

# Neural Networks

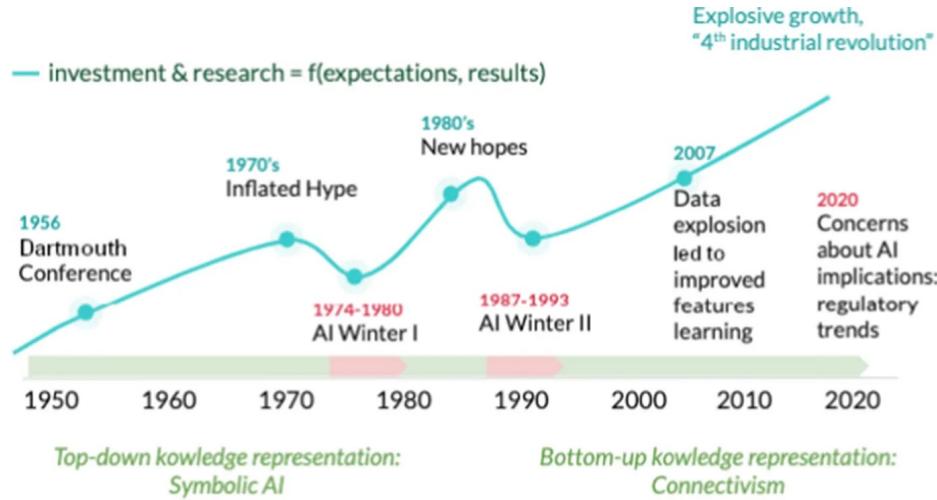
Battista Biggio



## Historical Notes on Neural Networks

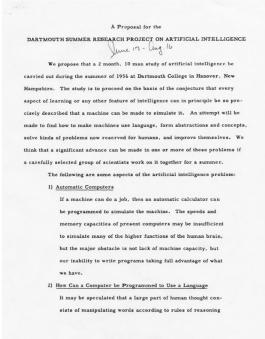
- Introduced in the 1950s as models of low-level human brain functions
  - neurons, networks of neurons
- Suited to numerical attributes
- Current evolution: **deep neural networks**, widely used in
  - Computer vision
  - Speech recognition

# Historical Notes on Neural Networks (and AI...)



From "The winter, the summer, and the summer dream of artificial intelligence in law", E. Francesconi

## 1955: Dartmouth Summer Research Project on AI



Left: Marvin Minsky, Claude Shannon, Ray Solomonoff and other scientists at the Dartmouth Summer Research Project on Artificial Intelligence (Photo: Margaret Minsky). Right: A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence (1955)

## Historical Notes on Neural Networks

- Inspired by findings in neuroanatomy and neurophysiology (19<sup>th</sup> – early 20<sup>th</sup> centuries)
- 1943: McCulloch and Pitts' model of neurons as *logic units*
- 1949: Hebb's model of changes in *synaptic strength* and *cell assemblies* as the origin of adaptation, learning and thinking
- 1957: Rosenblatt's **perceptron**:
  - network of McCulloch and Pitts' neural elements
  - *training procedures* for adjusting connection weights
  - applications to pattern recognition: *error-correction* training
  - perceptron learning algorithm

# Historical Notes on Neural Networks

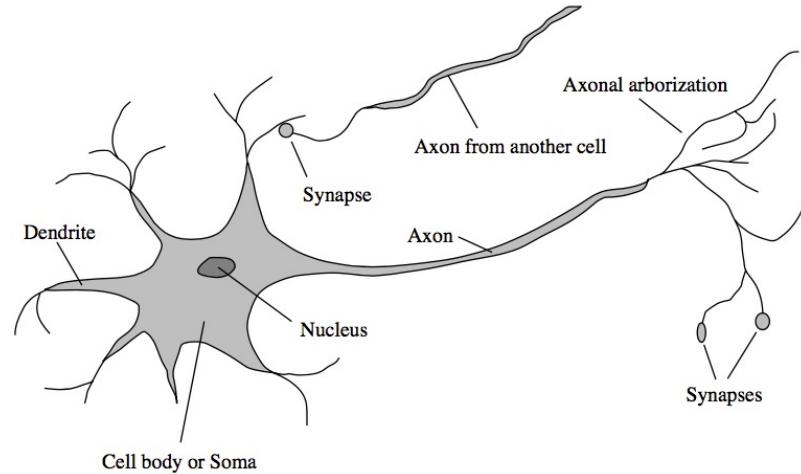
- 1970s:
  - limits of perceptrons (limited expressive power, lack of mathematical rigor)
  - drop of interest in neural networks [M.L. Minsky and S.A. Papert, *Perceptrons*, MIT Press, 1969]
- Mid–1980s:
  - *renaissance* of neural networks: the **connectionist** approach
  - learning algorithm: **back-propagation**
  - theoretical support: statistics, computational learning theory
  - Seminal work:
    - D.E. Rumelhart, J.L. McClelland (Eds.), *Parallel Distributed Processing*, MIT Press, 1986
    - D.E. Rumelhart, G.E. Hinton, R.J. Williams, *Learning representations by back-propagating errors*, Nature 323, 533-536, 1986

# Historical Notes on Neural Networks

- Since the 1990s - different kinds of ANNs:
  - Feed-Forward Multi-Layer Perceptron
  - Radial Basis Networks
  - Recurrent networks:
    - Long Short-Term Memory (LSTM) networks
    - Hopfield networks, associative memories
    - Boltzmann machines
- Applications in several fields: computer vision, pattern recognition, control systems, etc.
- Today: great interest on **deep** neural networks

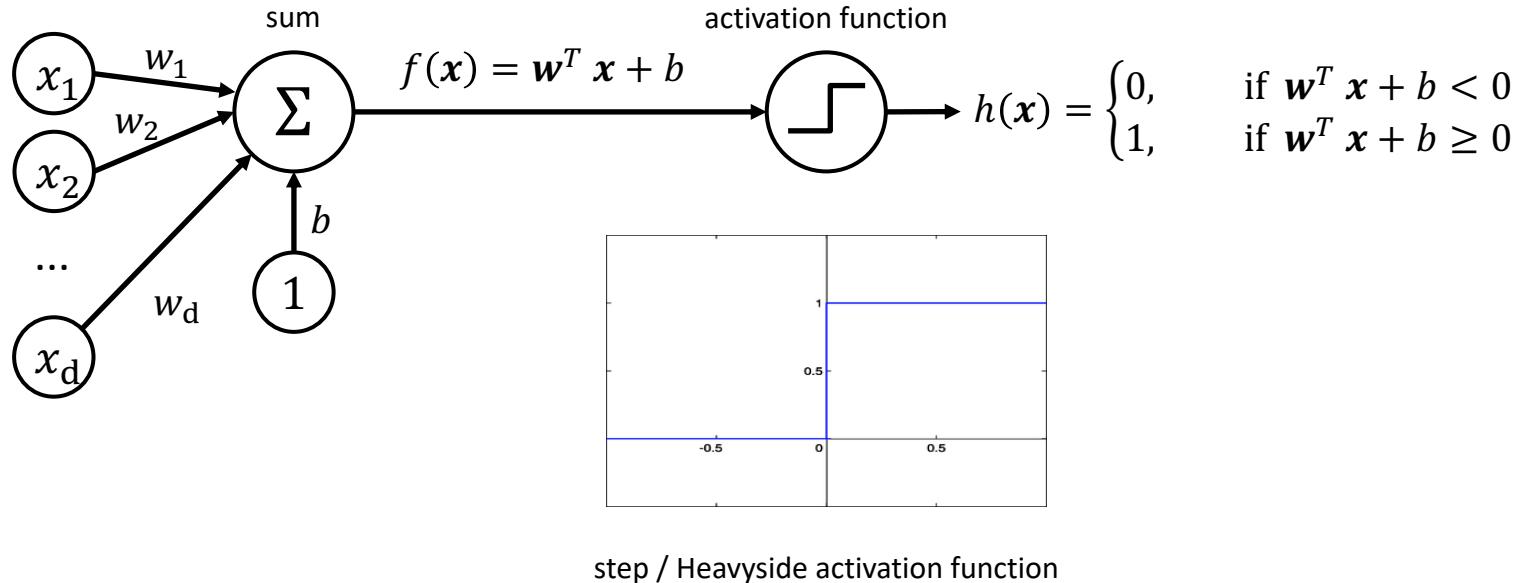
# Brains and Neurons

- Basic unit: **neuron** (nerve cell)
- $10^{11}$  neurons,  $10^{15}$  connections
- Apparently simple neuron behavior
  - *firing* (up to 103 Hz) in response to specific **patterns** of input signals
- Massive parallelism
- Robustness to *noise*
- *Learning capability*: novel connections between neurons and changing connection *strength* in response to external stimuli



