



Deep learning methods: a practical approach

FRANCESCO MORANDIN

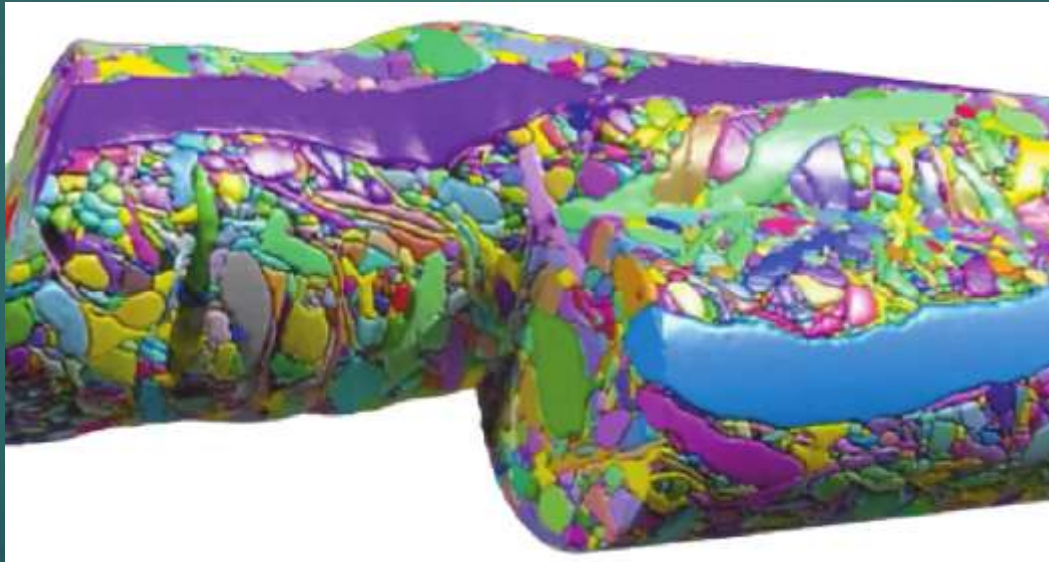


The seventies

NEURAL NETWORKS BEFORE COMPUTER

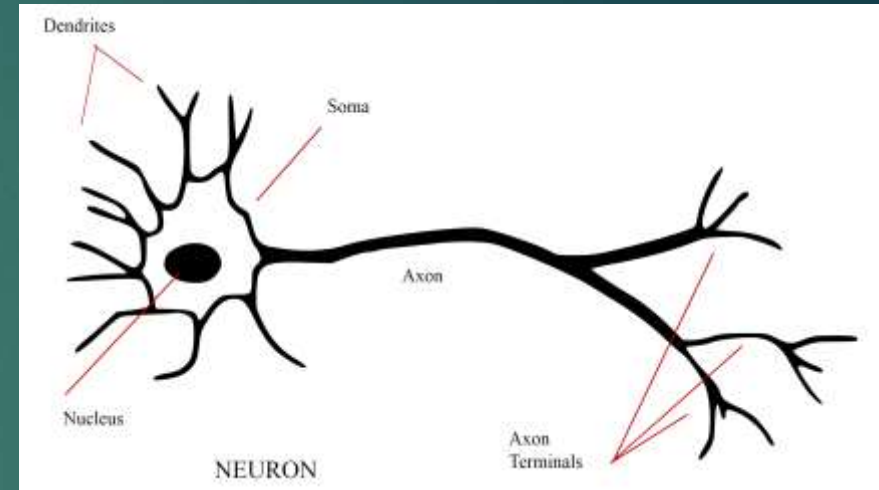
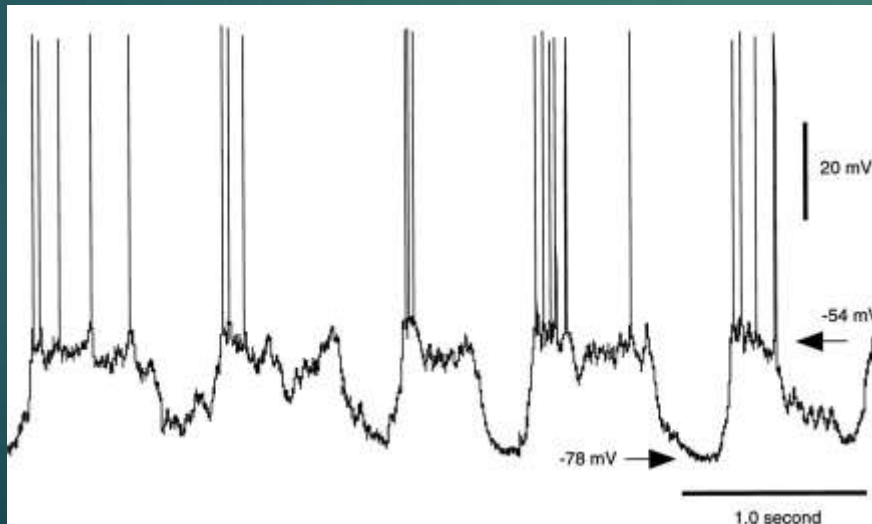
Human brain

- ▶ 86 billions of neurons
- ▶ From a thousand to 100 thousands connections each



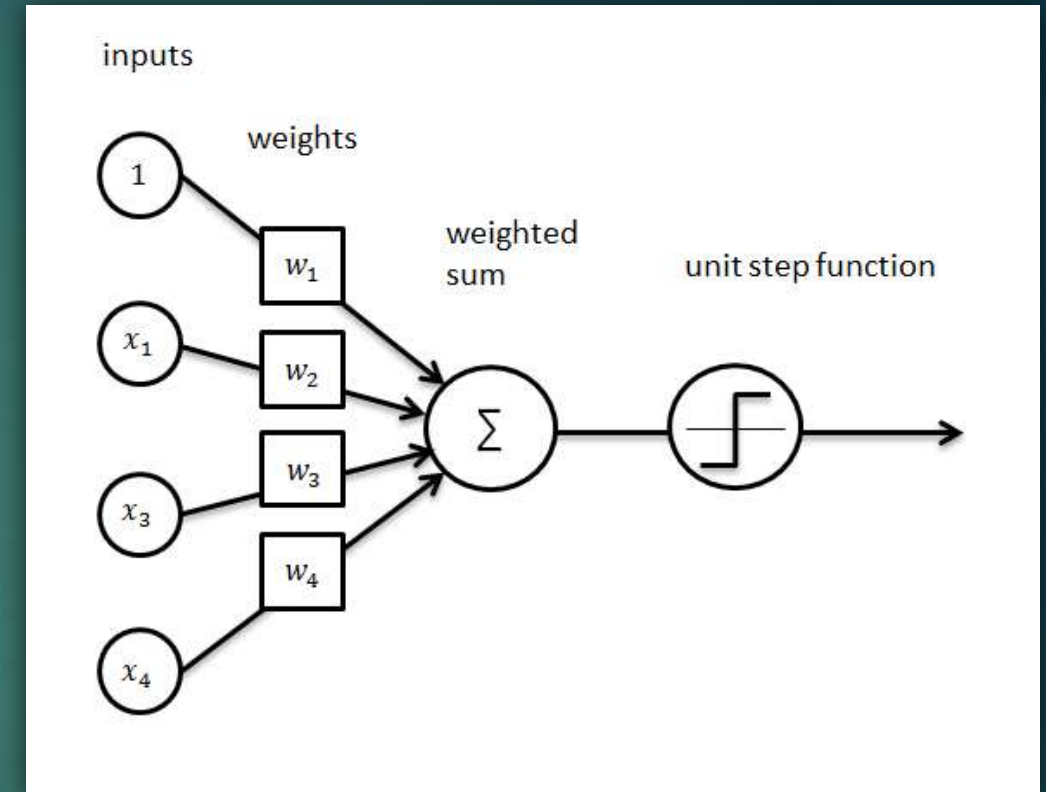
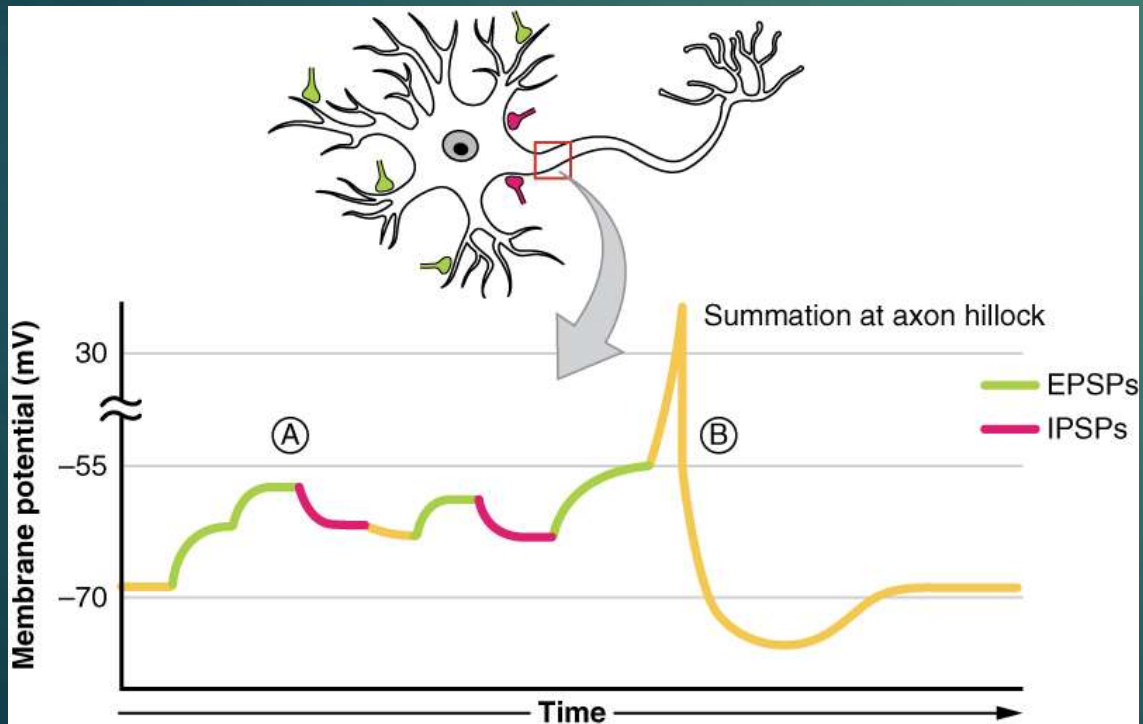
Biological neuron

- ▶ Soma, axon, dendrites, synapsis
- ▶ Action potential
- ▶ Neuron «fires» up to 100 times per second



