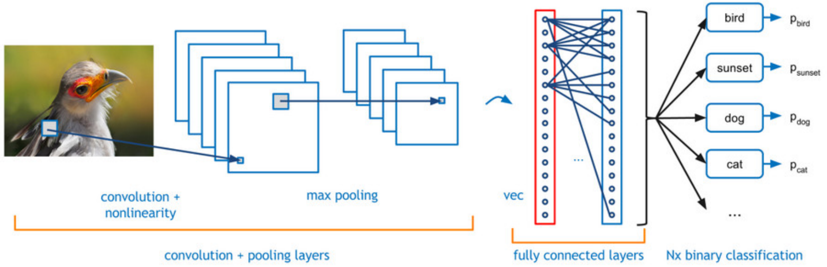
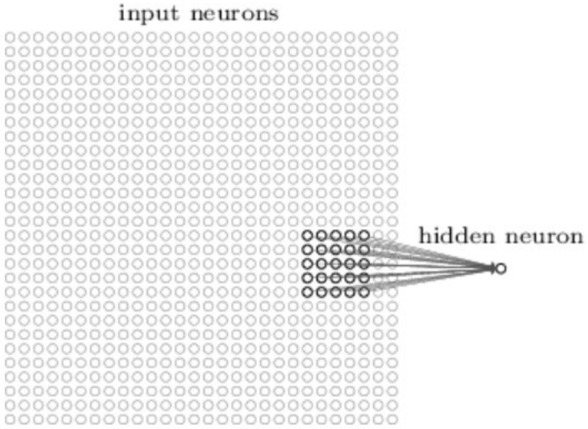
Convolutional network



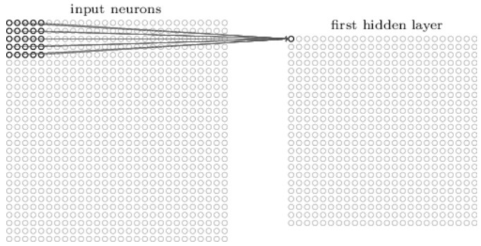
Convolutional layers

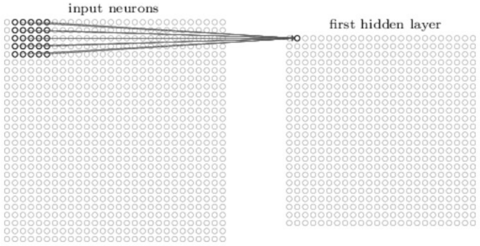


Every neuron is connected with a (typically small) *receptive field* of neurons in the lower layer.

Neuron is "standard": Computes a weighted sum of its inputs, applies an activation function.

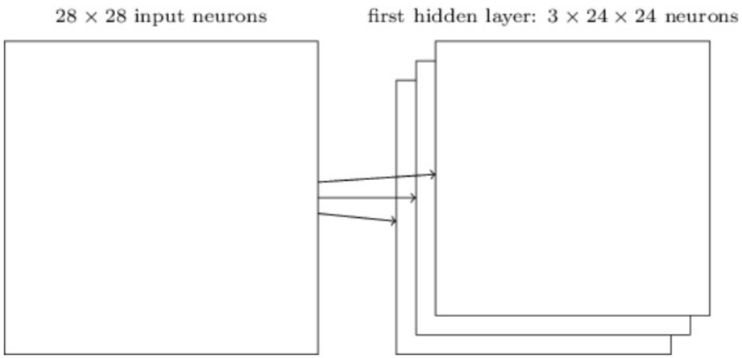
Convolutional layers





Neurons grouped into *feature maps* sharing weights.

Convolutional layers

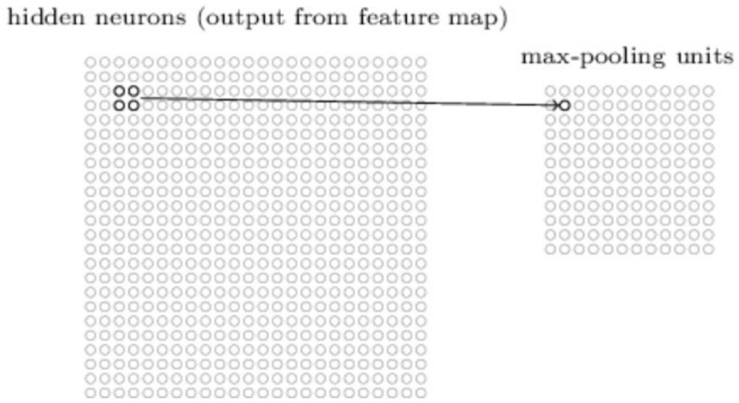


28×28 input neurons
first hidden layer: $3 \times 24 \times 24$ neurons

Each feature map represents a property of the input that is supposed to be spatially invariant.