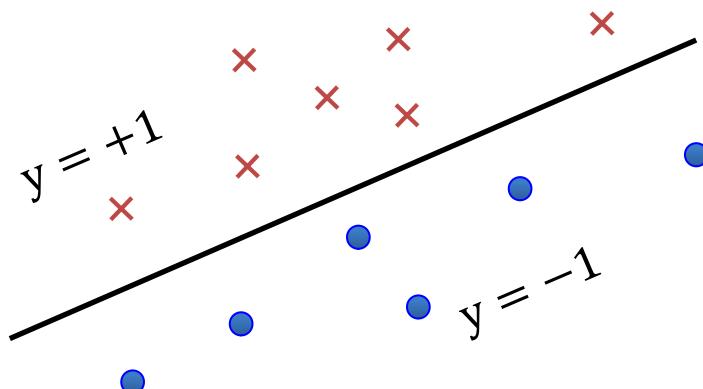
# The Perceptron

- McCulloch and Pitts' model of neurons as *logic units* (1943):



## Perceptron as a Linear Classifier

- $f(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + b$  corresponds to a linear discriminant function in feature space
- $f(\mathbf{x})$  is proportional to the distance of  $\mathbf{x}$  to the hyperplane
- $h(\mathbf{x})$  is equal to 1 in the positive-class region, and to 0 in the negative-class region



## The Perceptron Learning Algorithm

- It is known that experience and learning change (among other things) the connection *strength* between neurons in the human brain
- Can the perceptron emulate this behaviour?
- To this end, the **connection weights** must be modified by some *learning algorithm* to reproduce a desired input-output behaviour, according to a given set of examples
- Such an algorithm was devised by F. Rosenblatt in 1960

# The Perceptron Learning Algorithm

- Define a loss function (i.e., the perceptron loss)  
$$L(\mathbf{w}, b) = \sum_i \max(0, -y_i f(\mathbf{x}_i)) = -\sum_{i:y_i f(\mathbf{x}_i) < 0} y_i f(\mathbf{x}_i)$$
  - $i$  indexes the misclassified training samples

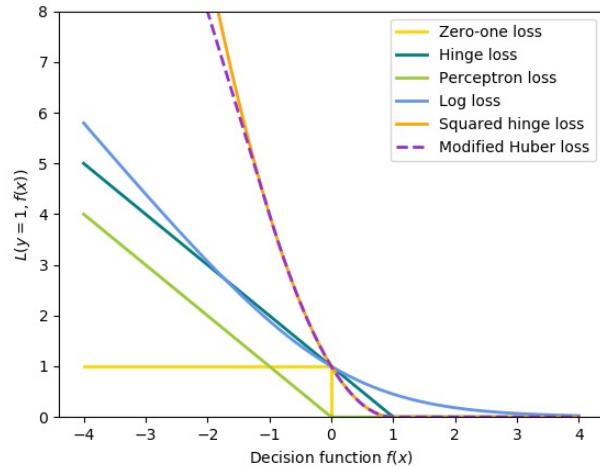
- Minimize it through gradient descent to find the hyperplane parameters

$$\mathbf{w}^*, b = \operatorname{argmin}_{\mathbf{w}, b} L(\mathbf{w}, b)$$

- Gradient computation (used to update the parameters)

$$\nabla_{\mathbf{w}} L(\mathbf{w}, b) = -\sum_{i:y_i f(\mathbf{x}_i) < 0} y_i \mathbf{x}_i$$

$$\nabla_b L(\mathbf{w}, b) = -\sum_{i:y_i f(\mathbf{x}_i) < 0} y_i$$



# The Perceptron Learning Algorithm

```
function Perceptron-Learning( $\{x_i, y_i\}_{i=1}^n$ )
    returns weight values  $w$ , bias  $b$ 
        randomly initialize  $w, b$ 
    repeat
        for i=1, ..., n
            if  $y_i f(x_i) < 0$ 
                 $w \leftarrow w - \eta \nabla_w L(w, b)$ 
                 $b \leftarrow b - \eta \nabla_b L(w, b)$ 
        until a stopping condition is satisfied
    return w
```

The parameter  $\eta$  is called ***learning rate***

Each for loop over the set of training samples is named ***epoch***

## Perceptron Convergence Theorem

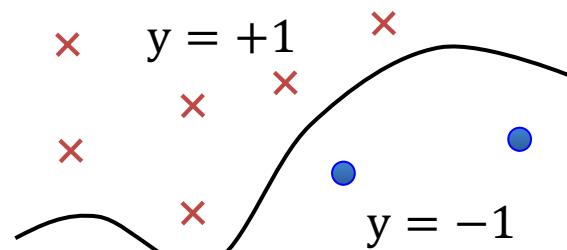
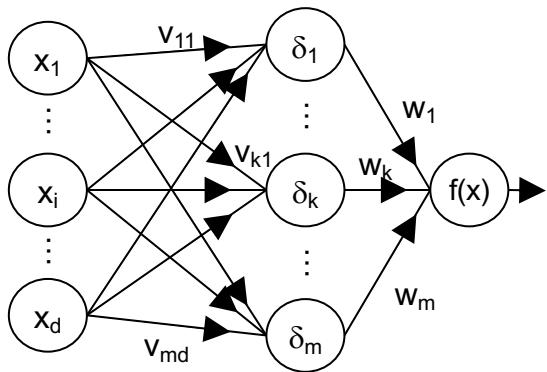
- **Perceptron convergence theorem** (F. Rosenblatt, 1960):  
If the training set is **linearly separable**, the perceptron learning algorithm **always** converges to a consistent hypothesis after a **finite** number of epochs, for **any**  $\eta > 0$ .
- As convergence is guaranteed for any  $\eta > 0$ , usually  $\eta = 1$  is set
- In practice we do not know in advance whether the training data is linearly separable
  - a suitable stop condition has to be used
- For instance, if the *training set* is **not** linearly separable, after a certain number of epochs the weights typically start *oscillating*
  - the detection of this behavior can be used as a stop condition

## Limits of the Perceptron

- The perceptron turned out to be an **oversimplified** model of neurons, not useful for understanding the human brain
- More accurate models are still being developed by neuroscientists
  - See, e.g., the Human Brain EU Project at <https://www.humanbrainproject.eu/en/>
- **Main limitation:** The perceptron can not represent non-linear functions of  $x$

# Perceptron Networks

- Neurons in the human brain are highly interconnected, yielding very complex *input-output* behaviors
- **Artificial neural networks** (ANN) made up of interconnected perceptrons can implement non-linear discriminant functions



