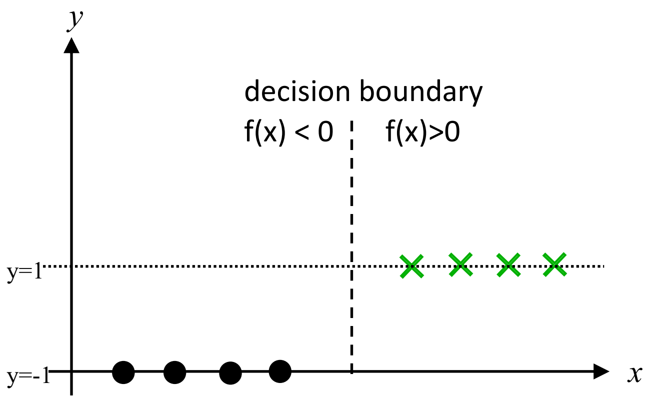
Deep Learning: writing the code

Maura Pintor

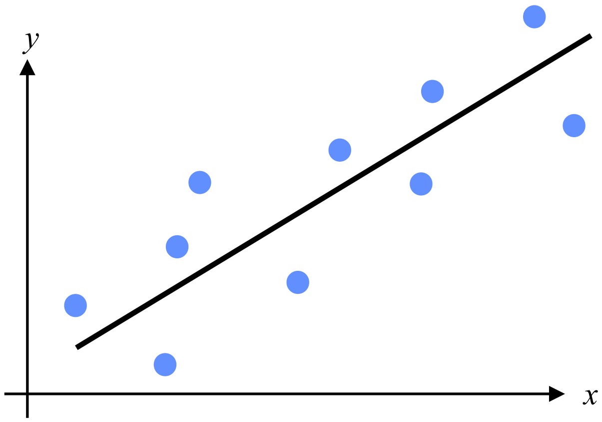
Classification vs Regression (with linear estimators)

Classification vs Regression

Classification (estimates y in a discrete set, e.g., {-1,1})



Regression estimates continuous-valued *y*



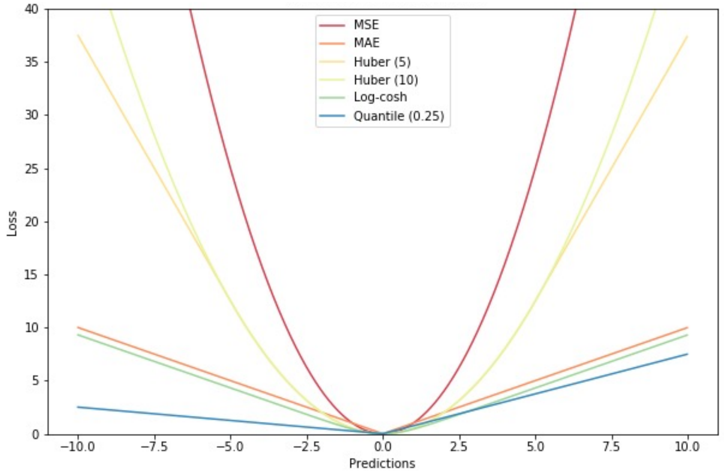
Open in Colab https://colab.research.google.com/github/maurapintor/ai4dev/blob/main/AI4Dev_04_dnns.ipynb

Classification vs Regression

- The loss functions used in the two problems reflect this behavior

- For classification problems, correctly classified points are assigned a loss equal to zero
  - e.g., the hinge loss gives zero penalty to points for which $yf(\boldsymbol{x}) \geq 1$

- For regression problems, the loss is zero only if $f(\boldsymbol{x})$ is exactly equal to y
  - e.g., the mean squared error (MSE) is given as the average of $\left(y - f(\boldsymbol{x})\right)^2$ over all points



SARDIGNA CHIRCAS
SARDEGNA RICERCHE

open:campus

Ridge Regression

- It uses the mean squared error (MSE) as the error function and l2 regularization on the feature weights:

$$L(\boldsymbol{w}) = \frac{1}{2n} ||\mathbf{X}\boldsymbol{w} - \boldsymbol{y}||^2 + \lambda ||\boldsymbol{w}||^2$$

- Minimizing $L(\boldsymbol{w})$ provides the following closed-form solution:

$$\boldsymbol{w} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\boldsymbol{y} \, ,$$

- being **I** the identity matrix, and $\lambda > 0$ a trade-off parameter.

- In this case, adding a small diagonal (ridge) to the (positive semi-definite) matrix $\mathbf{X}^T\mathbf{X}$ makes it more stable for pseudo-inversion (as it increases its minimum eigenvalue)

Ridge Regression

- Ridge regression can be solved in closed form, through matrix pseudo-inversion
  - Too computationally demanding for large feature sets and datasets

- It is also possible to solve it using gradient-descent procedures, including SGD, which is much faster and better suited to large, high-dimensional training sets

Learning as an Optimization Problem

Learning as an Optimization Problem

- We start by considering a simplified setting in which we aim to find the best parameters $\boldsymbol{\theta} = (\boldsymbol{w}, b)$ that minimize the loss function $L(D, \boldsymbol{\theta})$, being $D = (\boldsymbol{x}_i, y_i)_{i=1}^{n}$ the training dataset:

$$\boldsymbol{\theta}^* = \mathrm{argmin}_{\boldsymbol{\theta}}\, L(D, \boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f(\boldsymbol{x}_i; \boldsymbol{\theta}))$$

- The loss function quantifies the error that the classifier, parameterized by $\boldsymbol{\theta}$, is making on its predictions on the training data D
  - This is also known as the principle of **Empirical Risk Minimization (ERM)**

- How do we select the loss function $L(D, \boldsymbol{\theta})$ and solve the above problem?

SARDIGNA CHIRCAS
SARDEGNA RICERCHE

open:campus

Loss Minimization with Gradient Descent

Gradient-based Optimization

- Optimizing *smooth* functions is much easier and efficient, as we can exploit **gradients**
  - This is not possible for the 0-1 loss (it is flat almost everywhere with gradients equal to zero)

- The key idea of gradient-based optimization is to start from a random point in the parameter space (*random initialization*) and then iteratively update the parameters along the gradient direction. The gradient is the derivative of the loss function w.r.t. the classifier parameters:

$$L(D, \theta) = \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f(\boldsymbol{x}_i; \theta)), \quad \nabla_\theta L = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \ell(y_i, f(\boldsymbol{x}_i; \theta))$$

- It is the direction in the parameter space along which the objective maximally increases
  - Following the negative gradient will thus minimize our training loss!

