Machine Learning in a Nutshell

15-488 Spring '20

Lecture 33:
Neural Networks 2

Teacher:
Gianni A. Di Caro

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

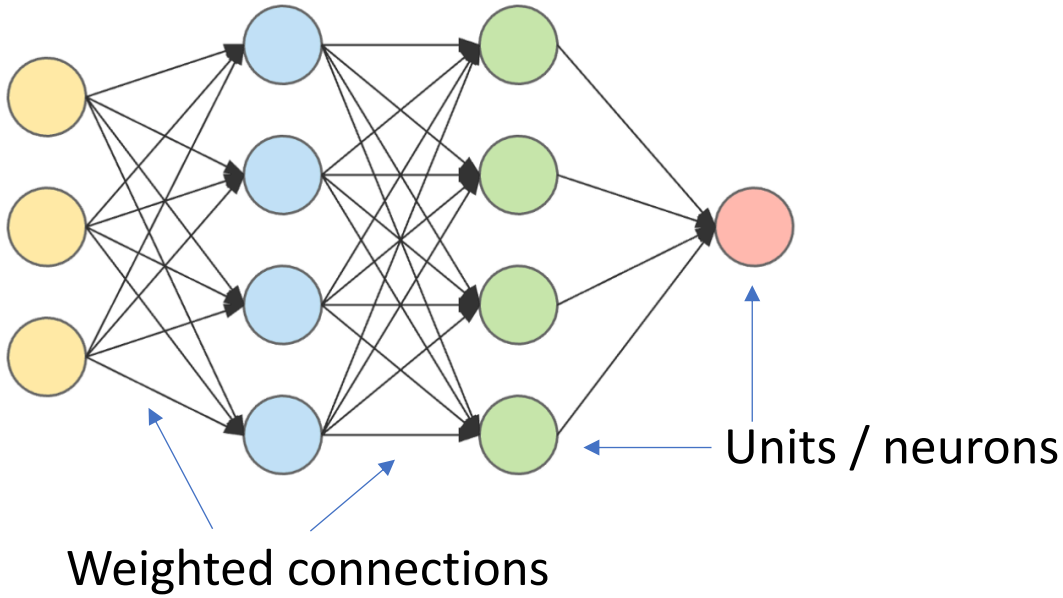
(Recap) Neural Networks: general definition

❖ **Goal**: Learn a generic non-linear mapping for making *predictions*, $f: X \rightarrow Y$

❑ A neural network aims to estimate f through a **parametric estimator** $f_{\boldsymbol{w}}$

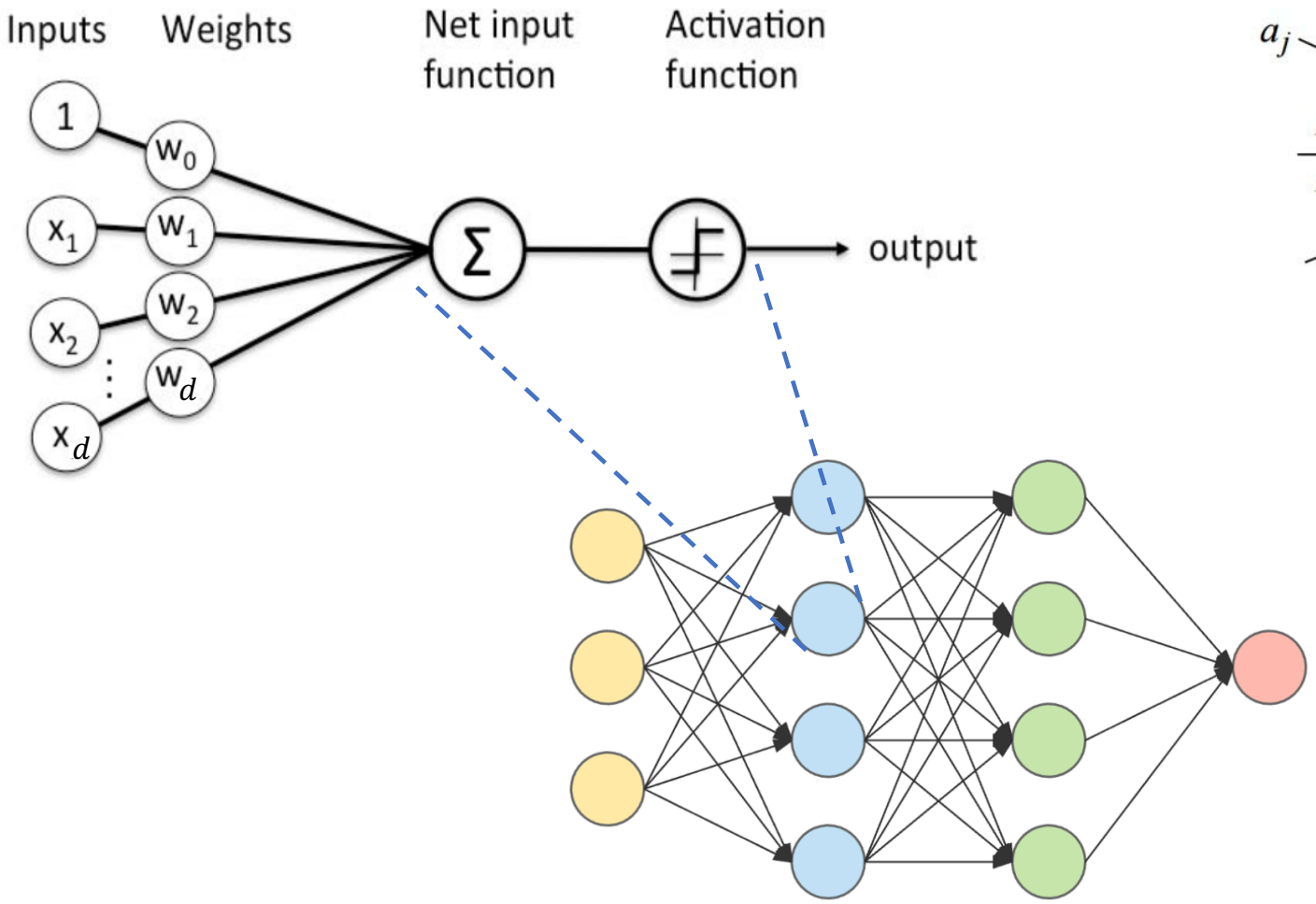
➢ $f_{\boldsymbol{w}}$ is given implicitly, through the definition of network of basic activation **units**, where the **parameters $\boldsymbol{w}$** to learn are the **connection weights** between the units

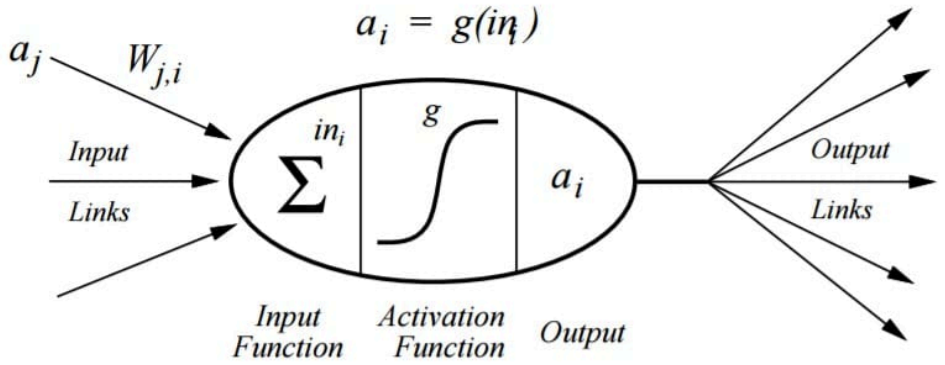
❖ A NN *mimics/abstracts* the behavior of a biological brain, where the units are the equivalent of the **neurons** and the connections are designed after **synaptic connections**



(Recap) Neural Networks: the units / neurons

❖ Each unit is a **perceptron** unit with a non-linear **activation function**





$$a_i = g(\sum_j W_{j,i} a_j)$$

A unit has three components:

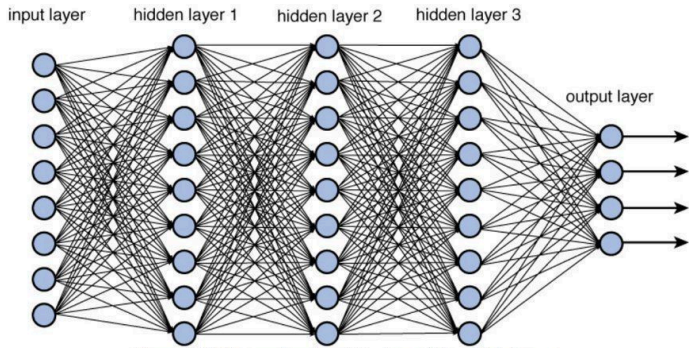
- **Input** function (linear)
- **Activation** function (non-linear)
- **Output** of the activation (fan-out)

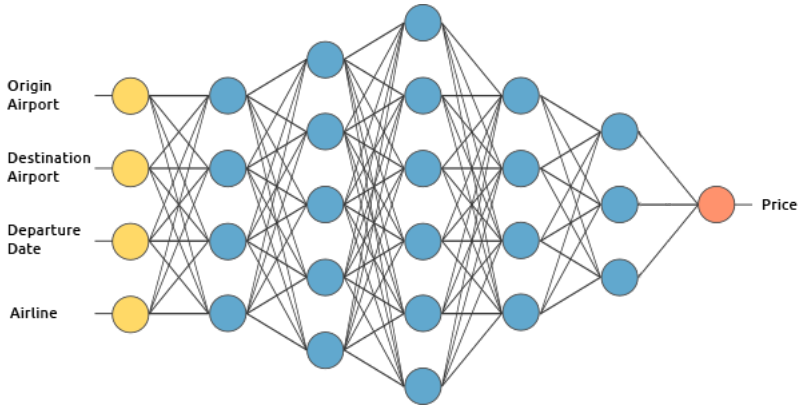
(Recap) Neural Networks: the connection topology

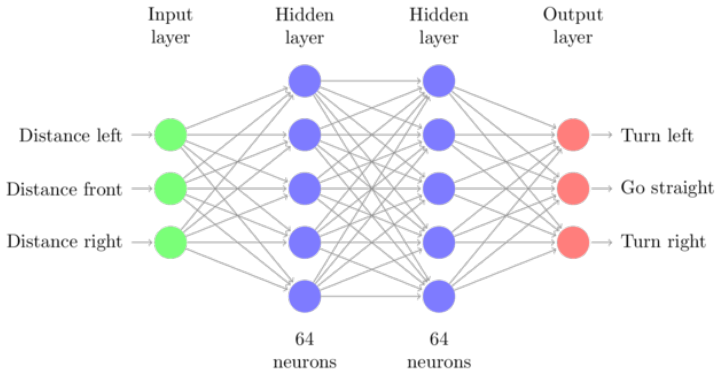
- Any arbitrary **connection topology** can be used (design choice)

❖ **Feed-forward** networks:

- ✓ Topology is organized in a **sequence of layers**
- ✓ Data processing proceeds through stages, from the input to output layers (**visible layers**), passing through intermediate layers (**hidden layers**)

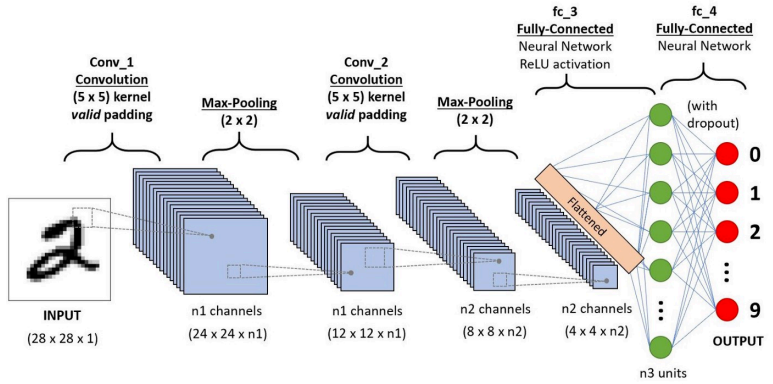






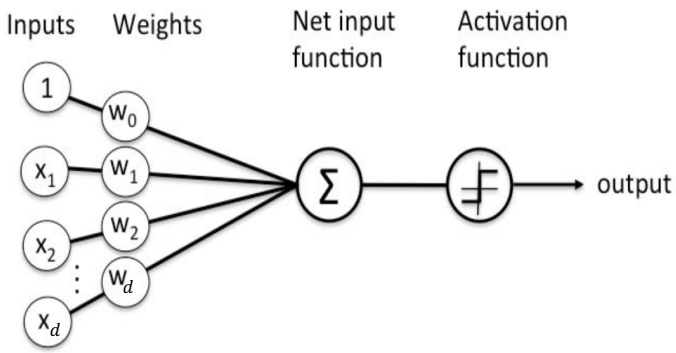
Multi-Layer Perceptron (MLP)

Fully connected (dense) layer: each unit of layer i is fully connected with the units of the previous layer

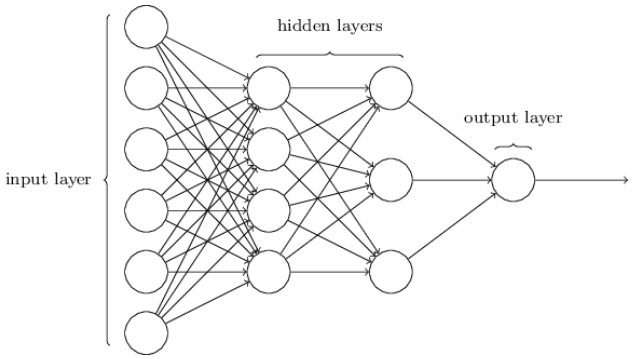


Convolutional Neural Network (CNN):
Not (all) fully connected layers

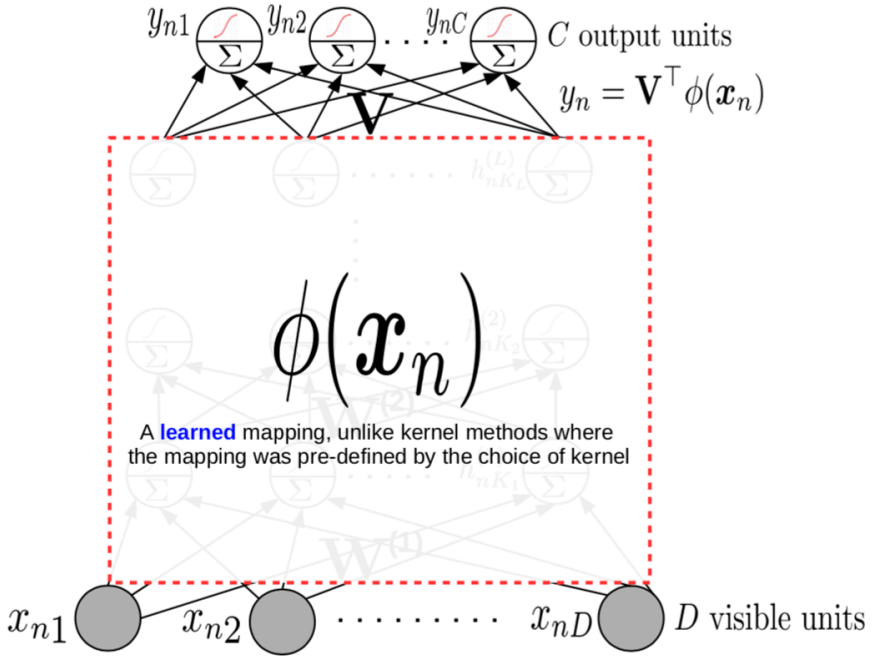
(Recap) Neural Networks: hidden layers and feature maps



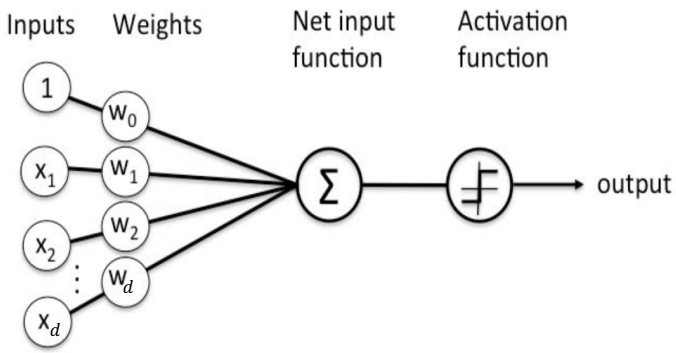
- ✓ Each unit gets <u>feature values as input</u> and performs a **(non-linear) feature extraction**
- ✓ The output can be used *directly,* or can be used as input feature for the units in the next layer



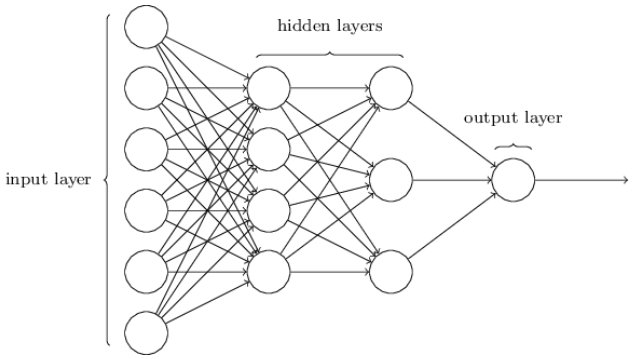
- ➢ The outputs from all units in a (hidden) layer generate a **feature vector**, which is the input for the next layer
- ❖ Multi-staged, recursive feature generation
- ✓ **Hidden layers learn a *feature map* $\boldsymbol{\phi}(\boldsymbol{x})$**
- ✓ $\boldsymbol{\phi}(\boldsymbol{x})$ realizes a highly non-linear transformation of the linear input features $\boldsymbol{x}$



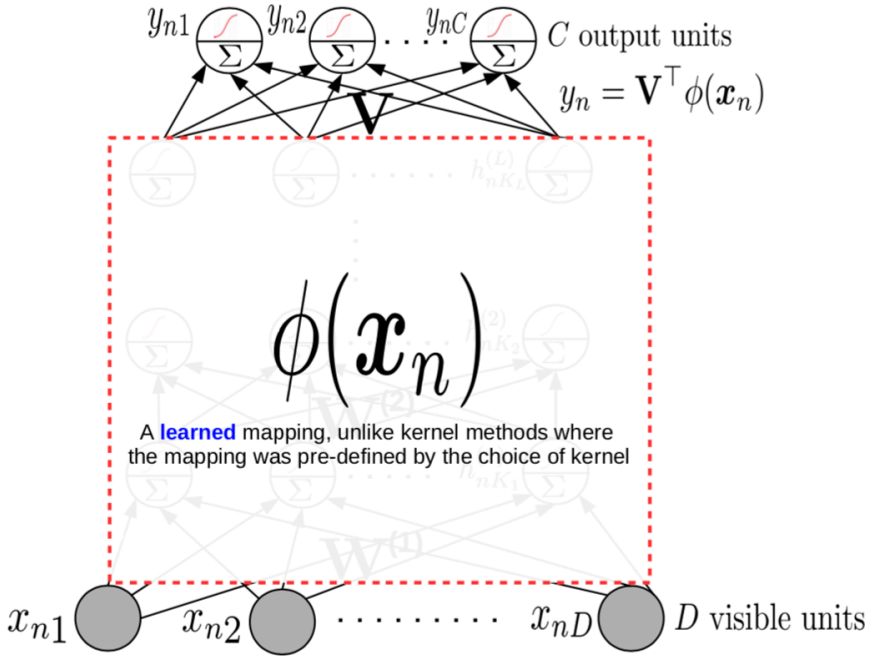
(Recap) Neural Networks: hidden layers and feature maps



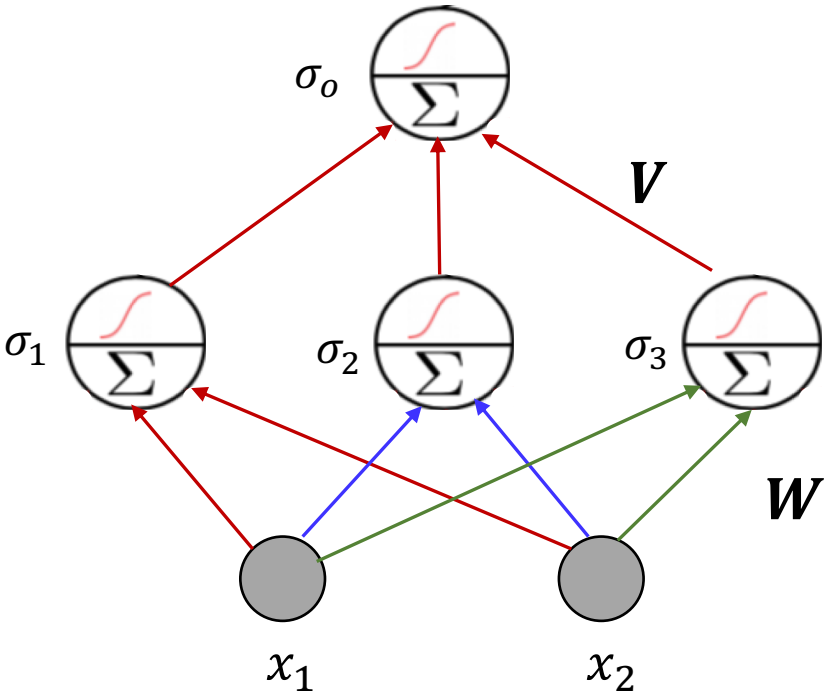
- ✓ Each unit gets <u>feature values as input</u> and performs a **(non-linear) feature extraction**
- ✓ The output can be used *directly,* or can be used as input feature for the units in the next layer

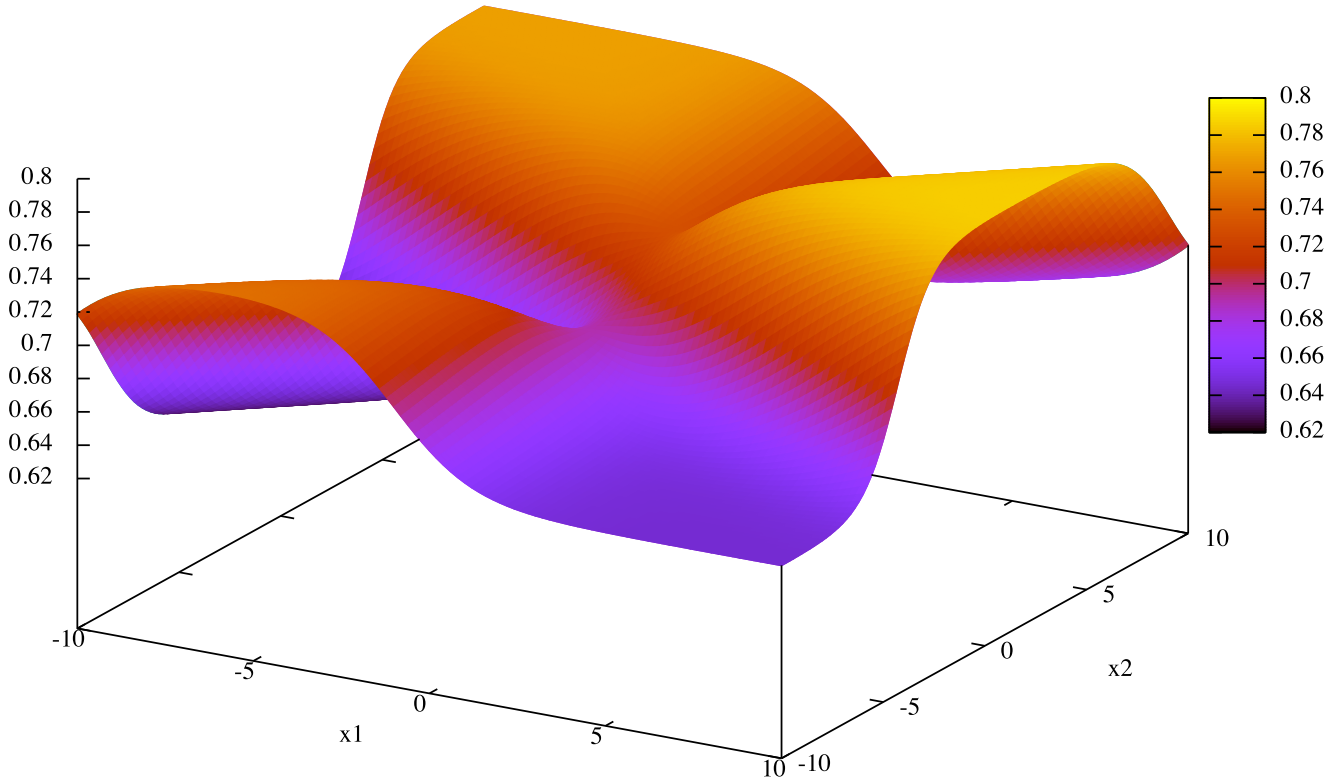


- ➢ The outputs from all units in a (hidden) layer generate a **feature vector**, which is the input for the next layer
- ❖ Multi-staged, recursive feature generation
- ✓ **Hidden layers learn a *feature map* $\boldsymbol{\phi}(\boldsymbol{x})$**
- ✓ $\boldsymbol{\phi}(\boldsymbol{x})$ realizes a highly non-linear transformation of the linear input features $\boldsymbol{x}$



A NN represents a Highly non-linear function!





$$y = \sigma\left(\ \sigma(x_1 w_{11} + x_2 w_{21})\ \right) \cdot v_1 + \sigma(x_1 w_{12} + x_2 w_{22}) \cdot v_2 + \sigma(x_1 w_{13} + x_2 w_{23}) \cdot v_3\ )$$

$$W_{2\times 3} = \begin{bmatrix} 0.5 & 0.8 & -0.9 \\ 1.2 & -0.9 & 0.1 \end{bmatrix} \qquad V_{3\times 1} = \begin{bmatrix} 0.7 \\ 0.6 \\ -0.5 \end{bmatrix}$$

Non-convex!