Typically, we consider several feature maps in a single layer.

Pooling layers



Neurons in the pooling layer compute simple functions of their receptive fields (the fields are typically disjoint):

- **Max-pooling** : maximum of inputs
- **L2-pooling** : square root of the sum of squres
- **Average-pooling** : mean
- . . .

Convolutional networks – architecture

Neurons organized in layers, $L_0, L_1, \ldots, L_n$, connections (typically) only from L_m to L_{m+1}.

Several types of layers:

- **input** layer L_0
- **dense** layer L_m: Each neuron of L_m connected with each neuron of L_{m-1}.
- **convolutional & pooling** layer L_m: Contains two sub-layers:
  - **convolutional layer**: Neurons organized into disjoint **feature maps**, all neurons of a given feature map *share weights* (but have different inputs)
  - **pooling layer**: Each (convolutional) feature map F has a corresponding **pooling map** P. Neurons of P
    - have inputs only from F (typically few of them),
    - compute a simple aggregate function (such as max),
    - have *disjoint inputs*.

Convolutional networks – architecture

- Denote
  - X a set of *input* neurons
  - Y a set of *output* neurons
  - Z a set of *all* neurons ($X, Y \subseteq Z$)
- individual neurons denoted by indices i, j etc.
  - ξ_j is the inner potential of the neuron j *after the computation stops*
  - y_j is the output of the neuron j *after the computation stops*

  (define $y_0 = 1$ is the value of the formal unit input)
- w_{ji} is the weight of the connection **from** i **to** j

  (in particular, w_{j0} is the weight of the connection from the formal unit input, i.e. $w_{j0} = -b_j$ where b_j is the bias of the neuron j)
- $j_{\leftarrow}$ is a set of all i such that j is adjacent from i
  (i.e. there is an arc **to** j from i)
- $j^{\rightarrow}$ is a set of all i such that j is adjacent to i
  (i.e. there is an arc **from** j to i)
- j_{share} is a set of neurons sharing weights with j
  i.e. neurons that belong to the same feature map as j

Convolutional networks – activity

- neurons of dense and convolutional layers:
  - inner potential of neuron j:

$$\xi_j = \sum_{i \in j_\leftarrow} w_{ji} y_i$$

  - activation function σ_j for neuron j (arbitrary differentiable):

$$y_j = \sigma_j(\xi_j)$$

- Neurons of pooling layers: Apply the "pooling" function:
  - max-pooling:

$$y_j = \max_{i \in j_\leftarrow} y_i$$

  - avg-pooling:

$$y_j = \frac{\sum_{i \in j_\leftarrow} y_i}{|j_\leftarrow|}$$

A convolutional network is evaluated layer-wise (as MLP), for each $j \in Y$ we have that $y_j(\vec{w}, \vec{x})$ is the value of the output neuron j after evaluating the network with weights $\vec{w}$ and input $\vec{x}$.

Convolutional networks – learning

Learning:

- Given a **training set** $\mathcal{T}$ of the form

  $$\left\{ \left(\vec{x}_k, \vec{d}_k\right) \;\middle|\; k = 1, \ldots, p \right\}$$

  Here, every $\vec{x}_k \in \mathbb{R}^{|X|}$ is an *input vector* end every $\vec{d}_k \in \mathbb{R}^{|Y|}$ is the desired network output. For every $j \in Y$, denote by d_{kj} the desired output of the neuron j for a given network input $\vec{x}_k$ (the vector $\vec{d}_k$ can be written as $\left(d_{kj}\right)_{j \in Y}$).

- **Error function – mean square error (for example):**

  $$E(\vec{w}) = \frac{1}{p} \sum_{k=1}^{p} E_k(\vec{w})$$

  where

  $$E_k(\vec{w}) = \frac{1}{2} \sum_{j \in Y} \left(y_j(\vec{w}, \vec{x}_k) - d_{kj}\right)^2$$

Convolutional networks – SGD

The algorithm computes a sequence of weight vectors $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$.

