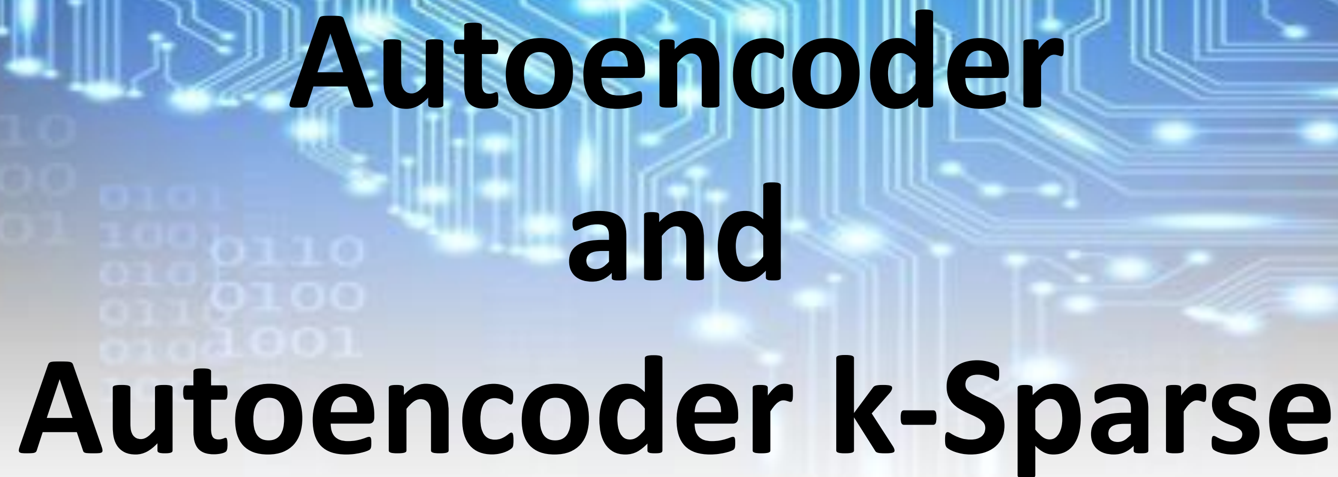


Autoencoder
and
Autoencoder k-Sparse

with **Caffe** libraries

http://caffe.berkeleyvision.org/

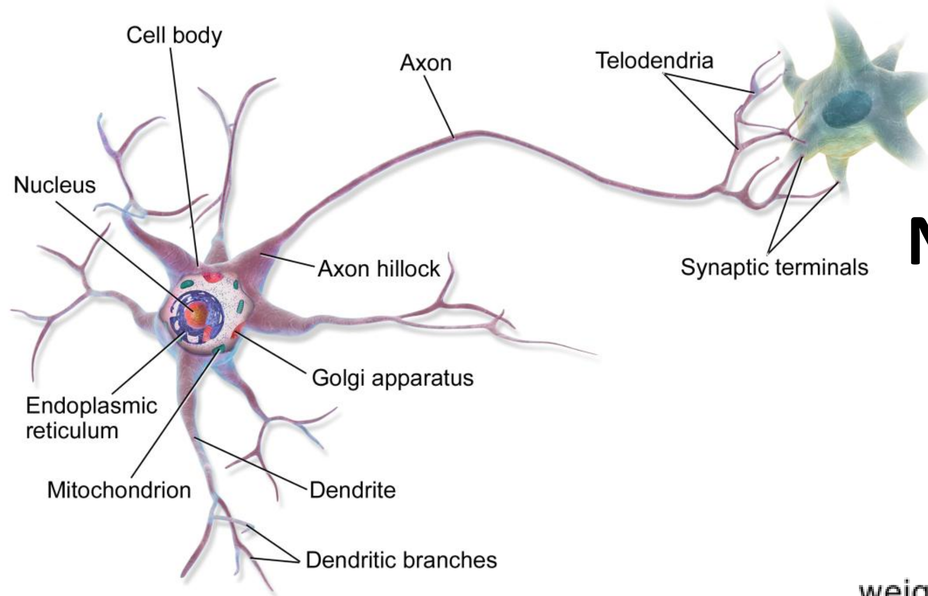
Guido Borghi

Summary

1) **Neural Networks**
2) **Caffe Libraries**
   - Anatomy (Blobs, Layers, Solver...)
   - CNN: *LeNet*
   - CNN: *CaffeNet*
3) **Autoencoder and Autoencoder k-Sparse**
   - *Olivetti Faces* dataset
   - *Faces in the Wild* dataset

Neural Network

Biological basics

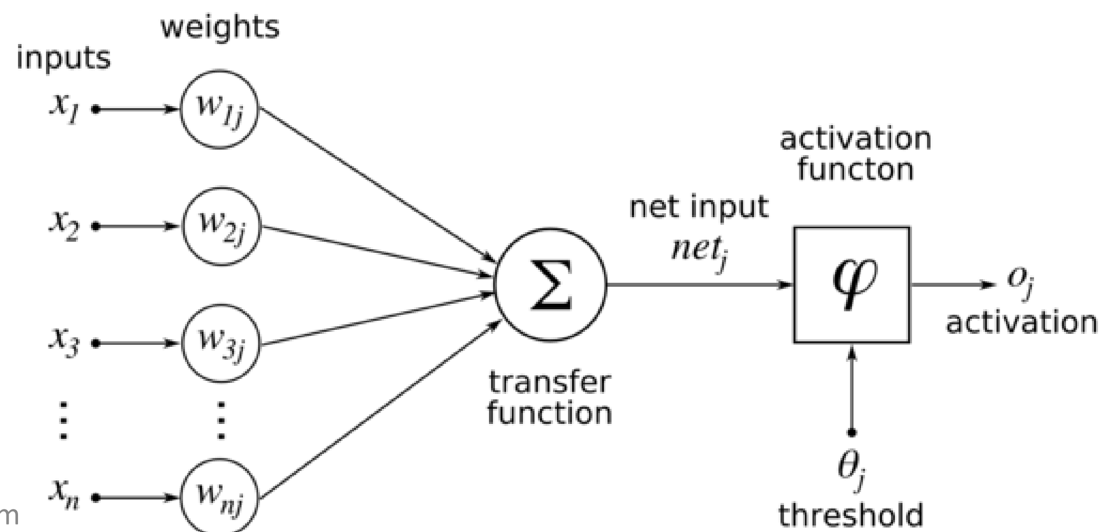


Natural Neuron

- Soma (cell body)
- Axon (to propagate signals)
- Dendrites (to receive signals)

First Artificial neuron

McCulloch – Pitts (1943)

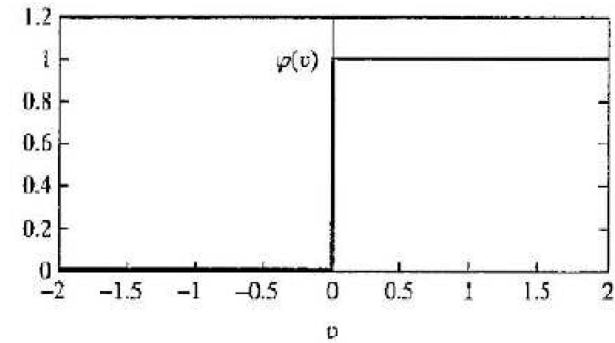


Guido Borghi - gdubrg@gmail.com

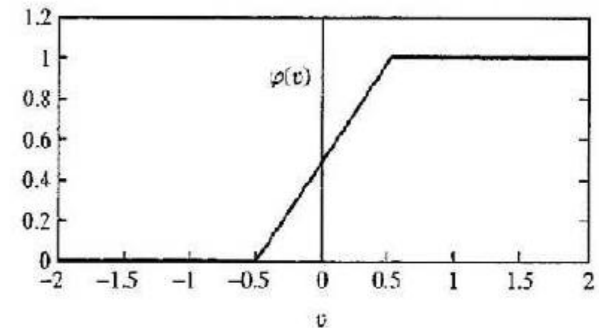
Artificial Neuron

- One **weight** for each input (w_i)
  - Excitatory channel if $\mathbf{w_i>0}$
  - Inhibitory channel if $\mathbf{w_i<0}$
- Activation Function (ϕ)
  - (a) Threshold
  - (b) Linear
  - (c) Sigmoid
  - ...
- Network output:

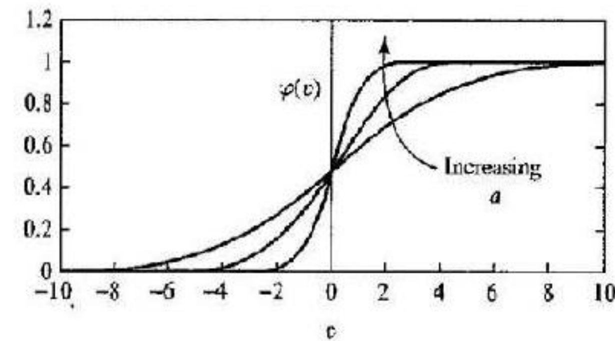
$$y_i = \Phi(A_i) = \Phi\left(\sum_{j=1}^{n} w_{j,i} y_j - \vartheta_i\right)$$



(a)



(b)



(c)

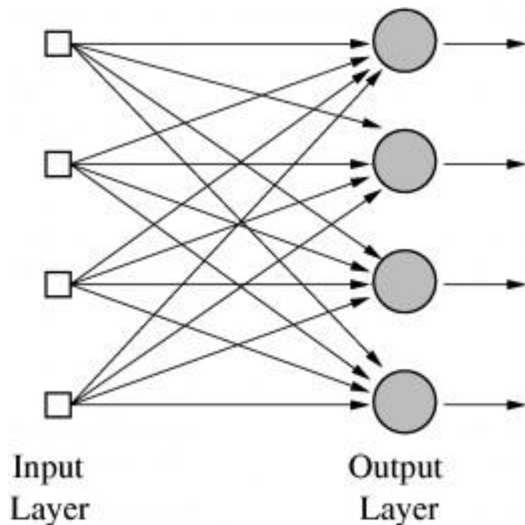
Neural Network

Topologies

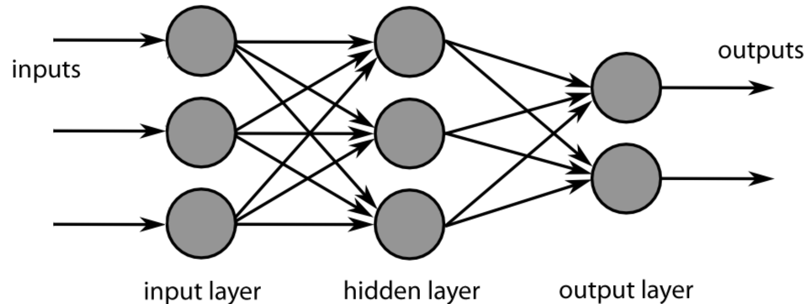
1. Single-layer Feedforward
2. Multi-layer Feedforward
3. Recurrent

Learning

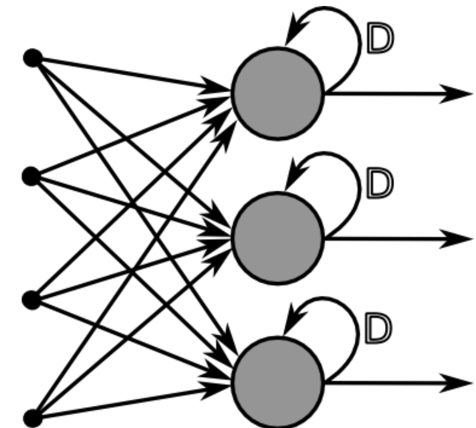
- Supervised
- Unsupervised



1



2



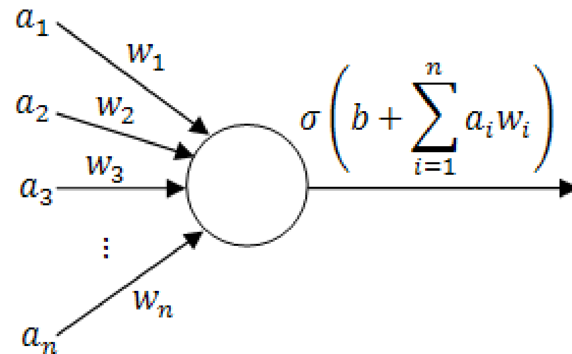
3

Guido Borghi - gdubrg@gmail.com

Single-layer Feedforward

Perceptron (Rosenblatt, 1957)

→ Linear Classification (supervisioned)



Learning: **Delta Rule** (or *Widrow-Hoff Rule)*

t = desidered output

y = network output

δ = error

x_i = neuron's output

η = **learning rate** [0 – 1]

$$\delta = t - y$$

$$\Delta w_i = \eta \delta x_i$$

$$w_i = w_i + \Delta w_i$$

Guido Borghi - gdubrg@gmail.com

Multi-layer Feedforward

Problem with Delta Rule with *hidden layers*

→Solution: **Backpropagation** (1986)

We compute an error function and we back propagate the error to the input. The aim is to minimize the error and adjusting weights.

Steps of the algorithm:

1. Feed-forward computation
2. Backpropagation to the output layer
3. Backpropagation to the hidden layer
4. Weight updates

Backpropagation (1)

We need a **differentiable** activation function: $f(net_j) = \frac{1}{1 + \exp(-net_j)}$

A possible error function: $E_k = \frac{1}{2} \sum_j (t_j - o_j)^2$

1. Compute error for **output** layer: $\delta_j = (t_j - o_j) f'(net_j) = (t_j - o_j) o_j (1 - o_j)$
2. Compute errors for **hidden** layers: $\delta_j = f'(net_j) \sum_s \delta_s w_{js} = o_j (1 - o_j) \sum_s \delta_s w_{js}$
3. Weight **updates**: $\Delta w_{ij}(n+1) = \eta \delta_j o_i + \alpha \Delta w_{ij}(n)$

An **epoch** is a measure of the number of times all of the training vectors are used once to update the weights.

- **Batch** Training: weight updates after all training set data
- **Sequential** Training: weight updates after a single example

Backpropagation (2)

How to minimize Error Function? → **Gradient Descent**

Weight update: $\Delta w_{ij} = -\eta \cdot \frac{\partial E}{\partial w_{ij}}$ (1) or $\Delta w_{ij}(t) = -\eta \cdot \frac{\partial E}{\partial w_{ij}(t)} + \alpha \cdot \Delta w_{ij}(t-1)$

α: momentum

The gradient: $\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$ with $net_j = \sum_i x_i w_{ij}$

And so we can assume $\delta_j = -\frac{\partial E}{\partial net_j}$ or $\delta_j = -\frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j}$ (2)

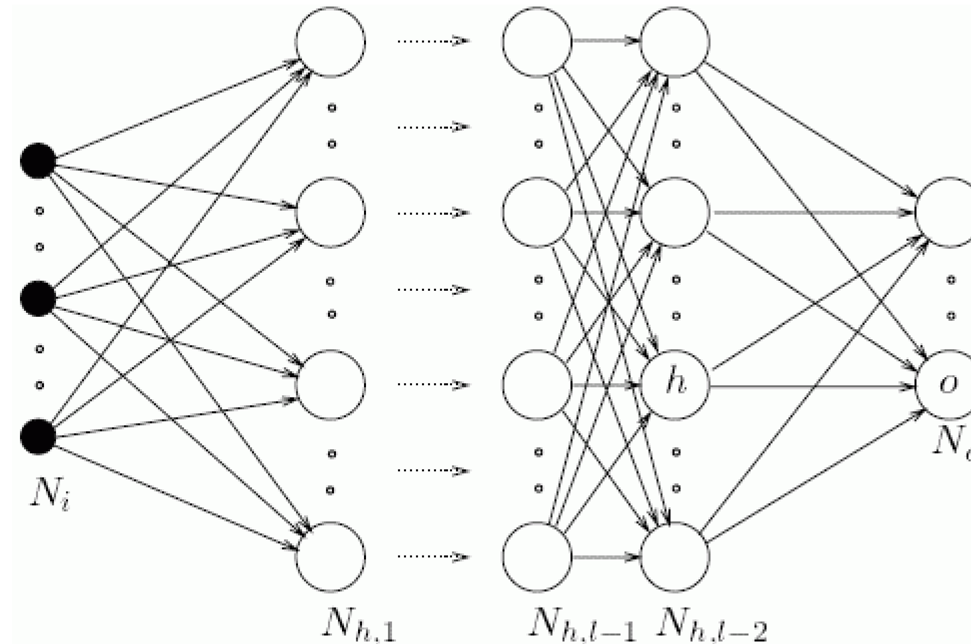
Error function is $E_k = \frac{1}{2} \sum_j (t_j - o_j)^2$ and so $\frac{\partial E}{\partial o_j} = -(t_j - o_j)$

Activation function $o_j = f(net_j)$, $\frac{\partial o_j}{\partial net_j} = f'(net_j)$. Besides $\frac{\partial net_j}{\partial w_{ij}} = x_i$

Finally we can rewrite (2) $\delta_j = (t_j - o_j) f'(net_j)$ [for output layers]

and $\delta_j = f'(net_j) \sum_s \delta_s w_{js}$ [for hidden layers] → rewrite (1)

Deep Learning



A lot of hidden layers.

