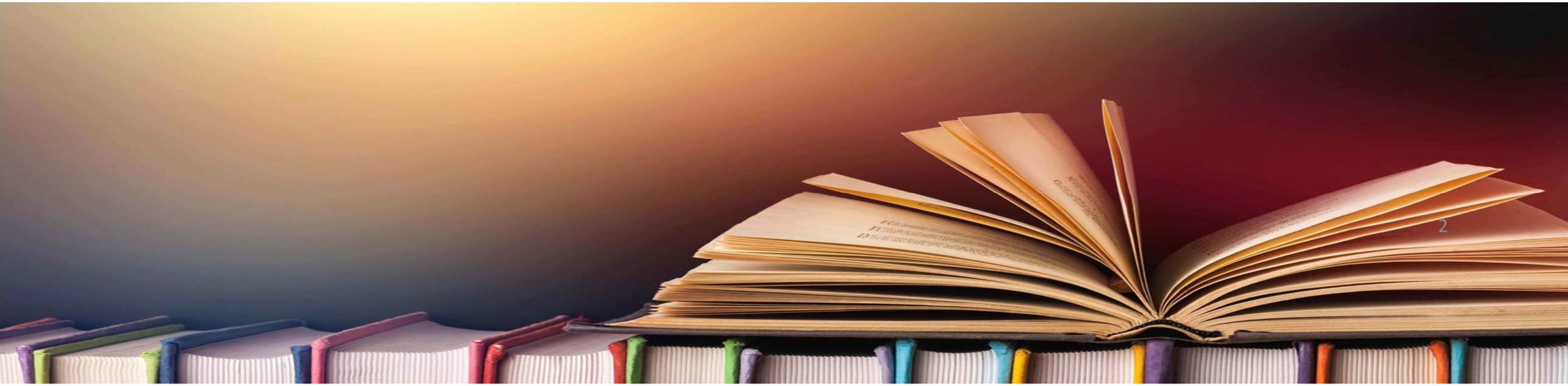
CAS in Applied Data Science Muerren 2025

Géraldine Schaller-Conti

Bibliography

- Deep Learning book (Goodfellow, Bengio, Courville)
- Machine Learning @ Stanford (Prof Andrew Ng)
- Hands-On Machine Learning with Scikit-Learn & Tensorflow (Aurélien Géron)



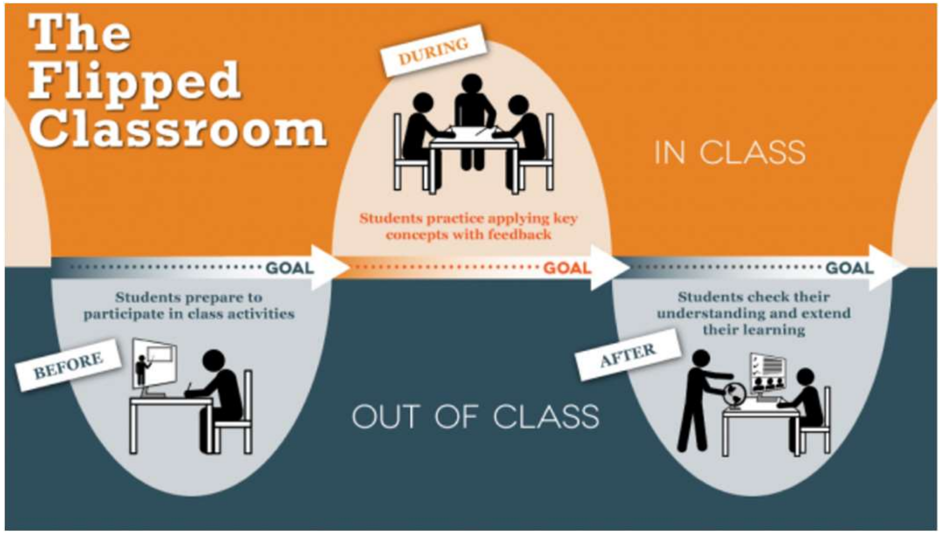
Teaching method

Inverted classroom based

- Introduction lectures
- Real content you learn yourself with the notebooks. *Either to put in practice your knowledge or to learn ahead of another lecture*

Why

- Supposed to be better
- More fun
- Learning by doing



To give back sense to being present (Marcel Lebrun)

Tutorial I : Introduction to torch

Link → Copy to drive

Introduction

- Goal : introduction to pytorch

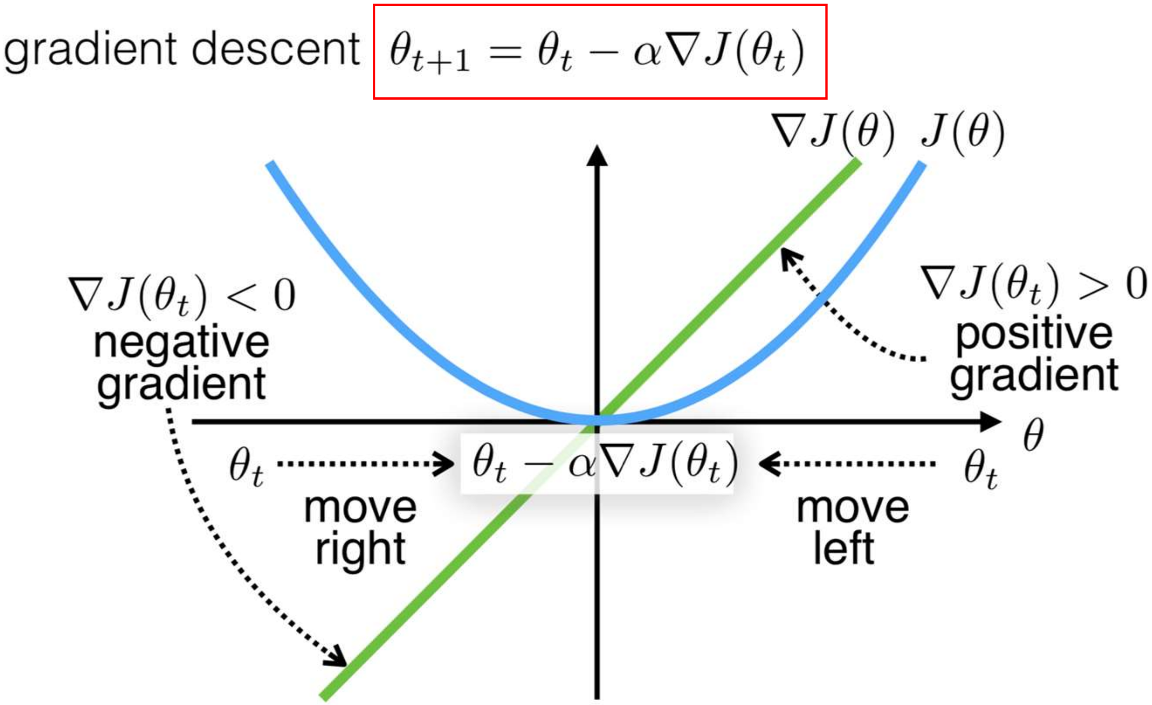
- Program : inverted classroom style
  - Theory
  - Overview to get the big picture of the Notebook
  - Work alone or in groups

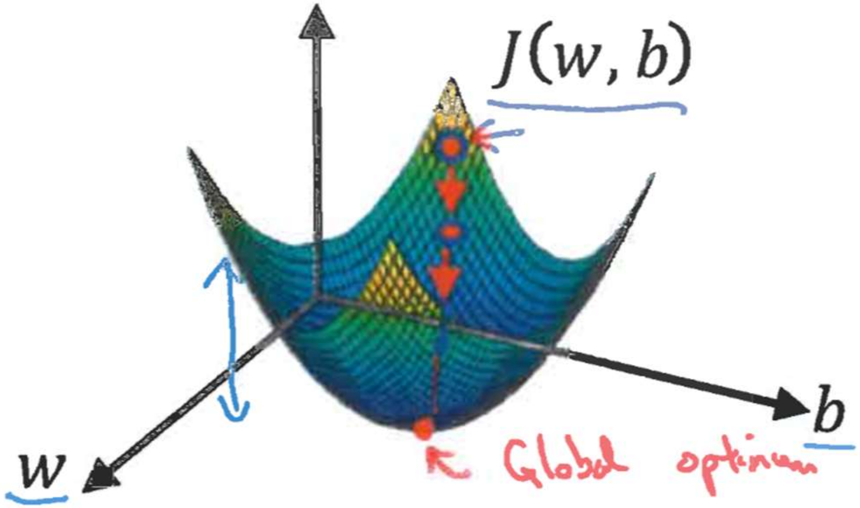
- Technical : Google Colab, Pytorch

Theory

Gradient Descent

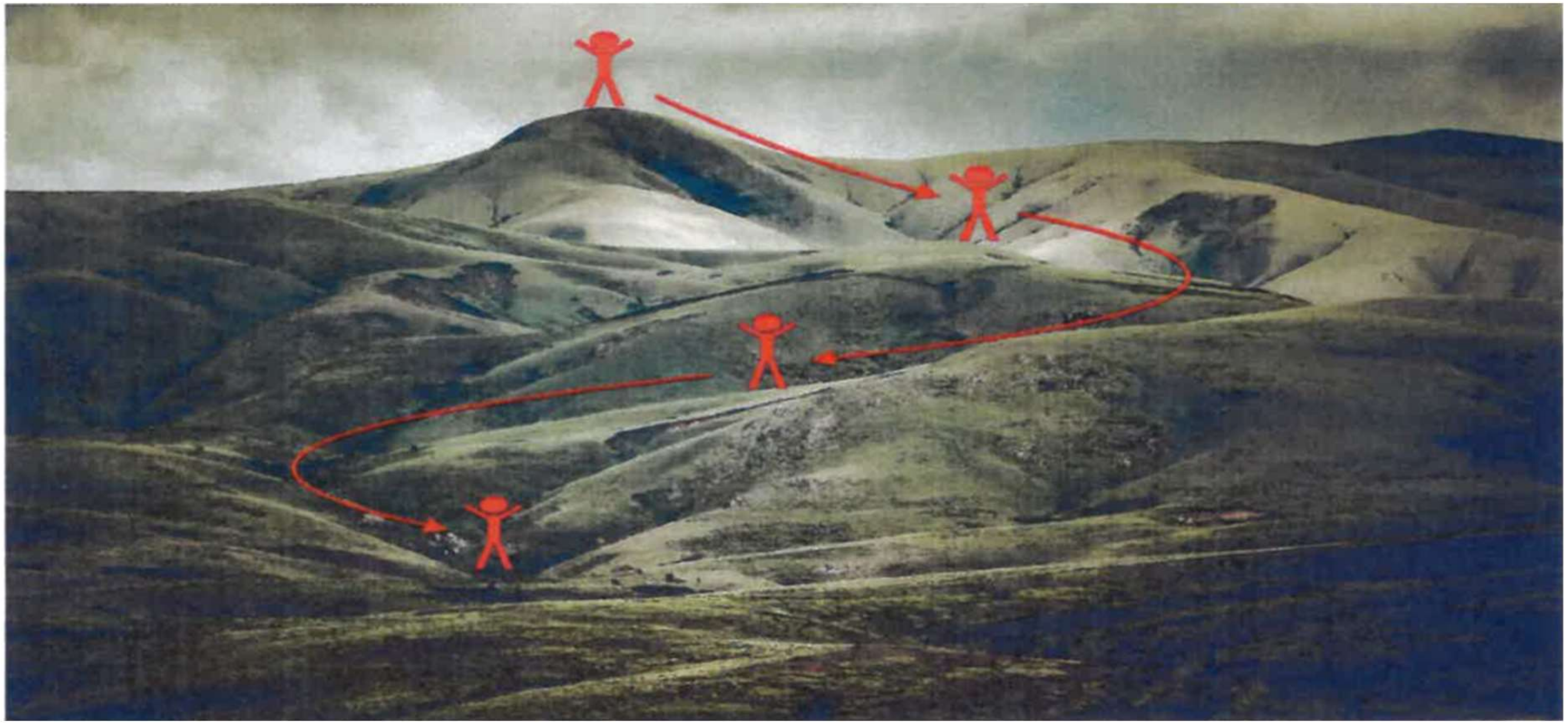
- Iterative method to find the parameters $\theta = (w, b)$ that minimize $J(\theta)$