- **Stochastic Gradient Descent (SGD)** uses a random subset of training samples in each iteration

Gradient Descent (a.k.a. Steepest Descent)

- The simplest gradient-based optimizer is the **steepest-descent** method

1. initialize θ, η, K, ε
2. for k in $\{0, 1, \ldots, K-1\}$:
3. $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \nabla L(\boldsymbol{\theta}_k)$
4.   if $|L(\boldsymbol{\theta}_k) - L(\boldsymbol{\theta}_{k+1})| < \varepsilon$:
5.   break

- The parameters $\boldsymbol{\theta}$ are updated at each iteration, until
  - a maximum of K iterations are reached, or the convergence/stop condition is met (lines 4-5 above)

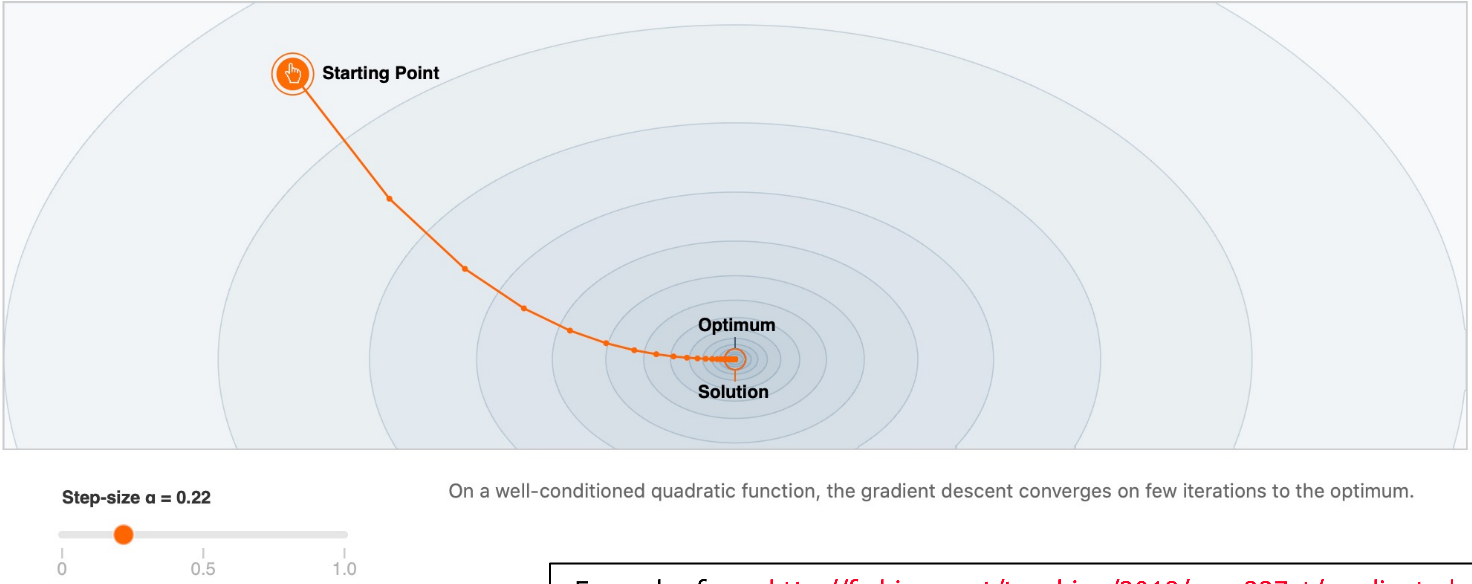
- The stop condition checks that the last update has not significantly modified the objective function (i.e., the training loss is almost constant, as ε is a small number)

- The learning rate (or gradient step size) η affects convergence. If it is too small, convergence is too slow; if it is too large, the algorithm may not even converge at all
  - Usually η is reduced across iterations to ensure convergence

Gradient Descent: Step Size and Convergence

Example: Steepest Descent on Quadratic Objective

- Well-conditioned quadratic objective, small step size

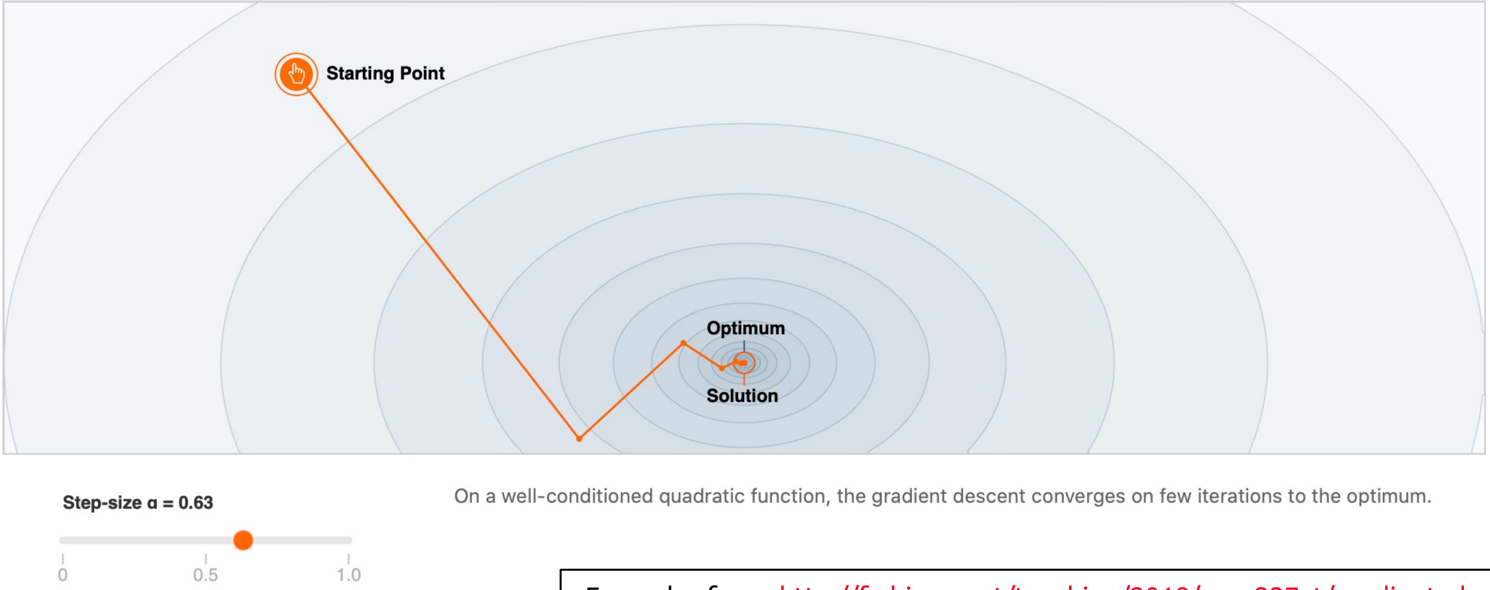


On a well-conditioned quadratic function, the gradient descent converges on few iterations to the optimum.

