CS5242 : Neural Networks and Deep Learning

Lecture 1 : Introduction to Deep Learning

Semester 2 2025/26

Xavier Bresson

https://twitter.com/xbresson

Department of Computer Science
National University of Singapore (NUS)

Outline

- AI/DL case studies
- A brief history of AI/DL
- 2012 DL breakthrough
- DL is a universal learning technique
- Terminology AI/ML/DL
- Key ingredients of DL
- AI is the new electricity
- Limitations of AI

Outline

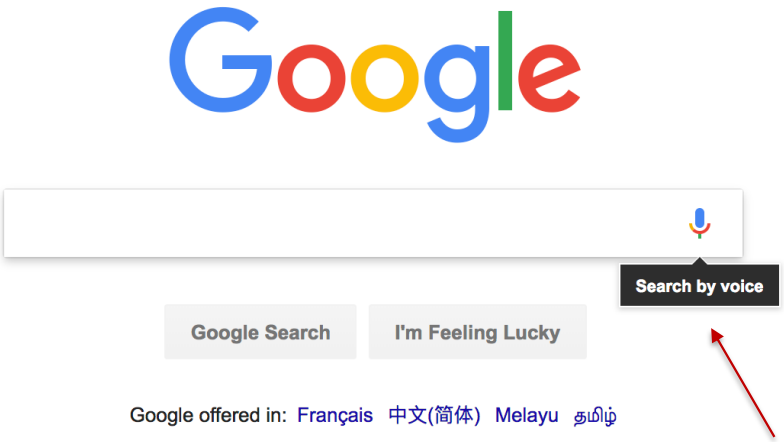
- AI/DL case studies
- A brief history of AI/DL
- 2012 DL breakthrough
- DL is a universal learning technique
- Terminology AI/ML/DL
- Key ingredients of DL
- AI is the new electricity
- Limitations of AI

Deep learning case studies

- AI/DL is more and more present in our daily life:
  - Voice recognition/synthesis
  - AI assistant
  - Machine translation
  - Object recognition
  - Self-driving cars
  - Games
  - Robots
  - Face recognition
  - Healthcare
  - Drug discovery

Google search

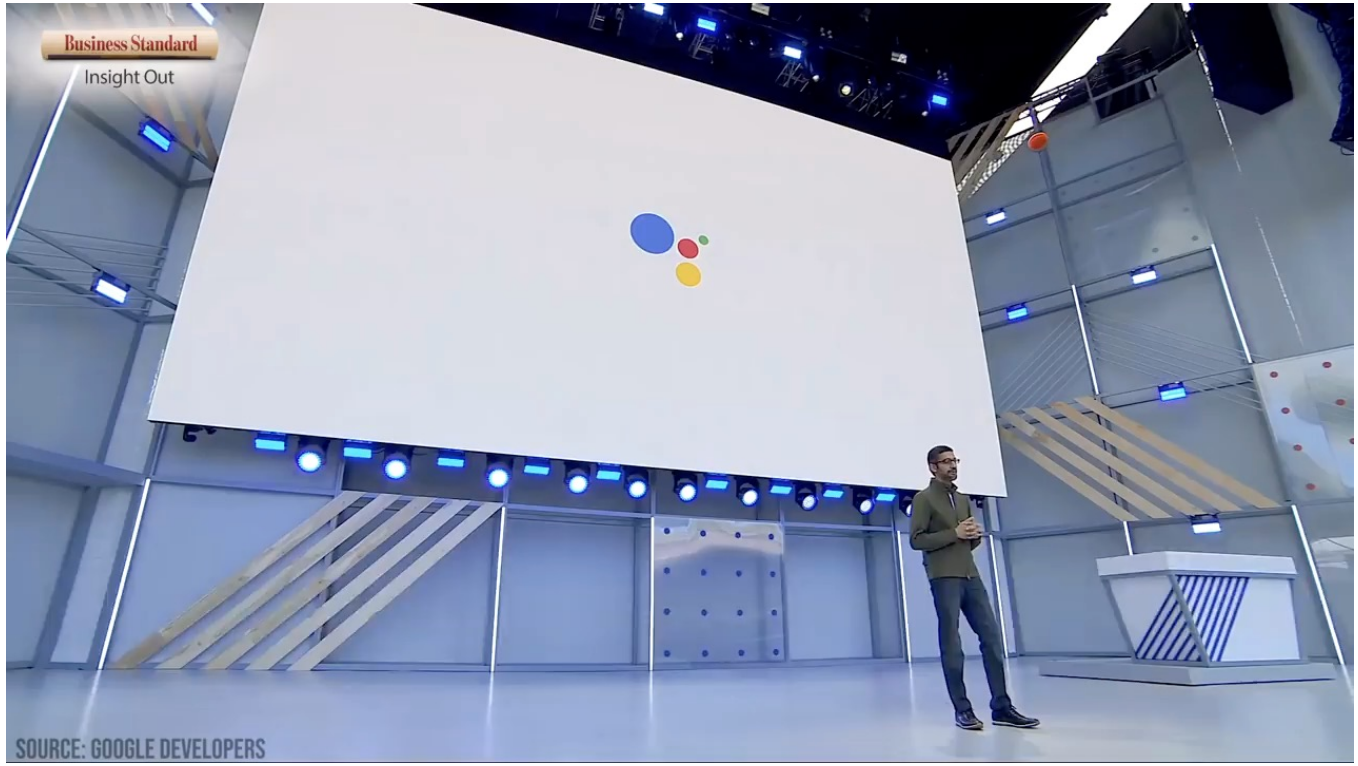
- Three neural networks
  - NN1 : Voice recognition w/ MLP (first product in 2012)
  - NN2 : Q&A w/ LSTM (first product in 2015)
  - NN3 : Voice synthesis w/ WaveNet (first product in 2016)



Example: What is deep learning?

2018 Google AI assistant

- Google AI assistant : Voice recognition - Q&A – voice synthesis



Google's May 2018 I/O developers conference

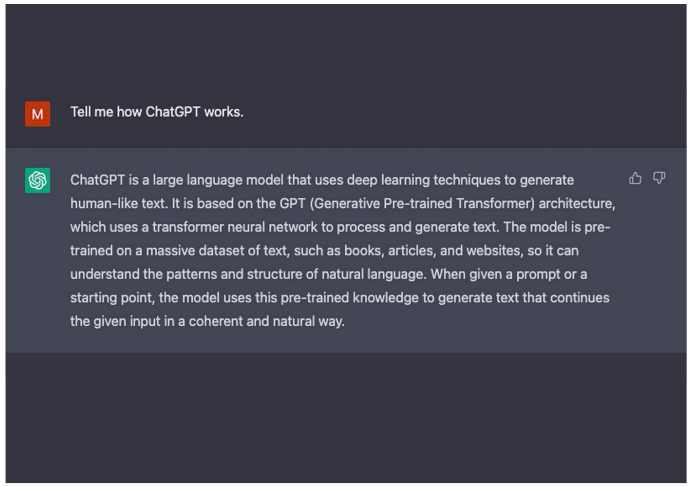
Turing test passed?

https://en.wikipedia.org/wiki/Turing_test

2022 OpenAI chatbot

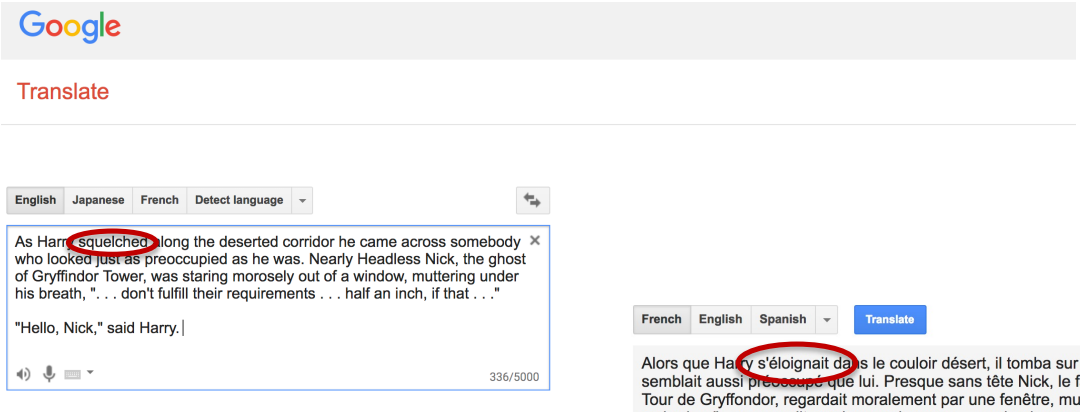
- ChatGPT revolution
  - Released on Nov 30, 2022.
  - Adoption to 100M users in 2 months.
  - Launched the GenAI industry ($137B in 2025).
  - LLMs are the new search engine, coder helper, math assistance, English teacher, etc.
  - They are English-prompt Wikipedia.
  - But they also hallucinate, create false facts.

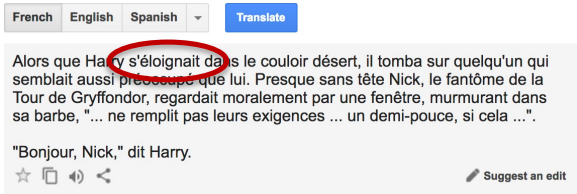
https://chat.openai.com

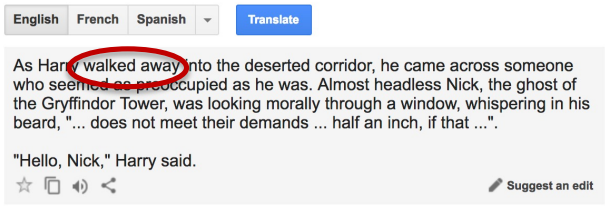


Machine translation

- Harry Potter and The Chamber of Secrets : English ⇒ French ⇒ English



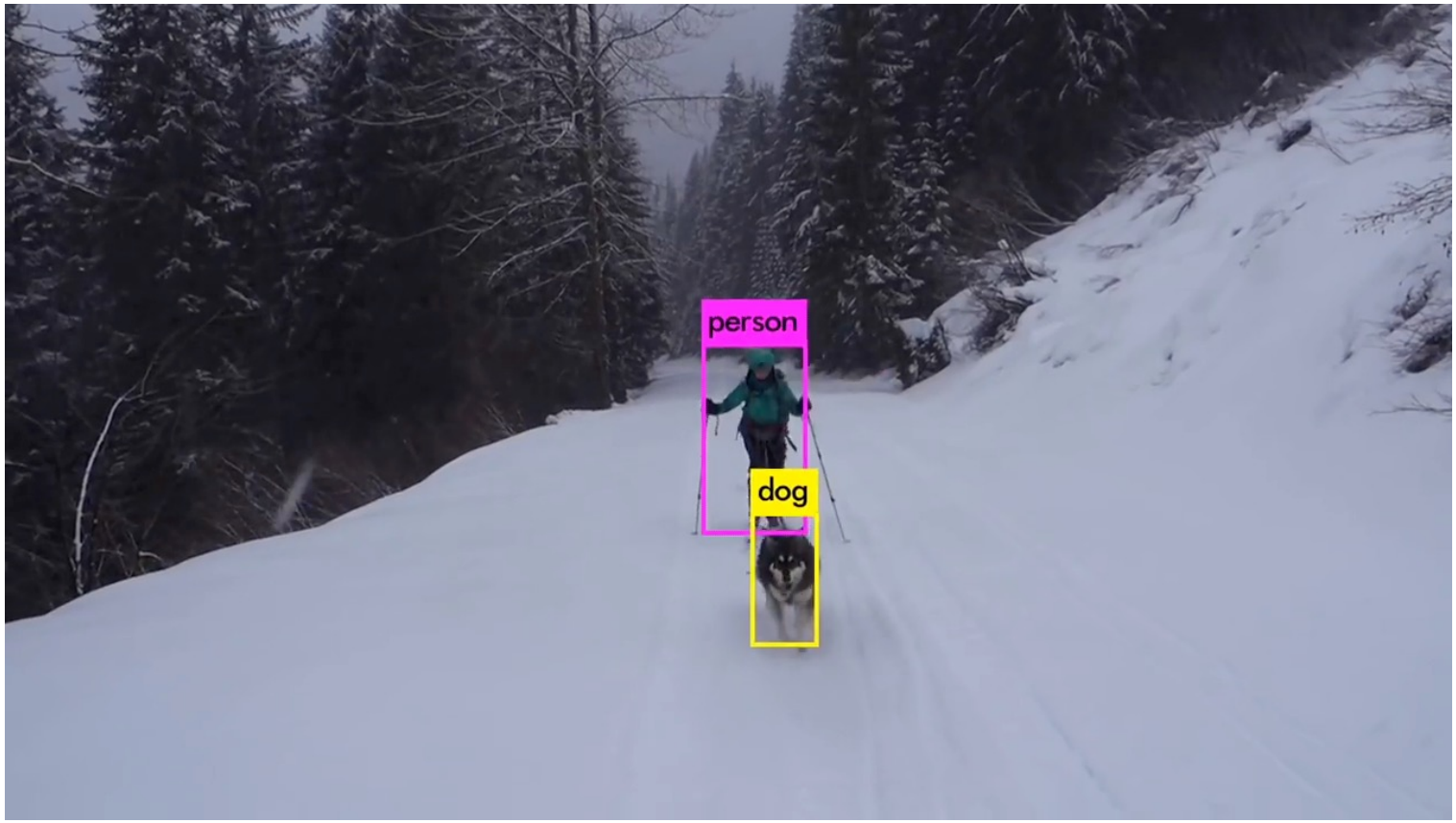




In 2020: Still far from professional translators, but some translators use it as first draft. Today in 2026, it is not true anymore! It is the end of this profession.

Object recognition

- Real-time object detection and recognition :



YOLO
(open-source)
https://pjreddie.com/darknet/yolo

Self-driving cars

- Requires to solve several computer vision tasks simultaneously :

NVIDIA
2017

In 2017: I thought that autonomous cars will be the most visible AI application in the coming decade.
It did not! ChatGPT was the AI breakthrough for general public to experience AI and apply AI to everyday life tasks.

Strategy games

- Google DeepMind AlphaGo defeated the world champions at Go, March 2016, May 2017

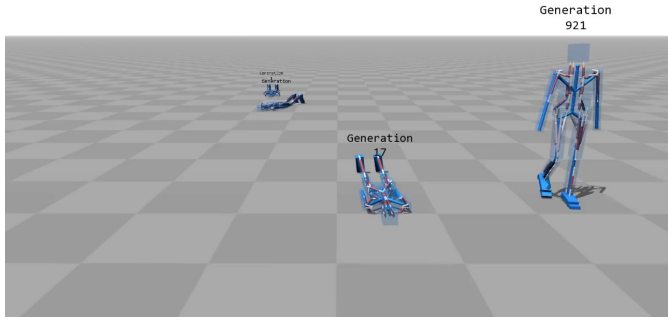
Number of possible board configurations is greater than the number of atoms in the universe

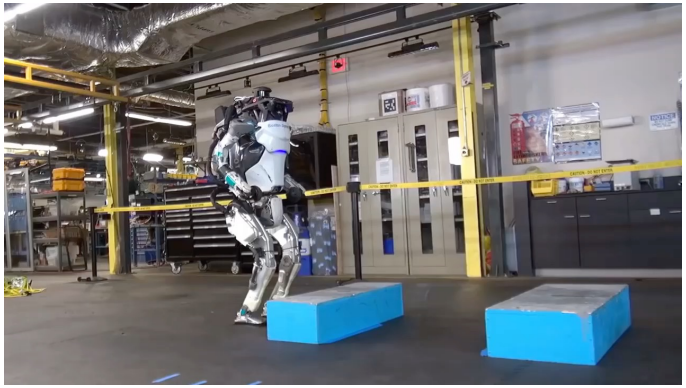
Doom (Facebook)
AI international competition

OpenAI Five: Dota2
The International 2018

Simulated and real-world robots

- Autonomous robots :





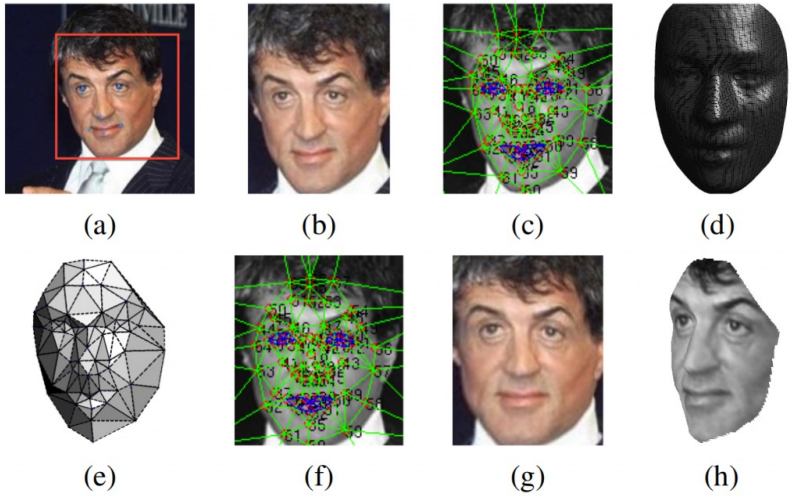
This is not AI-based robots (Boston Dynamics).

This is AI-based robots.

Today, in 2026, we are still far away from autonomous robots !

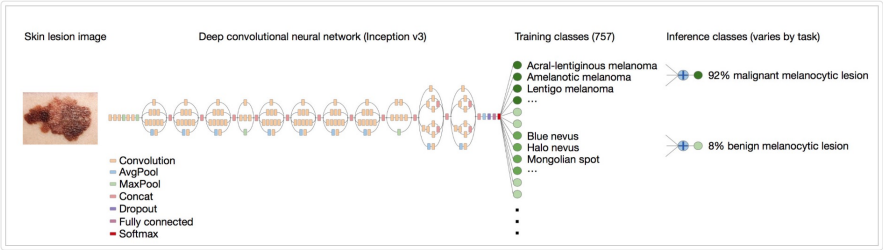
Face recognition

- Facebook face recognition system :



Healthcare

- Dermatology imaging : detection of skin cancer



Dataset: 129K clinical images consisting of 2,032 different diseases

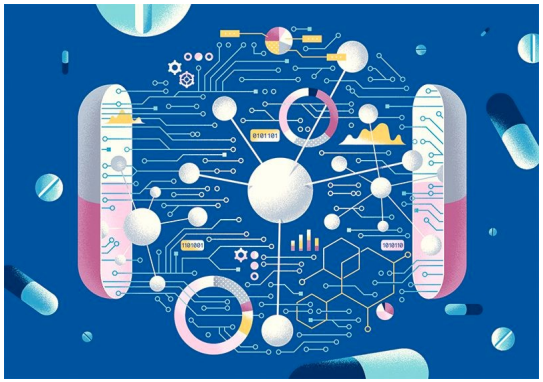
Performance on par or slightly better than >20 dermatologists, Feb 2017

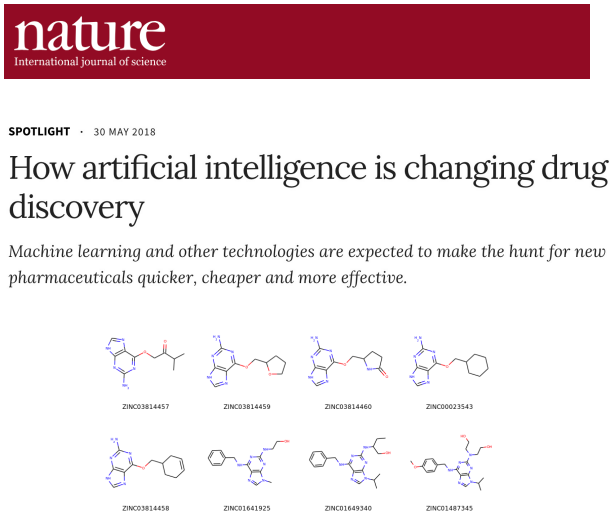
Shanghai Huashan Hospital elite radiologists vs. BioMind, an AI system, July 2018

AI made 87% prediction in 15 min.
Human doctors made 66% prediction in 30 min.

Drug discovery

- Learn to generate molecules with special properties - personalized to patients :





nature
International journal of science

SPOTLIGHT · 30 MAY 2018

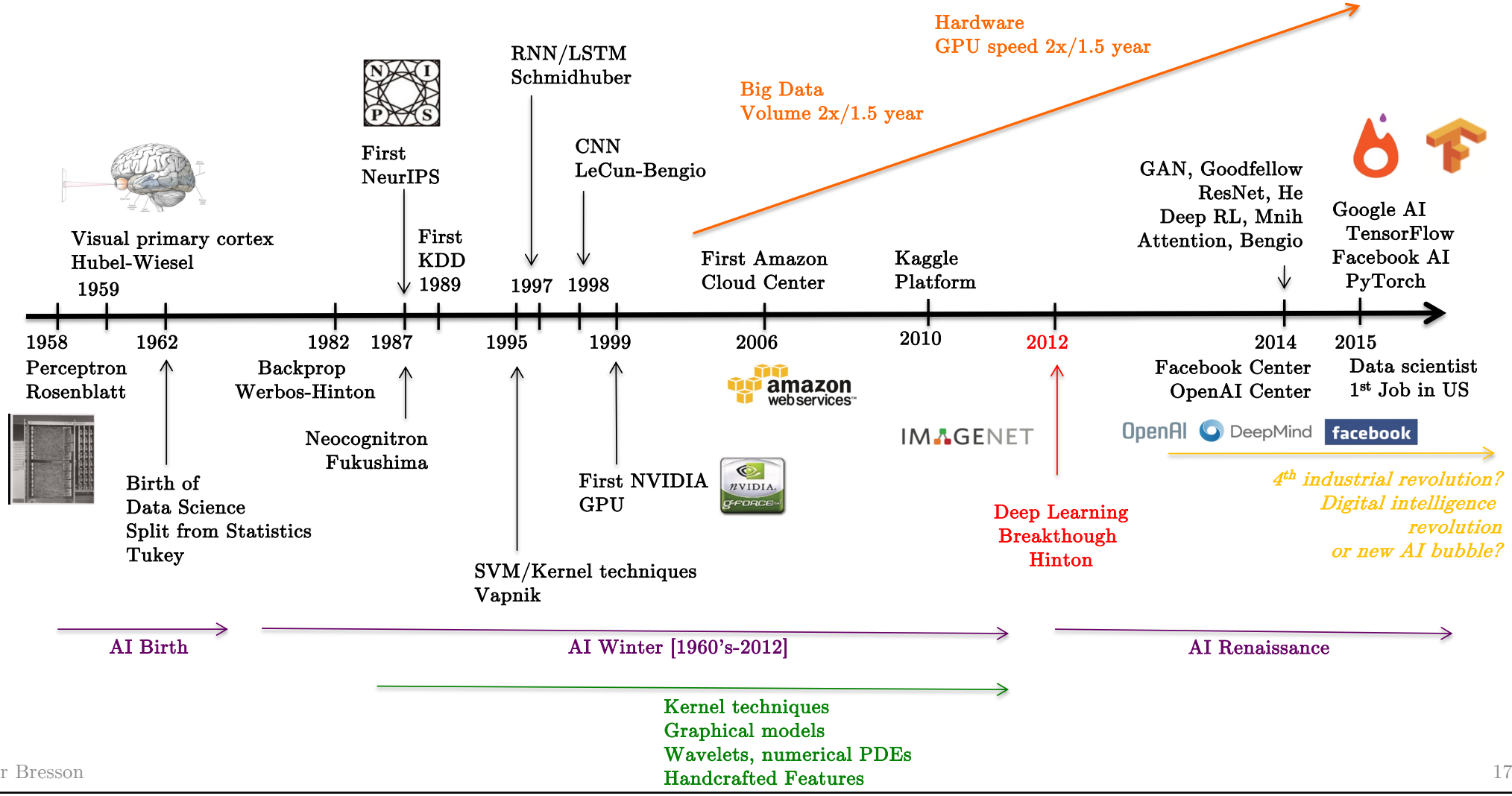
How artificial intelligence is changing drug discovery

Machine learning and other technologies are expected to make the hunt for new pharmaceuticals quicker, cheaper and more effective.

Outline

- AI/DL case studies
- A brief history of DL
- 2012 DL breakthrough
- DL is a universal learning technique
- Terminology AI/ML/DL
- Key ingredients of DL
- AI is the new electricity
- Limitations of AI

A brief history of DL



Outline

- AI/DL case studies
- A brief history of DL
- **2012 DL breakthrough**
- DL is a universal learning technique
- Terminology AI/ML/DL
- Key ingredients of DL
- AI is the new electricity
- Limitations of AI

2012: The year of DL breakthrough

- ImageNet challenge : International Image Classification Challenge
- Goal: Design the best algorithm for image recognition

  (one of the most basic problems in computer vision since 50 years)

Input: An image

Image recognition algorithm

Siamese Cat

Output: A category

2012: The year of DL breakthrough

- ImageNet challenge : International Image Classification Challenge

  - 1.4 million images
  - Collected from Flickr and other search engines.
  - 1000 object categories
  - Each image was manually labelled by a human being : They used a crowdsourcing approach to labeled the pictures with Amazon Mechanical Turk.

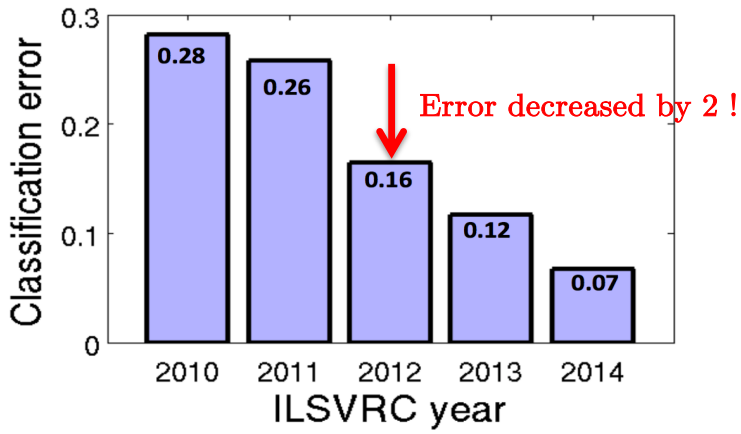
ImageNet dataset

2012: The year of DL breakthrough

- ImageNet challenge :
  - Training set: 1.2 million labelled images (given to the teams)
  - Test set: 0.2 million labelled images (kept by the organizers)
  - Each team use the training set to train their algorithm, then they submit it to the organizers.
  - The organizers evaluate each algorithm on the test set.

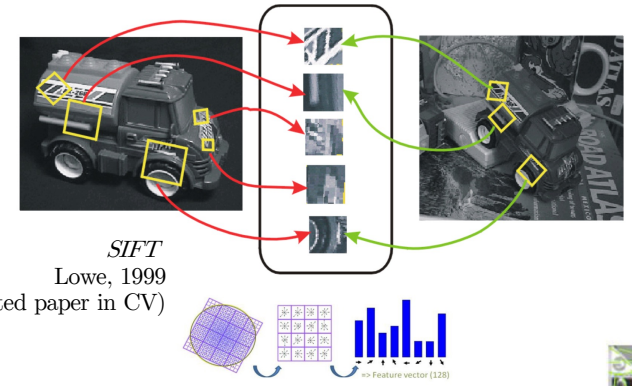
- Observe the plot between 2011 and 2012 :

  Change of paradigm : From 50 years of hand-crafted visual features to learned visual features from massive amount of data.



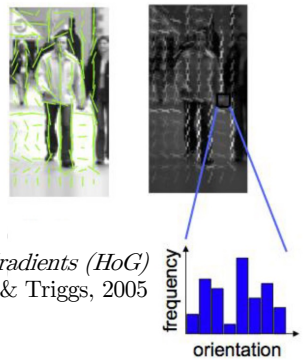
Before 2012

- Paradigm: Handcraft data features and feed them to a classifier.



SIFT
Lowe, 1999
(most cited paper in CV)

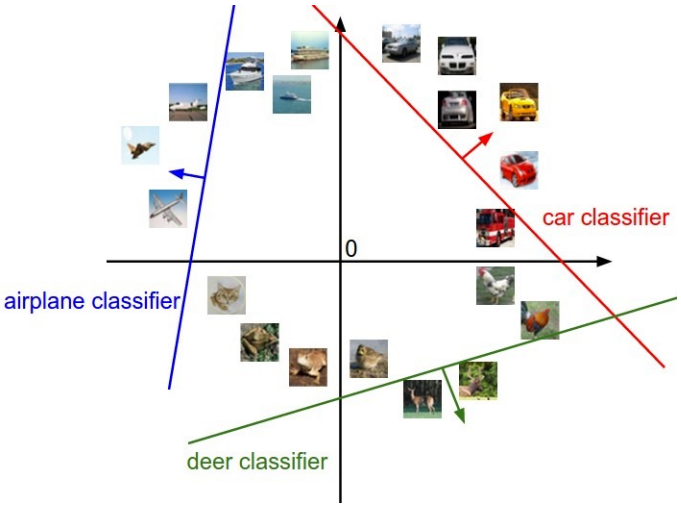
SIFT features



Histogram of Gradients (HoG)
Dalal & Triggs, 2005

HOG features

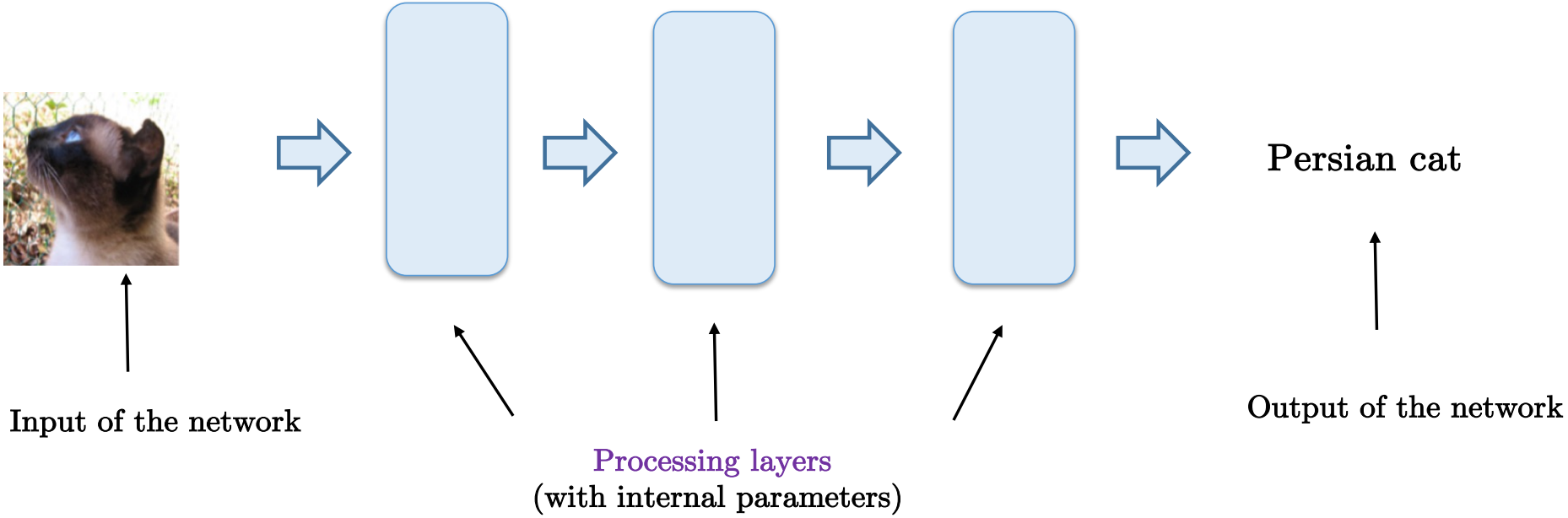
$\Rightarrow$



SVM classifier

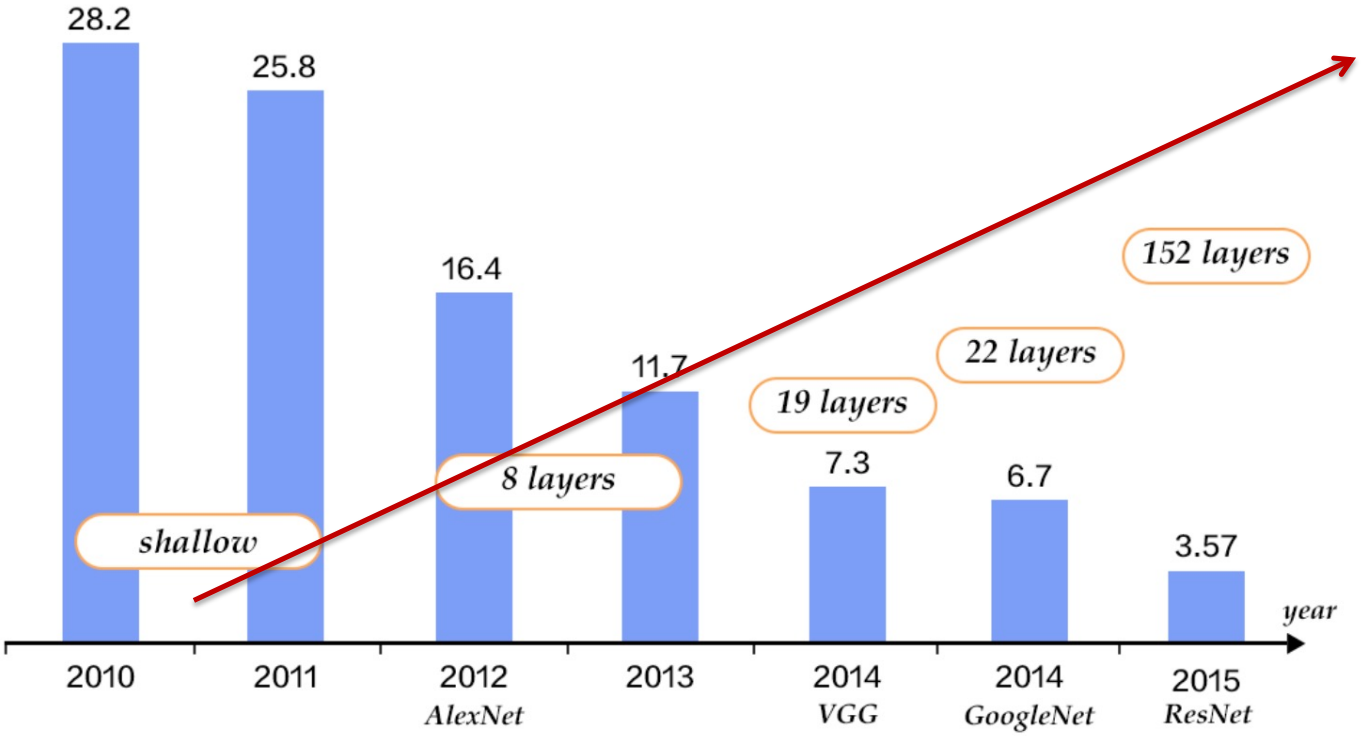
After 2012

- New paradigm: Learn data features and classifier together, a.k.a. end-to-end systems.
- Neural networks are the first class of models that can train end-to-end systems with large learning capacity by using multiple layers.



