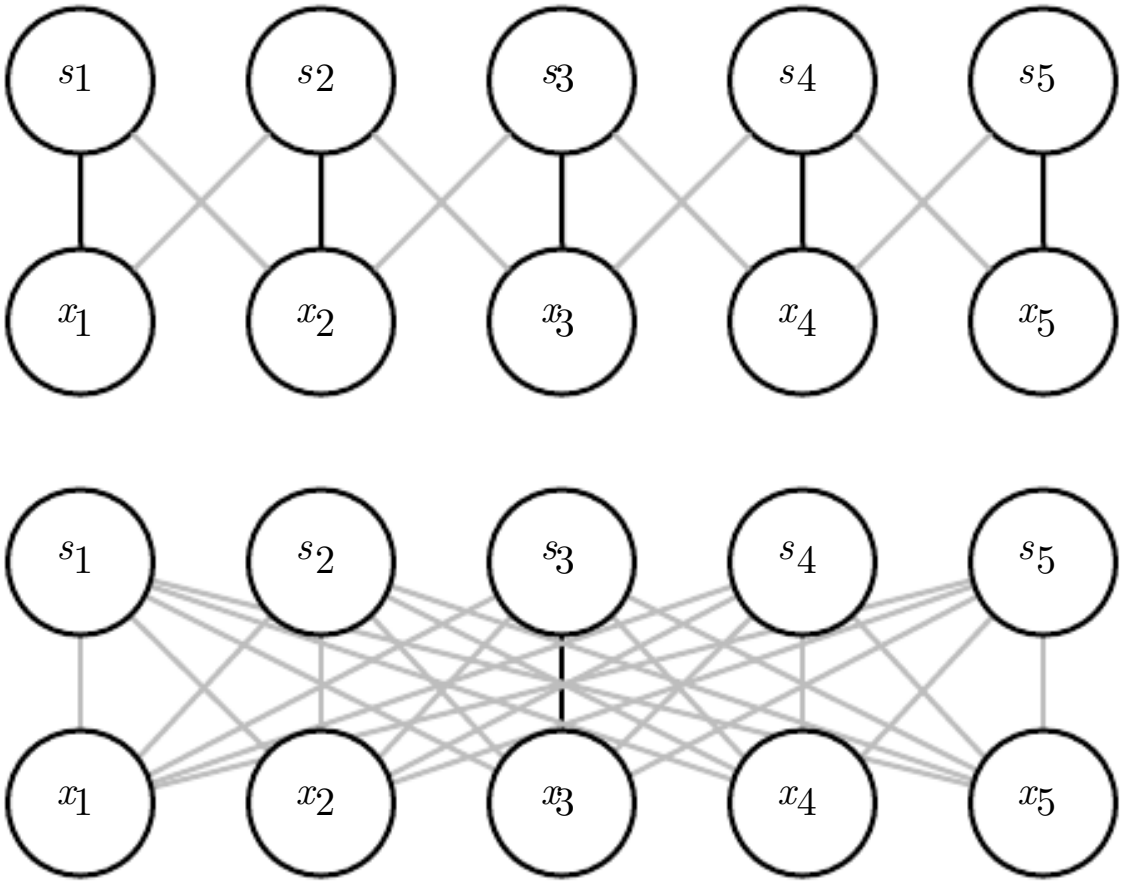
Convolutional Networks Have Sparse Connectivity (cont'd)

Complex Interactions from Simple Building Blocks



The receptive field of the units in the deeper layers of a convnet is larger than the receptive field of the units in the shallow layers.

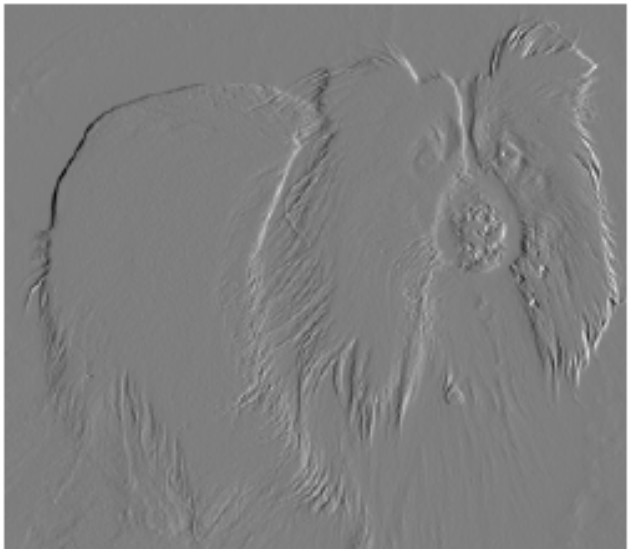
Parameter Sharing



In a convnet, each member of the kernel is used at every position of the input.