Biological neuron

- ▶ The neuron *sums* its inputs and *fires* when above threshold

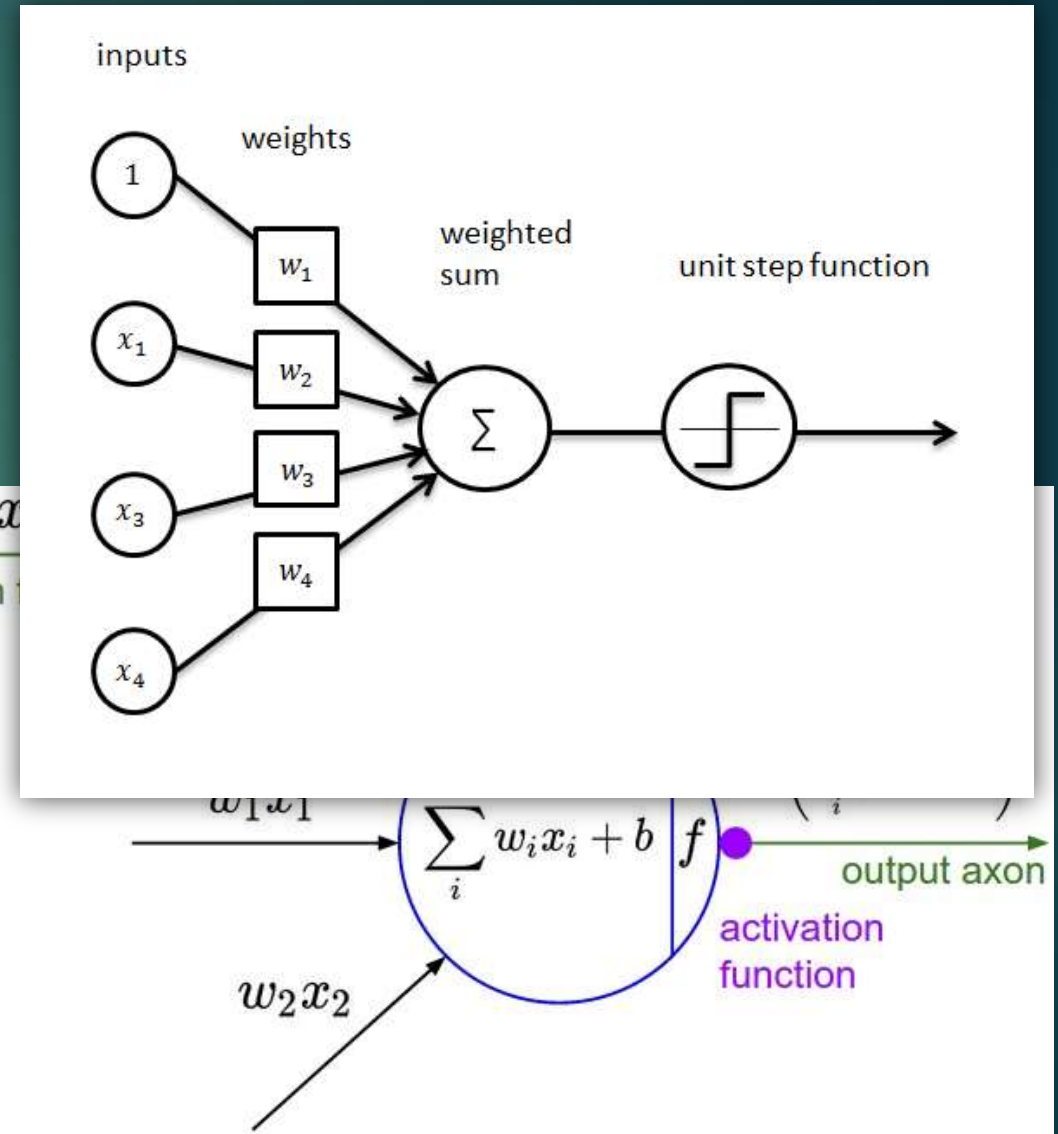


- ▶ Abstraction

Artificial neuron

Perceptron or unit

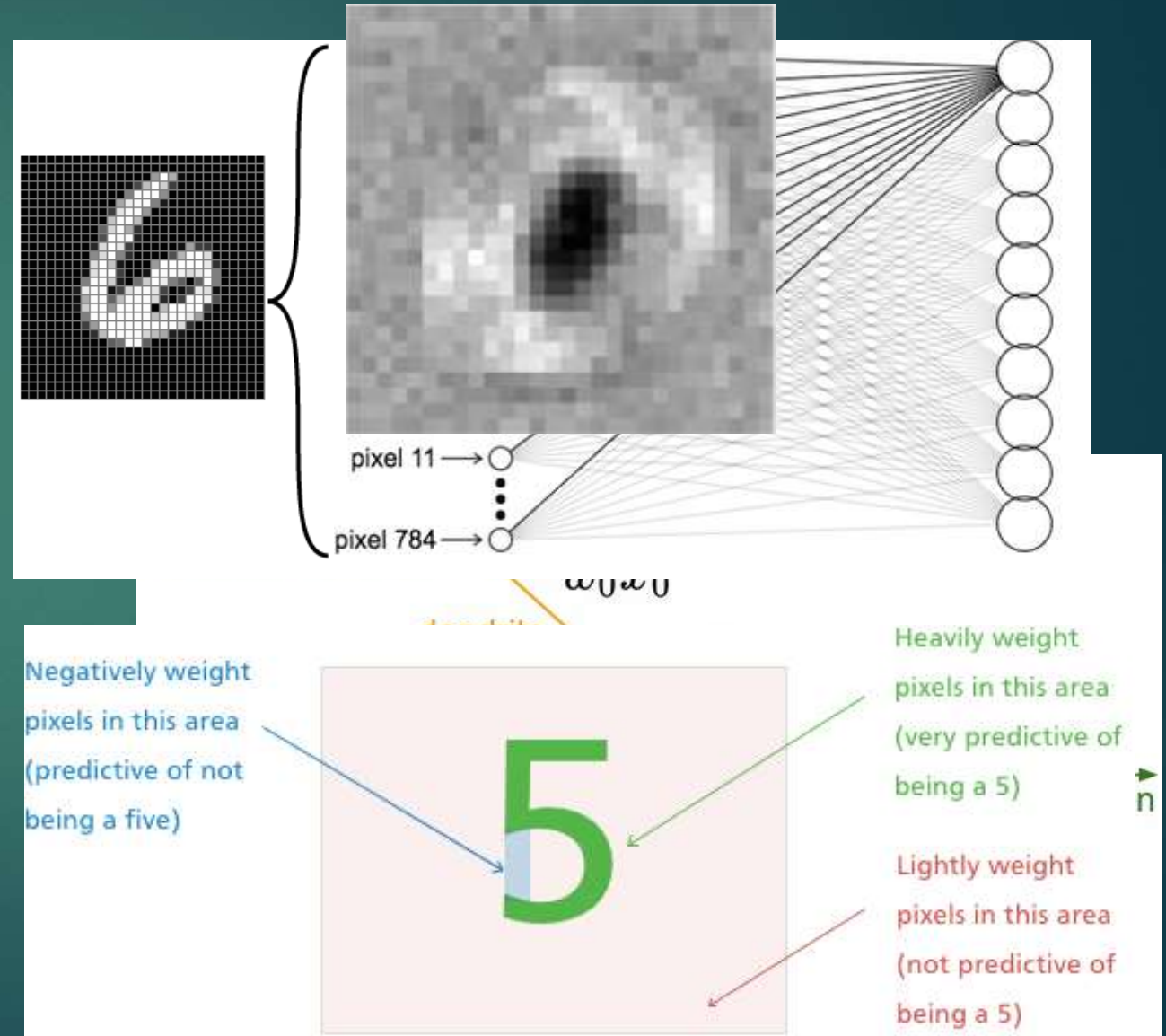
- ▶ Has many **inputs** and a single **output**
- ▶ Does just one thing; always the same
- ▶ Input x_i are added up...
...after they are weighted with w_i
- ▶ The result is shifted by b ...
...and passed through a function f
- ▶ Every neuron has its **weights** w_i , b
- ▶ What it does, depends on them



Artificial neuron

How do weights work?

- ▶ Example: **MNIST**, 28×28 pixels pictures of digits
- ▶ We want to recognize a digit
- ▶ Just one neuron should do:
inputs = pixels
- ▶ w_i : one for every input/pixel
- ▶ Some weights are **positive**, to activate with the intended digit
- ▶ Some weights are **negative**, to exclude other digits



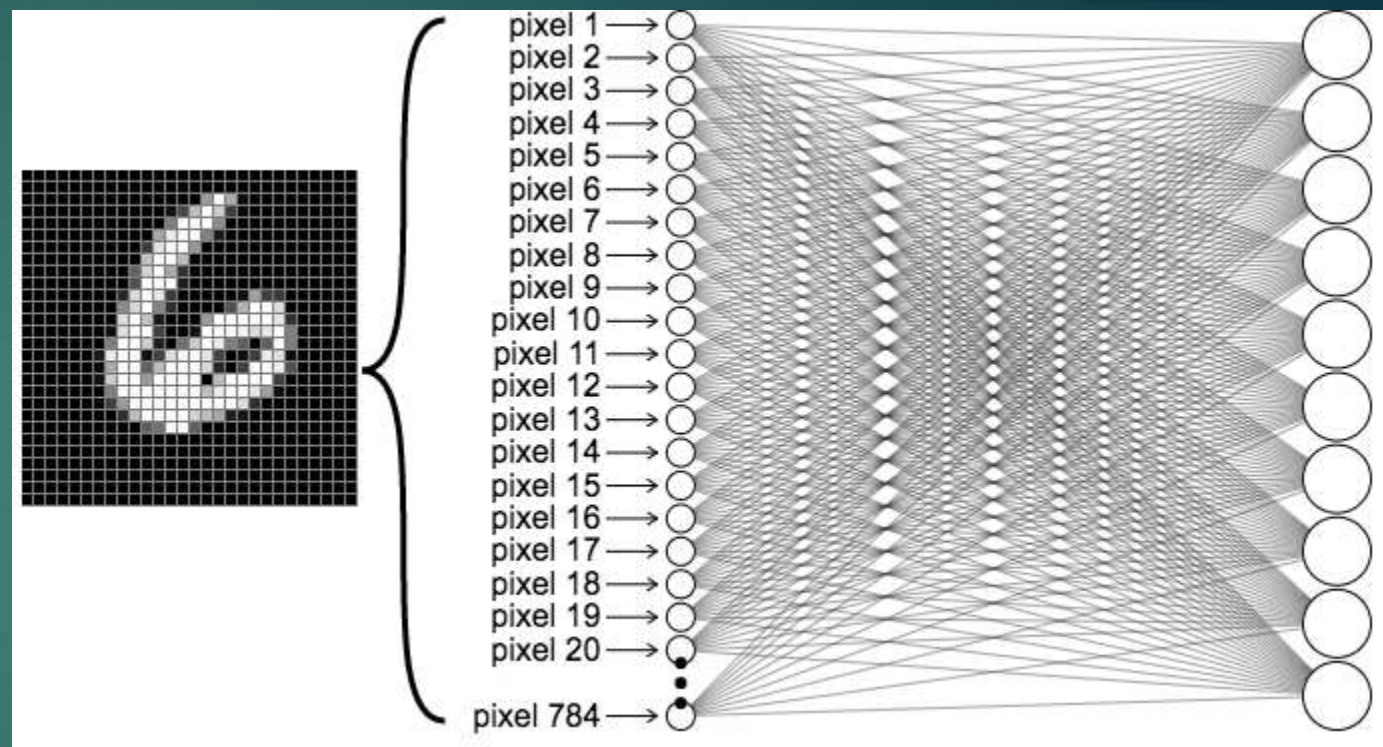
Artificial neuron

Does it really work?

- ▶ How to find these weights?
- ▶ Through **examples**:
learning by imitation
- ▶ Then, test on new examples
- ▶ Does this generalize well?
- ▶ Single neurons... **NO**
- ▶ Networks of neurons... **YES**

1.53%

0.21%



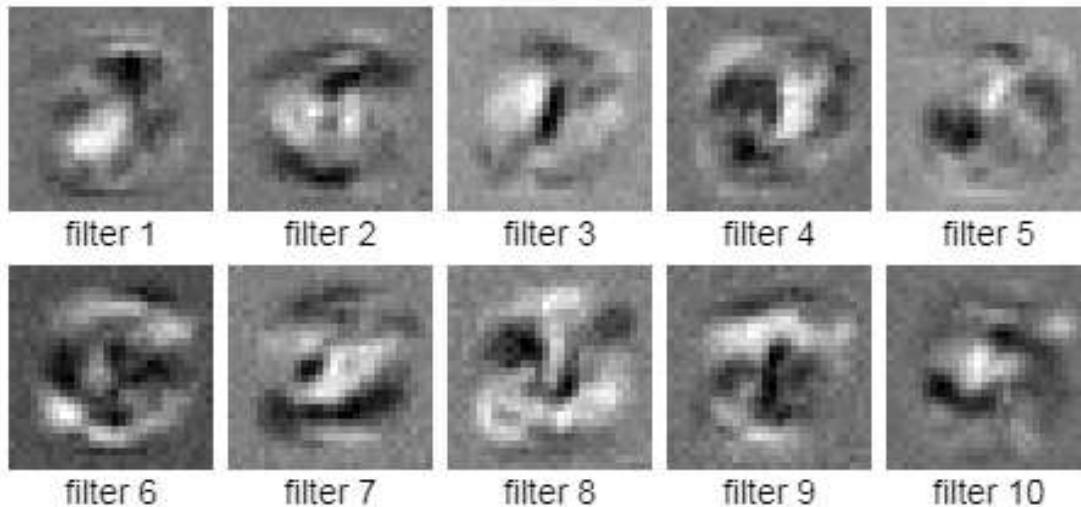
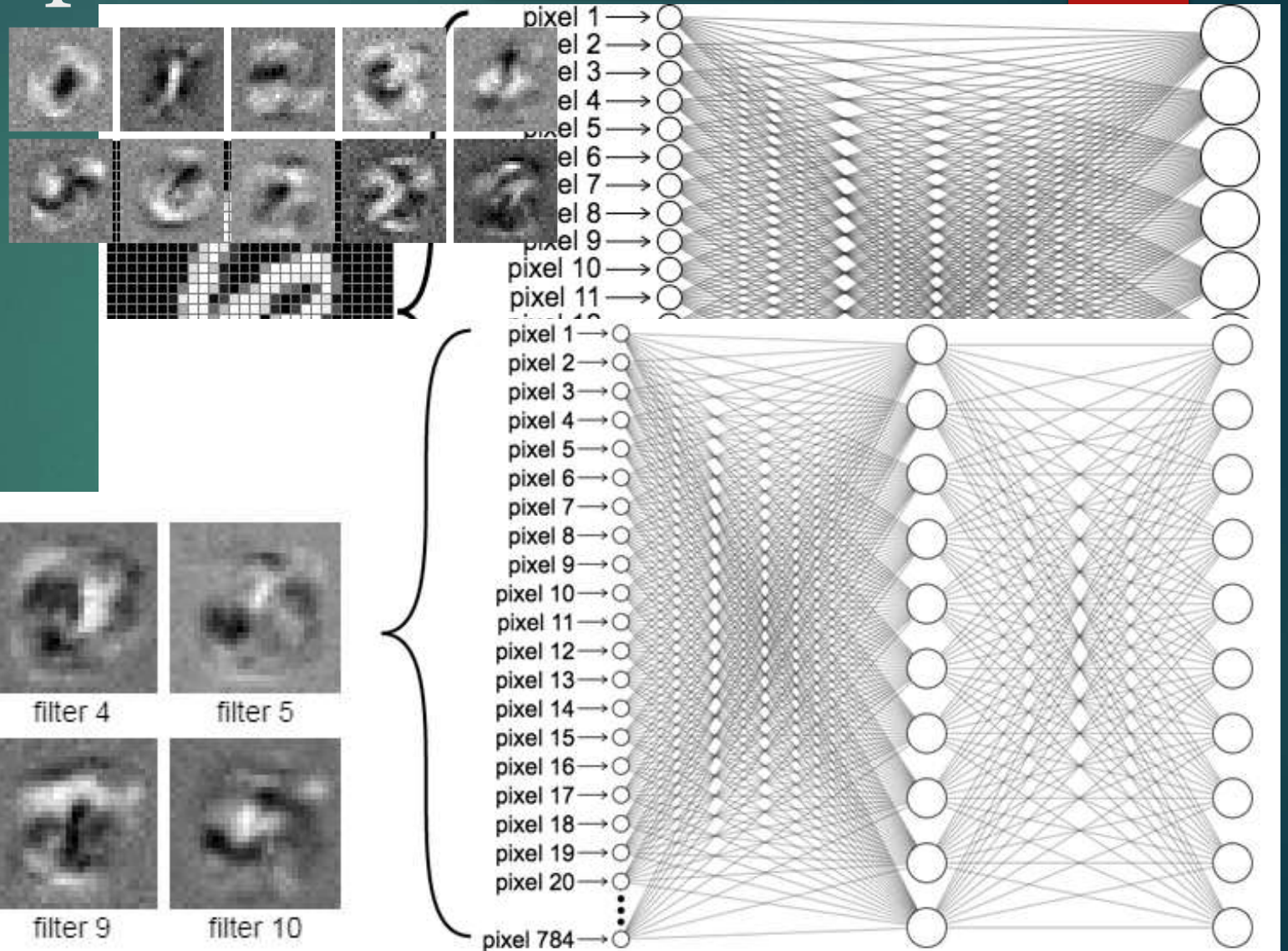
12.0%