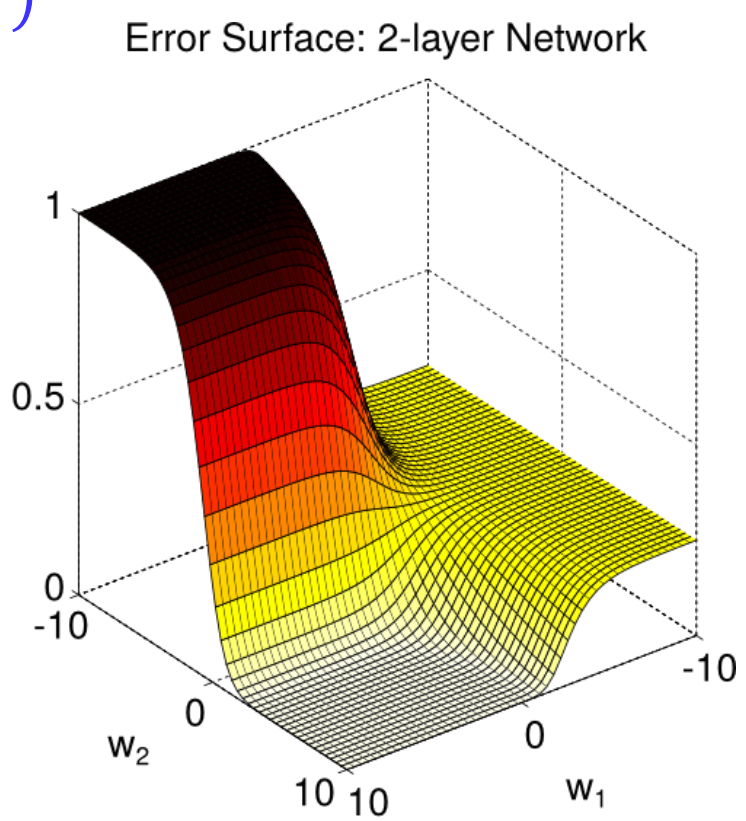
Goal: Minimize the expected error over the dataset

Goal: Given a dataset $\mathcal{D} = \left\{\left(\boldsymbol{x}^{(i)}, y^{(i)}\right)\right\}_{i=1}^{m}$, where $\boldsymbol{x}^{(i)} \in X \subseteq \mathbb{R}^d, y^{(i)} \in Y \subseteq \mathbb{R}^C$,
find $\boldsymbol{w} \in \mathcal{W} \subseteq \mathbb{R}^d$ that minimizes the expected error, the **loss**, over the dataset

Squared loss: $$E_{\mathcal{D}}[\boldsymbol{w}] = \frac{1}{2}\sum_{i=1}^{m}\left(f_{\boldsymbol{w}}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)^2$$

- This error measure defines a non-convex surface over the hypothesis (i.e. **weight**) space

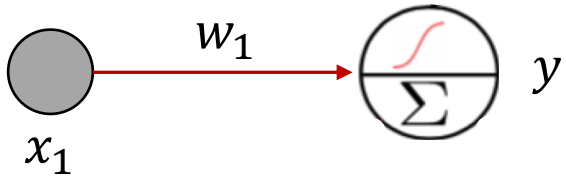
- Complex optimization problem: commonly approached using a **gradient descent approach**

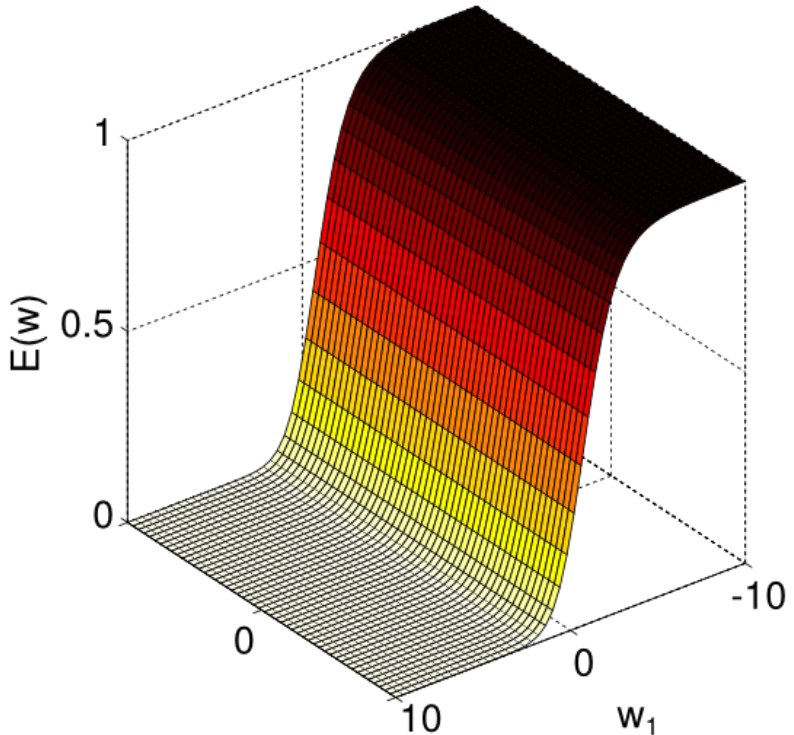


Minimize the expected squared error: non-convex error surface

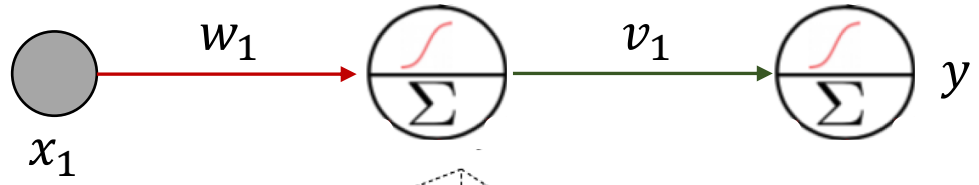
- Learn to classify $x_1 = 1$ as 1 $\quad E_{\mathcal{D}}[\boldsymbol{w}] = \frac{1}{2}\sum_{i=1}^{1}\left(f_{\boldsymbol{w}}(x^{(i)}) - 1\right)^2$

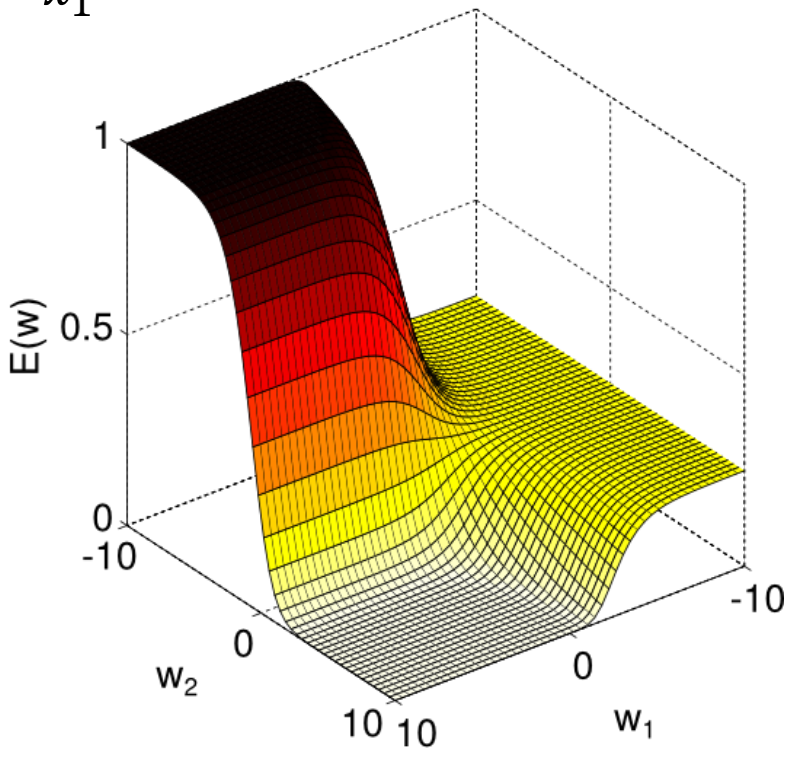
$y = \sigma(x_1 \cdot w_1)$



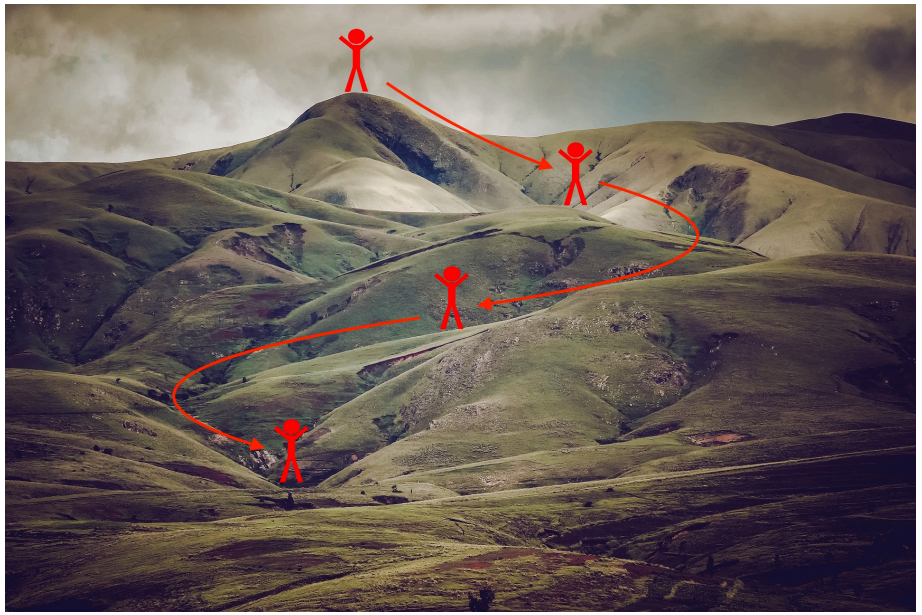


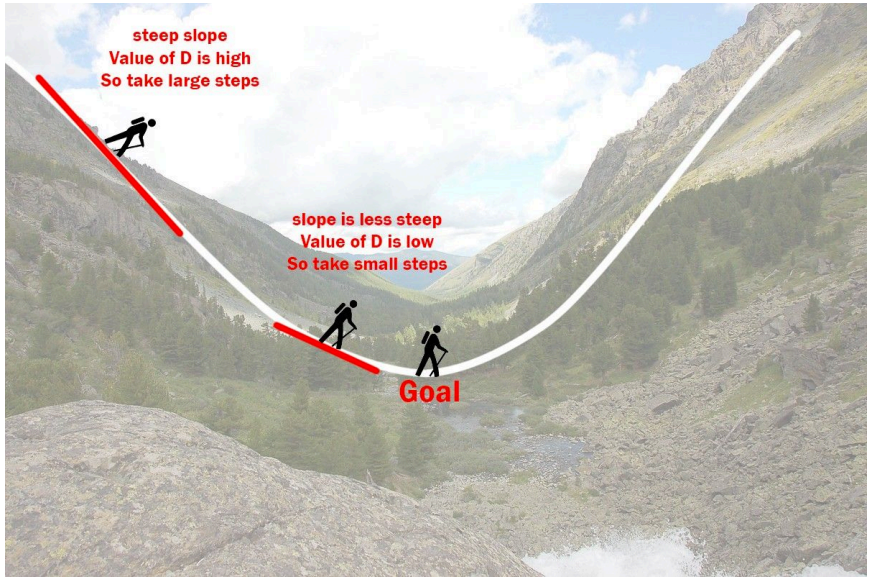
$y = \sigma\left(\left(\sigma(x_1 \cdot w_1)\right) \cdot w_2\right)$



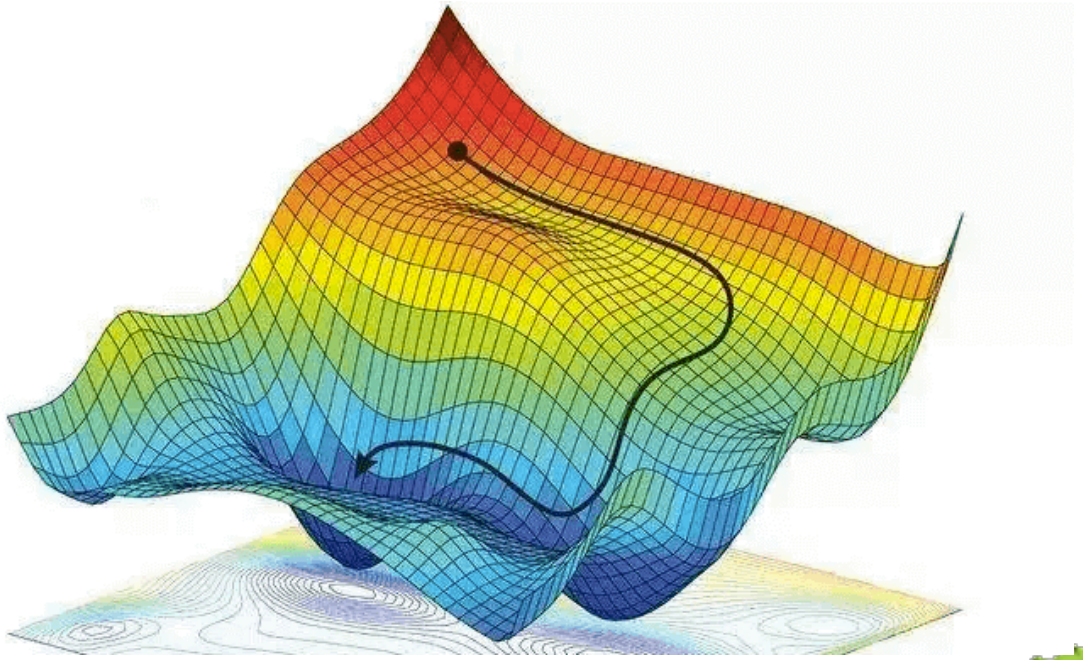


Gradient descent: a hiking illustration

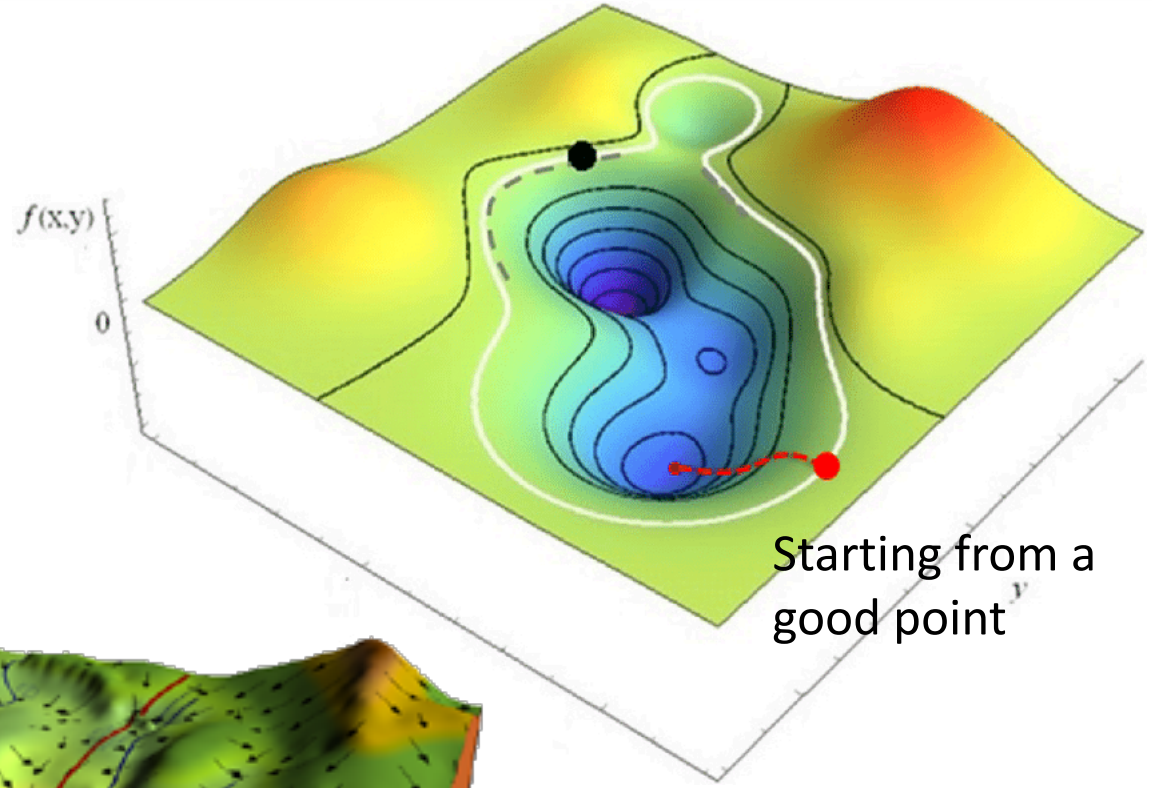




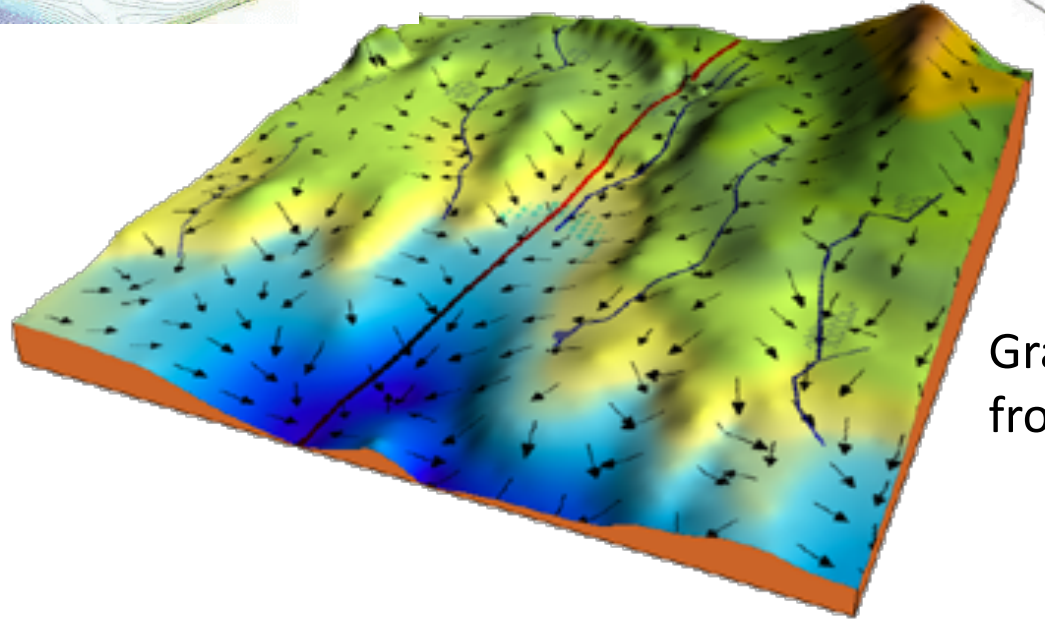
Gradient descent: an illustration on 3D functions



Imagine a ball rolling over
a potential field ...