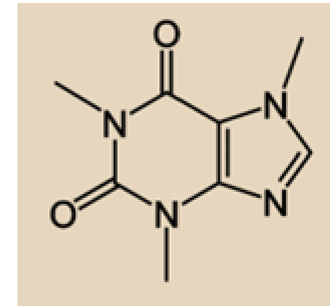
Useful to work with **images** (but not only).

Problems in supervisioned and unsupervisioned learning with backpropagation.

→ Solution: **Restricted Boltzman Machine** (Hinton, 2006)

Deep learning with Caffe

- A **deep learning framework**
- Created by *Yangqiong Jia* during his PhD at UC Berkeley
- Why use Caffe?
  - Clean Architecture
  - Readable & modifiable implementation
  - Speed (C++/CUDA architecture)
  - Community (https://github.com/BVLC/caffe)
- **Problems**
  - Not many comments in code
  - A lot of configuration file, parameters...



Caffe:
Convolutional Architecture for Fast Feature Embedding

Guido Borghi - gdubrg@gmail.com

Installation

Prerequisites:

- CUDA library version 6.5 (NVIDIA)
- BLAS (for matrix and vector computations)
- OpenCV
- Boost (portable C++ source libraries)
- Glog, gflags, protobuf, leveldb, snappy, hdf5, lmdb

Compilation:

- Modify the *Makefile.config.example* file
- `make all`
- `make test`
- `make runtest`

The anatomy of Caffe model

Blob

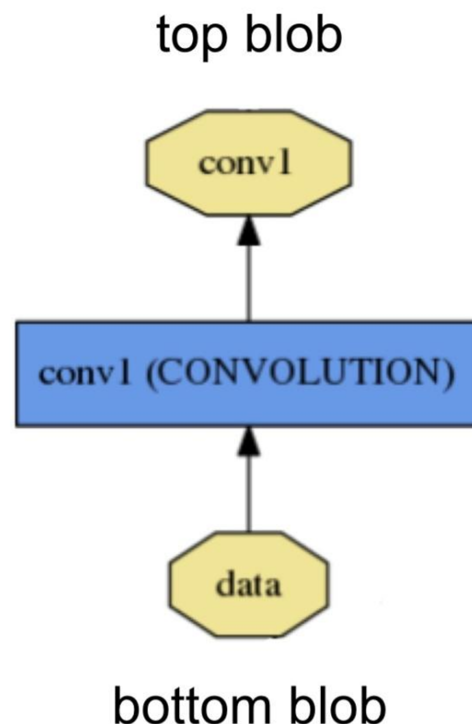
Blobs are 4-D arrays for storing and communicating information. Blobs provide a unified memory interface.

- Data:

  Number x K Channel x Height x Width

- Parameter:

  N Ouput x k Input x Height x Width



Data and derivatives flow through the net as blobs – a an array interface.

```
name: "conv1"
type: CONVOLUTION
bottom: "data"
top: "conv1"
… definition …
```

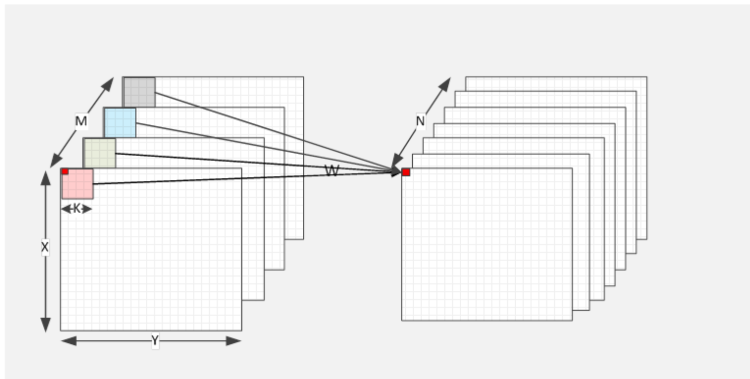
The anatomy of Caffe model

Layer

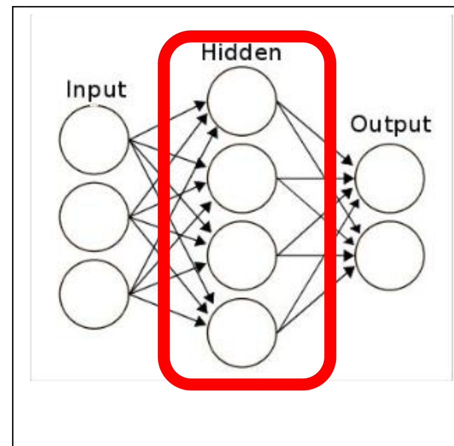
Layer is the **fundamental unit** of computation.

It is an astraction of the artificial neuron.

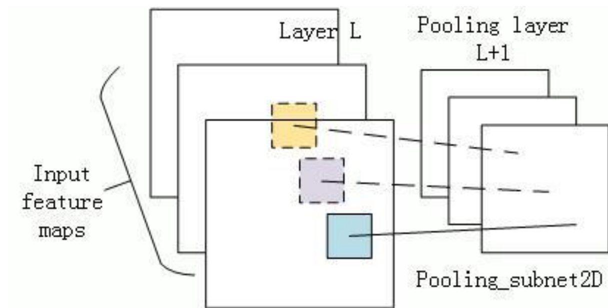
Examples of layers:



Convolutional Layer



Inner Product Layer



Pooling Layer

Data Layers

- **Database**
  - Parameters required:
    - `source`: the name of the directory containing the database
    - `batch_size`: the number of inputs to process at one time
- **HDF5 Input**
  - Parameters required:
    - `source`: the name of the file to read from
    - `batch_size`
- **Images, In-memory…**

Other layers

- **Activation Functions**:
  - Sigmoid, Hyperbolic Tangent, Absolute Value...

- **Loss Layers**

Loss drives learning by comparing an output to a target and assigning cost to minimize.

- Softmax
- Euclidean

$$\frac{1}{2N} \sum_{i=1}^{N} \| x_i^1 - x_i^2 \|_2^2$$

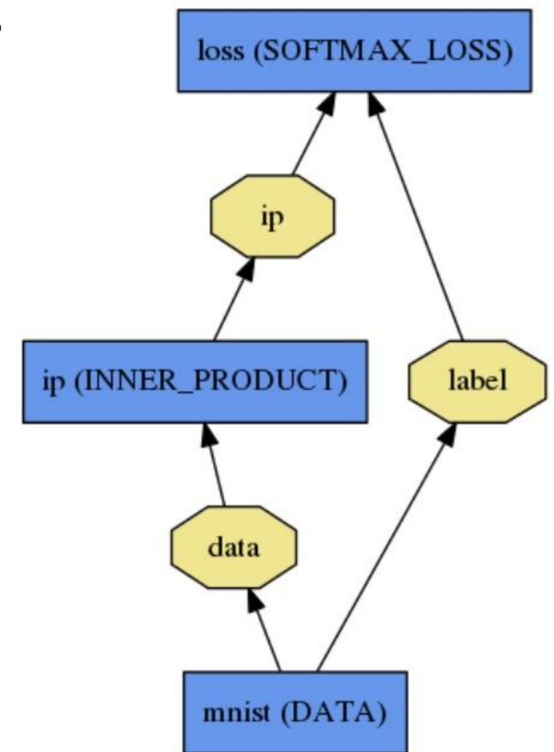
- Margin
- ...

Guido Borghi - gdubrg@gmail.com

The anatomy of Caffe model

Net

The net is a set of layers connected in a computation graph, a directed acyclic graph (DAG).



A typical net begins:

- Starts with a **data** layer
- Ends with a **loss** layer

The anatomy of Caffe model

Solver

Solver optimizes the network weights W to **minimize** the loss L(W) over the data D.