Multilayer perceptron

Hidden layers

- ▶ **Layer**-based nets are efficient
- ▶ Input, output and hidden layers
- ▶ More complexity, performance, capacity
- ▶ More weights, and *difficult* to interpret

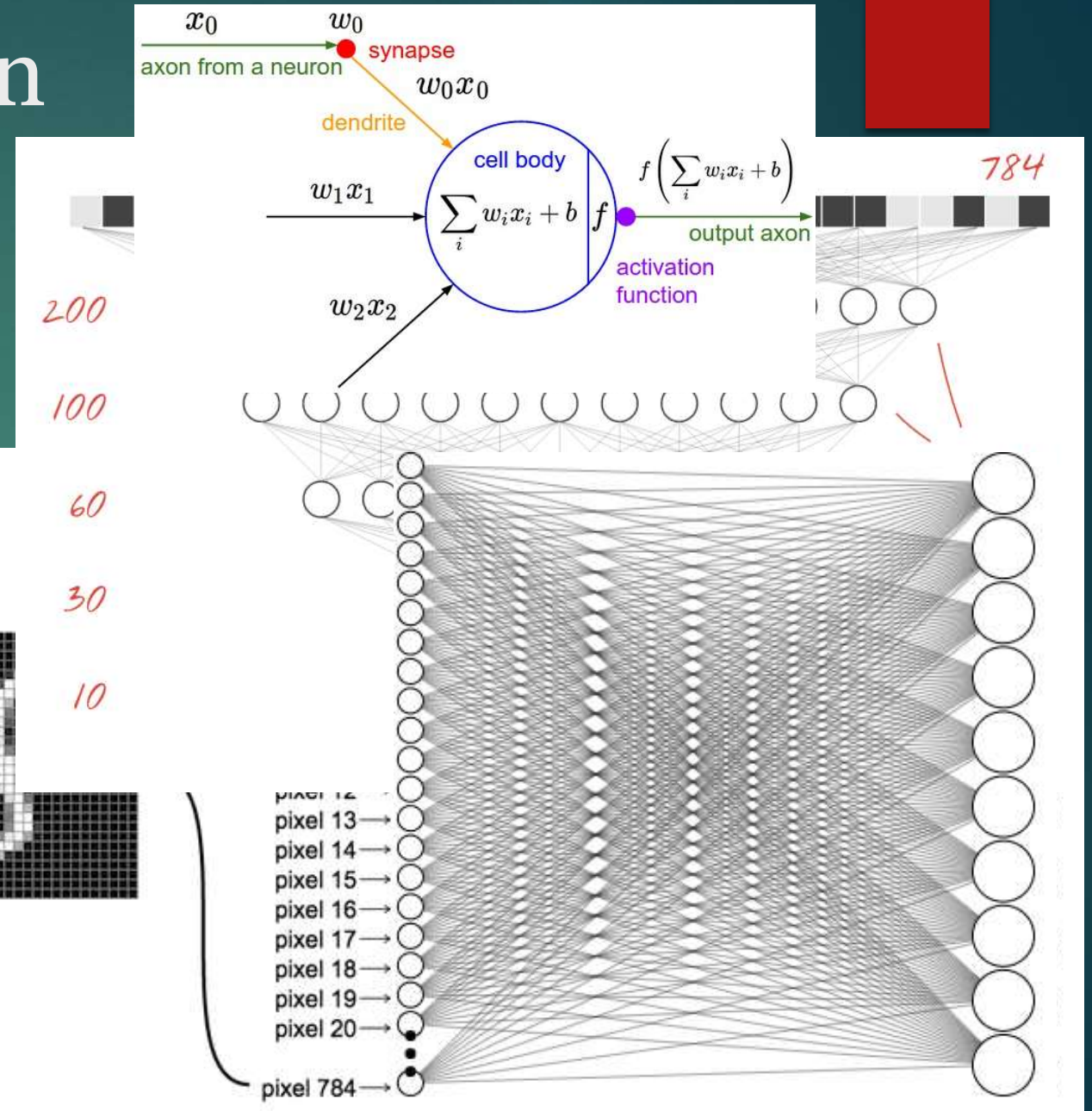
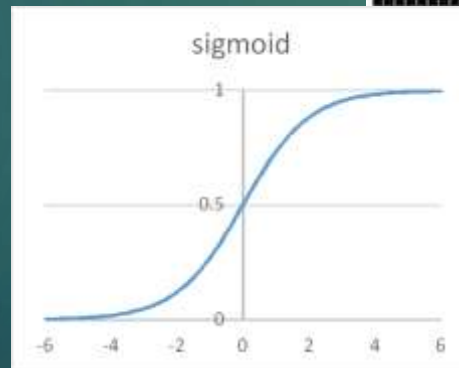
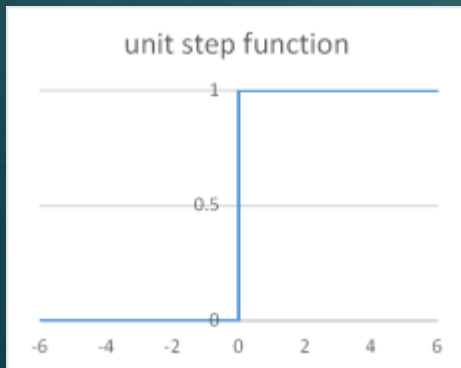
1 layer



Multilayer perceptron

Activation function

- ▶ In multilayer networks the activation function f is crucial
- ▶ Without, the network is equivalent to a 1 layer one
- ▶ There are several kinds



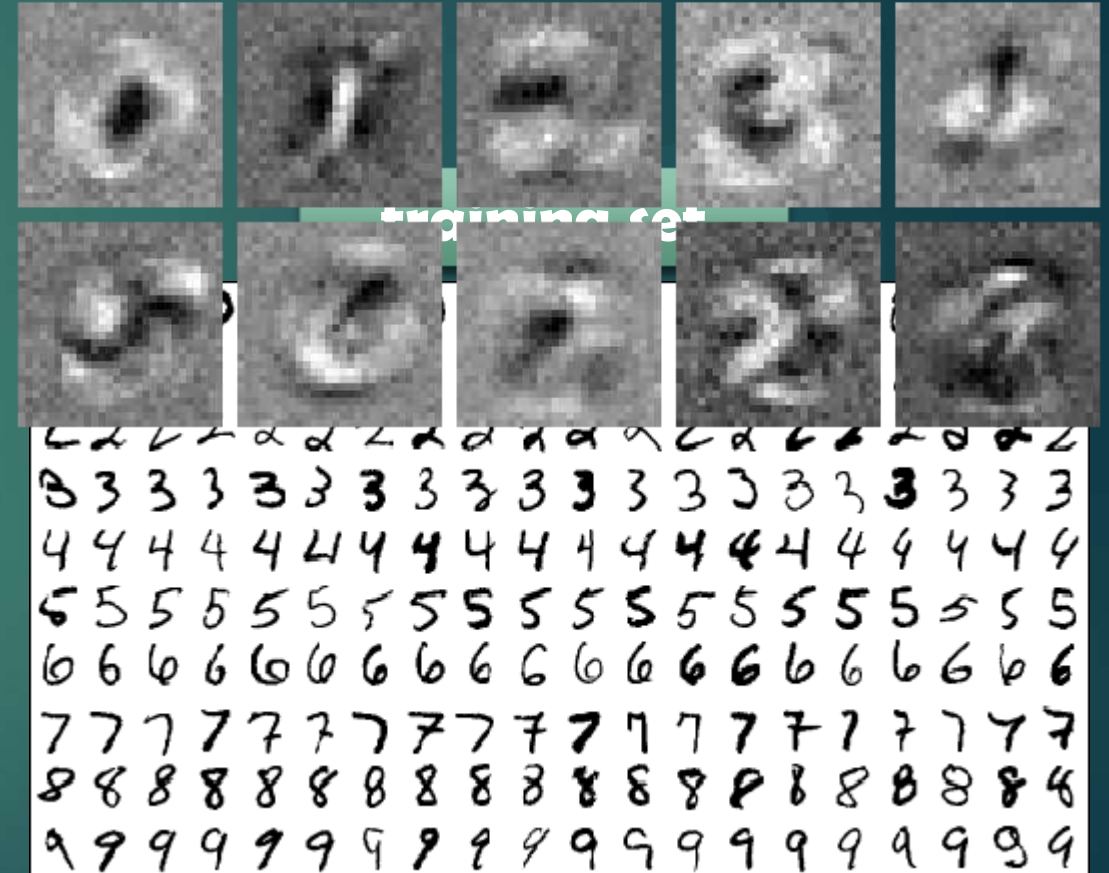


The nineties

SUPERVISED TRAINING SUCCESS

Trade data for weights

- ▶ The network does something useful only if w_i and b are suitably chosen
- ▶ **To train** a network means just to find good values for w_i and b
- ▶ To this end, one needs data:
a training set of examples
learning is by imitation
- ▶ Each example must include the inputs and the correct output



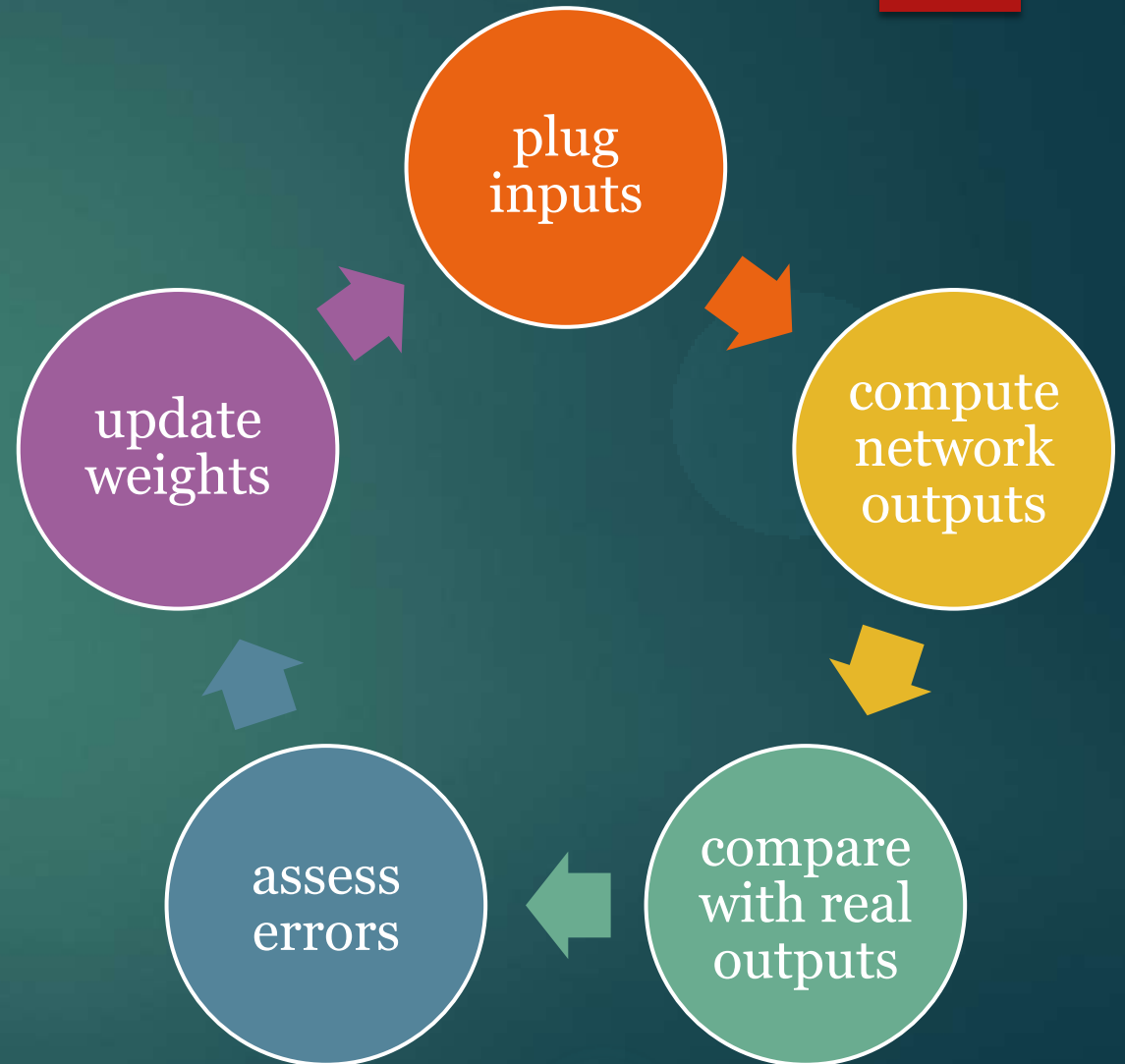
Training is searching

- Initialization:

 - random weights*

- Iterative procedure:

1. Network performance *assessed* on training set
2. Weights are *updated* to decrease errors
3. Cycle



Loss function

- ▶ Network performance is determined by comparing:
 - the real output y_k of training data $(x_{k,i}, y_k), k = 1, \dots, N$
 - the output predicted by the network, given the inputs

$$z_k = \text{net}(x_k, w) = f_{\text{out}}\left(\sum_l w_l f_l\left(\sum_j w_j f_j\left(\dots \sum_i w_i x_{k,i}\right)\right)\right), \quad k = 1, \dots, N$$

- ▶ The comparison is done with some sort of *distance*

$$d: \mathbb{R}^N \times \mathbb{R}^N \rightarrow \mathbb{R}, \quad d(y, z) = \text{loss}(y, z)$$

Loss function

- ▶ The comparison is done with some sort of *distance*

$$d: \mathbb{R}^N \times \mathbb{R}^N \rightarrow \mathbb{R}, \quad d(y, z) = \text{loss}(y, z)$$

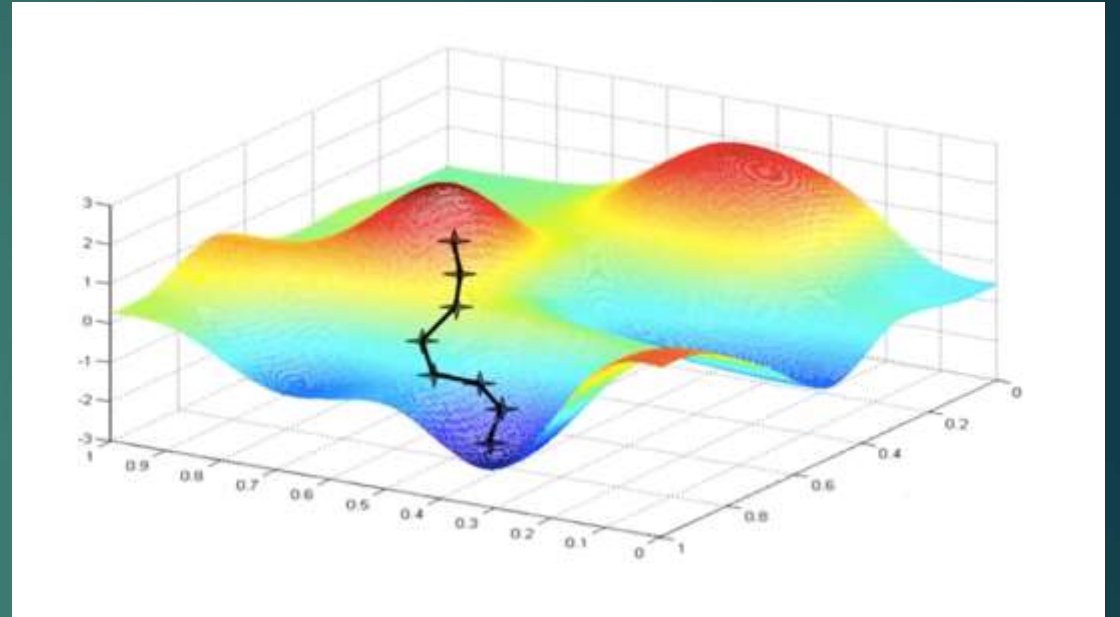
- ▶ The technical name is **loss function**
- ▶ Basic example is sum of squares of differences

$$\text{loss}(y, z) = \sum_{k=1}^N l(y_k, z_k) = \sum_{k=1}^N (y_k - z_k)^2 = \|y - z\|^2$$

Gradient descent

- ▶ Weights update should decrease loss
- ▶ So compute loss **gradient** with respect to weights
- ▶ And move in that direction

$$w' = w - \varepsilon \cdot \nabla_w \text{loss}(y, z)$$



Chain rule on a network

- ▶ Gradient is complicated, because *composition*!

$$\nabla_w \sum_{k=1}^N (y_k - z_k)^2, \quad z_k = f_{\text{out}} \left(\sum_l w_l f_l \left(\dots \sum_i w_i x_{k,i} \right) \right)$$

- ▶ **Chain rule** is the way
- ▶ Can be computed efficiently by
 - computing values forward
 - computing derivatives backwards

Chain rule and backprop

- ▶ Specifically, let $m = 0, 1, \dots, M$ denote the layers
- ▶ Let U_m denote the set of all the units of layer m
- ▶ Let $v^{(m)} \in \mathbb{R}^{U_m}$ be the vector with all their values
- ▶ So $v^{(0)} = x$ is the input and $v^{(M)} = z$ is the output
- ▶ Let $F_{M,m}: \mathbb{R}^{U_m} \rightarrow \mathbb{R}^{U_M}$ denote the function computed by the network from layer m to layer M , i.e. $v^{(M)} = F_{M,m}(v^{(m)})$
- ▶ Then the derivatives of $F_{M,m}$ can be computed recursively on m

Chain rule and backprop

$$\frac{\partial}{\partial u} F_{M,m}(\nu^{(m)}) = \sum_{v \in U_{m+1}} \frac{\partial}{\partial v} F_{M,m+1}(\nu^{(m+1)}) \cdot \frac{\partial}{\partial u} [F_{m+1}]_v(\nu^{(m)}), \quad u \in U_m$$

- ▶ Specifically, let $m = 0, 1, \dots, M$ denote the layers
- ▶ Let U_m denote the set of all the units of layer m
- ▶ Let $\nu^{(m)} \in \mathbb{R}^{U_m}$ be the vector with all their values
- ▶ Let $F_{M,m}: \mathbb{R}^{U_m} \rightarrow \mathbb{R}^{U_M}$ denote the function computed by the network from layer m to layer M , i.e. $\nu^{(M)} = F_{M,m}(\nu^{(m)})$
- ▶ Then the derivatives of $F_{M,m}$ can be computed recursively on m
- ▶ This is **backpropagation**

Mini-batches and SGD

- ▶ Gradient is a *huge sum*, over whole training set

$$\nabla_w \text{loss}(y, z) = \sum_{k=1}^N \frac{\partial}{\partial z} l(y_k, z_k) \cdot \nabla_w z_k$$

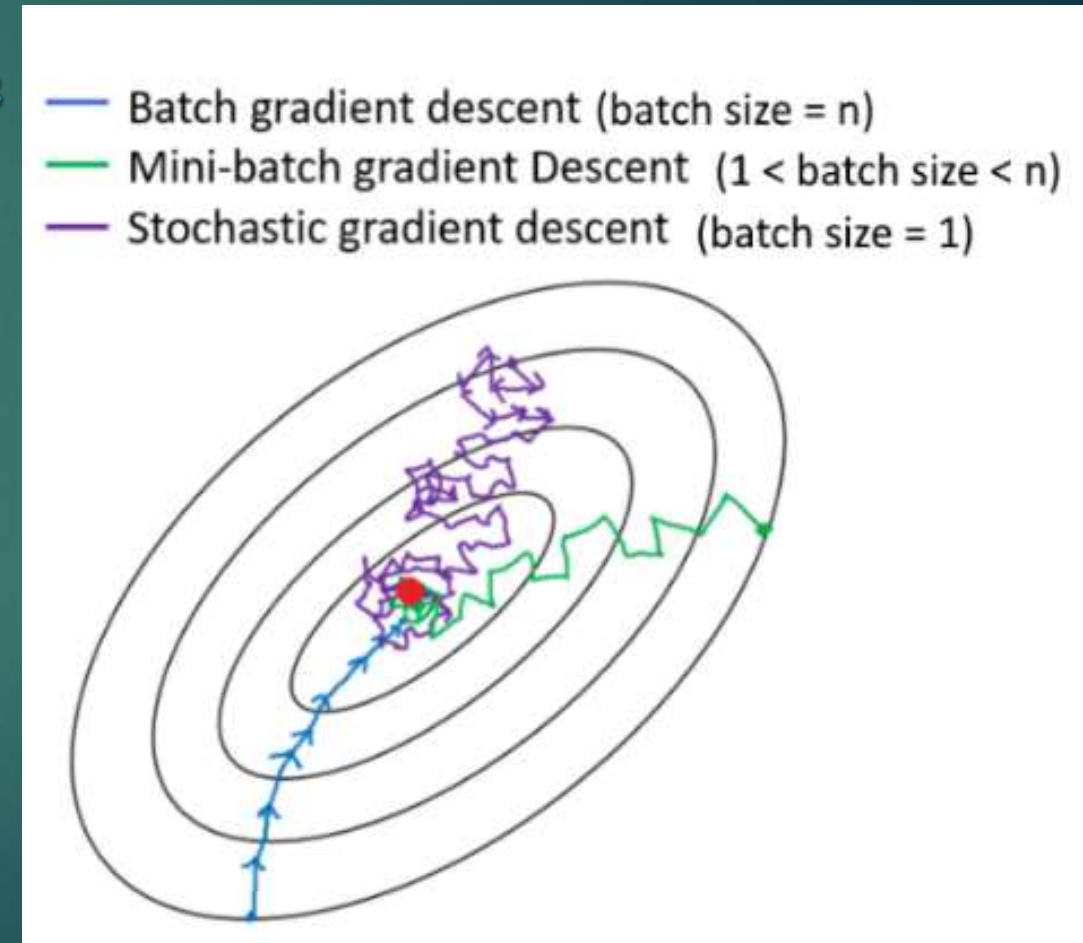
- ▶ Once computed, it gives the direction of next step
- ▶ One might accumulate gradients over *smaller subsets*...
- ▶ ...and make smaller, intermediate steps

$$w' = w - \varepsilon \cdot \nabla_w \text{loss}(y, z)$$

$$w^{(i+1)} = w^{(i)} - \varepsilon \cdot \sum_{k \in B_i} \frac{\partial}{\partial z} l(y_k, z_k) \cdot \nabla_w z_k, \quad i = 1, 2, \dots$$

Mini-batches and SGD

- ▶ These subsets are called **mini-batches**
- ▶ Should be sampled randomly
- ▶ (with or without replacement)
- ▶ Typical size from 128 to 4096 (vectorization)
- ▶ In stochastic gradient descent (SGD), size = 1
- ▶ Not very practical
- ▶ Definition not uniform



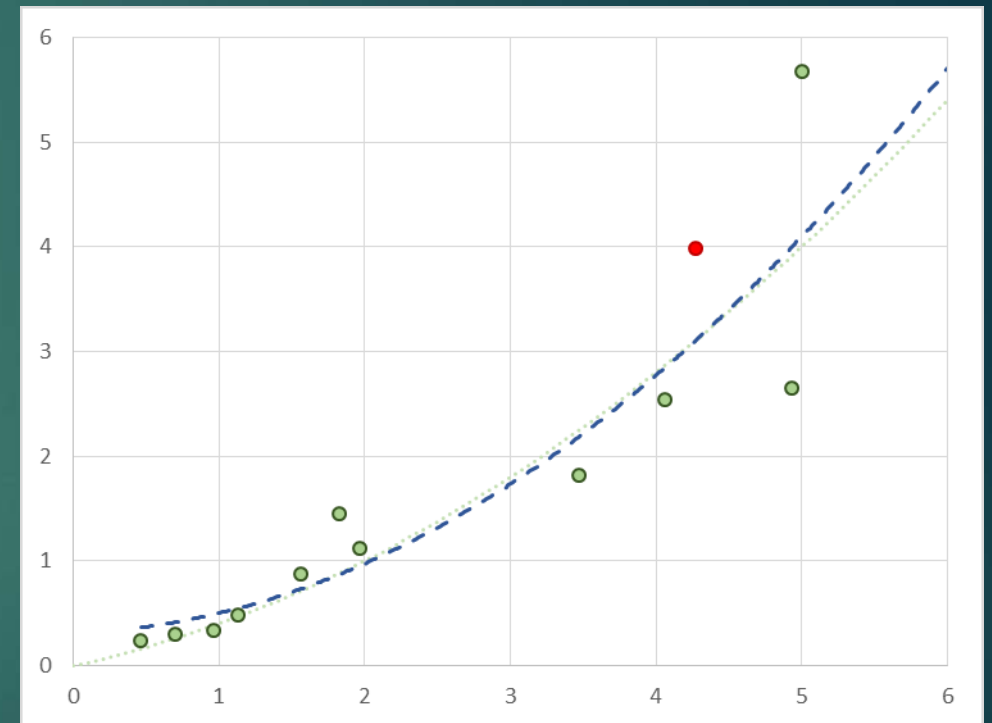
Machine learning

- ▶ Training is done on a given training dataset
- ▶ Network learns input-output association on those samples
- ▶ Then it is used on new input data to predict output
- ▶ But new data is different and never seen in training
- ▶ Will this work?
- ▶ Yes, under two conditions:
 - Input data should be similar, coherent
 - Network should be able to *generalize*