## Issues with Perceptron Networks (prior to the 70s)

- No target output can be defined for **hidden** units
- Rosenblatt's algorithm cannot be applied to set the weights of their input connections
  - alternative algorithms devised in the 1970s exhibited a too high **computational complexity**
- How to define a suitable network **architecture**, i.e., the number of perceptrons and the connections among them, for a given application?
- Can the architecture itself be chosen by the learning algorithm, together with connection weights?
- These difficulties contributed to a drop of interest in neural networks in the 1970s

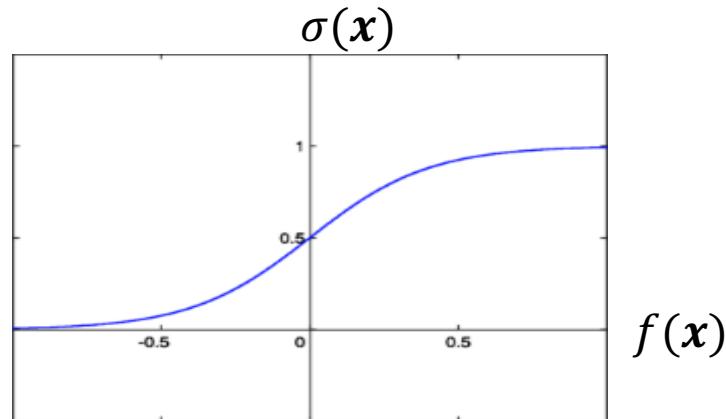
## The Renaissance of Neural Networks

- A practical solution to the aforementioned issues was found in the 1980s
- Learning the network architecture is too difficult
  - it is better to use a predefined architecture, and to learn only the connection weights
- Efficient learning algorithms can be devised for
  - specific architectures, like **feed-forward** networks
  - **continuous** activation functions, instead of the step function

## Continuous Activation Functions

- Sigmoid (or Logistic)

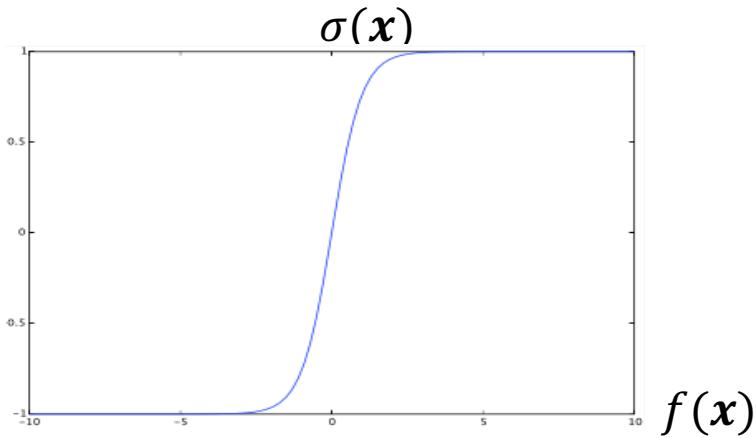
$$\sigma(x) = \frac{1}{1 + e^{-f(x)}} \in (0,1)$$



# Continuous Activation Functions

- Hyperbolic Tangent ( $\tanh$ )

$$\sigma(x) = \frac{e^{f(x)} - e^{-f(x)}}{e^{f(x)} + e^{-f(x)}} \in (-1,1)$$

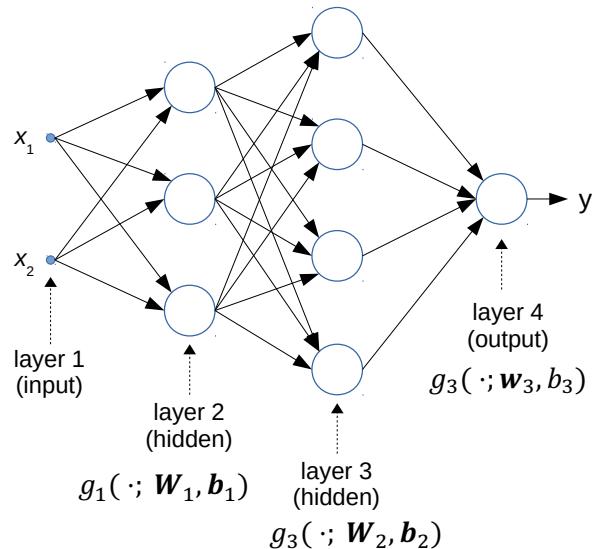


# Feed-forward Multi-layer Perceptron (MLP)

- Units are arranged into **layers** (hence the name *multi-layer*)
  - **input** layer: fictitious units corresponding to inputs
  - **output** layer: one unit for two-class problems,  $c$  units for  $c$ -class problems with  $c > 2$
  - one or more **hidden** layers
- No recurrent connections (hence the name *feed-forward*)
  - every hidden and output unit receives **inputs** only from units of the **previous** layer
  - every input and hidden unit sends its **output** only to units in the **next** layer
- Usually these networks are **fully-connected**
  - every input and hidden unit is connected to **all** units in the next layer

## MLP: An Example

- Fully-connected MLP with two inputs, two hidden layers of three and four neurons each, and one output unit (suitable for two-class classification)



$$\begin{aligned}f(\mathbf{x}; \boldsymbol{\theta}) &= g_3(\cdot; \mathbf{w}_3, b_3) \circ g_2(\cdot; \mathbf{W}_2, \mathbf{b}_2) \circ g_1(\mathbf{x}; \mathbf{W}_1, \mathbf{b}_1) \\&= g_3(g_2(g_1(\mathbf{x}; \mathbf{W}_1, \mathbf{b}_1); \mathbf{W}_2, \mathbf{b}_2); \mathbf{w}_3, b_3)\end{aligned}$$

$$g(\cdot; \mathbf{W}, \mathbf{b}) = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

$$\mathbf{W}_1 \in \mathbb{R}^{3 \times 2}, \quad \mathbf{b}_1 \in \mathbb{R}^3$$

$$\mathbf{W}_2 \in \mathbb{R}^{4 \times 3}, \quad \mathbf{b}_2 \in \mathbb{R}^4$$

$$\mathbf{w}_3 \in \mathbb{R}^4, \quad b_3 \in \mathbb{R}$$

## MLP Output Layer

- $\sigma(\mathbf{x})$  is a continuous output, it has to be converted to a class label
- For two-class problems, it can be thresholded
  - For example, if the sigmoid is used,  
 $\sigma(\mathbf{x}) < 0.5$  corresponds to label -1, and +1 otherwise
- For multi-class problems with  $c > 2$  classes,  $c$  output neurons can be used and combined in a one-vs-all manner
- A *softmax* layer can also be used to estimate the class posterior probabilities as:

$$p(y = j | \mathbf{x}) = \frac{e^{f_j(\mathbf{x})}}{\sum_{k=1}^c e^{f_k(\mathbf{x})}}$$

- The *softmax* function generalizes the logistic/sigmoid function to the multi-class case