Coordinates forward / backward, weight updates, and scoring.

For the next slide:

α: learning rate, μ: momentum, V_t: previous weight update,

W_t: current weights, V_{t+1}:update value, W_{t+1}: updated weights,

$\nabla L(W)$: gradient

Solver

- **SGD** (Stochastic gradient descent)

$$V_{t+1} = \mu V_t - \alpha \nabla L(W_t)$$

$$W_{t+1} = W_t + V_{t+1}$$

- **AdaGrad** (Adaptive Gradient)

$$(W_{t+1})_i = (W_t)_i - \alpha \frac{(\nabla L(W_t))_i}{\sqrt{\sum_{t'=1}^{t} (\nabla L(W_{t'}))_i^2}}$$

- **NAG** (Nesterov's accelerated gradient)

$$V_{t+1} = \mu V_t - \alpha \nabla L(W_t + \mu V_t)$$

$$W_{t+1} = W_t + V_{t+1}$$

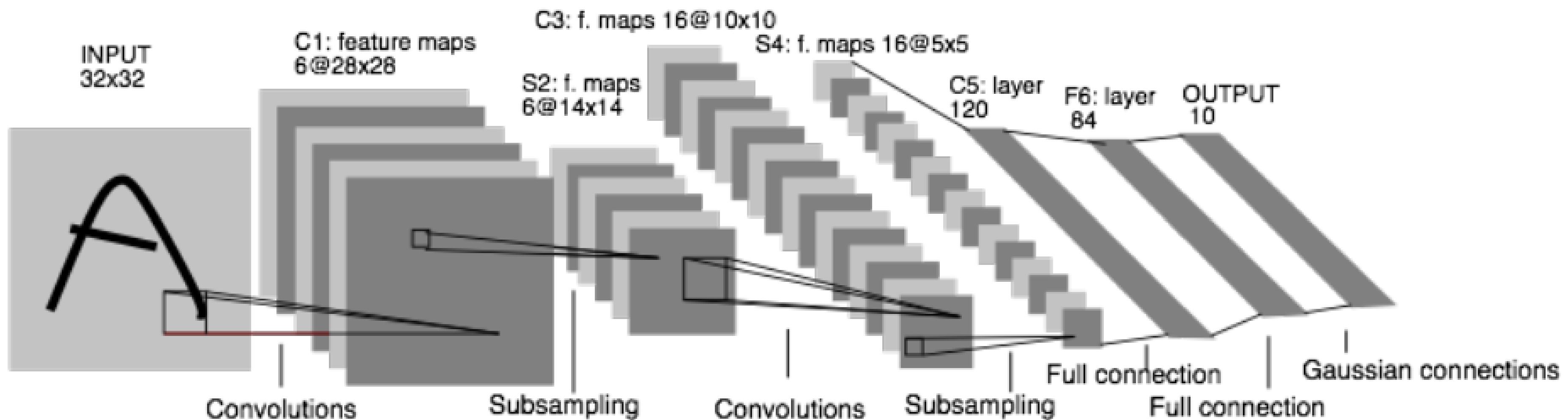
CNN (Convolutional Neural Networks)

- They are a special kind of **multi-layer** neural networks
- They are trained with a version of the **backpropagation** algorithm
- They are designed to recognize **visual patterns** directly from pixel images with:
  - extreme variability
  - robustness to distortions
  - robustness to simple geometric transformations

LeNet

- Slightly different version from the original LeNet implementation, replacing the sigmoid activations with Rectified Linear Unit (**ReLU**) activations for the neurons.
- Dataset: MNIST

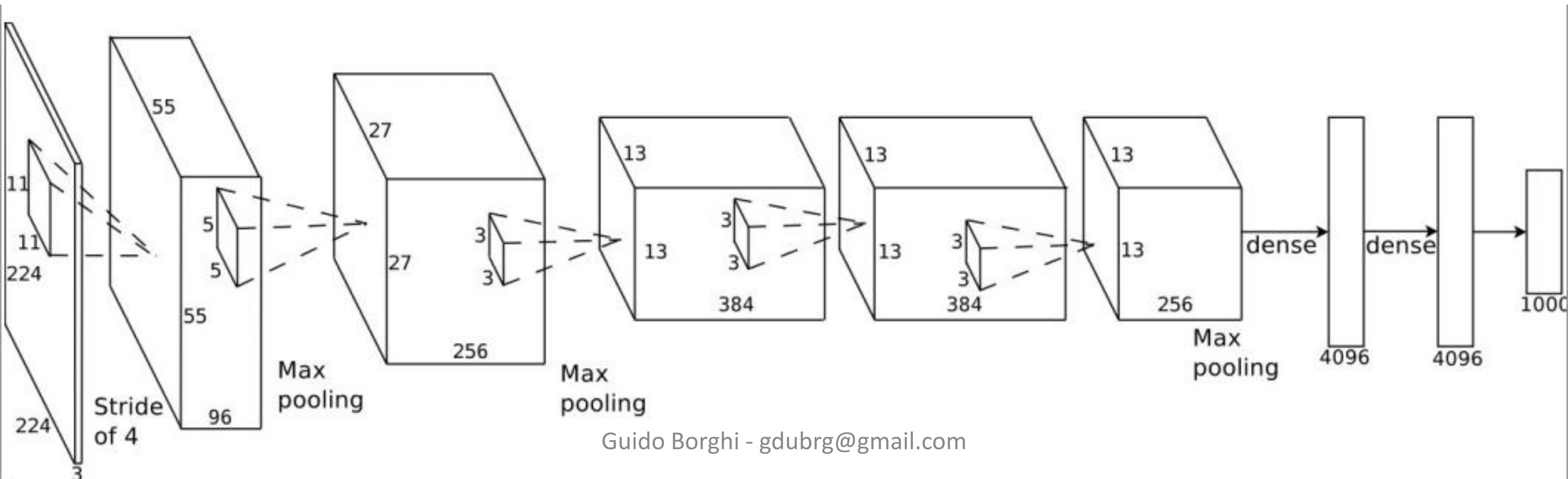
$$f(x) = \max(0, x)$$



CaffeNet

- Based on AlexNet
- Dataset: Imagenet
- **650,000** neurons, 630,000,000 connections, Final feature layer: 4096-dimensional

  → ~ **1 week** on 2 NVIDIA GPU

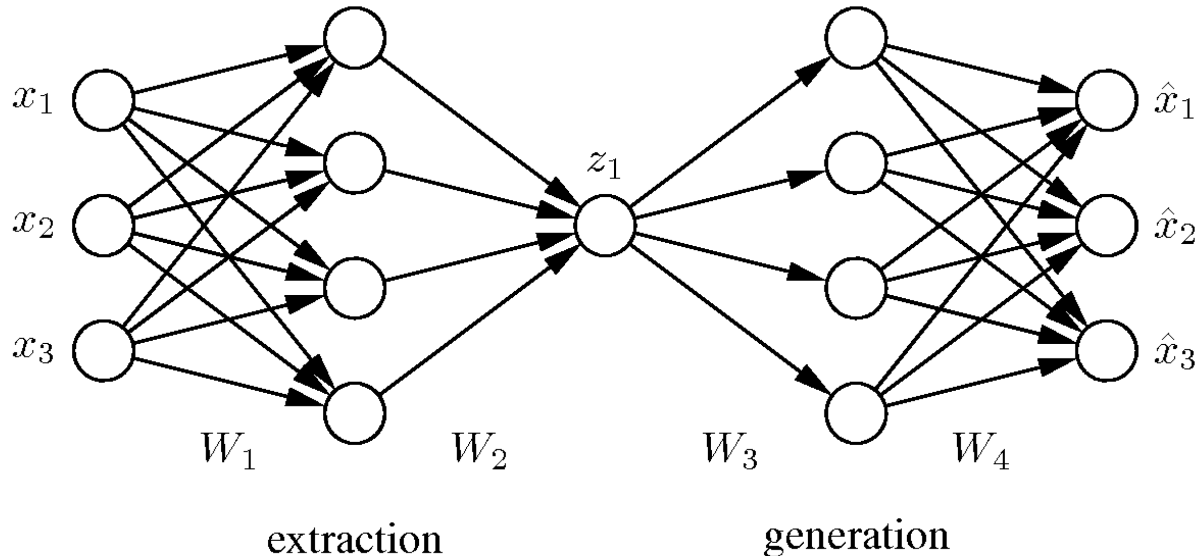


Guido Borghi - gdubrg@gmail.com

Autoencoder

- A way to learn a compressed, distributed representation for a set of data
- Dimensionality reduction
- **Architecture**:
  - Feedforward, non-recurrent neural net
  - Input layer, an output layer and one or more hidden layers connecting them
  - It is trained to *reconstruct* its own inputs