Machine learning

Common illness: overfitting

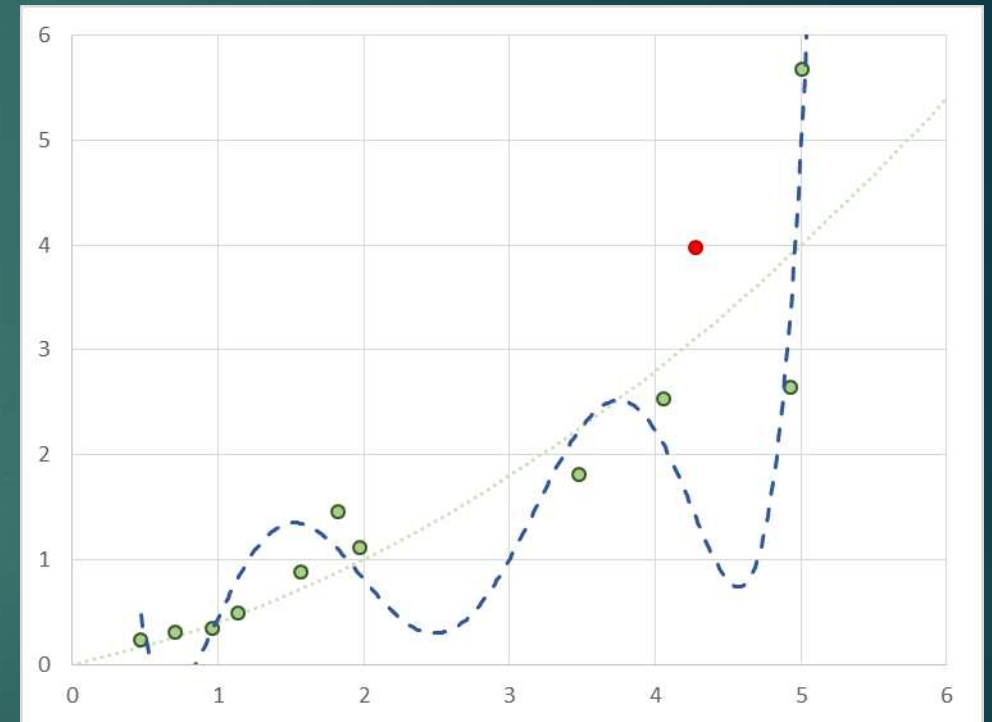
- ▶ Sometimes a network is *too well adapted* to training data...
- ▶ ...to the single points of the training data...
- ▶ ...then it is *bad* on new data
- ▶ No ability to generalize: **overfitting**
- ▶ Kind of like memorizing without understanding



Machine learning

Common illness: overfitting

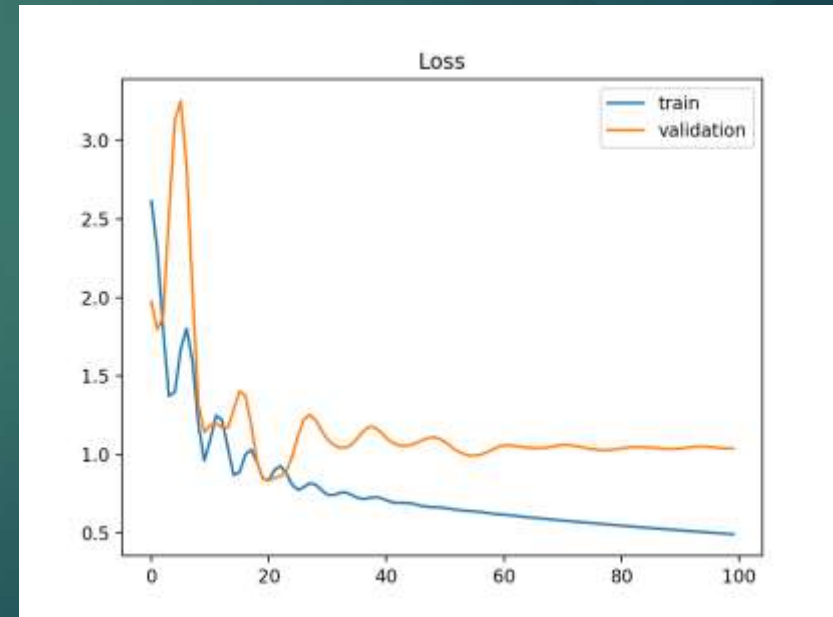
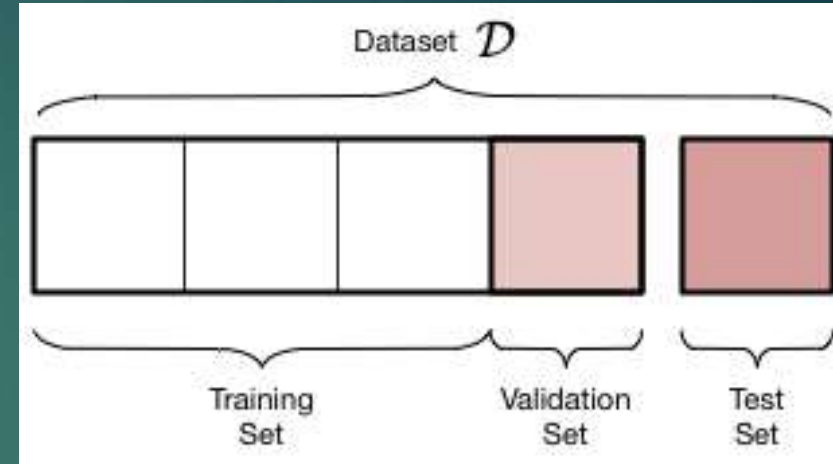
- ▶ Usually happens when:
 - one trains too much
 - the network is too large



Machine learning

Common cure: validation

- ▶ Easy to check if this is happening
- ▶ Keep a *fraction* of the data out of the training
- ▶ Use it to:
 - check overfitting
 - measure real performance
- ▶ It is called the **validation set**





The tens

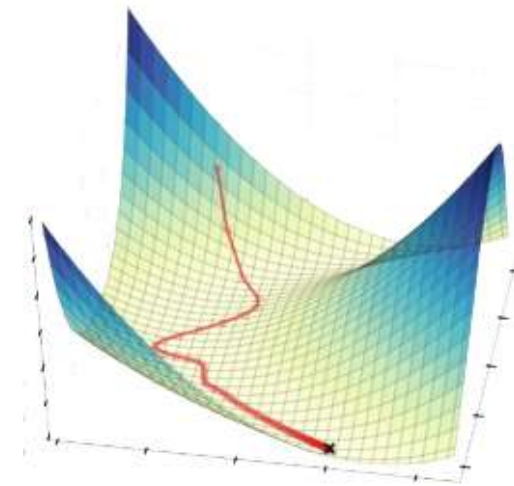
SUBTLE SCIENCE AND EXACT ART OF DEEP LEARNING

The second winter of neural networks

- ▶ U
- ▶ T
- ▶ P
- ▶ T
- ▶ A

WINTER
IS COMING

5 0 4 1



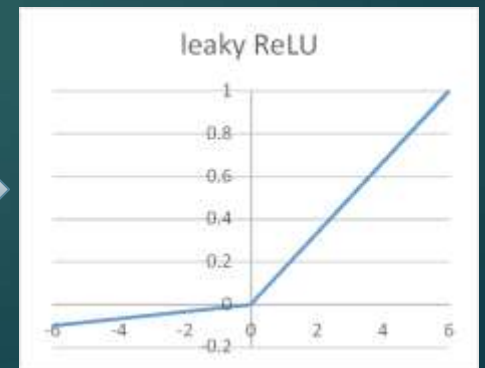
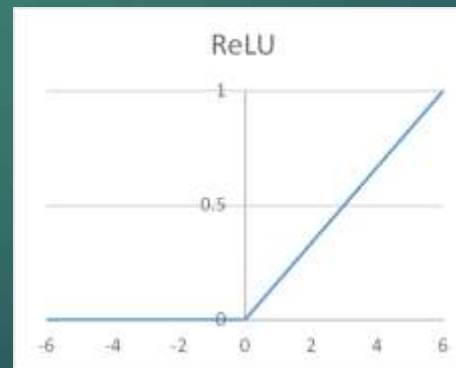
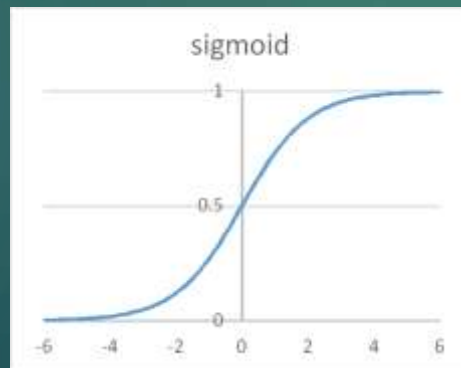
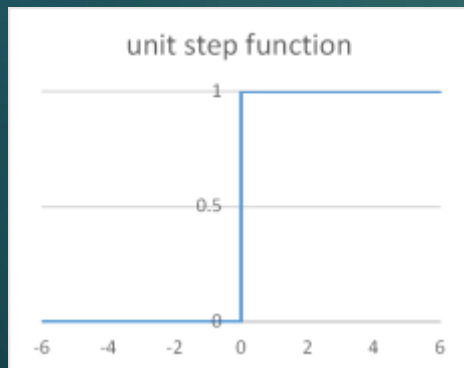
Subtle science and exact art of DL

- ▶ Then computers became more powerful
- ▶ GPU could be used effectively
- ▶ And a handful of artists made *small changes* to the recipe



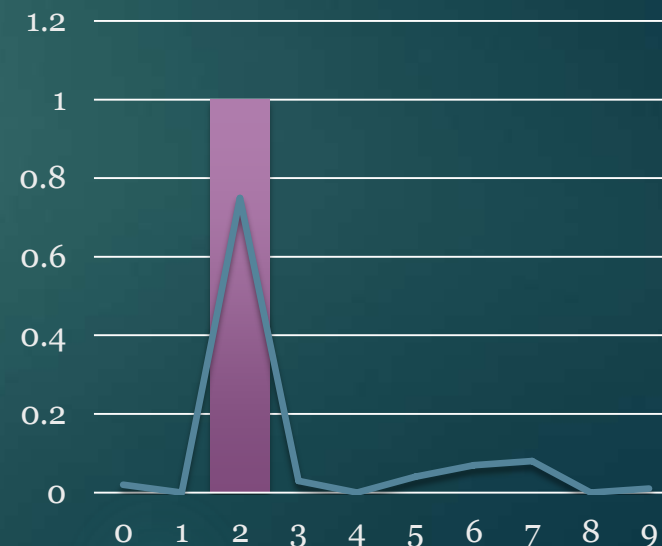
Why sigmoid activation?

- ▶ Its derivative is close to zero almost everywhere
- ▶ Vanishing gradient problem
- ▶ Rectified Linear Unit (**ReLU**)
- ▶ Leaky and parametric ReLU



Why sum of squares loss?

- ▶ Minimizing sum of squares is traditional
- ▶ Comes from *maximum likelihood estimation*...
- ▶ ...in the case of Gaussian error with uniform variance
- ▶ What about categorical output, like MNIST?
- ▶ Comparing
 - one-hot output (0,0,1,0,...)
 - probabilities prediction (0.02, 0.00, 0.75, ...)



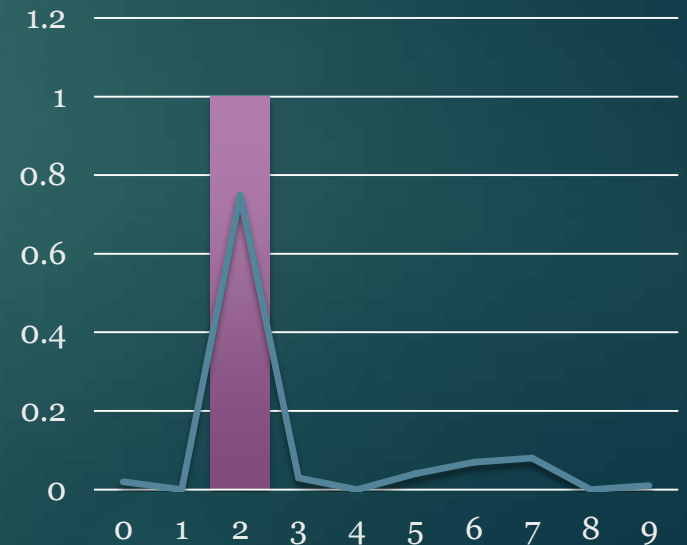
Why sum of squares loss?

- ▶ Maximum likelihood estimation gives **cross-entropy**

$$\log L(p_0, \dots, p_9 | x_0, \dots, x_9) = \log(p_0^{x_0} \cdots p_9^{x_9}) = \sum_i x_i \log p_i = -H(x, p) = c + D_{\text{KL}}(x \| p)$$

- ▶ Suitable loss depends on model

$$\log L(\mu | x) = \log\left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}\right) = \alpha - \beta(x - \mu)^2$$

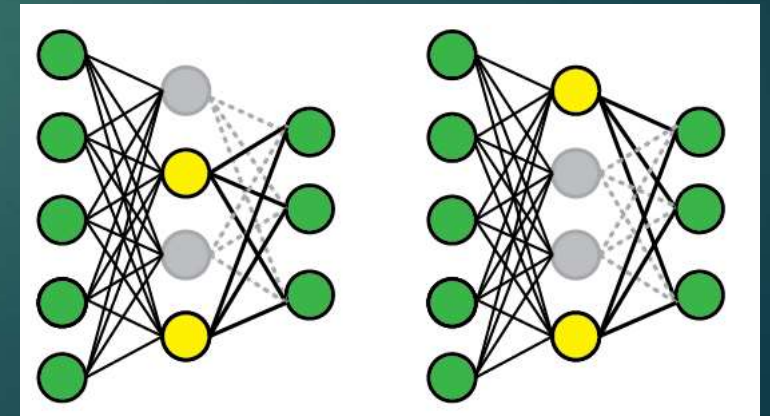


Why not to vaccinate?