Examples from: http://fa.bianp.net/teaching/2018/eecs227at/gradient_descent.html

Example: Steepest Descent on Quadratic Objective

- Well-conditioned quadratic objective, large step size

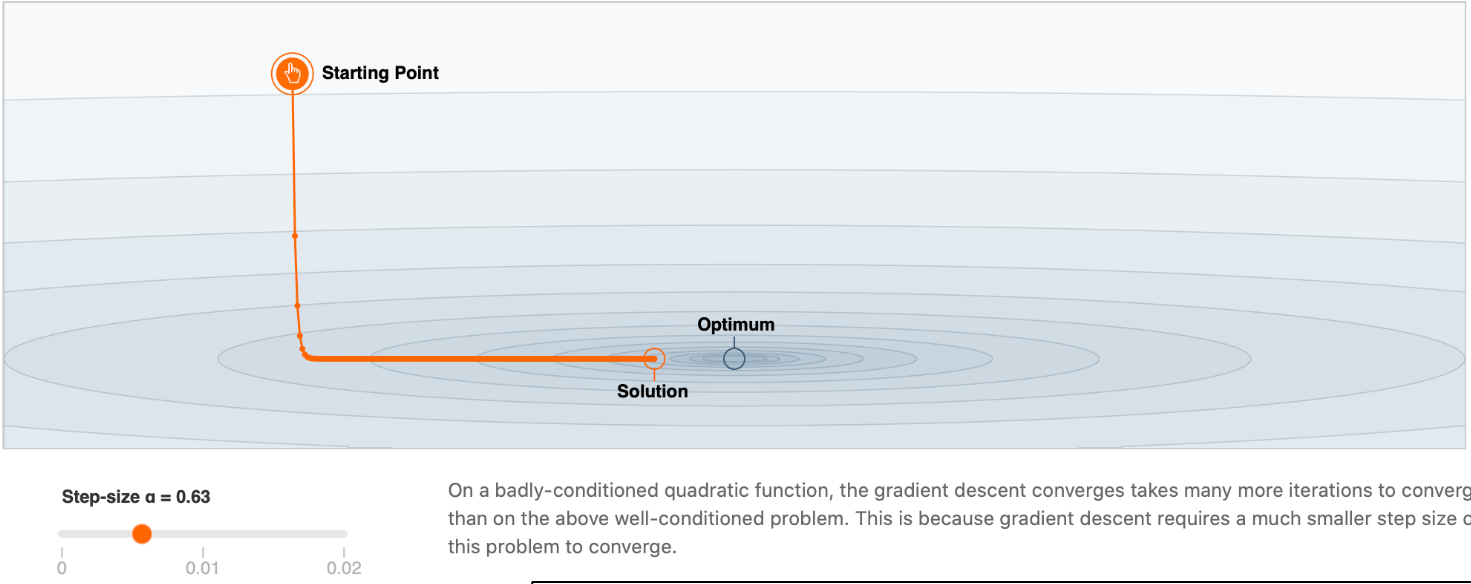


On a well-conditioned quadratic function, the gradient descent converges on few iterations to the optimum.

Examples from: http://fa.bianp.net/teaching/2018/eecs227at/gradient_descent.html

Example: Steepest Descent on Quadratic Objective

- Badly-conditioned quadratic objective, slow convergence



On a badly-conditioned quadratic function, the gradient descent converges takes many more iterations to converge than on the above well-conditioned problem. This is because gradient descent requires a much smaller step size on this problem to converge.

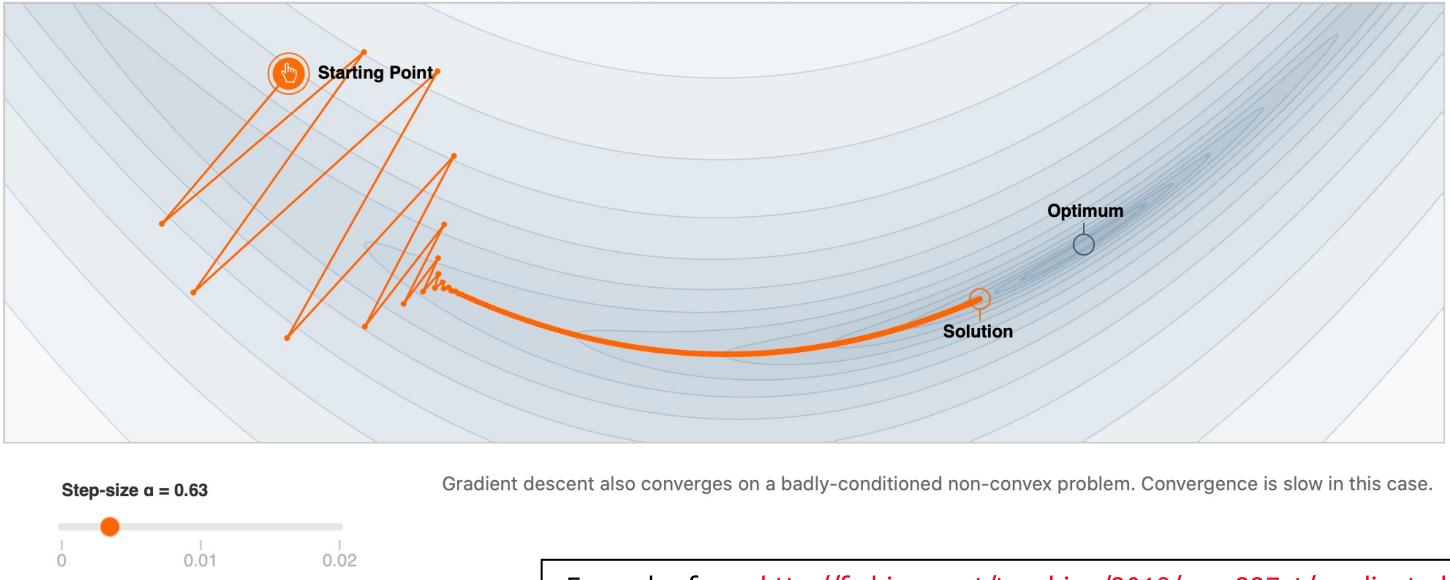
Examples from: http://fa.bianp.net/teaching/2018/eecs227at/gradient_descent.html