Autoencoder



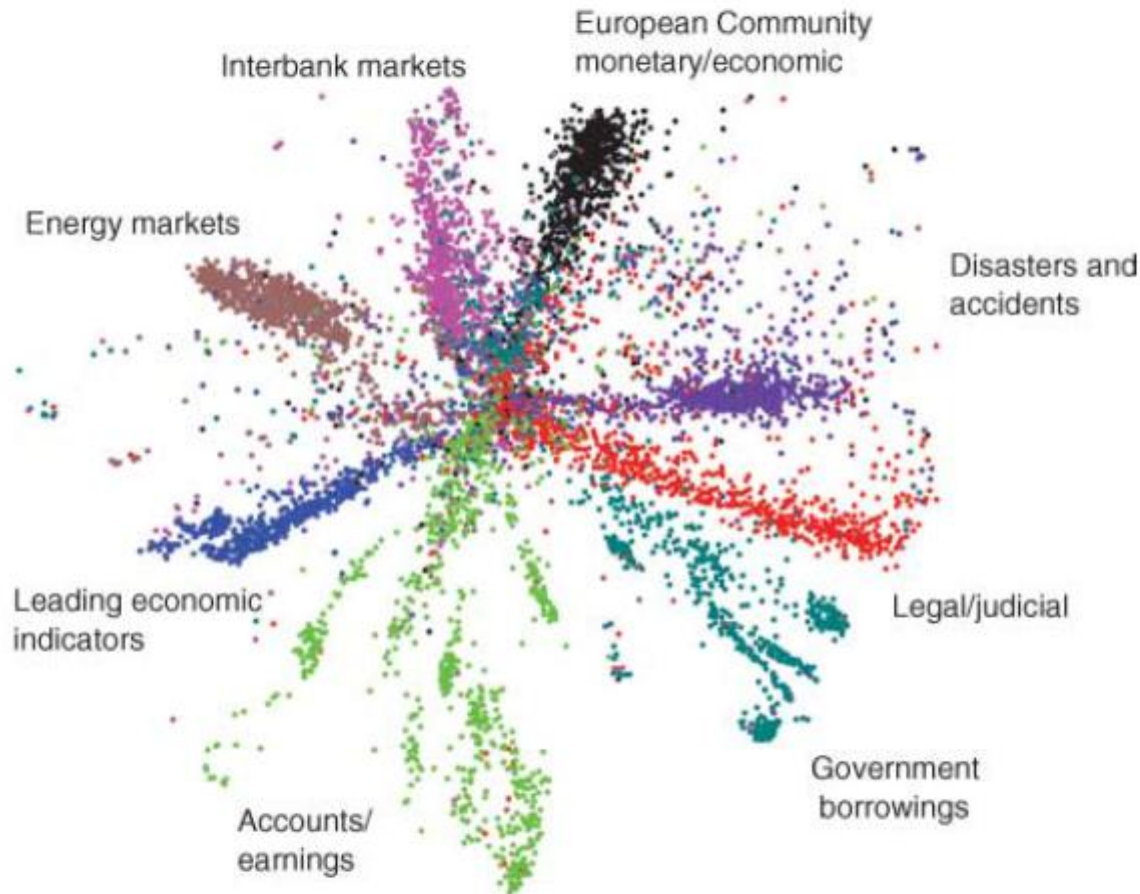
- # inputs = # outputs
- A possible Error Function: $\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2.$
- **Pre-train** to using initial weights that approximate the final solution

Autoencoder k-Sparse

- The k-sparse autoencoder is based on an autoencoder with linear activation functions
- After reconstructing the input from all of the hidden units, **we identify the k largest hidden units and set the others to zero**.
- This selection step acts as a **regularizer** that prevents the use of an overly large number of hidden units when reconstructing the input

  → Better results in **classification** tasks

The **codes** produced by an autoencoder



G.E.Hinton and R.R.Salakhutdinov, *Reducing the Dimensionality of Data with Neural Networks* (2006)

Datasets

- **The Olivetti faces**
  - **400** images (92x112)
  - 40 distinct subjects
  - Grey levels images

- **Faces in the Wild**
  - **30,281** images (86x86)
  - Faces have been automatically labeled

Autoencoder k-Sparse

```
layers {
  bottom: "data"
  top: "encode1"
  name: "encode1"
  type: INNER_PRODUCT
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
  inner_product_param {
    num_output: 1000
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
```

Hidden layer

```
layers {
  bottom: "encode1"
  top: "neuron1"
  name: "neuron1"
  type: TOPK
  topk_param {
  k: 100
  a: 1
  }
}
```

Top K

```
layers {
  bottom: "neuron1"
  top: "decode1"
  name: "decode1"
  type: INNER_PRODUCT
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
  inner_product_param {
    num_output: 784
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
```

Output layer

```
layers {
  bottom: "decode1"
  bottom: "flatdata"
  top: "l2_error"
  name: "loss"
  type: EUCLIDEAN_LOSS
  loss_weight: 1
}
```

Euclidean Loss

Autoencoder k-Sparse

Solver

```
net: "mnist_kSAE.prototxt"
test_state: { stage: 'test-on-test' }
test_iter: 100
test_interval: 1000
test_compute_loss: true
base_lr: 0.01
lr_policy: "fixed"
display: 100
max_iter: 50000
weight_decay: 0.0005
snapshot: 5000
snapshot_prefix:
"examples/mnist/mnist_kSAE"
momentum: 0.9
# solver mode: CPU or GPU
solver_mode: CPU
```

Anatomy of autoencoders k-Sparse

Olivetti: (92x112) – 1000 – (92x112)

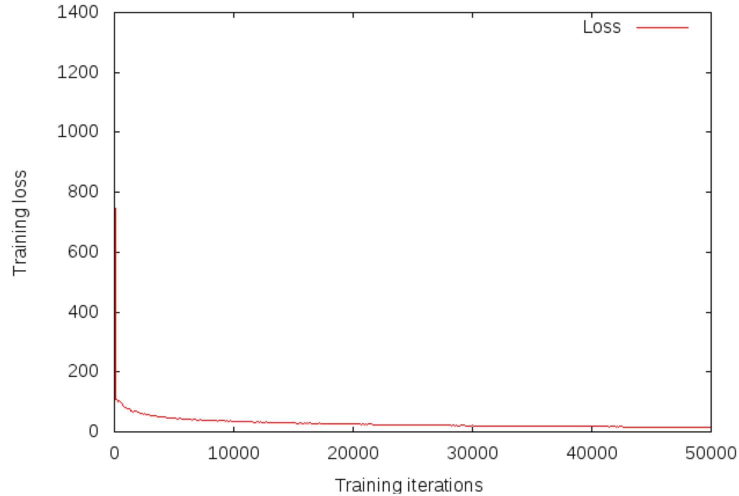
Wild: (55x55) – 1000 – (55x55)

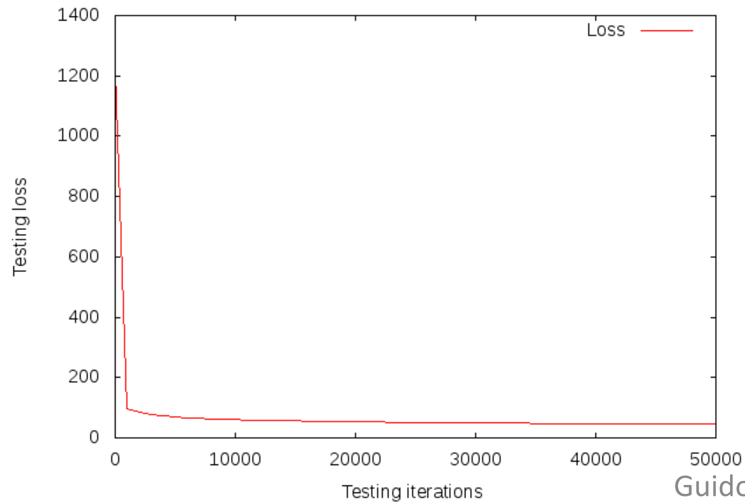
Data Input

```
name:
"MNIST_kSparse_Autoencoder"
layers {
  top: "data"
  name: "data"
  type: IMAGE_DATA
  image_data_param {
    source:
"examples/mnist_train_lmdb"
    backend: LMDB
    batch_size: 100
    is_color: false
    shuffle: true
  }
  transform_param {
    scale: 0.0039215684
  }
  include: { phase: TRAIN }
}
```