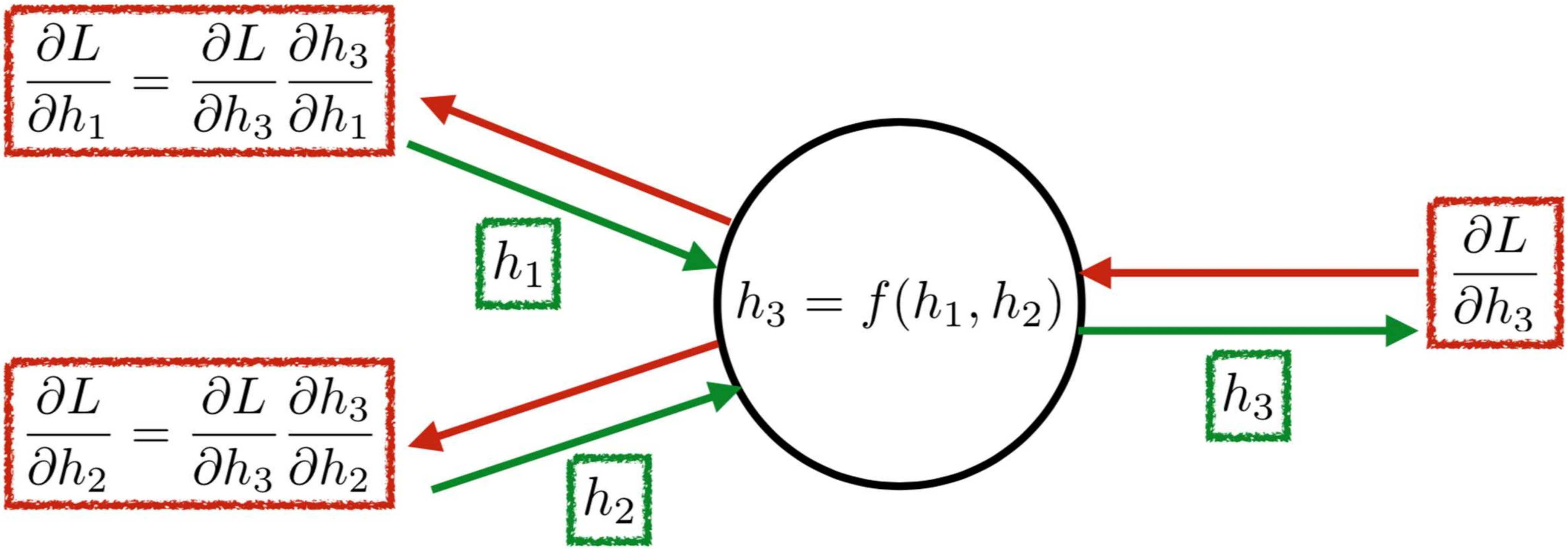
$$\nabla J(w) = \frac{dJ(w,b)}{dw} \qquad \nabla J(b) = \frac{dJ(w,b)}{db}$$

Gradient Descent Illustration



Backpropagation

- Efficient implementation of the chain-rule to compute derivatives with respect to network weights



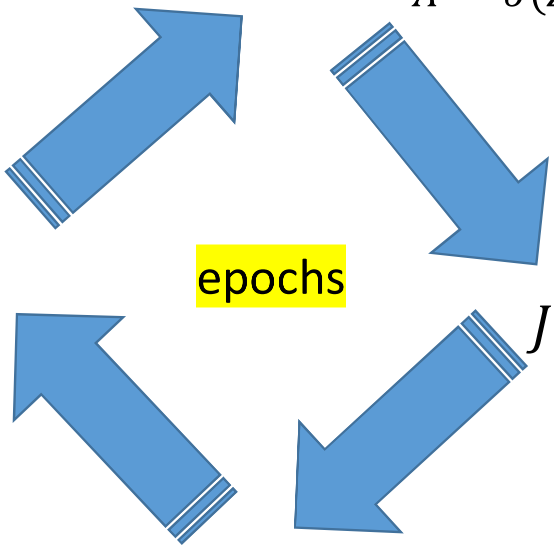
Training

- *Iterative* process

Forward propagation

$$Z = w^T x + b$$

$$A = \sigma(Z)$$



epochs

Cost function

$$J(w, b) = J(\theta)$$

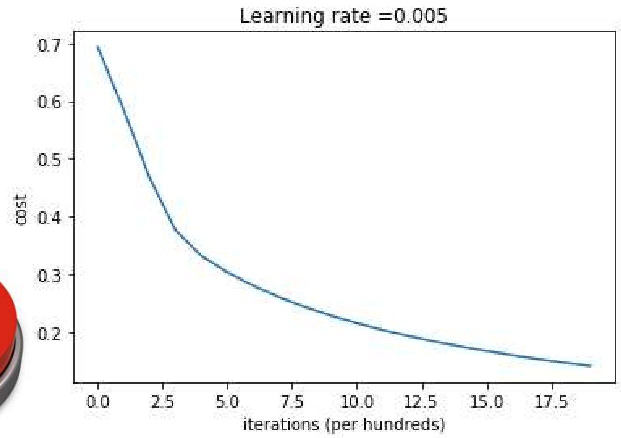
Parameter update (gradient descent)

learning rate α

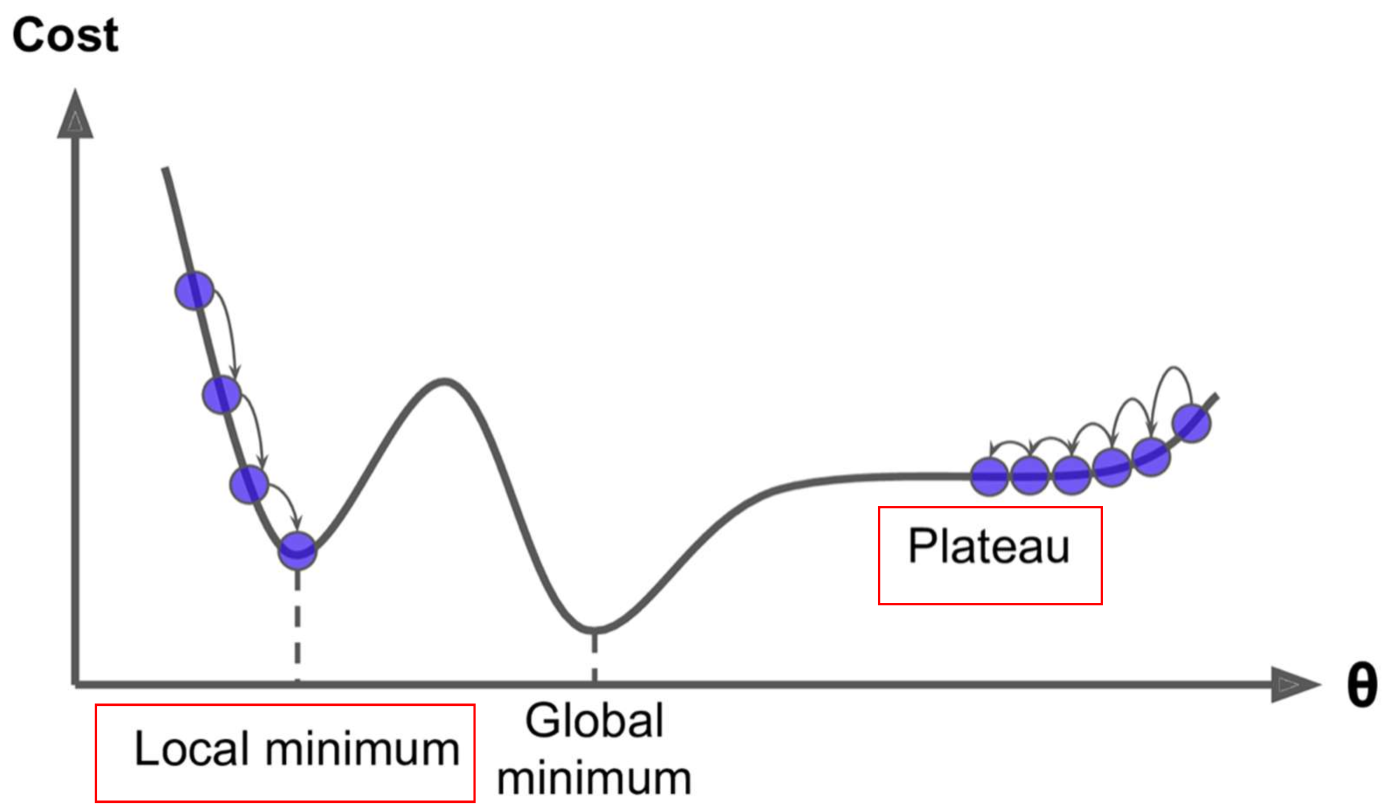
$$\boxed{\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t)}$$

Backward propagation
(dJ/dw, dJ/db)