The deeper the better

- Deep Learning = Neural networks with many layers
- The deeper the more learning capacity :



2018
1000s layers

Outline

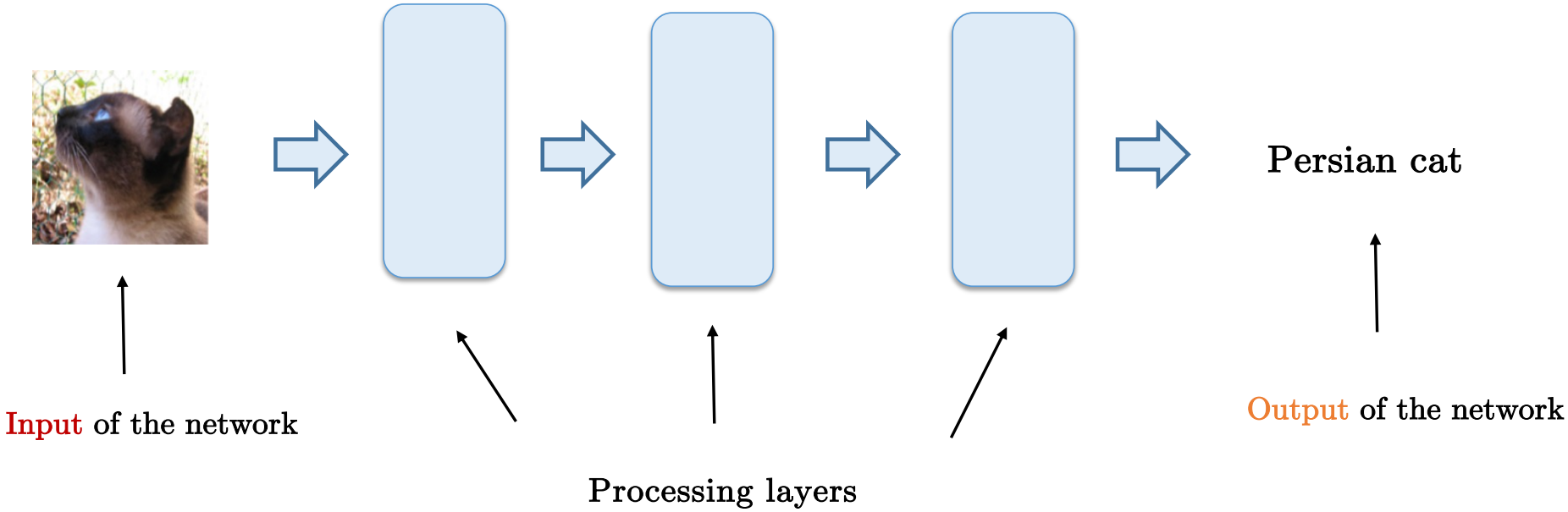
- AI/DL case studies
- A brief history of DL
- 2012 DL breakthrough
- **DL is a universal learning technique**
- Terminology AI/ML/DL
- Key ingredients of DL
- AI is the new electricity
- Limitations of AI

DL/NNs can be used for many tasks

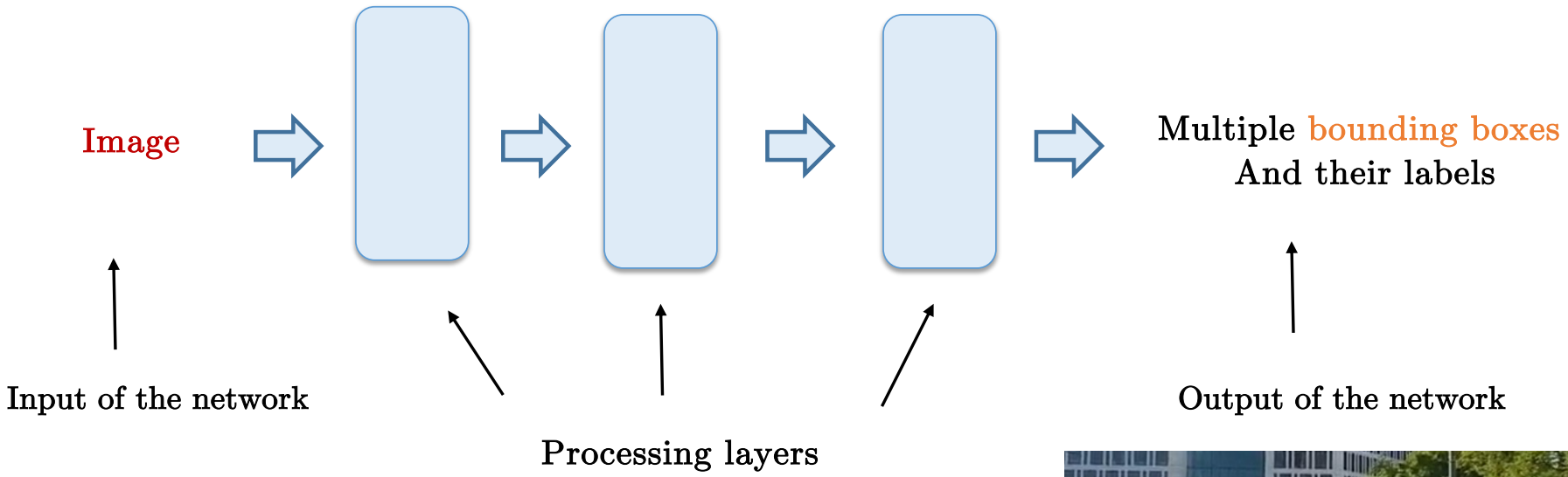
- Neural networks are universal learning techniques that can be applied to very distinct fields like computer vision, natural language processing, physics, chemistry, healthcare, biology, etc.

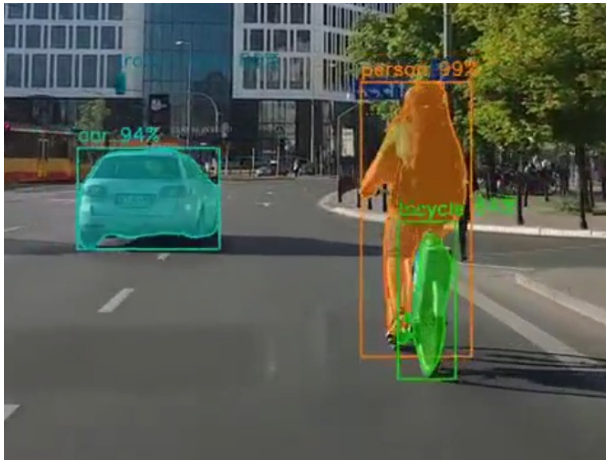
- Tasks where NNs provide the best performance (SOTA):
  - Image recognition
  - Object detection
  - Voice recognition
  - Speech synthesis
  - Translation
  - Recommender systems
  - ...

Image recognition

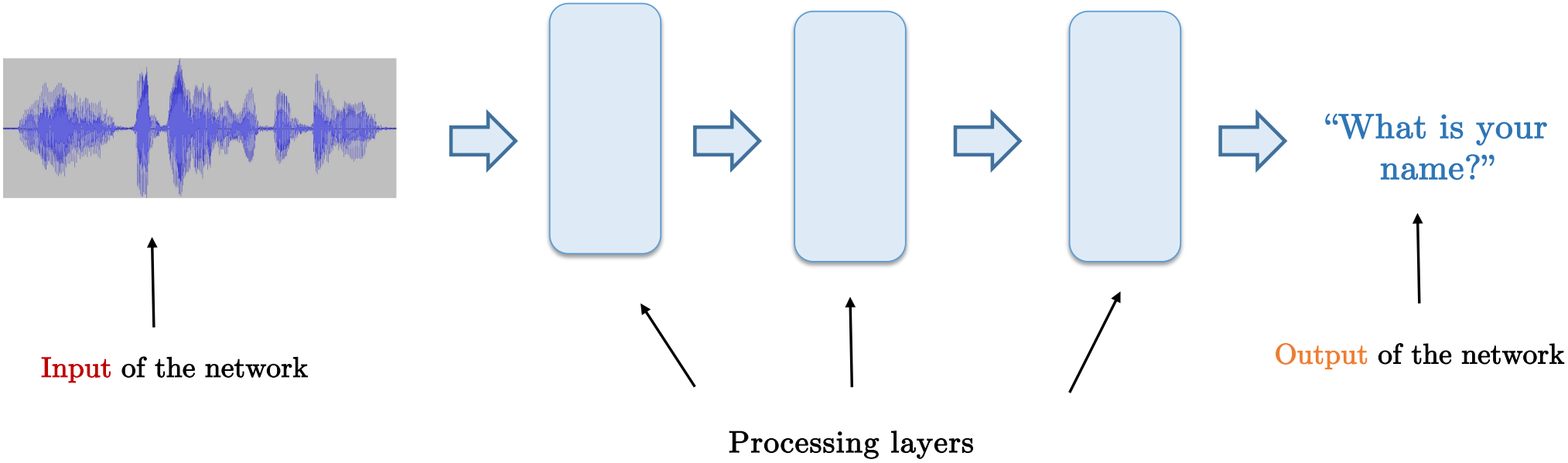


Object detection & recognition

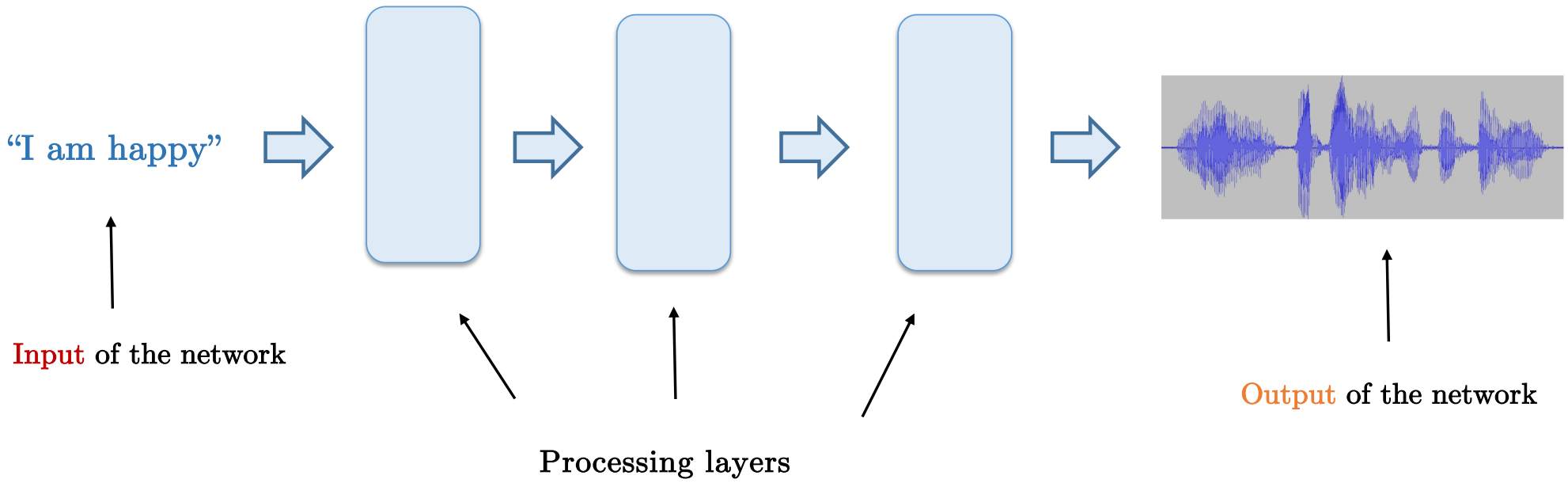




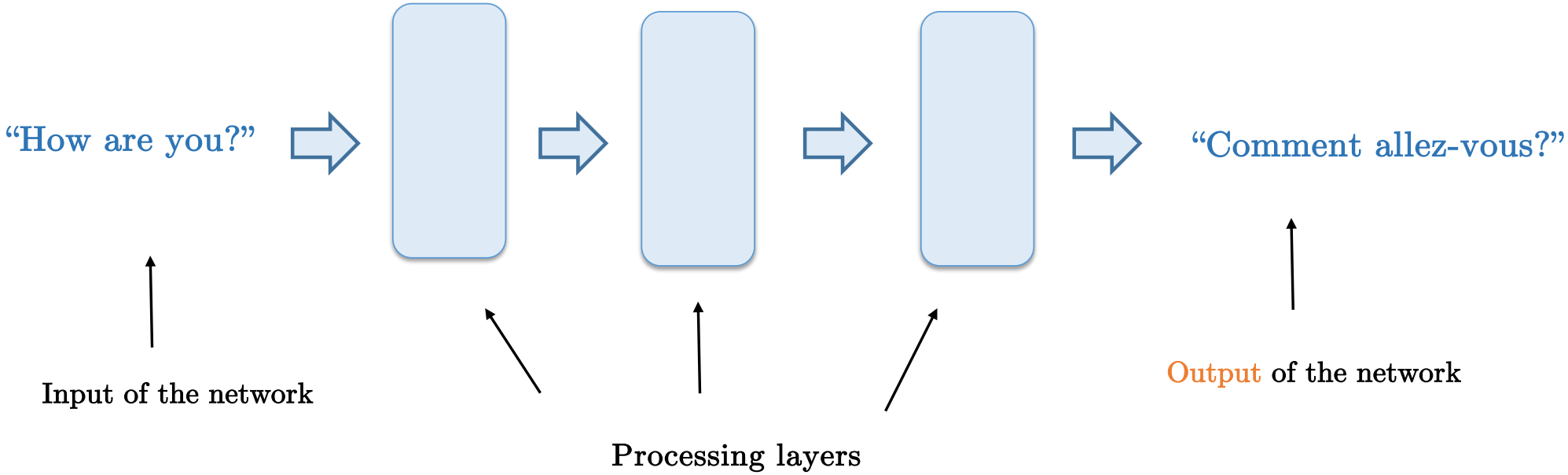
Voice recognition (speech-to-text)



Speech Synthesis (text-to-speech)

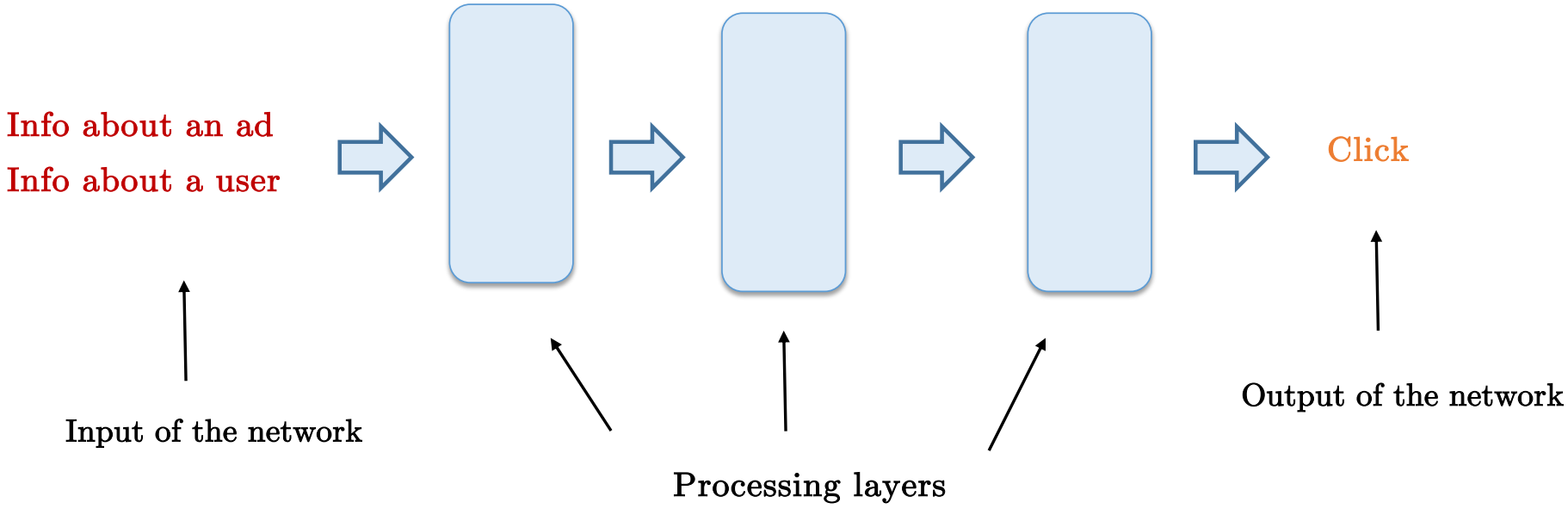


Translation (sequence-to-sequence)



Online advertisement

- Will this user click on this ad?

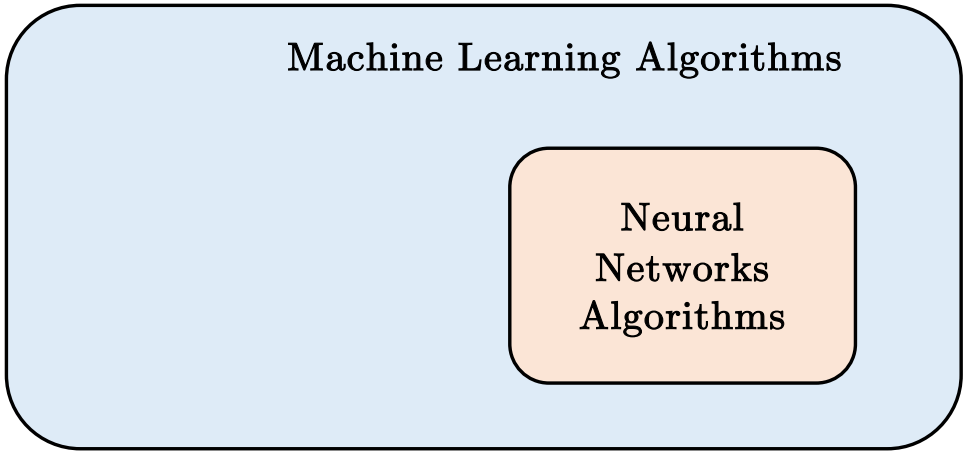


Outline

- AI/DL case studies
- A brief history of DL
- 2012 DL breakthrough
- DL is a universal learning technique
- **Terminology AI/ML/DL**
- Key ingredients of DL
- AI is the new electricity
- Limitations of AI

AI/ML/DL

- A.I. : Artificial intelligence, a vague term that can mean many things (no definition exists).
- Machine learning : Techniques that solve tasks from data learning.
- Deep Learning : Neural networks with many layers
- Nowadays, A.I. advances are exclusively based on deep learning algorithms.



Outline

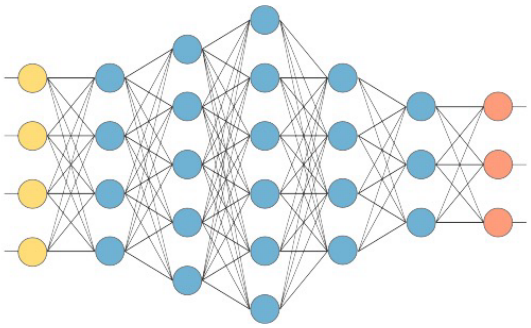
- AI/DL case studies
- A brief history of DL
- 2012 DL breakthrough
- DL is a universal learning technique
- Terminology AI/ML/DL
- **Key ingredients of DL**
- AI is the new electricity
- Limitations of AI

Key successful ingredients

- Deep Learning success = Big data + GPUs + Neural Networks

+

+



Big data (2016)

- We live in a digital society:
  - Eric Schmidt : "Every 2 days we create as much information as we did up to 2003."
  - Moore's law of data : Volume of data double every 1.5 years.

- Some datasets:
  - ImageNet : 1.4 millions images
  - Baidu faces : Baidu use a training set of 200 millions images for their face recognition system
  - Translation : Europarl parallel corpus
    - Extracted from the proceedings of the European parliament.
    - 2 million sentences translated in all of the languages of the EU
  - Self-driving cars : Uber's Self-Driving Cars Hit 2 Million Miles

Big data (2016)

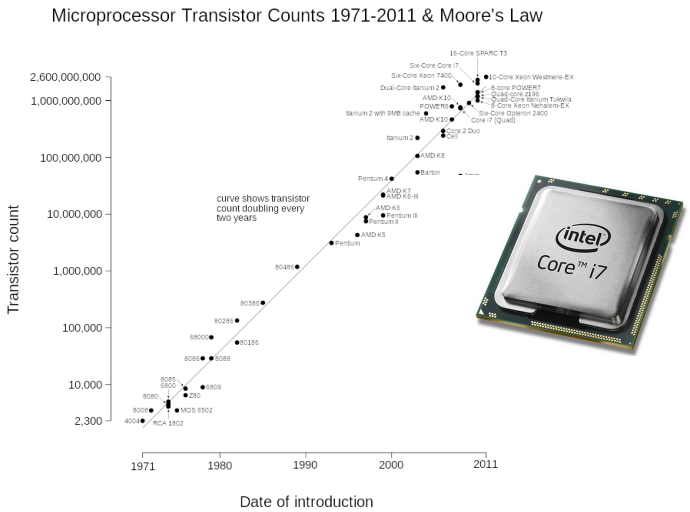
- How much data to get to super-human performances?
  - Rule of thumb: $n=d/\varepsilon$, d number of data features, ε is the accuracy
  - Example: ImageNet $d=256^2 \approx 1M$ and $\varepsilon \approx 1 \Rightarrow n=1.4M$ and 16% error (human error is 5%)
- Collecting data is easy but labeling data is challenging:
  - Labeling is time consuming (each data to a class).
  - It requires humans (learning ability bounded by human intelligence and bias).
  - Some important data is not accessible (e.g. medical data, nuclear meltdown).
- Deep learning is designed for labeled data (supervised learning). It is greedy in (labeled) data.
- Next AI revolution : Deep learning with unlabeled data, s.a. self-supervised learning.

Rise of computational power

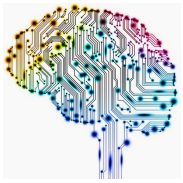
- Moore's law : 100x faster every decade
  - CPU, GPU, TPU, Cloud computing

1st GPU in 1999

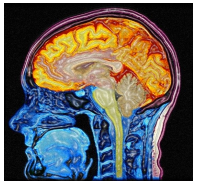
Amazon cloud in 2006



- Brain has 10^{14} synapses/10^3 Tflops = Computers in 2030 (100 times faster than today)

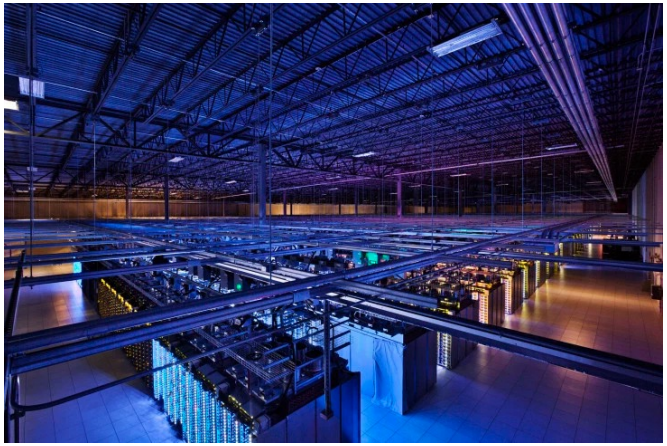


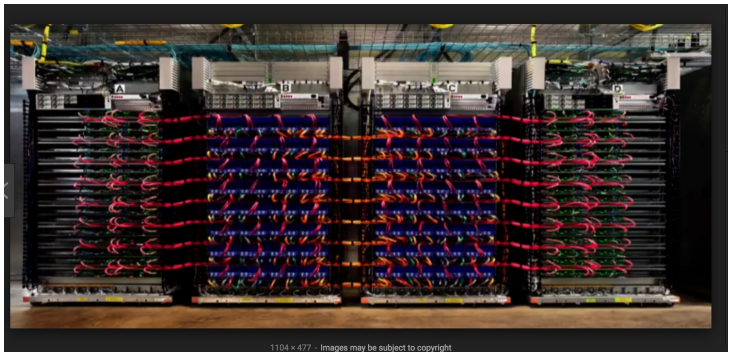
2030
=



Deep learning infrastructure

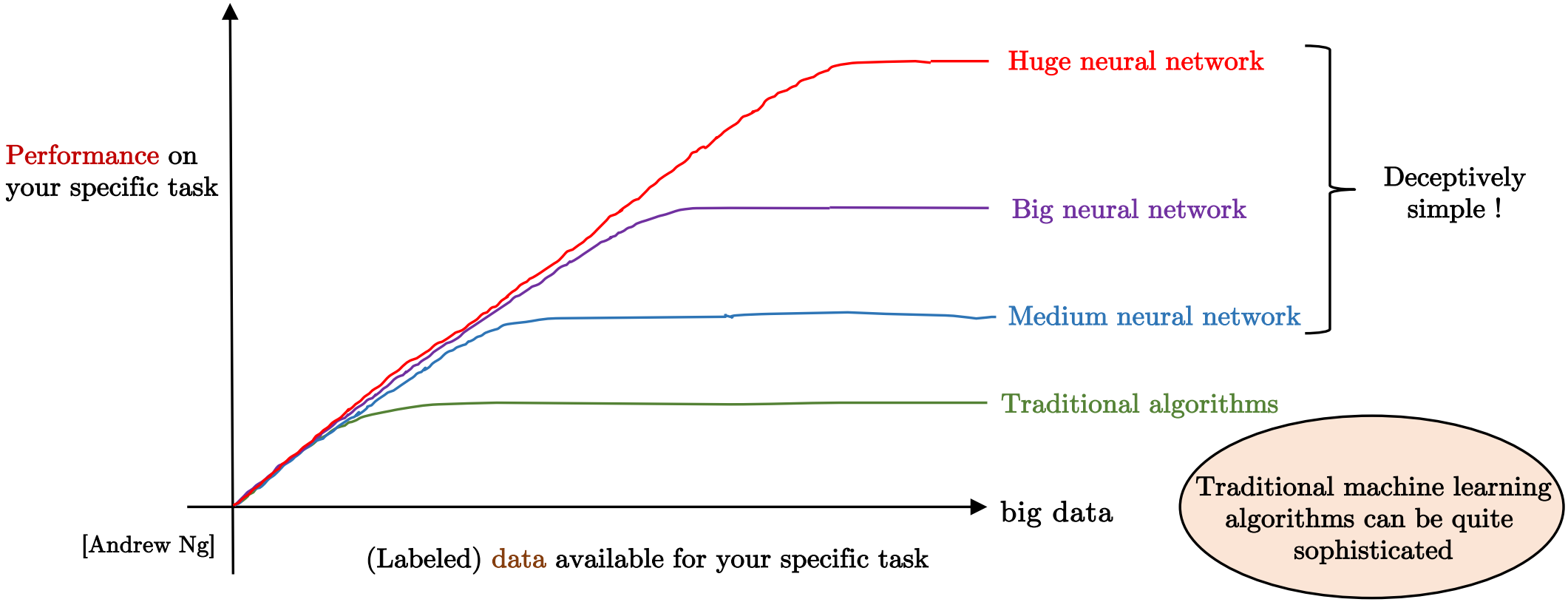
- Google Tensor Processing Unit (TPU) :
  - 2013: Google projects that they would need to double their datacenter to meet demand for speech recognition.
  - The TPU was designed, verified, built and deployed in data centers in just 15 months.





Machine learning algorithms

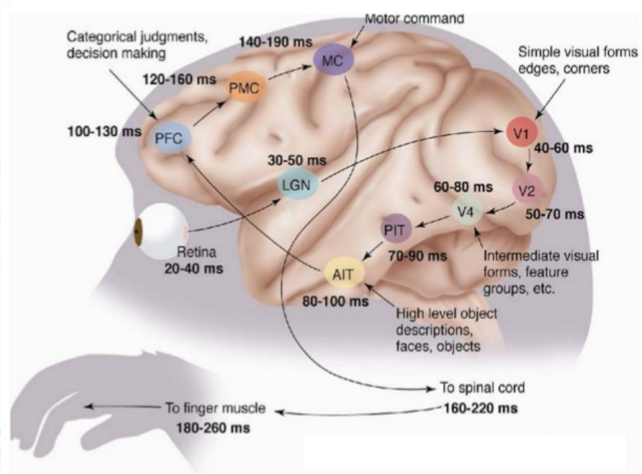
- Only deep learning systems are able to absorb (exploding) big data :



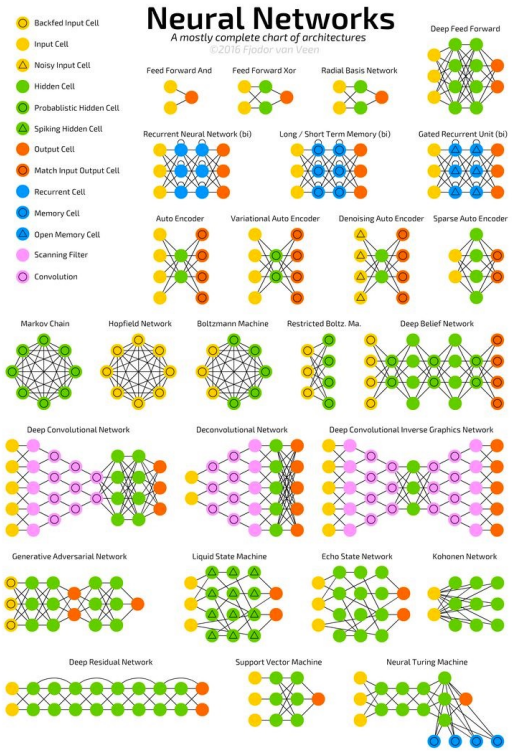
[Andrew Ng]

Deep learning architectures

- Most research works focus on the best architecture that solves a specific task (there is no DL model that can solve several tasks like the human brain).



Biological neural network



Artificial neural networks

- Most successful architectures:
  - Convolutional Neural Networks: Computer Vision, Video
  - Recurrent Neural Networks/ Transformers: NLP, Machine translation, speech-to-text

G. Hinton

Y. LeCun

Y. Bengio

Y. Schimdhuber

Turin Award 2019

Outline

- AI/DL case studies
- A brief history of DL
- 2012 DL breakthrough
- DL is a universal learning technique
- Terminology AI/ML/DL
- Key ingredients of DL
- AI is the new electricity
- Limitations of AI

AI is the new electricity

- "AI is the new electricity" - Andrew Ng

  Electricity has changed the whole industry, sector after sector like transport, manufacturing, agriculture, etc ⇒ AI has the same potential.

- Industrial revolutions :
  - 1st industrial revolution (machine revolution) [1760-1840]: From hand-tools to first machines (agriculture)
  - 2nd industrial revolution (electrical revolution) [1870-1914]: From mechanical machines to electricity
  - 3rd industrial revolution (digital revolution) [1947-2012]: From analogue electronic technology to digital (internet, smartphones)
  - 4th industrial revolution (smart digital revolution) [2012-]: From digital data to augmented intelligence

Outline

- AI/DL case studies
- A brief history of DL
- 2012 DL breakthrough
- DL is a universal learning technique
- Terminology AI/ML/DL
- Key ingredients of DL
- AI is the new electricity
- **Limitations of AI**

Current state of AI

- Modern AI breakthrough started in 2012 and there is no coming back – AI will be part of any machine systems.
- AI has a few impressive results but AI is in its infancy – immature engineering field (solutions are empirical, not robust, not scalable, not transferable).
- It will take (at least) a decade to develop a solid software, a hardware environment and a mathematical framework (statistics, optimization, etc) for AI.
- AI is a buzzword – Machine learning and data science are better names.
- Success is limited to perceptual tasks (vision, hearing) – no breakthrough results in cognitive tasks (reasoning, planning, common sense).

AI vs IA

- AI (Artificial Intelligence):
  - Terminator, Hollywood movies – Machines take over the world.
  - No sense
  - Definition of intelligence is unclear.

M. Jordan

- IA (Intelligence Augmentation):
  - Lots of sense – under development (will have a huge impact in our lives)
  - Better search engines, recommendation systems, natural language translation, Q&A's.
  - Augment people skills.

What is possible and not yet possible?

- Computer vision
  - Possible: Labeling of objects in visual scenes, predict what happens next
  - Not yet possible: Common-sense understanding of visual scenes
- Speech analysis
  - Possible: Speech-to-text and text-to-speech in many languages
  - Not yet possible: Common-sense understanding of auditory scenes
- Natural language processing
  - Possible: Adequate translation and Q&A (chatbot), code/math understanding
  - Not yet possible: Human-level dialog (Turing test)
- Robotics
  - Possible: Industrial programmed robots
  - Not yet possible: Robots that interact with humans and are autonomous.

Conclusion

- Deep Learning is a breakthrough in Machine Learning and AI.
  - DL has become the standard toolkit for analyzing data.
  - DL is highly flexible and thus can be applied to a large variety of applications.
  - Large-scale training (linear complexity) with distributed computing.
  - Supervised, reinforcement and self-supervised training.
  - DL boosts business analytics and revenues.
  - DL has been progressing fast, but unknown about its limits.

Course lectures

- Lecture 01 : Introduction
- Lecture 02 : PyTorch
- Lecture 03 : Vanilla Neural Networks
- Lecture 04 : Multi-Layer Perceptrons
- Lecture 05 : Convolutional Neural Networks
- Lecture 06 : Recurrent Neural Networks
- Lecture 07 : Attention Neural Networks
- Lecture 08 : Diffusion Models
- Lecture 09 : Reinforcement Learning
- Lecture 10 : Training Large Language Models

Questions?