Olivetti

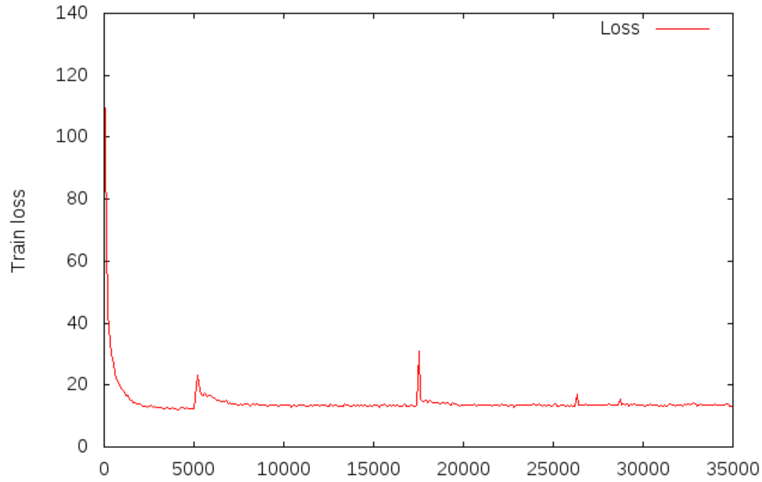
- **No top K**

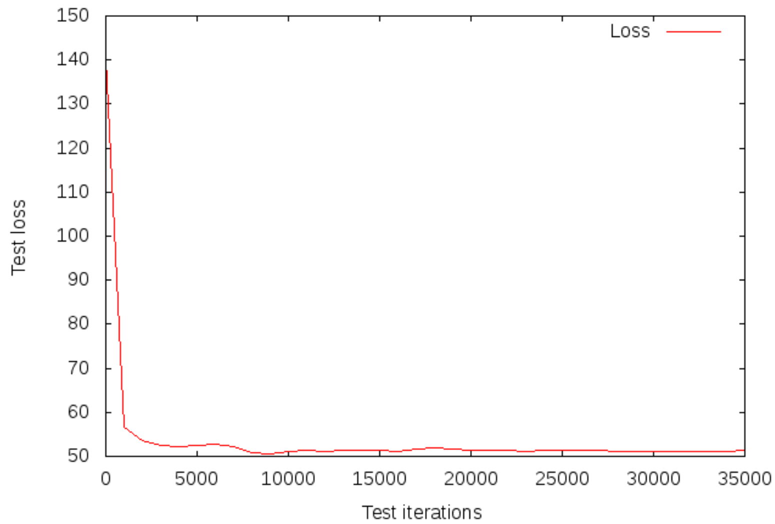




Olivetti

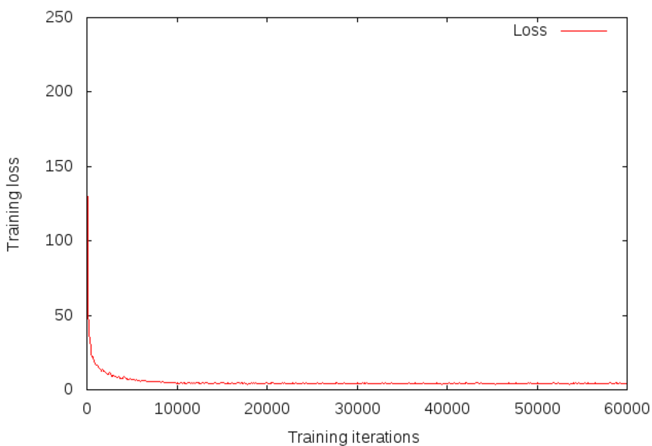
- **K = 70**

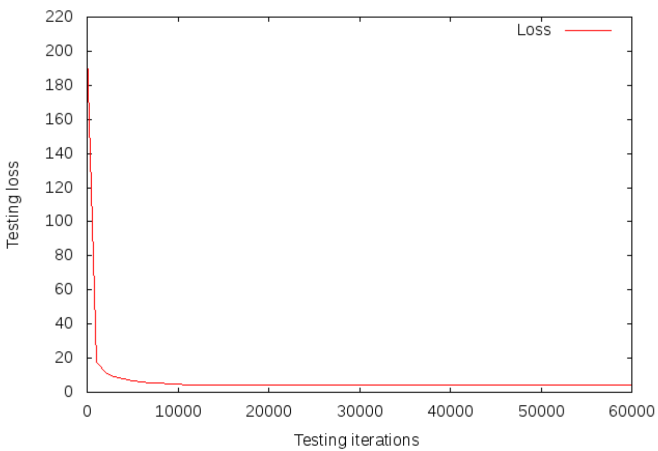




Faces in the Wild

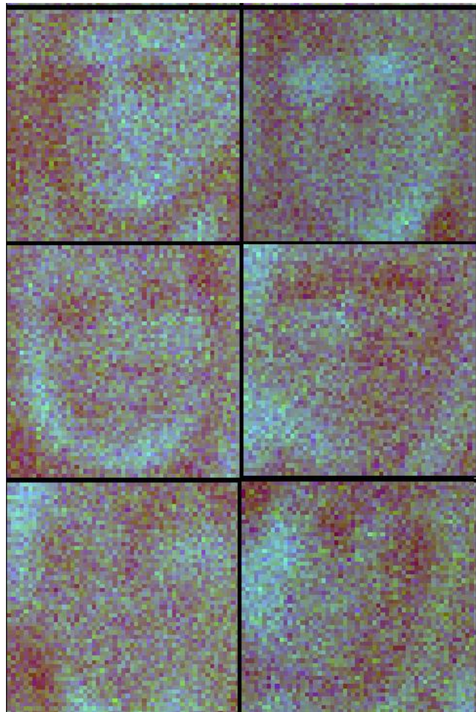
- **No top K**





Faces in the Wild

- **K = 25**



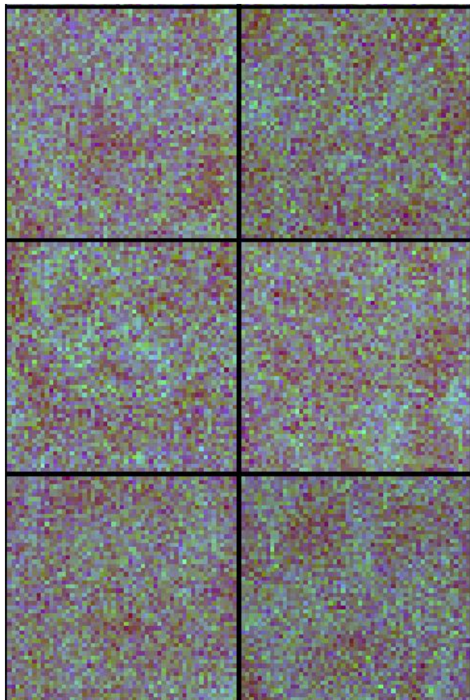
Guido Borghi - gdubrg@gmail.com

Faces in the Wild

- **K = 60**

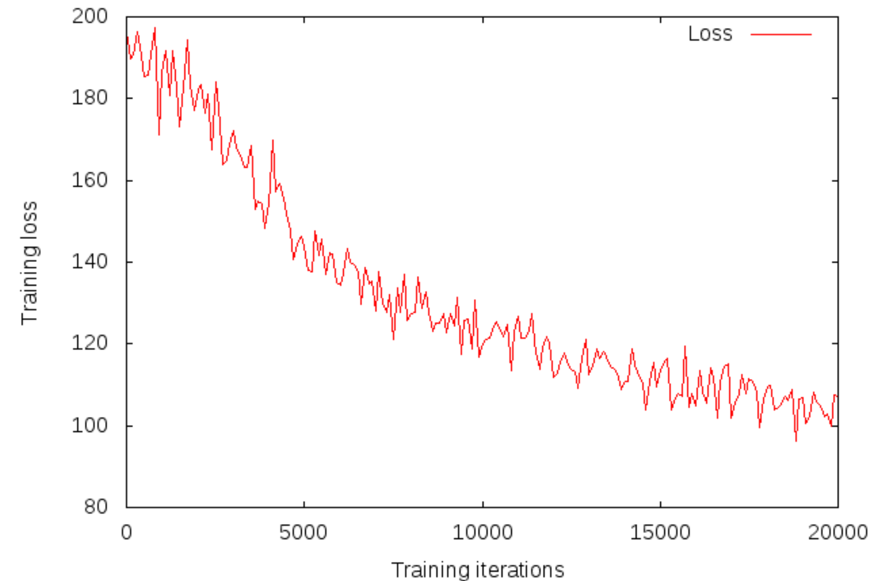
Faces in the Wild

- **K = 200**

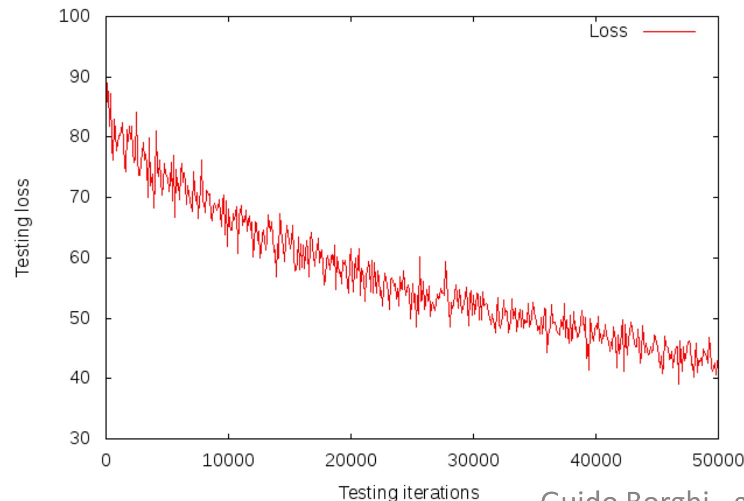


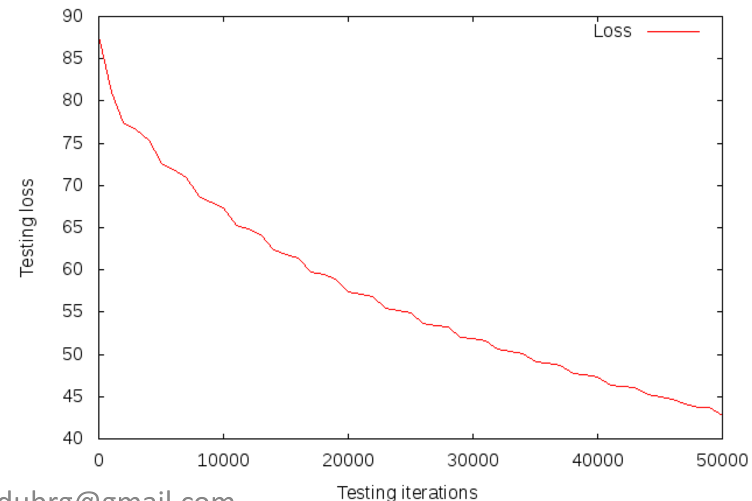
Faces in the Wild - Loss

- **Pre-training** (for all values of k)
  → We use a **Dropout Autoencoder** (50% drop)



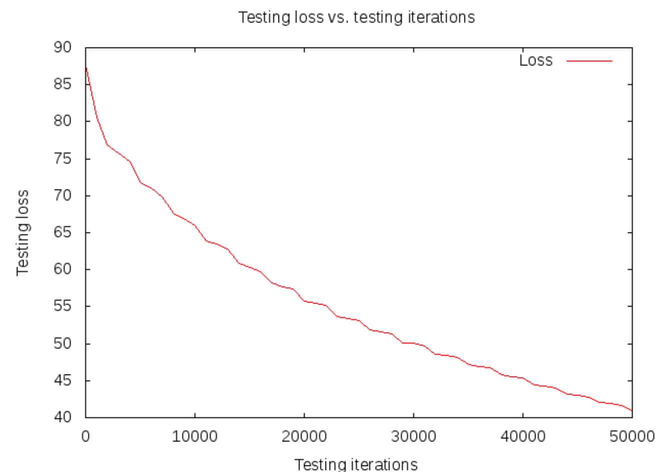