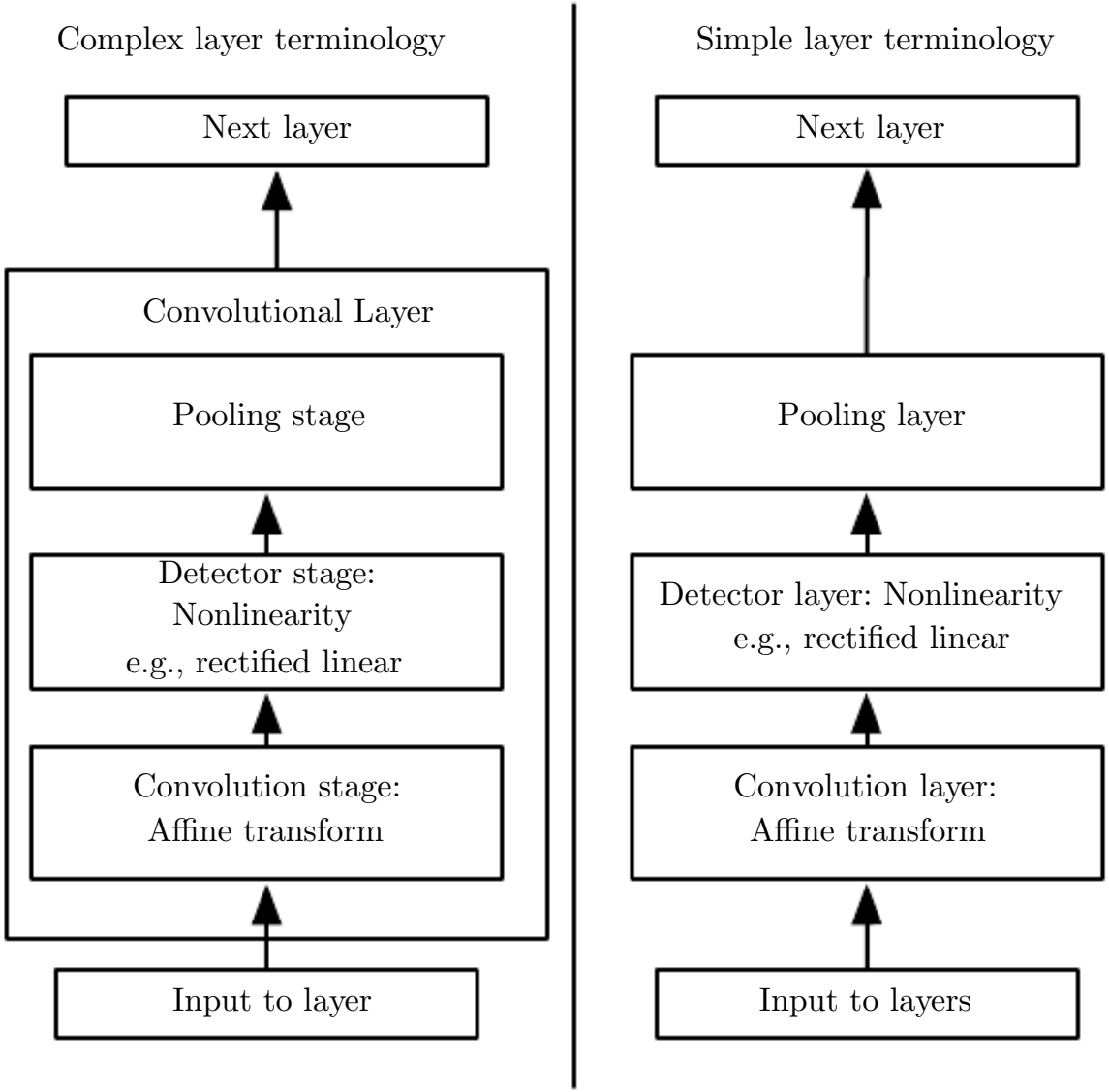
Parameter Sharing (cont'd)

Efficient Edge Detection



The input image has 280×320 pixels. The convolution kernel has 2 (two) elements and requires $319 \times 280 \times 3 = 267,960$ floating point operations (two multiplications and one addition per output pixel) to compute. Matrix multiplication of the same operation of edge detection requires $320 \times 280 \times 319 \times 280$, or over 8 billion matrix entries, making convolution 4 billion times more efficient.

The Convolutional Network Layer



Pooling

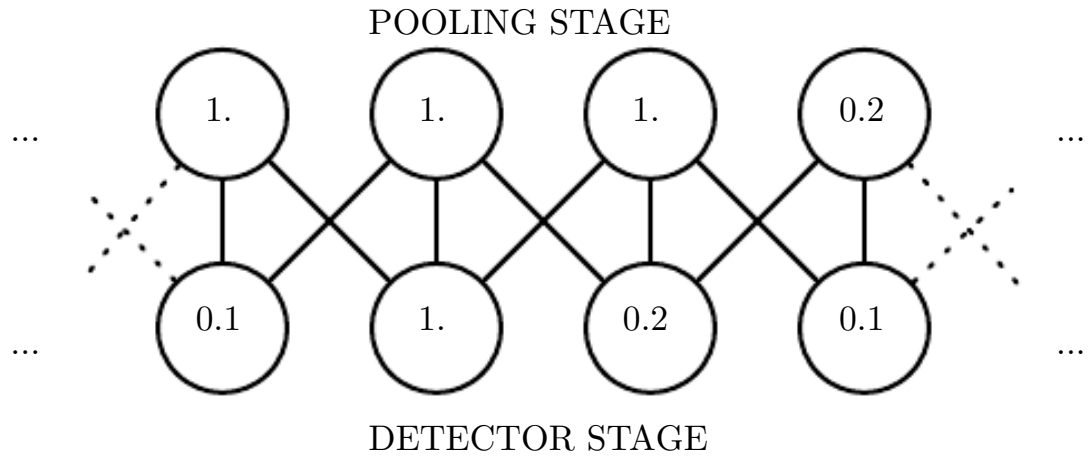
A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. For example,