CS5242 : Neural Networks and Deep Learning

Lecture 2 : PyTorch

Semester 2 2024/25

Xavier Bresson

https://twitter.com/xbresson

Department of Computer Science
National University of Singapore (NUS)

Outline

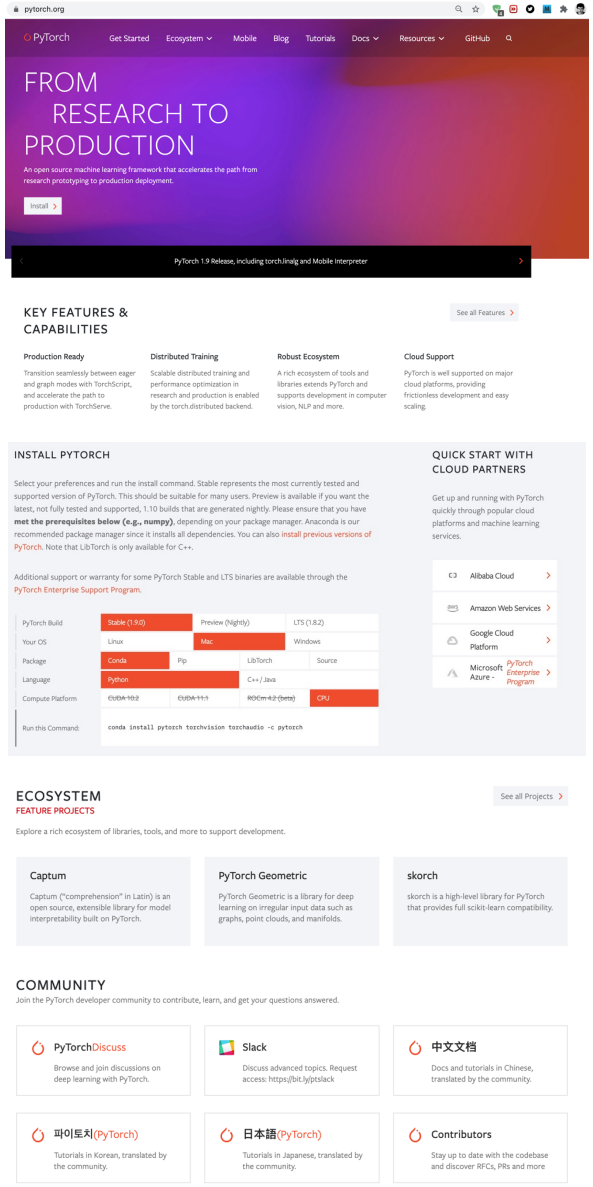
- PyTorch
- PyTorch for Coding Exercises
  - Google Colab
  - Local Installation
- PyTorch Labs

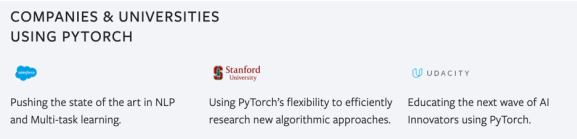
Outline

- **PyTorch**
- PyTorch for Coding Exercises
  - Google Colab
  - Local Installation
- PyTorch Labs

PyTorch

- PyTorch is an open source deep learning library in Python.
- Developed by FAIR (Facebook Artificial Intelligence Research).
- Original authors : Soumith Chintala, Adam Paszke, Sam Gross, Gregory Chanan.
- Initial release: October 2016
- PyTorch provides two high-level features :
  - Tensor computation (like numpy) with strong GPU acceleration
  - Deep Neural Networks built on an autodiff system
- https://pytorch.org





Outline

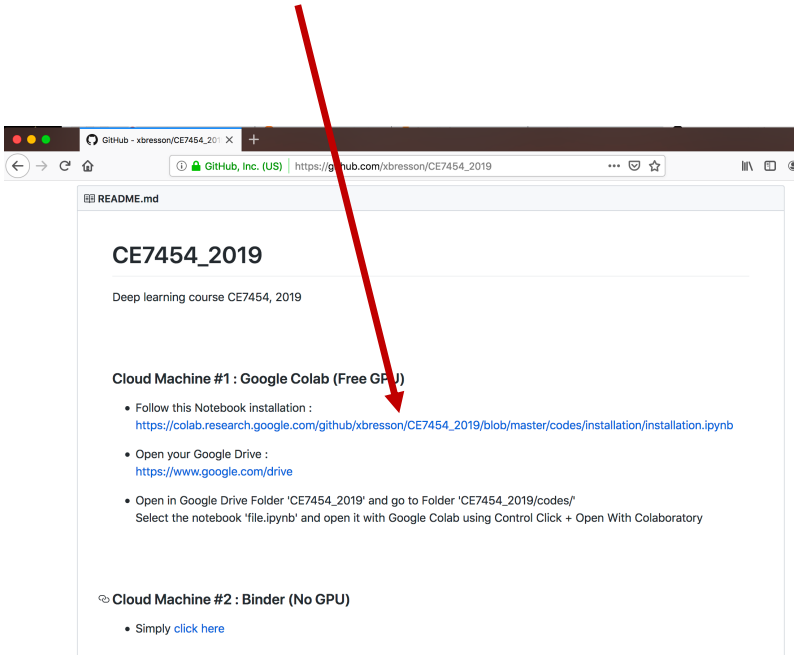
- PyTorch
- PyTorch for Coding Exercises
  - Google Colab
  - Local Installation
- PyTorch Labs

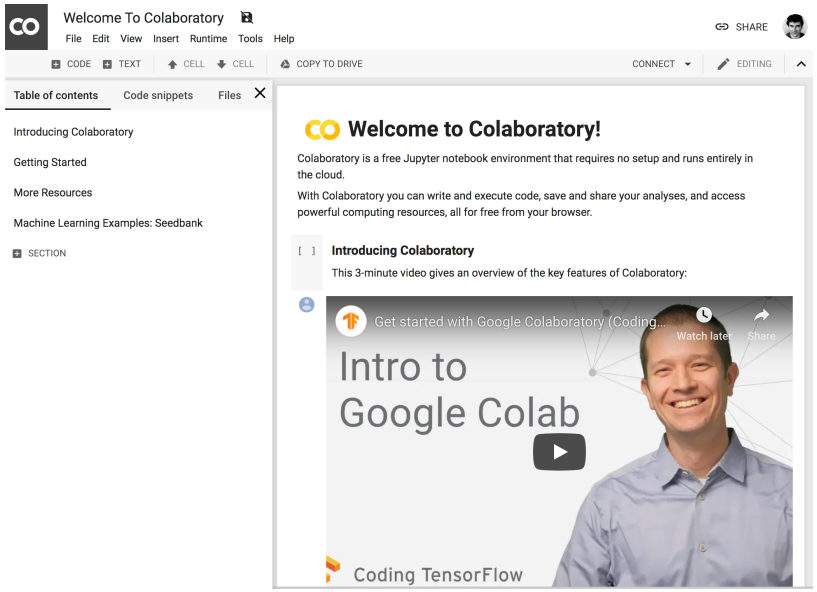
Google Colab

- Follow these instructions :
  - Go to the GitHub folder of the course :

    https://github.com/xbresson/CS5242_2025

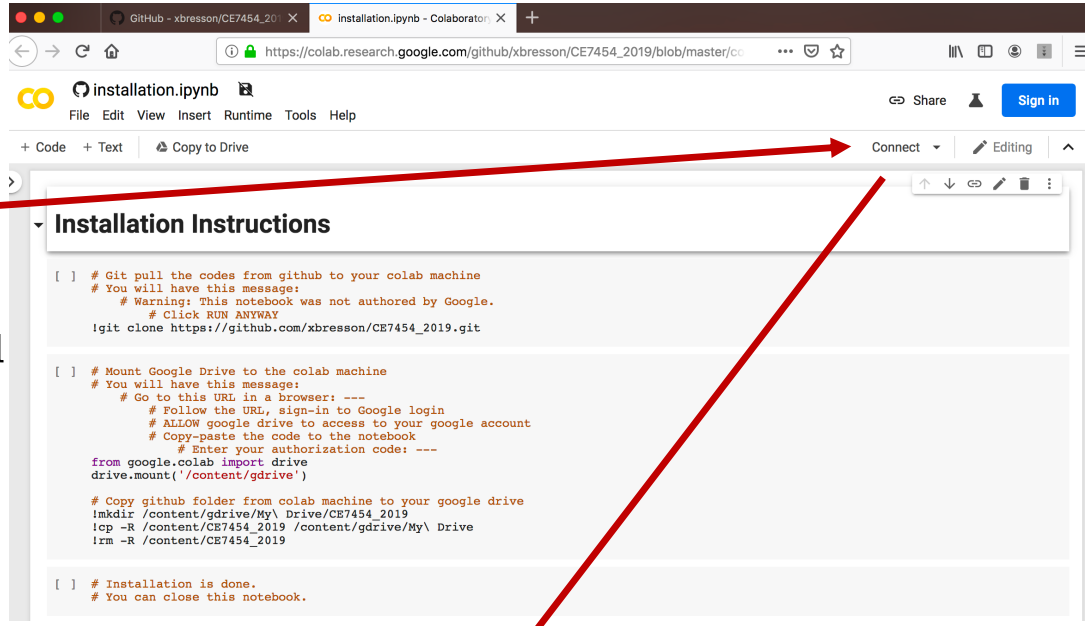
    Click on this link.

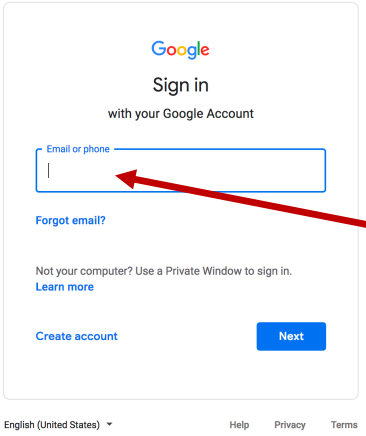


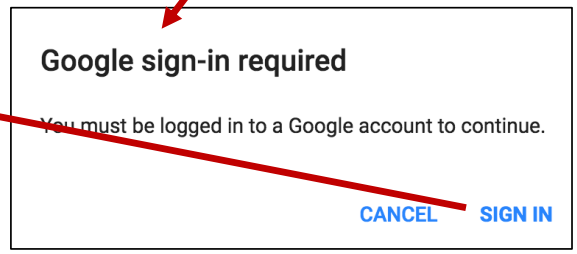


Google Colab

- Click on CONNECT.
- It will ask you to sign-in with your Gmail account.



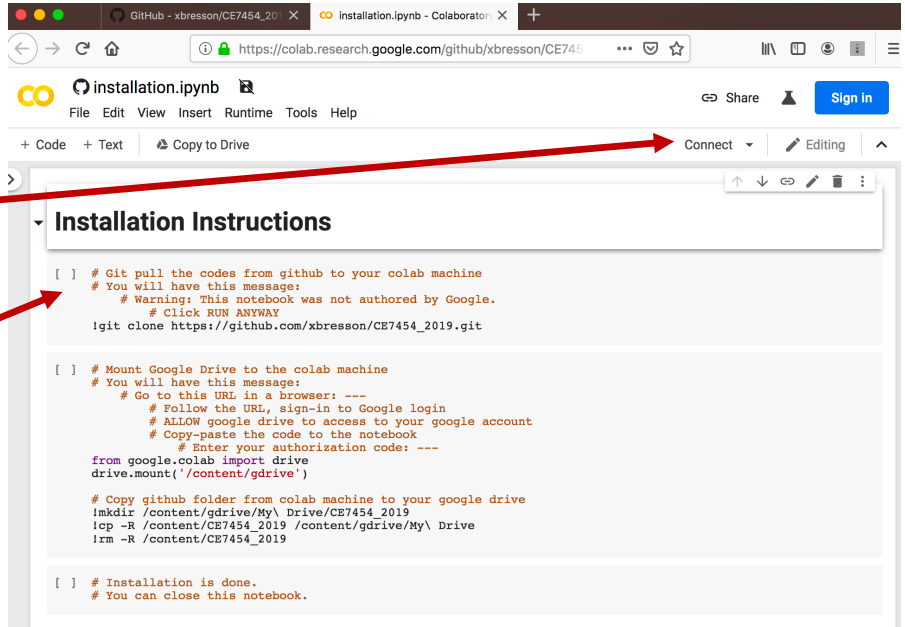


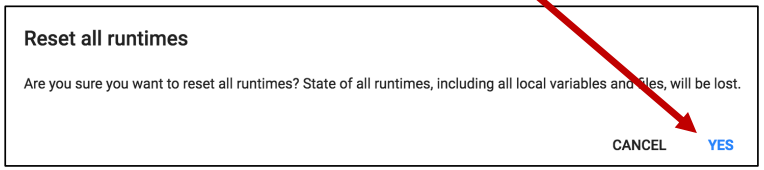


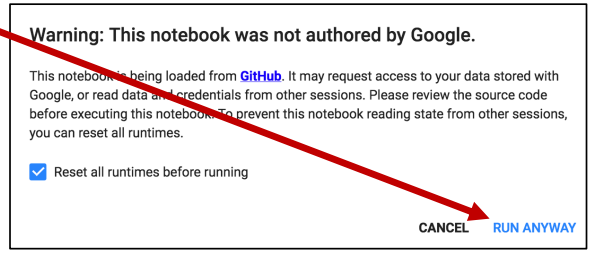
Google Colab

- Click on CONNECT again to start the Google Cloud machine.

- Run the first cell to clone the codes from GitHub to the Google Cloud machine.
  - It will give a warning, click on RUN ANYWAY.
  - Answer YES to the next question RESET ALL RUNTIMES.

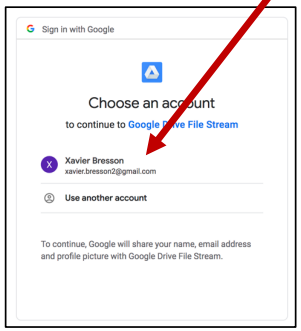


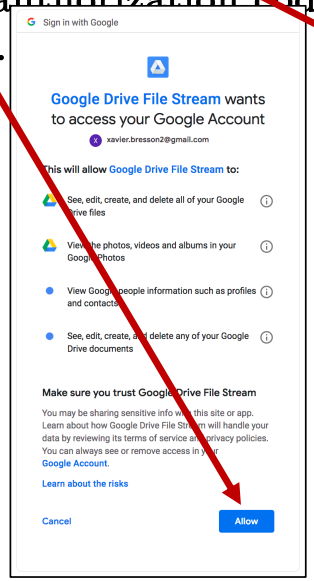


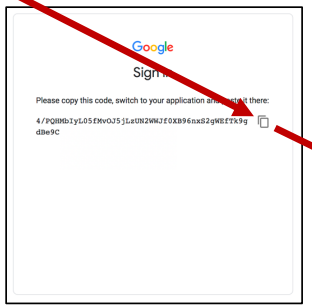


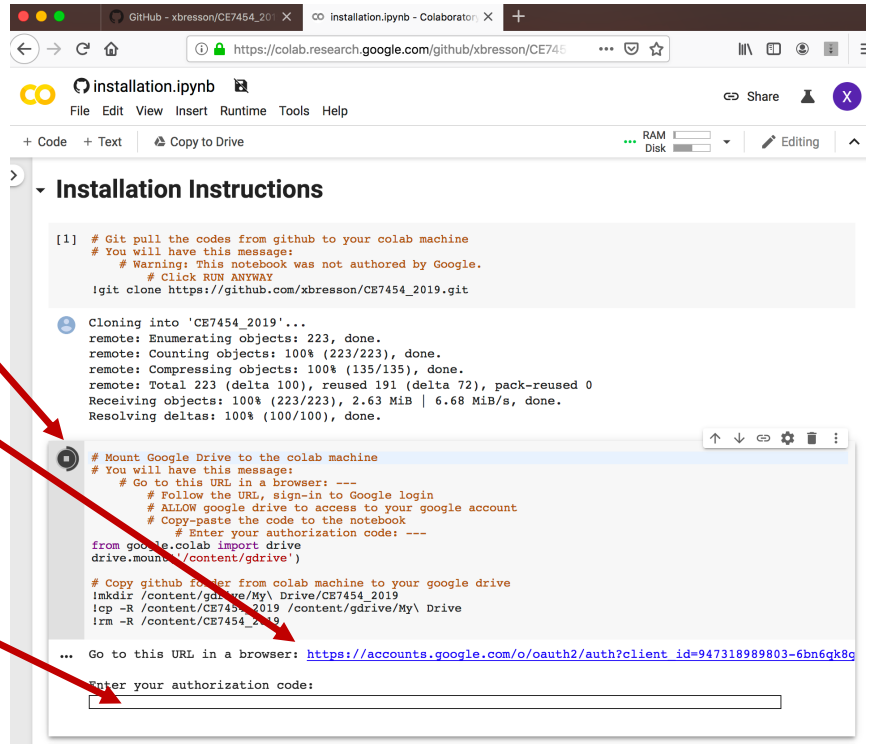
Google Colab

- Run the second cell to mount your Google Drive to the Google Cloud machine (all your codes will be saved in Google Drive).
  - Click on the provided URL.
  - Select your Gmail account.
  - ALLOW Google Drive File Stream.
  - Copy-paste the code to the notebook (Enter your authorization code) and press Return.



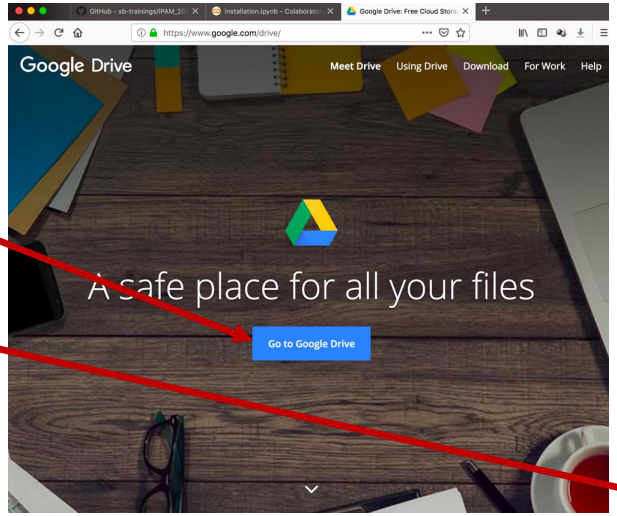


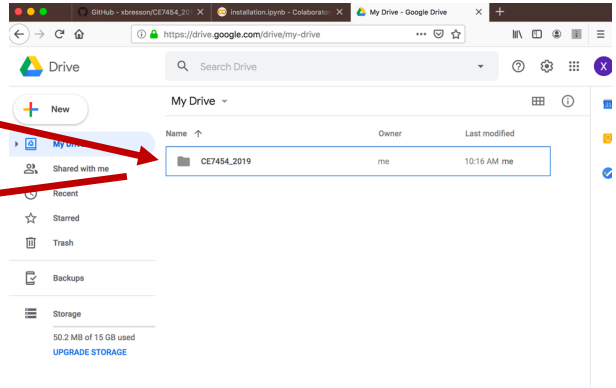


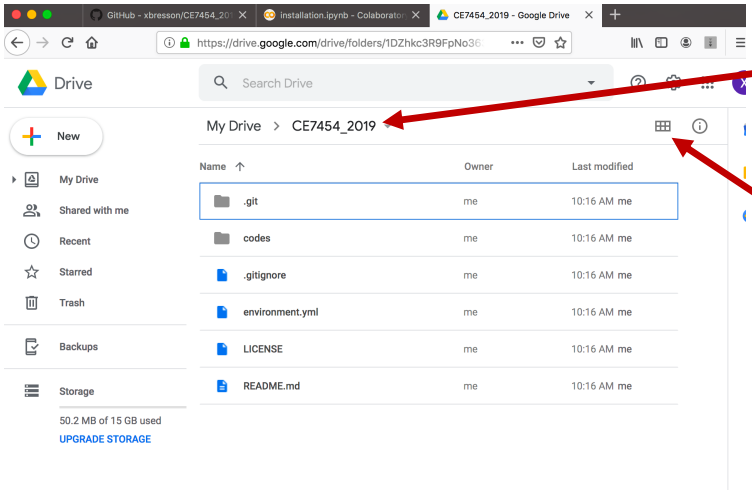


Google Colab

- Open your Google Drive :
  https://www.google.com/drive
- Go folder :
  - CS5242_2025_codes



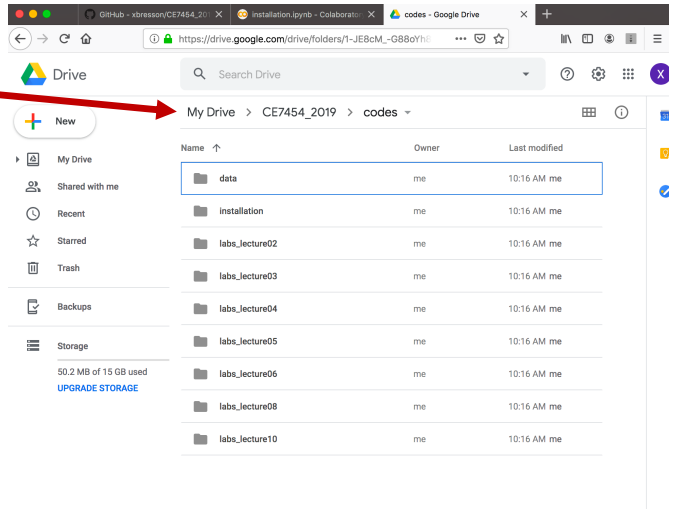


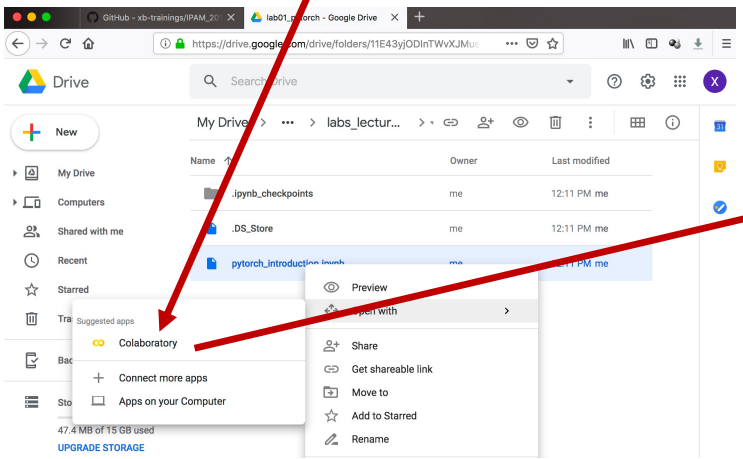


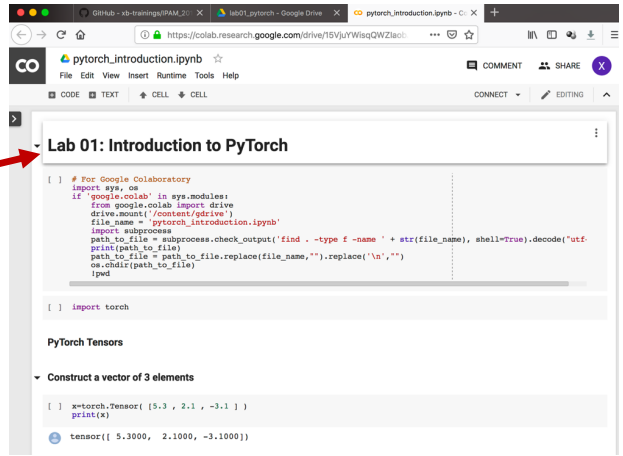
Click here for List View

Google Colab

- Go folder CS5242_2025_codes/
- Let us open the notebook pytorch_introduction.pynb in folder codes/labs_lecture02.
  - Select the notebook and open it using Control Click + Open With Colaboratory

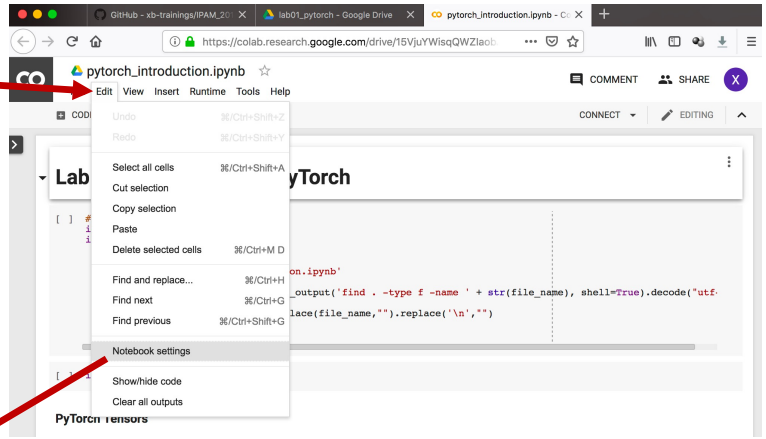


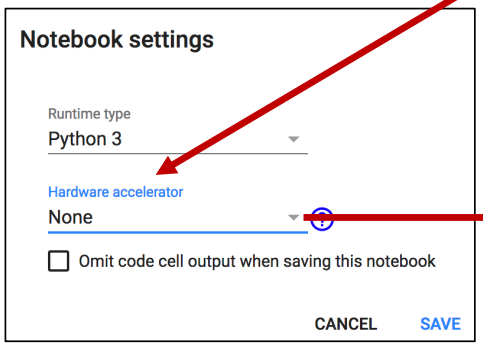


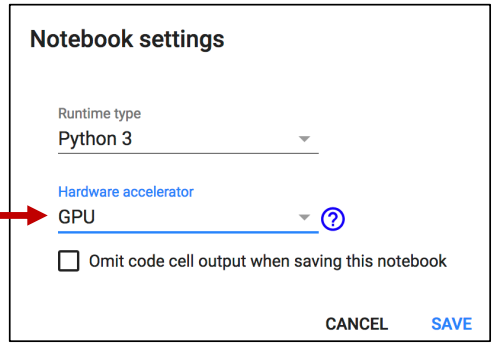


Google Colab

- GPU accelerator :
  - Select Edit in the menu and Notebook settings.
  - Select GPU in Hardware accelerator.







Outline

- PyTorch
- PyTorch for Coding Exercises
  - Google Colab
  - Local Installation
- PyTorch Labs

Local Installation

- Install Python and run the notebooks on your machine :

Local Installation for OSX & Linux

- Open a Terminal and type

```
# Conda installation
curl https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh -o miniconda.sh -J -L
curl https://repo.continuum.io/miniconda/Miniconda3-latest-MacOSX-x86_64.sh -o miniconda.sh -J -l
chmod +x miniconda.sh
./miniconda.sh
source ~/.bashrc

# Clone GitHub repo
git clone https://github.com/xb-trainings/ACE_NLP2022.git
cd ACE_NLP2022

# Install python libraries
conda env create -f environment.yml
source activate deeplearn_course

# Run the notebooks
jupyter notebook
```

https://github.com/xbresson/CS5242_2025#local-installation-for-osx--linux

Local Installation

CS5242_2021 — miniconda.sh — 155×44

```
Last login: Mon Aug  9 15:02:37 on ttys001
(base) xbresson@r-153-121-25-172 CS5242_2021 % curl https://repo.continuum.io/miniconda/Miniconda3-latest-MacOSX-x86_64.sh -o miniconda.sh -J -L -k
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100 42.3M  100 42.3M    0     0  23.2M      0  0:00:01  0:00:01 --:--:-- 30.5M
(base) xbresson@r-153-121-25-172 CS5242_2021 % chmod +x miniconda.sh
(base) xbresson@r-153-121-25-172 CS5242_2021 % ./miniconda.sh

Welcome to Miniconda3 py39_4.10.3

In order to continue the installation process, please review the license
agreement.
Please, press ENTER to continue
>>>
```

```
  wheel              pkgs/main/noarch::wheel-0.36.2-pyhd3eb1b0_0
  xz                 pkgs/main/osx-64::xz-5.2.5-h1de35cc_0
  yaml               pkgs/main/osx-64::yaml-0.2.5-haf1e3a3_0
  zlib               pkgs/main/osx-64::zlib-1.2.11-h1de35cc_3


Preparing transaction: done
Executing transaction: \
done
installation finished.
Do you wish the installer to initialize Miniconda3
by running conda init? [yes|no]
[yes] >>>
no change     /Users/xbresson/miniconda3/condabin/conda
no change     /Users/xbresson/miniconda3/bin/conda
no change     /Users/xbresson/miniconda3/bin/conda-env
no change     /Users/xbresson/miniconda3/bin/activate
no change     /Users/xbresson/miniconda3/bin/deactivate
no change     /Users/xbresson/miniconda3/etc/profile.d/conda.sh
no change     /Users/xbresson/miniconda3/etc/fish/conf.d/conda.fish
no change     /Users/xbresson/miniconda3/shell/condabin/Conda.psm1
no change     /Users/xbresson/miniconda3/shell/condabin/conda-hook.ps1
no change     /Users/xbresson/miniconda3/lib/python3.9/site-packages/xontrib/conda.xsh
no change     /Users/xbresson/miniconda3/etc/profile.d/conda.csh
no change     /Users/xbresson/.zshrc
No action taken.
If you'd prefer that conda's base environment not be activated on startup,
   set the auto_activate_base parameter to false:

conda config --set auto_activate_base false

Thank you for installing Miniconda3!
(base) xbresson@r-153-121-25-172 CS5242_2021 %
```

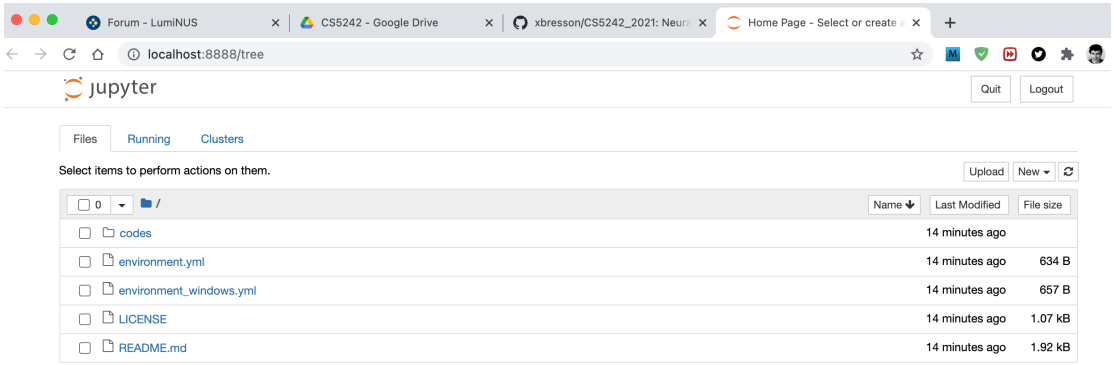
Local Installation

```
(base) xbresson@r-153-121-25-172 CS5242_2021 % conda
usage: conda [-h] [-V] command ...

conda is a tool for managing and deploying applications, environments and packages.

Options:

positional arguments:
  command
    clean        Remove unused packages and caches.
    compare      Compare packages between conda environments.
    config       Modify configuration values in .condarc. This is modeled after the git config command. Writes to the user .condarc file
                 (/Users/xbresson/.condarc) by default.
    create       Create a new conda environment from a list of specified packages.
    help         Displays a list of available conda commands and their help strings.
    info         Display information about current conda install.
    init         Initialize conda for shell interaction. [Experimental]
    install      Installs a list of packages into a specified conda environment.
    list         List linked packages in a conda environment.
    package      Low-level conda package utility. (EXPERIMENTAL)
    remove       Remove a list of packages from a specified conda environment.
    uninstall    Alias for conda remove.
    run          Run an executable in a conda environment. [Experimental]
    search       Search for packages and display associated information. The input is a MatchSpec, a query language for conda packages. See examples
                 below.
    update       Updates conda packages to the latest compatible version.
    upgrade      Alias for conda update.

optional arguments:
  -h, --help     Show this help message and exit.
  -V, --version  Show the conda version number and exit.

conda commands available from other packages:
  env
(base) xbresson@r-153-121-25-172 CS5242_2021 %
```

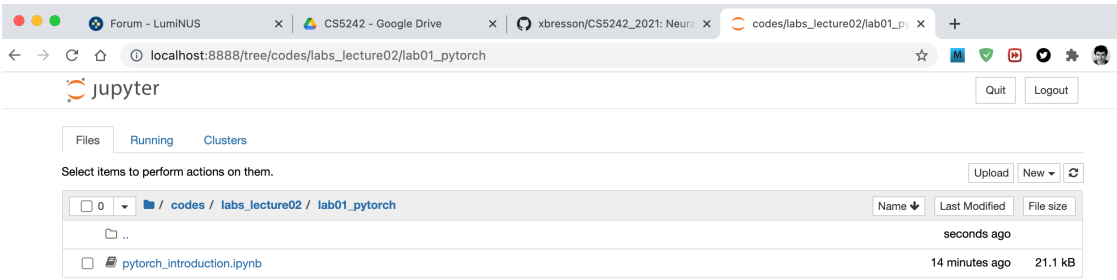
```
(base) xbresson@r-153-121-25-172 tmp % git clone https://github.com/xbresson/CS5242_2021.git
Cloning into 'CS5242_2021'...
remote: Enumerating objects: 201, done.
remote: Counting objects: 100% (201/201), done.
remote: Compressing objects: 100% (117/117), done.
remote: Total 201 (delta 82), reused 187 (delta 71), pack-reused 0
Receiving objects: 100% (201/201), 2.82 MiB | 6.62 MiB/s, done.
Resolving deltas: 100% (82/82), done.
(base) xbresson@r-153-121-25-172 tmp % cd CS5242_2021
(base) xbresson@r-153-121-25-172 CS5242_2021 % conda env create -f environment.yml
```