## Selecting the Neural Network Architecture

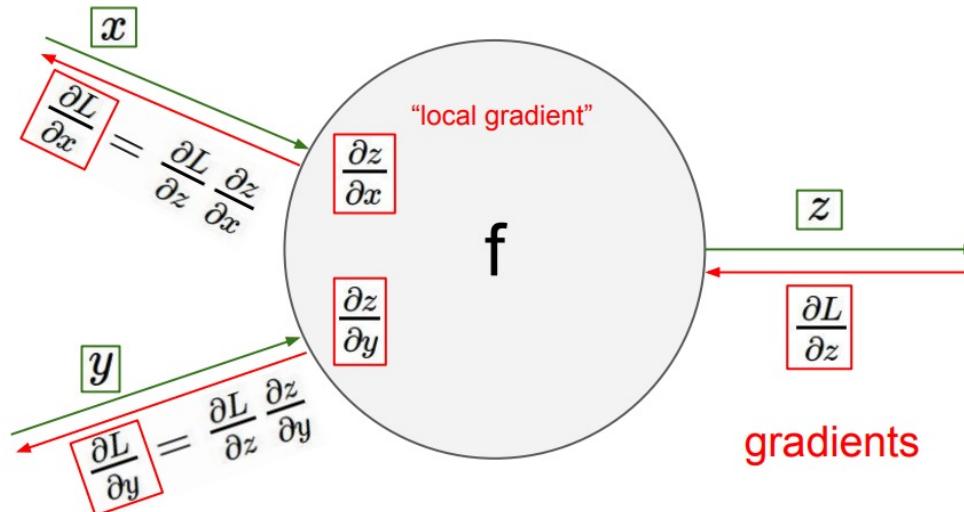
- In practical classification problems the target function (i.e., the relationship between feature values and class label) is **unknown**
- This means that also the best MLP architecture (in terms of number of hidden layers and of hidden units) is unknown
- An iterative trial-and-error design approach is usually adopted
  - start with a small network (e.g., one hidden layer and a few hidden units)
  - evaluate increasingly complex architectures (by increasing the number of hidden units and/or hidden layers) until a desired generalization capability is attained
- This agrees with the Occam's razor, promoting simpler solutions over complex ones

## Back-Propagation Learning

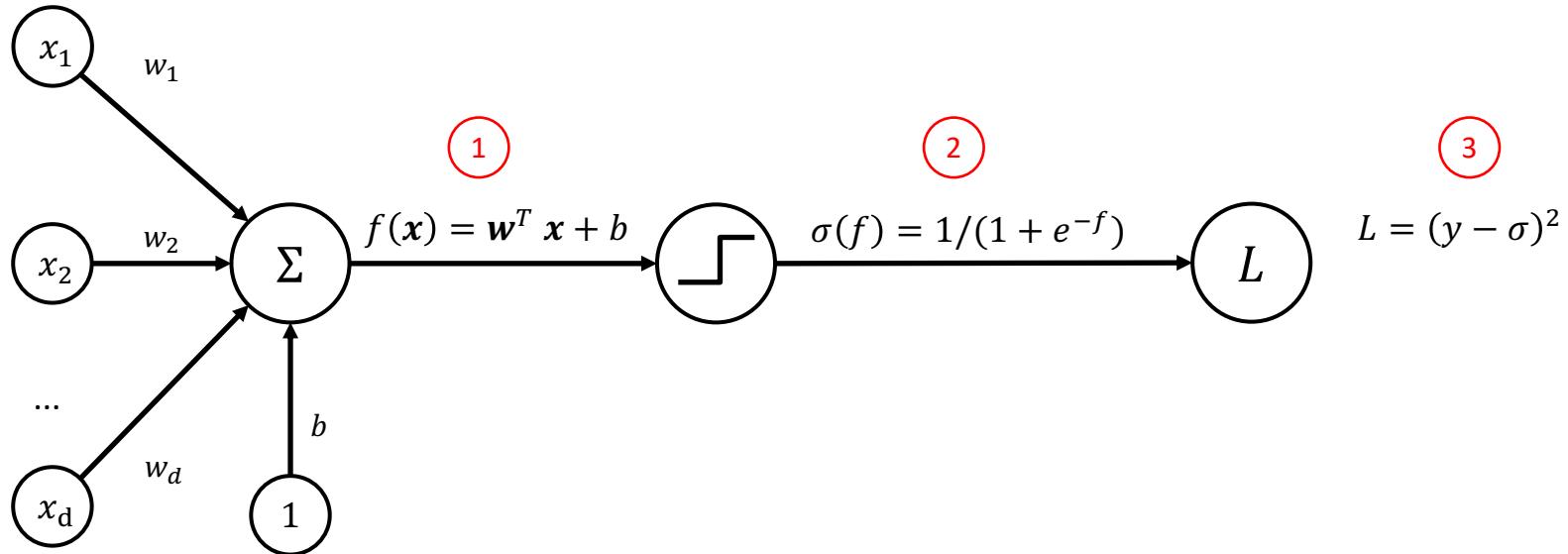
- Efficient learning algorithm devised in the 1980s for feed-forward MLP networks
- It exploits the continuous, differentiable activation function of each unit, which makes the network output a **continuous** and **differentiable** function of the network inputs
- To this end:
  - a **continuous, differentiable loss function** is used to evaluate the difference between the network output and the target output of a given example (e.g., MSE)
  - the loss function can be minimized over a training set using a gradient-based procedure

# Back-Propagation Learning: Intuition

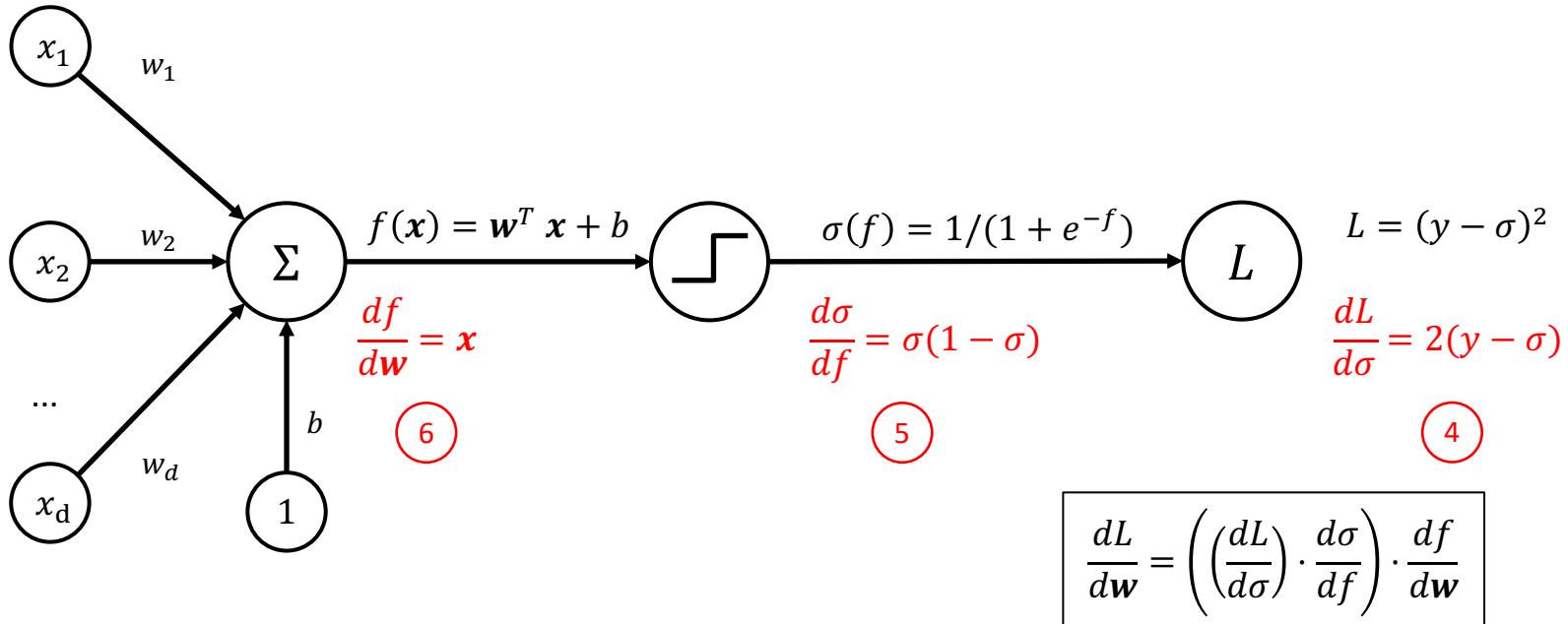
- **Forward step:** compute output function (e.g., loss) given input (e.g., one sample)
- **Backward step:** compute gradient in reverse-mode automatic differentiation, via the chain rule; e.g.,  $dL/dx = dL/dz * dz/dx$