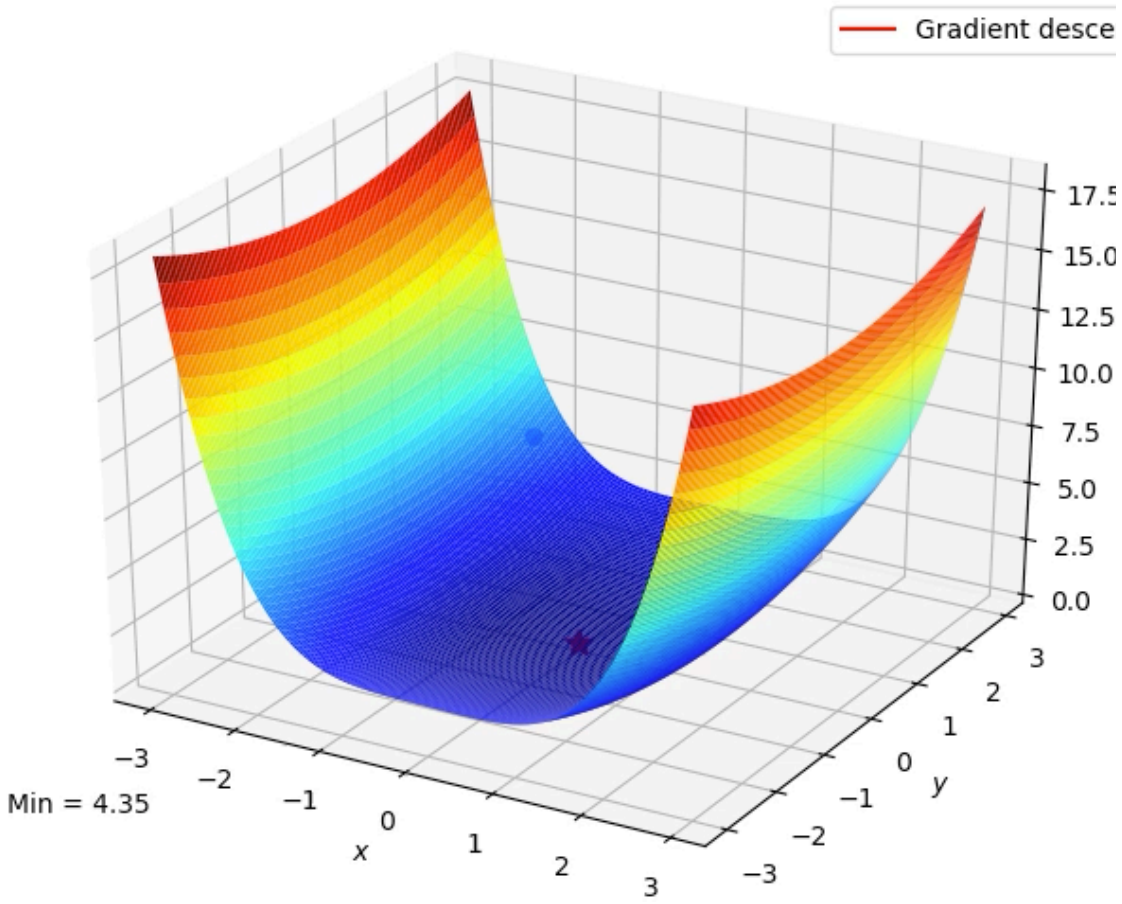


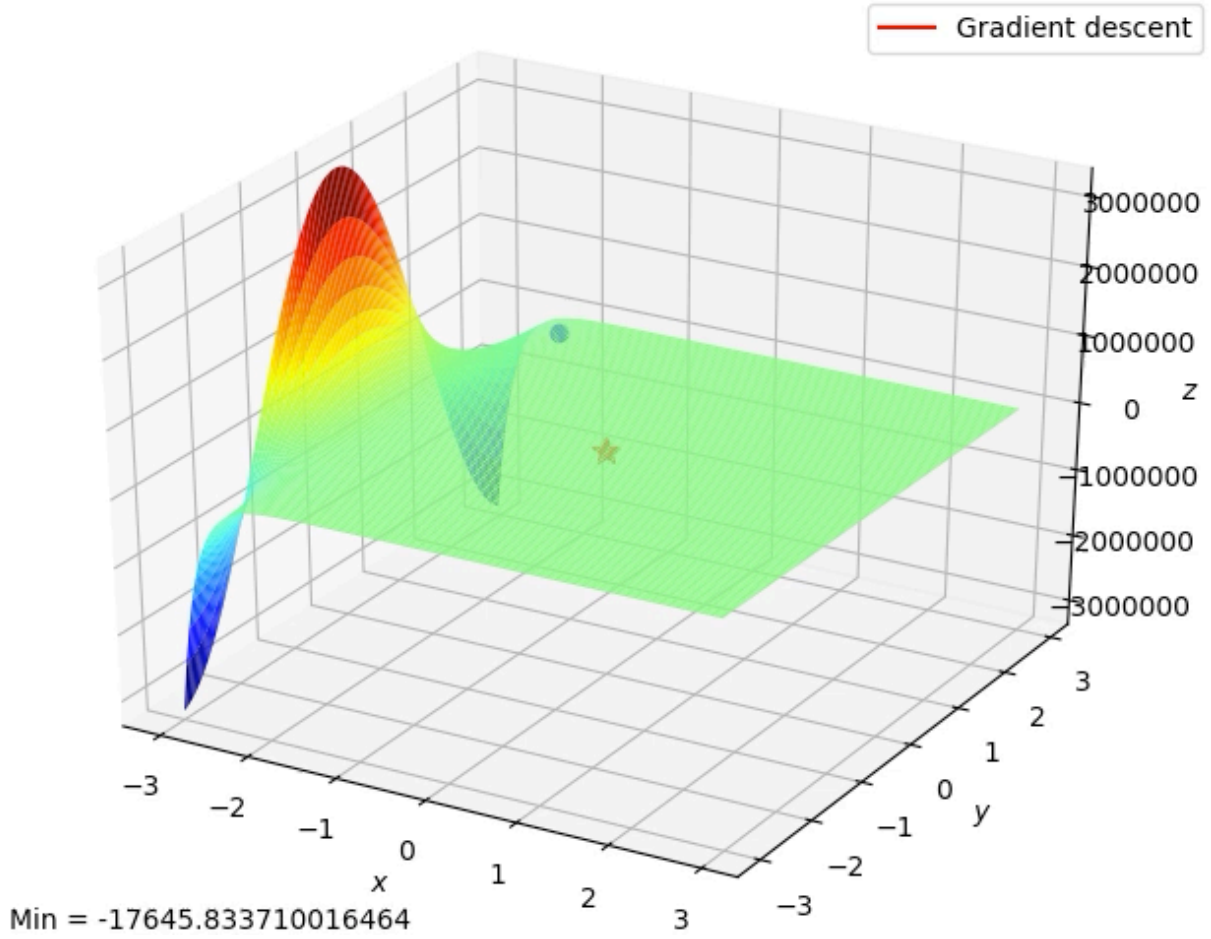
Starting from a
good point



Gradient vector directions
from different starting points

Gradient descent: Convex vs. Non-Convex function



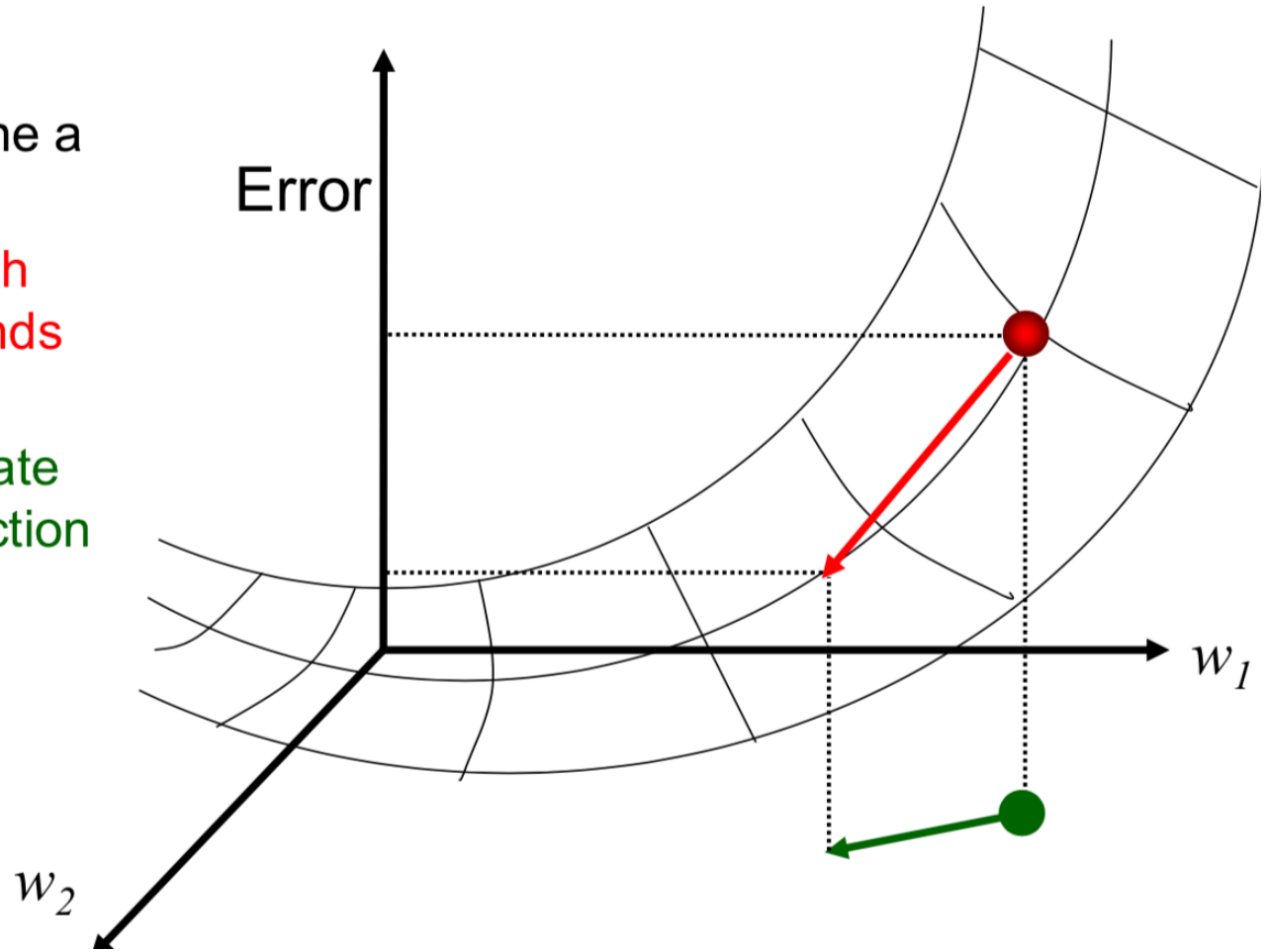


Gradient descent to minimize the expected squared error

Gradient descent is an iterative process aimed at finding a minimum in the error surface.

on each iteration

- current weights define a point in this space
- find direction in which error surface descends most steeply
- take a step (i.e. update weights) in that direction



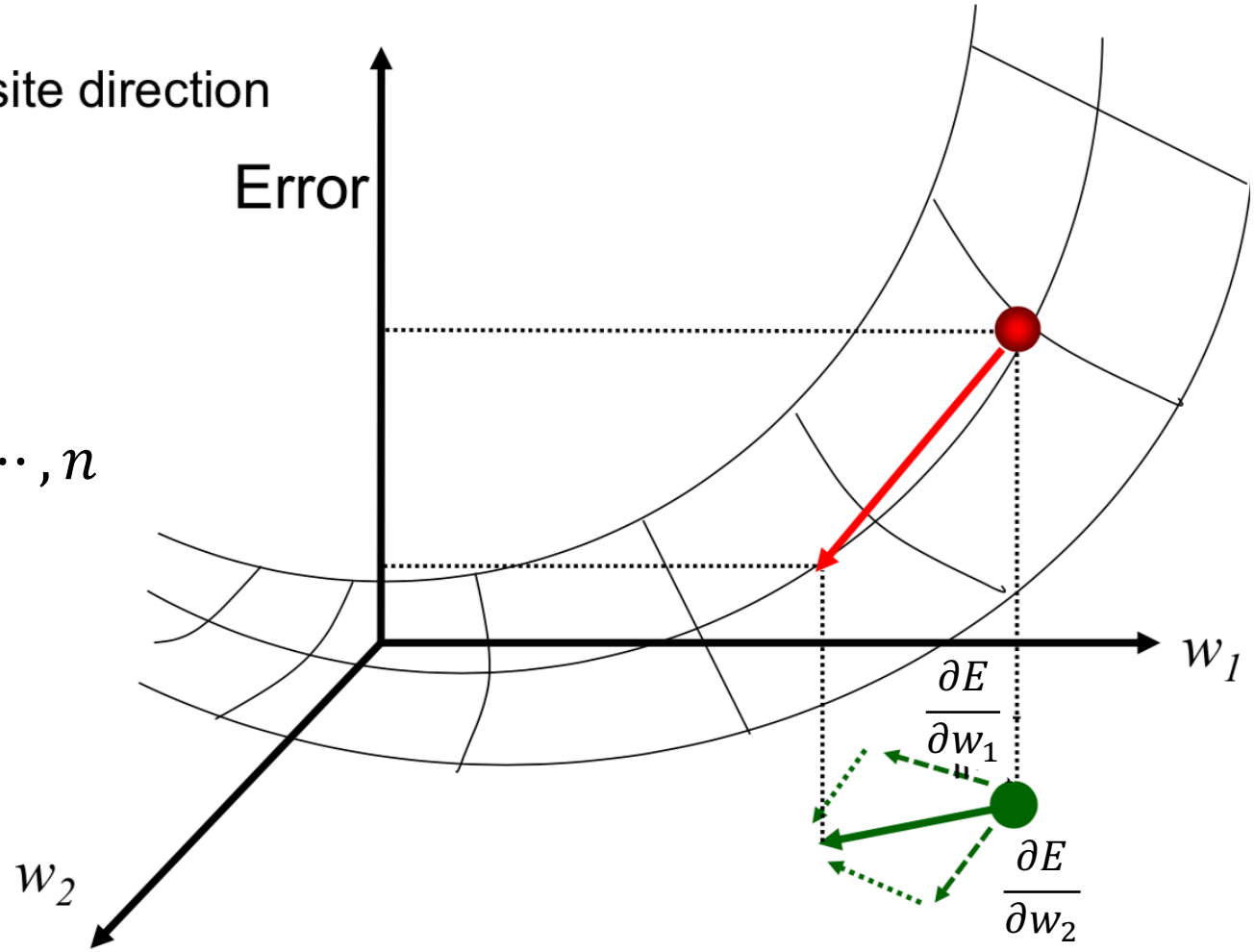
Gradient descent to minimize the expected squared error

Calculate the gradient of E: $\nabla E(\boldsymbol{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n} \right]$

Take a step in the opposite direction

$$\Delta \boldsymbol{w} = -\eta \nabla E(\boldsymbol{w})$$

$$\Delta w_i = -\eta \nabla E(w_i) \quad i = 1, \cdots, n$$



Gradient descent implementations

Batch mode Gradient Descent:

Do until satisfied

1. Compute the gradient $\nabla E_D[\boldsymbol{w}]$
2. $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla E_D[\boldsymbol{w}]$

Incremental (stochastic) Gradient Descent:

Do until satisfied

- For each training example d in D

1. Compute the gradient $\nabla E_d[\boldsymbol{w}]$
2. $\mathbf{w} \leftarrow \boldsymbol{w} - \eta \nabla E_d[\boldsymbol{w}]$

Note: Incremental Gradient Descent can approximate *Batch* arbitrarily closely if η made small enough

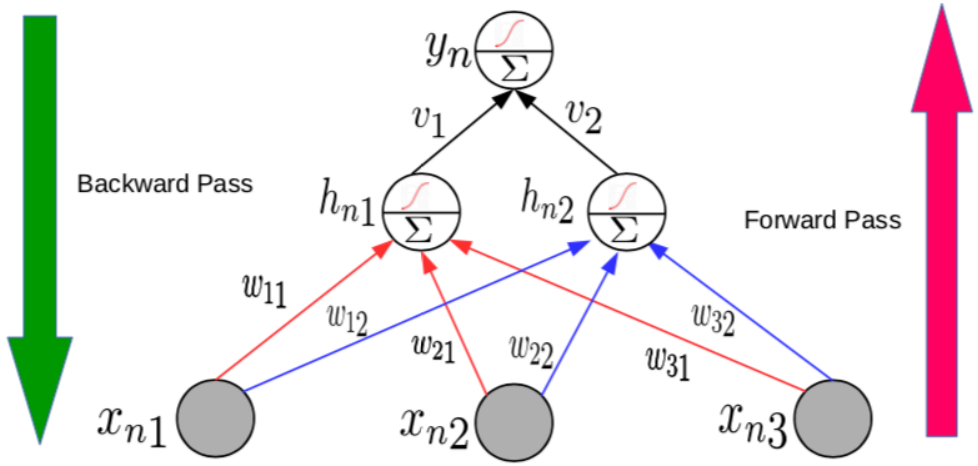
Mini-batch Gradient Descent:

Do until satisfied

1. Select (randomly) a set m of examples in D
2. Compute the gradient over the set m
3. Update the weights

Learning using backpropagation

- Backprop iterates between a forward pass and a backward pass



- Forward pass computes the errors $\boldsymbol{e}_n$ using the current parameters
- Backward pass computes the gradients and updates the parameters, starting from the parameters at the top layer and then moving backwards
- Implementing backprop by hand may be very cumbersome for complex, very deep NNs

Backpropagation: convergence and design choices

- Balance **mini-batch size** (accuracy of gradient estimate vs. computations)
- Organize learning in **epochs**, where each epoch is a full sweep of the training set using SGD, based on the chosen batch sizes

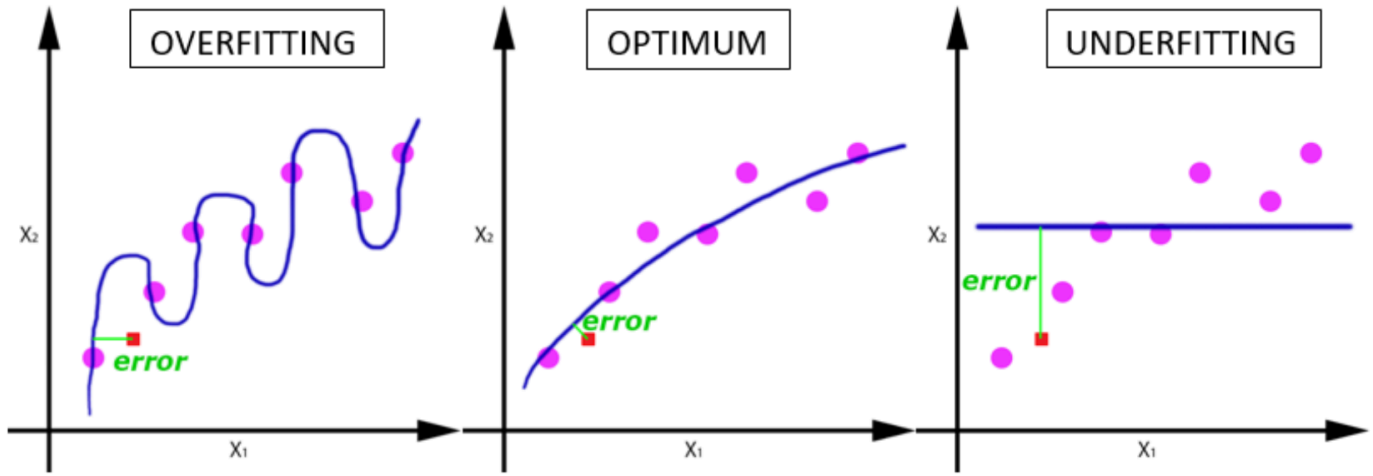
Epoch 1:
Weights[0] → ○ for m in batches:
❖ SGD(m)

Epoch 2:
Weights[1] → ○ for m in batches:
❖ SGD(m)

→ ⋯

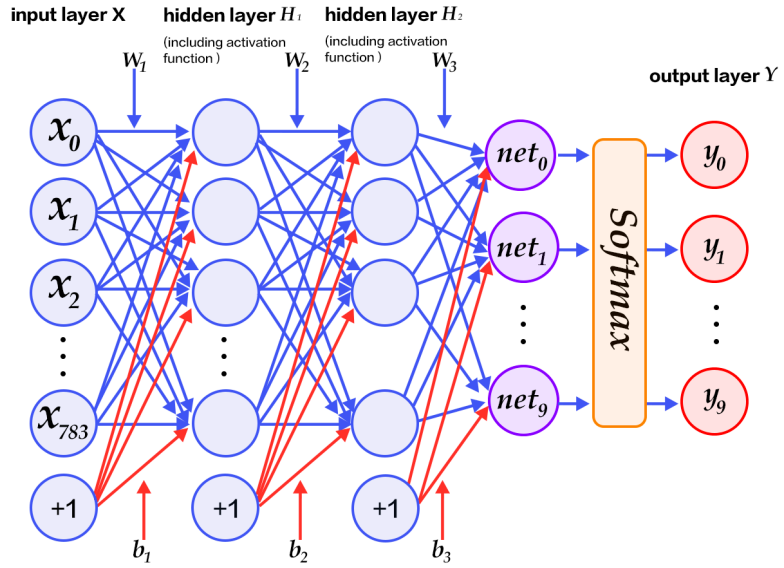
Epoch N:
○ for m in batches:
❖ SGD(m)

Too many epochs

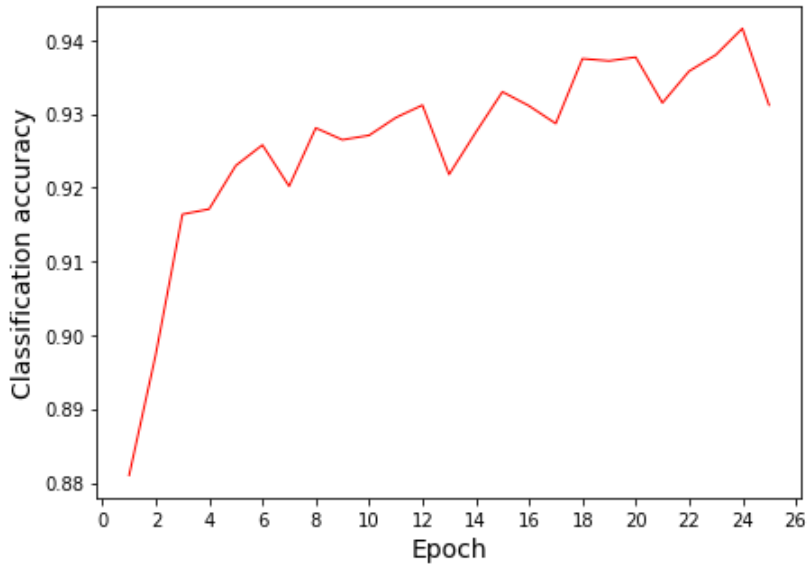


Too little epochs

MLP example



MNIST hand-written character recognition

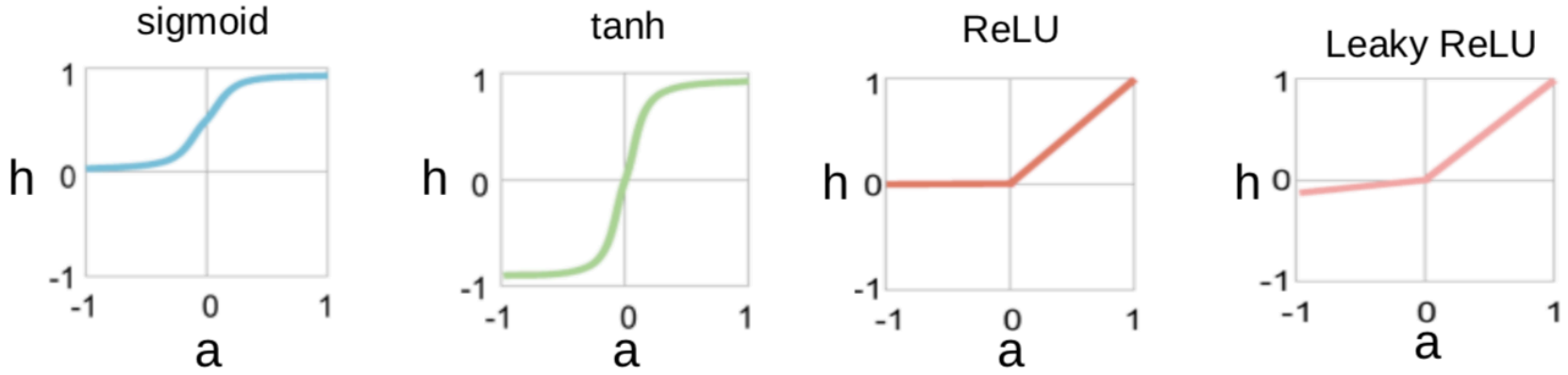



```
net.SGD(training_data, epochs=25,
        mini_batch_size=10, eta=3.0, test_data)
```

Check the Notebook for code!

Common activation functions: Optimization matters!

- Some common activation functions



- Sigmoid: $h = \sigma(a) = \frac{1}{1+\exp(-a)}$
- tanh (tan hyperbolic): $h = \frac{\exp(a)-\exp(-a)}{\exp(a)+\exp(-a)} = 2\sigma(2a) - 1$
- ReLU (Rectified Linear Unit): $h = \max(0, a)$
- Leaky ReLU: $h = \max(\beta a, a)$ where β is a small postive number
- Several others, e.g., Softplus $h = \log(1 + \exp(a))$, exponential ReLU, maxout, etc.
- Sigmoid, tanh can have issues during backprop (saturating gradients, non-centered)
- ReLU/leaky ReLU currently one of the most popular (also cheap to compute)

Expressive capabilities of NN: the good

Boolean functions:

- Every Boolean function can be represented by a network with a single hidden layer
- But might require exponential (in number of inputs) hidden units

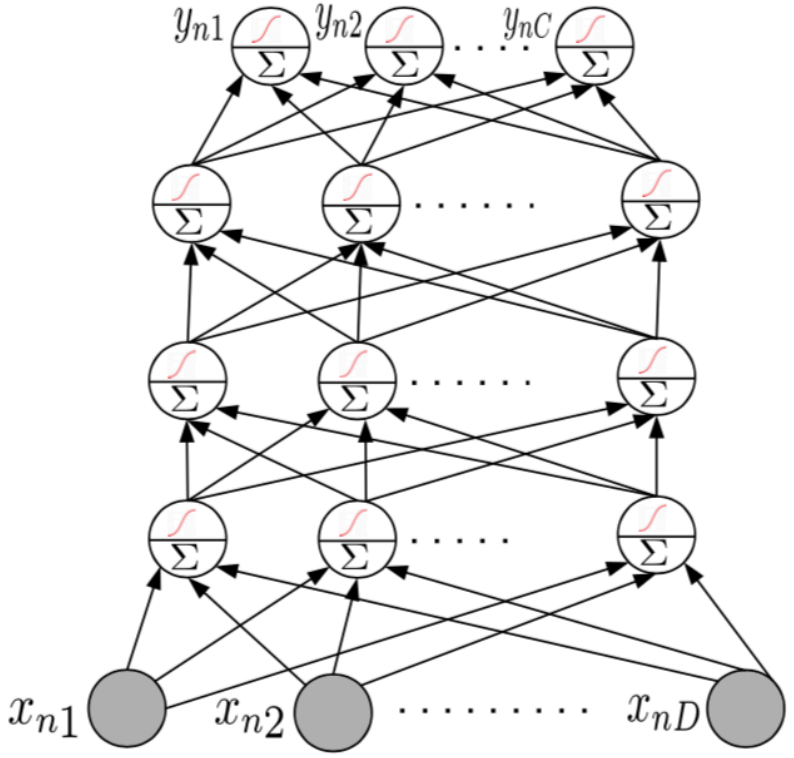
Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989; Hornik et al. 1989]

- Any function can be approximated to arbitrarily accuracy by a network with two hidden layers [Cybenko 1988]

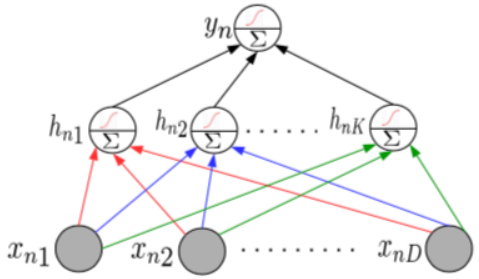
… the bad: design issues

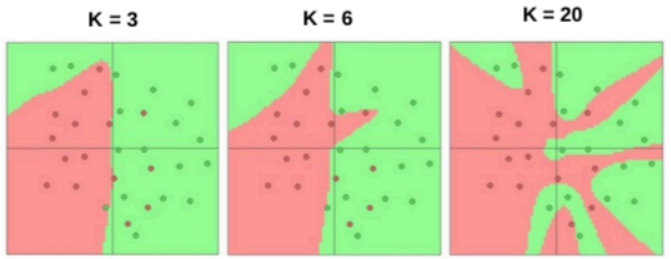
- Much of the magic lies in the hidden layer(s)
- As we've seen, hidden layers learn and detect good features
- However, we need to consider a few aspects
  - Number of hidden layers, number of units in each hidden layer
  - Why bother about many hidden layers and not use a single very wide hidden layer (recall Hornick's universal function approximator theorem)?
  - Complex network (several, very wide hidden layers) or simple network (few, moderately wide hidden layers)?
  - Aren't deep neural network prone to overfitting (since they contain a huge number of parameters)?