- weights in $\vec{w}^{(0)}$ are randomly initialized to values close to 0
- in the step $t+1$ (here $t = 0, 1, 2\ldots$), weights $\vec{w}^{(t+1)}$ are computed as follows:
  - Choose (randomly) a set of training examples $T \subseteq \{1, \ldots, p\}$
  - Compute

    $$\vec{w}^{(t+1)} = \vec{w}^{(t)} + \Delta\vec{w}^{(t)}$$

    where

    $$\Delta\vec{w}^{(t)} = -\varepsilon(t) \cdot \frac{1}{|T|} \sum_{k \in T} \nabla E_k(\vec{w}^{(t)})$$

Here T is a *minibatch* (of a fixed size),

- $0 < \varepsilon(t) \leq 1$ is a *learning rate* in step $t+1$
- $\nabla E_k(\vec{w}^{(t)})$ is the gradient of the error of the example k

Note that the random choice of the minibatch is typically implemented by randomly shuffling all data and then choosing minibatches sequentially. **Epoch** consists of one round through all data.

Backprop

Recall that $\nabla E_k(\vec{w}^{(t)})$ is a vector of all partial derivatives of the form $\frac{\partial E_k}{\partial w_{ji}}$.

How to compute $\frac{\partial E_k}{\partial w_{ji}}$?

First, switch from derivatives w.r.t. w_{ji} to derivatives w.r.t. y_j:

- Recall that for every w_{ji} where j is in a dense layer, i.e. does not share weights:

$$\frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j} \cdot \sigma_j'(\xi_j) \cdot y_i$$

- Now for every w_{ji} where j is in a convolutional layer, that is shares w_{ji} with neurons of j_{share}:

$$\frac{\partial E_k}{\partial w_{ji}} = \sum_{r \in j_{share}} \frac{\partial E_k}{\partial y_r} \cdot \sigma_r'(\xi_r) \cdot y_r$$

- Neurons of pooling layers do not have weights.

Backprop

Now compute derivatives w.r.t. y_j:

- for every $j \in Y$:

  $$\frac{\partial E_k}{\partial y_j} = y_j - d_{kj}$$

  This holds for the mean-square error, for other error functions the derivative w.r.t. outputs will be different.

- for every $j \in Z \smallsetminus Y$ such that $j^{\rightarrow}$ is either a dense layer, or a convolutional layer:

  $$\frac{\partial E_k}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \sigma_r'(\xi_r) \cdot w_{rj}$$

- for every $j \in Z \smallsetminus Y$ such that $j^{\rightarrow}$ is max-pooling: Then $j^{\rightarrow} = \{i\}$ for a single "max" neuron and we have

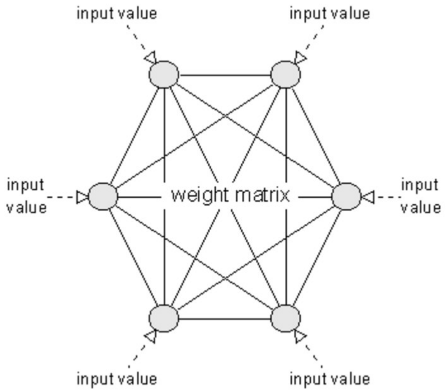
  $$\frac{\partial E_k}{\partial y_j} = \begin{cases} \frac{\partial E_k}{\partial y_i} & \text{if } j = arg\ max_{r \in i_{\leftarrow}} y_r \\ 0 & \text{otherwise} \end{cases}$$

I.e. gradient can be propagated from the output layer downwards as in MLP.

Convolutional networks – conclusions

- Conv. nets. are nowadays the most used networks in image processing (and also in other areas where input has some local, "spatially" invariant properties)
- Typically trained using backpropagation.
- Due to the weight sharing allow (very) deep architectures.
- Typically extended with more adjustments and tricks in their topologies.

Recurrent networks – Hopfield network



Auto-associative network: Given an input, the network outputs a training example (encoded in its weights) "similar" to the given input.

Hopfield network

Architecture:

- complete topology, i.e. output of each neuron is input to all neurons
- all neurons are both input and output
- denote by $\xi_1, \ldots, \xi_n$ inner potentials and by $y_1, \ldots, y_n$ outputs (states) of individual neurons
- denote by w_{ji} the weight of connection from a neuron $i \in \{1, \ldots, n\}$ to a neuron $j \in \{1, \ldots, n\}$
- assume $w_{jj} = 0$ for every $j = 1, \ldots, n$
- **For now:** no neuron has a bias

Hopfield network

Learning: Training set

$$\mathcal{T} = \{\vec{x}_k \mid \vec{x}_k = (x_{k1}, \ldots, x_{kn}) \in \{-1, 1\}^n, k = 1, \ldots, p\}$$

The goal is to "store" the training examples of $\mathcal{T}$ so that the network is able to *associate* similar examples.

Hebb's learning rule: If the inputs to a system cause the same pattern of activity to occur repeatedly, the set of active elements constituting that pattern will become increasingly strongly interassociated. That is, each element will tend to turn on every other element and (with negative weights) to turn off the elements that do not form part of the pattern. To put it another way, the pattern as a whole will become "auto-associated".