SARDIGNA CHIRCAS SARDEGNA RICERCHE

open:campus

Example: Steepest Descent on Non-convex Objective

- Badly-conditioned non-convex objective, slow convergence and initial instability



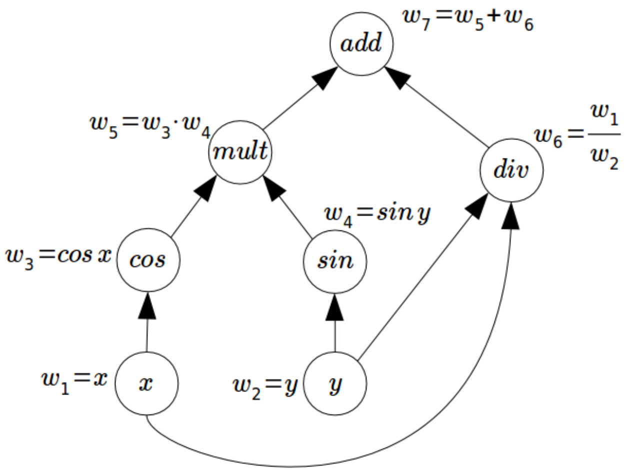
Gradient descent also converges on a badly-conditioned non-convex problem. Convergence is slow in this case.

Examples from: http://fa.bianp.net/teaching/2018/eecs227at/gradient_descent.html

Automatic gradients

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

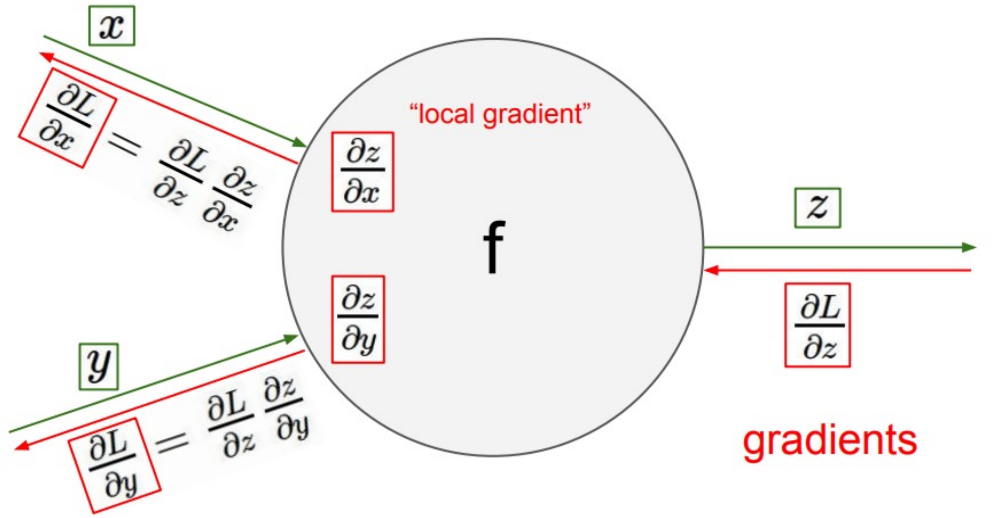


$f(x,y)=\cos x \sin y + x/y$

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

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



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

Multiclass Classification with DNNs

Deep Learning with PyTorch

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
```

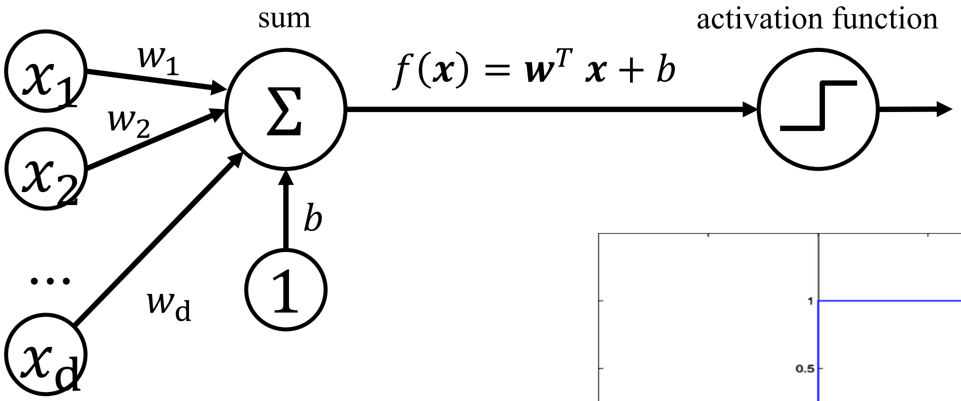
3. Optimization

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

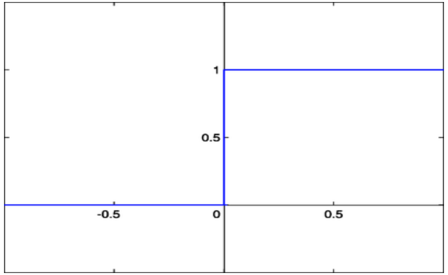
Open in Colab https://colab.research.google.com/github/maurapintor/ai4dev/blob/main/AI4Dev_05_dnns_mnist.ipynb

The Perceptron

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



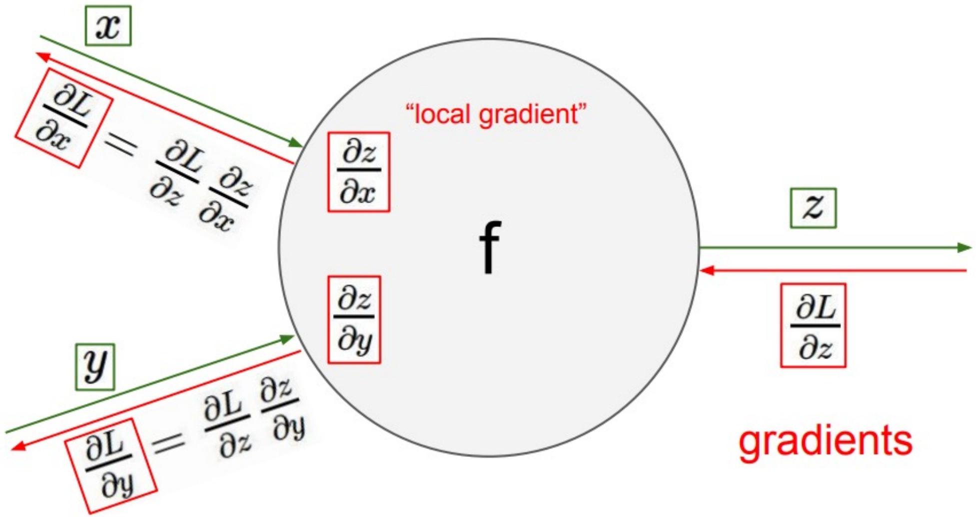