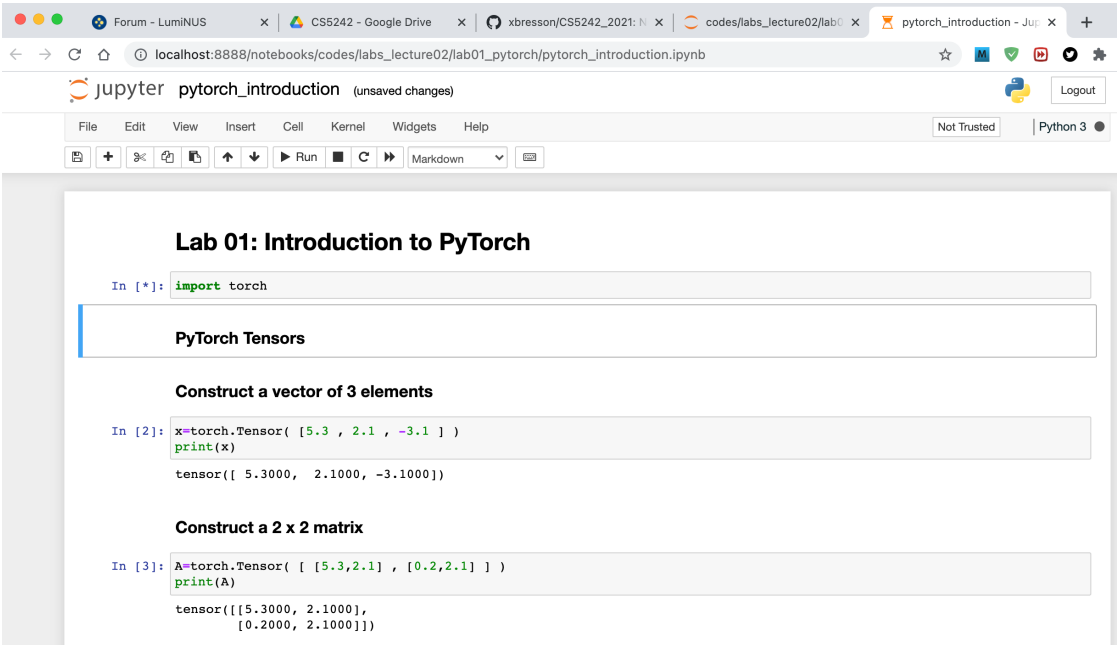
Local Installation

```
Collecting protobuf>=3.8.0
  Using cached protobuf-3.17.3-cp37-cp37m-macosx_10_9_x86_64.whl (1.0 MB)
Requirement already satisfied: six>=1.9 in /Users/xbresson/miniconda3/envs/deeplearn_course/lib/python3.7/site-packages (from protobuf>=3.8.0->tensorboardx
==2.2->-r /Users/xbresson/Documents/Dropbox/10_NUS_2021_now/05_My_Teaching/06_CS5242_sem1_21:22/04_Github/tmp/CS5242_2021/condaenv.1o03j884.requirements.tx
t (line 2)) (1.16.0)
Collecting future
  Using cached future-0.18.2-py3-none-any.whl
Installing collected packages: future, pyglet, protobuf, cloudpickle, tensorboardx, gym, fastprogress
Successfully installed cloudpickle-1.6.0 fastprogress-1.0.0 future-0.18.2 gym-0.18.0 protobuf-3.17.3 pyglet-1.5.0 tensorboardx-2.2

done
#
# To activate this environment, use
#
#     $ conda activate deeplearn_course
#
# To deactivate an active environment, use
#
#     $ conda deactivate

(base) xbresson@r-153-121-25-172 CS5242_2021 % conda activate deeplearn_course
(deeplearn_course) xbresson@r-153-121-25-172 CS5242_2021 % jupyter notebook
[I 12:18:06.635 NotebookApp] Writing notebook server cookie secret to /Users/xbresson/Library/Jupyter/runtime/notebook_cookie_secret
[I 12:18:08.268 NotebookApp] Serving notebooks from local directory: /Users/xbresson/Documents/Dropbox/10_NUS_2021_now/05_My_Teaching/06_CS5242_sem1_21:22/
04_Github/tmp/CS5242_2021
[I 12:18:08.268 NotebookApp] Jupyter Notebook 6.4.2 is running at:
[I 12:18:08.268 NotebookApp] http://localhost:8888/?token=0e66862970f67d6ba62bcef6ffbe79d851bf26d95f292340
[I 12:18:08.268 NotebookApp]  or http://127.0.0.1:8888/?token=0e66862970f67d6ba62bcef6ffbe79d851bf26d95f292340
[I 12:18:08.268 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 12:18:08.275 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///Users/xbresson/Library/Jupyter/runtime/nbserver-87552-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=0e66862970f67d6ba62bcef6ffbe79d851bf26d95f292340
     or http://127.0.0.1:8888/?token=0e66862970f67d6ba62bcef6ffbe79d851bf26d95f292340
```



Local Installation





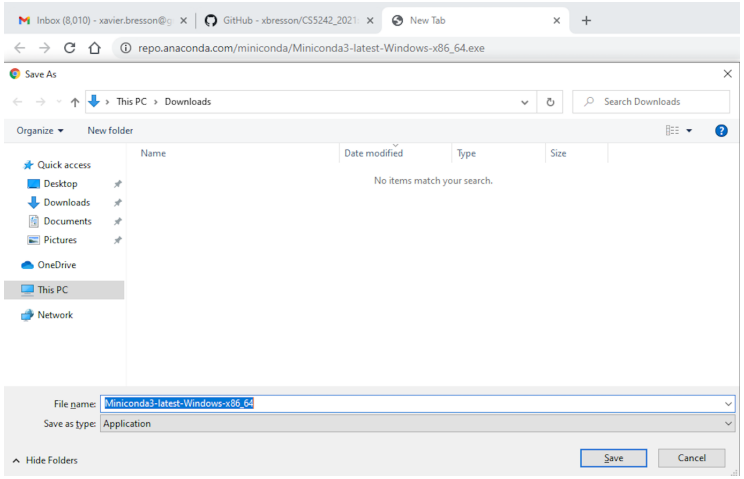
Local Installation

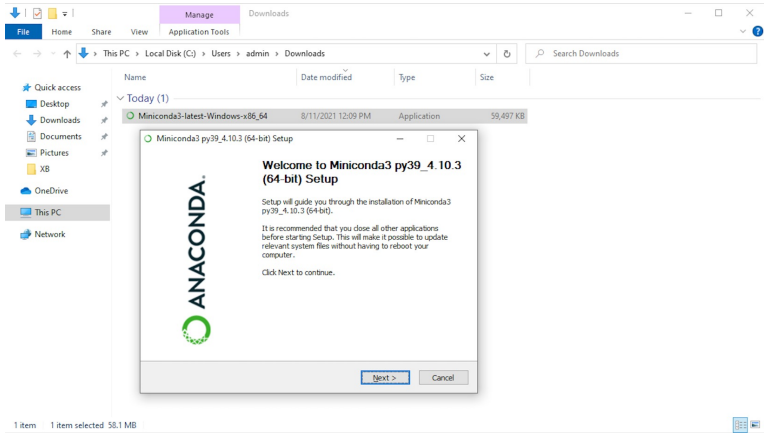
Local Installation for Windows

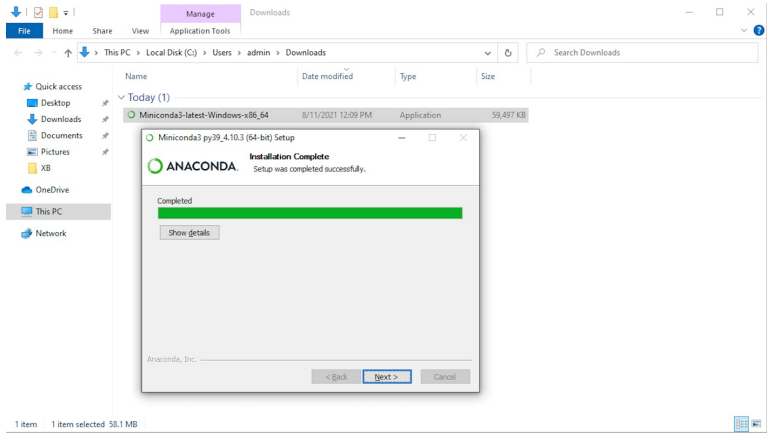
```
# Install Anaconda
https://repo.anaconda.com/miniconda/Miniconda3-latest-Windows-x86_64.exe

# Open an Anaconda Terminal
Go to Application => Anaconda3 => Anaconda Prompt

# Install git : Type in terminal
conda install git

# Clone GitHub repo
git clone https://github.com/xb-trainings/ACE_NLP2022.git
cd ACE_NLP2022

# Install python libraries
conda env create -f environment_windows.yml
conda activate deeplearn_course

# Run the notebooks
jupyter notebook
```

https://github.com/xbresson/CS5242_2025#local-installation-for-windows

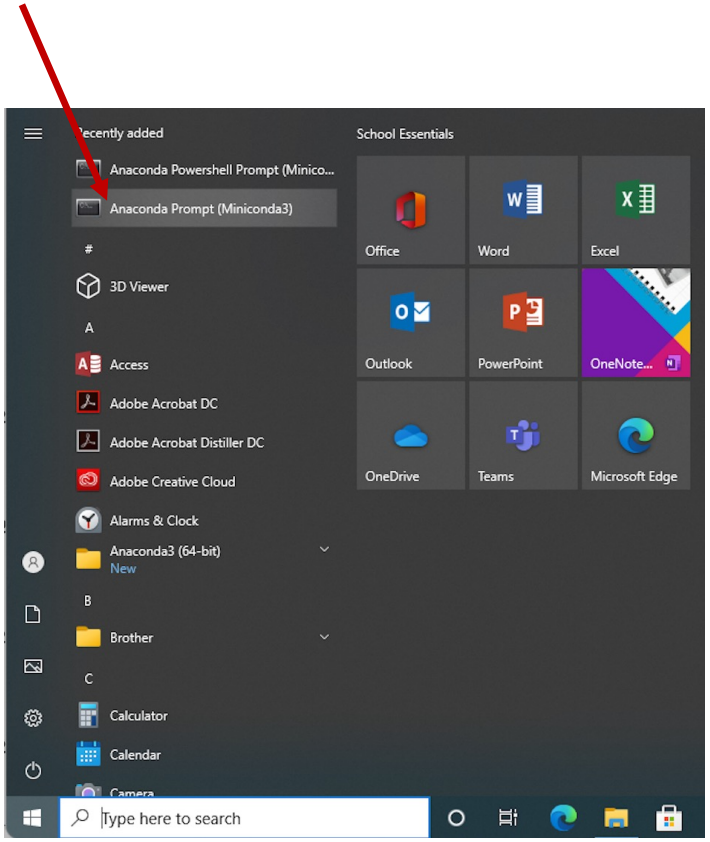
Local Installation

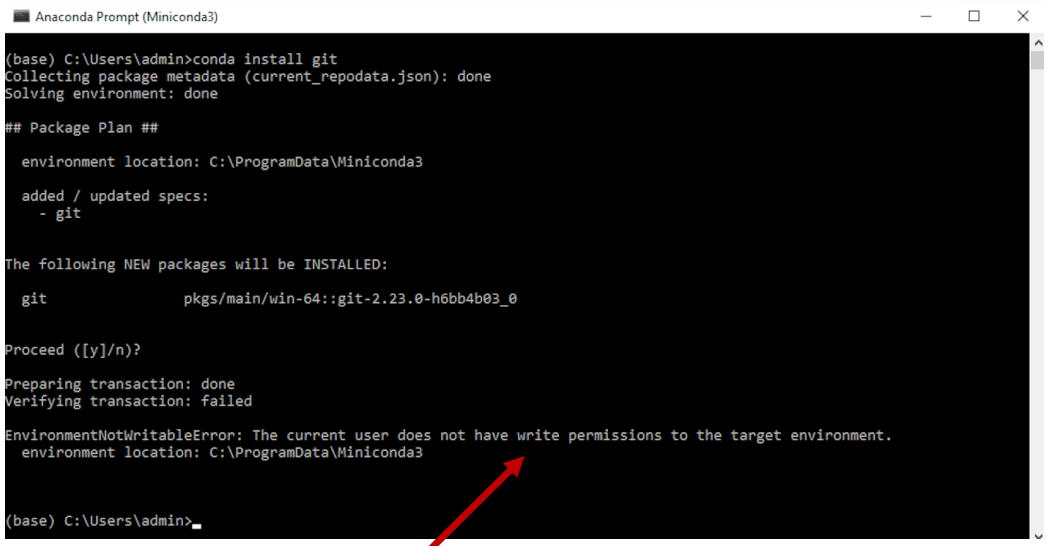






Local Installation

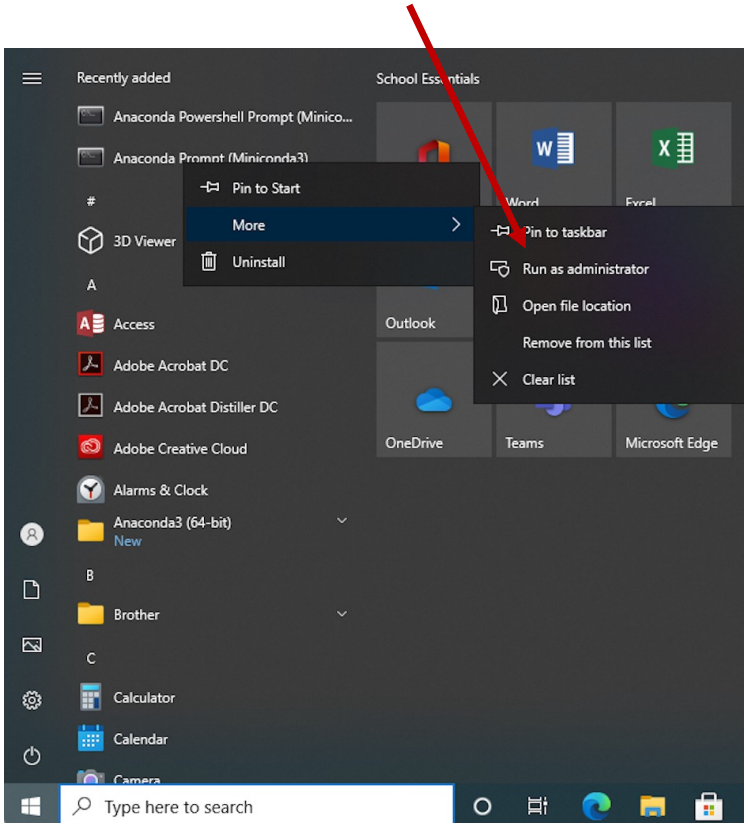


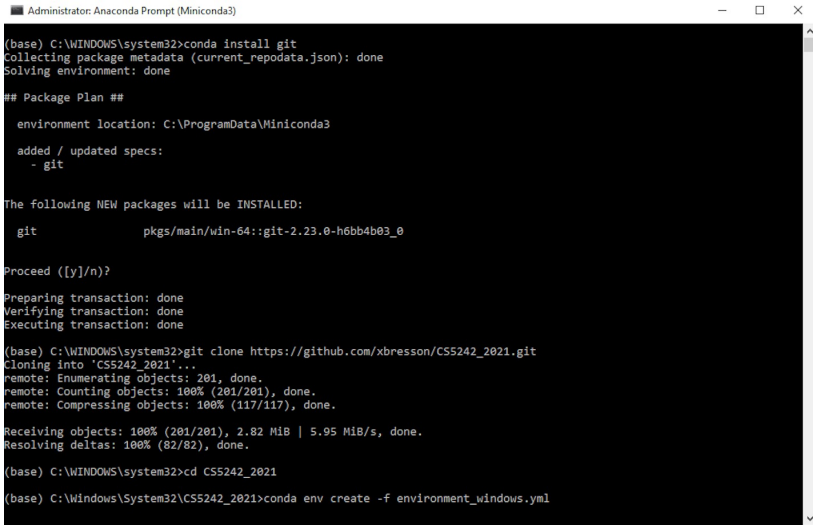


If you need admin rights to run Miniconda, then follow the next slide.

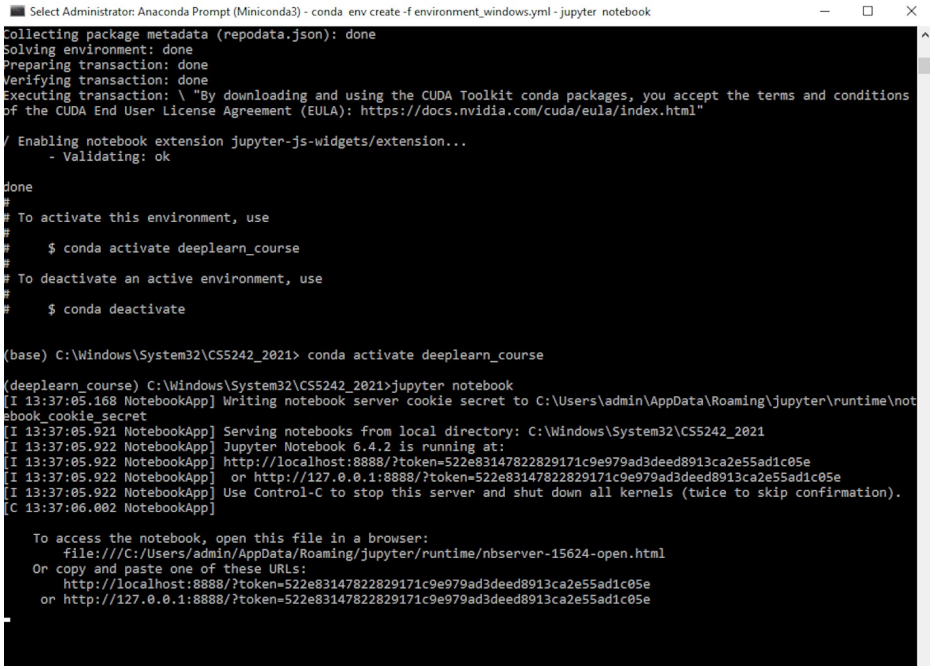
Local Installation

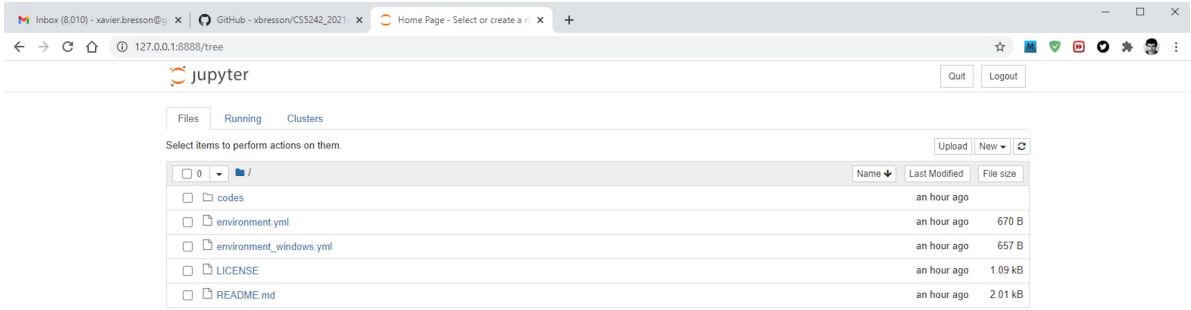
Secondary click then select “Run as administrator”.



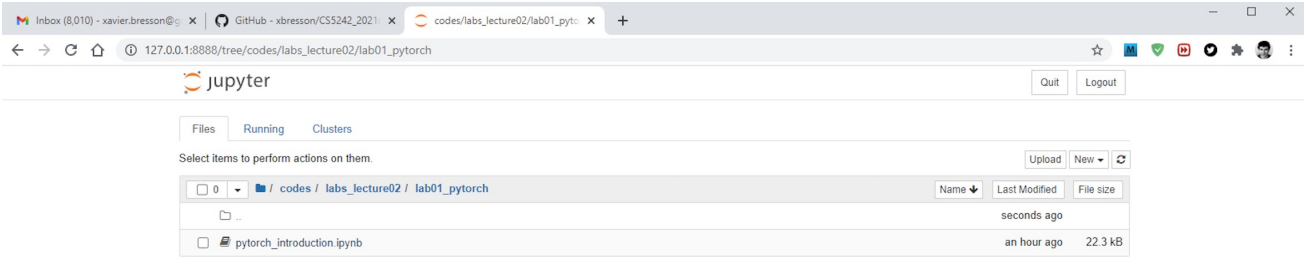


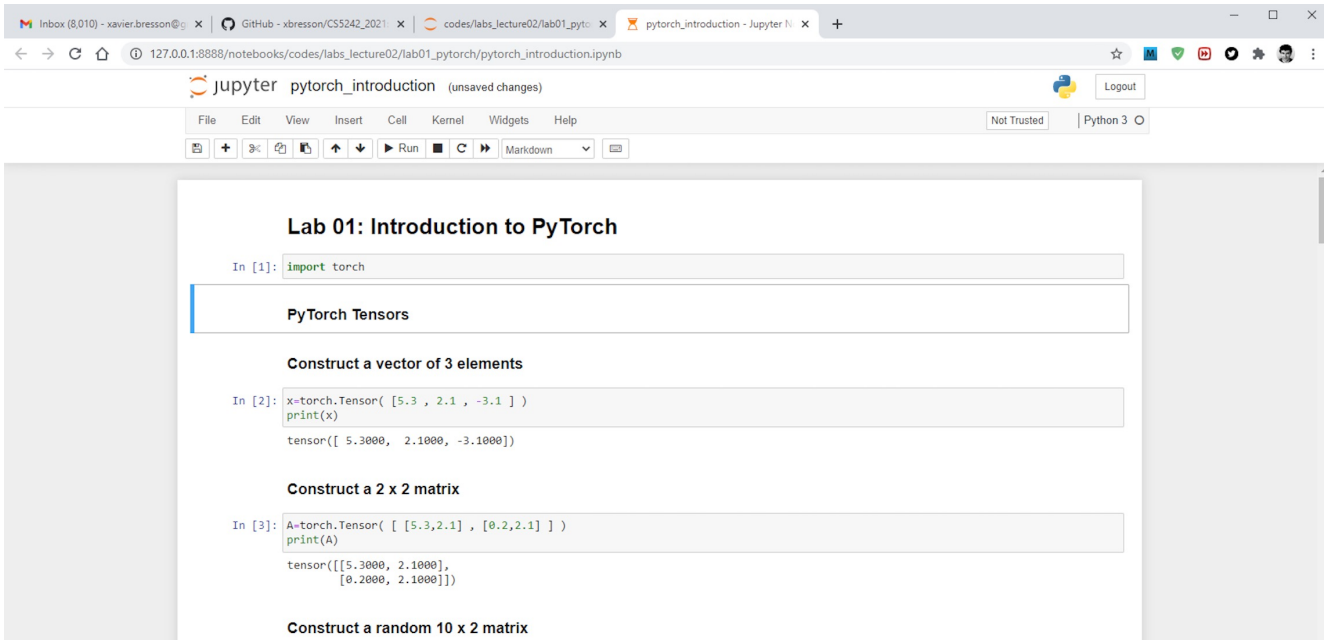
Local Installation





Local Installation



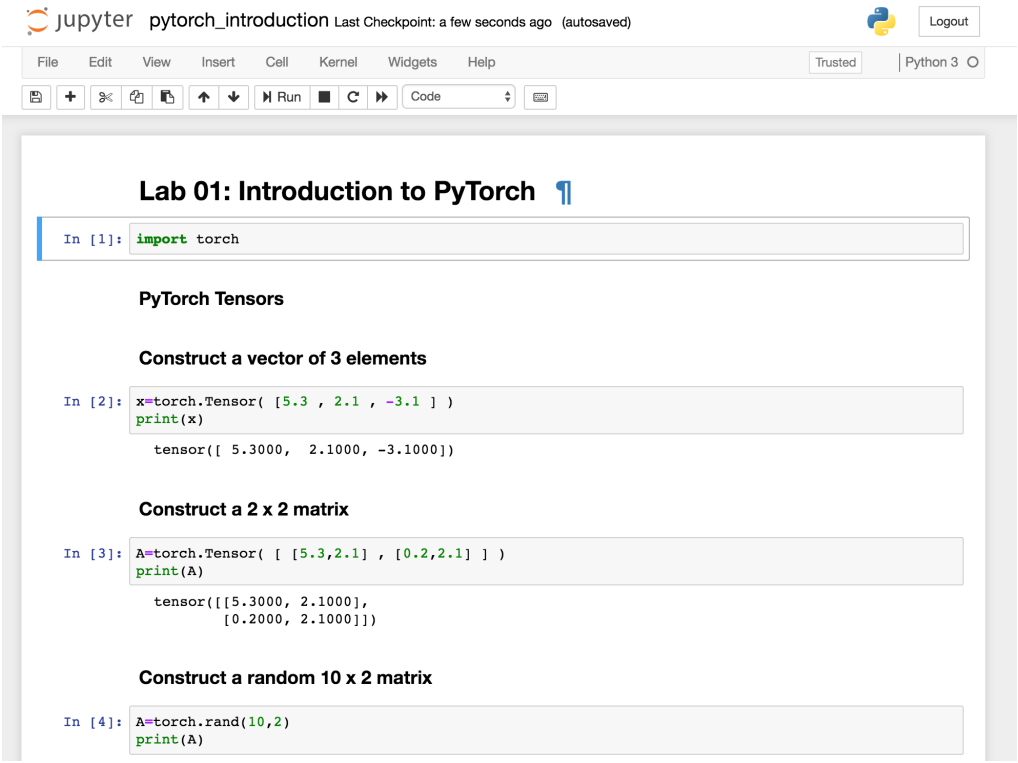


Outline

- PyTorch
- PyTorch for Coding Exercises
  - Google Colab
  - Local Installation
- **PyTorch Labs**

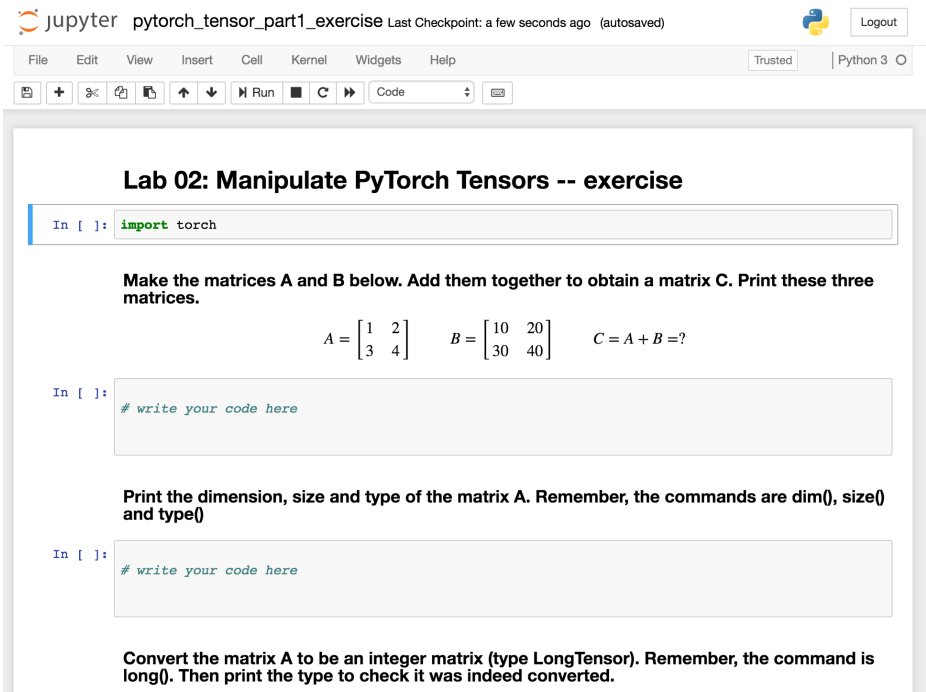
Lab 01

- Introduction to PyTorch



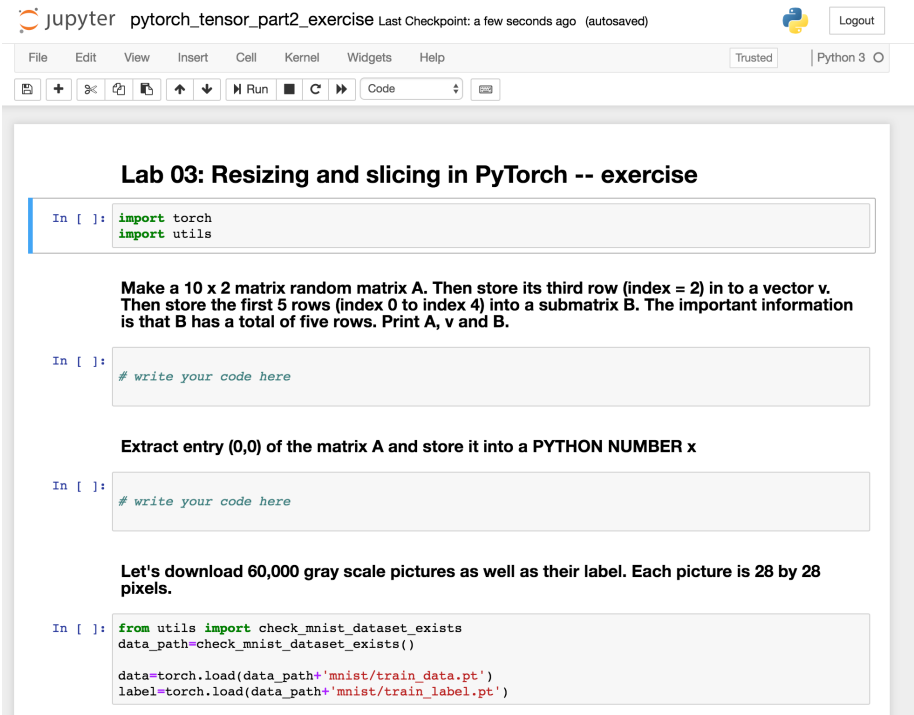
Lab 02

- Manipulating PyTorch tensors – basic linear algebra



Lab 03

- Manipulating PyTorch tensors – reshaping and slicing



jupyter pytorch_tensor_part2_exercise Last Checkpoint: a few seconds ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Lab 03: Resizing and slicing in PyTorch -- exercise

In [ ]:
```
import torch
import utils
```

Make a 10 x 2 matrix random matrix A. Then store its third row (index = 2) in to a vector v. Then store the first 5 rows (index 0 to index 4) into a submatrix B. The important information is that B has a total of five rows. Print A, v and B.

In [ ]:
```
# write your code here
```

Extract entry (0,0) of the matrix A and store it into a PYTHON NUMBER x

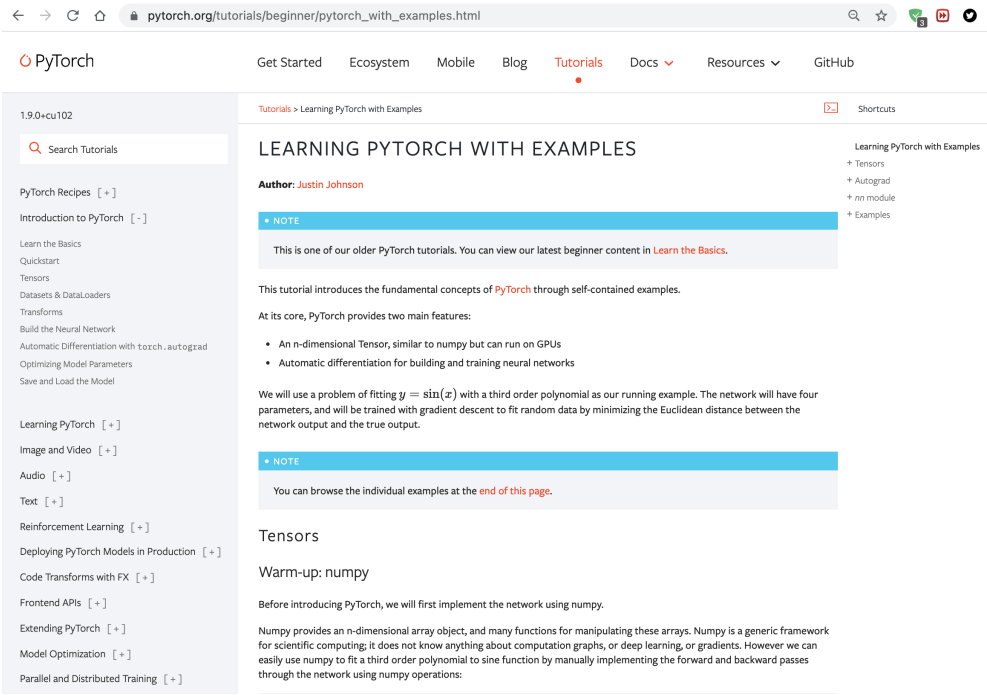
In [ ]:
```
# write your code here
```

Let's download 60,000 gray scale pictures as well as their label. Each picture is 28 by 28 pixels.

In [ ]:
```
from utils import check_mnist_dataset_exists
data_path=check_mnist_dataset_exists()

data=torch.load(data_path+'mnist/train_data.pt')
label=torch.load(data_path+'mnist/train_label.pt')
```

More Labs

- **Learning PyTorch with Examples**



https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

Questions?

CS5242 : Neural Networks and Deep Learning

Lecture 3 : Vanilla Neural Networks
Part 1 : Inference and Learning

Semester 2 2024/25

Xavier Bresson

https://twitter.com/xbresson

Department of Computer Science
National University of Singapore (NUS)

Outline

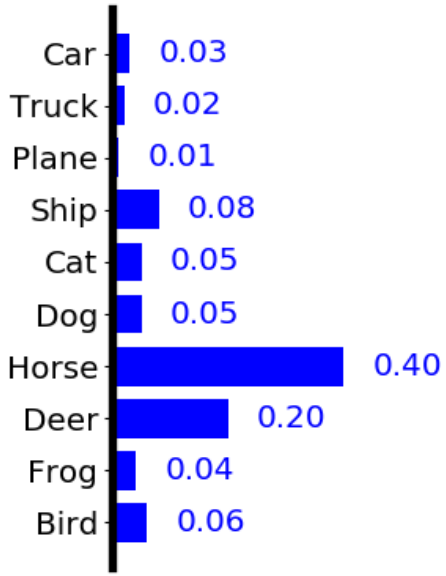
- Image classification task
- Neural networks for image classification
- Forward pass for inference
- Understanding linear layers
- Understanding the backward pass
- Matrix formulation
- Mini-batch learning

Outline

- Image classification task
- Neural networks for image classification
- Forward pass for inference
- Understanding linear layers
- Understanding the backward pass
- Matrix formulation
- Mini-batch learning

Classification task

Algorithm



Input

$3 \times 32 \times 32$ tensor

Output

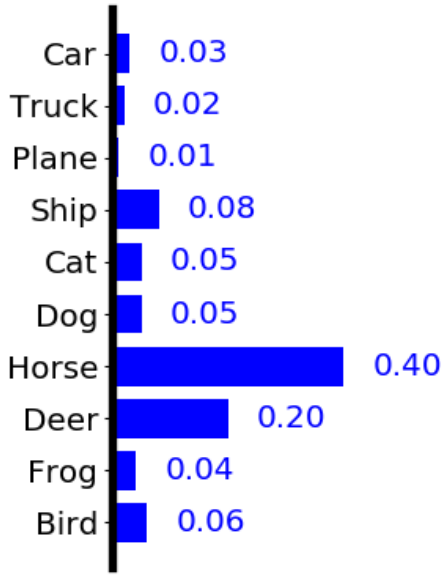
Probabilities over classes

Outline

- Image classification task
- **Neural networks for image classification**
- Forward pass for inference
- Understanding linear layers
- Understanding the backward pass
- Matrix formulation
- Mini-batch learning

Neural network as classification algorithm

Neural Net



Input

$3 \times 32 \times 32$ tensor

The classification task is solved by the forward pass, which consists in passing the input image through a series of linear and non-linear operations to obtain the solution.