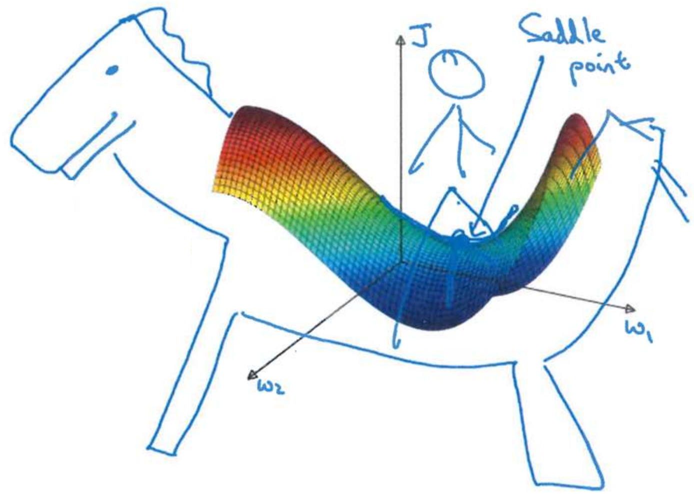
Learning curve



Optimization pitfalls



Saddle point



Tools

data structure & analysis

main package for scientific computing in Python

interactive coding environments embedded in a webpage

Python-based ecosystem of open-source software for mathematics, science, and engineering.

famous library to plot graphs in Python

provides simple and efficient tools for data mining and data analysis

h5py : common package to interact with a dataset that is stored on an H5 file

Overview of the notebook

Tutorial I (1)

1) Load necessary libraries (common libraries) and data

2) Create model (class SimpleModel)
   - y = x * (x+2)

3) Run the model
   - Test it, look at the output type

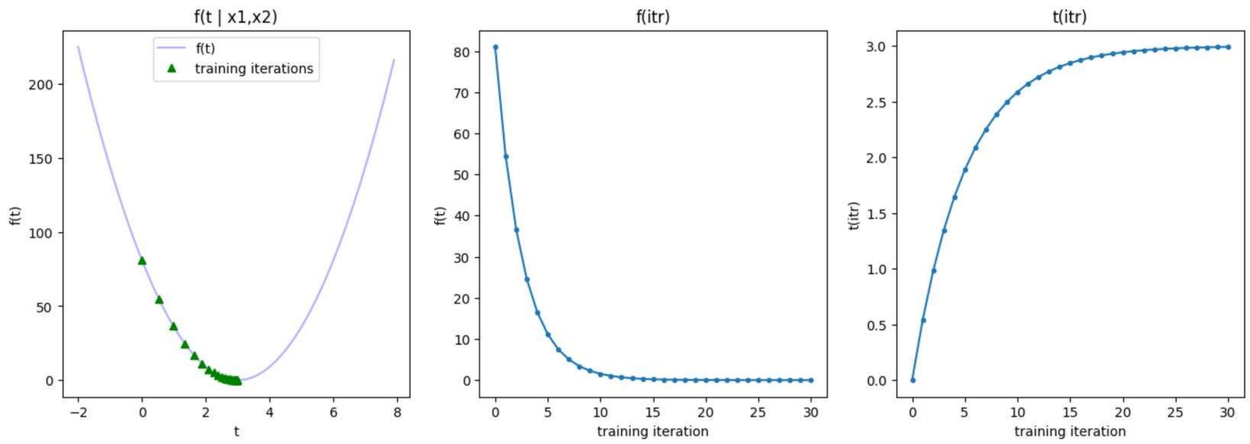
4) Tensor operations
   - New model : $sum_i(x_i+2)$ (class SimpleModel2)
   - Same for several 1D arrays at one (class SimpleMOdel3) : add axis=1

5) Exercise 1 : calculate mean of array's elements

Tutorial I (1)

6) Optimization :

- New parabolic function to optimize (class FLayer) – trainable parameter is t initialized to 0
- *Gradient descent* applied, SGD optimizer defined
- Plot f(t) results



7) Exercise 2 : change the parabolic function, the alpha learning rate,…

Exercise 1

Copier le code

```
# Define the MeanModel class
class MeanModel(nn.Module):
    def __init__(self):
        super(MeanModel, self).__init__()

    def forward(self, x):
        return torch.mean(x)


# Define data
arr = torch.tensor([[1, 2, 3, 4, 5], [2, 3, 4, 5.1, 6], [25, 65, 12, 12,


# Instantiate the model
model = MeanModel()


# Run the model
result = model(arr)
```

Tutorial II : Optimization and NN introduction

[Link](#) → Copy to drive

Introduction

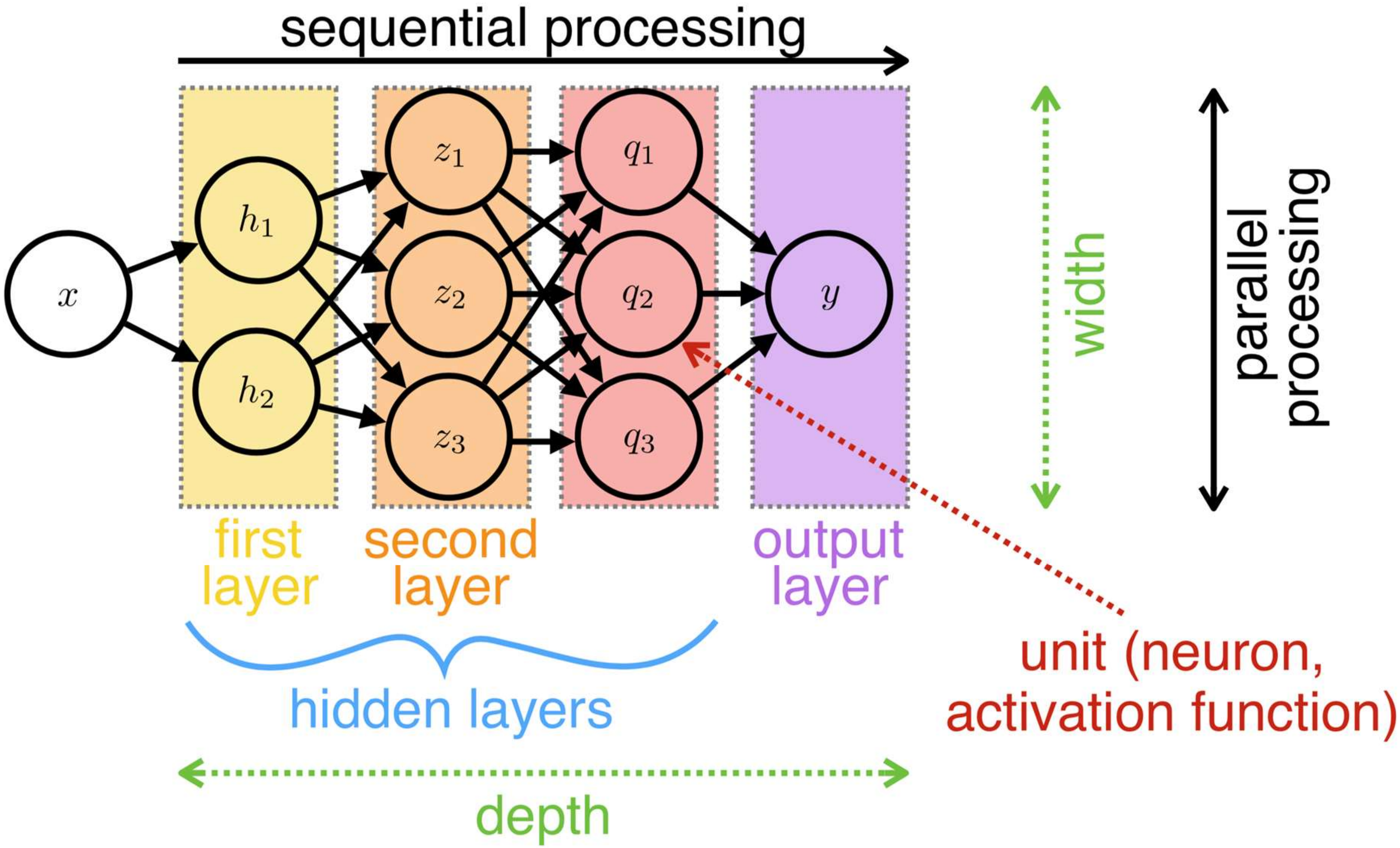
- Goal : see how to do optimization in torch and NN introduction

- Program : inverted classroom style
  - Theory
  - Overview to get the big picture of the Notebook
  - Work alone or in groups

- Technical : Google Colab, Pytorch

Theory

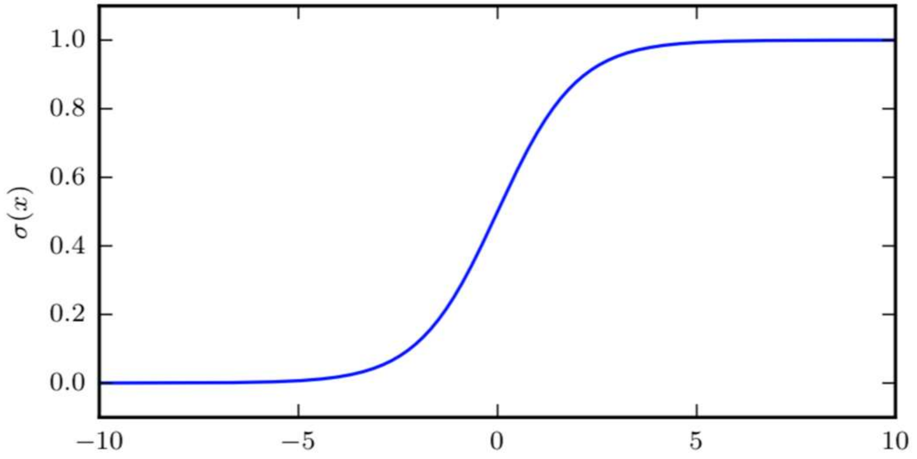
Network model



Sigmoid and softmax

Sigmoid (*two-class* classifier) :

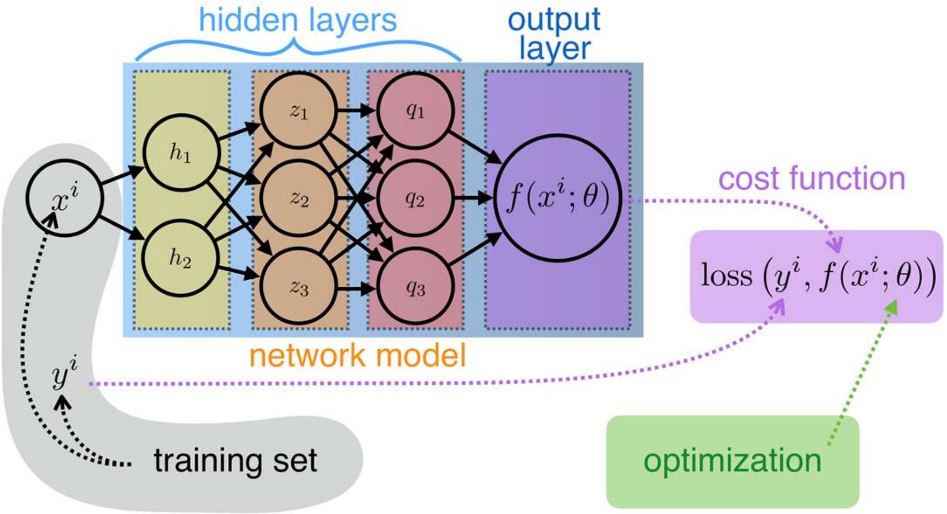
$$\boxed{\sigma(x) = \frac{1}{1+e^{-x}}}$$