## Forward Pass



## Backward Pass

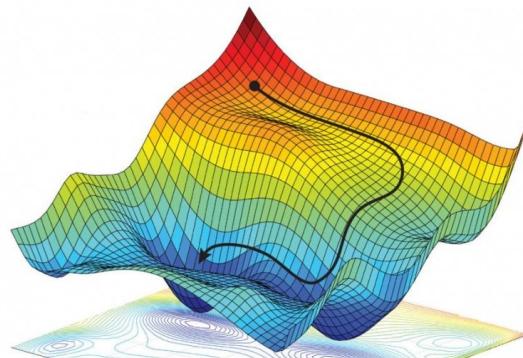


# Back-Propagation Learning

```
function BACK-PROPAGATION ( $T$ )
returns weight values  $w$ 
    randomly choose the initial weight values  $w$ 
    repeat
        for each  $(x^k, t_k) \in T$  do
            compute the network output  $y(x^k)$  (forward-propagation)
            update the weights  $w$  (back-propagation)
        end for
        until a stopping condition is satisfied
    return  $w$ 
```

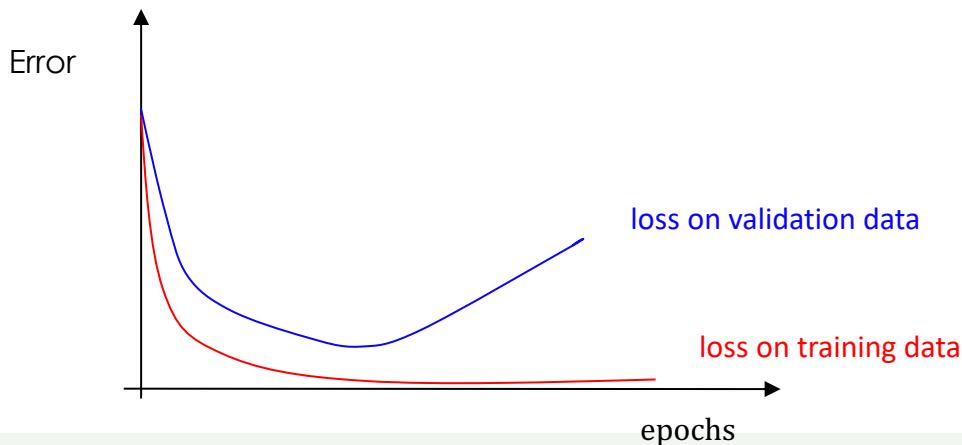
# Back-Propagation Learning: Issues

1. The error function usually has many **local minima**
  - gradient descent does not guarantee convergence to the smallest error on training examplesA **multi-start** strategy can be used to mitigate this problem
  - the algorithm is run several times starting from **different** random weights, and the solution with minimum error is chosen
2. Like all classifiers also NNs are prone to **over-fitting**  
**Early-stopping** can be used to tackle this issue



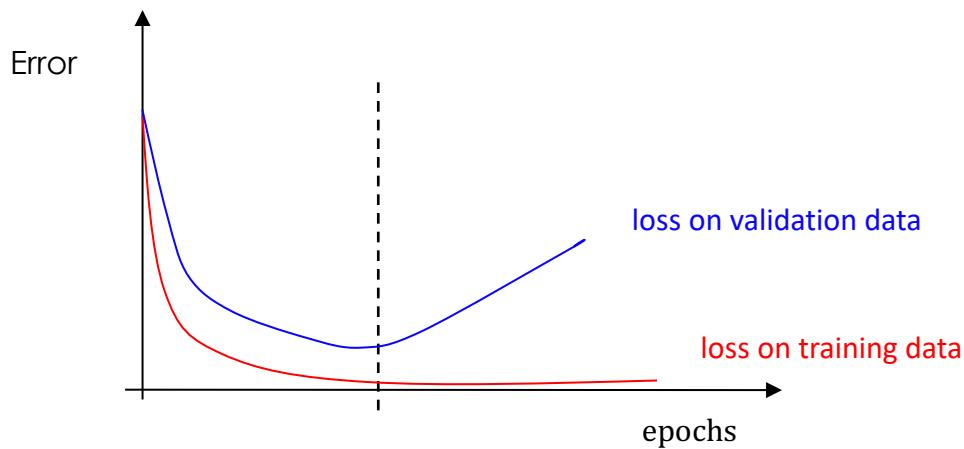
# Overfitting

- Overfitting can be detected by running the back-propagation algorithm on a training set made up of a **subset** of the available examples, and by evaluating the error function also on (a subset of) the remaining examples, called **validation set**
- If the loss on validation data starts increasing, overfitting is occurring



## Early Stopping

- The back-propagation algorithm is stopped when the error function starts increasing on validation examples



# Regularization

- Another way of mitigating overfitting is to use a *regularized* objective function

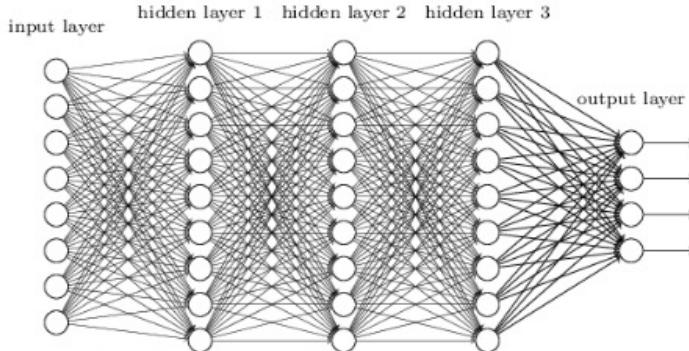
$$E(\mathbf{w}) = \frac{1}{2n} \sum_{i=1}^n (a(\mathbf{x}_i) - y_i)^2 + \lambda \underbrace{\Omega(\mathbf{w})}_{\text{regularization term}}$$

- $\lambda$  is a trade-off parameter between loss on training data and weight regularization
  - l1 and l2 norms are typically used for weight regularization

# Deep Learning

# Deep Neural Networks (DNNs)

- DNNs are a recent popular extension of NNs (but early ideas date back to the 1970s), inspired by the structure of the brain
- Basically, they are multi-layer networks with **many** hidden layers



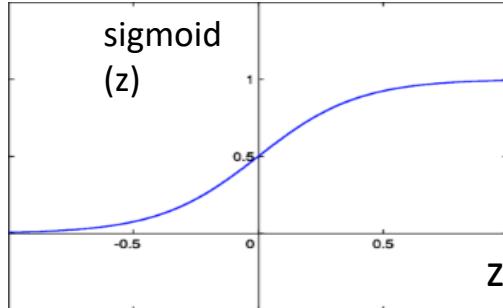
taken from <http://neuralnetworksanddeeplearning.com>

- *Ad-hoc* modifications to activation functions and training procedures have been introduced to avoid drawbacks of standard ones (e.g., very slow convergence)