Representational power of hidden layers

- Consider an NN with a single hidden layer

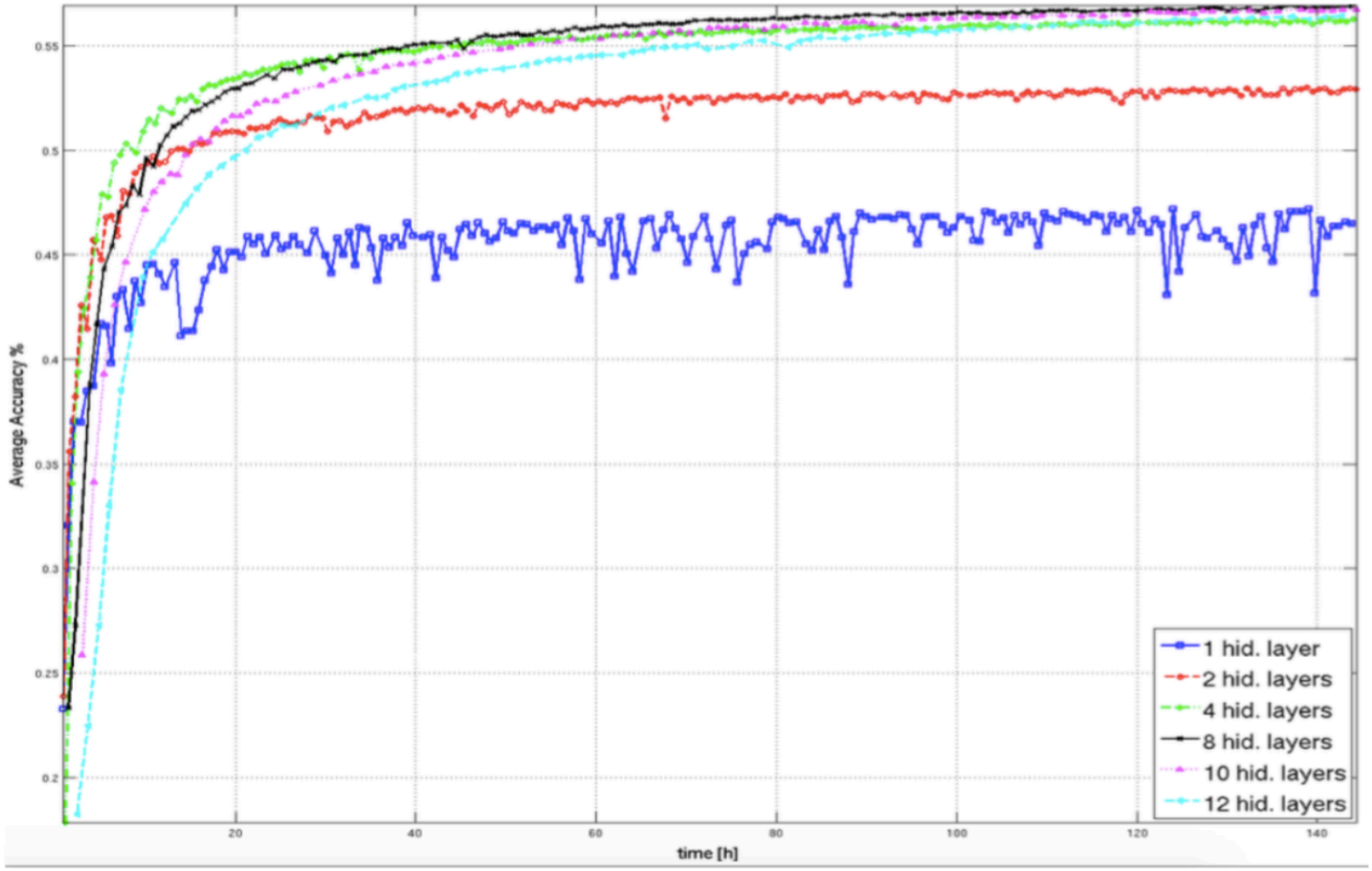




- Recall that each hidden unit "adds" a function to the overall function
- Increasing the number of hidden units will learn more and more complex function
- Very large K seems to overfit. Should we instead prefer small K?
- No! It is better to use large K and regularize well. Here is a reason/justification:
  - Simple NN with small K will have a few local optima, some of which may be bad
  - Complex NN with large K will have many local optimal, all equally good
  - Note: The above interesting behavior of NN has some theoretical justifications (won't discuss here)
- We can also use multiple hidden layers (each sufficiently large) and regularize well

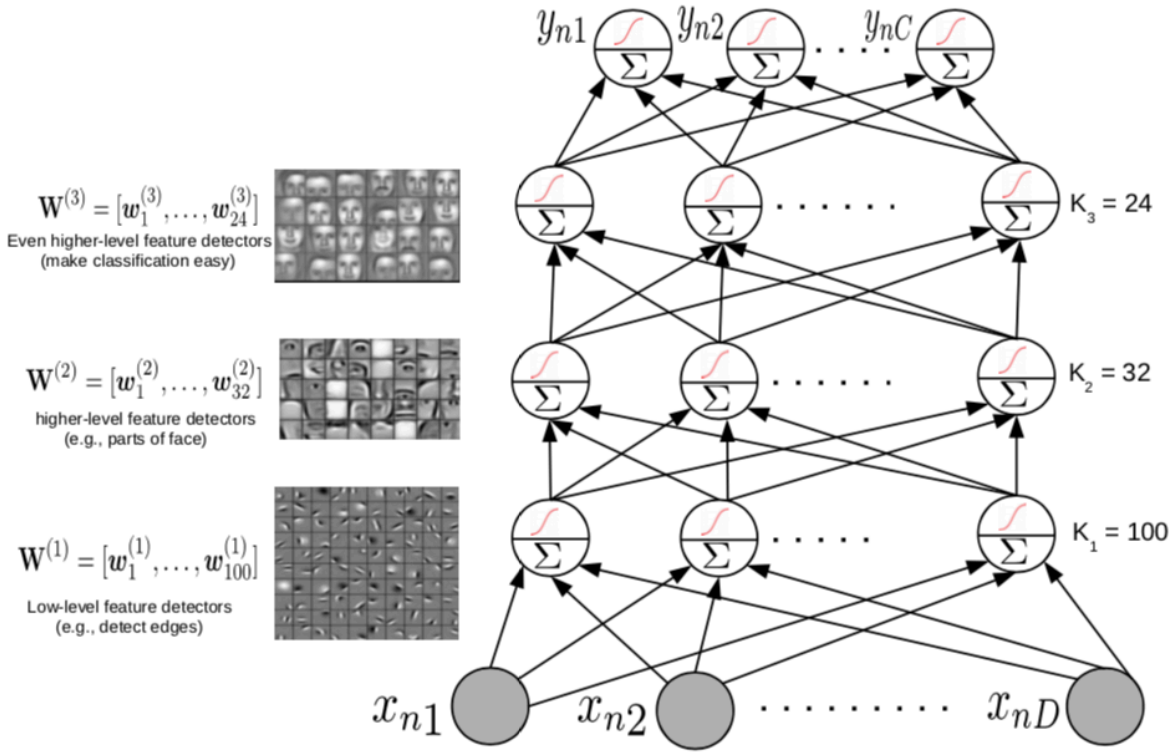
Effect of increasing the hidden layers

- Speech recognition task



Wide or deep?

- While very wide single hidden layer can approx. any function, often we prefer many hidden layers



- Higher layers help learn more directly useful/interpretable features (also useful for compressing data using a small number of features)

Convolutional Neural Networks

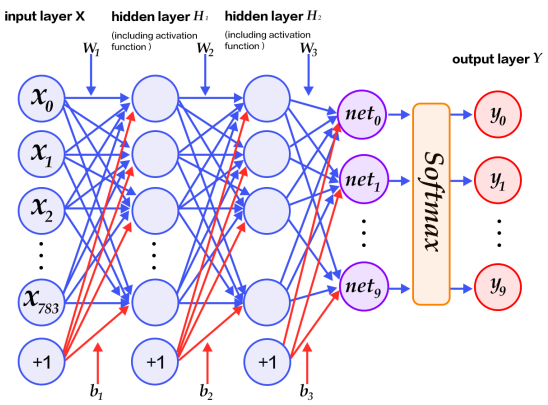
The material in the following slides wasn't really / fully covered during the lecture, but it should be relatively accessible (check also the notebook's section on CNNs examples).

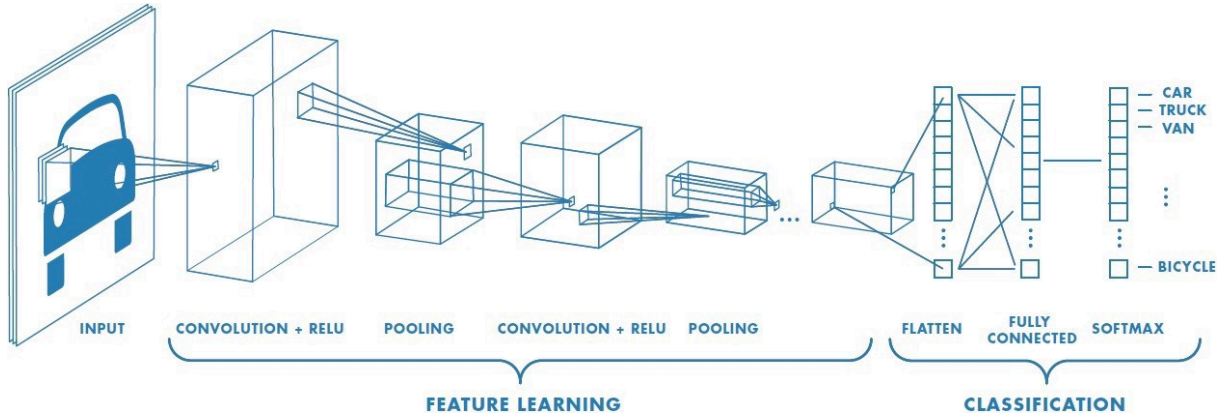
Core reasons behind trends and current successes of ML

- ✓ Massive amounts of **data** to learn from
- ✓ Many (+- feasible) ways to **label** the data

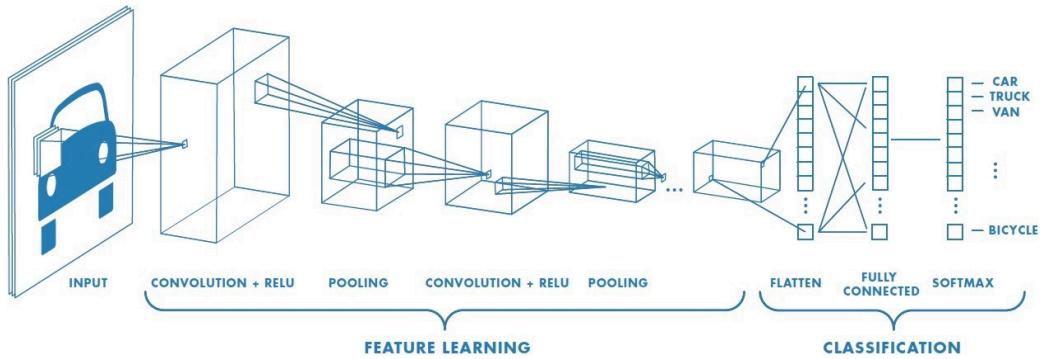
- ✓ Growth in computing power, ***parallel*** **GPUs** for matrix/tensor processing (hundred times faster than CPUs), that allows to crunch the available data

- ✓ **Network models** / function approximators (**CNNs**) that can be "effectively" trained over large datasets with BP techniques, with a lot of of parallelism, given that the inputs are *appropriate / complexity affordable*





Spatial coherence, locality, hierarchical dimensionality reduction



- Convolutional layers take advantage of **inherent properties** of n-dimensional signals such as images, speech, text, and their **spatial (and/or temporal) coherence**

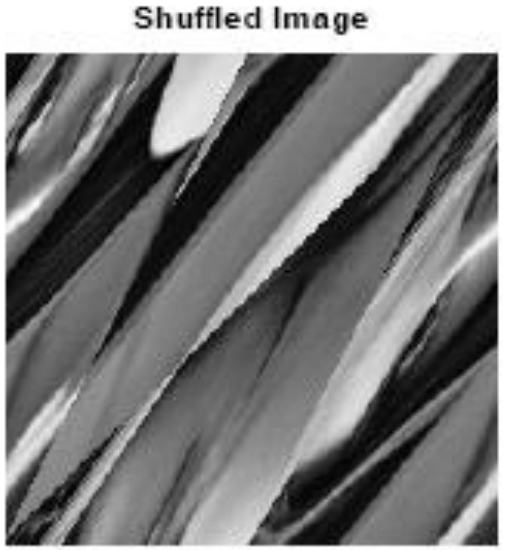
- Convolution is applied on **patches of adjacent pixels,** because adjacent pixels together are meaningful in an image
- **Local connectivity**: dramatic reduction in the number of operations needed to process an input image

- Pooling layers **further reduce the size of the internal representations by** downscaling them → Extract higher-order features

  - This is possible because in all layers retained network features are organized spatially like an image, and thus downscaling them makes sense, as reducing the size of the image

Fully-connected MLP are insensitive to order in their inputs

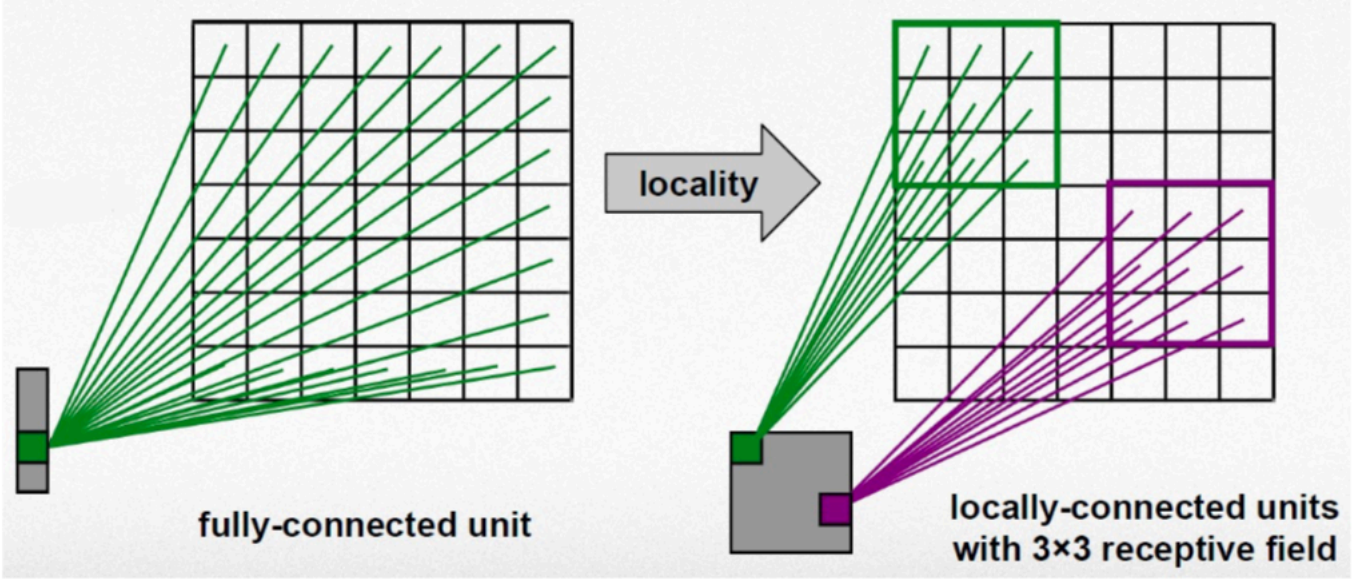
A

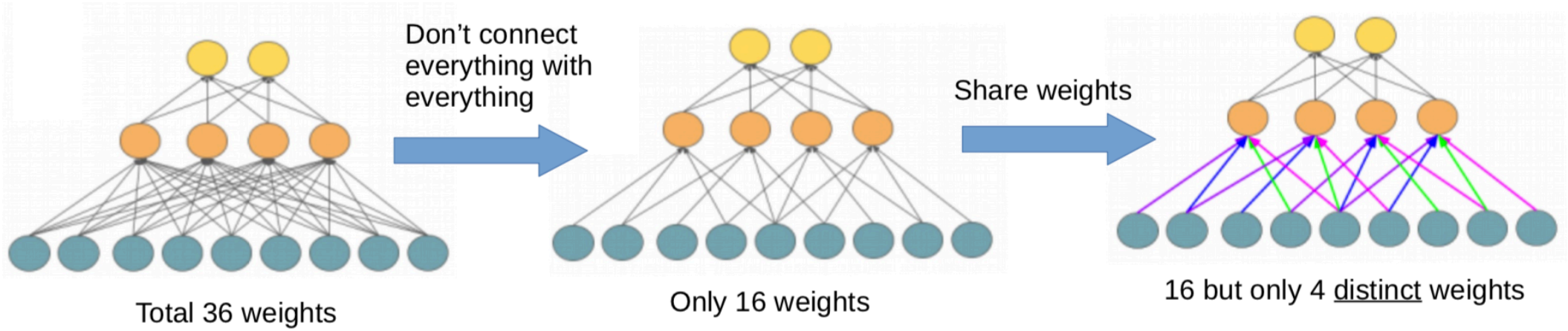


B

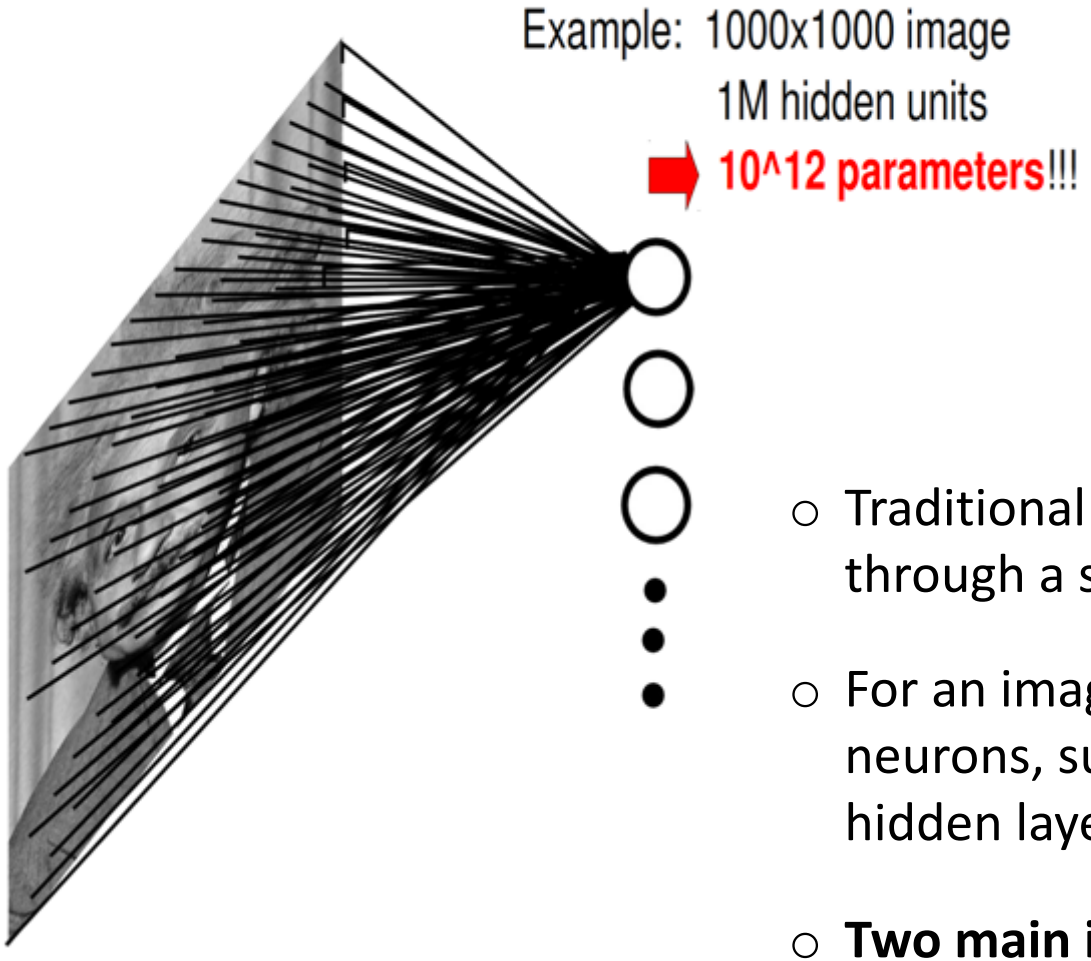
If all images are shuffled in the same way, the trained MLP would have the same performance

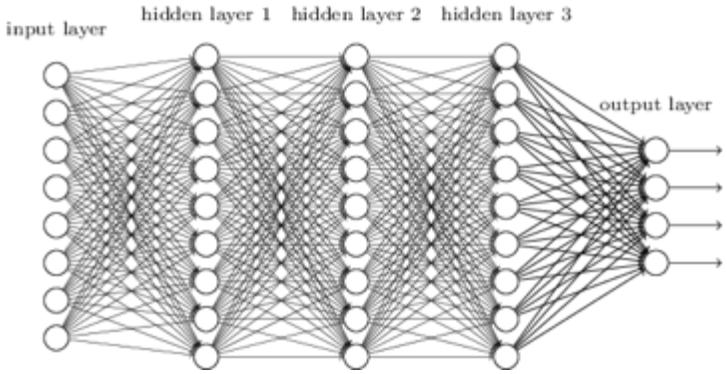
Spatial coherence, locality, hierarchical dimensionality reduction





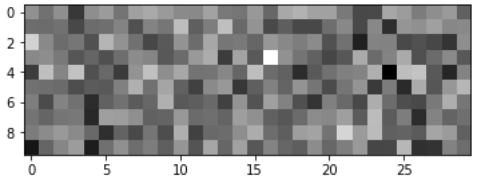
Beyond MLPs



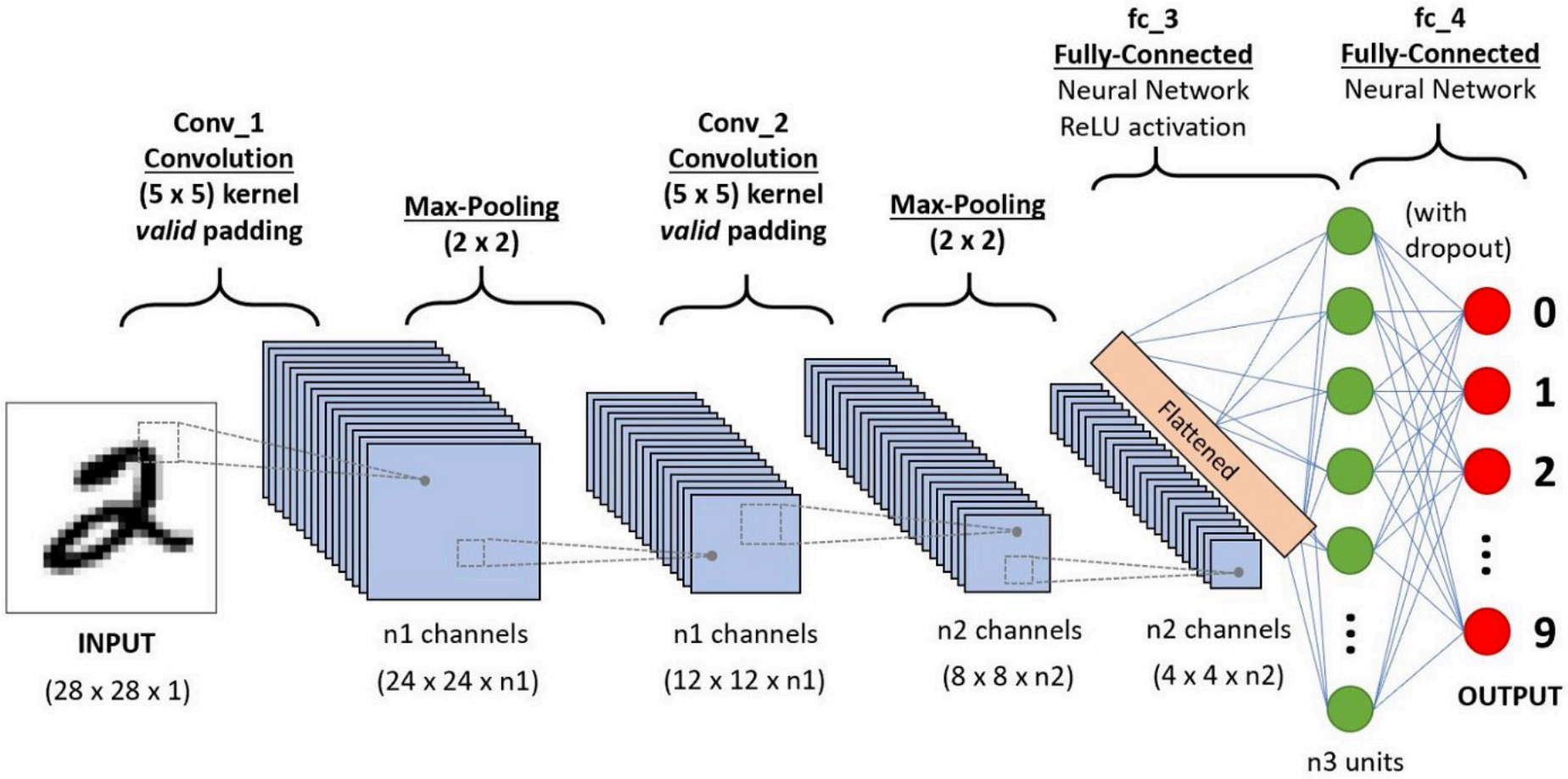


- Traditional MLPs receive as input a *single vector* and transforms it through a series of (fully connected) hidden layers
- For an image (32w, 32h, 3c), the input layer has 32×32×3=3072 neurons, such that a *single* fully-connected neuron in the first hidden layer would have 3072 weights …
- **Two main issues**: space-time complexity and lack of structure, locality of information
- Features extracted at each layer are pretty ... Random-looking!