Softmax (*multi-class* classifier) :

$$\boxed{\mathrm{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}}$$

Optimization



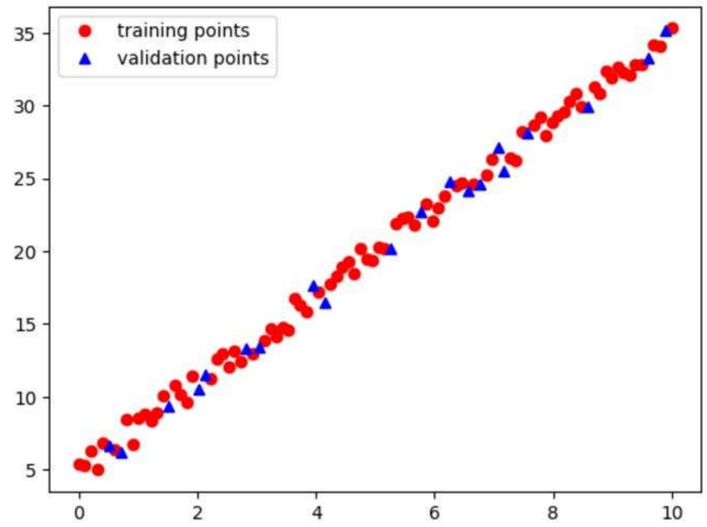
- Given a task we define
  - Training data $\{x^i, y^i\}_{i=1,\dots,m}$
  - Network $f(x;\theta)$
  - Cost function $J(\theta) = \sum_{i=1}^{m} \mathrm{loss}\left(y^i, f(x^i;\theta)\right)$
  - Parameter initialization (weights, biases)
    - *random weights*, *biases initialized to small values (0.1)*
- Next, we *optimize the network parameters θ* (training)
- In addition, we have to set values for hyperparameters

Overview of the notebook

Tutorial II (1)

1) Load necessary libraries (common libraries)

2) Linear regression
   - Generate data points (80 training, 20 test)
   - Linear function (class Linear)
   - Loss function (def loss_f)
   - Train the model
   - Evaluate the model



3) Exercise 1 : play with the parameters of the linear regression and the batch size

Tutorial II (2)

4) Explanation of the training loop with pseudcode

5) Building blocks of a Neural network
   - Model (class Dense)
   - Forward pass

5) Build a NN
   - Multilayer NN (dense1, dense2)
   - Overall model (class MyModel)

Tutorial III : Fully connected NNs

Link → Copy to drive

Introduction

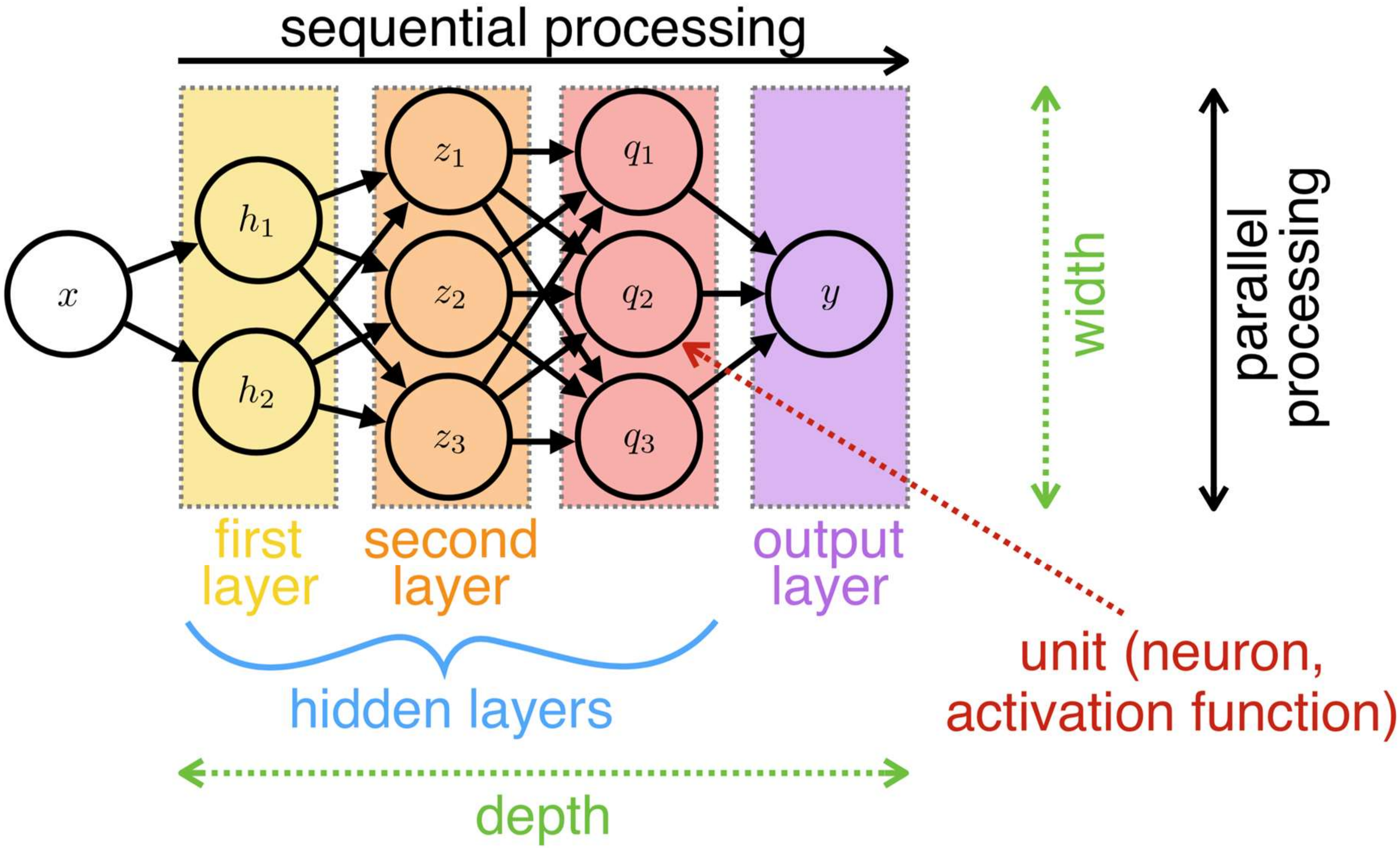
- Goal : handwritten digit recognition with a fully connected NN
- Program : inverted classroom style
  - Theory
  - Overview to get the big picture of the Notebook
  - Work alone or in groups
- Technical : Google Colab, Pytorch

Theory

MNIST Dataset

- MNIST database with hand-written digits
- 60000 training images and 10000 testing images
- Created in 1994
- 128x128 binary images

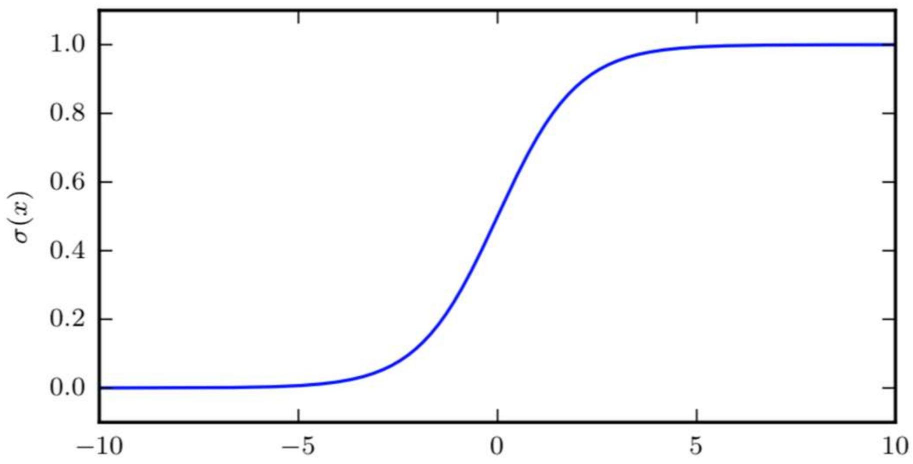
Network model



Sigmoid and softmax

Sigmoid (*two-class* classifier) :

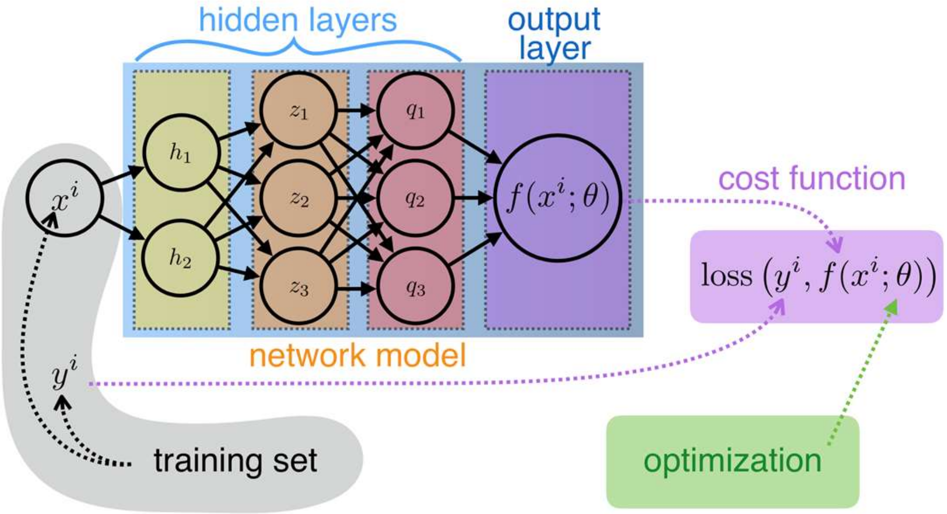
$$\boxed{\sigma(x) = \frac{1}{1 + e^{-x}}}$$



Softmax (*multi-class* classifier) :

$$\boxed{\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}}$$

Optimization



- Given a task we define
  - Training data $\{x^i, y^i\}_{i=1,\ldots,m}$
  - Network $f(x;\theta)$
  - Cost function $J(\theta) = \sum_{i=1}^{m} \text{loss}\left(y^i, f(x^i;\theta)\right)$
  - Parameter initialization (weights, biases)
    - *random weights*, *biases initialized to small values (0.1)*
- Next, we *optimize the network parameters θ* (training)
- In addition, we have to set values for hyperparameters

Overview of the notebook

Tutorial III (1)

1) Load necessary libraries (common libraries) and data

2) Training loop (similar to Tutorial II)
   - Explanation

3) Building blocks of a NN (similar to Tutorial II)

4) Structure of a NN
   - Definition of a model and the forward pass (class MyModel)

Tutorial III (2)

5) Load the data (MNIST dataset)
   - Training data, test data, normalization
   - Plot examples