$$h(\boldsymbol{x}) = \begin{cases} 0, & \text{if } \boldsymbol{w}^T\ \boldsymbol{x} + b < 0 \\ 1, & \text{if } \boldsymbol{w}^T\ \boldsymbol{x} + b \geq 0 \end{cases}$$



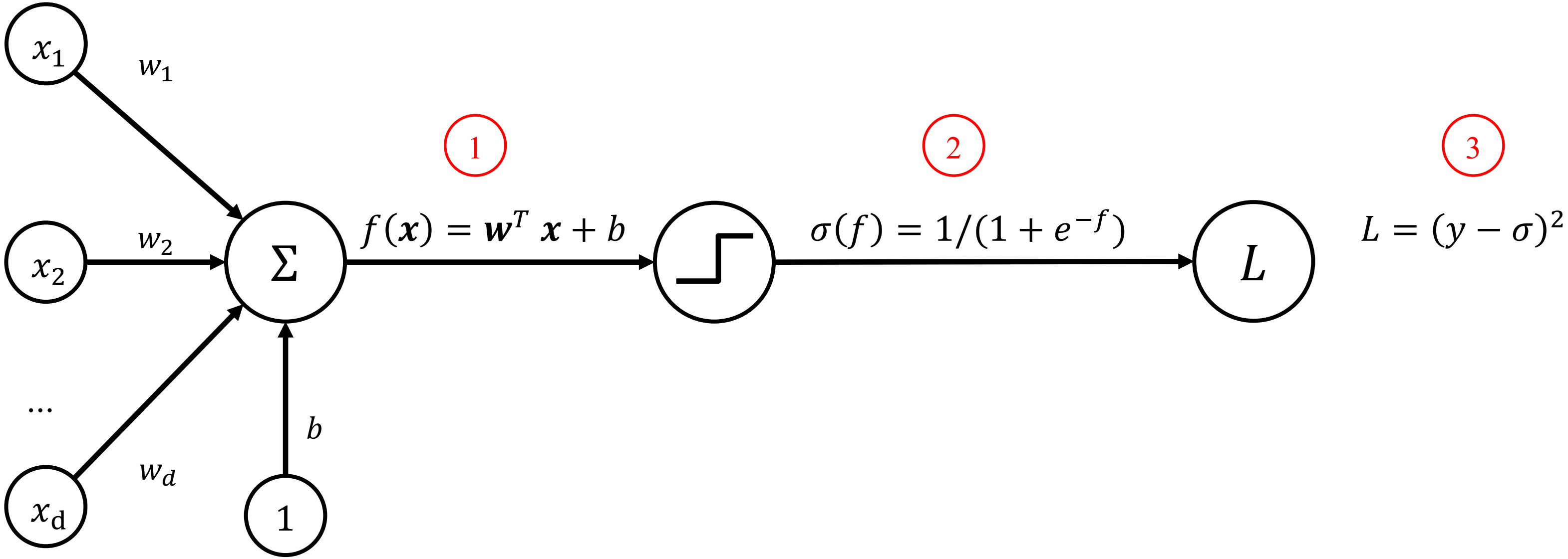
step / Heavyside activation function

Back-Propagation Learning: Intuition

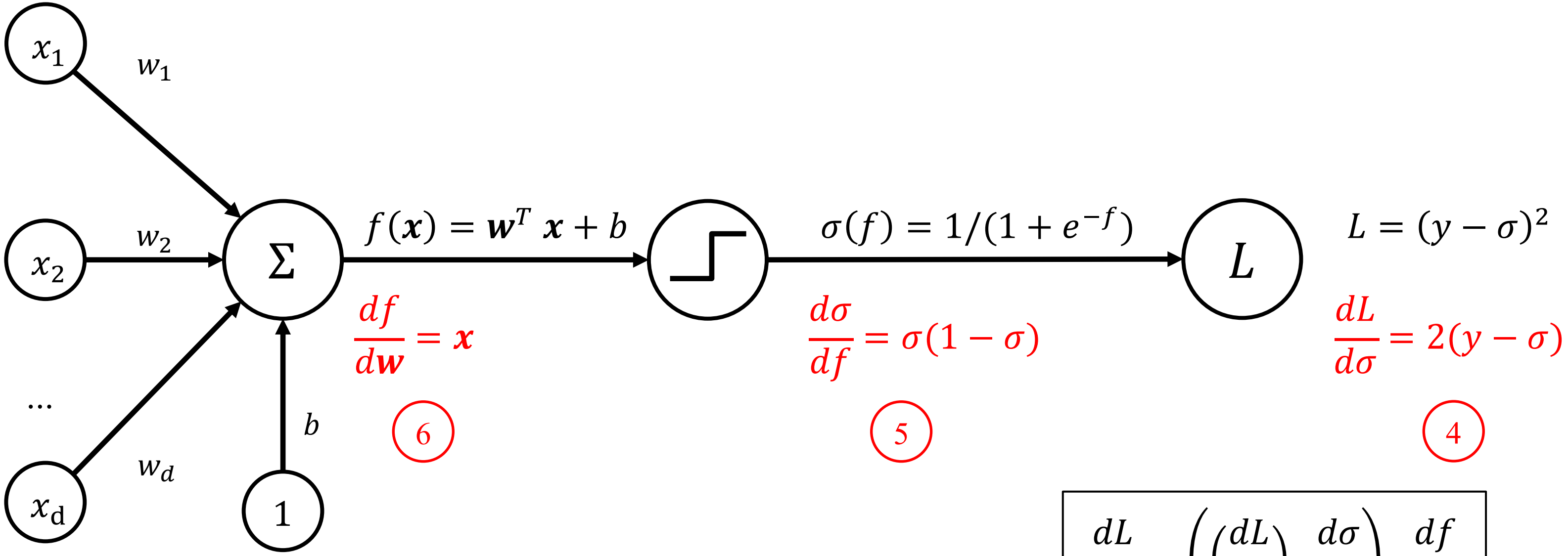
- **Forward step:** compute output function (e.g., loss) given input (e.g., one sample)
- **Backward step:** compute gradient in reverse-mode automatic differentiation, via the chain rule; e.g., dL/dx = dL/dz * dz/dx



Forward Pass



Backward Pass



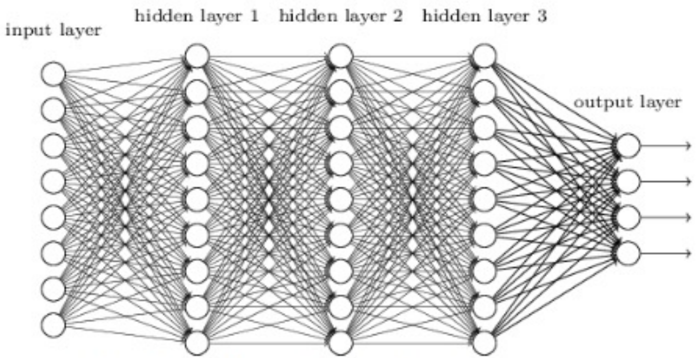
$$\boxed{\frac{dL}{d\boldsymbol{w}} = \left(\left(\frac{dL}{d\sigma}\right) \cdot \frac{d\sigma}{df}\right) \cdot \frac{df}{d\boldsymbol{w}}}$$

Back-Propagation Learning

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

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

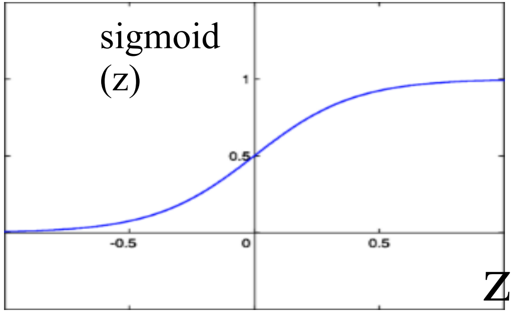
- *Ad-hoc* modifications to activation functions and training procedures have been introduced to avoid drawbacks of standard ones (e.g., very slow convergence)

ReLU Activations

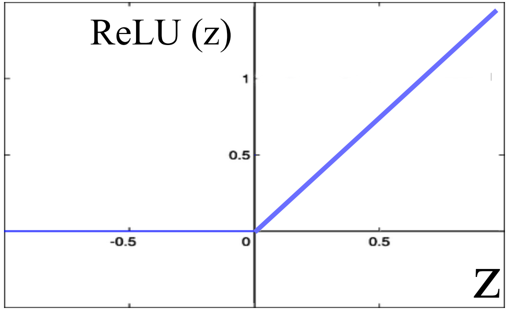
- **Problem:** sigmoid takes on values in (0,1). Its gradient z (1- z) is closer to 0 than z.
  - The net effect is that, propagating the gradient back to the initial layers, it tends to become 0 (**vanishing gradient problem**).