Overfitting season ahead

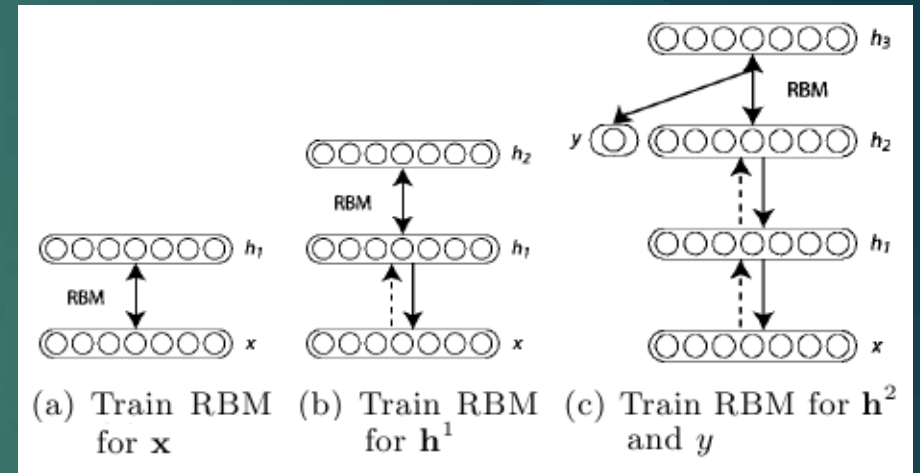
- ▶ Overfitted models typically have:
 - larger weights
 - specialized subnets
- ▶ **Regularization** was introduced to reduce both
- ▶ There are two ways
 - regularizing term in loss function
 - **dropout** while training

$$\nabla_w [\text{loss}(y, z) + \|w\|^2]$$



Why random initialization?

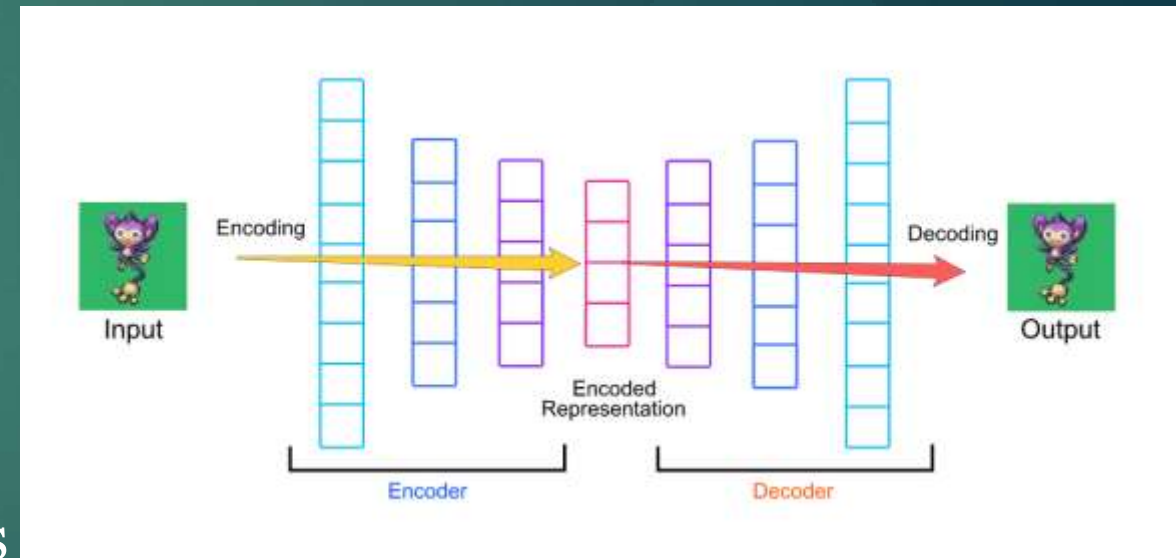
- ▶ Before training, weights are randomly initialized
- ▶ This is convenient, but maybe we can do better
- ▶ One way is to pre-train each layer...
- ▶ ...one after the other...
- ▶ ...using *only their inputs*



Is this even possible!?

Teaching with questions

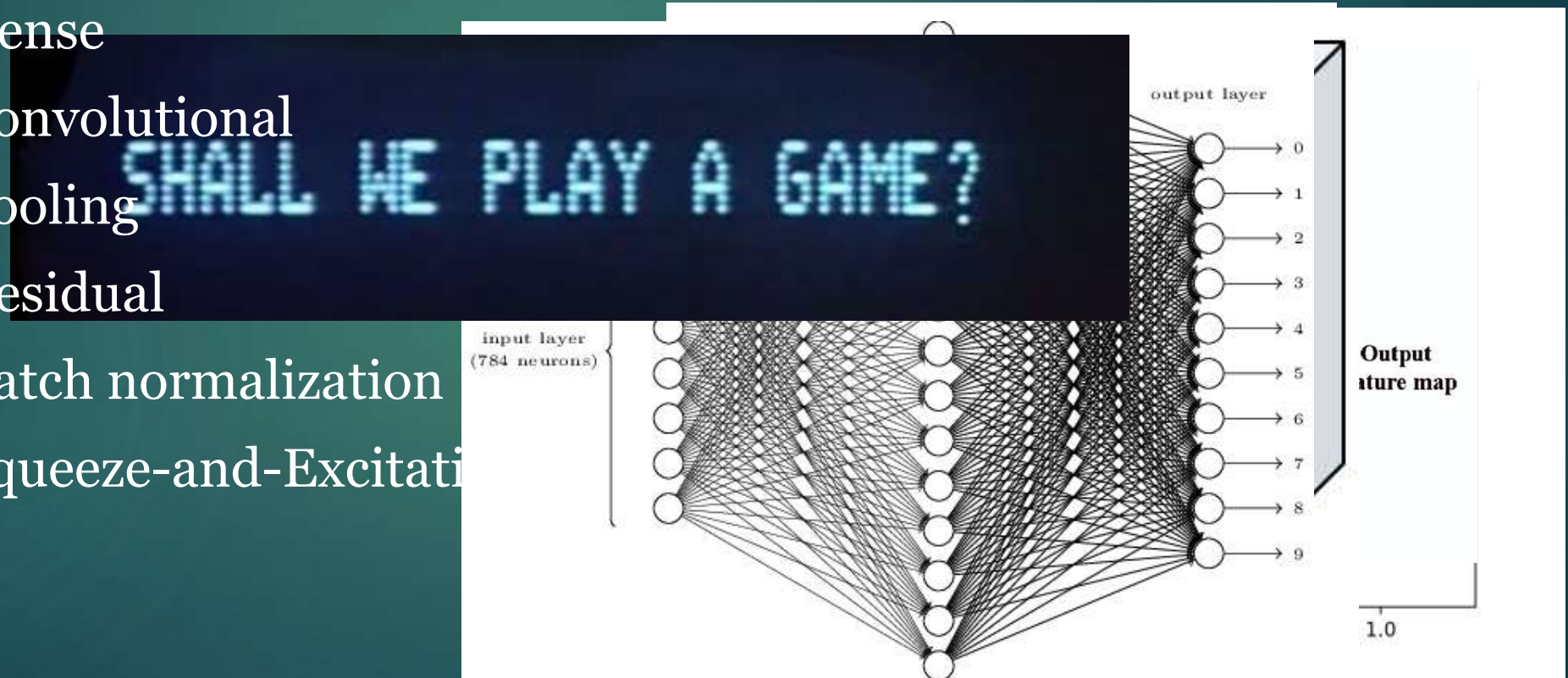
- ▶ There are ways to train without using the output
- ▶ It is called **unsupervised** training
- ▶ Basically, one plugs again the input as output
- ▶ There must be some *bottleneck* to make it nontrivial
- ▶ You get **autoencoders** and weights initialization



Fifty shades of Deep Learning

► Network layers come in many flavors

- Dense
- Convolutional
- Pooling
- Residual
- Batch normalization
- Squeeze-and-Excitation

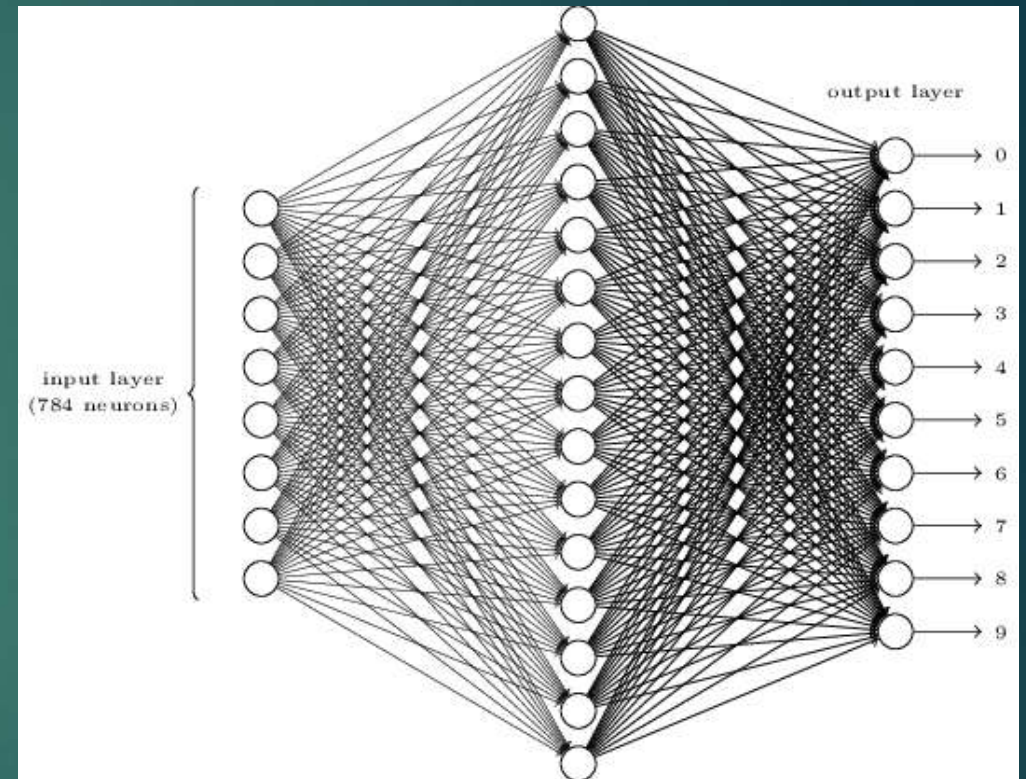


Dense layer

- ▶ The basic model, described in the examples above
- ▶ Also called **fully connected**
- ▶ Every unit has for inputs the whole previous layer
- ▶ Number of weights is the *product*

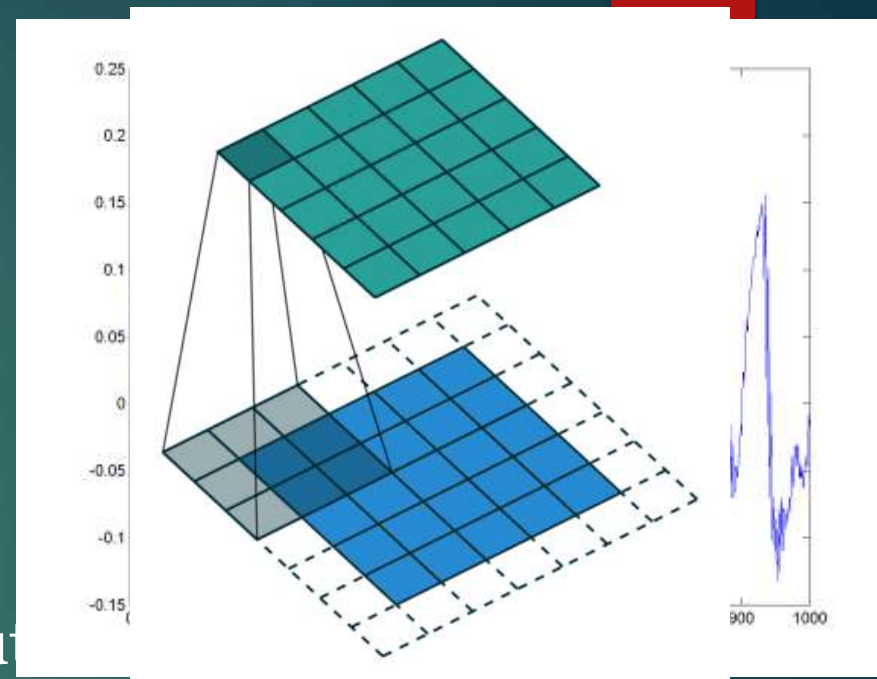
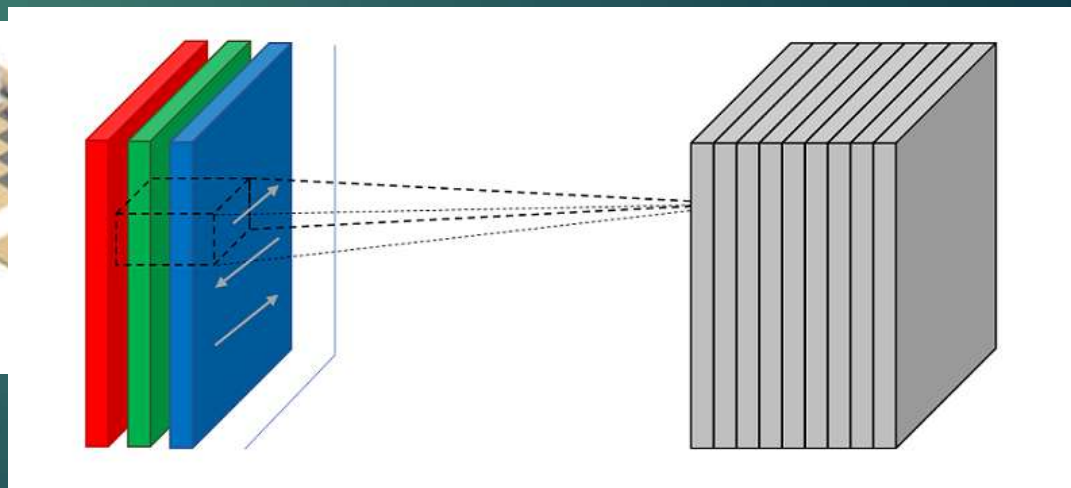
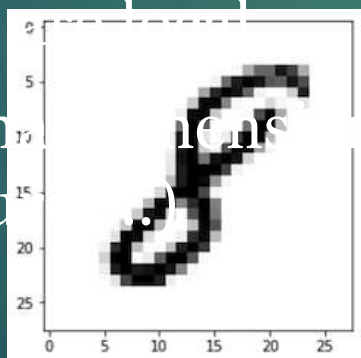
$$\text{hidden} = 28 \times 28 \times 128 = 100352$$