6) Build a NN
   - Model, loss function, optimizer
   - Training function
   - Testing function
   - Train the model (loss curves)
   - Get the accuracy of the model

Tutorial III (2)

7) Exercise 1 : build a NN with two layers

Exercises

Exercise 1

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

# Définition du réseau
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(784, 1500)  # Input size = 784 (ex: MNIST), output = 1500
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(1500, 10)  # Output size = 10 (classes)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.softmax(x)
        return x

# Données fictives pour l'exemple
# 784 features (ex: pixels de MNIST), 1000 échantillons
X = torch.rand(1000, 784)
y = torch.randint(0, 10, (1000,))  # Classes entre 0 et 9

# Dataset et DataLoader
dataset = TensorDataset(X, y)
dataloader = DataLoader(dataset, batch_size=64, shuffle=True)

# Initialisation du modèle, de la fonction de perte et de l'optimiseur
model = SimpleNN()
criterion = nn.CrossEntropyLoss()  # Perte adaptée pour classification
optimizer = optim.Adam(model.parameters(), lr=0.001)  # Taux d'apprentissage initial
```

```python
# Entraînement du modèle
def train_model(dataloader, model, criterion, optimizer, epochs=10, lr_scheduler=None):
    for epoch in range(epochs):
        total_loss = 0
        for inputs, labels in dataloader:
            optimizer.zero_grad()  # Réinitialiser les gradients
            outputs = model(inputs)  # Passage avant
            loss = criterion(outputs, labels)  # Calcul de la perte
            loss.backward()  # Calcul des gradients
            optimizer.step()  # Mise à jour des paramètres
            total_loss += loss.item()

        # Ajustement dynamique du taux d'apprentissage
        if lr_scheduler:
            lr_scheduler.step()

        print(f"Epoch {epoch+1}/{epochs}, Loss: {total_loss:.4f}, LR: {optimizer.param_gro