  - From a practical perspective, this slows down the training procedure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

- Rectified Linear Unit (**ReLU**) activations can overcome this issue

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



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



Multiclass Linear Classifiers

- Linear functions can be also naturally extended to multiclass problems

$$f(\boldsymbol{x}) = \mathbf{W}^{\mathrm{T}}\boldsymbol{x} + \boldsymbol{b}$$

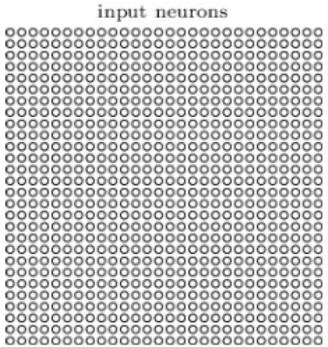
$$\begin{bmatrix} 0.2 & -0.5 & 0.1 & 2.0 \\ 1.5 & 1.3 & 2.1 & 0.0 \\ 0 & 0.25 & 0.2 & -0.3 \end{bmatrix} \begin{bmatrix} 56 \\ 231 \\ 24 \\ 2 \end{bmatrix} + \begin{bmatrix} 1.1 \\ 3.2 \\ -1.2 \end{bmatrix} = \begin{array}{l} \begin{bmatrix} -96.8 \\ 437.9 \\ 61.95 \end{bmatrix} \end{array} \begin{array}{l} \text{Cat score} \\ \text{Dog score} \\ \text{Ship score} \end{array}$$

Convolutional Neural Networks (ConvNets)

Convolutional Neural Networks

- DNNs are widely employed for computer vision tasks, for which specialized architectures have been proposed, named ***convolutional neural networks*** (CNNs)

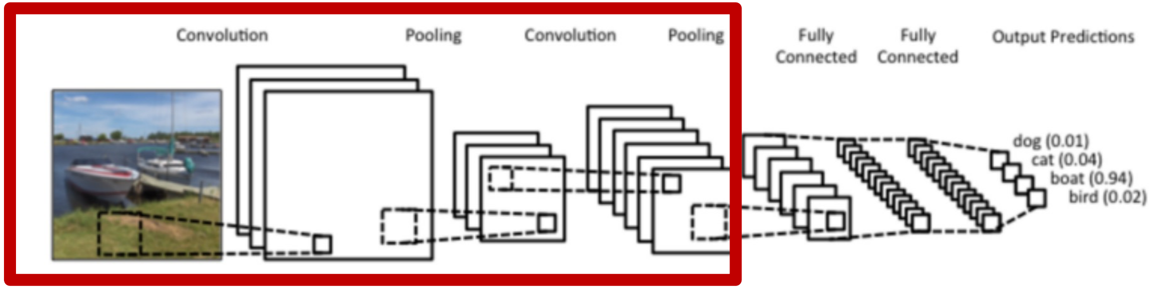
- In this case the CNN input is a raw image. Accordingly, the CNN architecture is not fully connected, but it exploits the **spatial adjacency** between pixels. Input units are arranged into an array