Output

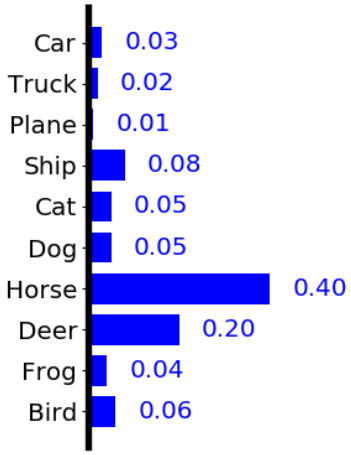
Probabilities over classes

Outline

- Image classification task
- Neural networks for image classification
- **Forward pass for inference**
- Understanding linear layers
- Understanding the backward pass
- Matrix formulation
- Mini-batch learning

Forward pass

- Inference/prediction:
  - How to take 3072 numbers and convert them into 10 positive numbers that sum to one (probability function)?



Input

$3 \times 32 \times 32 = 3072$ numbers

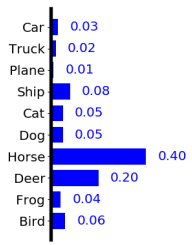
Output

10 positive numbers that sum to one

Simplest possible transformations

- Inference is done by a linear layer (LL) + a (non-linear) softmax layer.
  - This is the simplest way to take 3072 numbers and to convert them into 10 positive numbers that sum to one (probability function).

$$\begin{bmatrix} .14 \\ .72 \\ .54 \\ .82 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ .75 \end{bmatrix} \xrightarrow{\text{Linear Transformation}} \begin{bmatrix} -1.6 \\ -0.9 \\ -1.1 \\ -2.5 \\ 0.1 \\ 0.5 \\ 1.7 \\ 1.2 \\ -0.3 \\ 0.2 \end{bmatrix} \xrightarrow{\text{Softmax Function}} \begin{bmatrix} .01 \\ .03 \\ .02 \\ .01 \\ .08 \\ .11 \\ .38 \\ .23 \\ .05 \\ .08 \end{bmatrix}$$



3072 numbers

10 numbers

10 positive numbers that sum to one

Linear layer

$$Wx + b$$

- Linear transformation:
  - Operation that changes the data dimension from 3072 to 10.
  - Linear layers have both a weight matrix and a bias vector:

$$\begin{bmatrix} .14 & -.20 & -.39 & -.45 & \cdots & -.50 \\ -.22 & -.52 & .44 & -.69 & \cdots & -.75 \\ -.06 & .32 & -.29 & .94 & \cdots & .97 \\ .68 & .32 & .33 & -.84 & \cdots & .34 \\ .55 & -.30 & .90 & .05 & \cdots & .79 \\ -.15 & .02 & .45 & .84 & \cdots & -.08 \\ .24 & -.65 & -.38 & .38 & \cdots & .63 \\ .17 & .27 & -.67 & -.99 & \cdots & -.22 \\ -.74 & .43 & .08 & .47 & \cdots & .95 \\ .15 & -.78 & .59 & -.60 & \cdots & .38 \end{bmatrix} \begin{bmatrix} .12 \\ .32 \\ .74 \\ .65 \\ \vdots \\ .42 \\ .18 \end{bmatrix} + \begin{bmatrix} 2.5 \\ -1.2 \\ -3.4 \\ 1.7 \\ 5.6 \\ -3.1 \\ 2.4 \\ -5.0 \\ 4.7 \\ 1.2 \end{bmatrix} = \begin{bmatrix} -1.6 \\ -0.9 \\ -1.1 \\ -2.5 \\ 0.1 \\ 0.5 \\ 1.7 \\ 1.2 \\ -0.3 \\ 0.2 \end{bmatrix}$$

Weight matrix W
(10 x 3072)

Input data x
3072
numbers

Bias vector b

analogous an "activation threshold". Not impt

10 values

Qn: Is there a point to project data into a higher dimensionality subspace ?

A: Good as if we learn in higher dimensions, we have more "learning capacity"

Bad as we risk facing the problems associated with curse of dimensionality

Linear layer in PyTorch

$$\begin{bmatrix} .14 \\ .72 \\ .54 \\ .82 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ .75 \end{bmatrix}$$

Weight matrix W

Bias vector b

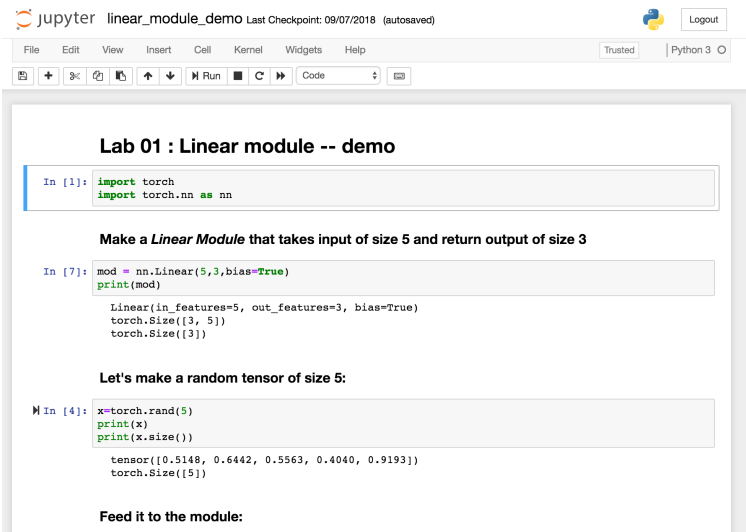
$$\begin{bmatrix} .14 & -.20 & -.39 & -.45 & \cdots & -.50 \\ -.22 & -.52 & .44 & -.69 & \cdots & -.75 \\ -.06 & .32 & -.29 & .94 & \cdots & .97 \\ .68 & .32 & .33 & -.84 & \cdots & .34 \\ .55 & -.30 & .90 & .05 & \cdots & .79 \\ -.15 & .02 & .45 & .84 & \cdots & -.08 \\ .24 & -.65 & -.38 & .38 & \cdots & .63 \\ .17 & .27 & -.67 & -.99 & \cdots & -.22 \\ -.74 & .43 & .08 & .47 & \cdots & .95 \\ .15 & -.78 & .59 & -.60 & \cdots & .38 \end{bmatrix} \quad \begin{bmatrix} 2.5 \\ -1.2 \\ -3.4 \\ 1.7 \\ 5.6 \\ -3.1 \\ 2.4 \\ -5.0 \\ 4.7 \\ 1.2 \end{bmatrix}$$

$$\begin{bmatrix} -1.6 \\ -0.9 \\ -1.1 \\ -2.5 \\ 0.1 \\ 0.5 \\ 1.7 \\ 1.2 \\ -0.3 \\ 0.2 \end{bmatrix}$$

Linear layer
with parameters to be learned

Lab 01

- Linear layer module



jupyter linear_module_demo Last Checkpoint: 09/07/2018 (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Lab 01 : Linear module -- demo

In [1]:
```
import torch
import torch.nn as nn
```

Make a *Linear Module* that takes input of size 5 and return output of size 3

In [7]:
```
mod = nn.Linear(5,3,bias=True)
print(mod)
```

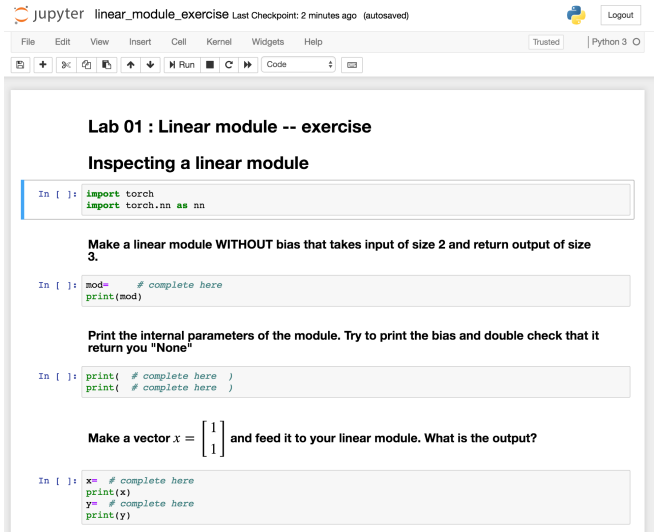
```
Linear(in_features=5, out_features=3, bias=True)
torch.Size([3, 5])
torch.Size([3])
```

Let's make a random tensor of size 5:

In [4]:
```
x=torch.rand(5)
print(x)
print(x.size())
```

```
tensor([0.5148, 0.6442, 0.5563, 0.4040, 0.9193])
torch.Size([5])
```

Feed it to the module:



jupyter linear_module_exercise Last Checkpoint: 2 minutes ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Lab 01 : Linear module -- exercise

Inspecting a linear module

In [ ]:
```
import torch
import torch.nn as nn
```

Make a linear module WITHOUT bias that takes input of size 2 and return output of size 3.

In [ ]:
```
mod=     # complete here
print(mod)
```

Print the internal parameters of the module. Try to print the bias and double check that it return you "None"

In [ ]:
```
print(  # complete here  )
print(  # complete here  )
```

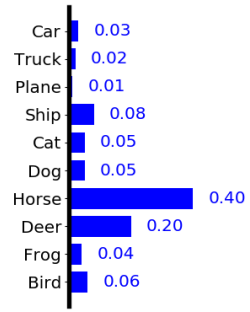
Make a vector $x = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ **and feed it to your linear module. What is the output?**

In [ ]:
```
x=  # complete here
print(x)
y=  # complete here
print(y)
```

Softmax layer

- Softmax layer converts the 10 scores into a probability distribution over the 10 classes:

$$\begin{bmatrix} -1.6 \\ -0.9 \\ -1.1 \\ -2.5 \\ 0.1 \\ 0.5 \\ 1.7 \\ 1.2 \\ -0.3 \\ 0.2 \end{bmatrix} \xrightarrow{\text{exp()}} \begin{bmatrix} 0.2 \\ 0.4 \\ 0.3 \\ 0.1 \\ 1.1 \\ 1.6 \\ 5.5 \\ 3.3 \\ 0.7 \\ 1.2 \end{bmatrix} \xrightarrow{\text{divide by sum()}} \begin{bmatrix} .01 \\ .03 \\ .02 \\ .01 \\ .08 \\ .11 \\ .38 \\ .23 \\ .05 \\ .08 \end{bmatrix}$$



sum = 14.53

sum = 1

Scores

Positive numbers

Probabilities over classes

Softmax layer

- These two operations (exponentiate then unit normalization) can be done into one single step by the Softmax function:

$$\text{Softmax}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}\right) = \begin{bmatrix} \frac{e^{x_1}}{e^{x_1}+e^{x_2}+e^{x_3}+e^{x_4}} \\ \frac{e^{x_2}}{e^{x_1}+e^{x_2}+e^{x_3}+e^{x_4}} \\ \frac{e^{x_3}}{e^{x_1}+e^{x_2}+e^{x_3}+e^{x_4}} \\ \frac{e^{x_4}}{e^{x_1}+e^{x_2}+e^{x_3}+e^{x_4}} \end{bmatrix}$$

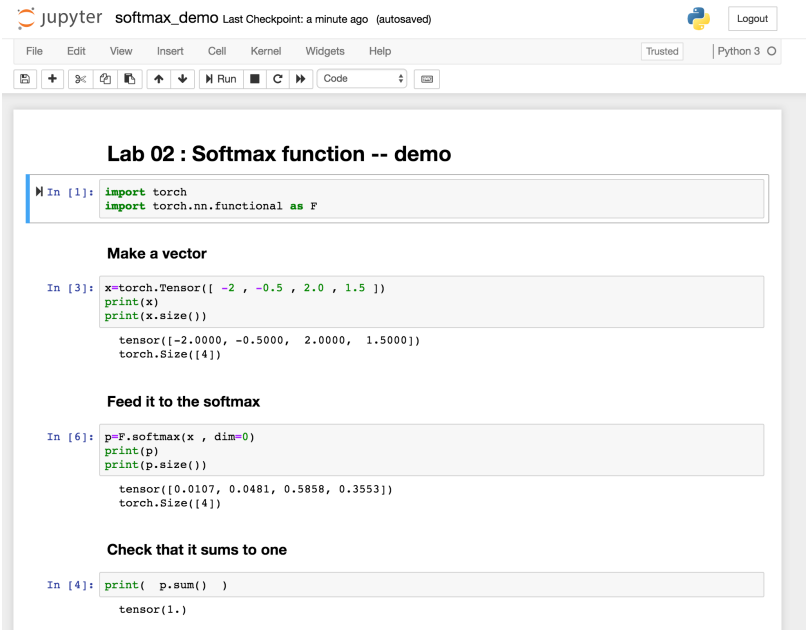
- It is called "Softmax" because it detects the Max : it's a max-detector. It does this by greatly boosting the largest value and greatly minimizing the smallest value using the exponential fn.

Examples:

$$\text{Softmax}\left(\begin{bmatrix} -2.0 \\ -0.5 \\ 2.0 \\ 1.5 \end{bmatrix}\right) = \begin{bmatrix} .01 \\ .04 \\ .59 \\ .36 \end{bmatrix} \qquad \text{Softmax}\left(\begin{bmatrix} -2.0 \\ -0.5 \\ 7.0 \\ 1.5 \end{bmatrix}\right) = \begin{bmatrix} .00 \\ .00 \\ .99 \\ .01 \end{bmatrix}$$

Lab 02

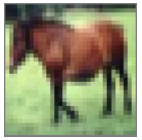
- Softmax layer in PyTorch



One-layer neural network

← theoretically simplest NN.

$$\begin{bmatrix} .14 \\ .72 \\ .54 \\ .82 \\ \vdots \\ .75 \end{bmatrix}$$



Weight matrix W

Bias vector b

$$\begin{bmatrix} .14 & -.20 & -.39 & -.45 & \cdots & -.50 \\ -.22 & -.52 & .44 & -.69 & \cdots & -.75 \\ -.06 & .32 & -.29 & .94 & \cdots & .97 \\ .68 & .32 & .33 & -.84 & \cdots & .34 \\ .55 & -.30 & .90 & .05 & \cdots & .79 \\ -.15 & .02 & .45 & .84 & \cdots & -.08 \\ .24 & -.65 & -.38 & .38 & \cdots & .63 \\ .17 & .27 & -.67 & -.99 & \cdots & -.22 \\ -.74 & .43 & .08 & .47 & \cdots & .95 \\ .15 & -.78 & .59 & -.60 & \cdots & .38 \end{bmatrix} \quad \begin{bmatrix} 2.5 \\ -1.2 \\ -3.4 \\ 1.7 \\ 5.6 \\ -3.1 \\ 2.4 \\ -5.0 \\ 4.7 \\ 1.2 \end{bmatrix}$$

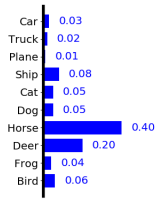
Linear layer
with parameters to be learned

$$\begin{bmatrix} -1.6 \\ -0.9 \\ -1.1 \\ -2.5 \\ 0.1 \\ 0.5 \\ 1.7 \\ 1.2 \\ -0.3 \\ 0.2 \end{bmatrix}$$

Softmax

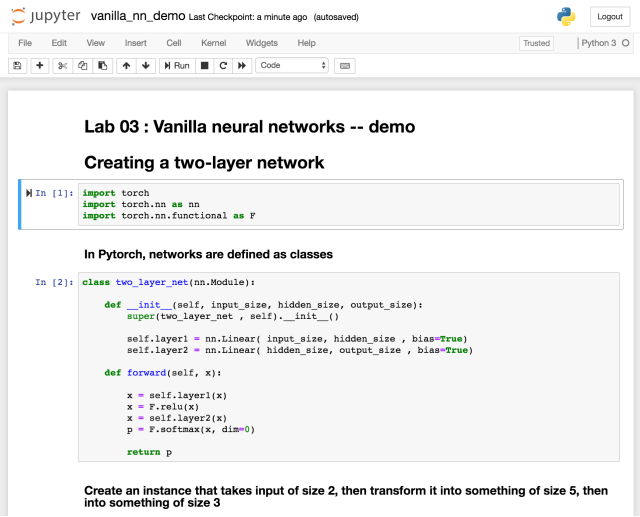
Softmax layer
(no parameters to learn)

$$\begin{bmatrix} .01 \\ .03 \\ .02 \\ .01 \\ .08 \\ .11 \\ .38 \\ .23 \\ .05 \\ .08 \end{bmatrix}$$



Lab 03

- Design a vanilla neural network in PyTorch :



jupyter vanilla_nn_demo Last Checkpoint: a minute ago (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

Lab 03 : Vanilla neural networks -- demo

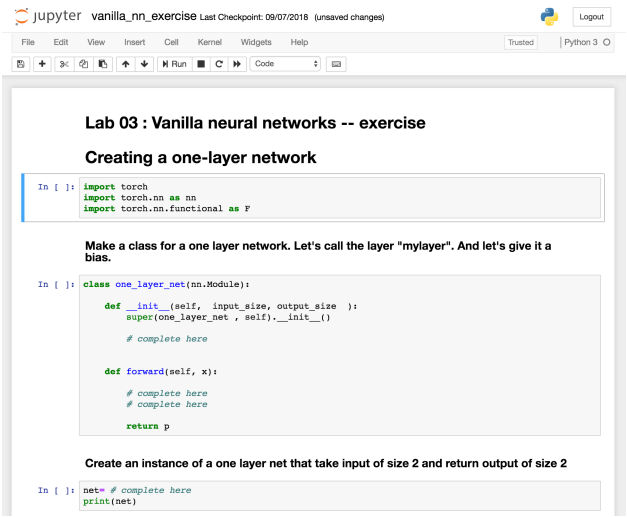
Creating a two-layer network

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

In Pytorch, networks are defined as classes

In [2]:
```
class two_layer_net(nn.Module):

    def __init__(self, input_size, hidden_size, output_size):
        super(two_layer_net , self).__init__()

        self.layer1 = nn.Linear( input_size, hidden_size , bias=True)
        self.layer2 = nn.Linear( hidden_size, output_size , bias=True)

    def forward(self, x):

        x = self.layer1(x)
        x = F.relu(x)
        x = self.layer2(x)
        p = F.softmax(x, dim=0)

        return p
```

Create an instance that takes input of size 2, then transform it into something of size 5, then into something of size 3



jupyter vanilla_nn_exercise Last Checkpoint: 09/07/2018 (unsaved changes)

Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

Lab 03 : Vanilla neural networks -- exercise

Creating a one-layer network

In [ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

Make a class for a one layer network. Let's call the layer "mylayer". And let's give it a bias.

In [ ]:
```
class one_layer_net(nn.Module):

    def __init__(self,  input_size, output_size  ):
        super(one_layer_net , self).__init__()

        # complete here


    def forward(self, x):

        # complete here
        # complete here

        return p
```

Create an instance of a one layer net that take input of size 2 and return output of size 2

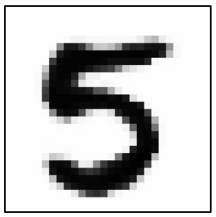
In [ ]:
```
net= # complete here
print(net)
```

Outline

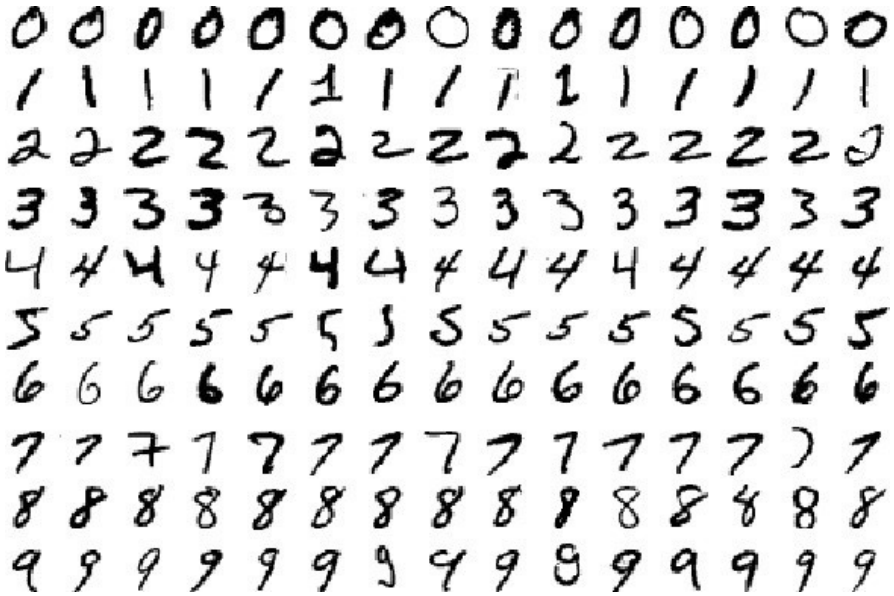
- Image classification task
- Neural networks for image classification
- Forward pass for inference
- **Understanding linear layers**
- Understanding the backward pass
- Matrix formulation
- Mini-batch learning

MNIST dataset

- Number of classes: 10
- Training set: 50,000 labelled images
- Test set: 10,000 labelled images



28 x 28 grayscale image



MNIST dataset

- For simplicity, we will pretend :
  - We have only 4 classes: Zero, One, Two, Three.
  - We have no bias b.

classes

pixels

Weight matrix: 4 x 784

$$\begin{bmatrix} .17 \\ .42 \\ .94 \\ .87 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ .32 \\ .27 \end{bmatrix} \Rightarrow \begin{bmatrix} -.12 & -.17 & -.15 & \cdots & -.09 & -.13 \\ -.14 & -.11 & -.16 & \cdots & -.12 & -.09 \\ .19 & .22 & .15 & \cdots & .21 & .18 \\ -.12 & -.08 & -.09 & \cdots & -.14 & -.16 \end{bmatrix} \Rightarrow \begin{bmatrix} -0.1 \\ -0.5 \\ 0.9 \\ 2.2 \end{bmatrix}$$

Linear layer

with parameters W to be learned

What linear layers do?

- A linear layer is a template matching algorithm !
- Each row of the weight matrix W is a template (encoding some data property):

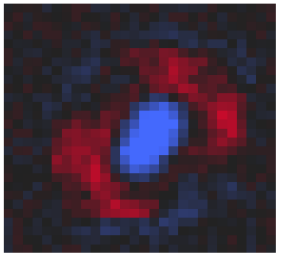
$$\begin{bmatrix} -.12 & -.17 & -.15 & \cdots & -.09 & -.13 \\ -.14 & -.11 & -.16 & \cdots & -.12 & -.09 \\ .19 & .22 & .15 & \cdots & .21 & .18 \\ -.12 & -.08 & -.09 & \cdots & -.14 & -.16 \end{bmatrix}$$

4 x 784 weight matrix W

Row zero reshaped as (28 x 28)

$$\begin{bmatrix} .14 & -.20 & -.39 & -.45 \\ -.94 & -.05 & .57 & .42 \\ .45 & -.46 & .97 & .08 \\ -.72 & .77 & -.81 & -.56 \\ -.34 & -.25 & -.39 & -.50 \end{bmatrix}$$

||

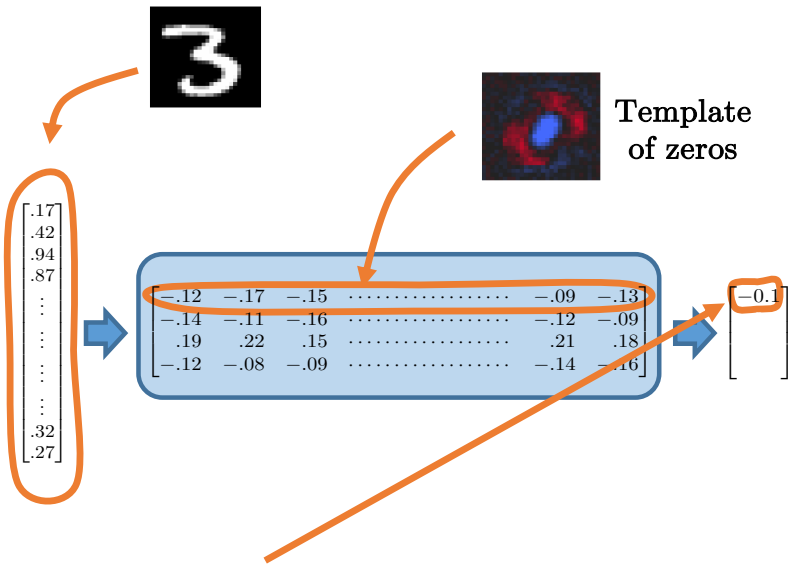


← looks like image of number 0.

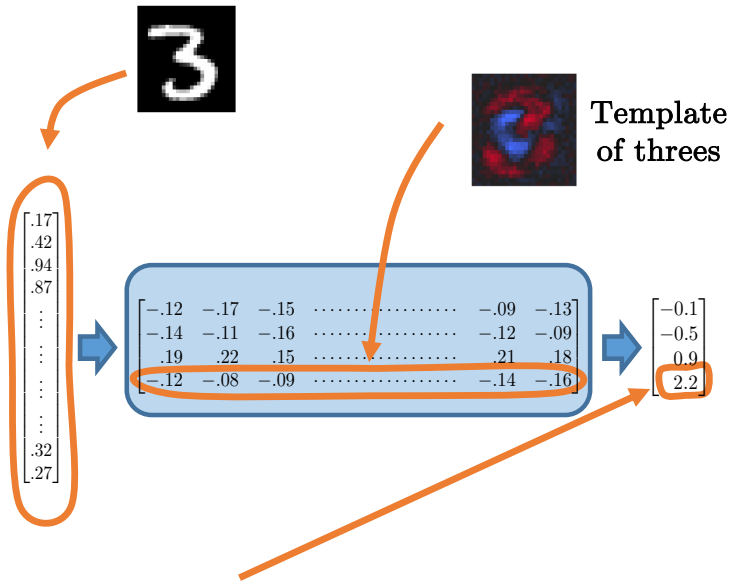
If input has high agreement with this template, then the score will be high

Template matching

- Template matching performs vector-vector multiplication between data and class template.
  - It produces a matching score between data and class template.



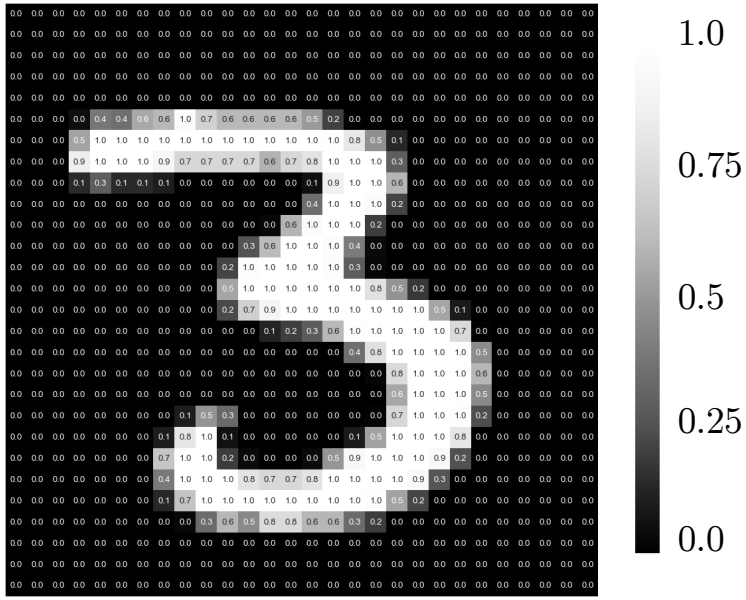
Matching score between the input data and the template of zeros

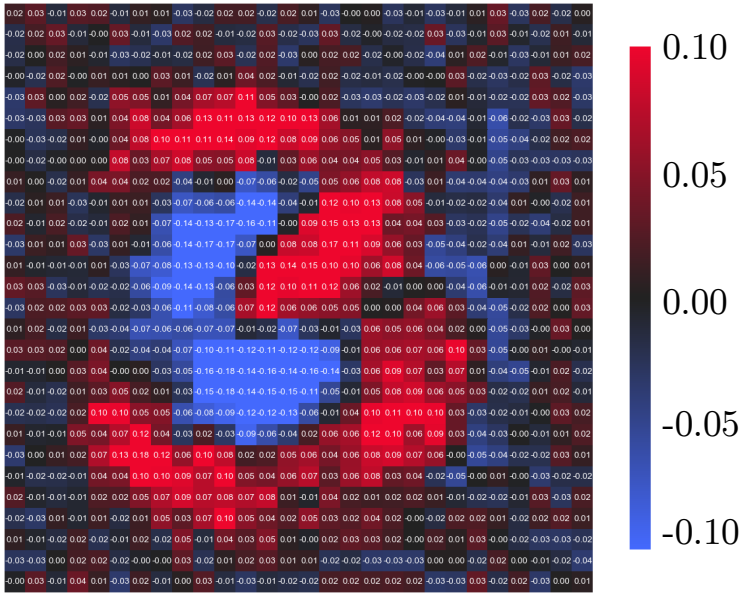


Matching score between the input data and the template of threes

Template matching

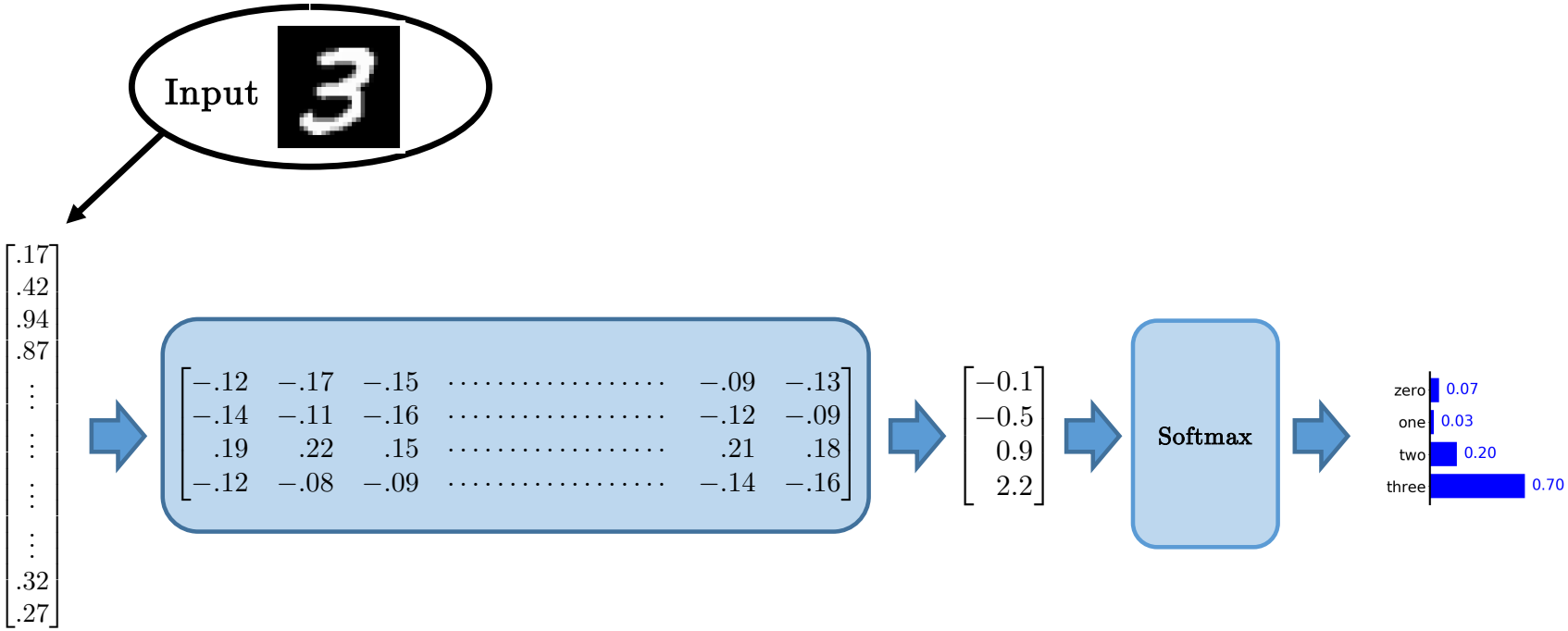
- **Matching score** = **inner product of input image and class template**





From score to probability

- Scores are converted into probabilities with the softmax layer :



Template matching

- Template matching property of DL architectures is a first principle.
- However, deep learning is more than template matching !
- Most research focus on finding the right architecture to solve complex tasks (that cannot be solved by simple template matching only).

Outline

- Image classification task
- Neural networks for image classification
- Forward pass for inference
- Understanding linear layers
- **Understanding the backward pass**
- Matrix formulation
- Mini-batch learning

How to learn the network parameters?