# Ajustement du learning rate avec ReduceLROnPlateau
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patien

# Entraîner le modèle
train_model(dataloader, model, criterion, optimizer, epochs=10, lr_scheduler=scheduler)
```

Copier le code

Tutorial IV : Convolutions

Link → Copy

Introduction

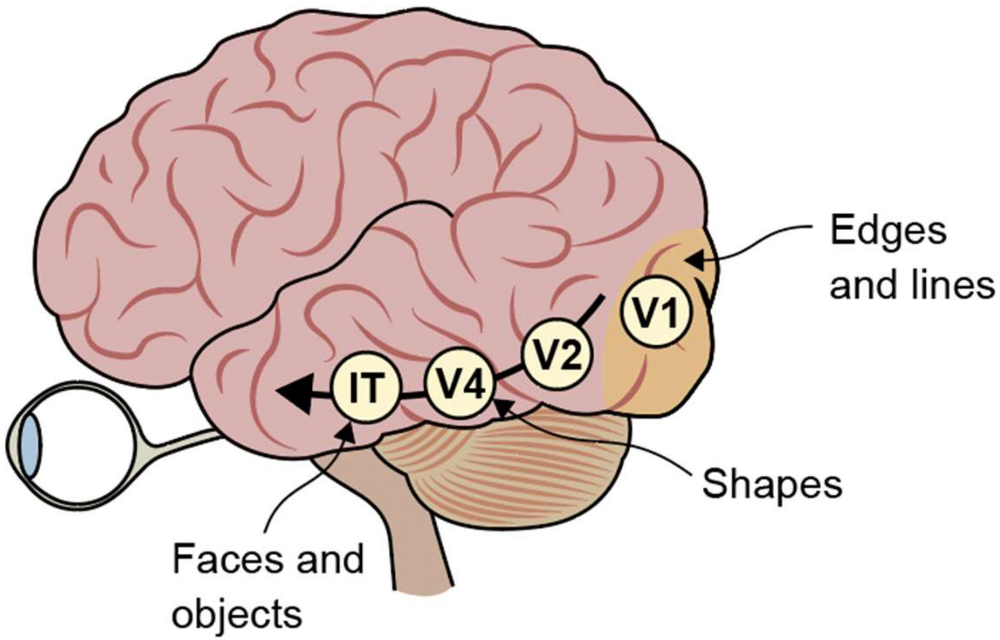
- Goal : basics to perform image recognition (Inception)

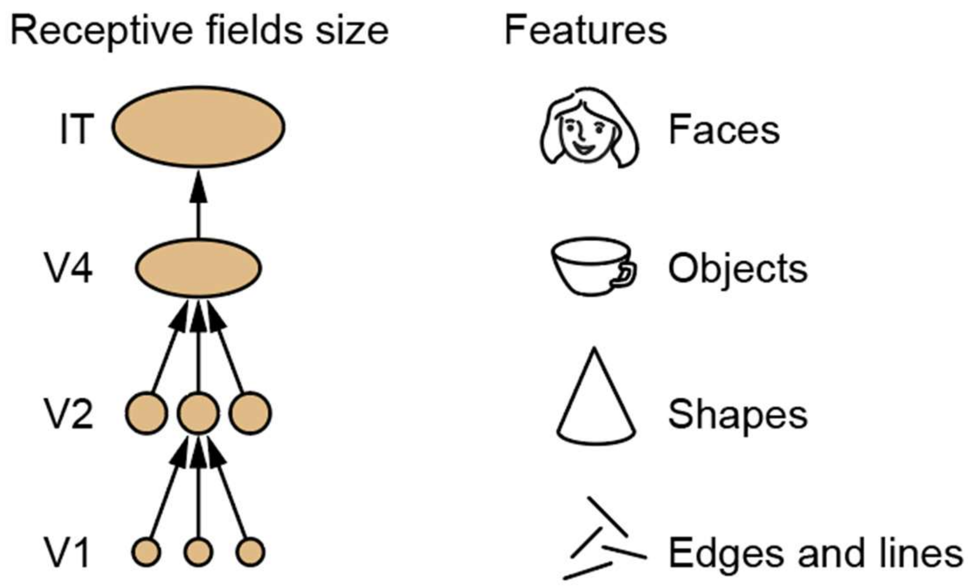
- Program : inverted classroom style
  - Theory
  - Overview to get the big picture of the Notebook
  - Work alone or in groups

- Technical : Google Colab, Pytorch

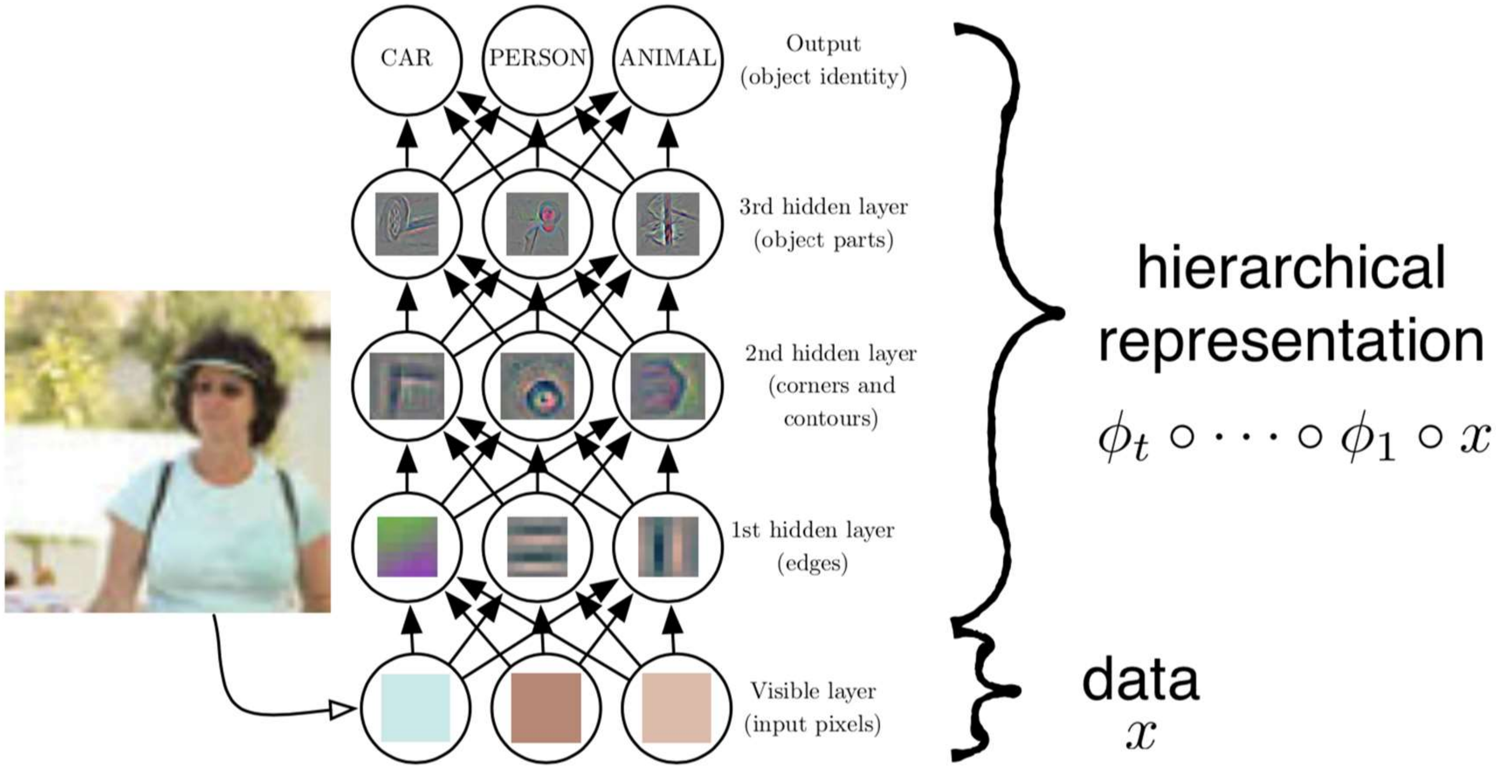
Theory

Human vision





Computer vision

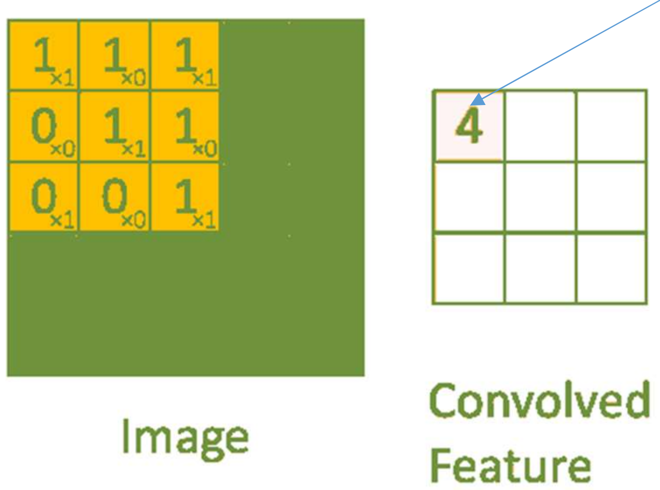


Kernel (filter)

- Used to detect features (vertical/horizontal filter,...)

- Different kernels to create different feature maps → learn to see various patterns and details in images

1	0	1
0	1	0
1	0	1



how well the pattern in the kernel matches the content in that part of the image

= Feature map

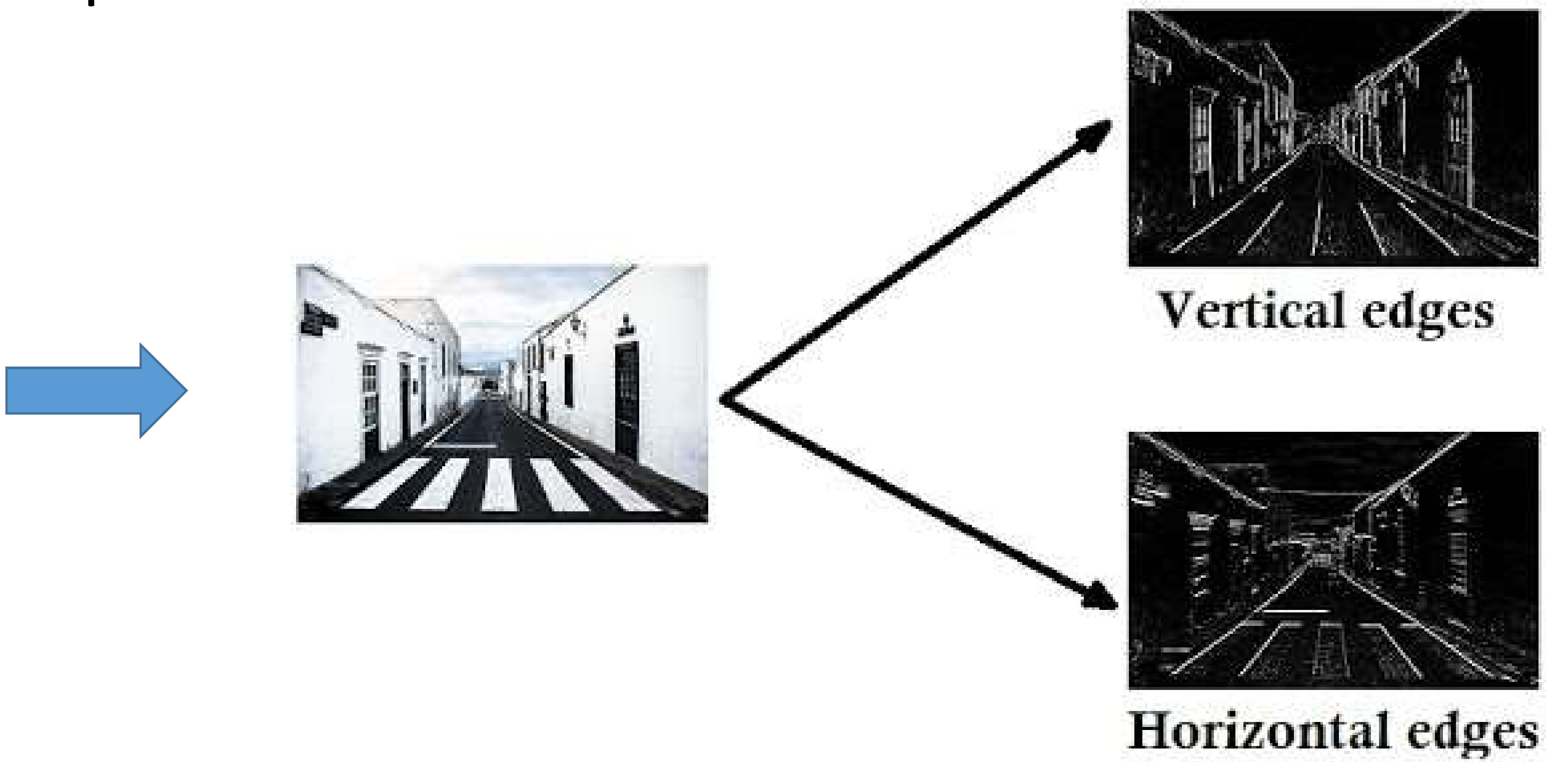
Kernel example

1	0	-1
1	0	-1
1	0	-1

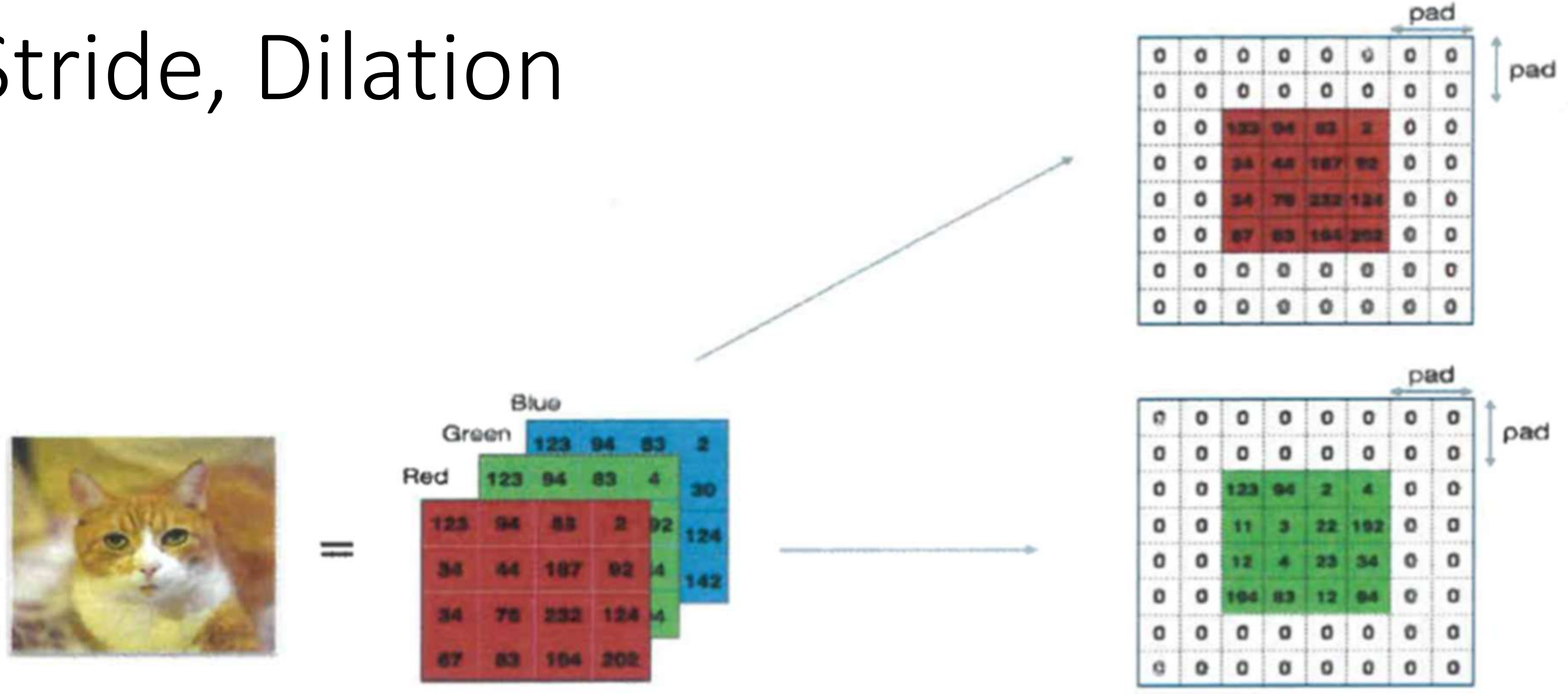
Vertical