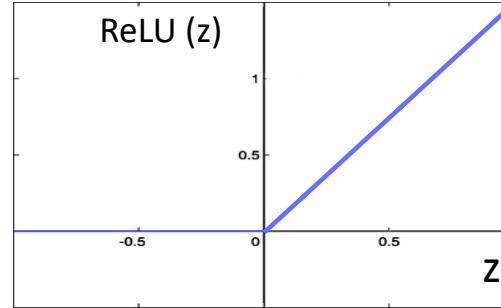
# ReLU Activations

- **Problem:** sigmoid takes on values in  $(0,1)$ . Its gradient  $z(1-z)$  is closer to 0 than  $z$ .
  - The net effect is that, propagating the gradient back to the initial layers, it tends to become 0 (**vanishing gradient problem**).
  - From a practical perspective, this slows down the training procedure of the initial layers
- Rectified Linear Unit (**ReLU**) activations can overcome this issue

$$\text{sigmoid}(z) = \frac{1}{1+e^{-z}}$$

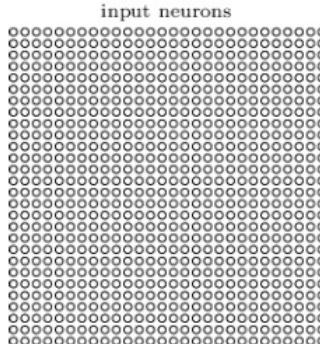


$$\text{ReLU}(z) = \max(0, z)$$



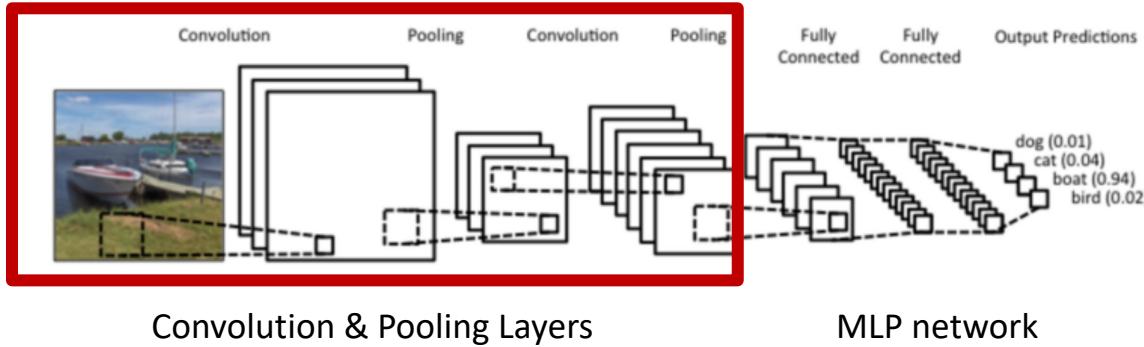
# Convolutional Neural Networks

- DNNs are widely employed for computer vision tasks, for which specialized architectures have been proposed, named ***convolutional neural networks*** (CNNs)
- In this case the CNN input is a raw image. Accordingly, the CNN architecture is not fully connected, but it exploits the **spatial adjacency** between pixels. Input units are arranged into an array



taken from <http://neuralnetworksanddeeplearning.com>

# Convolutional Neural Networks



# Convolutional Neural Networks

- CNNs hidden layers carry out specific image processing operations. They basically alternate two kinds of layers:
  - **filtering** layers, whose connection weights (that determine the filter implemented), are **learnt**
  - **pooling** layers, which have predefined connection weights, and carry out a downsampling operation on the outputs of the previous layer
- The upper layers consist of a standard, fully-connected feed-forward network
- Key difference between standard (*shallow*) and deep networks is that deep models aim to learn the feature representation (end-to-end learning) rather than using handcrafted / engineered features
  - CNNs do so by stacking convolution and pooling layers

# Filtering (Convolutional Units)



I

-1	-1	-1
2	2	2
-1	-1	-1

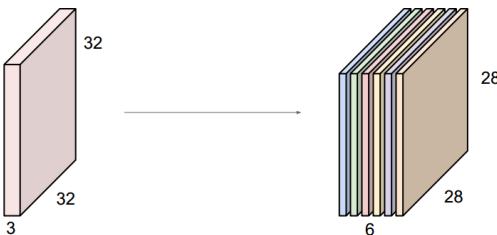
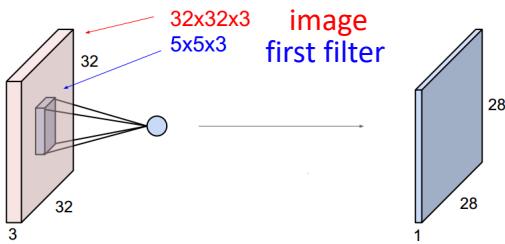
K



I \* K

# Filtering (Convolutional Units)

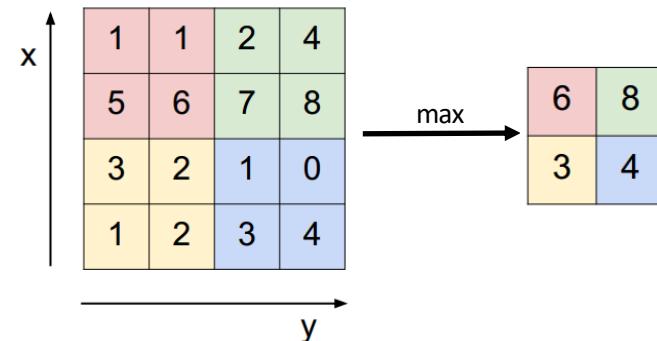
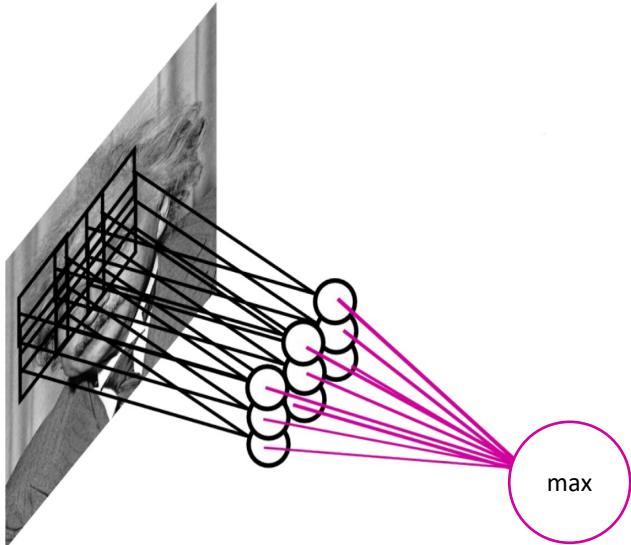
- Normally, several filters are packed together and learnt automatically during training
  - For more details on convolutions, see: <https://cs231n.github.io/convolutional-networks/>



- Conv. filter/kernel:  $5 \times 5 \times 3$
- Stride (how much we slide the filter)
- Padding (zeros around the border)
- Depth: number of filters

# Pooling

- Max pooling is a way to simplify the network architecture, by downsampling the number of neurons resulting from filtering operations