Mathematically speaking:

$$w_{ji} = \sum_{k=1}^{p} x_{kj} x_{ki} \qquad 1 \leq j \neq i \leq n$$

Intuition: "Neurons that fire together, wire together".

Hopfield network

Learning: Training set

$$\mathcal{T} = \{\vec{x}_k \mid \vec{x}_k = (x_{k1}, \ldots, x_{kn}) \in \{-1, 1\}^n, k = 1, \ldots, p\}$$

Hebb's rule:

$$w_{ji} = \sum_{k=1}^{p} x_{kj} x_{ki} \qquad 1 \leq j \neq i \leq n$$

Note that $w_{ji} = w_{ij}$, i.e. the weight matrix is symmetric.

Learning can be seen as poll about equality of inputs:

- If $x_{kj} = x_{ki}$, then the training example votes for "i equals j" by adding one to w_{ji}.
- If $x_{kj} \neq x_{ki}$, then the training example votes for "i does not equal j" by subtracting one from w_{ji}.

Hopfield network

Activity: Initially, neurons set to the network input $\vec{x} = (x_1, \ldots, x_n)$, thus $y_j^{(0)} = x_j$ for every $j = 1, \ldots, n$.

Cyclically update states of neurons, i.e. in step $t + 1$ compute the value of a neuron j such that $j = (t \mod p) + 1$, as follows:

Compute the inner potential:

$$\xi_j^{(t)} = \sum_{i=1}^{n} w_{ji} y_i^{(t)}$$

then

$$y_j^{(t+1)} = \begin{cases} 1 & \xi_j^{(t)} > 0 \\ y_j^{(t)} & \xi_j^{(t)} = 0 \\ -1 & \xi_j^{(t)} < 0 \end{cases}$$

Hopfield network – activity

The computation stops in a step t^* if the network is for the first time in a *stable* state, i.e.

$$y_j^{(t^*+n)} = y_j^{(t^*)} \qquad (j = 1, \ldots, n)$$

Theorem

Assuming symmetric weights, computation of a Hopfiled network always stops for every input.

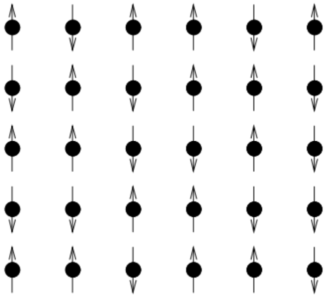
This implies that a given Hopfiled network computes a function from $\{-1, 1\}^n$ to $\{-1, 1\}^n$ (determined by its weights).

Denote by $\vec{y}(W, \vec{x}) = \left(y_1^{(t^*)}, \ldots, y_n^{(t^*)}\right)$ the value of the network for a given input $\vec{x}$ and a weight matrix W.
Denote by $y_j(W, \vec{x}) = y_j^{(t^*)}$ the component of the value of the network corresponding to the neuron j.

If W is clear from the context, we write only $y(\vec{x})$ a $y_j(\vec{x})$.

Ising model – an analogy

Simple models of magnetic materials resemble Hopfield network.



- atomic magnets organized into square-lattice
- each magnet may have only one of two possible orientations (in the Hopfield network $+1$ a -1)
- orientation of each magnet is influenced by an external magnetic field (input of the network) as well as orientation of the other magnets
- weights in the Hopfiled net model determine interaction among magnets

Energy function

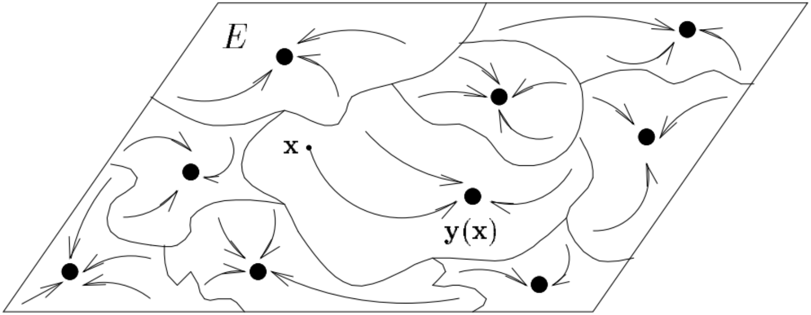
Energy function E assigns to every state $\vec{y} \in \{-1, 1\}^n$ a (potential) energy:

$$E(\vec{y}) = -\frac{1}{2} \sum_{j=1}^{n} \sum_{i=1}^{n} w_{ji} y_j y_i$$

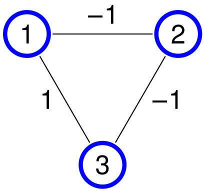
- states with low energy are stable (few neurons "want to" change their states), states with high energy are not stable
- i.e. large (positive) $w_{ji} y_j y_i$ is stable and small (negative) $w_{ji} y_j y_i$ is not stable

The energy does not increase during computation: $E(\vec{y}^{(t)}) \geq E(\vec{y}^{(t+1)})$, stable states $\vec{y}^{(t^*)}$ correspond to local minima of E.