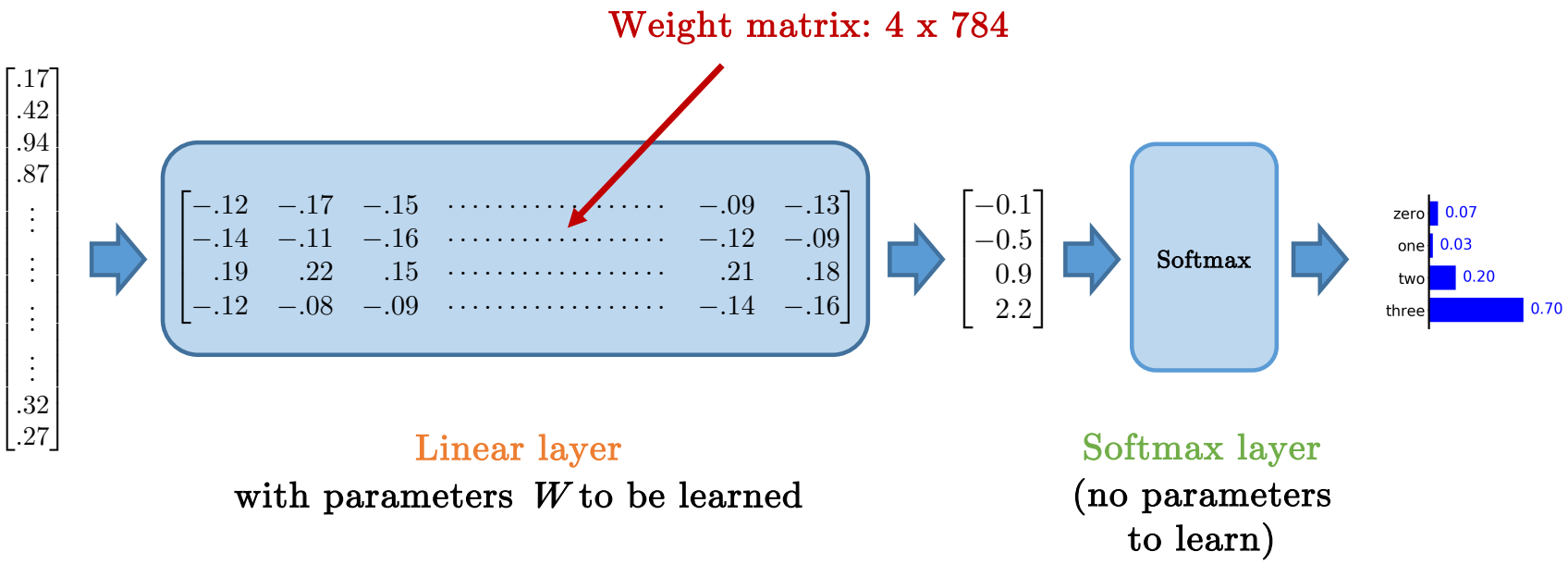
- So far, we have done inference : We have used the weight parameters that best classify images.
- We need to learn these parameters (here the weight and bias of the linear layer).
- Network parameters are learned by backpropagation:
  - A backward pass that backpropagates the error of prediction to adjust the weights on the network.
  - The update rule of the weights is :

$$\text{template} \quad = \quad \text{template} \quad + \quad \text{lr} \times (\text{error}) \times (\text{input picture})$$

$$W = W + \text{lr}\,(\text{error vector} \otimes \text{input picture})$$

Template learning

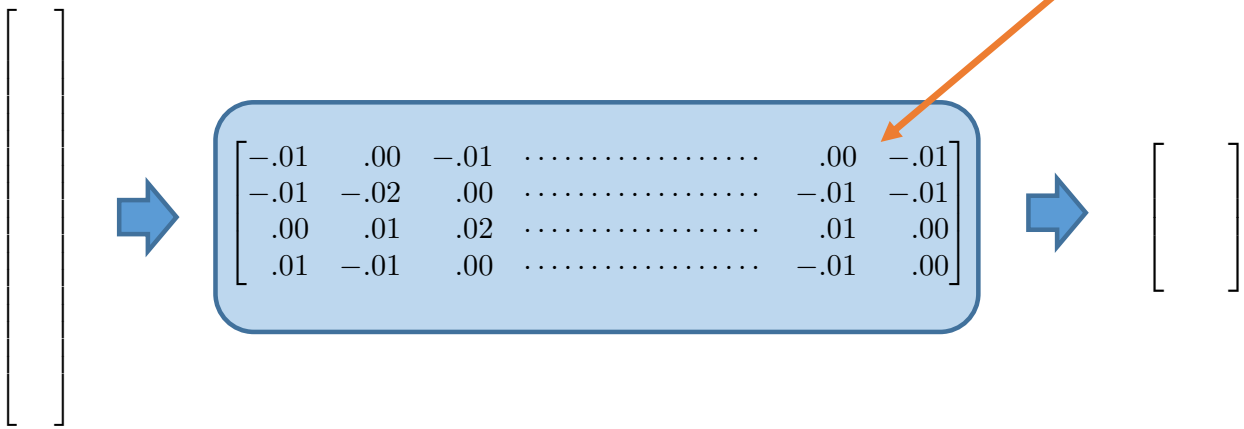
- For simplicity, we will still pretend :
  - We have only 4 classes: Zero, One, Two, Three.
  - We have no bias b.



Template learning

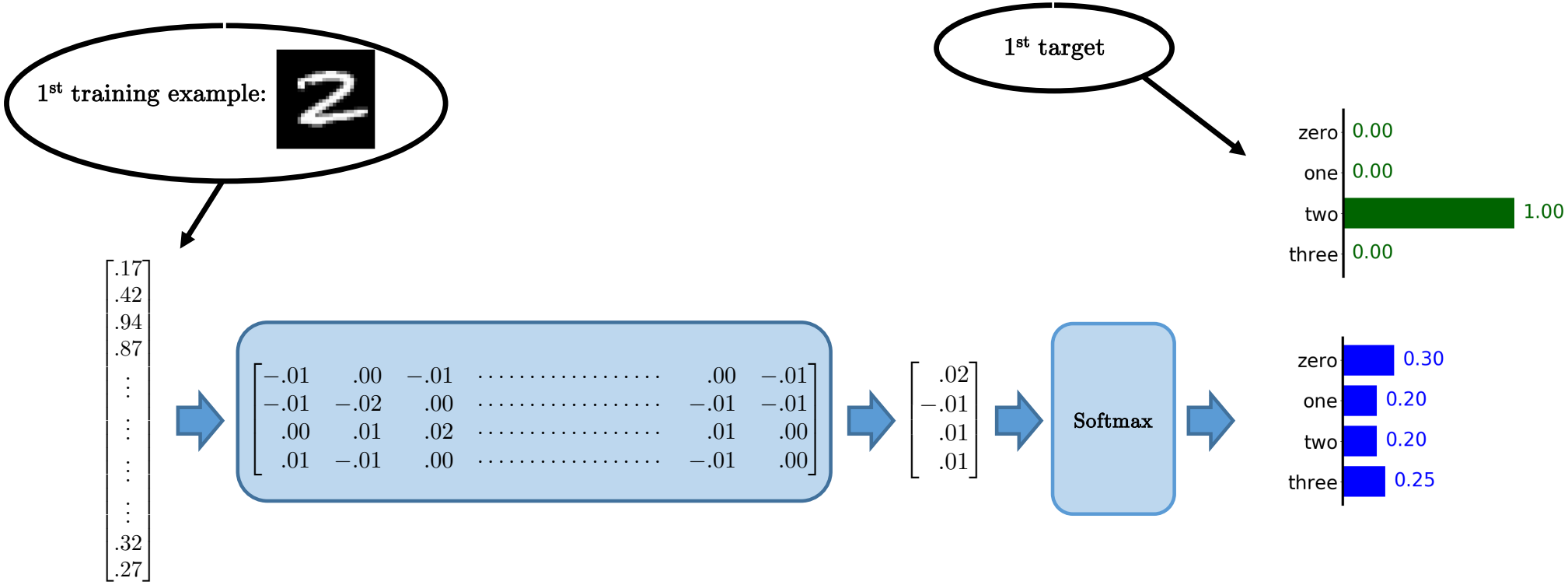
- How does the network learn good templates?
- We will use an iterative step that will update progressively the templates to minimize the error of prediction.
- Initialization:
  - At the beginning of the training the weight matrix is initialized with small random numbers.

why small ?



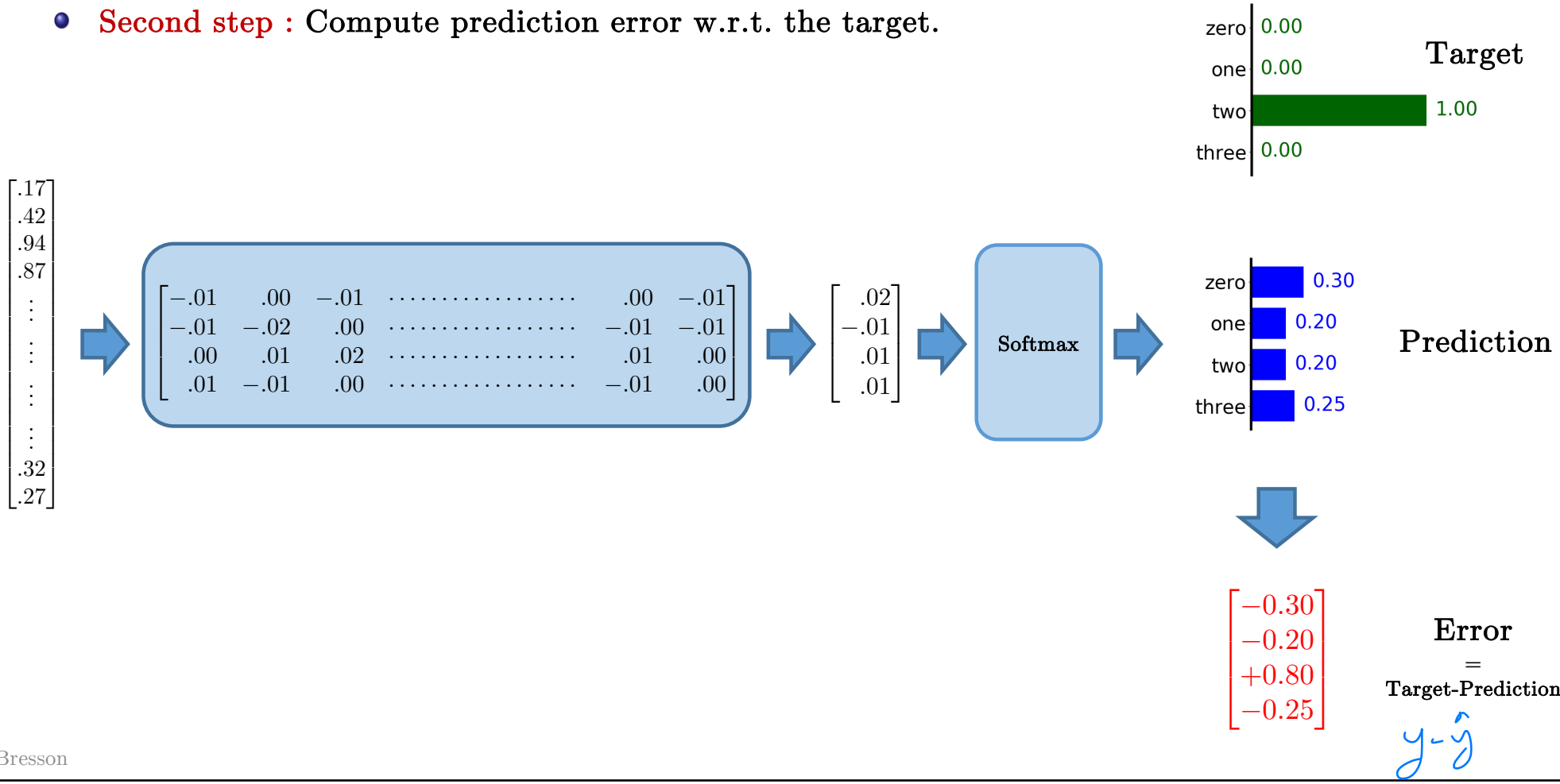
Forward pass

- **First step :** Compute the class probabilities for the 1st training data.



Prediction error

- **Second step :** Compute prediction error w.r.t. the target.



$$\begin{bmatrix} .17 \\ .42 \\ .94 \\ .87 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ .32 \\ .27 \end{bmatrix} \Rightarrow \begin{bmatrix} -.01 & .00 & -.01 & \cdots & .00 & -.01 \\ -.01 & -.02 & .00 & \cdots & -.01 & -.01 \\ .00 & .01 & .02 & \cdots & .01 & .00 \\ .01 & -.01 & .00 & \cdots & -.01 & .00 \end{bmatrix} \Rightarrow \begin{bmatrix} .02 \\ -.01 \\ .01 \\ .01 \end{bmatrix} \Rightarrow \text{Softmax} \Rightarrow \text{Prediction} \Rightarrow \begin{bmatrix} -0.30 \\ -0.20 \\ +0.80 \\ -0.25 \end{bmatrix}$$

Template update

- **Third step :** Update templates

$y - \hat{y}$

$$\text{template} = \text{template} + \text{lr} \times (\text{error}) \times (\text{input picture})$$

- Each time the network is presented with a picture, let say:

- It adds this picture to the correct class template (here class 2). $(y-\hat{y})_2 \geq 0 \because y_2 = 1$ and $0 \leq \hat{y}_2 \leq 1$

  $\therefore$ We add a scaled-down version of the image to template

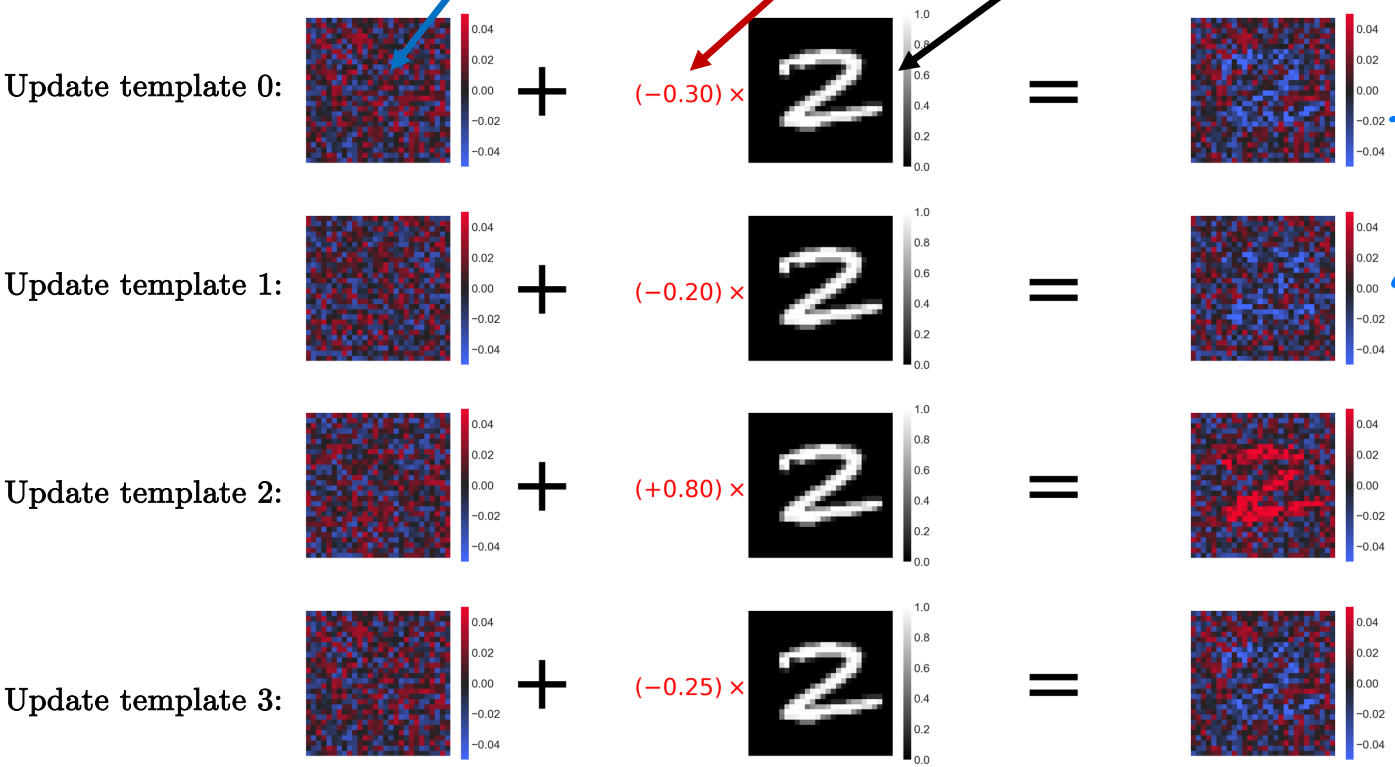
- And removes it from the other class templates (here classes 0, 1, 3). $(y-\hat{y})_{i \neq 2} \leq 0 \because y_{i \neq 2} = 0$ and $0 \leq \hat{y}_{i \neq 2} \leq 1$

  $\therefore$ We subtract a scaled-down version of the image to template

Template update

- **Applying the template update:**

$$\text{template} = \text{template} + \text{lr} \times (\text{error}) \times (\text{input picture})$$



Gives low score if input looks like 2

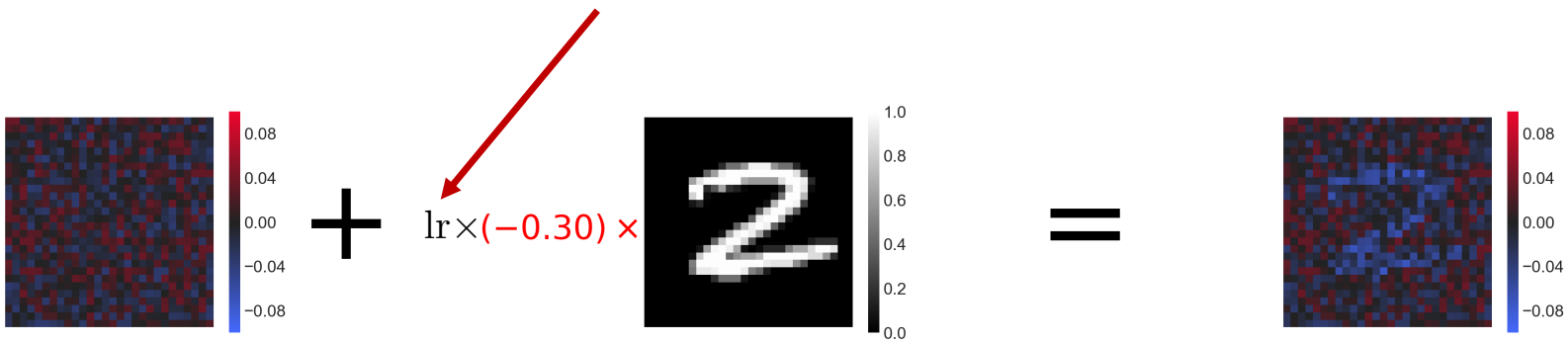
Gives high score if input looks like 2.

How does batching work here ?

Learning rate

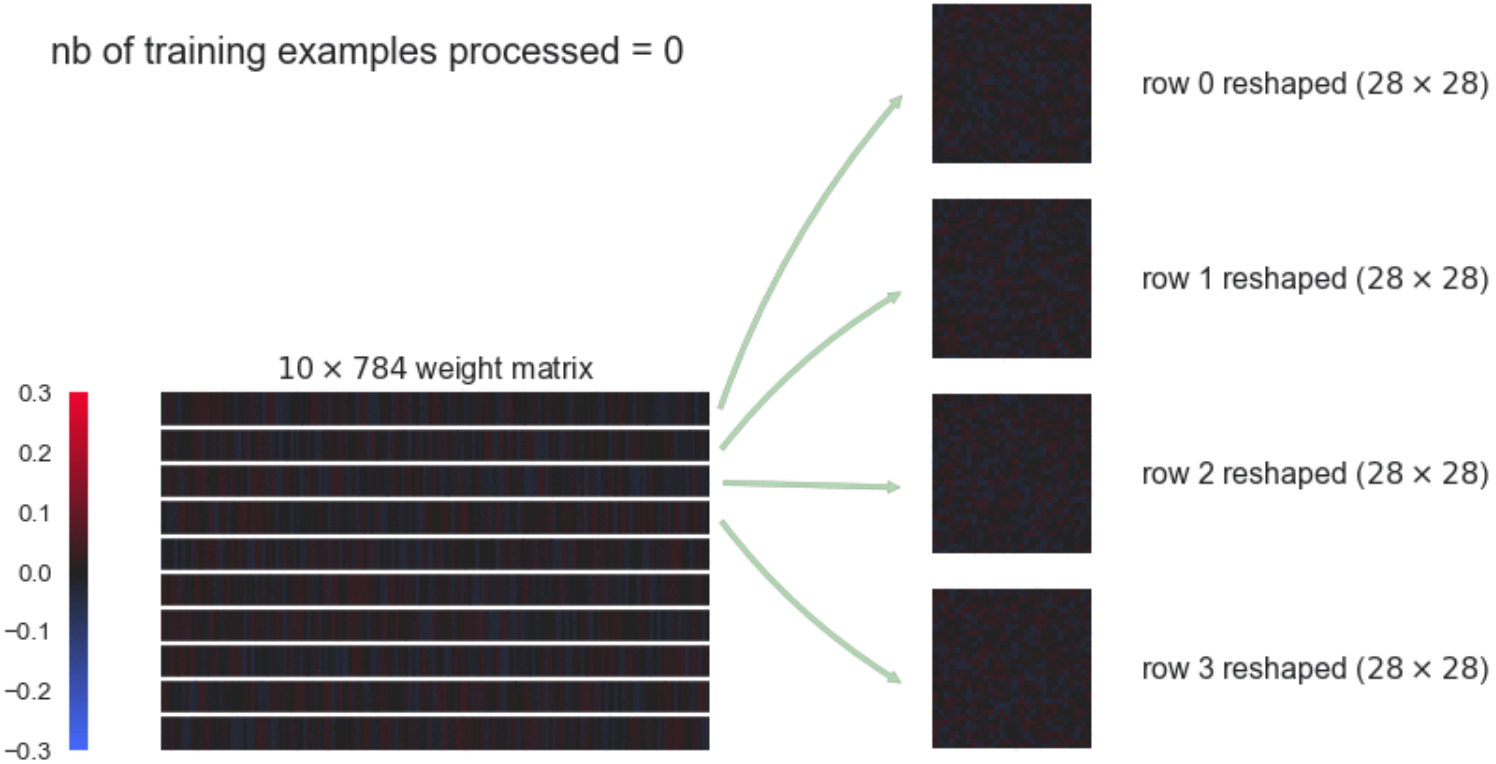
- The learning rate (lr) is a very important hyper-parameter.
- The learning rate controls the learning speed:
  - Too small : Learning is too long.
  - Too large: Learning is too fast and it does not work.
- Typical choice to update the templates is

$$\text{lr} \approx 0.01$$



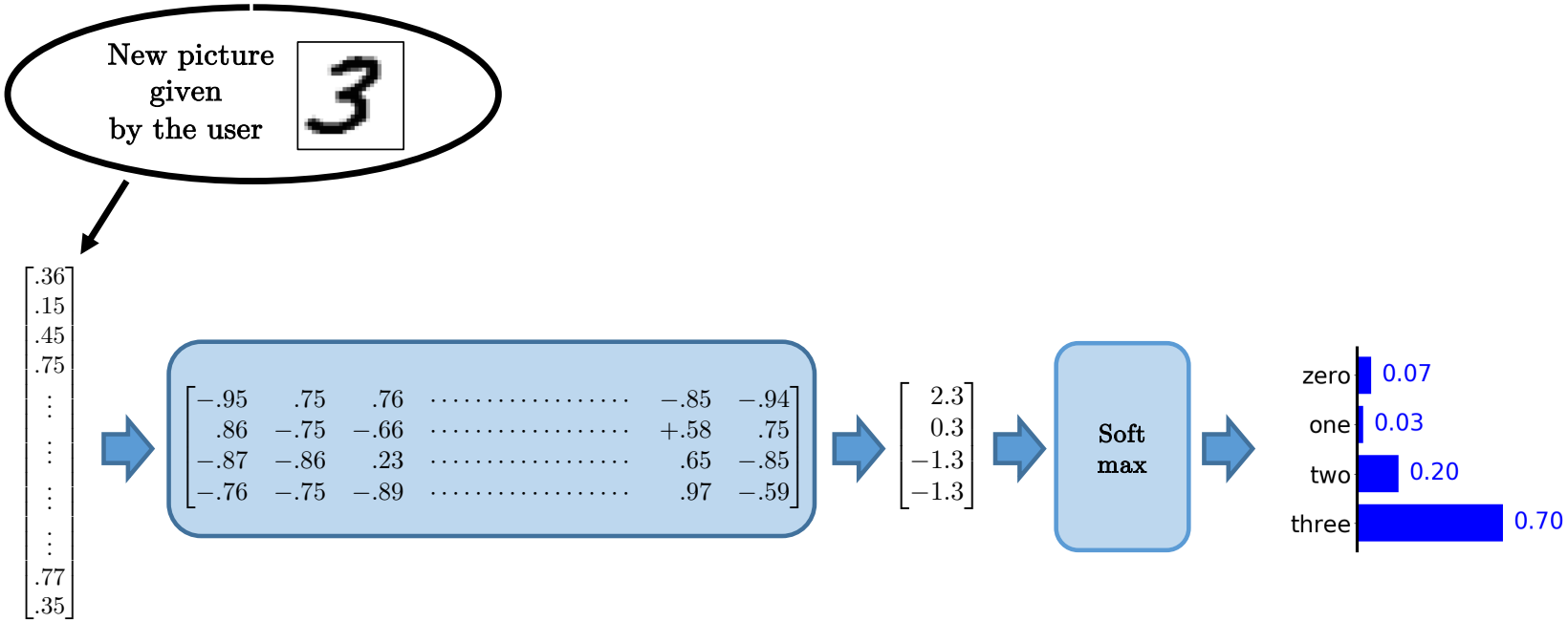
Iterate

- Iterate this process with all training data and multiple times:
  - After many iterations, the templates look good :



Network ready for inference

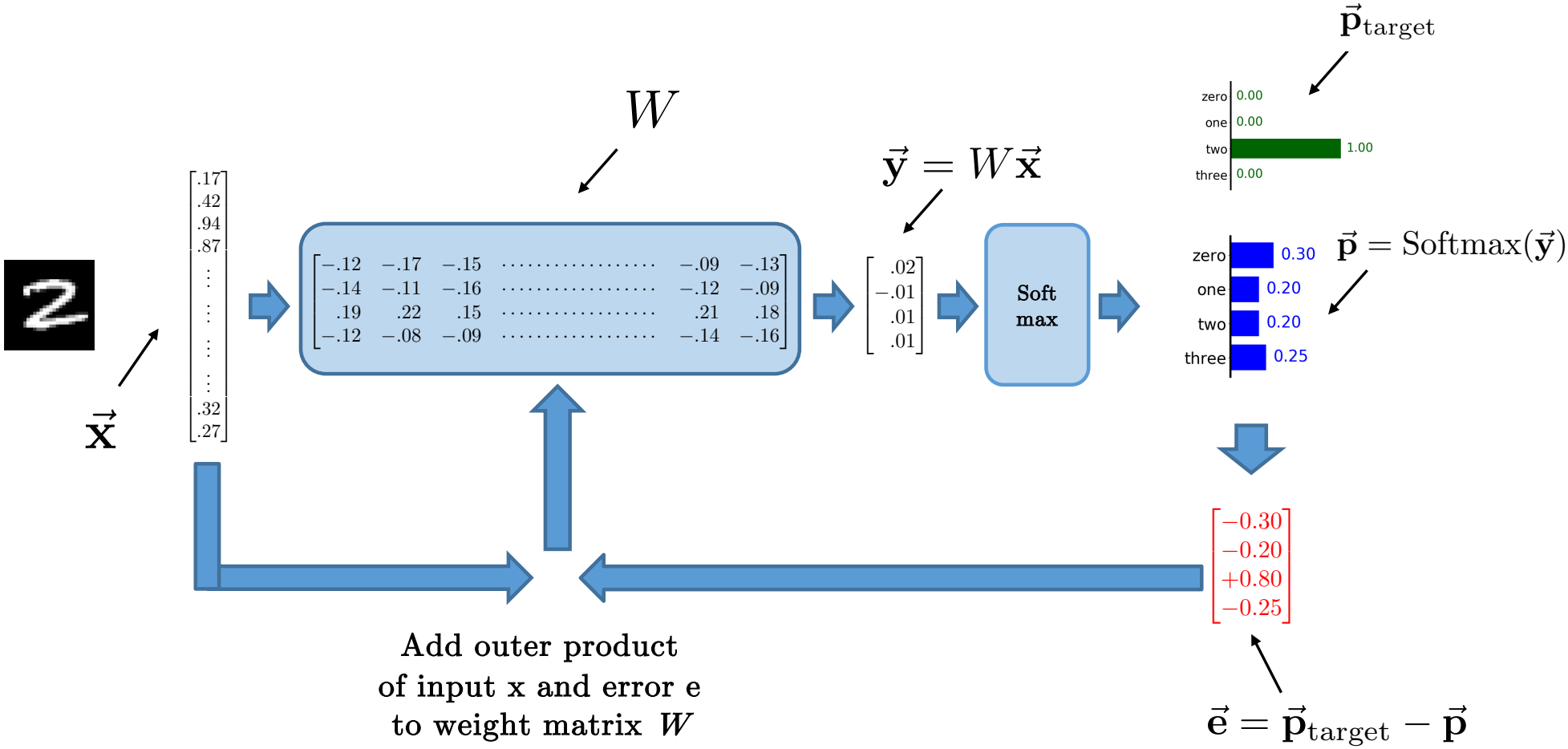
- After many iterations, the templates do not update anymore, the training is finished.
- The network is ready for inference for new data.



Outline

- Image classification task
- Neural networks for image classification
- Forward pass for inference
- Understanding linear layers
- Understanding the backward pass
- **Matrix formulation**
- Mini-batch learning

Matrix notation



Add outer product
of input x and error e
to weight matrix W

$$W = W + \text{lr}\,(\vec{\mathbf{e}} \otimes \vec{\mathbf{x}})$$

Matrix formulation of outer product

outer product

$$\vec{\mathbf{e}} \otimes \text{(input picture)} = \begin{bmatrix} -0.30 \\ -0.20 \\ +0.80 \\ -0.25 \end{bmatrix} \overbrace{\begin{bmatrix} .17 & .42 & .94 & \cdots & \cdots & .32 & .27 \end{bmatrix}}^{\text{input image}}$$

$$= \underbrace{\begin{bmatrix} -0.30 \times \text{input picture} \\ -0.20 \times \text{input picture} \\ +0.80 \times \text{input picture} \\ -0.25 \times \text{input picture} \end{bmatrix}}_{\text{dim } ?}$$

$\underbrace{4 \times 784}_{\text{same dim as weight matrix } W.}$

Forward and backward pass

- **Forward pass:**

$$\vec{\mathbf{y}} = W\vec{\mathbf{x}}$$ Compute the scores

$$\vec{\mathbf{p}} = \mathrm{Softmax}(\vec{\mathbf{y}})$$ Compute the probabilities

- **Backward pass:**

$$\vec{\mathbf{e}} = \vec{\mathbf{p}}_{\mathrm{target}} - \vec{\mathbf{p}}$$ Compute the error

$$W = W + \mathrm{lr}\,(\vec{\mathbf{e}} \otimes \vec{\mathbf{x}})$$ Update the templates

Lab 04

- Train a one layer net

jupyter train_vanilla_nn_demo Last Checkpoint: 09/08/2018 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Lab 04 : Train vanilla neural network -- demo ¶

Training a one-layer net on MNIST

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from random import randint
import utils
```

randint(a,b) returns a random integer between a and b:

In [2]:
```
idx=randint(5,10) # generate an integer randomly from 5 to 10 included.

print(idx)
```

```
8
```

Download the TRAINING SET (data+labels)

In [3]:
```
from utils import check_mnist_dataset_exists
data_path=check_mnist_dataset_exists()
```

In [4]:
```
train_data=torch.load(data_path+'mnist/train_data.pt')

print(train_data.size())
```

```
torch.Size([60000, 28, 28])
```

jupyter train_vanilla_nn_exercise Last Checkpoint: 3 minutes ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Lab 04 : Train vanilla neural network -- exercise

Training a one-layer net on FASHION-MNIST

In [ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from random import randint
import utils
```

Download the TRAINING SET (data+labels)

In [ ]:
```
from utils import check_fashion_mnist_dataset_exists
data_path=check_fashion_mnist_dataset_exists()

train_data=torch.load(data_path+'fashion-mnist/train_data.pt')
train_label=torch.load(data_path+'fashion-mnist/train_label.pt')
print(train_data.size())
print(train_label.size())
```

Download the TEST SET (data only)

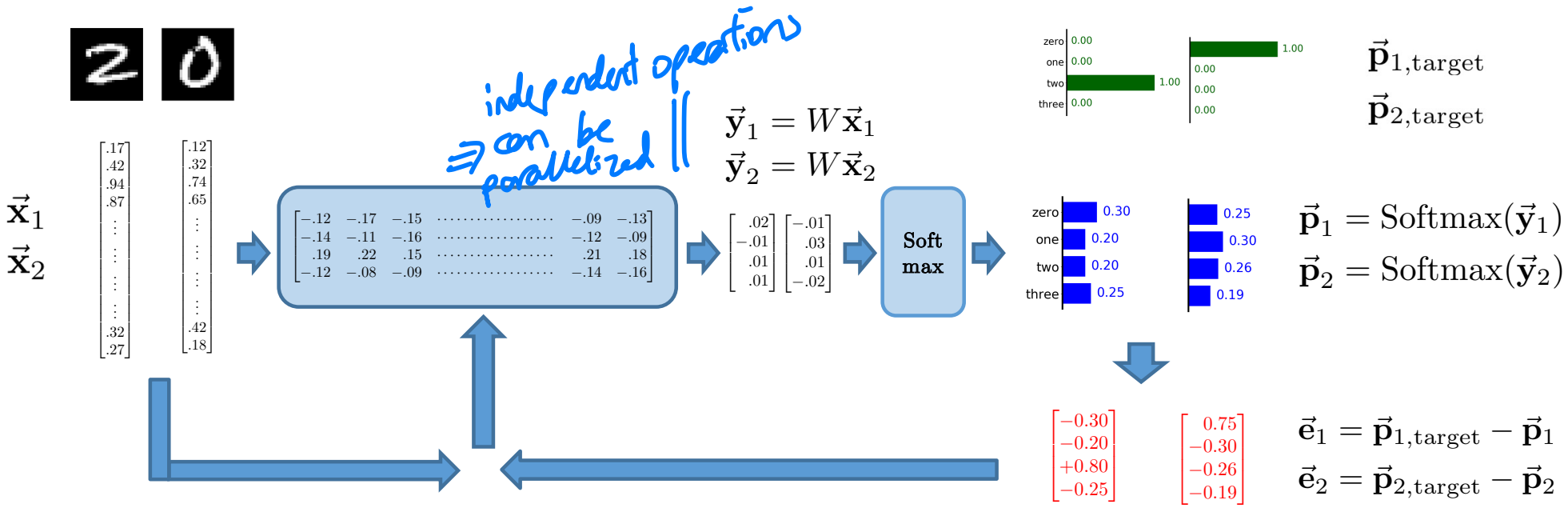
In [ ]:
```
test_data=torch.load(data_path+'fashion-mnist/test_data.pt')
print(test_data.size())
```

Outline

- Image classification task
- Neural networks for image classification
- Forward pass for inference
- Understanding linear layers
- Understanding the backward pass
- Matrix formulation
- **Mini-batch learning**

Mini-batch

- Rather than processing one image data at a time, let us process multiple images at a time :



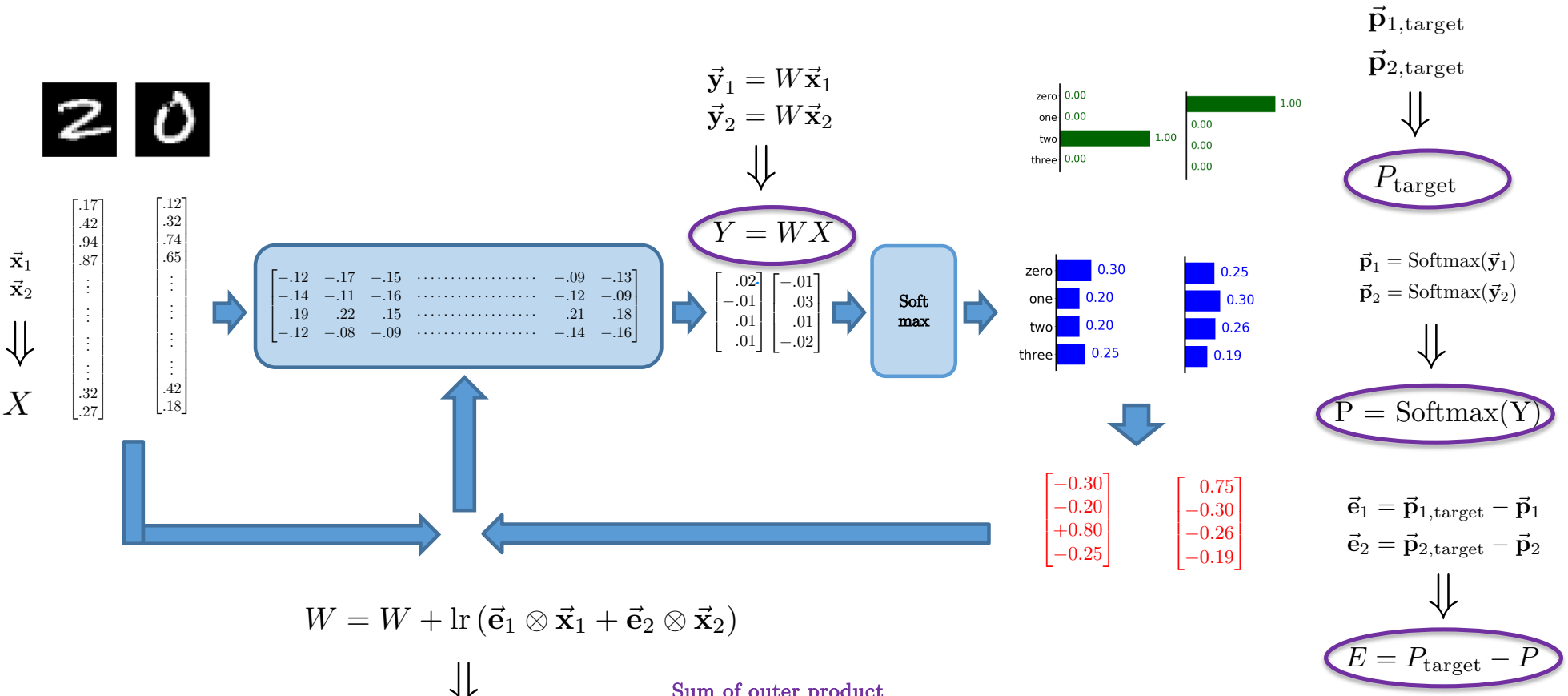
$$W = W + \mathrm{lr}\,(\vec{\mathbf{e}}_1 \otimes \vec{\mathbf{x}}_1) + \mathrm{lr}\,(\vec{\mathbf{e}}_2 \otimes \vec{\mathbf{x}}_2)$$

$$W = W + \sum_b^B lr\,(\vec{e}_b \otimes \vec{x}_b)$$

Qn: Why not update W using the average $\frac{1}{B}\sum_b^B lr\,(\vec{e}_b \otimes \vec{x}_b)$?