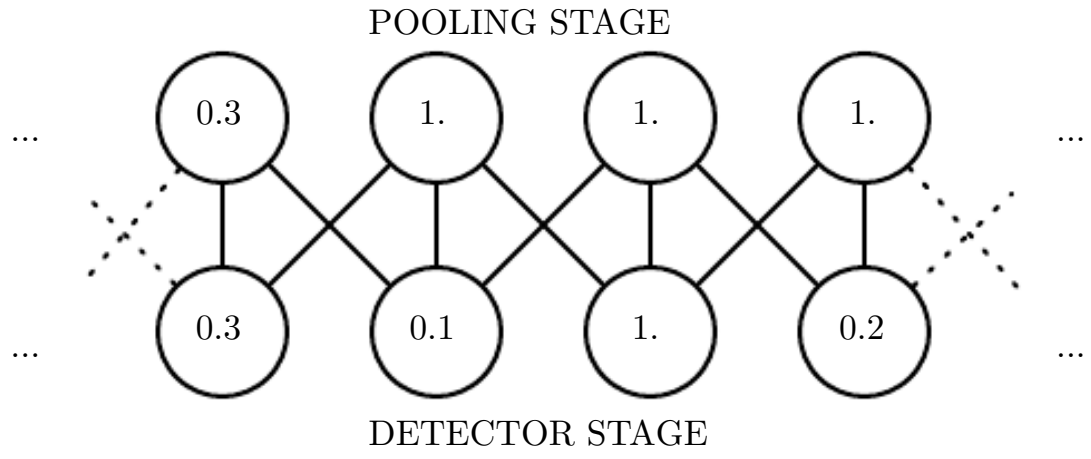
- max pooling reports the maximum output within a rectangular neighborhood;
- the average of a rectangular neighborhood;
- the L2 norm of a rectangular neighborhood;
- a weighted average based on the distance from the central pixel.

Pooling helps to make the representation become invariant to small translations of the input.
Invariance to local translation can be a very useful property if we care more about whether some feature is present than exactly where it is.

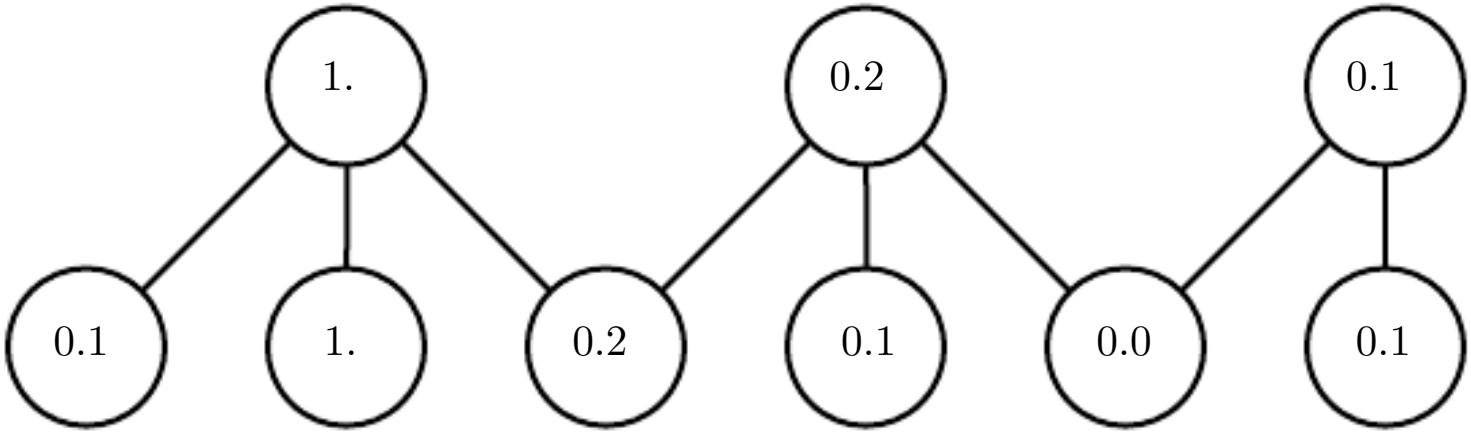
Pooling (cont'd)

Max Pooling Introduces Invariance with Respect to Small Translations





Pooling with Downsampling



max-pooling with a pool width of 3 and a stride between pools of 2 reduces the representation size by a factor of 2. The computational and statistical burden on the next layer is considerably reduced.