$$\text{output} = 128 \times 10 = 1280$$



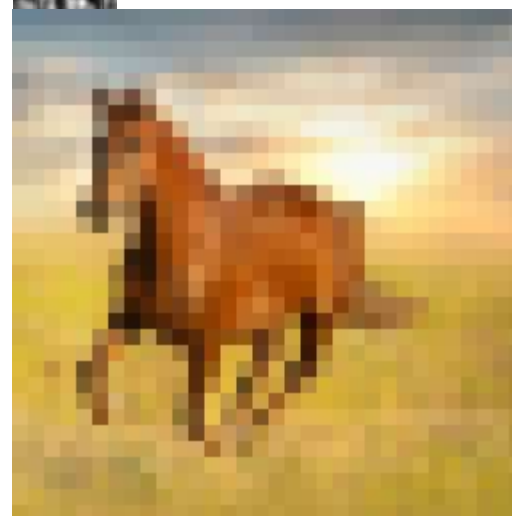
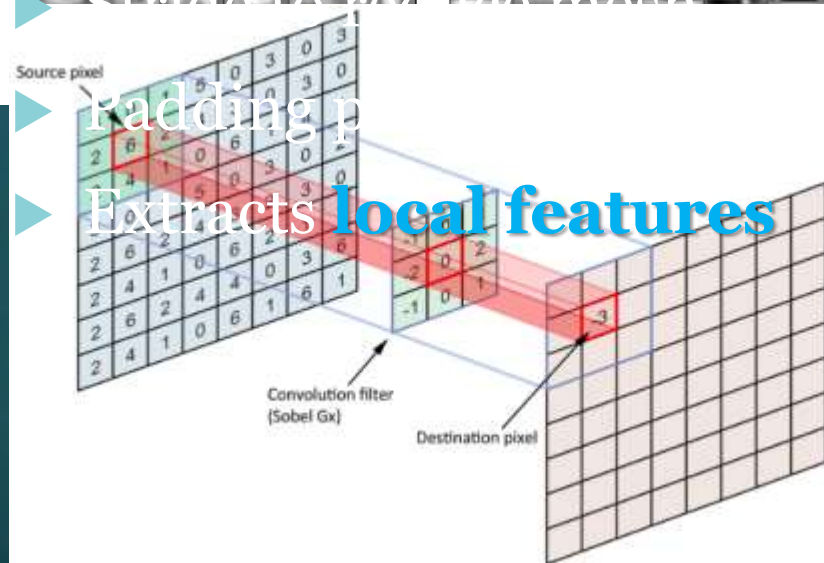
Convolutional layer

- ▶ Basic model for pictures
- ▶ Also for input data with *spatial structure*
- ▶ One, two or three spatial dimensions
- ▶ Units have same spatial structure as the input
- ▶ Connections are local
- ▶ One additional dimension is allowed (colors, features)



Convolutional layer

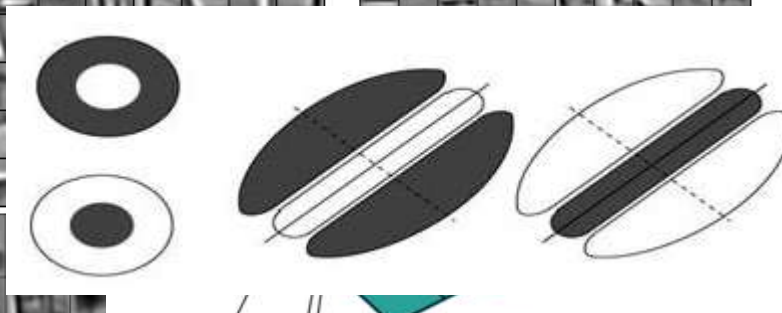
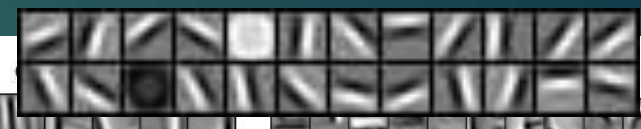
- ▶ Model expects invariance by translations
- ▶ Units *share weights* and have local inputs
- ▶ Typically a square of odd size
- ▶ Stride to reduce mesh



Visual

Retina and LGN

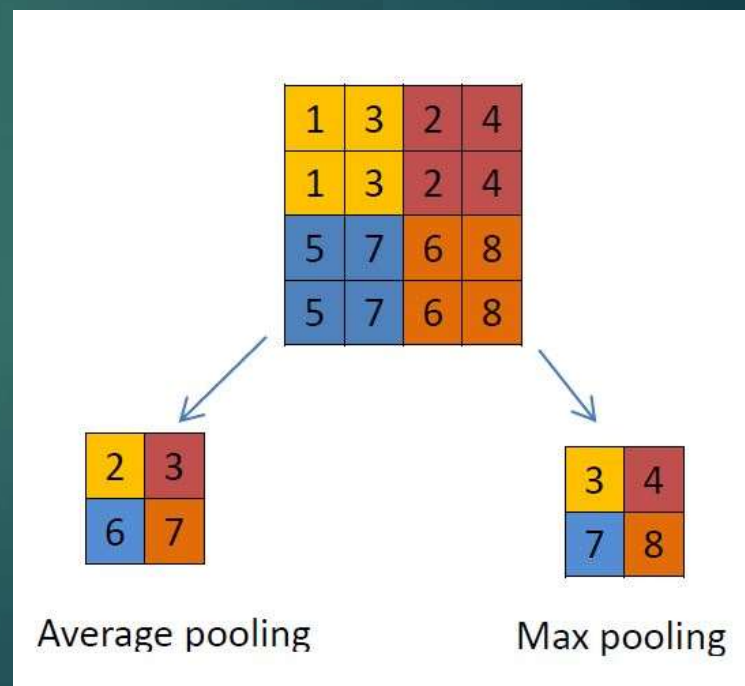
Primary visual cortex



r-complex

Pooling layer

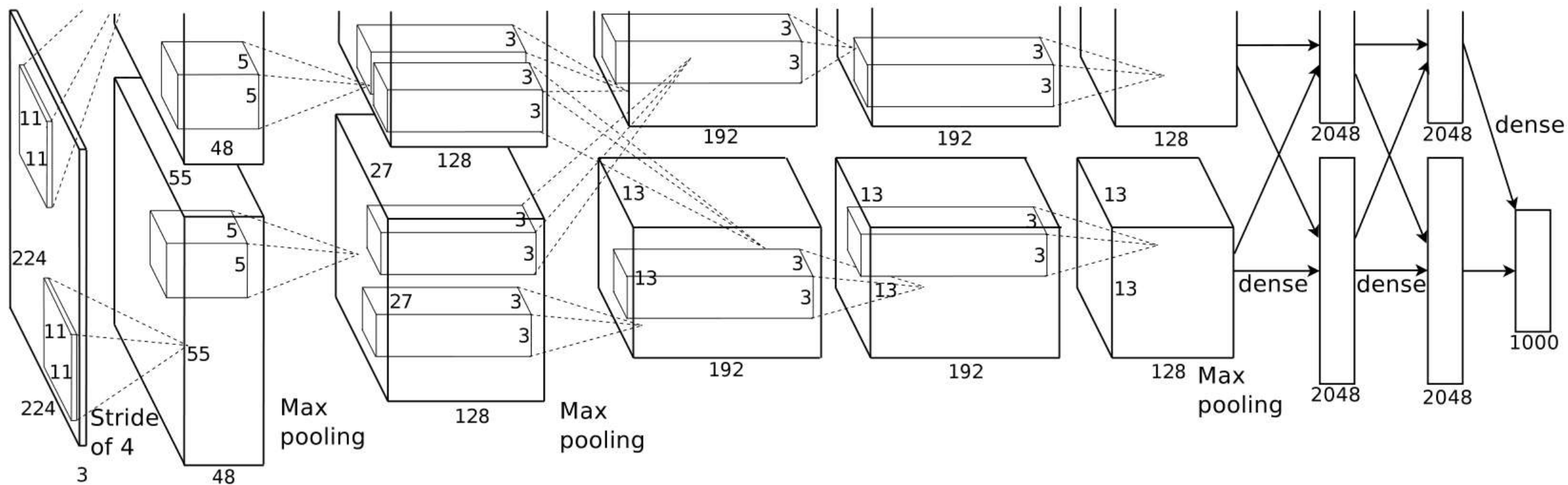
- ▶ For pictures and data with spatial structure
- ▶ Used to downsample preserving information
- ▶ Typically 2x2 boxes as input for units
- ▶ Operation is max or average



Bonus: AlexNet

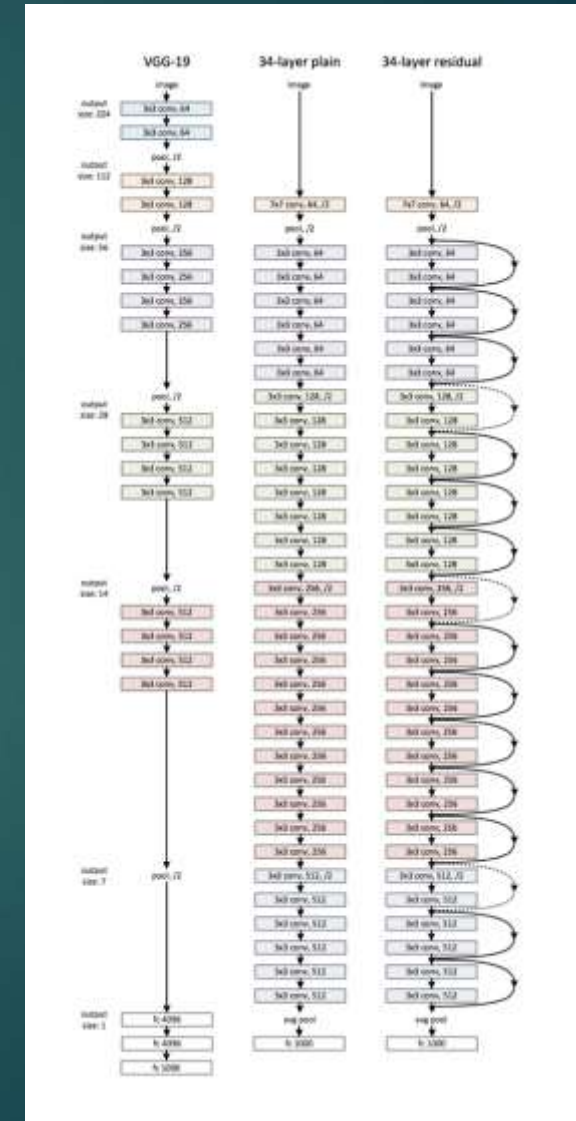
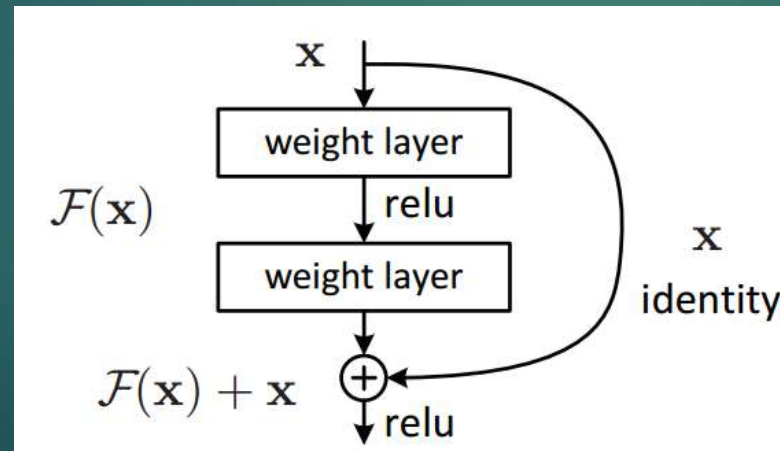
ImageNet competition results

0.5



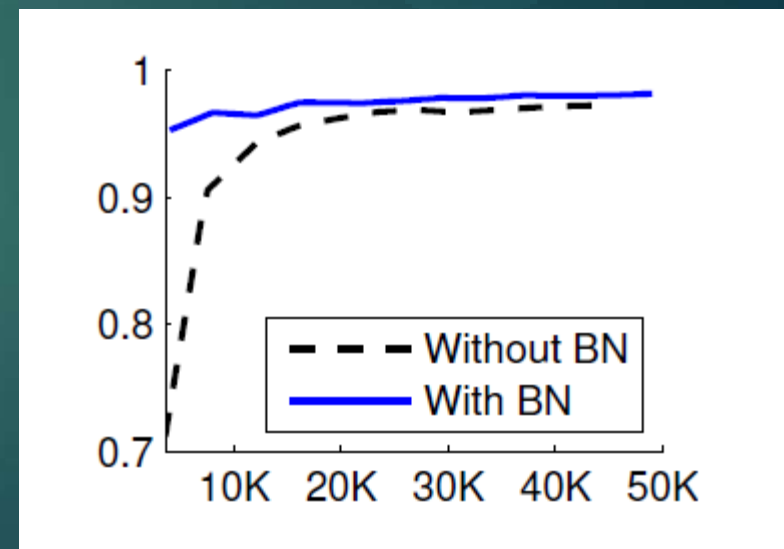
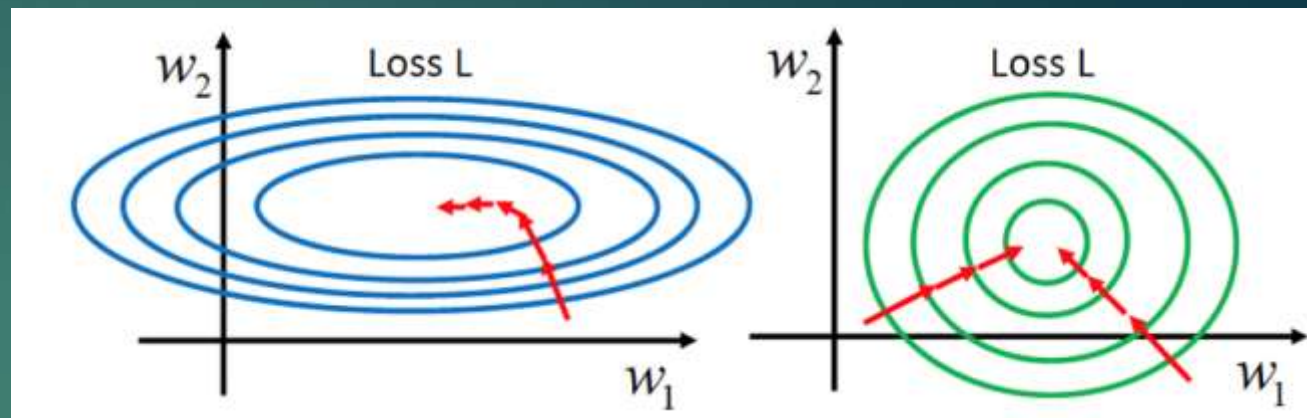
Residual layer