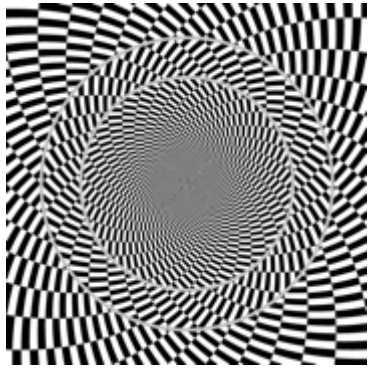
MNIST MLP



A Convolutional Neural Network (CNN) architecture



Images are “multi-dimensional”



Have a local *structure* and *correlations*

Have distinctive *features* in space and in frequency domains

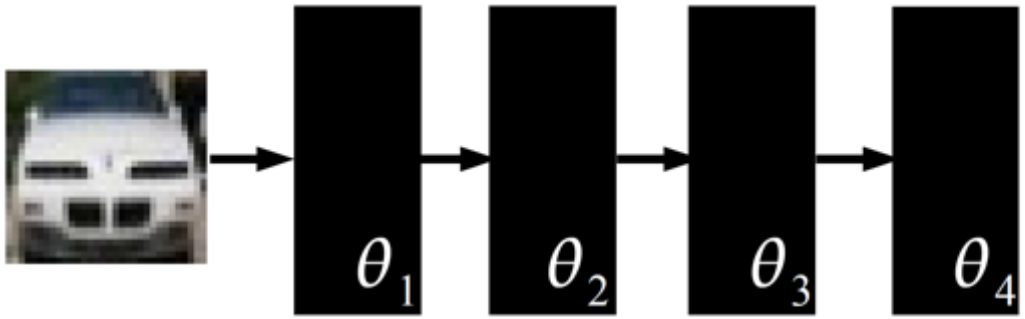
Different recipes

Solution #1: freeze first N-1 layer (engineer the features)
It makes it shallow!

"Classical"
Pattern recognition

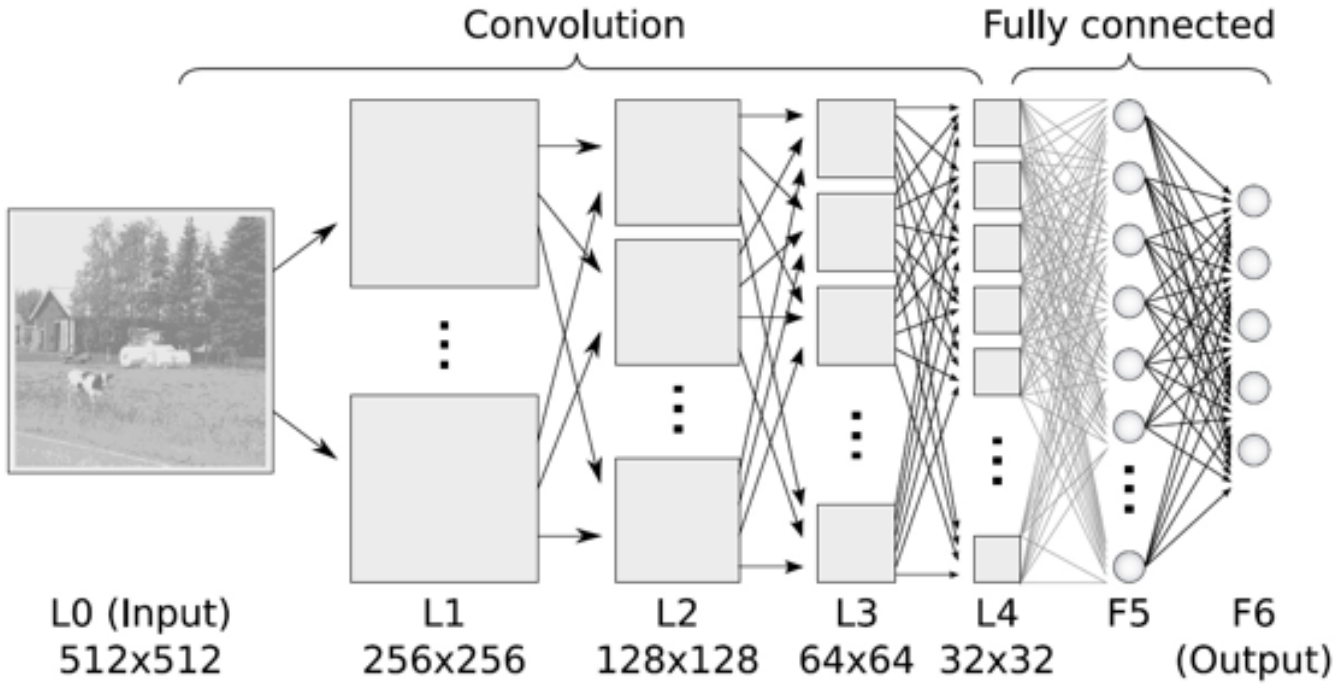
"Hot"
Convolutional
Neural Networks

Optimization is difficult: non-convex, non-linear system



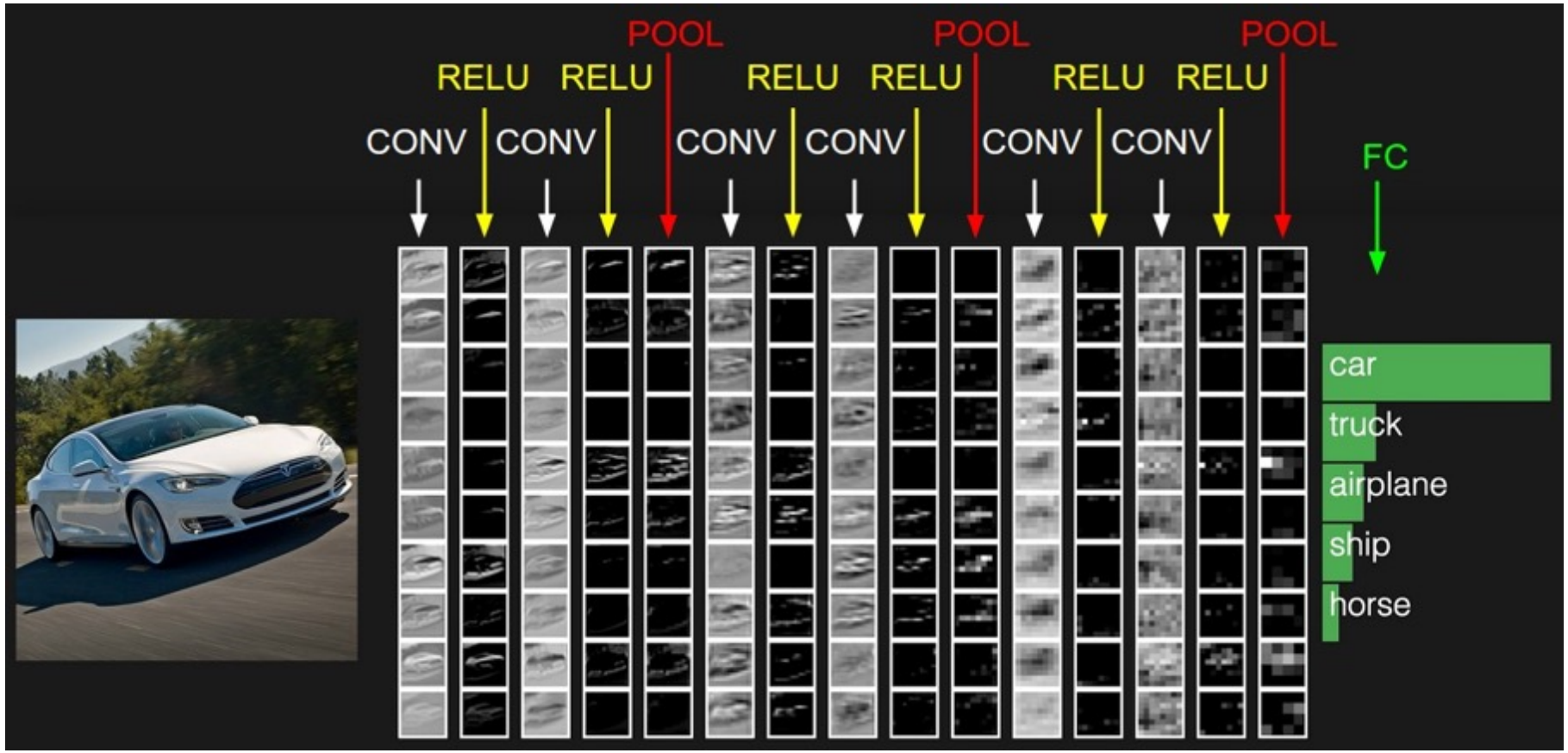
Solution #2: live with it!
It will converge to a local minimum.

Convolutional NNs



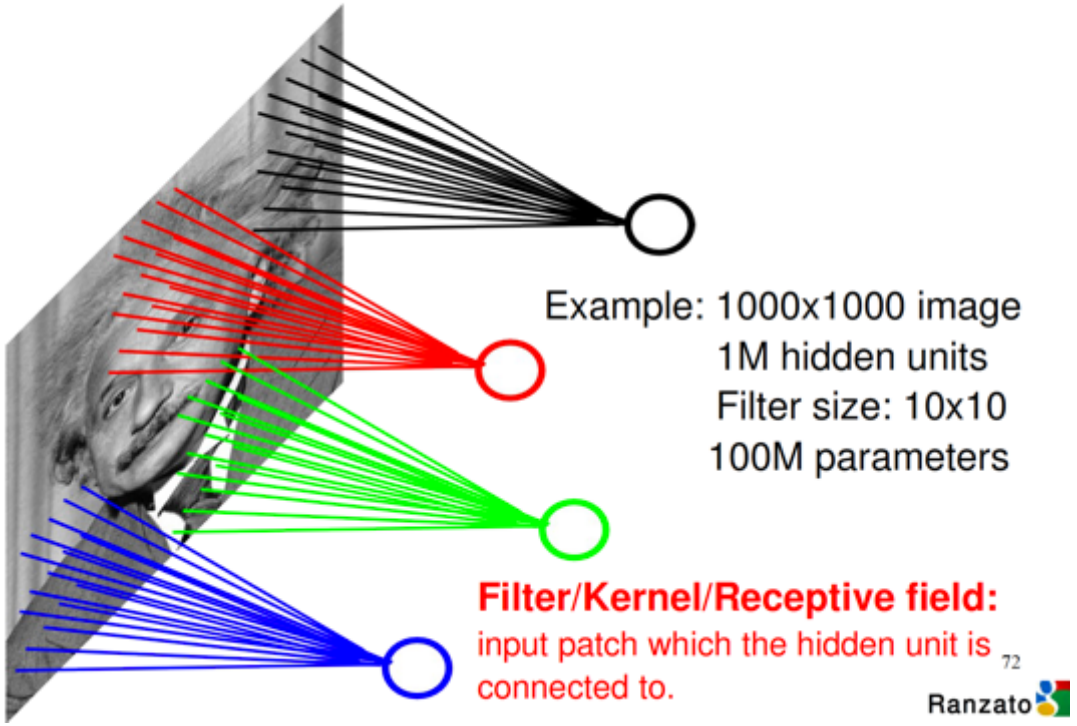
- Not anymore fully connected
- Locality of processing
- Weight sharing for parameter reduction
- Learn the parameters of multiple convolutional filter banks
- Compress to extract salient features and favor generalization

Convolutional NN

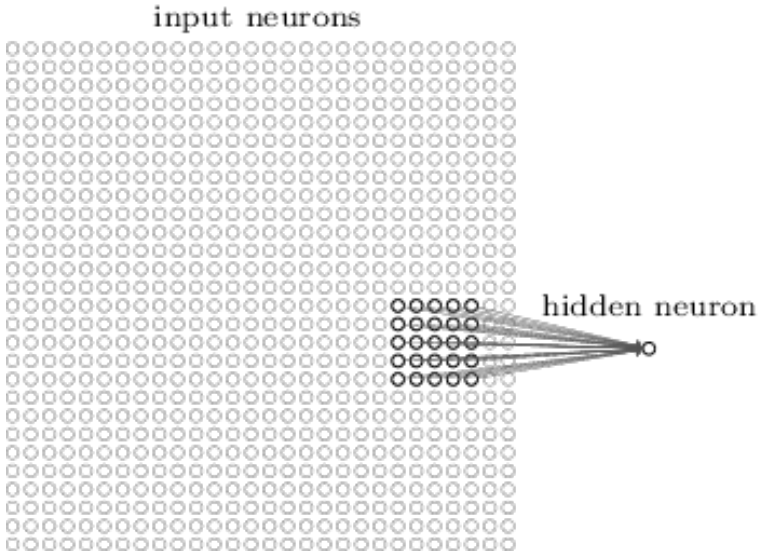


http://cs231n.github.io/

Locality of information: Receptive fields



28×28 = 784
input image

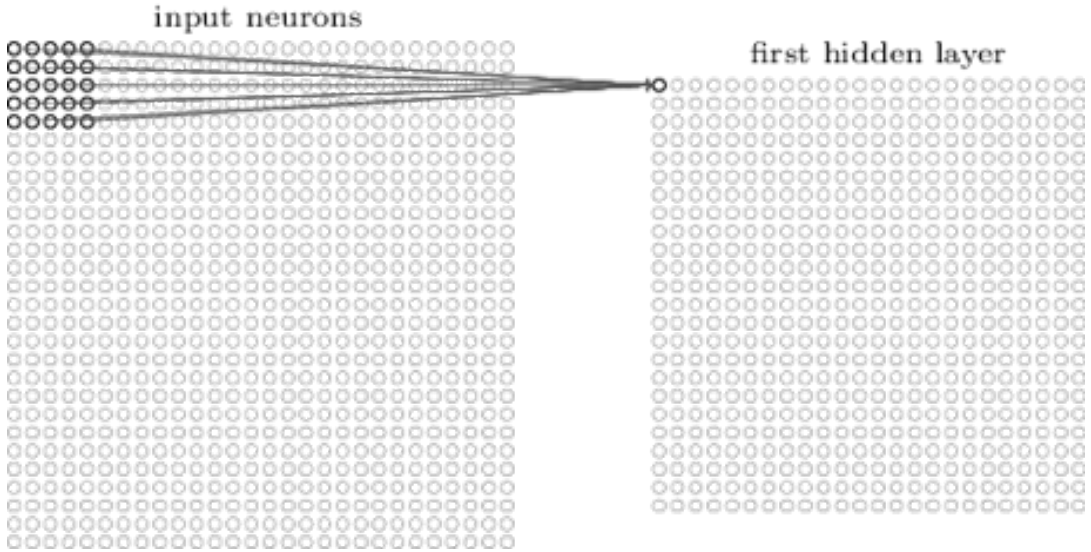


5×5 = 25
input pixels

How many neurons in the 1st hidden layer?

(Filter) Stride

Let's *slide* the 5×5 mask over all input pixels

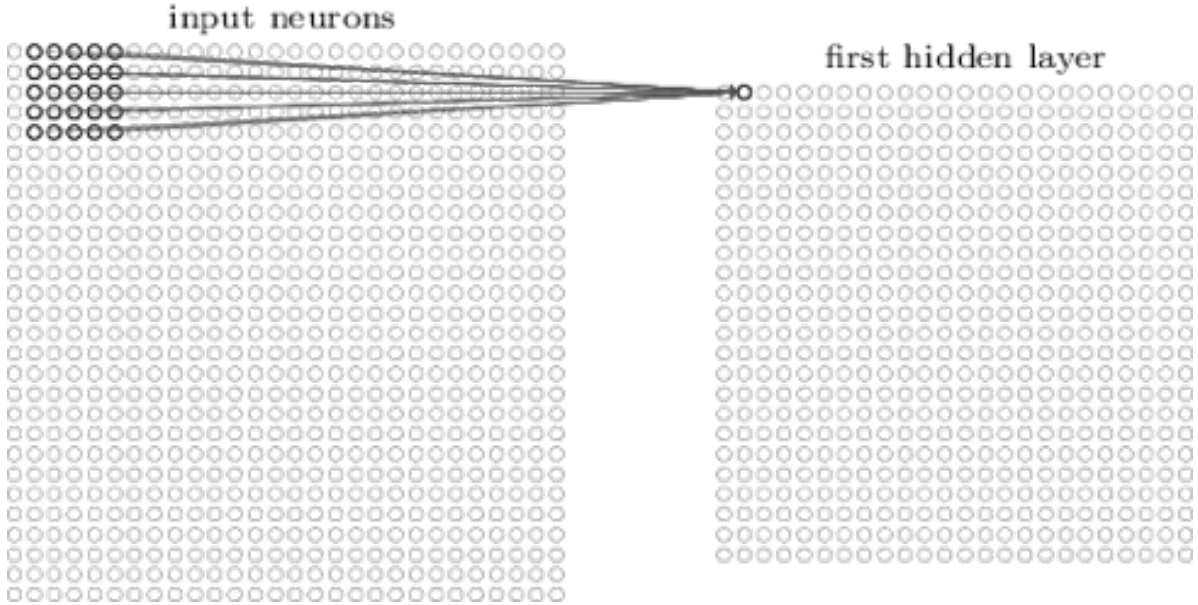


Stride length = 1

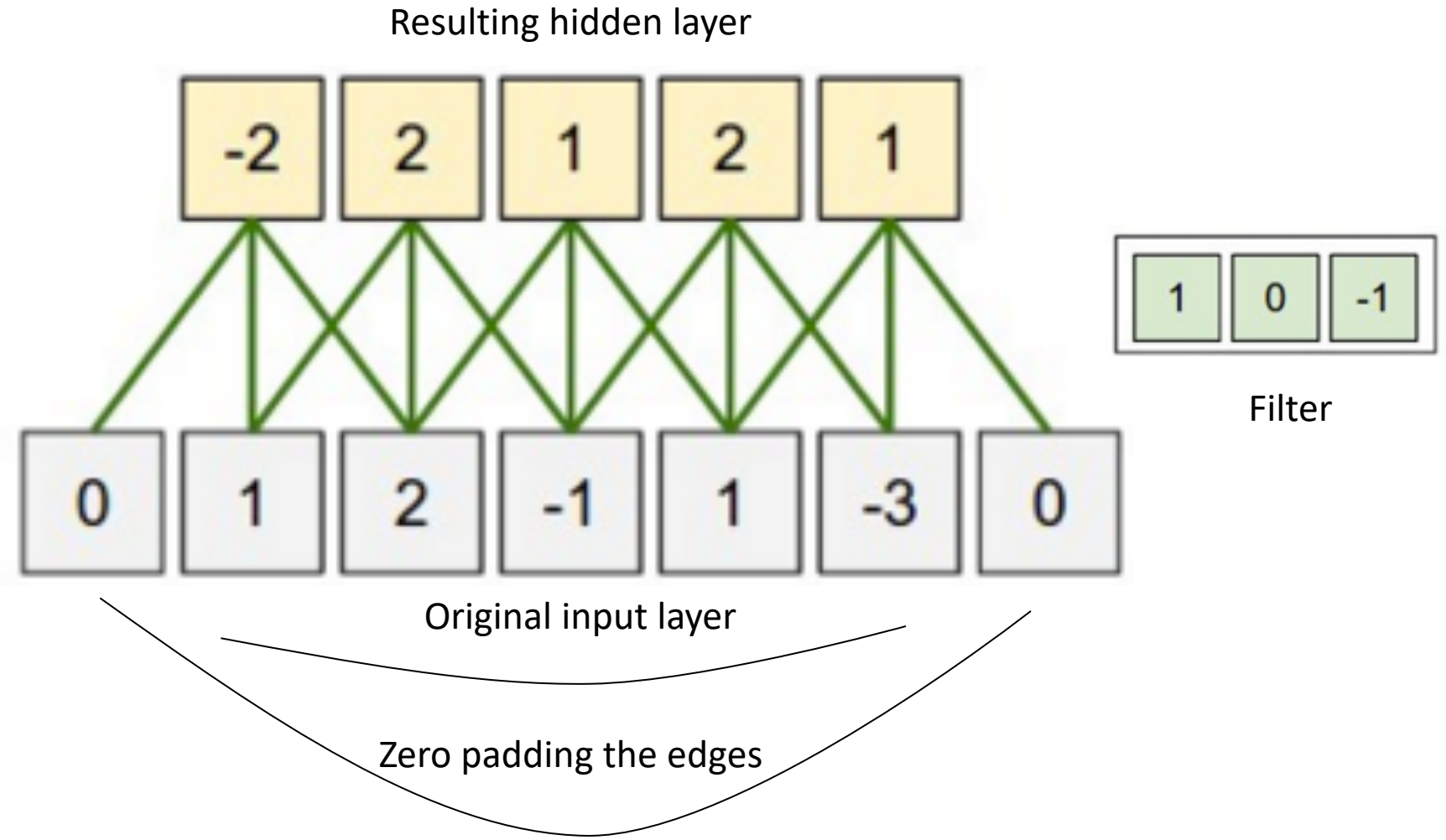
Any stride can be used … with some precautions

How many neurons in the 1st hidden layer?

24×24

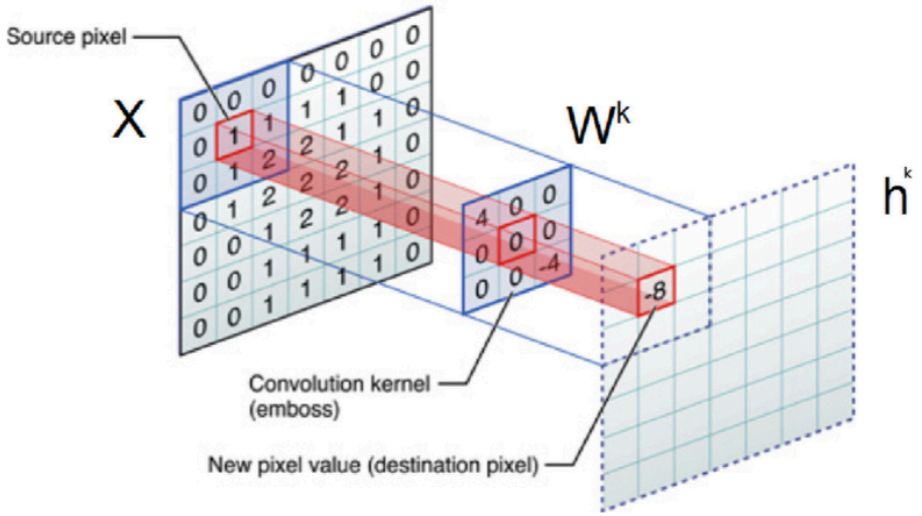


(Filter) Padding



Convolution image filter

- An operation that captures local (e.g., spatial) properties of a signal



- Mathematically, the operation is defined as

$$h_{ij}^k = g((W^k * \mathbf{X})_{ij} + b_k)$$

where W^k is a filter, $*$ is the convolution operator, and g is a nonlinearity

- Usually several filters $\{W^k\}_{k=1}^K$ are applied (each will produce a separate "feature map"). These filters have to be learned (these are the weights of the NN)

Shared weights

- What is the precise relationship between the neurons in the receptive field and that in the hidden layer?
- What is the *activation value* of the hidden layer neuron?