taken from http://neuralnetworksanddeeplearning.com

Open in Colab https://colab.research.google.com/github/maurapintor/ai4dev/blob/main/AI4Dev_06_convnets.ipynb

Convolutional Neural Networks



Convolution & Pooling Layers

MLP network

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

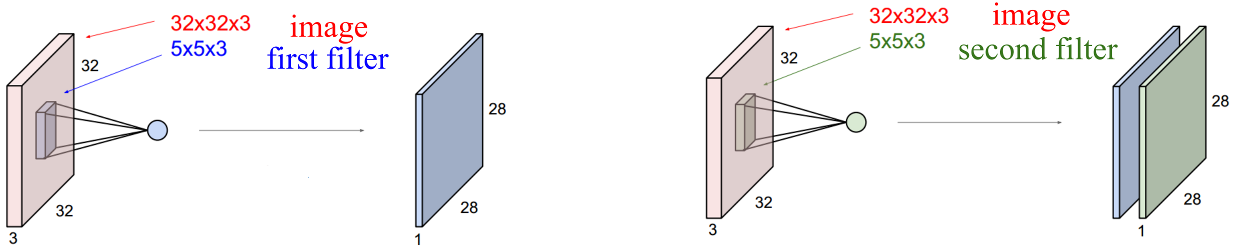
Filtering (Convolutional Units)

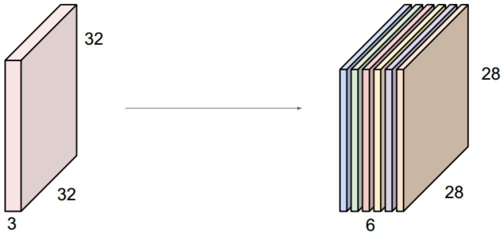
-1	-1	-1
2	2	2
-1	-1	-1

I **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/

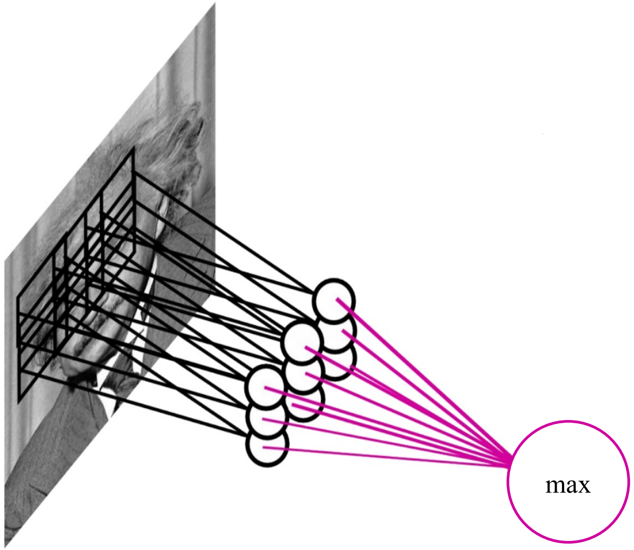


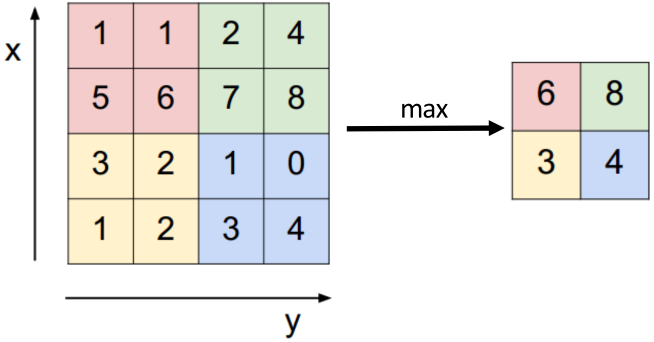


- Conv. filter/kernel: 5x5x3
- Stride (how much we slide the filter)
- Padding (zeros around the border)
- Depth: number of filters

Pooling

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

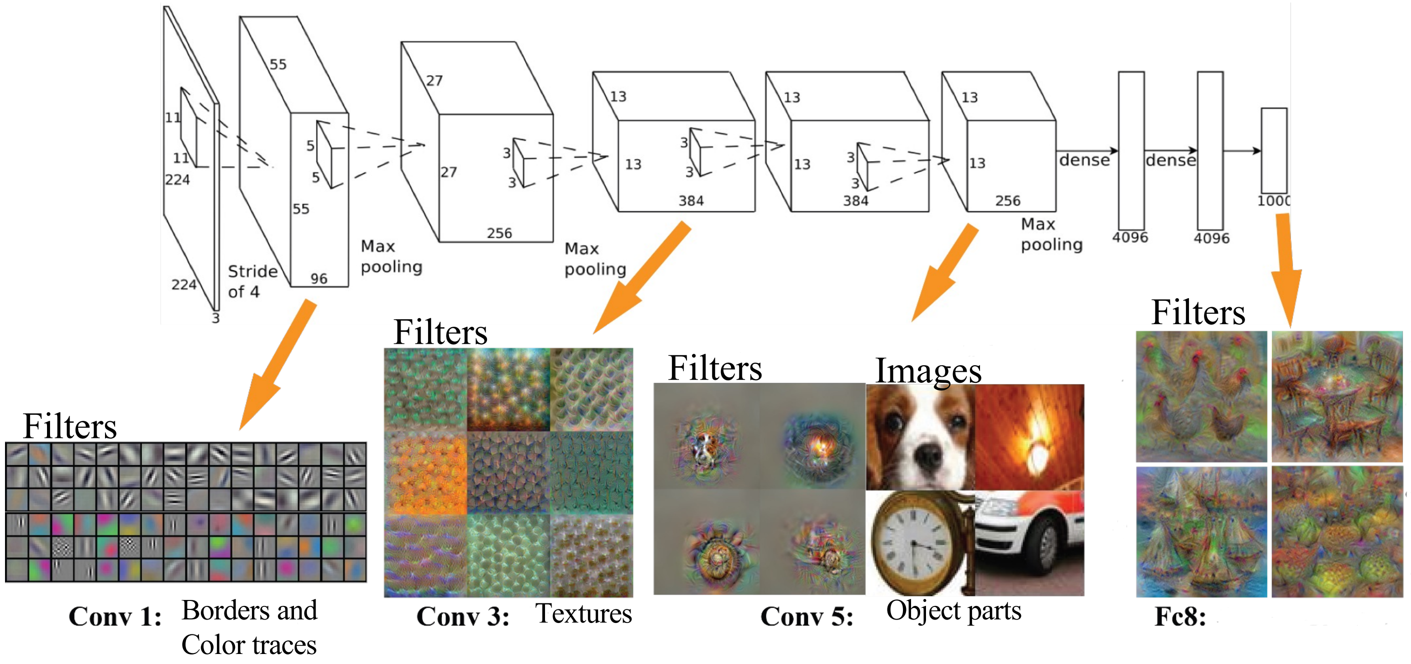




Slide credits: M. A. Ranzatto e Fei Fei Li

Convolutional Neural Networks

- The deep network gradually learns more complex and abstract notions



Avoiding Overfitting

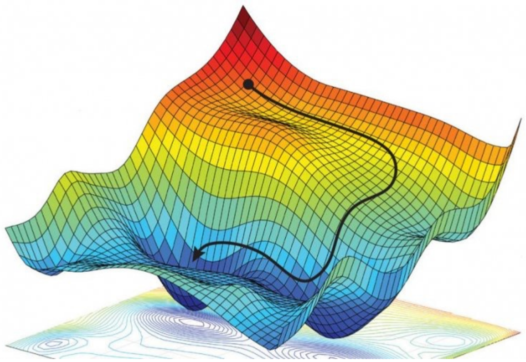
Back-Propagation Learning: Issues

1. The error function usually has many **local minima**
   - gradient descent does not guarantee convergence to the smallest error on training examples