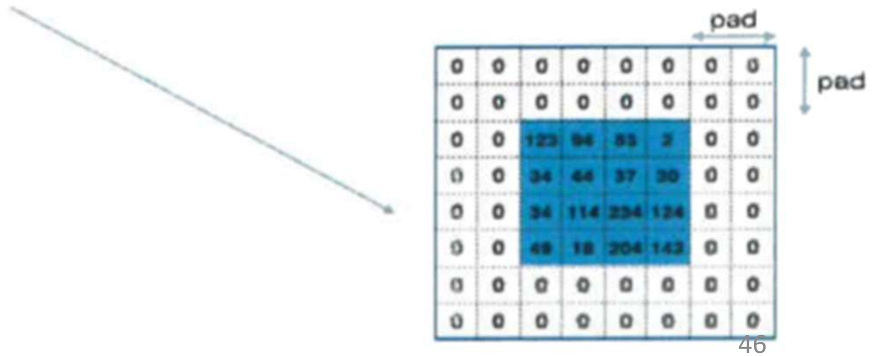
1	1	1
0	0	0
-1	-1	-1

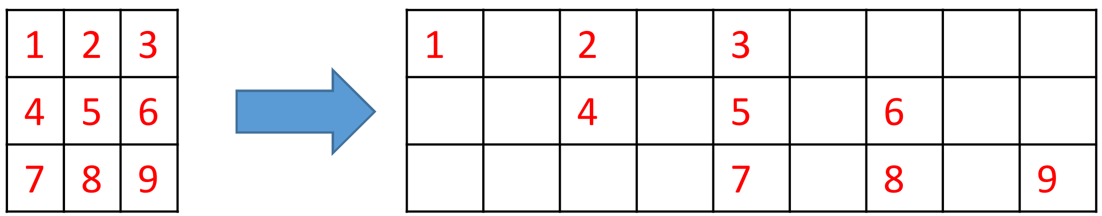
Horizontal



Padding, Stride, Dilation

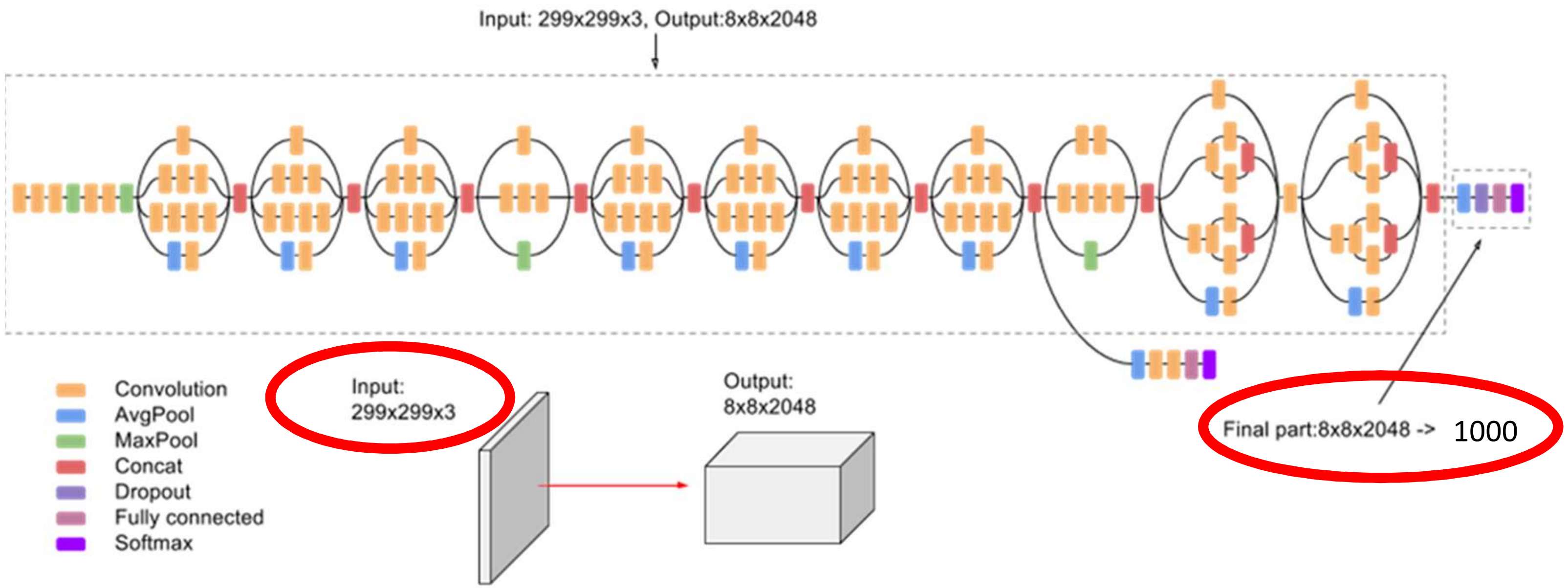






Inception V3 model

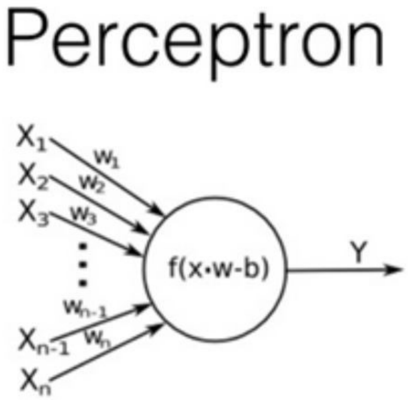
- Deep learning model based on Convolutional Neural Networks
- Used for image classification
- Released in year 2015
- It has 42 layers

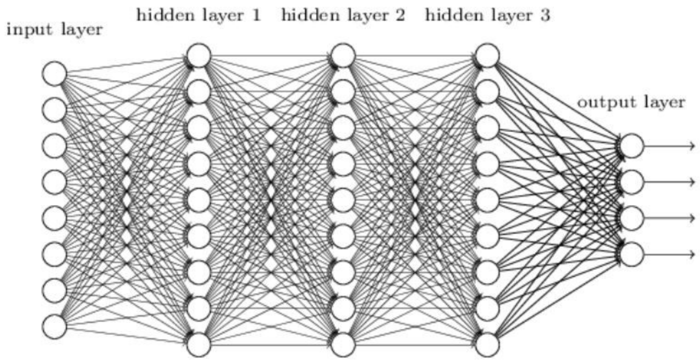


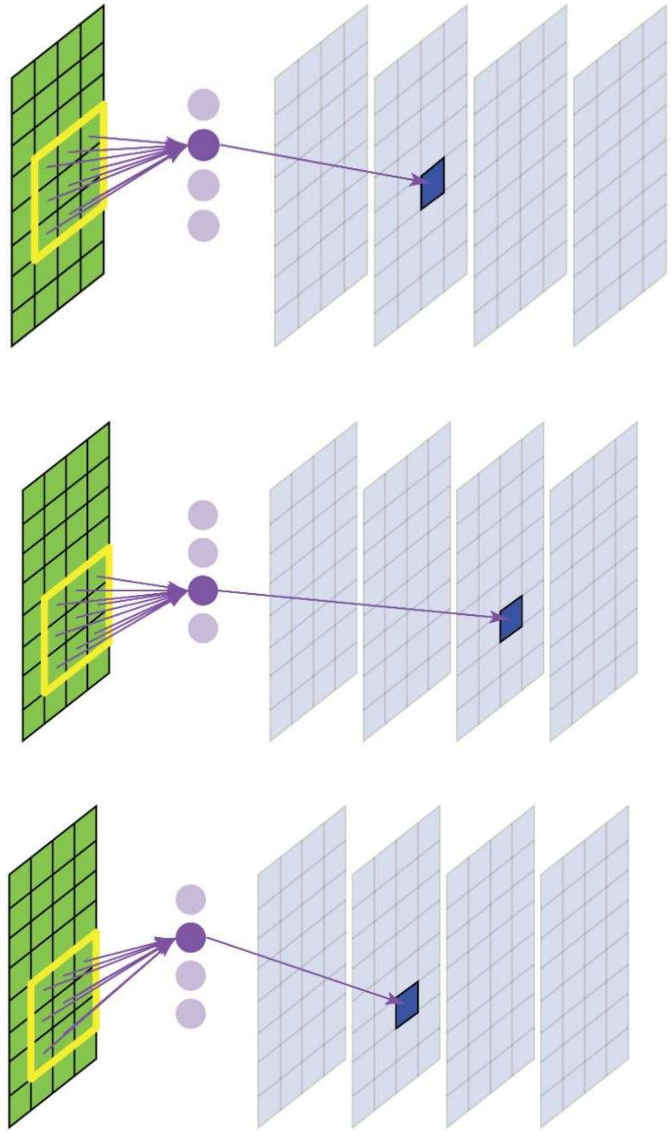
Overview of the notebook

Tutorial IV (1)

1) Load necessary libraries (common libraries and personal modules)
2) Images
3) Convolutions

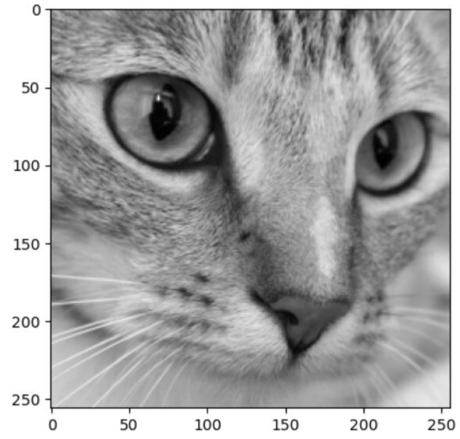






Tutorial IV (2)

- Pre-processing of the image
  - Gray-scale, cropping, float conversion, normalization
  - Add dimensions (batch, channel, height, width) (in get_convolved)
  - Convert Numpy to pytorch (in get_convolved)



- Define the convolution
  - Forward model (class Model)
  - Apply 4 convolutions one after each other (inside class Model, call conv_2d function)

- Define the filter
  - Identity filter
  - Convert to np.array (in get_convolved)
  - Add dimensions
  - Convert Numpy to pytorch