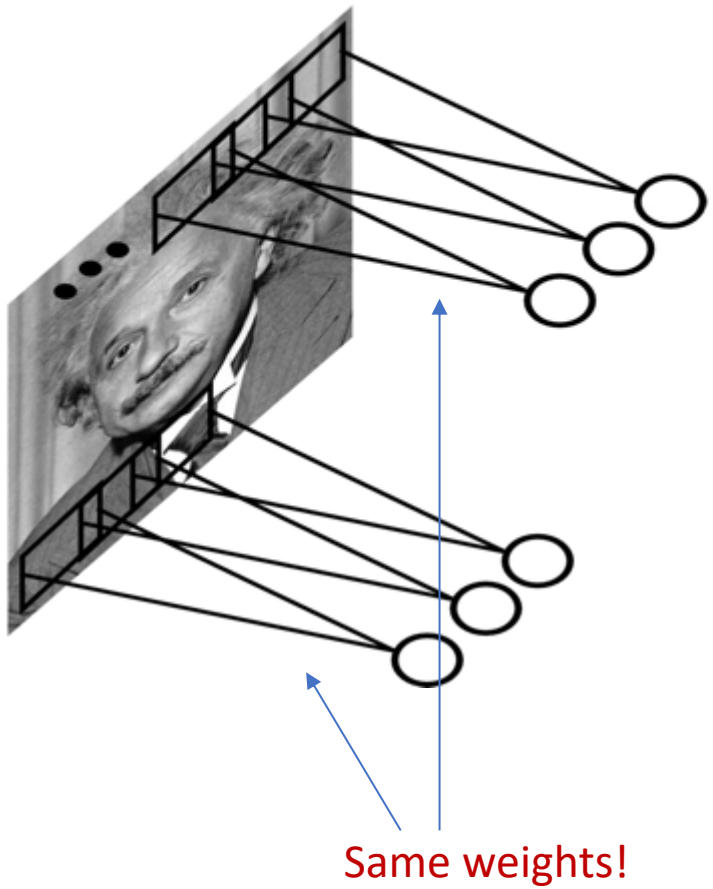
$$\sigma\left(b + \sum_{l=0}^{4}\sum_{m=0}^{4} w_{l,m} a_{j+l,k+m}\right)$$

- σ is the selected activation function, a are the activation values of the neurons in the receptive field
- The *same* weights w and bias b are used for each of the 24×24 hidden neurons

Feature map

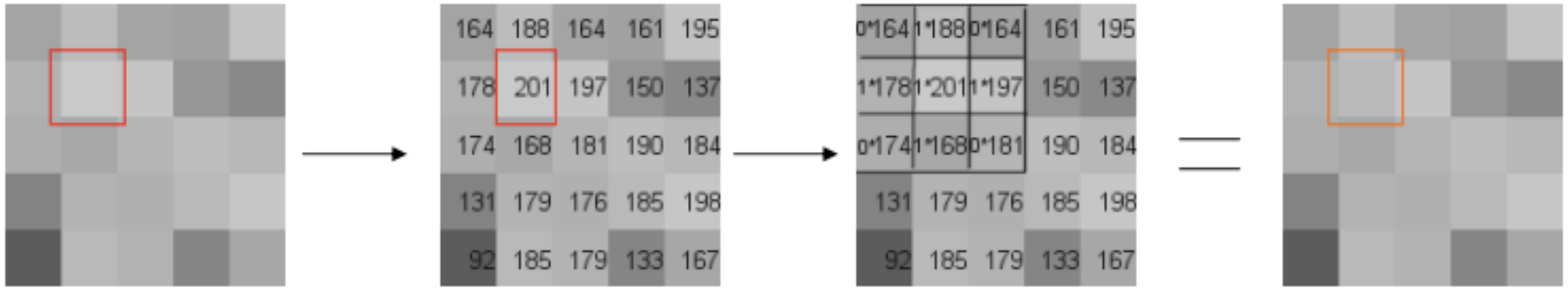
- All the neurons in the first hidden layer detect exactly the same **feature**, just at different locations in the input image.
- **“Feature”:** the kind of input pattern (e.g., a local edge) that determine the neuron to ”fire” or, more in general, produce a certain response level
- Why this makes sense? Suppose the weights and bias are (learned) such that the hidden neuron can pick out a vertical edge in a particular local receptive field. That ability is also likely to be useful at other places in the image. And so it is useful to apply the same feature detector everywhere in the image.

Feature map



- The map from the input layer to the hidden layer is therefore a **feature map:** all nodes detect the same feature in different parts of the image
- The map is defined by the shared weights and bias
- The shared map is the result of the application of **convolutional filter** (defined by weights and bias)

Convolution image filter



Original image

Image with color values placed over it

Image with 3x3 kernel placed over it

Output image

164	188	164
178	201	197
174	168	181

Color values

×

0	1	0
1	1	1
0	1	0

Kernel

Divided by the sum of the kernel

932\5 = new pixel color

Convolution image filter

1	1	1
1	1	1
1	1	1

Unweighted 3x3 smoothing kernel

0	1	0
1	4	1
0	1	0

Weighted 3x3 smoothing kernel with Gaussian blur

0	-1	0
-1	5	-1
0	-1	0

Kernel to make image sharper

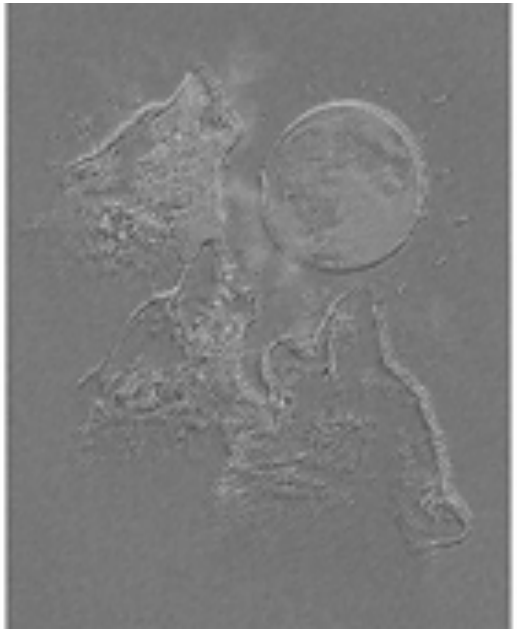
-1	-1	-1
-1	9	-1
-1	-1	-1

Intensified sharper image

Gaussian Blur

Sharpened image

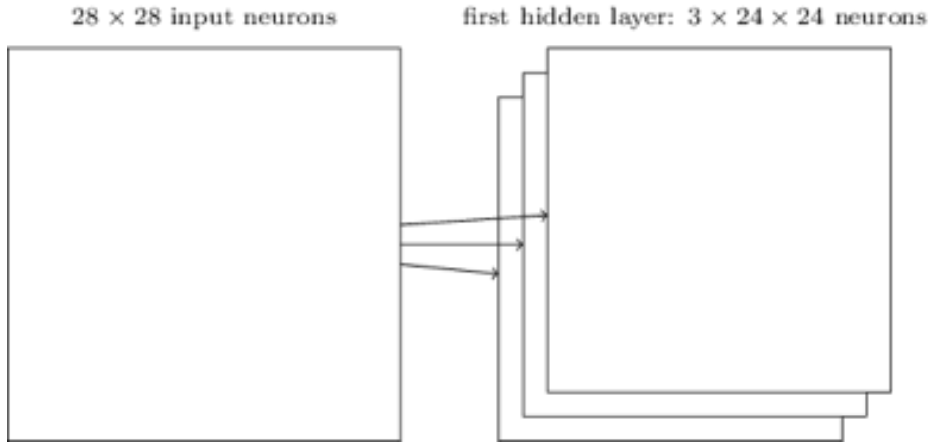
Convolution image filter


Filter banks

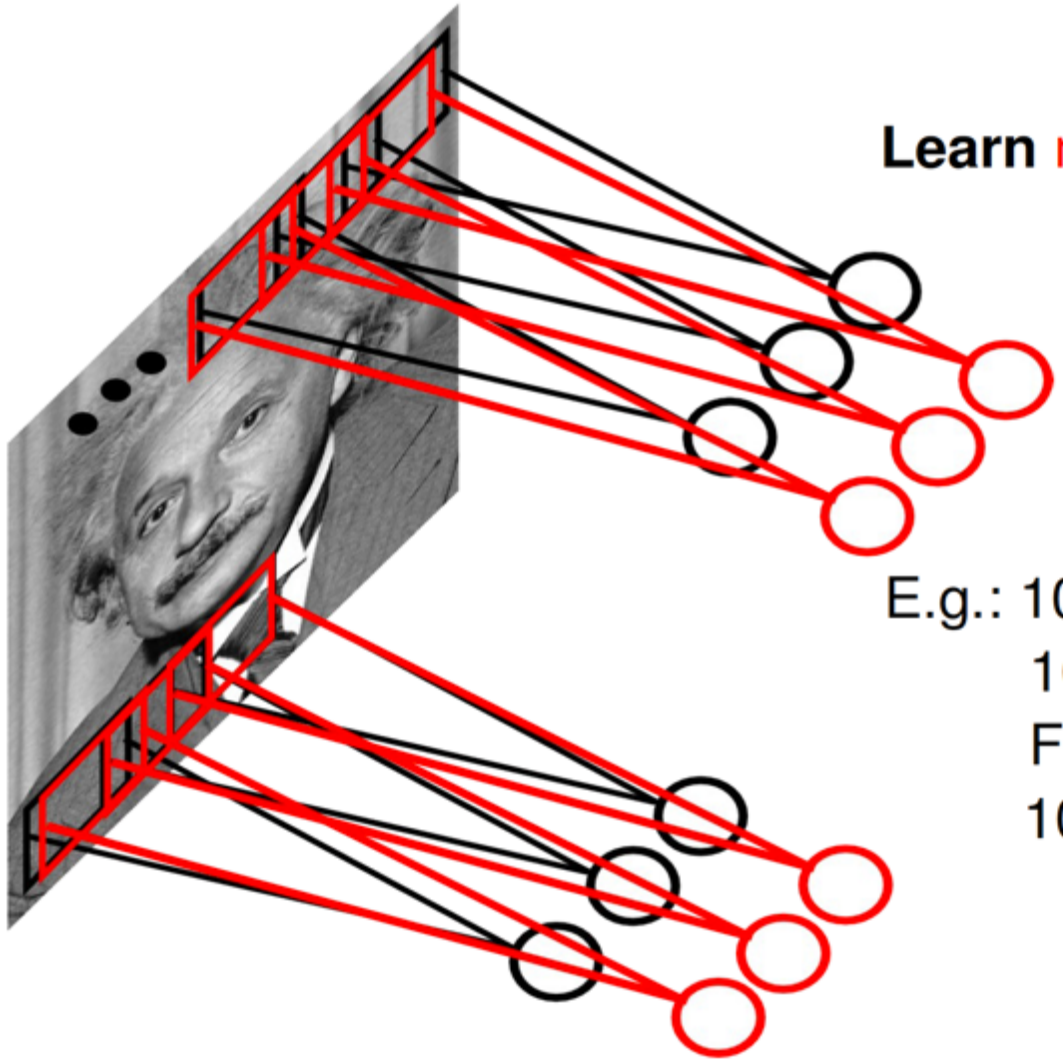
Why only one filter? (one feature map)

- ✓ At the *i*-th hidden layer *n* filters can be active in parallel
- ✓ A **bank of convolutional filters**, each learning a *different* feature (different weights and bias)



- 3 feature maps, each defined by a set of 5×5 shared weights and one bias
- The result is that the network can detect 3 different kinds of features, with each feature being detectable across the entire image.

Filter banks



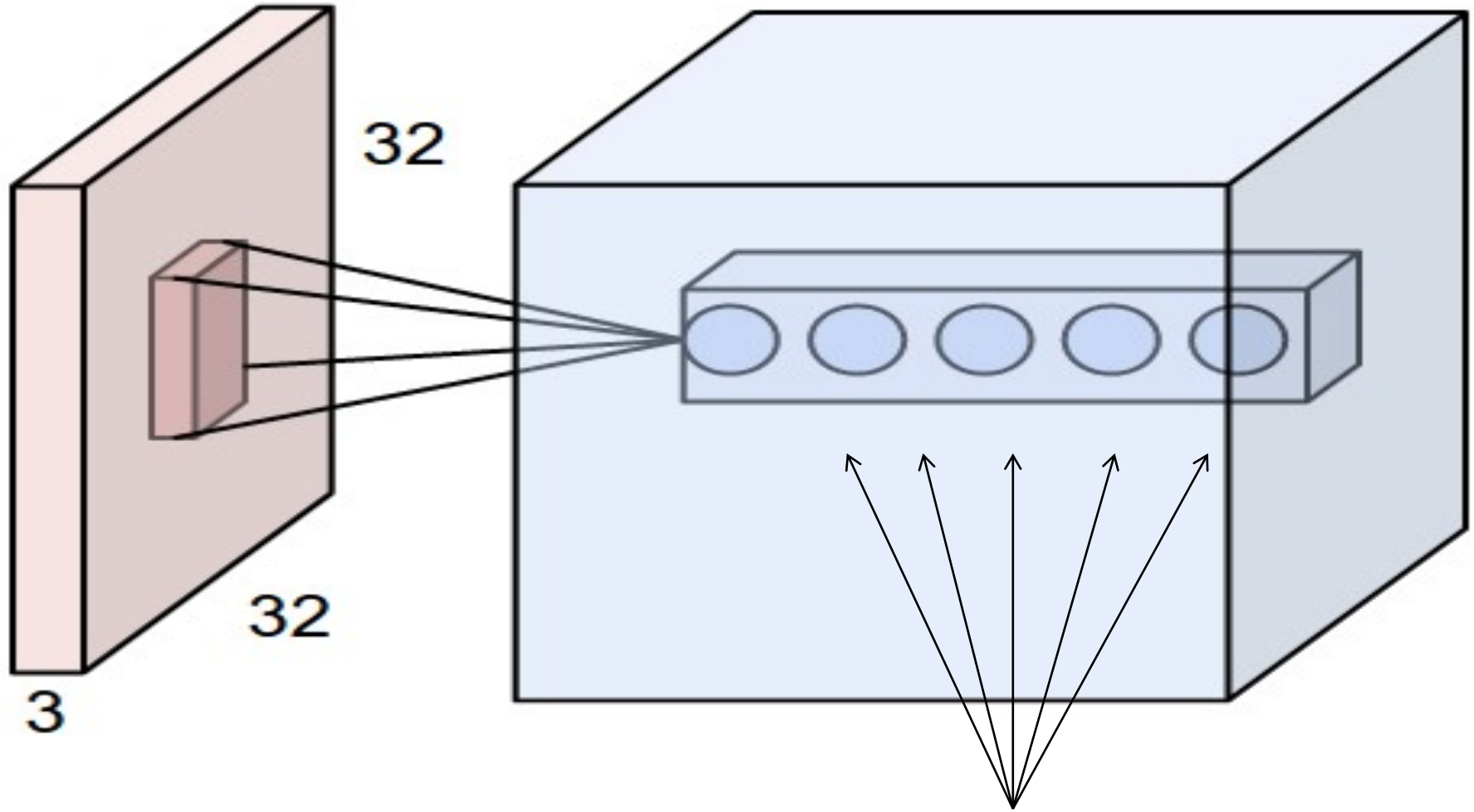
Learn multiple filters.