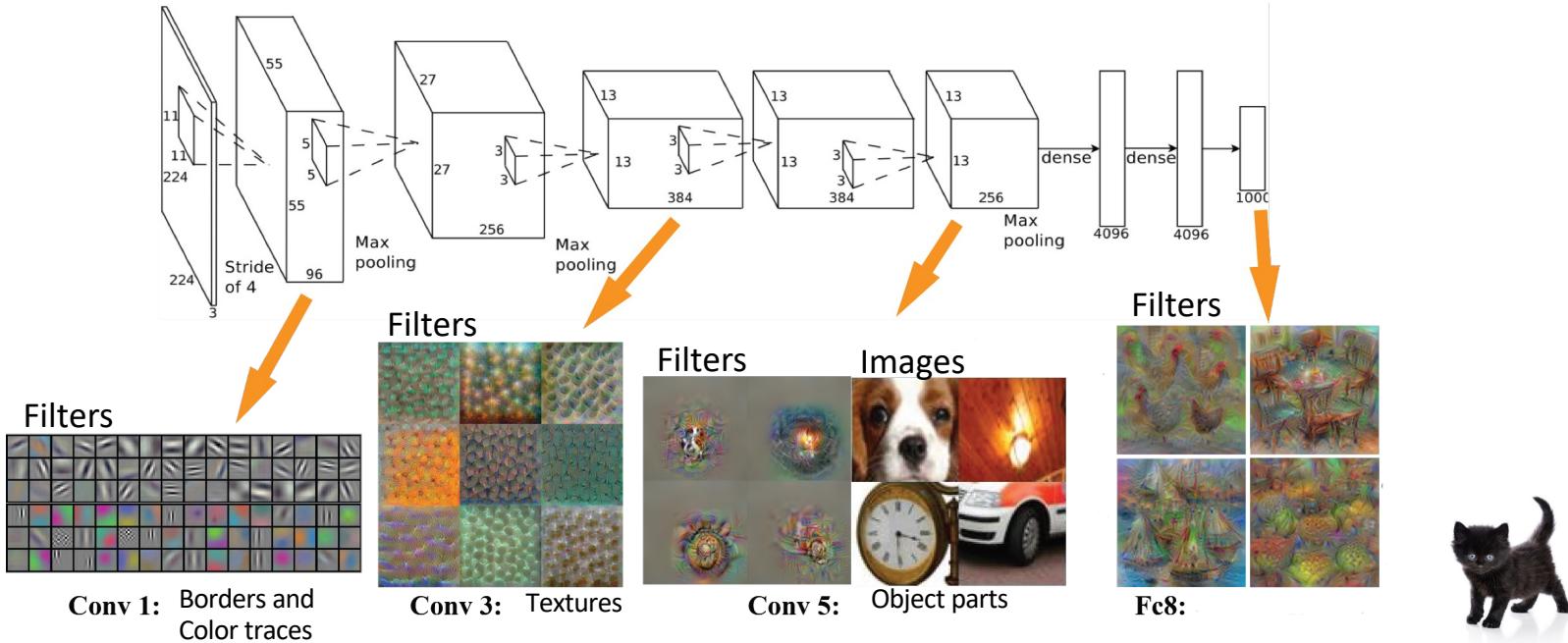


Slide credits: M. A. Ranzatto e Fei Fei Li

open:campus

# Convolutional Neural Networks

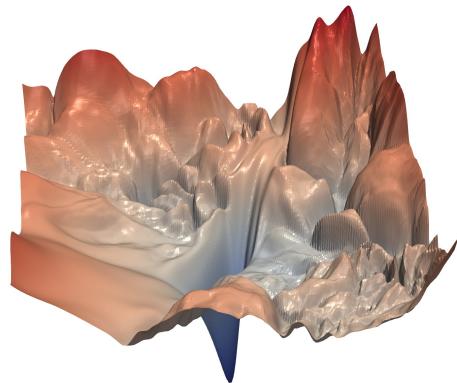
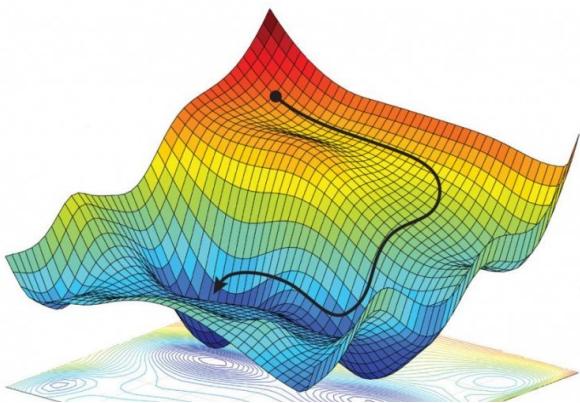
- The deep network gradually learns more complex and abstract notions



# Optimization

# Optimizing Deep Networks

- Neural networks aim to learn a mapping function  $f(x; \theta) = g_k \circ g_{k-1} \circ \dots \circ g_1(x; W_1, b_1)$  which is typically not convex w.r.t.  $\theta = (W_1, b_1 \dots, W_k, b_k)$
- Learning the classifier parameters (e.g., via cross-entropy loss minimization) is then much more challenging, as it requires solving a non-convex optimization problem



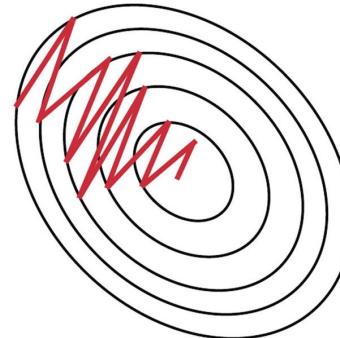
Visualizing the loss landscape of NNs, <https://arxiv.org/pdf/1712.09913.pdf>

# Optimizing Deep Networks

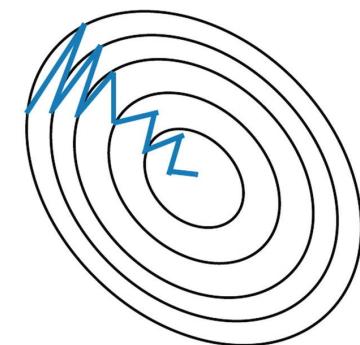
- Non-convex problem with large number of parameters and data points (up to millions)
  - The data may not even fit in memory
  - Easy to get stuck in bad local minima, slow convergence rates for gradient-based methods
- For this reason, many variants of *online* gradient-based optimizers have been proposed
  - Data is loaded in batches
  - Gradient is computed for the current batch and used to update the parameters (similarly to Stochastic Gradient Descent, SGD)
- **Most popular ones:** Momentum, Adam, Adagrad, RMSProp, etc.
  - [https://d2l.ai/chapter\\_optimization/index.html](https://d2l.ai/chapter_optimization/index.html)

# Momentum

- The gradient  $\mathbf{g}_k$  is averaged across iterations, with weight  $\beta$ 
  - $\mathbf{v}_k = \beta\mathbf{v}_{k-1} + \mathbf{g}_k$
  - $\theta_k = \theta_{k-1} - \eta_k \mathbf{v}_k$
- With  $\beta=0$ , it is equivalent to the steepest descent method
- Momentum works as a smoothing operator on the objective function, facilitating convergence



Stochastic Gradient Descent **without** Momentum



Stochastic Gradient Descent **with** Momentum

More at: [https://d2l.ai/chapter\\_optimization/momentum.html](https://d2l.ai/chapter_optimization/momentum.html)

# Dropout and Batch Normalization

- Techniques/regularizers to facilitate training
- **Dropout:** randomly de-activate some neurons during training (to prevent overfitting)
- **Batch Normalization:** normalize inputs to have zero mean and unit variance
  - mean and variance are estimated for each batch separately

# Open-source Frameworks

- Good news: DNN learning is widely automated nowadays
  - Many available open-source frameworks (Tensorflow, PyTorch, Keras, MXNet, ...)
- Basic idea: model as a computational graph (i.e., a sequence of instructions!)
  - Each node represents a function that transforms the input data (tensors)
  - Autodifferentiation is readily available – we can differentiate through computer programs!
- Computational advantages
  - Gradients are computed efficiently and readily available
  - Parallel computation via GPUs (very efficient for parallel matrix multiplication)
  - Horizontal scalability / cloud computing
- Models can be served in production

# TensorFlow

The TensorFlow homepage features a large orange header with the text "An end-to-end open source machine learning platform". Below this, a white navigation bar includes links for "Install", "Learn", "API", "Resources", "Community", and "Why TensorFlow". A search bar with the placeholder "Cerca" and a language dropdown are also present. The main content area contains a "TensorFlow" tab, "For JavaScript", "For Mobile & IoT", and "For Production" tabs. A descriptive paragraph states: "The core open source library to help you develop and train ML models. Get started quickly by running Colab notebooks directly in your browser." A prominent "Get started with TensorFlow" button is located below this text. To the right, a stylized diagram illustrates the platform's integration across various devices and applications, including a laptop, smartphone, server, car, airplane, and butterfly, connected by a network of lines.