A: It's the same as the given formulation but setting $lr^{new} = \frac{1}{B} lr$

Mini-batch matrix formulation



$$W = W + \text{lr}\left(\vec{\mathbf{e}}_1 \otimes \vec{\mathbf{x}}_1 + \vec{\mathbf{e}}_2 \otimes \vec{\mathbf{x}}_2\right)$$

$$\Downarrow$$

$$W = W + \text{lr}\left(EX^T\right)$$

Sum of outer product can be written as a mat-mat multiplication.

error matrix

$X = \begin{bmatrix} | & | \\ x_1 & x_2 \\ | & | \end{bmatrix}$

$Y = \begin{bmatrix} | & | \\ y_1 & y_2 \\ | & | \end{bmatrix}$

classes $\rightarrow$ $C \times B$ or $4 \times B$

predictors $B \times P$ or $B \times 784$

$C \times B$

Matrix formulation of outer product

$$\vec{\mathbf{e}}_1 \otimes \vec{\mathbf{x}}_1 + \vec{\mathbf{e}}_2 \otimes \vec{\mathbf{x}}_2 = \underbrace{\begin{bmatrix} -0.30 \\ -0.20 \\ +0.80 \\ -0.25 \end{bmatrix}}_{C \times 1} \underbrace{\begin{bmatrix} .17 & .42 & .94 & .87 & \cdots & \cdots & \cdots & \cdots & .32 & .27 \end{bmatrix}}_{1 \times p} + \begin{bmatrix} +0.75 \\ -0.30 \\ -0.26 \\ -0.19 \end{bmatrix} \begin{bmatrix} .12 & .32 & .74 & .65 & \cdots & \cdots & \cdots & \cdots & .42 & .18 \end{bmatrix}$$

$$\Downarrow$$

$$\underbrace{\underset{C \times B}{E}\,\underset{B \times p}{X^T}}_{C \times p} = \begin{bmatrix} -0.30 & +0.75 \\ -0.20 & -0.30 \\ +0.80 & -0.26 \\ -0.25 & -0.19 \end{bmatrix} \begin{bmatrix} .17 & .42 & .94 & .87 & \cdots & \cdots & \cdots & \cdots & .32 & .27 \\ .12 & .32 & .74 & .65 & \cdots & \cdots & \cdots & \cdots & .42 & .18 \end{bmatrix}$$

Forward and backward pass for mini-batch

- **Forward pass:**

$$Y = WX$$ Compute the scores

$$\mathrm{P} = \mathrm{Softmax}(\mathrm{Y})$$ Compute the probabilities

- **Backward pass:**

$$E = P_{\text{target}} - P$$ Compute the error

$$W = W + \mathrm{lr}\left(EX^T\right)$$ Update the templates

Serial vs Batch

- What is the difference between serial and batch learning?
  - Serial learning :

is first processed by

$$\begin{bmatrix} -.01 & .00 & -.01 & \cdots & .00 & -.01 \\ -.01 & -.02 & .00 & \cdots & -.01 & -.01 \\ .00 & .01 & .02 & \cdots & .01 & .00 \\ .01 & -.01 & .00 & \cdots & -.01 & .00 \end{bmatrix}$$

The errors is computed and then the weights are updated.

Then

is processed by

$$\begin{bmatrix} -.06 & .03 & .03 & \cdots & .00 & -.01 \\ .08 & .03 & -.04 & \cdots & -.01 & -.01 \\ -.04 & -.09 & .05 & \cdots & .01 & .15 \\ .12 & -.05 & .03 & \cdots & -.01 & .18 \end{bmatrix}$$

The errors is computed and then the weight are updated.

  - Serial mode : The weights have been updated twice.
    - When the second image is processed, it benefits of better freshly updated weights. As a consequence the network trains fast.

"We want to update W as much as possible, but it is slow" ? why do we want to update W often

? But using this weight update formula, wouldn't the resulting W be the same irregardless of serial or batch?

Serial:

$$W^{(1)} = W^{(0)} + lr(\vec{e_1} \otimes \vec{x_1})$$

$$W^{(2)} = W^{(1)} + lr(\vec{e_2} \otimes \vec{x_2})$$

$$= \underbrace{W^{(0)} + lr(\vec{e_1} \otimes \vec{x_1}) + lr(\vec{e_2} \otimes \vec{x_2})}$$

Batch

$$W^{(1)} = \underbrace{W^{(0)} + lr(\vec{e_1} \otimes \vec{x_1}) + lr(\vec{e_2} \otimes \vec{x_2})}$$

same?

Serial vs Batch

- Batch learning :

are both processed by

$$\begin{bmatrix} -.01 & .00 & -.01 & \cdots\cdots\cdots\cdots & .00 & -.01 \\ -.01 & -.02 & .00 & \cdots\cdots\cdots\cdots & -.01 & -.01 \\ .00 & .01 & .02 & \cdots\cdots\cdots\cdots & .01 & .00 \\ .01 & -.01 & .00 & \cdots\cdots\cdots\cdots & -.01 & .00 \end{bmatrix}$$

The errors are computed and then the weight are updated.

Batch mode : The weights have been updated only once.

Best accuracy. Can prove mathematically ☺ How

more context switching

Serial vs Batch

slow ∵ many data reads from hard drive to memory. Batching requires less hard drive reads. Read more at once

- **No batch (single data):** So processing one image after the other, and updated the weights each time, seems clearly better.

Batch size = 1

- **Mini-batch:** On the other hand, processing images by batches of e.g. 200, and updating the weights only every 200 images, is also an attractive option for GPUs.

Batch size = 200

- **Full batch:** What about doing batch of 60,000 (all training set) and processing everything in parallel? Not good:
  - Slow update: updating the weights only every 60,000 images.
  - Overfitting issue

Batch size = 60,000

Best choice!

Lab 05

- Train a net with minibatch

jupyter minibatch_training_demo Last Checkpoint: 3 minutes ago (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

Lab 05 : Train with mini-batches -- demo

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from random import randint
import utils
```

Warm up

Make a random matrix with 5 rows

In [2]:
```
A=torch.rand(5,3)
print(A)
```

```
tensor([[0.2976, 0.5865, 0.8216],
        [0.7939, 0.6613, 0.5833],
        [0.7961, 0.5487, 0.4077],
        [0.2582, 0.0407, 0.4789],
        [0.0092, 0.7814, 0.7776]])
```

Choose at random two indices in {0,1,2,3,4}

In [3]:
```
indices=torch.LongTensor(2).random_(0,5) # generate an integer randomly from 0 to 5-1.
print(indices)
```

```
tensor([3, 2])
```

jupyter minibatch_training_exercise Last Checkpoint: 3 minutes ago (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

Lab 05 : Train with mini-batches -- exercise

In [ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from random import randint
import utils
```

Download the data and print the sizes

In [ ]:
```
from utils import check_fashion_mnist_dataset_exists
data_path=check_fashion_mnist_dataset_exists()
```

In [ ]:
```
train_data=torch.load(data_path+'fashion-mnist/train_data.pt')

print(train_data.size())
```

In [ ]:
```
train_label=torch.load(data_path+'fashion-mnist/train_label.pt')

print(train_label.size())
```

In [ ]:
```
test_data=torch.load(data_path+'fashion-mnist/test_data.pt')

print(test_data.size())
```

Make a one layer net class

Questions?

CS5242 : Neural Networks and Deep Learning

Lecture 3 : Vanilla Neural Networks
Part 2 : Loss and Optimization

Semester 2 2024/25

Xavier Bresson

https://twitter.com/xbresson

Department of Computer Science
National University of Singapore (NUS)

Outline

- Quality of neural network
- Maximizing the quality by gradient ascent
- Learning rate strategy
- Neural network loss
- Minimizing the loss by gradient descent
- Mini-batch/stochastic gradient descent
- Summary

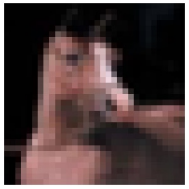
Outline

- Quality of neural network
- Maximizing the quality by gradient ascent
- Learning rate strategy
- Neural network loss
- Minimizing the loss by gradient descent
- Mini-batch/stochastic gradient descent
- Summary

Neural network quality

- Quality of neural networks :
  - Evaluate the confidence of the net to predict the correct class over the whole training set.

- First, assume for simplicity that the training set consists of
  - 4 classes : Horse, Cat, Dog, Deer
  - 5 labelled pictures



HORSE

CAT

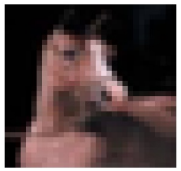
DOG

DEER

DOG

Neural network quality

Training set



HORSE

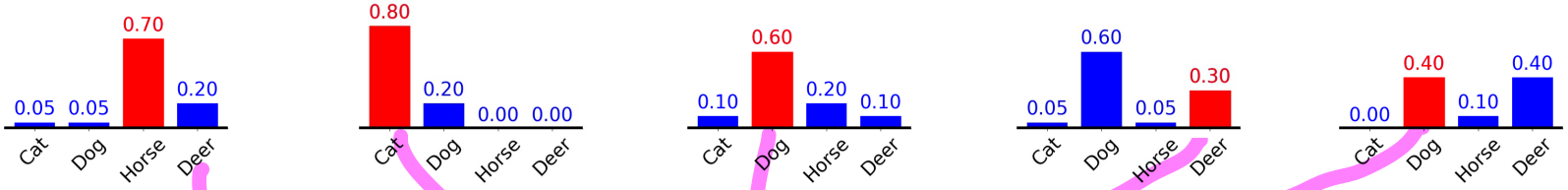
CAT

DOG

DEER

DOG

Network prediction



Predicted probability
of the ground truth class

Quality of the network

why do we use this ? why not arithmetic mean ?

$$Q_{\text{mean}} = (0.70 \times 0.80 \times 0.60 \times 0.30 \times 0.40)^{1/5} = 0.53$$

Geometric mean

often used when dealing with probabilities

In average, the network gives a 53% confidence score to the correct class

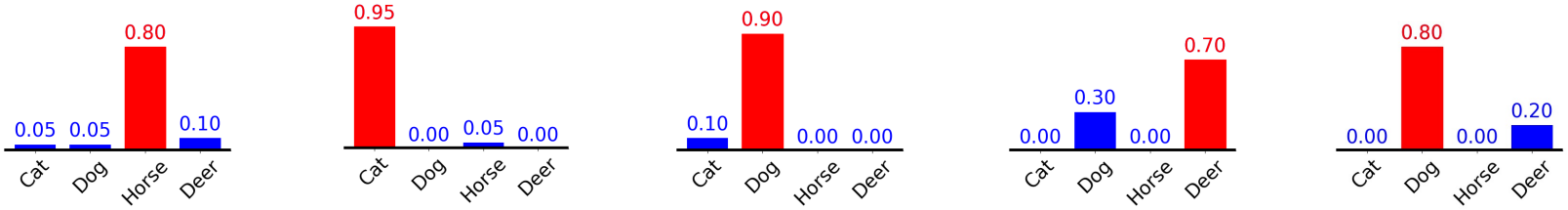
But often, we interact with the log instead

Good NN quality

Training set

HORSE	CAT	DOG	DEER	DOG

Network prediction



Quality of the network

$$Q_{\text{mean}} = (0.80 \times 0.95 \times 0.90 \times 0.70 \times 0.80)^{1/5} = 0.83$$

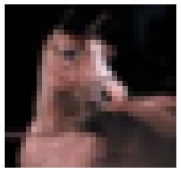
In average, the network gives a 83% confidence score to the correct class

Optimal classification: $Q_{\text{mean}} \approx 1.00$

Read: Universal Approx. Theorem.

Bad NN quality

Training set



HORSE

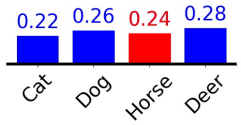
CAT

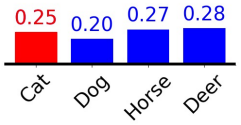
DOG

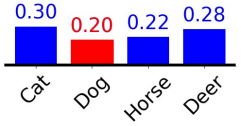
DEER

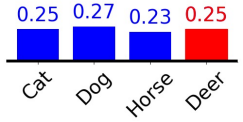
DOG

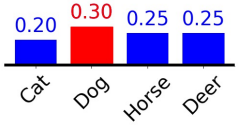
Network prediction











Quality of the network

$$Q_{\text{mean}} = (0.24 \times 0.25 \times 0.20 \times 0.25 \times 0.30)^{1/5} = 0.24$$

In average, the network gives a 24% confidence score to the correct class

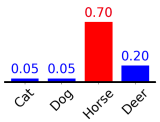
Random classification: $Q_{\text{mean}} \approx 0.25$ $\approx 1/K$

Neural network quality

The ith picture of the training set

The class index of the ith picture

Network output for the ith picture

	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$
$\mathbf{x}^{(i)}$					
$\mathrm{cl}(i)$	3	1	2	4	2
$\mathbf{p}^{(i)}$	 $= \begin{bmatrix} 0.05 \\ 0.05 \\ 0.70 \\ 0.20 \end{bmatrix}$	$\begin{bmatrix} 0.80 \\ 0.20 \\ 0.00 \\ 0.00 \end{bmatrix}$	$\begin{bmatrix} 0.05 \\ 0.60 \\ 0.20 \\ 0.10 \end{bmatrix}$	$\begin{bmatrix} 0.05 \\ 0.60 \\ 0.05 \\ 0.30 \end{bmatrix}$	$\begin{bmatrix} 0.00 \\ 0.40 \\ 0.10 \\ 0.40 \end{bmatrix}$

$$
\begin{aligned}
Q_{\text{mean}} &= (0.70 \times 0.80 \times 0.60 \times 0.30 \times 0.40)^{1/5} \\
&= \Big((\text{entry cl}(1) \text{ of } \mathbf{p}^{(1)}) \times (\text{entry cl}(2) \text{ of } \mathbf{p}^{(2)}) \times \cdots \times (\text{entry cl}(5) \text{ of } \mathbf{p}^{(5)}) \Big)^{1/5} \\
&= \left(\prod_{i=1}^{5} \text{entry cl}(i) \text{ of vector } \mathbf{p}^{(i)} \right)^{1/5}
\end{aligned}
$$

Ground truth class of datapoint i

Neural network quality

- With a training set containing N labeled pictures :

The class index of the ith picture (integer)

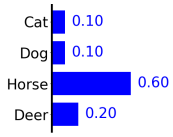
$$Q_{\text{mean}} = \left(\prod_{i=1}^{N} \underbrace{\text{entry cl}(i) \text{ of vector } \mathbf{p}^{(i)}}_{} \right)^{1/N}$$

Network output for the ith picture $\mathbf{x}^{(i)}$:

$$\mathbf{p}^{(i)} = \text{Softmax}(W\mathbf{x}^{(i)} + \mathbf{b})$$

How confident the network is that the ith picture belongs to the class cl(i)

Vector with 4 entries

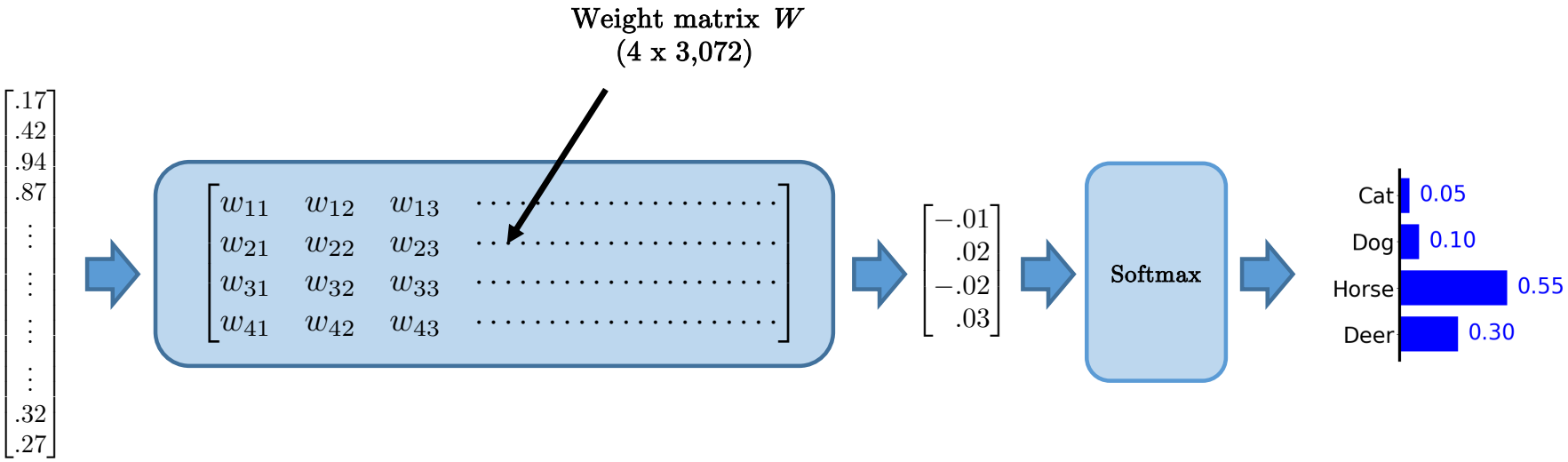


Outline

- Quality of neural network
- **Maximizing the quality by gradient ascent**
- Learning rate strategy
- Neural network loss
- Minimizing the loss by gradient descent
- Mini-batch/stochastic gradient descent
- Summary

Maximizing the quality of neural networks

- Learning = Maximizing the quality Q_{mean} of the network over its parameters W.
- Maximizing Q_{mean} will be done by gradient ascent. $\rightarrow \frac{dQ_{mean}}{dW}$
- Let us consider a vanilla one-layer NN :



Quality function

- The weight matrix W contains the 4 templates of size 3,072 :

$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & \cdots\cdots\cdots\cdots \\ w_{21} & w_{22} & w_{23} & \cdots\cdots\cdots\cdots \\ w_{31} & w_{32} & w_{33} & \cdots\cdots\cdots\cdots \\ w_{41} & w_{42} & w_{43} & \cdots\cdots\cdots\cdots \end{bmatrix}$$

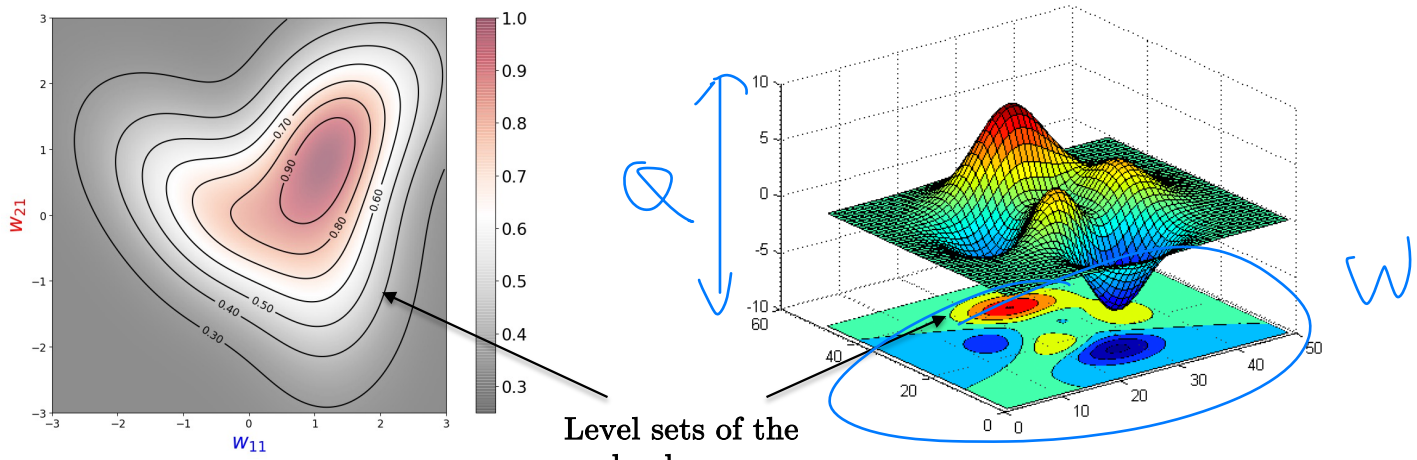
- The Quality Q_{mean} is a function of $\underbrace{\text{12,288}}_{\text{4 x 3,072}}$ parameters/weights :

$$\underbrace{Q_{\text{mean}}(w_{11}, w_{12}, w_{13}, \ldots \;;\; w_{21}, w_{22}, w_{23}, \ldots \;;\; w_{31}, w_{32}, w_{33}, \ldots \;;\; w_{41}, w_{42}, w_{43}, \ldots)}_{Q_{\text{mean}}(W)}$$

Landscape of quality function

- For each configuration W of the network, there exists a value $Q_{mean}(W)$.
  - All the possible configurations W provide the landscape of Q_{mean}.
- As Q_{mean} has 12k variables, it is not possible to visualize this landscape.
- Let's assume that Q_{mean} has only 2 parameters for visualization purpose :

$Q_{\text{mean}}(w_{11}, w_{21})$



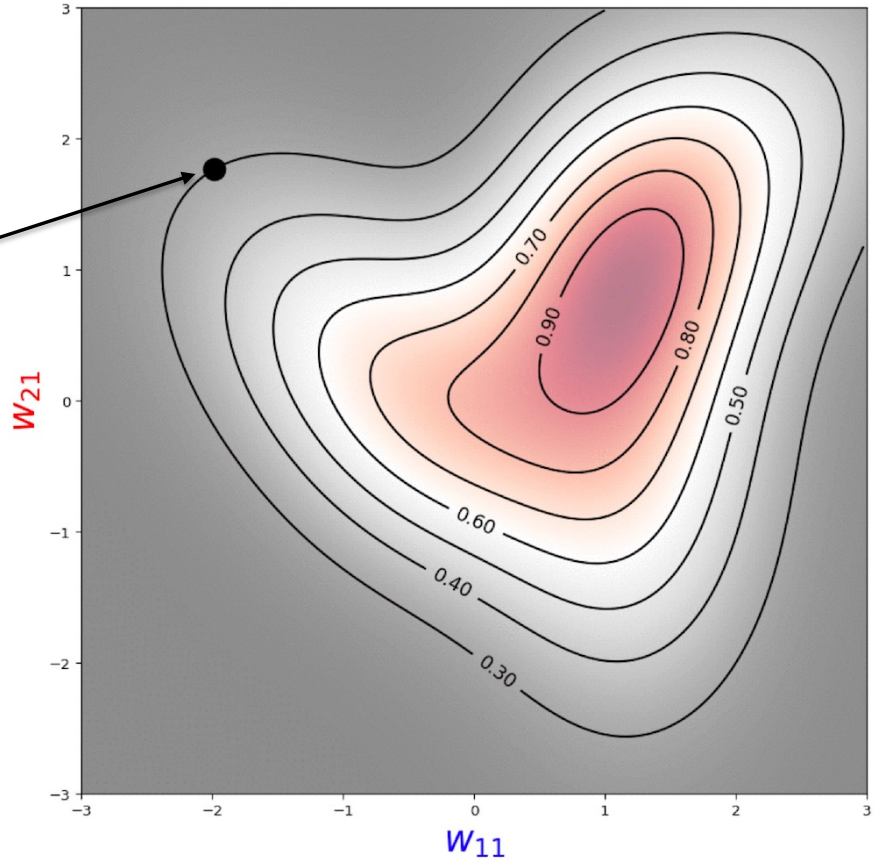
Level sets of the landscape

Maximizing the quality

- Gradient ascent algorithm:
  - Start randomly.
  - Move in the direction of the steepest ascent.
  - Stop when reaching the maximum.

Initialization

Start randomly



Network Quality:

$Q_{mean} =$ 0.25

Network Parameters:

$w_{11} =$ -2.20
$w_{12} =$ 2.20

Update scheme

- Gradient ascent:

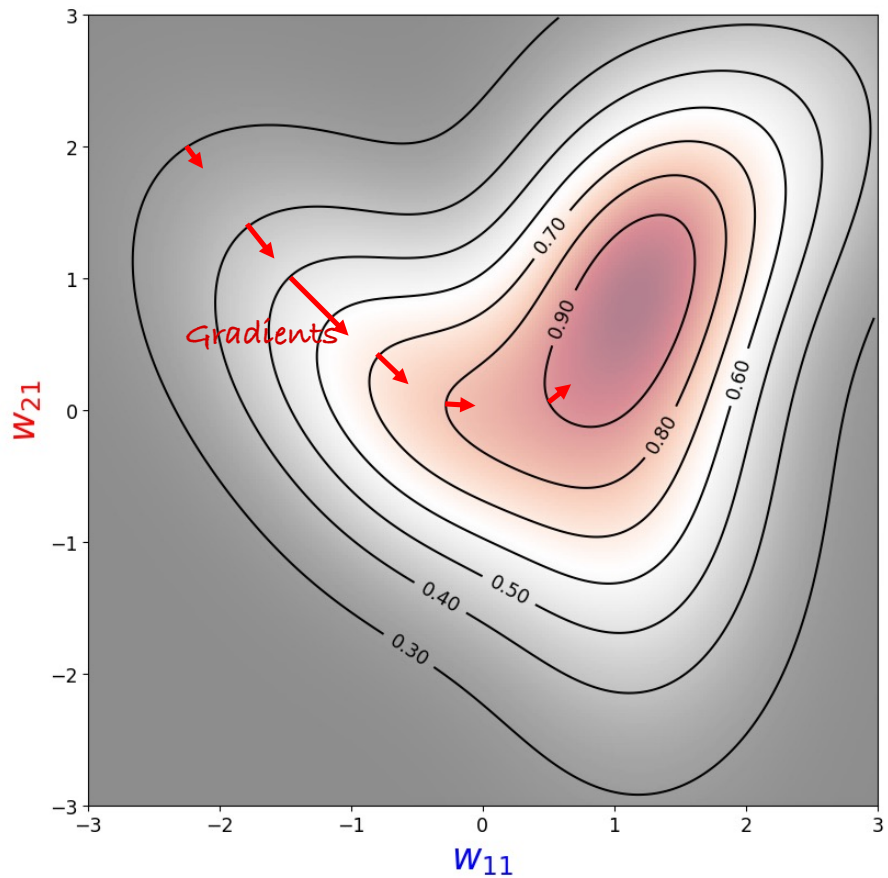
  Follow the gradient!

- The parameters are updated according to the equation:

$$\begin{bmatrix} w_{11} \\ \\ w_{21} \end{bmatrix} = \begin{bmatrix} w_{11} \\ \\ w_{21} \end{bmatrix} + \mathrm{lr} \begin{bmatrix} \frac{\partial Q}{\partial w_{11}} \\ \\ \frac{\partial Q}{\partial w_{21}} \end{bmatrix}$$

- So the displacements are proportional to the gradient.

- The gradient is:
  - Perpendicular to the level sets.
  - Its magnitude is proportional to the slope of the landscape.

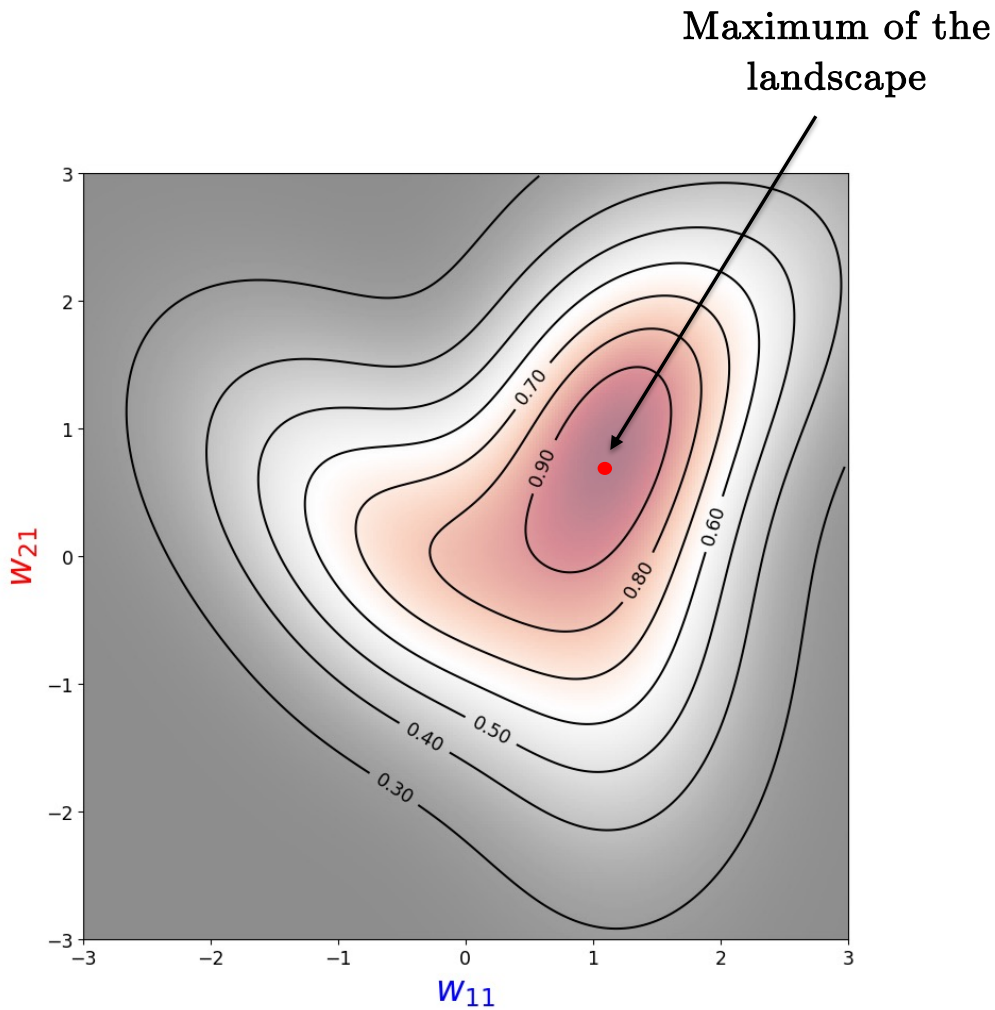


Maximum

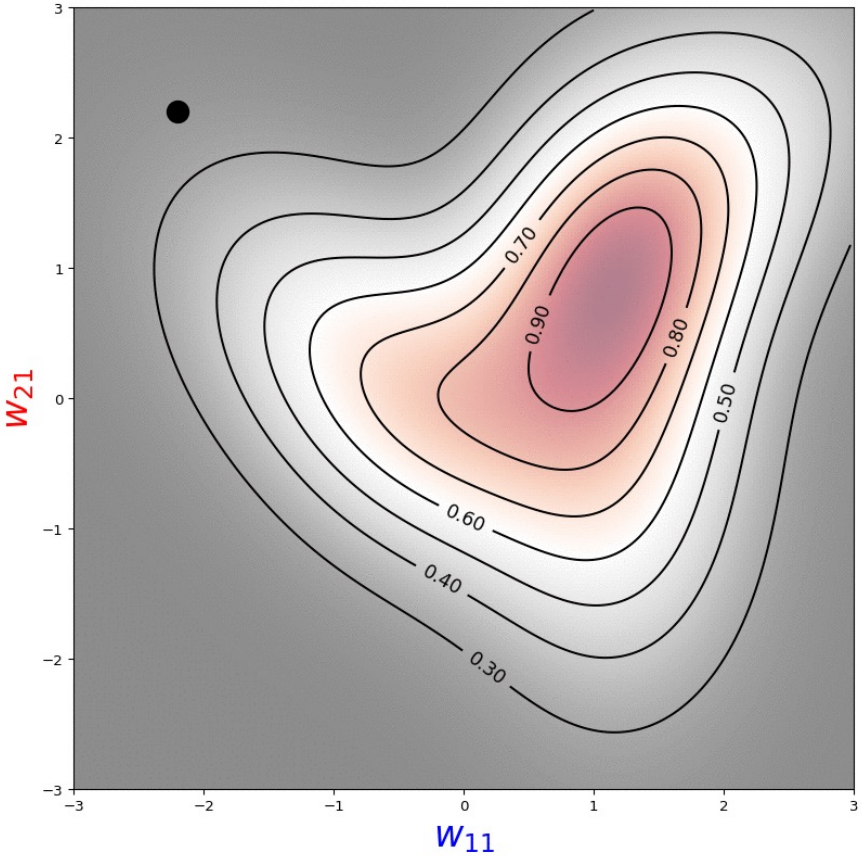
- At the maximum:
  - We reach the top of the mountain, which is perfectly flat.
  - We stop moving since the magnitude of the gradient is zero.