E.g.: 1000x1000 image
100 Filters
Filter size: 10x10
10K parameters

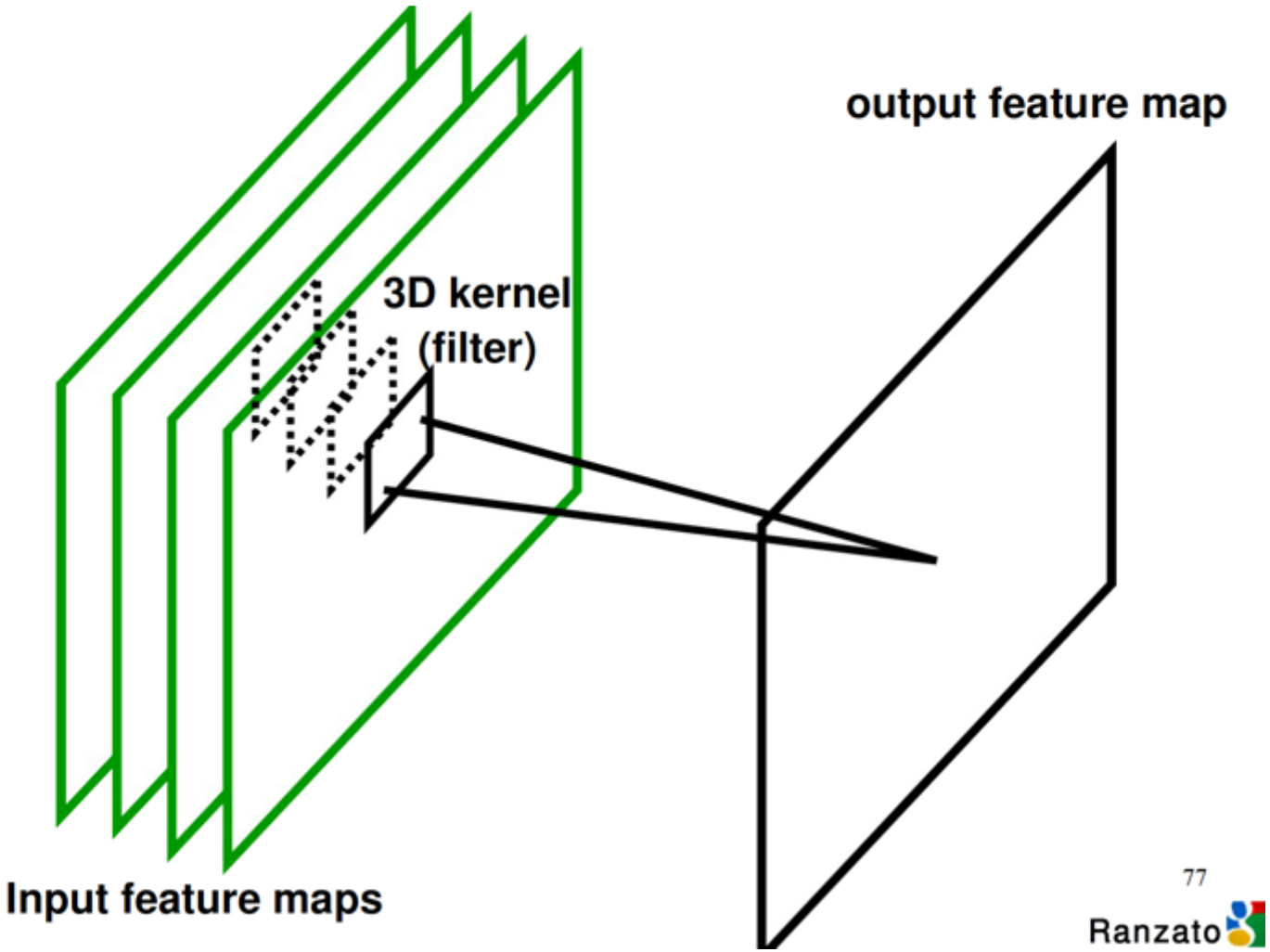
75

Ranzato

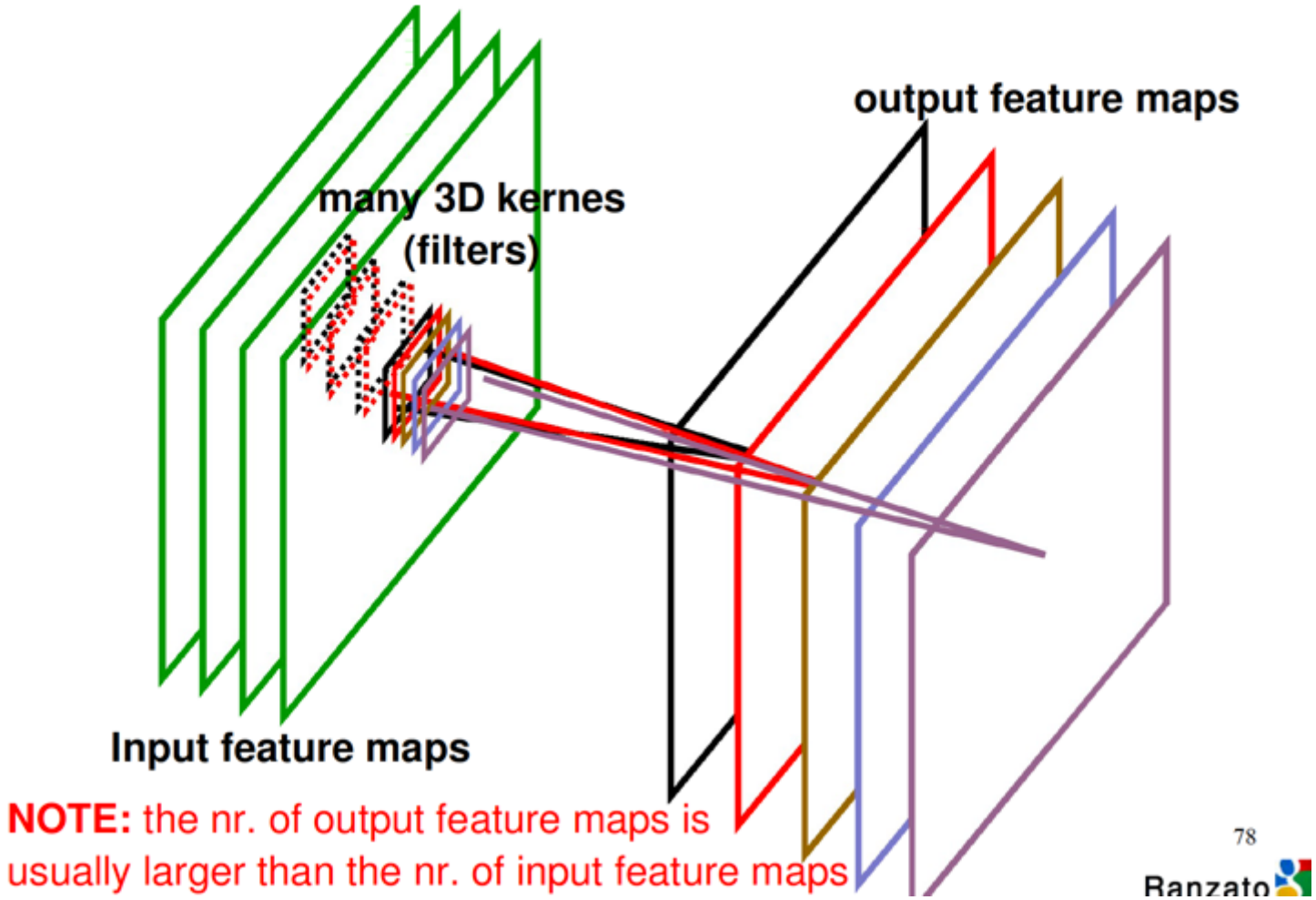
Volumes and Depths



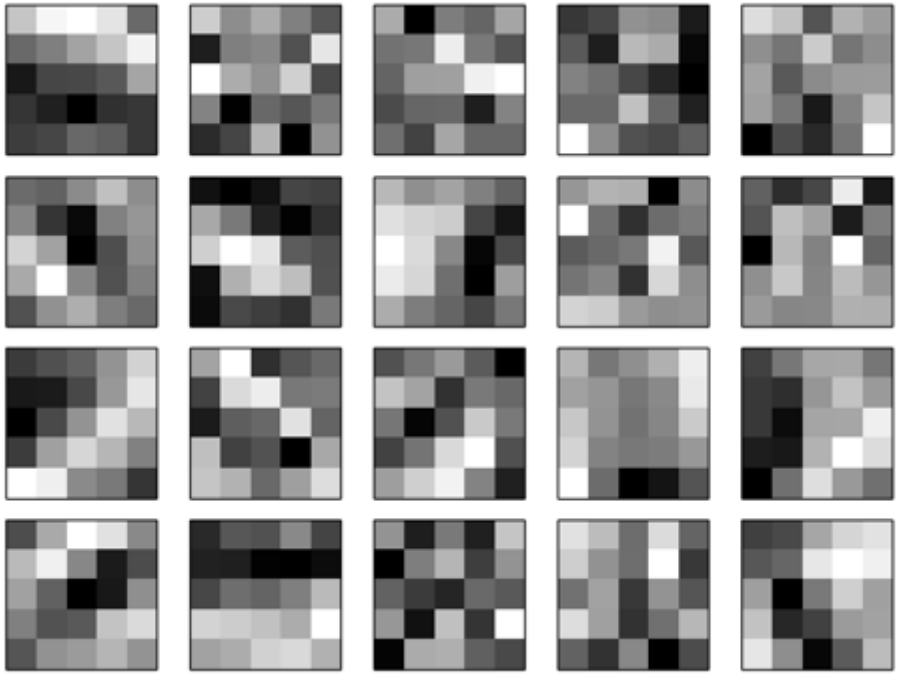
Multiple feature maps



Multiple feature maps

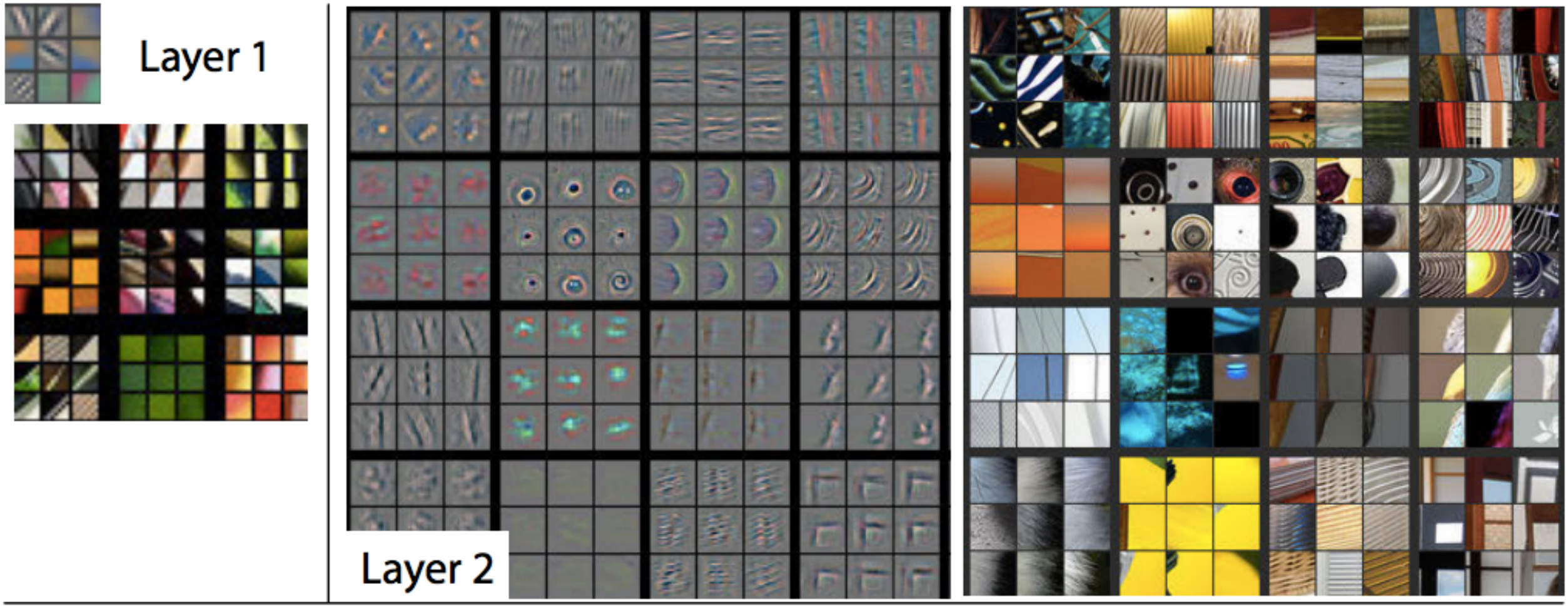


Character recognition example



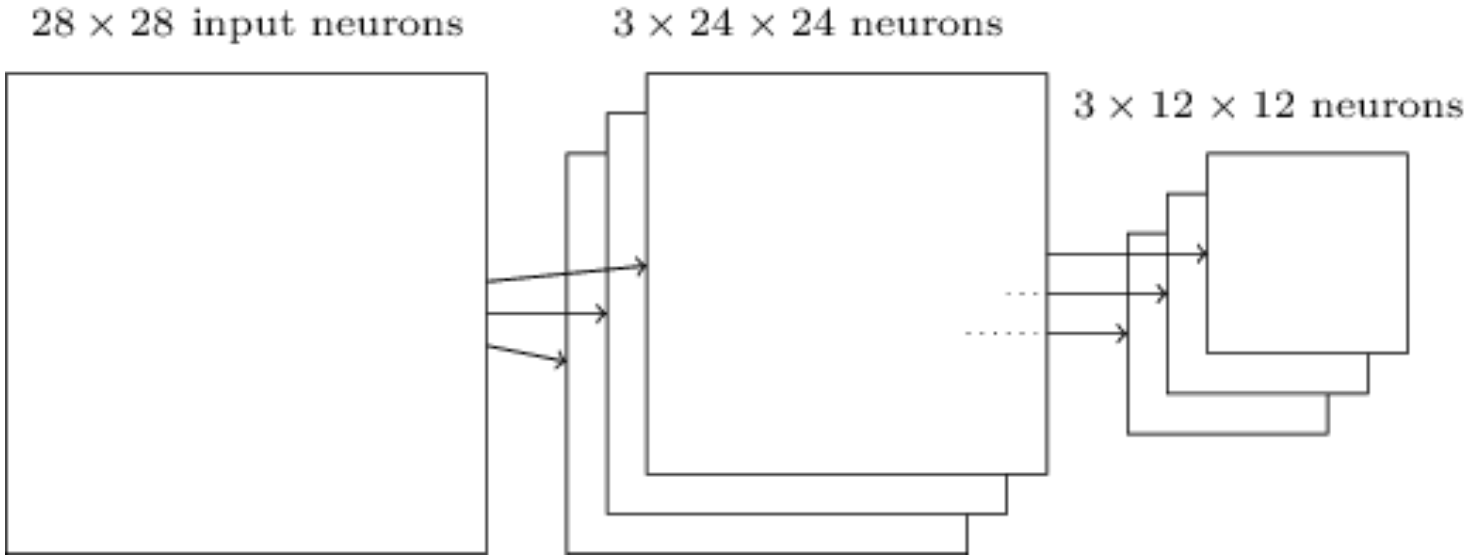
Darker blocks mean a larger weight, so the feature map responds more to the corresponding input pixels.
Some spatial correlations are “there”

A more exciting example

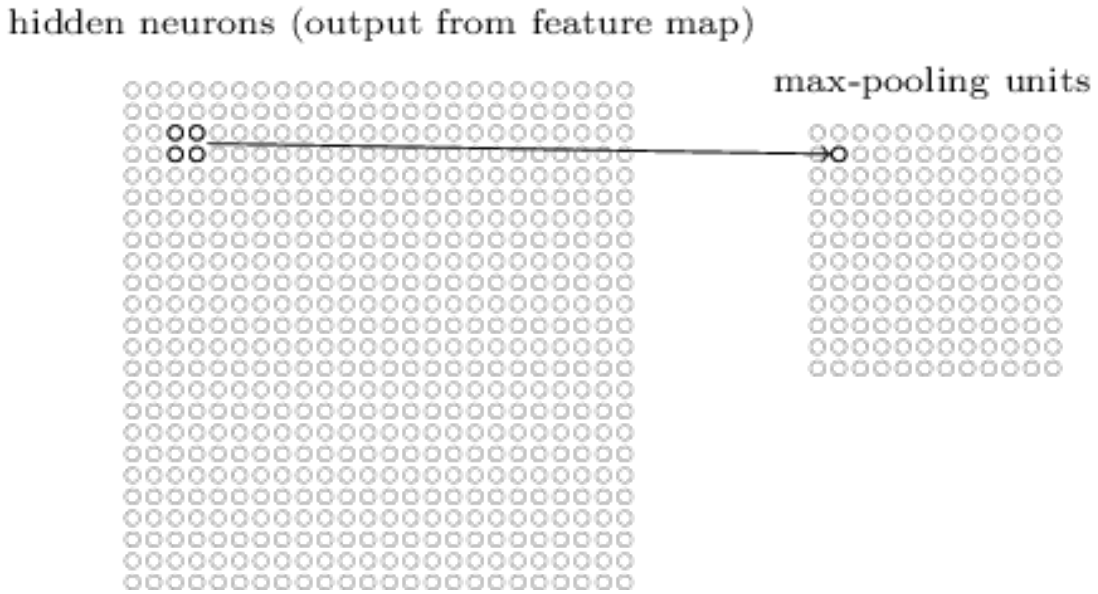


Pooling layers

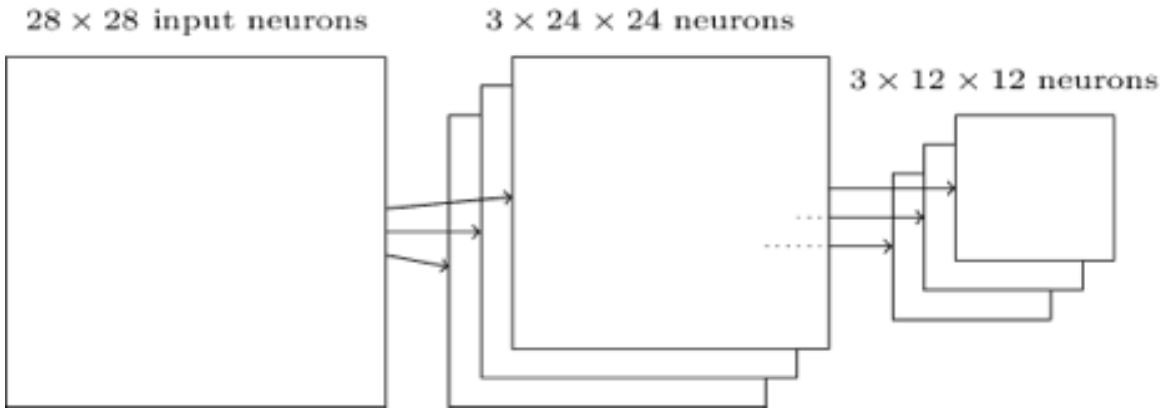
- **Pooling layers** are usually used immediately after convolutional layers.
- Pooling layers simplify / subsample / compress the information in the output from the convolutional layer
- A pooling layer takes each feature map output from the convolutional layer and **prepares a condensed feature map**



Pooling layers



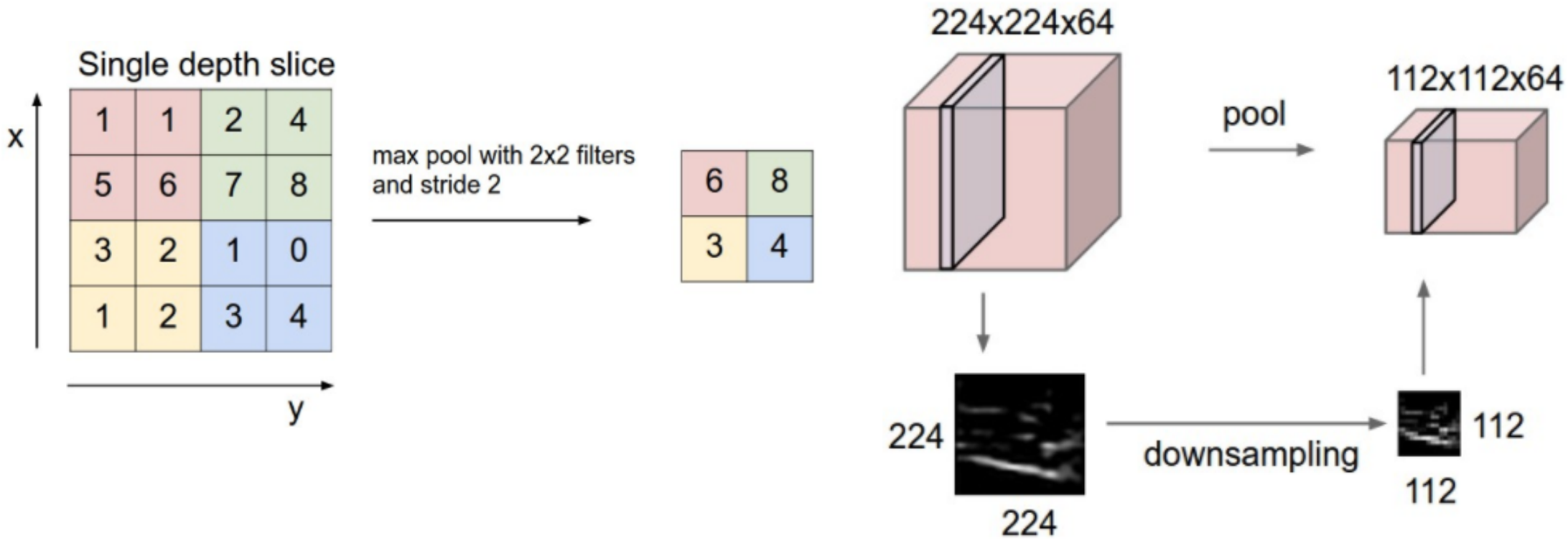
Each neuron in the pooling layer summarizes a region of $n \times n$ neurons in the previous hidden layer, which results in **subsampling**



Max-Pooling

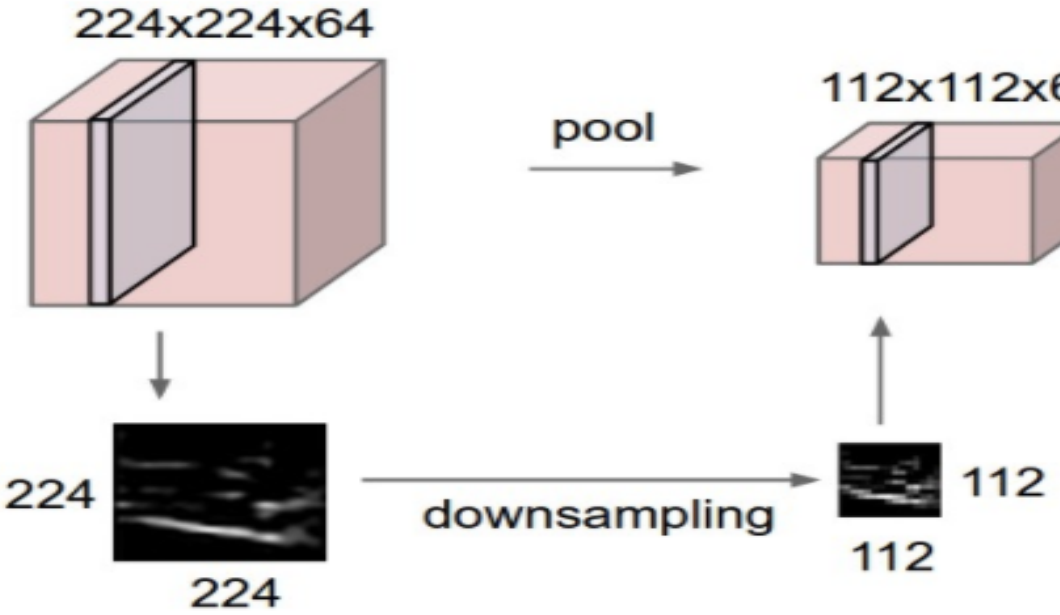
How to do pooling?

Max-pooling: a pooling unit simply outputs the *maximum activation* in the input region

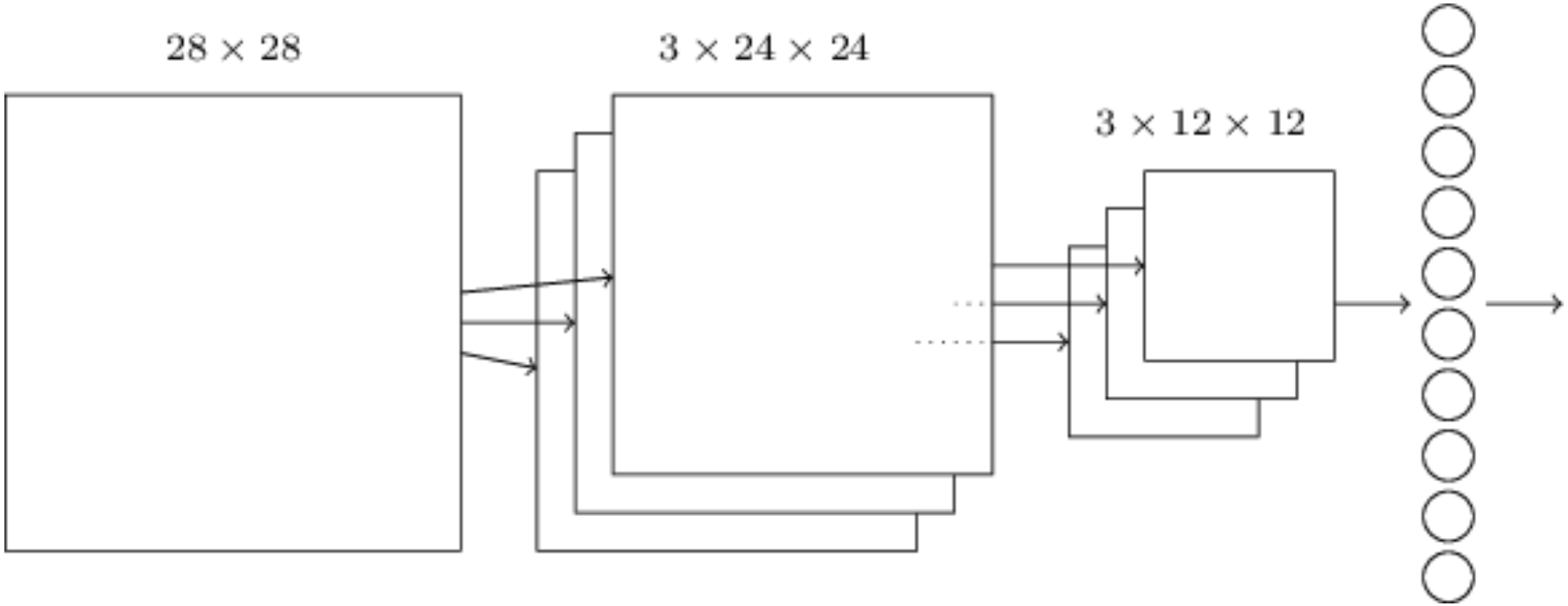


Max-Pooling

- Max-pooling as a way for the network to ask whether a given feature is found anywhere in a region of the image. It then **throws away the exact positional information**.
- Once a feature has been found, its exact location isn't as important as its rough location relative to other features.
- A big benefit is that there are many fewer pooled features, and so this helps **reduce the number of parameters needed in later layers**.



Putting all together



- The final, output layer is a **fully connected** one
- The transfer function can be a *soft-max function*, to probabilistically weight each possible output (e.g., for a classification task)

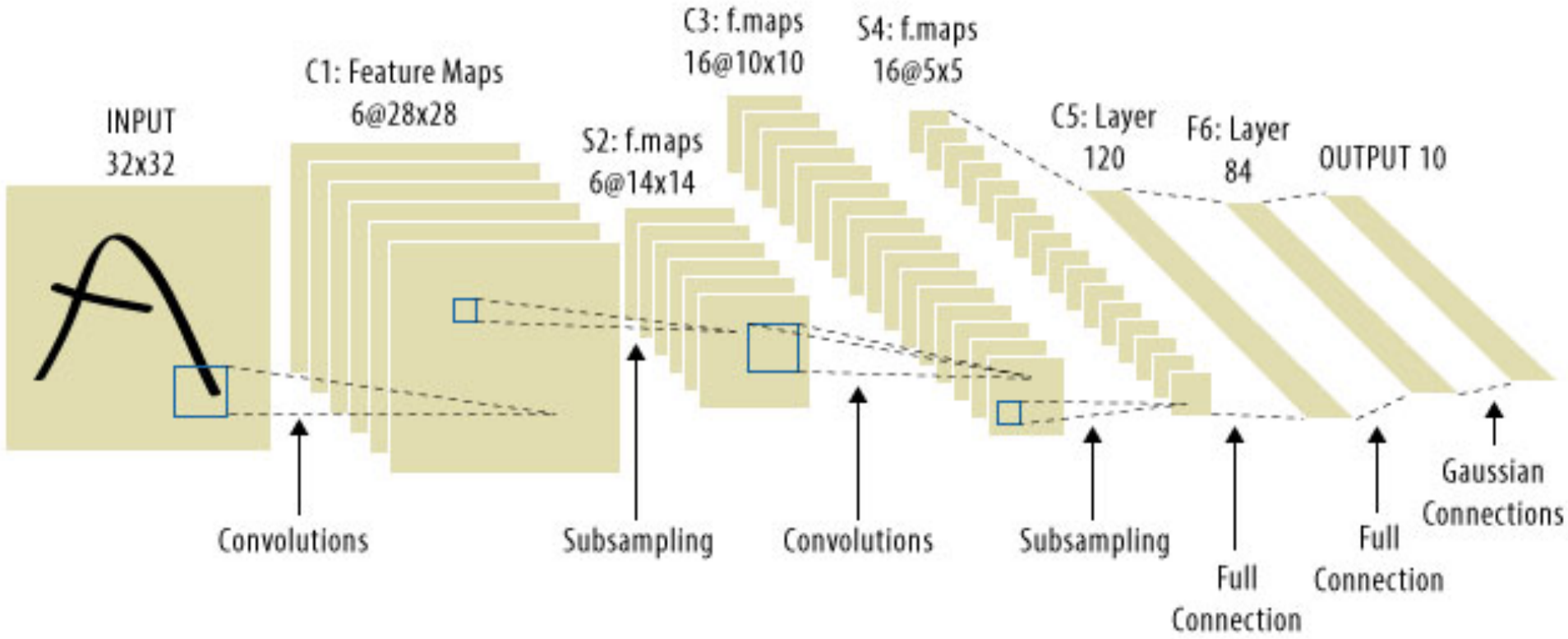
Soft-Max function (Multi-class logistic regression!)