Energy landscape



Hopfield – example



y_1	y_2	y_3	E
1	1	1	1
1	1	−1	1
1	−1	1	−3
1	−1	−1	1
−1	1	1	1
−1	1	−1	−3
−1	−1	1	1
−1	−1	−1	1

- Hopfield network with three neurons
- trained on a single training example $(1, -1, 1)$ using Hebb's rule

  (note that $(-1, 1, -1)$ has also been "stored" into the network)

Hopfield network – convergence

Observe that

- the energy does not increase during computation: $E(\vec{y}^{(t)}) \geq E(\vec{y}^{(t+1)})$
- if the state is updated in a step $t+1$, then $E(\vec{y}^{(t)}) > E(\vec{y}^{(t+1)})$
- there are only finitely many states, and thus, eventually, a local minimum of E is reached.

This proves that computation of a Hopfield network always stops.

Hopfield network – phantoms

The energy function E may have local minima that do not correspond to training examples (so called phantoms).

Phantoms can be "unlearned" e.g. using the following rule: Given a phantom $(x_1, \ldots, x_n) \in \{-1, 1\}^n$ and weights w_{ji}, then new weights w'_{ji} are computed by

$$w'_{ji} = w_{ji} - x_i x_j$$

(i.e. similar to Hebb's rule but with the opposite sign)

Reproduction – statistical analysis

Capacity of Hopfield network is defined as the ratio p/n of number of training examples the net is able to learn over the number of neurons.

Assume that training examples are chosen randomly: each component of $\vec{x}_k$ is set to 1 with probability $1/2$ and to -1 with probability $1/2$.

Consider a configuration W obtained by learning using the Hebb's rule.

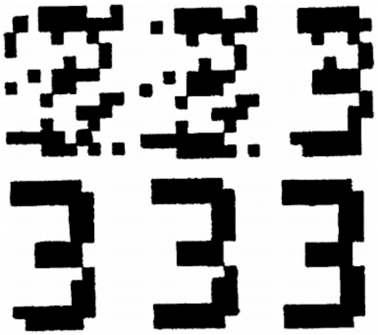
Denote

$$\beta = \mathbf{P}\left[\vec{x}_k = \vec{y}(W, \vec{x}_k) \text{ pro } k = 1, \ldots, p\right]$$

Then for $n \to \infty$ and $p \leq n/(4 \log n)$ we have $\beta \to 1$.

I.e. the maximum number of examples that can be effectively stored in Hopfield net is proportional to $n/(4 \log n)$.

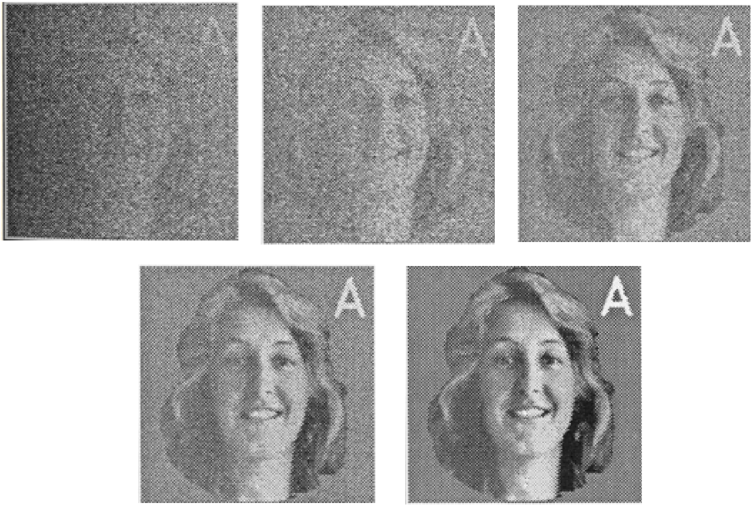
Hopfield network – example



- figures 12×10

  (120 neurons, -1 is white and 1 is black)
- learned 8 figures
- input generated with 25% noise
- image shows the activity of the Hopfield network

Hopfield network – example



Hopfield network – example