- **Training** (k=25)



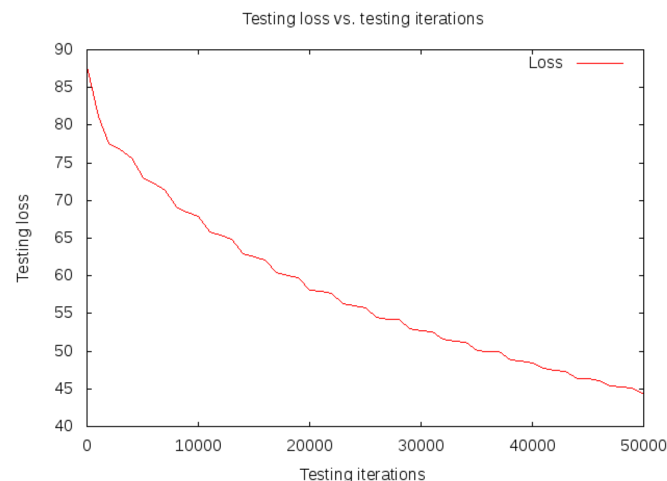


Faces in the Wild - Loss

- K = 200



- K = 60



Pretraining + Training = 3 + 5 = ~ **8 hours** (on *i7-950 @3.07 GHz*)

Considerations about **Sparsity Level (K)**

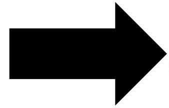
- **Large** values of k → **very local features**
  - Too primitive
  - They could be used to pre-training Deep Neural Nets
- **Middle** values of k → **global features**
- **Too much** sparsity → **too global features**
  - They do not factor the input into parts

Test 1

- **K = 60**

Test 2

- **K = 25**



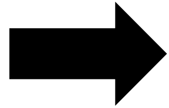
Guido Borghi - gdubrg@gmail.com

Test 3

- **K = 60**

Test 4

- **K = 25**



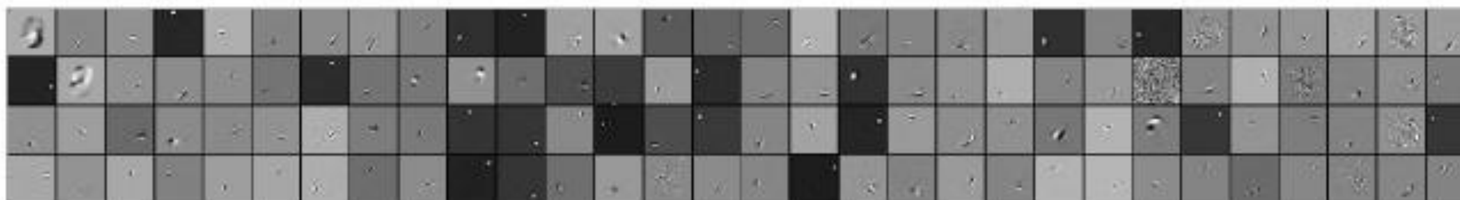
Reconstructed input (faces)