   A **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**

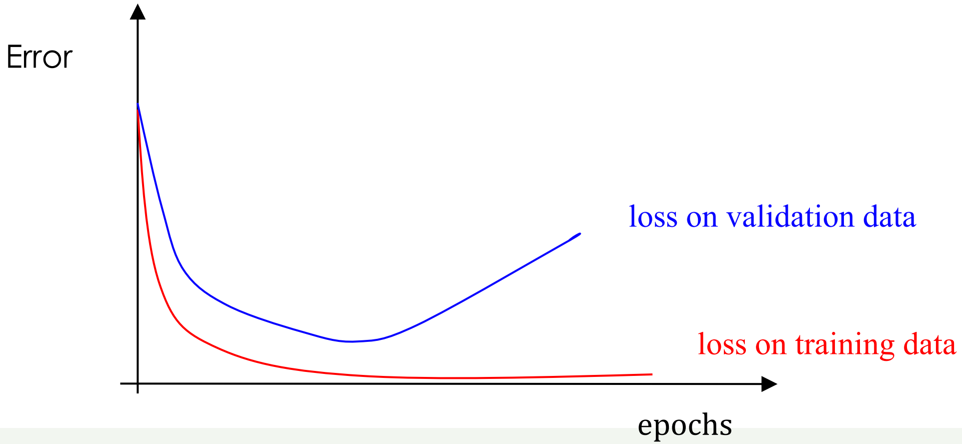
   Early-stopping can be used to tackle this issue



Overfitting

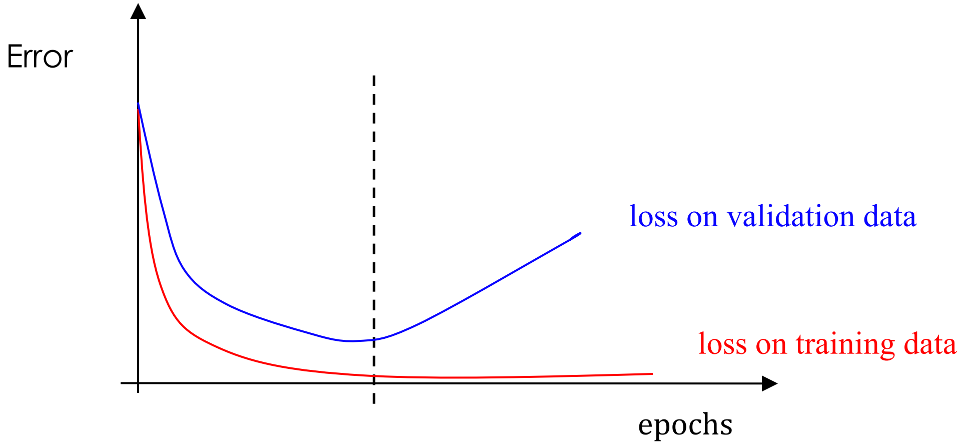
- Overfitting can be detected by running the back-propagation algorithm on a training set made up of a **subset** of the available examples, and by evaluating the error function also on (a subset of) the remaining examples, called **validation set**

- 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

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

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

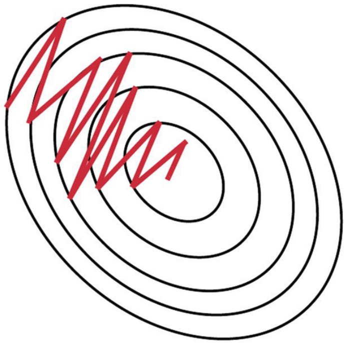
Momentum

- The gradient $\boldsymbol{g}_k$ is averaged across iterations, with weight β
  - $\boldsymbol{v}_k = \beta \boldsymbol{v}_{k-1} + \boldsymbol{g}_k$
  - $\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} - \eta_k \boldsymbol{v}_k$

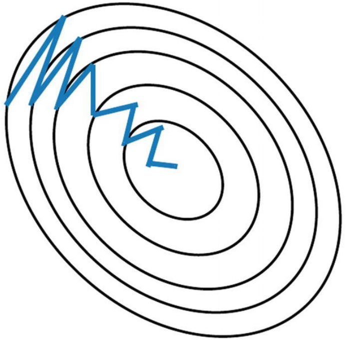
- With β=0, it is equivalent to the steepest descent method

- Momentum works as a smoothing operator on the objective function, facilitating convergence

More at: https://d2l.ai/chapter_optimization/momentum.html



Stochastic Gradient Descent **withhout** Momentum



Stochastic Gradient Descent **with** Momentum

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