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

- The **soft-max function** σ "squashes" a *K*-dimensional real-valued vector **z** to a *K*-dimensional [0,1] normalized vector
- In the final, fully connected layer, σ can be used to express the probability of the *j*-th component of the output ***y*** (e.g, the probability that the digit in the image sample **x** is "7")

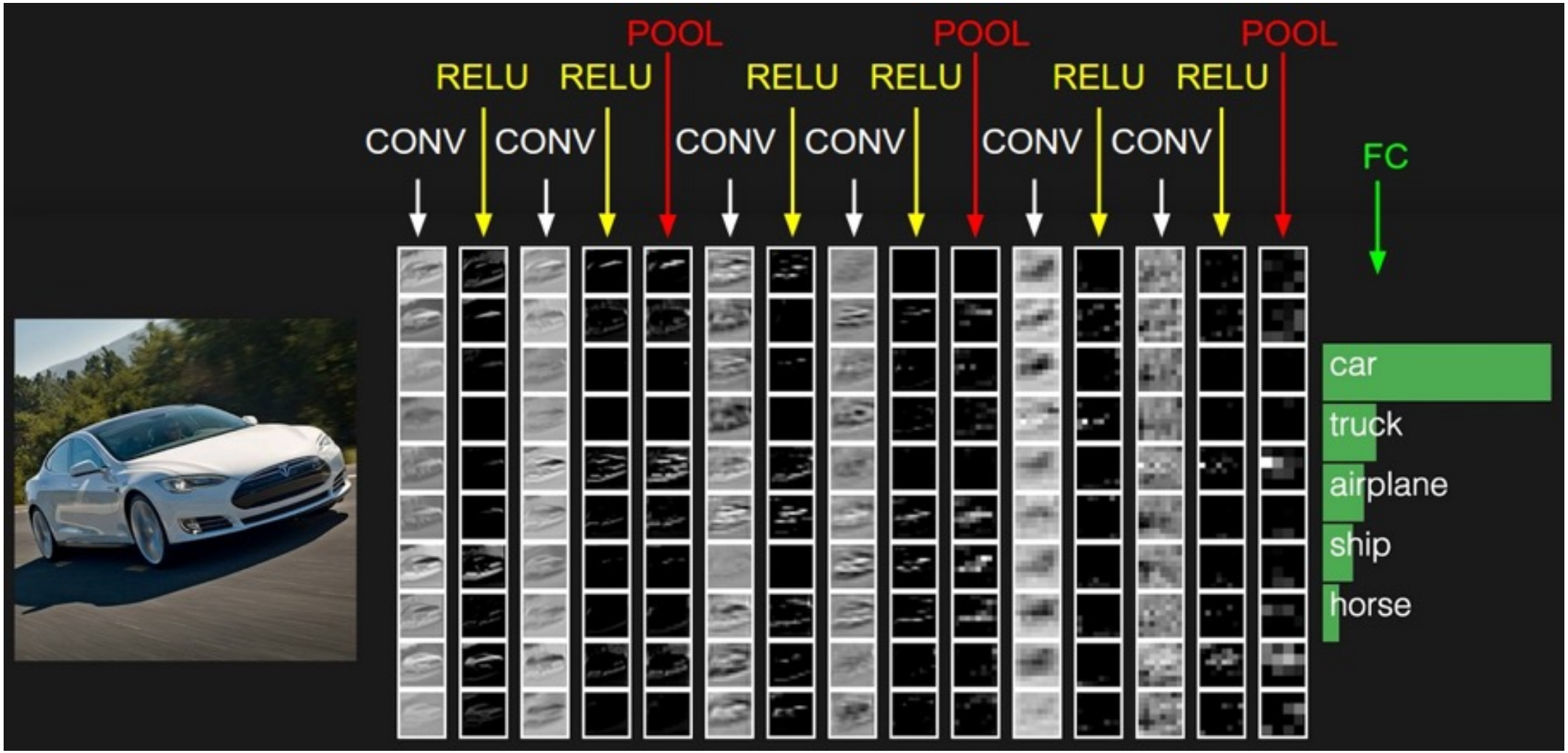
$$P(y = j|\mathbf{x}) = \frac{e^{\mathbf{x}^\mathsf{T}\mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\mathsf{T}\mathbf{w}_k}}$$

Convolutional NN

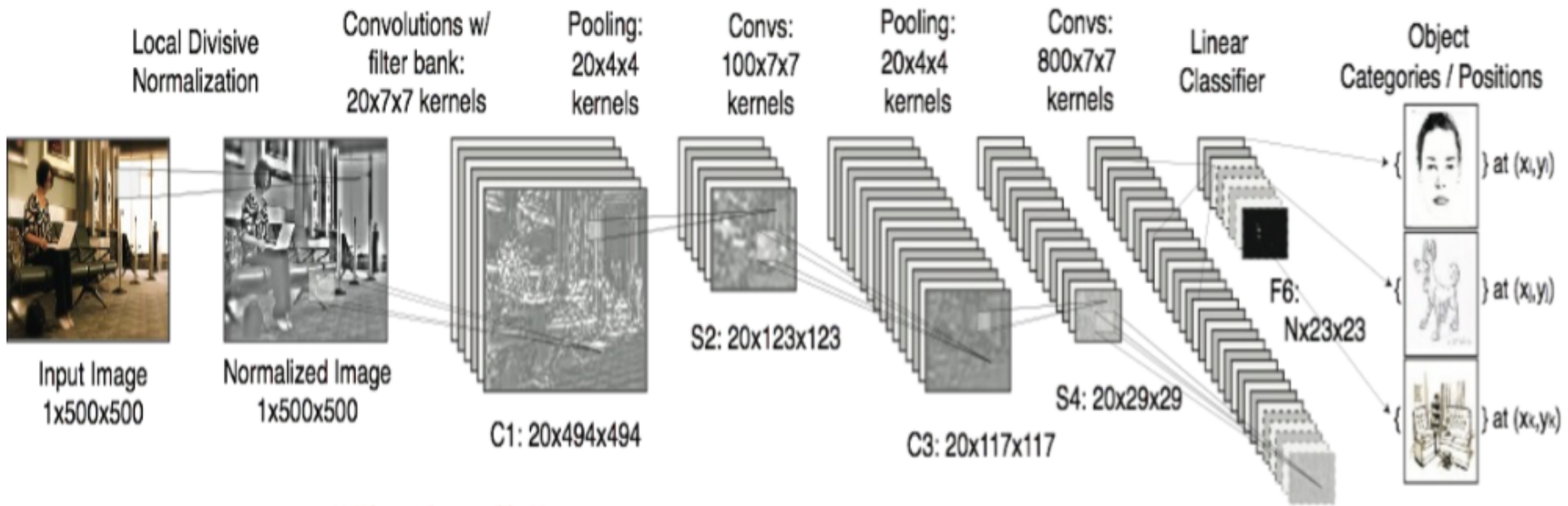


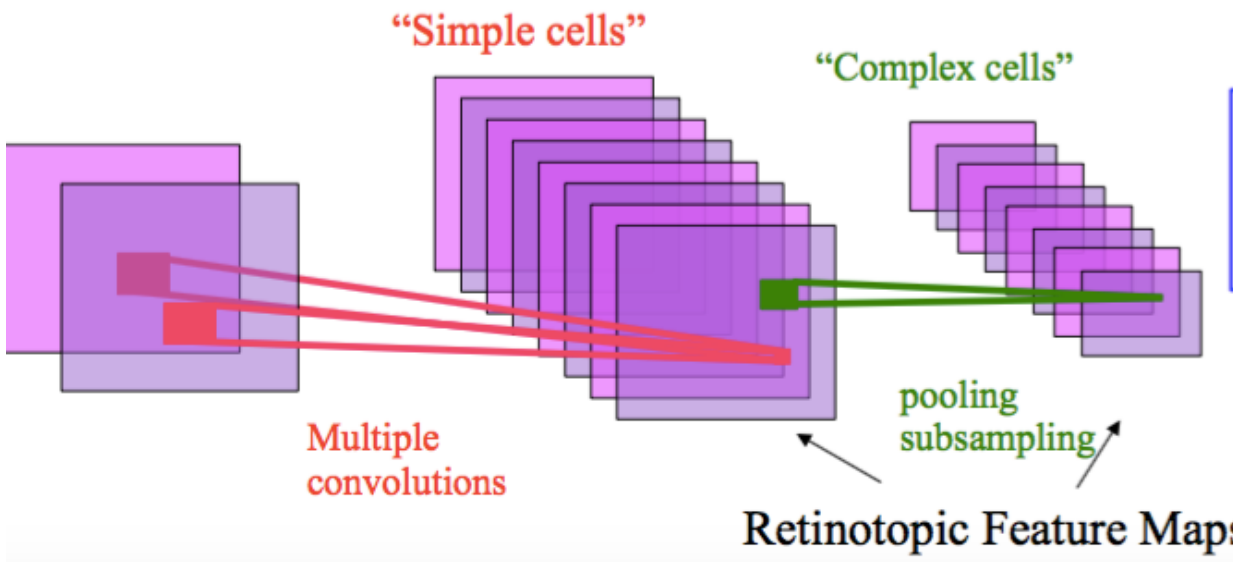
340,098 connections, but *only* 60,000 free, trainable parameters thanks to weight sharing

Convolutional NN



Convolutional NN

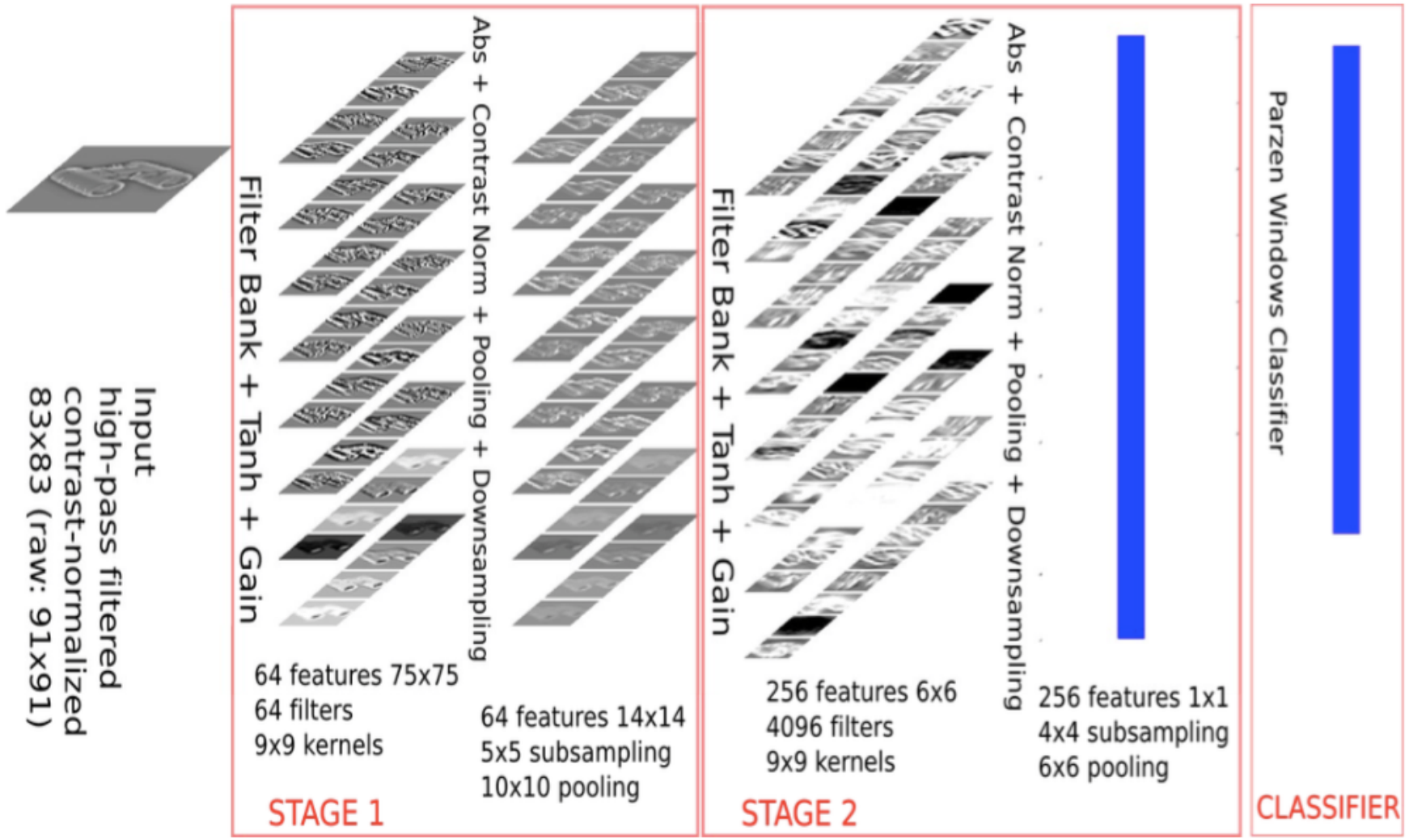




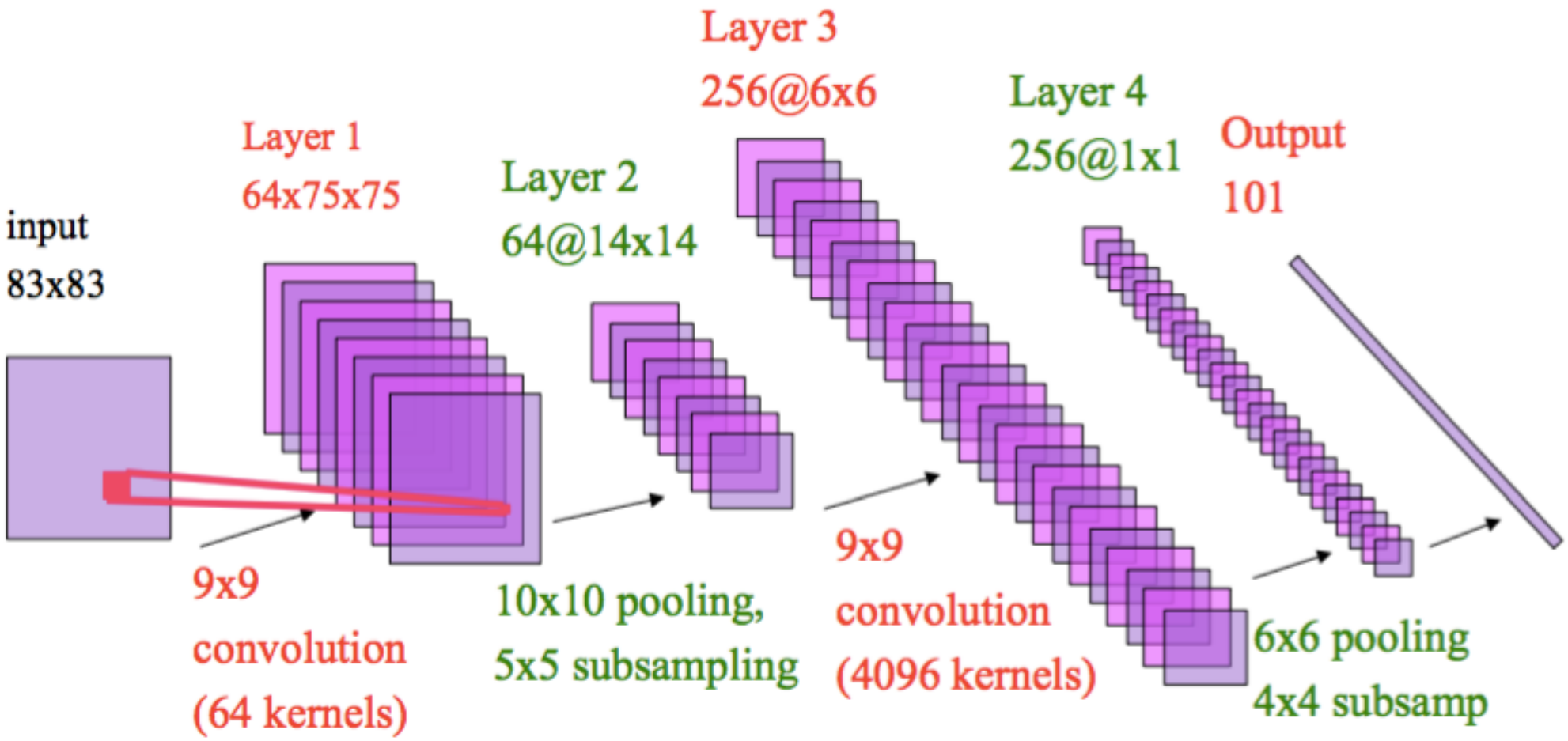
- Training is supervised
- With stochastic gradient descent

[LeCun et al. 89]
[LeCun et al. 98]

Convolutional NN



Convolutional NN



Convolutional NN

Object Recognition [Krizhevsky, Sutskever, Hinton 2012]

Y LeCun
MA Ranzato

- Method: large convolutional net
  - 650K neurons, 832M synapses, 60M parameters
  - Trained with backprop on GPU
  - Trained "with all the tricks Yann came up with in the last 20 years, plus dropout" (Hinton, NIPS 2012)
  - Rectification, contrast normalization,...
- Error rate: 15% (whenever correct class isn't in top 5)
- Previous state of the art: 25% error
- A REVOLUTION IN COMPUTER VISION
- Acquired by Google in Jan 2013
- Deployed in Google+ Photo Tagging in May 2013

Learning / Optimization?

- **Modified back propagation**

- CNNs use *weight sharing* as opposed to feed-forward networks. During both forward and back-propagation convolutions have to be used where the weights and the activations are the functions in the convolution equation.

- *Pooling layers* do not do any learning themselves hence during forward pass, the "winning unit" has its index noted and consequently the gradient is passed back to this unit during the backward pass in the case of max-pooling

What if I don’t have much data?

- In practice, very few people train an entire Convolutional Network from scratch (with random initialization)
- It is (usually) hard to have a dataset of sufficient size!
- It is common to *pretrain* (maybe for days/weeks) a CNN on a very large dataset (e.g. ImageNet, which contains 1.2 million images with 1000 categories), and then use the trained CNN either as an **initialization** or a **fixed feature extractor** for the task of interest.
- → **Transfer learning!**

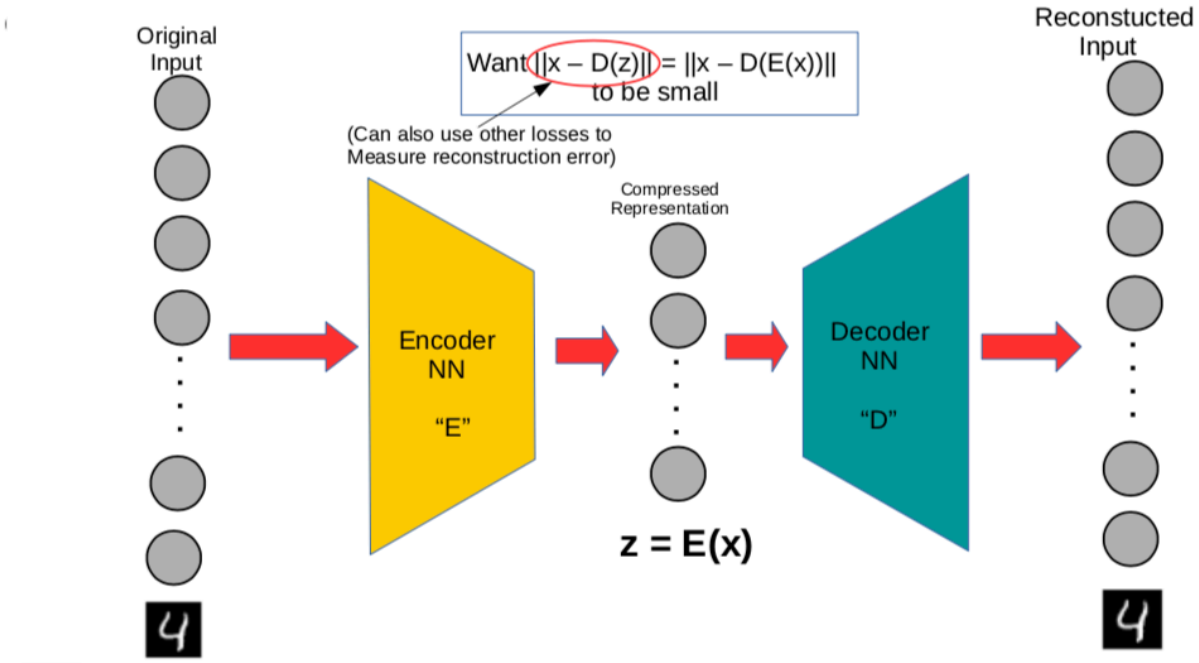
Transfer learning scenarios

- **Pretrained CNN as fixed feature extractor**: remove the last fully-connected layer, treat the rest of the CNN as a fixed feature extractor for the new dataset, add the last classification layer, and *retrain the final classifier* on top of the CNN

- **Fine-tuning the pretrained CNN**: As in the previous scenario, but in addition *fine-tune* the weights of the pretrained network by continuing the back-propagation. It is possible to fine-tune all the layers, or to keep some of the earlier layers fixed (e.g., to avoid overfitting) and only fine-tune some higher-level portions of the network (that usually learn features that are more specific to the training dataset)

Deep NN for unsupervised learning

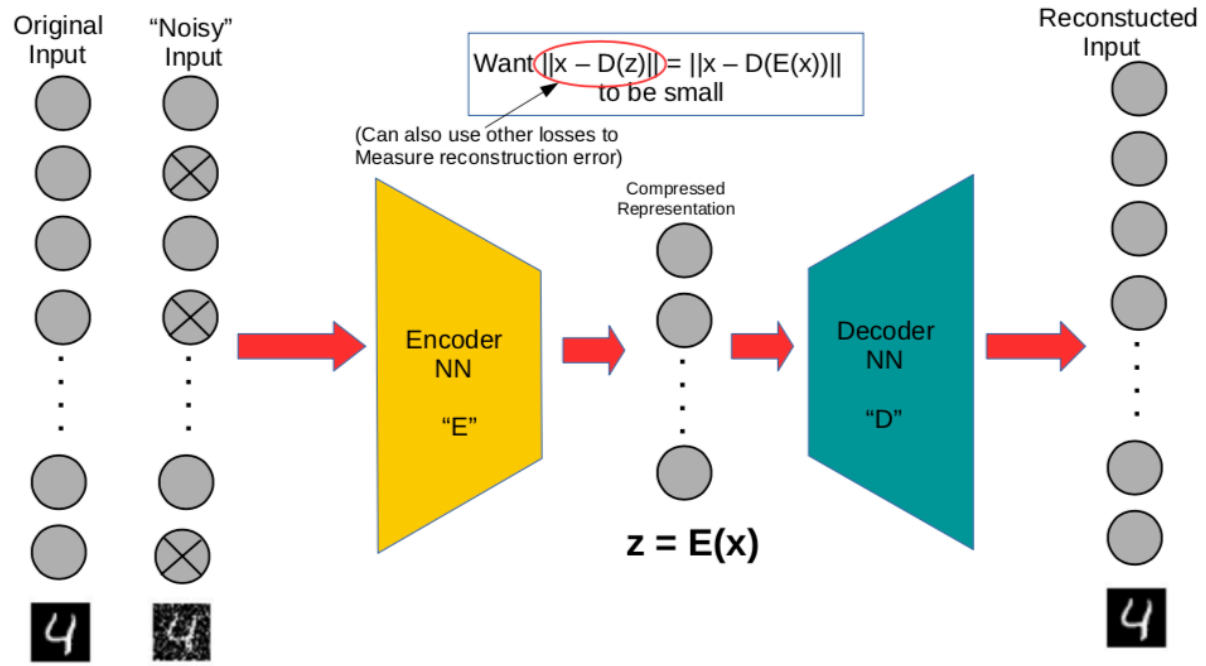
- Auto-encoder (AE) is a popular deep neural network unsupervised feature learning



- If size $\boldsymbol{z}$ is $K < D$, auto-encoders can be used for dimensionality reduction too
- For linear encoder/decodder with $E(\boldsymbol{x}) = \mathbf{W}^\top \boldsymbol{x}$, $D(\boldsymbol{z}) = \mathbf{W}\boldsymbol{z}$ and squared loss, AE is akin to PCA

Deep NN for unsupervised learning

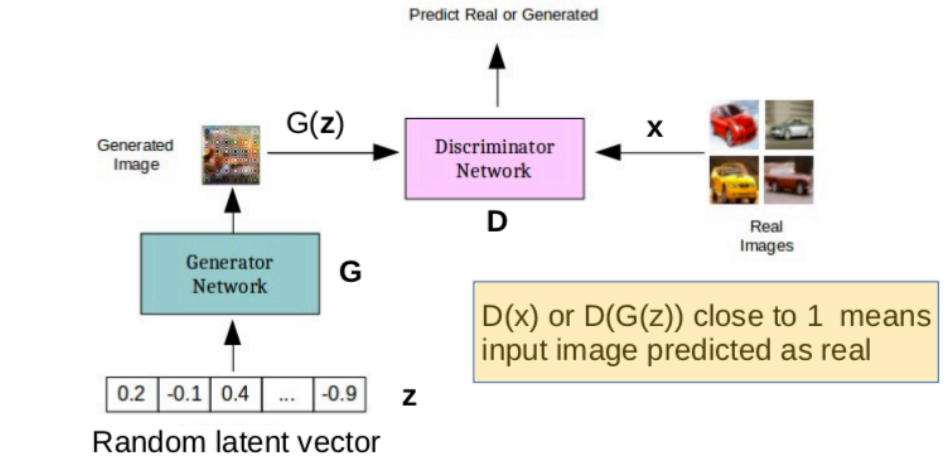
- Denoising auto-encoders: Inject noise in the inputs before passing to to encoder



- Many ways to introduct "noise": Inject zero-mean Gaussian noise, "zero-out" some features, etc
- Especially useful when $K > D$ ($\boldsymbol{z}$ to be a copy of $\boldsymbol{x}$ with $K - D$ zeros) - overcomplete autocoders

Deep NN for Generative Adversarial Learning

- A model that can learn to generate highly real looking data (Goodfellow et al, 2014)
- A game between a Generator and a Discriminator
- Both are modeled by deep neural networks
- Discriminator: A classifier to predict real vs fake data
- Generator transforms a random $\boldsymbol{z}$ to produce fake data
- Discriminator's Goal: Make $D(x) \rightarrow 1$, $D(G(z)) \rightarrow 0$
- Generator's Goal: Make $D(G(z)) \rightarrow 1$ (fool discr.)
- At the game's equilibrium, the generator starts producing data from the true data distribution $p_{data}(x)$



$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$