- K = 25

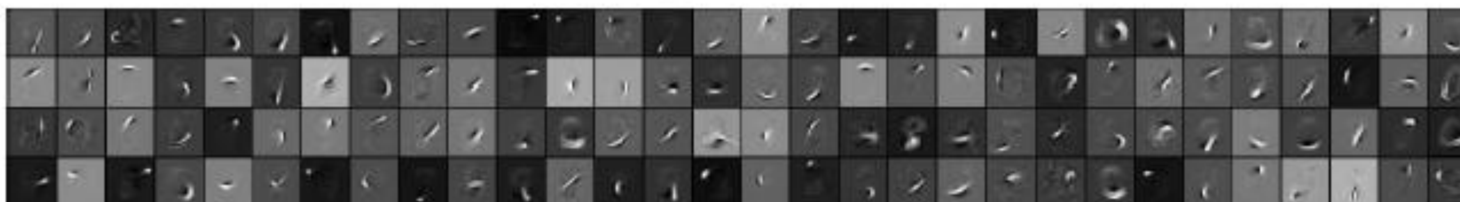
- K = 60

- K = 200

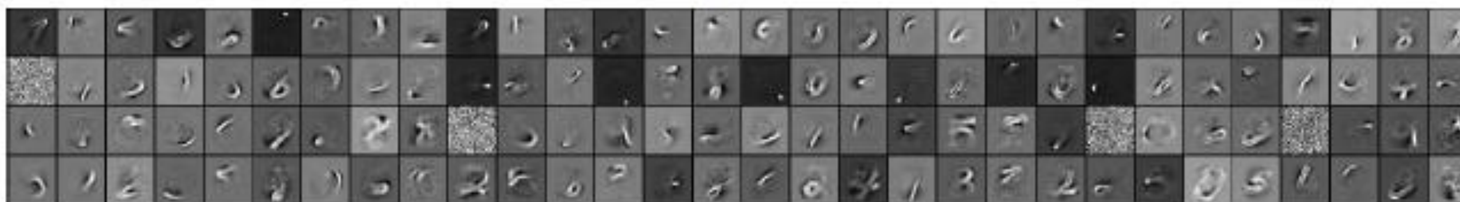
Autoencoder k-Sparse with **MNIST**



(a) $k = 70$



(b) $k = 40$

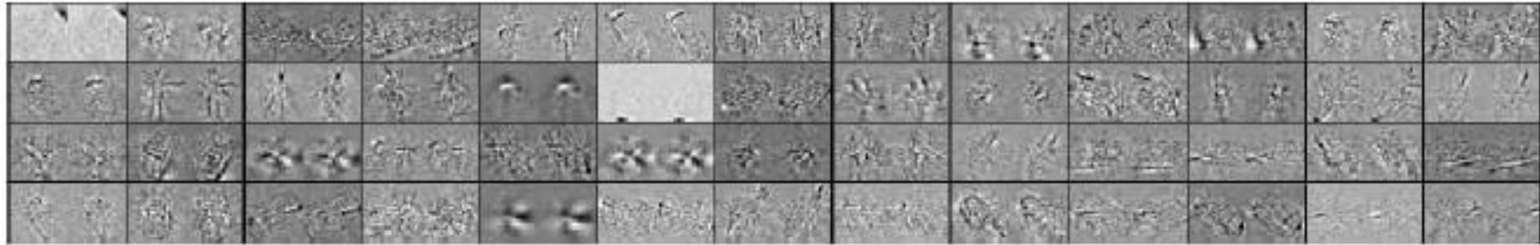


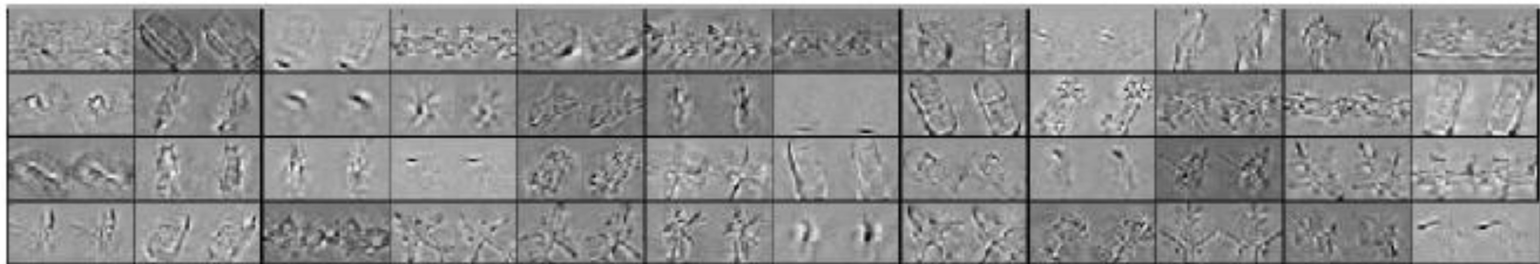
(c) $k = 25$

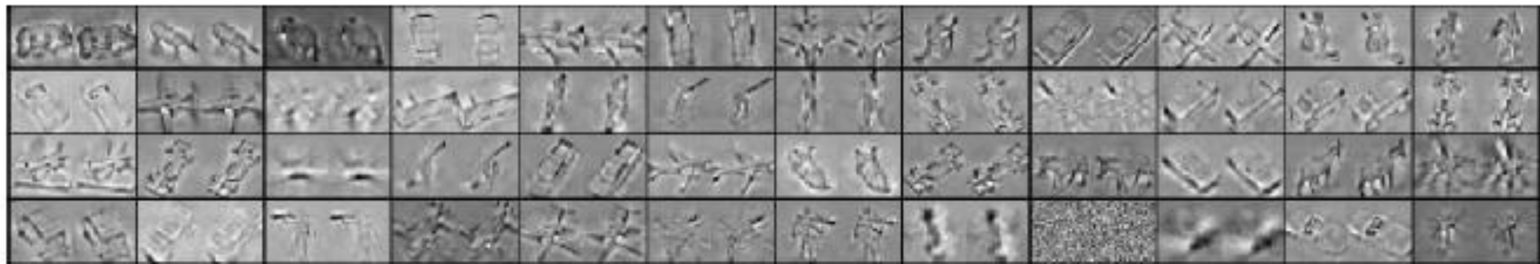
(d) $k = 10$

Alireza Makhzani, Brendan Frey, k-Sparse Autoencoders (2014)

Autoencoder k-Sparse with **NORB**


(a) $k = 200$


(b) $k = 150$

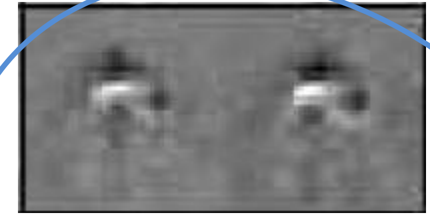

(c) $k = 50$

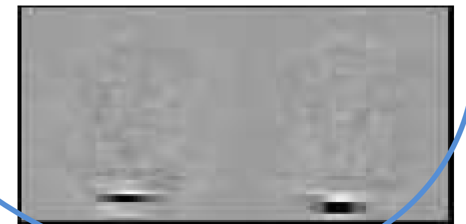
Alireza Makhzani, Brendan Frey, k-Sparse Autoencoders (2014)

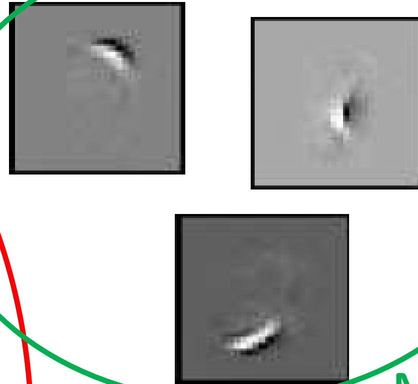
Final considerations (I)

- Some filters like **Gabor filter**

NORB

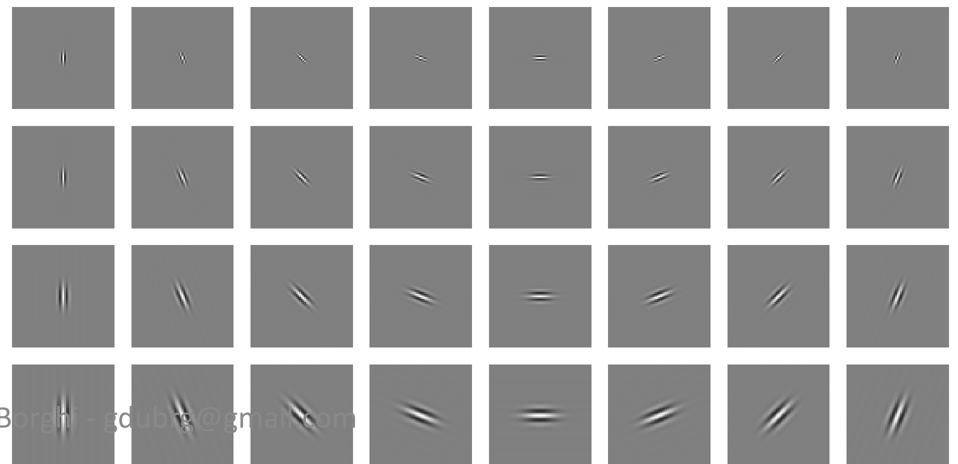






MNIST

WILD



Final considerations (II)

- Some filters remember **Viola et Jones**