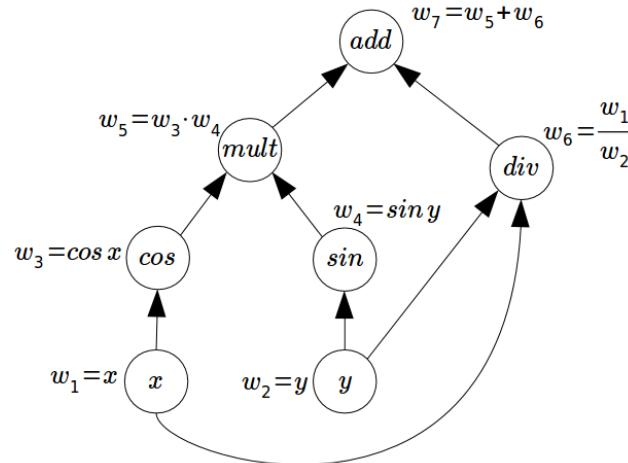


## KEY FEATURES & CAPABILITIES

[See all Features >](#)

# Automatic/Algorithmic Differentiation

- Gradients are readily available from the computation graph
  - *Dual numbers*: each node evaluates the function along with its gradient
  - Derivative of output w.r.t inputs is obtained through the chain rule
- Forward-mode autodifferentiation
  - accumulates derivatives going forward, initializing x and y
  - convenient when n\_inputs << n\_outputs
- Reverse-mode autodifferentiation
  - accumulates derivatives going backward, initializing the output
  - convenient when n\_outputs << n\_inputs



$$f(x,y) = \cos x \sin y + x/y$$

Read more at: <http://jmlr.org/papers/volume18/17-468/17-468.pdf>  
<https://mathematical-tours.github.io/book-sources/optim-ml/OptimML.pdf>

# Deep Learning with PyTorch

[https://pytorch.org/tutorials/beginner/blitz/cifar10\\_tutorial.html](https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html)

## 1. Load data

```
transform = transforms.Compose(  
    [transforms.ToTensor(),  
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])  
  
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,  
                                         download=True, transform=transform)  
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,  
                                         shuffle=True, num_workers=2)  
  
testset = torchvision.datasets.CIFAR10(root='./data', train=False,  
                                         download=True, transform=transform)  
testloader = torch.utils.data.DataLoader(testset, batch_size=4,  
                                         shuffle=False, num_workers=2)  
  
classes = ('plane', 'car', 'bird', 'cat',  
          'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

## 2. Define network

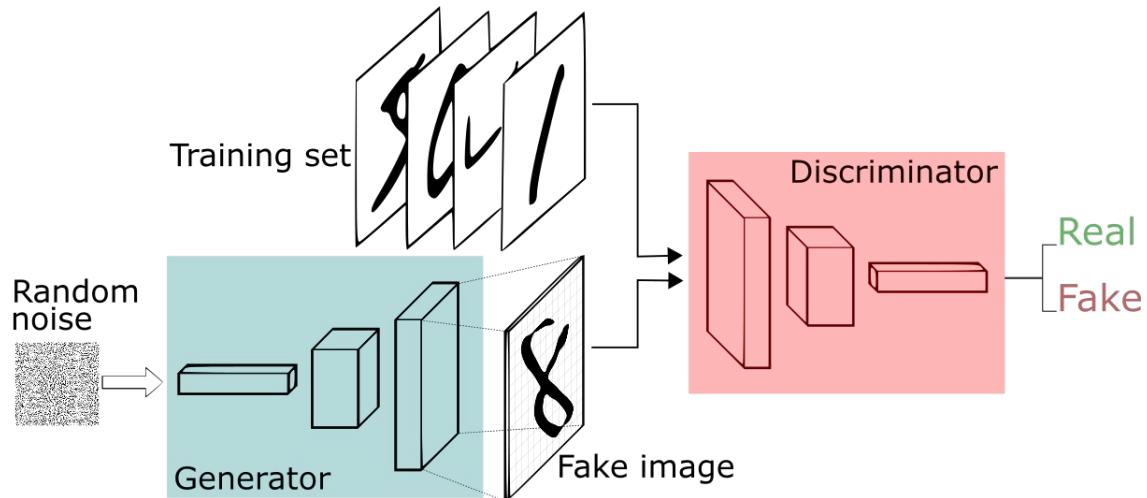
```
class Net(nn.Module):  
    def __init__(self):  
        super(Net, self).__init__()  
        self.conv1 = nn.Conv2d(3, 6, 5)  
        self.pool = nn.MaxPool2d(2, 2)  
        self.conv2 = nn.Conv2d(6, 16, 5)  
        self.fc1 = nn.Linear(16 * 5 * 5, 120)  
        self.fc2 = nn.Linear(120, 84)  
        self.fc3 = nn.Linear(84, 10)  
  
    def forward(self, x):  
        x = self.pool(F.relu(self.conv1(x)))  
        x = self.pool(F.relu(self.conv2(x)))  
        x = x.view(-1, 16 * 5 * 5)  
        x = F.relu(self.fc1(x))  
        x = F.relu(self.fc2(x))  
        x = self.fc3(x)  
        return x
```

## 3. Optimization

```
criterion = nn.CrossEntropyLoss()  
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

# Beyond DNNs for Classification

# Generative Adversarial Networks (GANs)

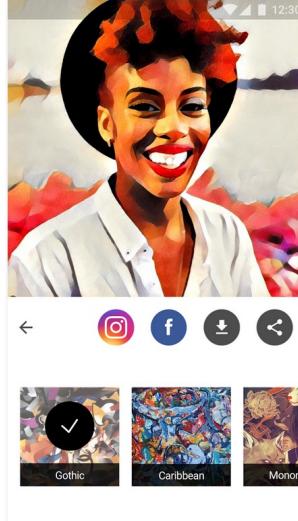
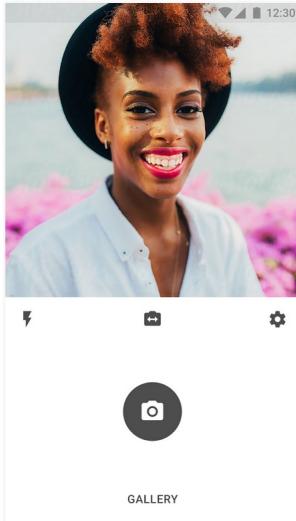


$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))]$$

Ian Goodfellow et al., GANs <https://arxiv.org/pdf/1406.2661.pdf>

# Applications of Generative DNNs

- Gatys et al. Image Style Transfer Using Convolutional Neural Networks, CVPR 2016
  - Prisma (Android): <https://play.google.com/store/apps/details?id=com.neuralprisma>
  - Prisma (iOS): <https://itunes.apple.com/us/prisma-art-filters-photo-effects/id1122649984?mt=8>



# Applications of Generative DNNs

this small bird has a pink breast and crown, and black primaries and secondaries.



this magnificent fellow is almost all black with a red crest, and white cheek patch.



the flower has petals that are bright pinkish purple with white stigma



this white and yellow flower have thin white petals and a round yellow stamen



Reed et al., Generative Adversarial Text to Image Synthesis <https://arxiv.org/pdf/1605.05396.pdf>

## References / Books

- <http://neuralnetworksanddeeplearning.com>
- <http://deeplearning.stanford.edu/tutorial/>
- <http://www.deeplearningbook.org/>
- <http://deeplearning.net/>
- <https://d2l.ai/index.html>