- ▶ It's a double layer with a skip connection
- ▶ Introduced in 2015 (won ILSVRC 2015)
- ▶ Improves deep networks training dramatically
- ▶ Allows depth of *hundreds* of layers



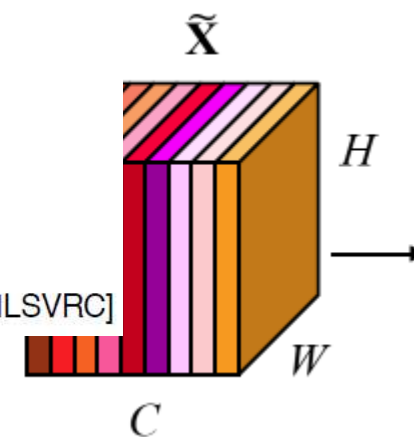
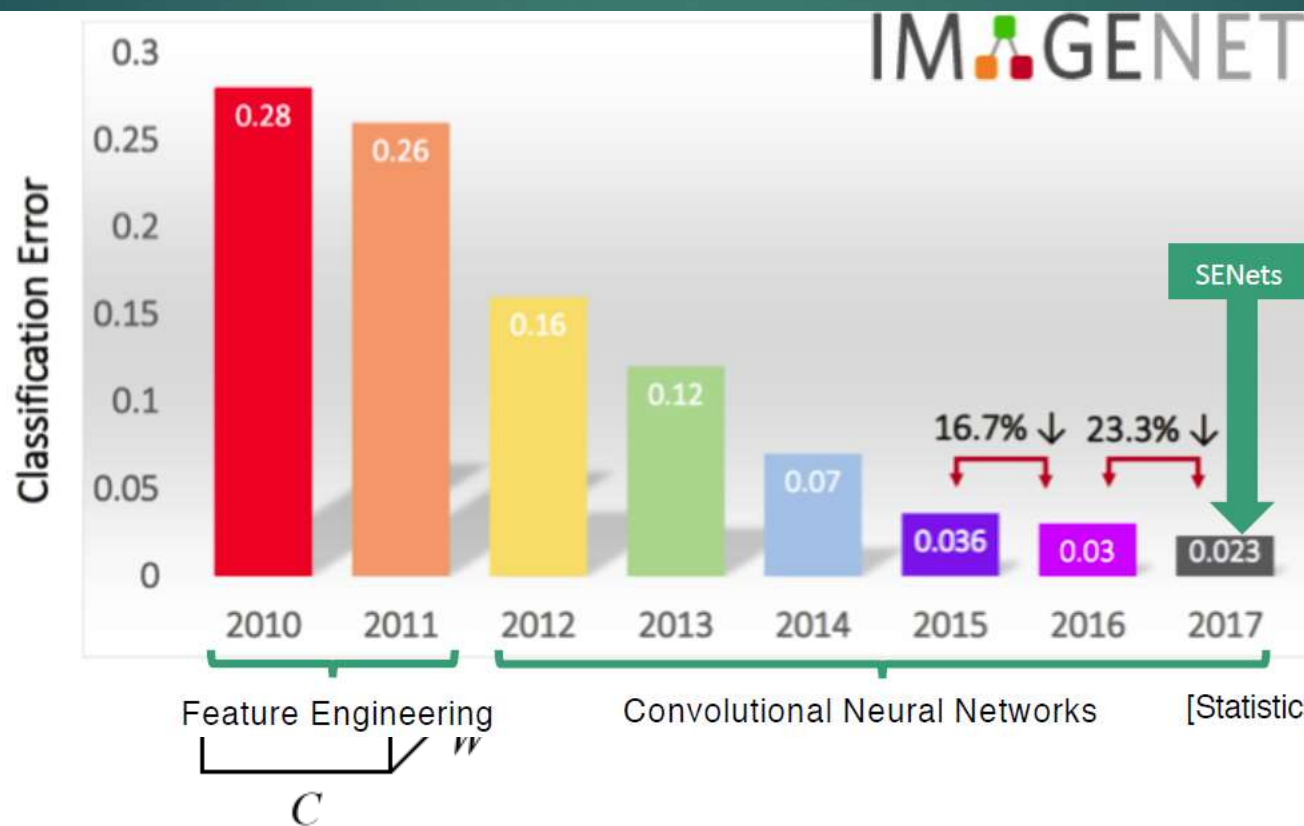
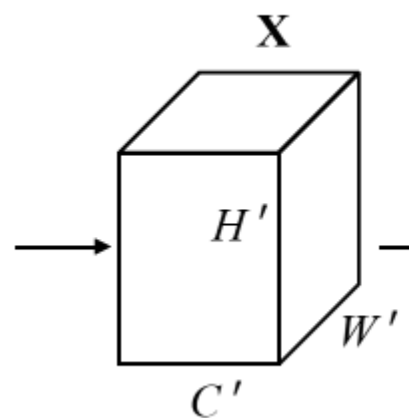
Batch normalization

- ▶ It's a procedure to keep numbers in a good range
- ▶ When the network is deep...
- ▶ ...small changes in a *lower* layer...
- ▶ ...reflect in large shifts in *higher* layers...
- ▶ ...making training very difficult
- ▶ BatchNorm standardizes the output of layers
- ▶ And **learns** the lost *scale* and *shift* parameters like weights



Squeeze-and-Excitation

- ▶ Introduced in 2017 (won and *closed* ILSVRC)
- ▶ Kind of
- ▶ Yields
- ▶ Based

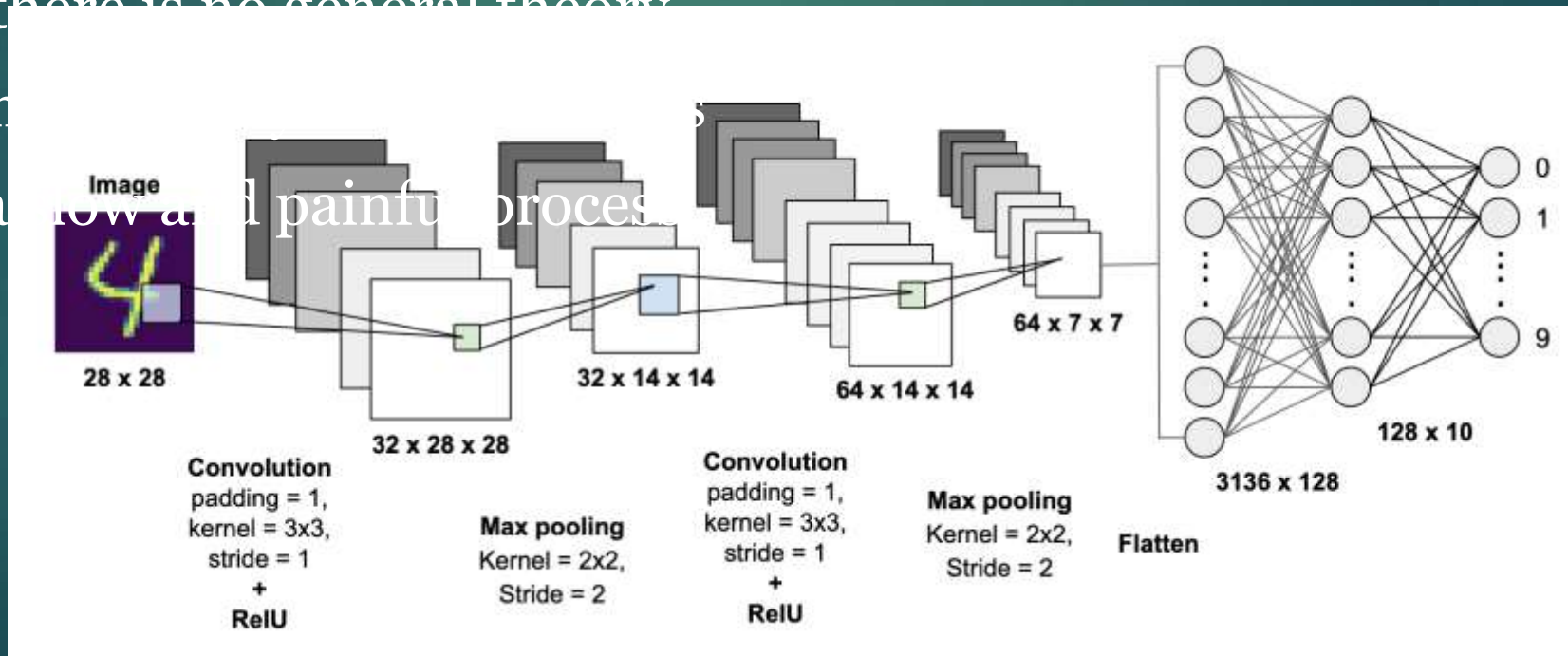




Tools of the trade

An experimental discipline

- ▶ Neural networks sort of work out-of-the-box
- ▶ But, there are *many choices*
- ▶ And there is no general theory
- ▶ One must experiment
- ▶ It is a slow and painful process



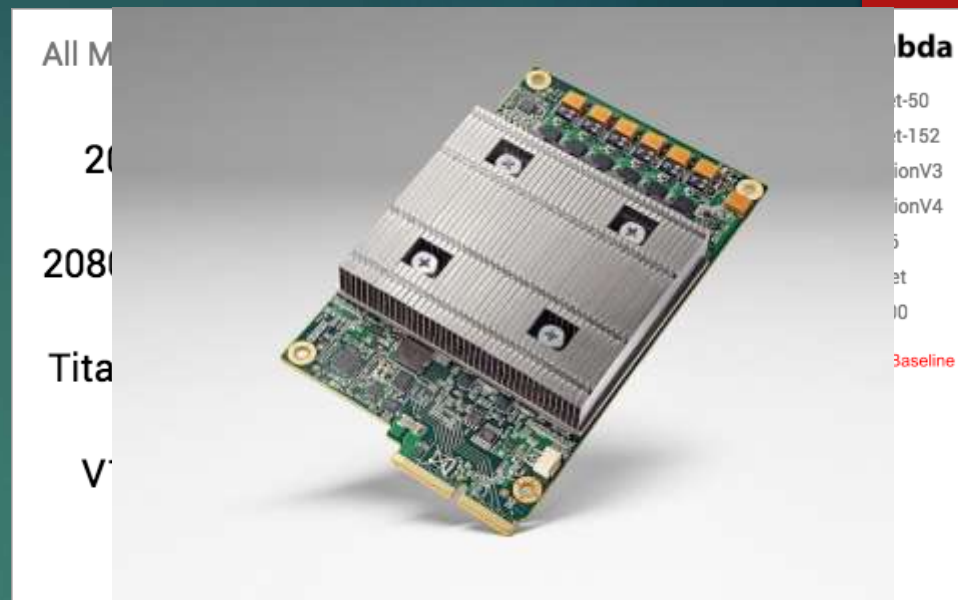
An experimental discipline

- ▶ Maybe you need to try different structures
- ▶ Different **hyperparameters**
- ▶ **Transform** variables
- ▶ **Augment** input data by transformations
- ▶ It is very important to have a platform:
 - fast and powerful
 - well organized



Computing power

- ▶ CPU is not enough
- ▶ GPU:
 - NVIDIA, and powerful
 - maybe AMD with ROCm
- ▶ TPU in the future, possibly
- ▶ Google Colab



Software, libraries

- ▶ Several platforms:
 - Caffe (2013)
 - Tensorflow (2015)
 - PyTorch (2016)
 - Keras (interface)
- ▶ Most are accessed in python
- ▶ Some possibilities in C++

The word "Caffe" in a red, serif font.The TensorFlow logo, featuring an orange 3D cube icon and the word "TensorFlow" in a dark blue, sans-serif font.The PyTorch logo, featuring an orange flame-like icon and the word "PyTorch" in a black, sans-serif font.The Keras logo, featuring a red square with a white letter "K" and the word "Keras" in a black, sans-serif font.The Python logo, featuring the blue and yellow snakes icon and the word "python" in a grey, sans-serif font, with a trademark symbol.

Google Colaboratory

- ▶ Powerful GPU/TPU for free
- ▶ Libraries pre-installed and working
- ▶ Jupyter notebook interface
- ▶ Allows fast experiments
- ▶ Let's get rocked!

Bibliography

- ▶ **Deep Learning** – by Goodfellow, Bengio, Courville – 2016
- ▶ **A 'Brief' History of Neural Nets and Deep Learning** – by Andrey Kurenkov – 2018
- ▶ **Internet**
- ▶ **Papers**