```
flt_mtx = [
    [ 0, 0, 0, 0, 0,],
    [ 0, 0, 0, 0, 0,],
    [ 0, 0, 1, 0, 0,],
    [ 0, 0, 0, 0, 0,],
    [ 0, 0, 0, 0, 0,],
] # identity transformation
```

Tutorial IV (3)

- Use it ! (ims_convolved = get_convolved(img_raw, flt_mtx)
  - You get back 5 figures

0	0	0	0	0
0	0	0	0	0
0	0	30	0	0
0	0	0	0	0
0	0	0	0	0

0	0	0	0	0
0	0	0	0	0
0	0	-1	0	0
0	0	0	0	0
0	0	0	0	0

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	1	0	0
0	0	0	0	0

- Exercise :

1. experiment with different filters and understand what they do, e.g.:

- identity transformation
- identity transformation with positive non-unit values
- identity transformation with negative unit value
- identity transformation off center
- blurring with box filter
- edge detection with + and - bands
- try whatever you like

2. experiment with convolution parameters:

- padding = 1, 2, 3
- stride = 2
- dilation = 2

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

0	-1	1	0	0
0	-1	1	0	0
0	-1	1	0	0
0	-1	1	0	0
0	-1	1	0	0

Tutorial IV (4)

filter type	effect
gaussian	bluring
first derivative of gaussian	detection of edges
second derivative of gaussian	detection of peaks

- Most common filters
  - Define 1D functions
  - Create 2D filters by repeating the 1D filters along axis 0 (np.tile)
  - Multiply by transpose() to get the horizontal dimension (filter size does not change)
  - Use them ! (ims_convolved = get_convolved(img_raw, flt_mtx)

4) Homework (leave it for now)

Tutorial IV (5)

5) Load a pretrained model (inception V3) from torchhub

6) Test the model

- Preprocessing of the image (cropping, shuffle sizes, totensor, normalize adds batch size) → [1,3,299,299]
- Desactivate the gradient (eval mode)
- Get the logits (1000 values), then the probabilities (applying softmax)
- Print out the 5 most probable classes
- Do the same with 100 classes

Tutorial V : Transfer Learning

[Link]

Introduction

- Goal : use the Inception model to classify images of different nature (it/de), learn how to save a model

- Program : inverted classroom style
  - Theory
  - Overview to get the big picture of the notebook
  - Work alone or in groups

- Technical : Google Colab, Pytorch

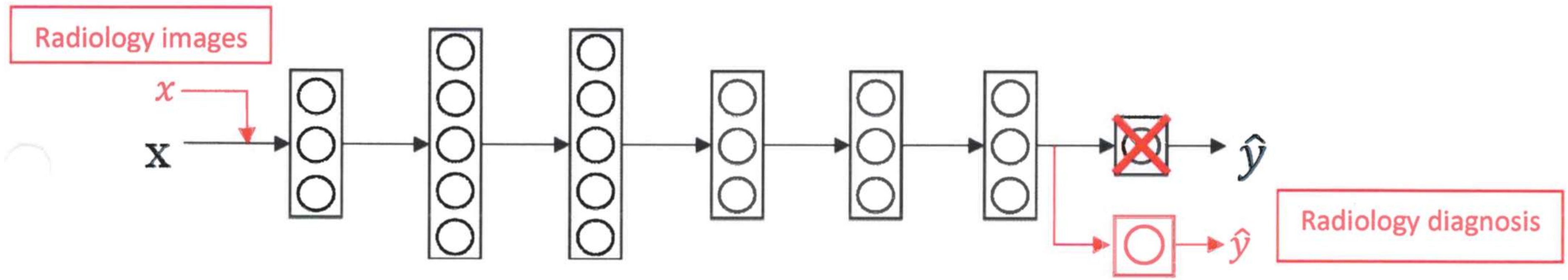
Theory

Transfer Learning

- Try to find an existing neural network that accomplishes a similar task to the one you are trying to tackle

- reuse the lower layers of this network
  - Output layer should usually be replaced

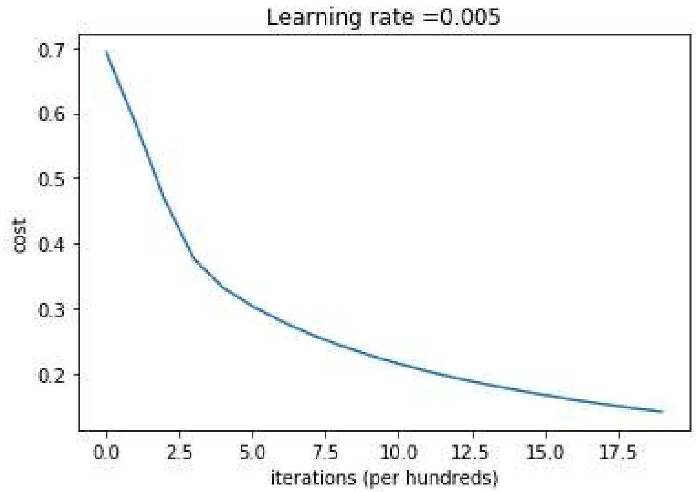
- Speeds up training and requires much fewer training data



Training Loop

- *Iterative* process

Learning curve



Forward propagation

Make a prediction

Cost function

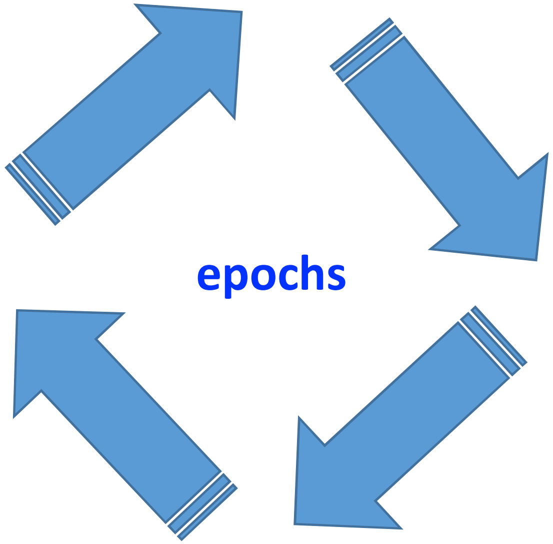
Compute the error function

Backward propagation

Measure the error contribution from each connection

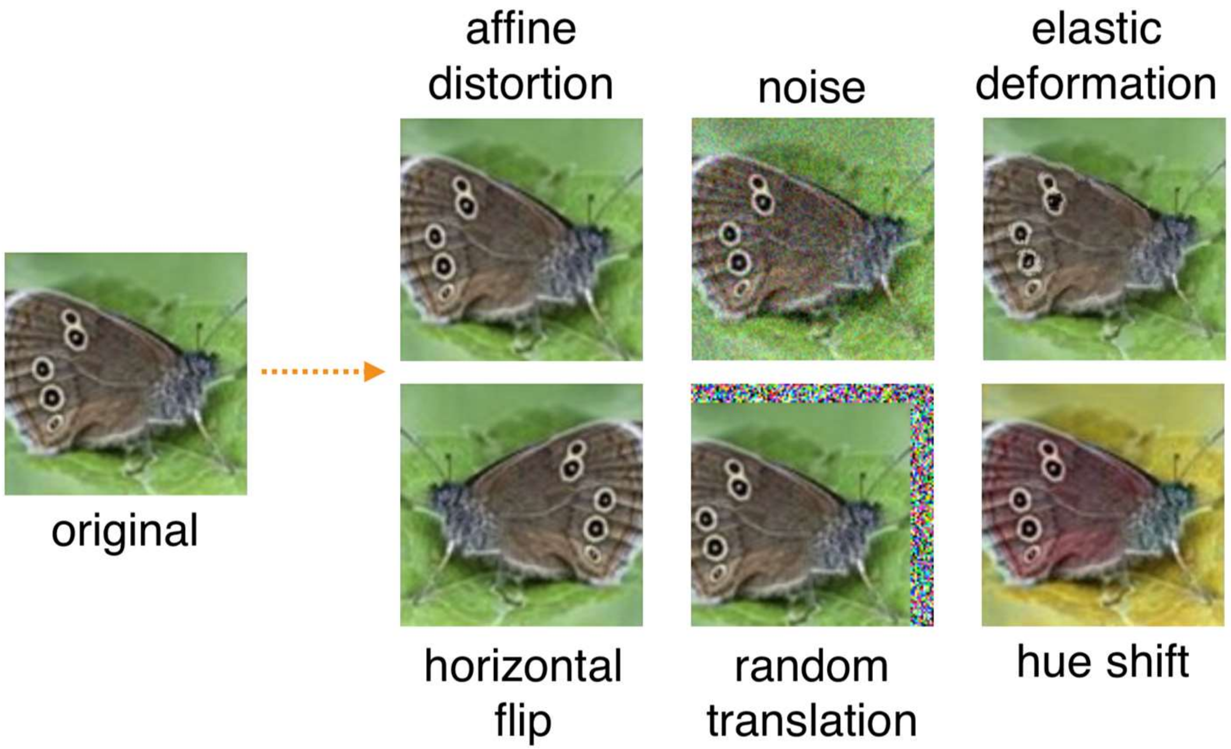
Gradient descent

Parameter update



Dataset Augmentation

- Apply realistic transformations to data to create new synthetic samples, with same label



Overview of the notebook

Tutorial V (1)

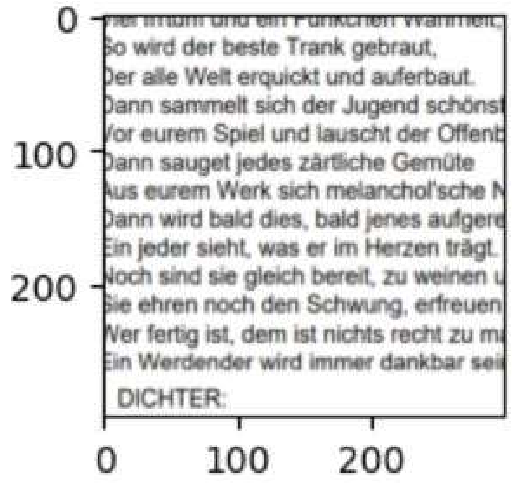
1) Load necessary libraries (common libraries and personal modules)
2) Transfer Learning

- Load the inception model (base_model)
- Build new model using the base_model (model)
  - Define a head function (in_features, n_classes=2) : use of sigmoid
- Optimization : define the loss function (criterion) and the optimizer
- Helper functions to get the prediction (get_predictions) and to compute the batch accuracy (calculate_accuracy_batch)

Tutorial V (2)

3) Dataset

- Images :
  - Get them from ML3 folder
  - Preprocessing using transforms.Compose (resize, tensor, normalize)
  - Shape [3,299,299]
  - Transform to numpy array for display (im_numpy)
- Labels



- Split dataset into train/test samples
  - Torch.utils.data.random_split
  - Train_test_split with stratify enabled from scikit-learn
- Create data loaders for training/val (beware shuffle param)
- Example : batch of 10 images, plot them, print logits and output classes (res)

Tutorial V (3)

4) Training

- Train_model function
  - Contains the loop on epochs
  - Calls the train and validate functions (beware the params)
  - Save history (loss, accuracy) and model for a given epoch

  - Train function : reset gradients, compute logits and loss, compute gradients (backward prop), update parameters, calculate accuracy of the batch (helper functions), returns train loss and train accuracy

  - Validate function: structure BUT (no optimizer, no backward, no update of the params), returns test loss and test accuracy

- Plot_history function
- Run it ! history = train_model(...) using 70 epochs

Tutorial V (4)

5) Load trained variables from checkpoint

- Choose epochs values
- Load corresponding models
- Call validate function to get the validation loss and validation accuracy (see how it evolves)

6) Save final model for inference

7) Inference :

- Load the model, eval() mode
- Get an image, preprocess (convert to tensor, add batch dimension
- Get the logits and associated class

Tutorial V (4)

8) Improve the results : data augmentation

- Load images from ML3 folder
- Preprocess (resize, Randomcrop, tensor, normalize)
- Convert to Numpy for plotting purposes (im_numpy)
- The rest of the code is similar to previous code

9) Exercise

Tutorial VI : RNNs

Link

Introduction

- Goal : use Recurrent Neural Networks to predict and generate text sequence
- Program : inverted classroom style
  - Theory
  - Overview to get the big picture of the notebook
  - Work alone or in groups
- Technical : Google Colab, Pytorch

Theory

Overview of the notebook

Tutorial VI (1)

1) Load necessary libraries (common libraries and personal modules)

2) Text data

- Read the data (rnn.txt), print the first 100 words

3) Build the dataset

- 2 dictionaries : word → id (dictionary) and id → word (reverse_dictionary)
  - Build_dictionaries function
- Vocabulary size = 493 (0=most common word)
- Helper functions to get sequences of int or words (text_to_ints, ints_to_text)
- Print example : first 100 words (or int), length of input data=2118 (words_as_int)

Tutorial VI (2)

3) Data streaming

- Create dataset using WordDataSet class to create blocks of text
  - Block length = n_input+1 = 3+1 = 4
- Create DataLoader with batch size=50, preprocess data (separate input and target sequences, stack data and convert Numpy→tensors, put them to GPU)
  - Length of dataset = Number of blocks in sequence = Total length/Block length = 2118/4=529
- Example : print the 50 samples that are in the 1st batch

Tutorial VI (3)

4) Construct model

- Create class RNN
  - Embedding layer (vocab_size, embedding_dim=128)
  - Loop to add the 3 LSTM layers
  - FC layer with vocab_size
- Define sequence of 3 words (n_input), define dimension of 3 LSTM, call the RNN class
- Investigate the model using Tensorboard
  - Create an input of size=5 (x), transform to tensor, add batch
  - SummaryWriter to save the model and be able to open it with tensorboard
  - Print the output size of y : [5,1,493]
- Test the NOT trained model and see that it is bad
  - Get the first batch (break), apply the model, get the predictions, compare to true

Tutorial VI (4)

5) Train the model

- Params : n_input = 3, batch_size=50 , one LSTM layer (128),n_epochs=200
- Create the data loader (preprocess data)
- Optimization part : criterion, optimizer (RMSprop)
- Training loop on epochs, then on batches
  - Initialize gradients
  - Get the output (seq_len, batch_size, vocab_size), reshape it to (seq_len*batch_size, vocab_size)
  - Reshape labels (seq_len, batch_size) to (seq_len*batch_size)
  - Compute the loss between output and true labels
  - Backward prop, param update
  - Compute loss and accuracy
- Plot loss and accuracy

Tutorial VI (5)

6) Generate text with RNN

- Function to generate text (gen_long)
  - Input parameters : model, input sequence, number of words to generate (128)
  - No_grad() because we are in an evaluation mode
  - Loop on number of words, convert to tensor, predict, …)