At the maximum location: $w_{11} = 1.1$, $w_{21} = 0.8$

We have: $$\begin{bmatrix} \frac{\partial Q}{\partial w_{11}} \\ \frac{\partial Q}{\partial w_{21}} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$



Gradient ascent in action



Network Quality:

$Q_{mean} =$ 0.25

Network Parameters:

$w_{11} =$ -2.20
$w_{12} =$ 2.20

High-dimensional quality function

- The previous Q_{mean} function has 2 variables.
  - The gradient vector ∇Q_{mean} has thus 2 elements.

$$Q_{\text{mean}}(w_{11}, w_{21}) \qquad \nabla Q_{\text{mean}} = \begin{bmatrix} \frac{\partial Q}{\partial w_{11}} \\ \\ \frac{\partial Q}{\partial w_{21}} \end{bmatrix}$$

- But the original quality Q_{mean} is a function of 12,288 parameters/weights.
  - The gradient vector ∇Q_{mean} has consequently 12,288 elements :

$$Q_{\text{mean}}(w_{11}, w_{12}, w_{13}, \ldots \; ; \; w_{21}, w_{22}, w_{23}, \ldots \; ; \; w_{31}, w_{32}, w_{33}, \ldots \; ; \; w_{41}, w_{42}, w_{43}, \ldots) \qquad \nabla Q_{\text{mean}} = \begin{bmatrix} \partial Q/\partial w_{11} \\ \partial Q/\partial w_{12} \\ \partial Q/\partial w_{13} \\ \vdots \\ \partial Q/\partial w_{21} \\ \partial Q/\partial w_{22} \\ \partial Q/\partial w_{23} \\ \vdots \\ \partial Q/\partial w_{31} \\ \partial Q/\partial w_{32} \\ \partial Q/\partial w_{33} \\ \vdots \\ \partial Q/\partial w_{41} \\ \partial Q/\partial w_{42} \\ \partial Q/\partial w_{43} \\ \vdots \end{bmatrix}$$

High-dimensional update formula

$$w_{ij} = w_{ij} + \mathrm{lr}\,\frac{\partial Q}{\partial w_{ij}} \quad \Longrightarrow \quad \begin{bmatrix} w_{11} \\ w_{12} \\ w_{13} \\ \vdots \\ w_{21} \\ w_{22} \\ w_{23} \\ \vdots \\ w_{31} \\ w_{32} \\ w_{33} \\ \vdots \\ w_{41} \\ w_{42} \\ w_{43} \\ \vdots \end{bmatrix} = \begin{bmatrix} w_{11} \\ w_{12} \\ w_{13} \\ \vdots \\ w_{21} \\ w_{22} \\ w_{23} \\ \vdots \\ w_{31} \\ w_{32} \\ w_{33} \\ \vdots \\ w_{41} \\ w_{42} \\ w_{43} \\ \vdots \end{bmatrix} + \mathrm{lr} \begin{bmatrix} \partial Q/\partial w_{11} \\ \partial Q/\partial w_{12} \\ \partial Q/\partial w_{13} \\ \vdots \\ \partial Q/\partial w_{21} \\ \partial Q/\partial w_{22} \\ \partial Q/\partial w_{23} \\ \vdots \\ \partial Q/\partial w_{31} \\ \partial Q/\partial w_{32} \\ \partial Q/\partial w_{33} \\ \vdots \\ \partial Q/\partial w_{41} \\ \partial Q/\partial w_{42} \\ \partial Q/\partial w_{43} \\ \vdots \end{bmatrix}$$

Matrix representation

- **Scalar form** of the gradient ascent equation:

$$w_{ij} \quad = \quad w_{ij} \quad + \quad \mathrm{lr} \; \frac{\partial Q}{\partial w_{ij}}$$

- **Matrix form** of the gradient ascent equation:

$$\underbrace{\begin{bmatrix} w_{11} & w_{12} & w_{13} & \cdots\cdots \\ w_{21} & w_{22} & w_{23} & \cdots\cdots \\ w_{31} & w_{32} & w_{33} & \cdots\cdots \\ w_{41} & w_{42} & w_{43} & \cdots\cdots \end{bmatrix}}_{W} =$$

$$\underbrace{\begin{bmatrix} w_{11} & w_{12} & w_{13} & \cdots\cdots \\ w_{21} & w_{22} & w_{23} & \cdots\cdots \\ w_{31} & w_{32} & w_{33} & \cdots\cdots \\ w_{41} & w_{42} & w_{43} & \cdots\cdots \end{bmatrix}}_{W} + \mathrm{lr} \underbrace{\begin{bmatrix} \partial Q/\partial w_{11} & \partial Q/\partial w_{12} & \partial Q/\partial w_{13} & \cdots\cdots \\ \partial Q/\partial w_{21} & \partial Q/\partial w_{22} & \partial Q/\partial w_{23} & \cdots\cdots \\ \partial Q/\partial w_{31} & \partial Q/\partial w_{32} & \partial Q/\partial w_{33} & \cdots\cdots \\ \partial Q/\partial w_{41} & \partial Q/\partial w_{42} & \partial Q/\partial w_{43} & \cdots\cdots \end{bmatrix}}_{\text{Gradient reshaped as a matrix } \frac{\partial Q}{\partial W}}$$

$$w \leftarrow w + \frac{\partial Q}{\partial w}$$

Outline

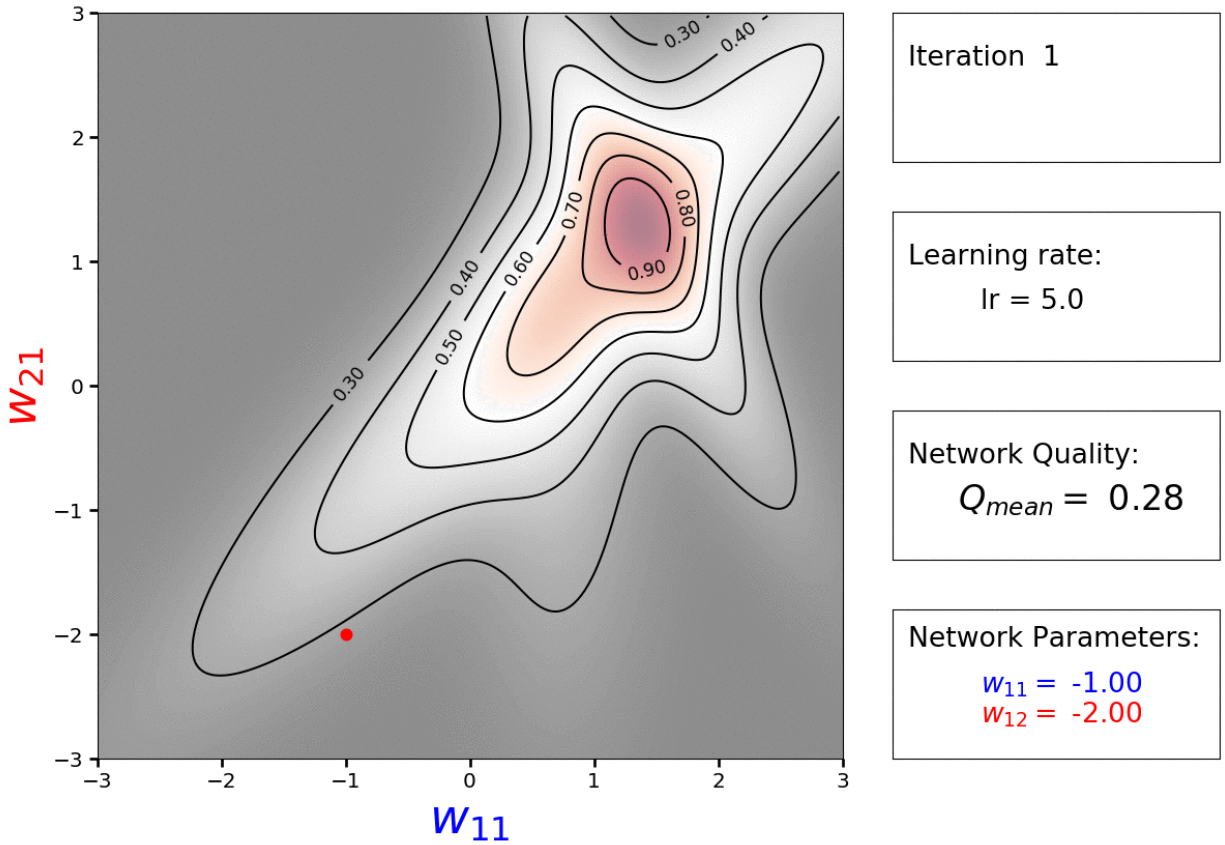
- Quality of neural network
- Maximizing the quality by gradient ascent
- **Learning rate strategy**
- Neural network loss
- Minimizing the loss by gradient descent
- Mini-batch/stochastic gradient descent
- Summary

Learning rate

- It is important to carefully choose the learning rate:
  - If the learning rate is too big, the algorithm will diverge.
  - If the learning rate is too small, it will take too long to reach the maximum.

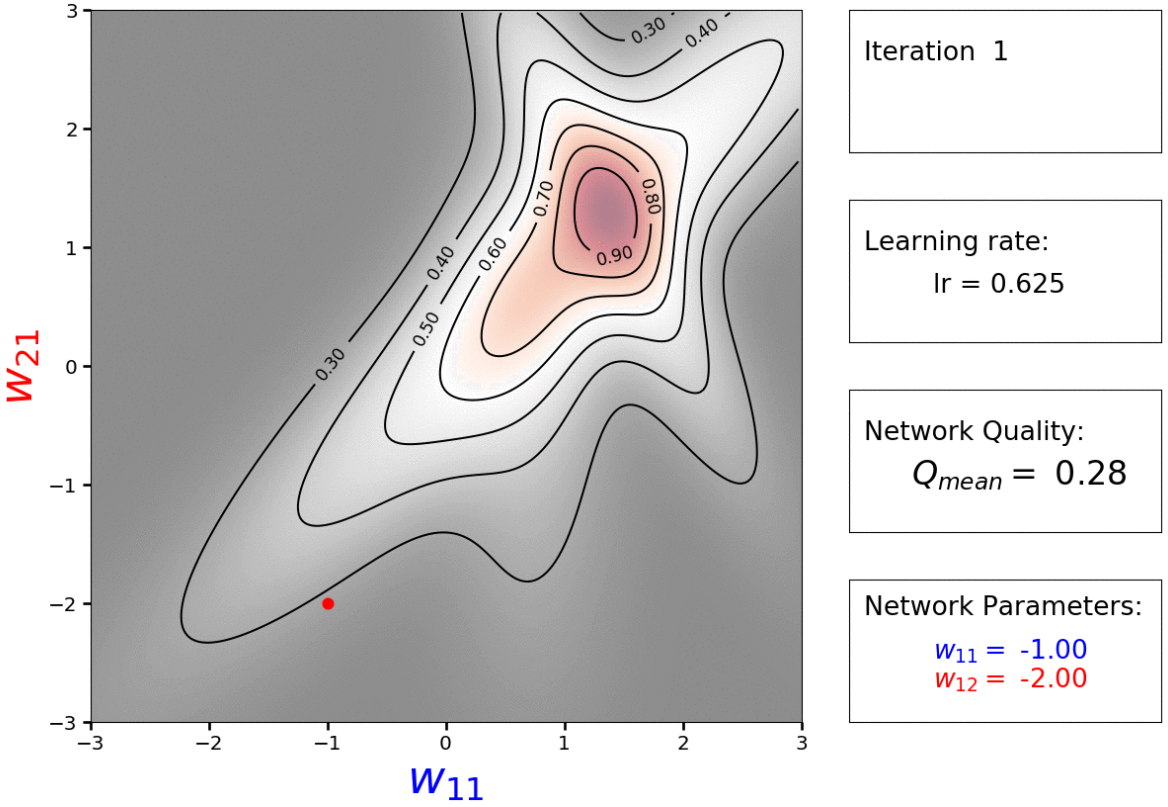
Learning rate too large

- $\mathrm{lr} = 5.0$



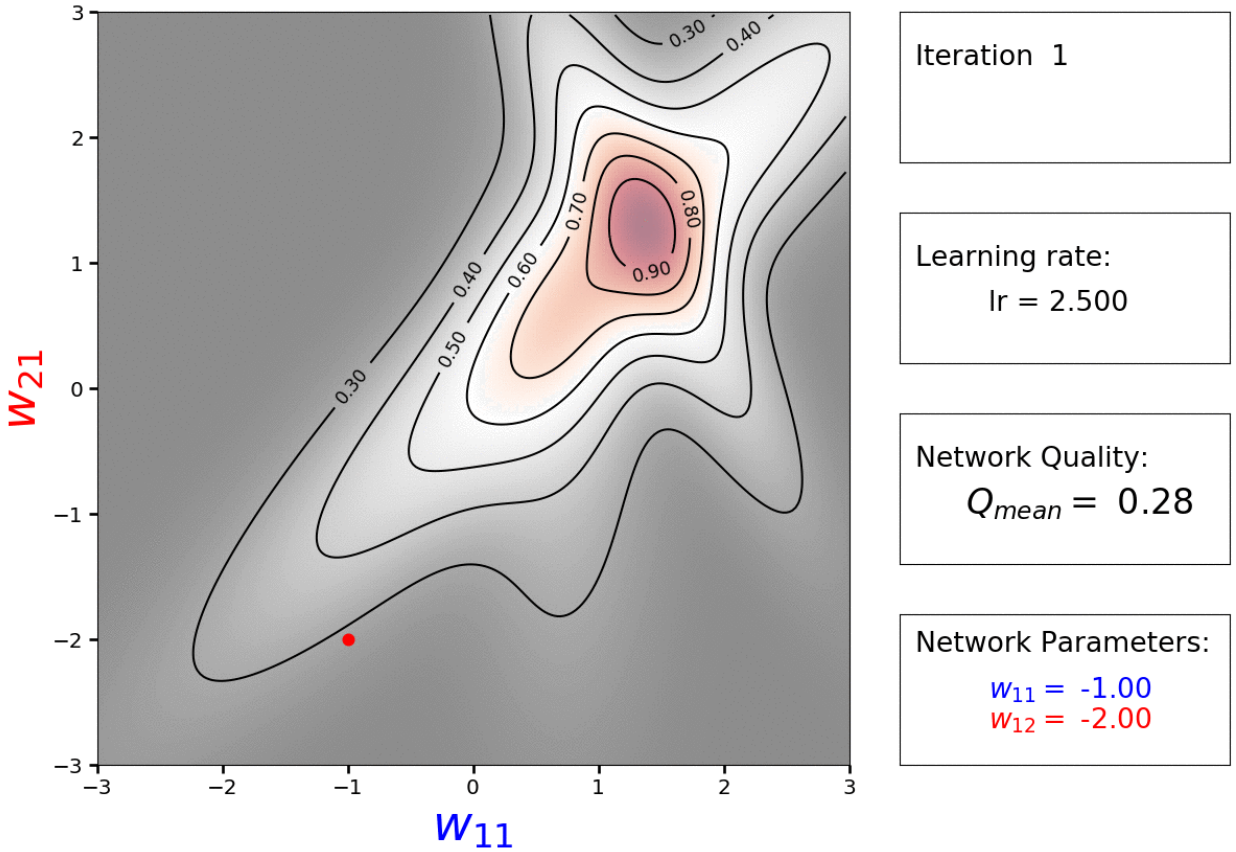
Learning rate too small

- lr=0.625
  - 8 times smaller than lr=5.0



Learning rate schedule

- Best way to go !
  - Make a learning rate schedule :
    - Start with lr=2.5.
    - Divide by 2 every 10 iterations.



Outline

- Quality of neural network
- Maximizing the quality by gradient ascent
- Learning rate strategy
- Neural network loss
- Minimizing the loss by gradient descent
- Mini-batch/stochastic gradient descent
- Summary

Optimizing product is slow

- Quality function is a product of functions:

$$Q_{\text{mean}} = \left(\prod_{i=1}^{N} \text{entry cl}(i) \text{ of vector } \mathbf{p}^{(i)} \right)^{1/N}$$

- Gradient of a product is computationally expensive (e.g. derivate of a product of 50,000 terms)

$$w_{ij} \quad = \quad w_{ij} \quad + \quad \text{lr} \quad \frac{\partial Q}{\partial w_{ij}}$$

Also, numerically unstable
-> potentially leads to
vanishing/exploding gradients

- It is much cheaper to compute the gradient of a sum.
  - How? The log function over a product provides a sum.

New loss

- **News loss = - log of neural network quality**

$$\mathcal{L} \underset{\text{definition}}{=} -\log Q_{\text{mean}} = -\log\left(\prod_{i=1}^{N} \text{entry cl}(i) \text{ of vector } \mathbf{p}^{(i)})\right)^{1/N}$$

$$= -\frac{1}{N} \log\left(\prod_{i=1}^{N} \text{entry cl}(i) \text{ of vector } \mathbf{p}^{(i)}\right)$$

$$= -\frac{1}{N} \sum_{i=1}^{N} \log\left(\text{entry cl}(i) \text{ of vector } \mathbf{p}^{(i)}\right)$$

Cross-entropy loss

This function is much easier to differentiate.

$$= -\frac{1}{N} \sum_{i=1}^{N} \sum_{c}^{C} y_c \log(\hat{y}_c)$$

$= 0$ if datapoint i not class c

Cross-entropy loss

- The new loss is called cross-entropy loss.
- Cross-entropy is a standard function in information theory.
- Cross-entropy is a function of W :

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} -\log\left(\text{entry cl}(i) \text{ of vector } \mathbf{p}^{(i)}\right)$$

?

$$\log\left(\text{entry 2 of vector } \begin{bmatrix} p_1^{(i)} \\ p_2^{(i)} \\ p_3^{(i)} \\ p_4^{(i)} \end{bmatrix}\right) = \log\left(p_2^{(i)}\right)$$

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} -\text{entry cl}(i) \text{ of vector } \log\left(\mathbf{p}^{(i)}\right)$$

$$\mathcal{L}(w_{11}, w_{12}, \ldots) = \frac{1}{N} \sum_{i=1}^{N} -\text{entry cl}(i) \text{ of vector } \text{LogSoftmax}(W\mathbf{x}^{(i)})$$

Function of
12,288 parameters.

Cross-entropy loss

$$\mathcal{L}(w_{11}, w_{12}, \ldots) = \overbrace{\frac{1}{N} \sum_{i=1}^{N} \underbrace{-\text{entry cl}(i) \text{ of vector } \log\left(\mathbf{p}^{(i)}\right)}_{\ell^{(i)}(w_{11}, w_{12}, \ldots)}}^{\text{Average loss over all data points}}$$

Loss of the ith data point =

- log of the probability given to the correct class

Step 1 — What cross-entropy actually is

For logits z and true class y:

$$\mathcal{L} = -\log\left(\frac{e^{z_y}}{\sum_j e^{z_j}}\right)$$

That's it. That's the whole story.

This can be written as:

$$\mathcal{L} = -\log(\text{softmax}(z)_y)$$

Step 2 — Break it into two pieces

Notice this can be separated into:

1. Compute log-probabilities

$$\log p = \log \text{softmax}(z)$$

2. Pick the correct class and negate it

$$-\log p_y$$

That second step is exactly **Negative Log-Likelihood (NLL)**.

So:

$$\text{CrossEntropy} = \text{LogSoftmax} + \text{NLL}$$

Cross-entropy loss

	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$
$\mathbf{x}^{(i)}$					
$\mathrm{cl}^{(i)}$	3	1	2	4	2
$\mathbf{p}^{(i)}$	$\begin{bmatrix}0.05\\0.05\\0.70\\0.20\end{bmatrix}$	$\begin{bmatrix}0.80\\0.20\\0.00\\0.00\end{bmatrix}$	$\begin{bmatrix}0.05\\0.60\\0.20\\0.10\end{bmatrix}$	$\begin{bmatrix}0.05\\0.60\\0.05\\0.30\end{bmatrix}$	$\begin{bmatrix}0.00\\0.40\\0.10\\0.40\end{bmatrix}$

$$\text{Average Loss} = \frac{1}{5}\sum_{1}^{5} -\text{entry cl}(i) \text{ of vector } \log\left(\mathbf{p}^{(i)}\right)$$

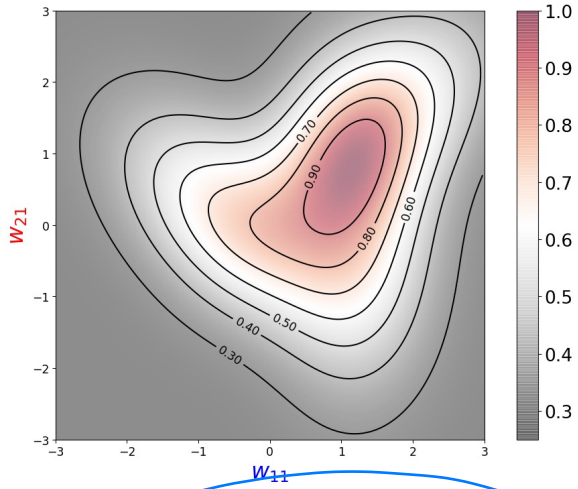
$$= \frac{1}{5}\Big(-\log(0.70) - \log(0.80) - \log(0.60) - \log(0.30) - \log(0.40)\Big)$$

$$= \frac{1}{5}\Big(\ 0.37 + 0.22 + 0.51 + 1.20 + 0.92\ \Big) = 0.68$$

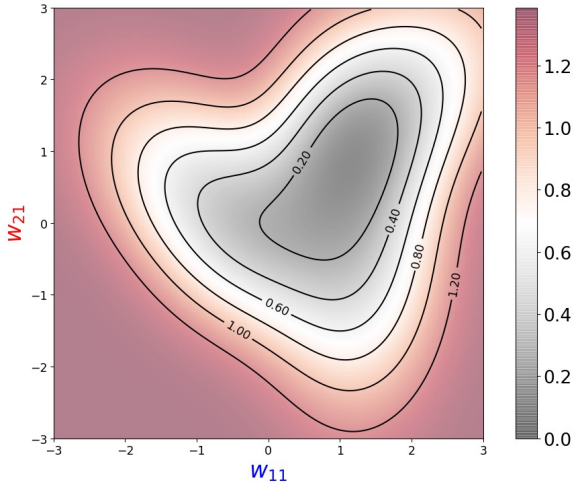
Will be =0 for a perfect network, as all terms in the summation are log(1)=0

Range of values

$$Q_{\text{mean}}$$



$$\mathcal{L} := -\log(Q_{\text{mean}})$$



$$0.25 \leq Q_{\text{mean}} \leq 1$$

Assuming 4 class labels, then by random chance, a poor network will be correct 25% of the time

By def$^{\underline{n}}$: $Q := \left(\prod_{i=1}^{N} \mathcal{L}(i)\right)^{1/N}$

$\therefore$ For a random model that outputs 1/4 for every class, we have $Q = ((0.25)\hat{}\,N)\hat{}\,1/N = 0.25$

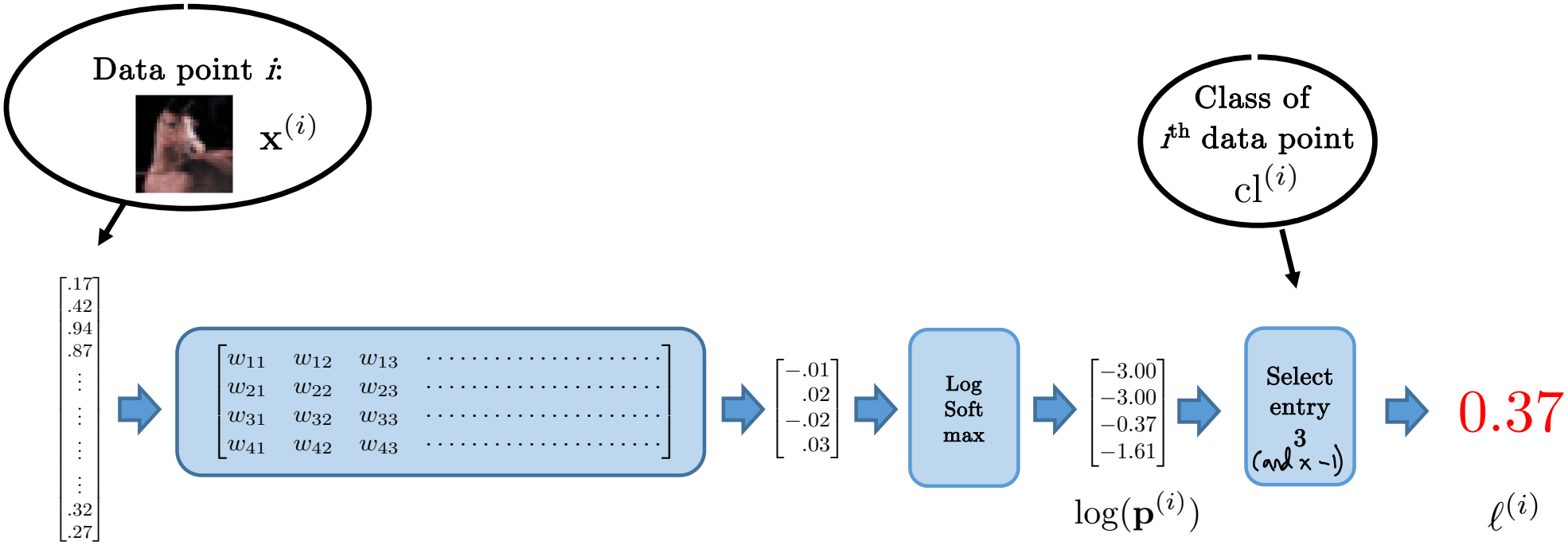
$$0 \leq \mathcal{L} \leq 1.4$$

$-\log(1)$

$-\log(0.25)$

We will want L to be as close to 0 as possible.

Representing the network and the loss



Input:
An image and
its class

Output:
The loss of
i^{th} data point

Outline

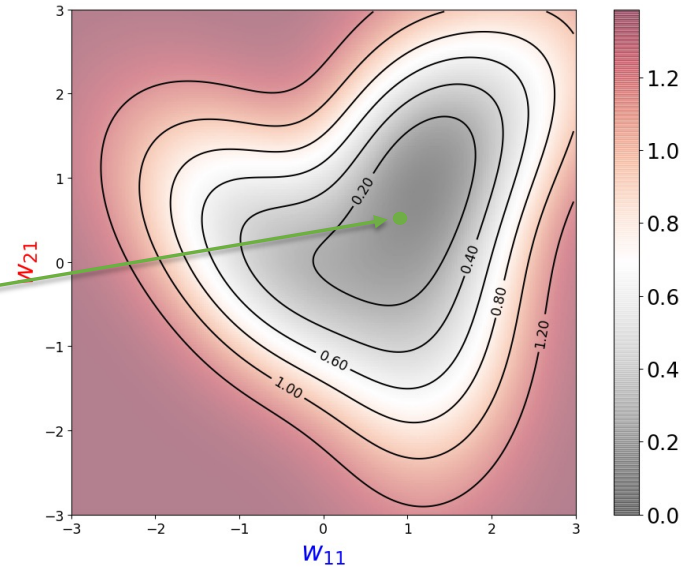
- Quality of neural network
- Maximizing the quality by gradient ascent
- Learning rate strategy
- Neural network loss
- **Minimizing the loss by gradient descent**
- Mini-batch/stochastic gradient descent
- Summary

Loss

- The total loss over the whole training set is:

$$\mathcal{L}(w_{11}, w_{12}, \ldots) \quad = \quad \frac{1}{N} \sum_{i=1}^{N} \quad \ell^{(i)}(w_{11}, w_{12}, \ldots)$$

We want to tune the internal parameters of the network in order to minimize the total loss (ideally make it equal to 0)



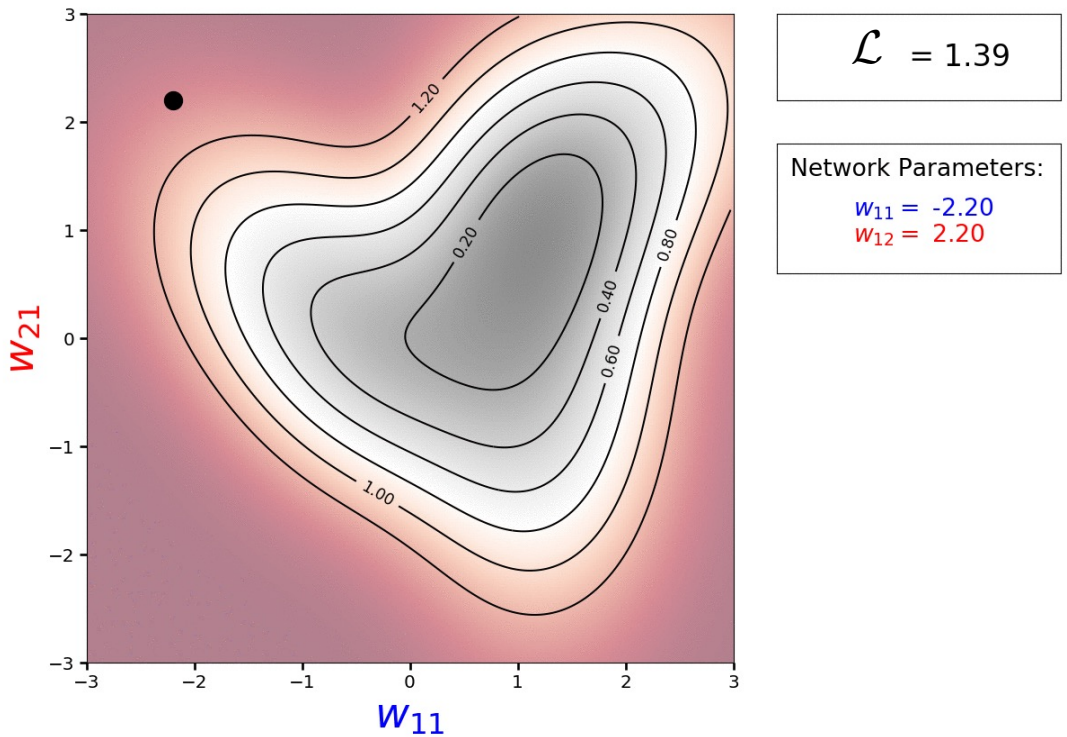
Loss minimization

- The total loss is minimized by gradient descent:

$$w_{rs} = w_{rs} - \mathrm{lr}\,\frac{\partial \mathcal{L}}{\partial w_{rs}}$$

Minimization

$$\begin{bmatrix} w_{11} \\ w_{12} \end{bmatrix} = \begin{bmatrix} w_{11} \\ w_{12} \end{bmatrix} - \mathrm{lr} \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial w_{11}} \\ \frac{\partial \mathcal{L}}{\partial w_{12}} \end{bmatrix}$$



Matrix representation

- **Matrix form** of the gradient descent equation:

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & \cdots\cdots\cdots \\ w_{21} & w_{22} & w_{23} & \cdots\cdots\cdots \\ w_{31} & w_{32} & w_{33} & \cdots\cdots\cdots \\ w_{41} & w_{42} & w_{43} & \cdots\cdots\cdots \end{bmatrix} =$$

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & \cdots\cdots\cdots \\ w_{21} & w_{22} & w_{23} & \cdots\cdots\cdots \\ w_{31} & w_{32} & w_{33} & \cdots\cdots\cdots \\ w_{41} & w_{42} & w_{43} & \cdots\cdots\cdots \end{bmatrix} - \mathrm{lr} \underbrace{\begin{bmatrix} \partial\mathcal{L}/\partial w_{11} & \partial\mathcal{L}/\partial w_{12} & \partial\mathcal{L}/\partial w_{13} & \cdots\cdots\cdots \\ \partial\mathcal{L}/\partial w_{21} & \partial\mathcal{L}/\partial w_{22} & \partial\mathcal{L}/\partial w_{23} & \cdots\cdots\cdots \\ \partial\mathcal{L}/\partial w_{31} & \partial\mathcal{L}/\partial w_{32} & \partial\mathcal{L}/\partial w_{33} & \cdots\cdots\cdots \\ \partial\mathcal{L}/\partial w_{41} & \partial\mathcal{L}/\partial w_{42} & \partial\mathcal{L}/\partial w_{43} & \cdots\cdots\cdots \end{bmatrix}}$$

$$W = W - \mathrm{lr}\,\frac{\partial\mathcal{L}}{\partial W}$$

Outline

- Quality of neural network
- Maximizing the quality by gradient ascent
- Learning rate strategy
- Neural network loss
- Minimizing the loss by gradient descent
- **Mini-batch/stochastic gradient descent**
- Summary

Gradient descent

- Loss:

$$\mathcal{L}(w_{11}, w_{12}, \ldots) = \frac{1}{N} \sum_{i=1}^{N} \ell^{(i)}(w_{11}, w_{12}, \ldots)$$

(Average) loss

Loss of data point $x^{(i)}$

- (Full batch) gradient descent:

$$W = W - \mathrm{lr}\, \frac{\partial \mathcal{L}}{\partial W}$$

$$\frac{\partial \mathcal{L}}{\partial W} = \underbrace{\frac{1}{N} \sum_{i=1}^{N} \frac{\partial \ell^{(i)}}{\partial W}}$$

The gradient matrix $\frac{\partial \mathcal{L}}{\partial W}$ is the average of 50,000 gradient matrices $\frac{\partial \ell^{(i)}}{\partial W}$!

Mini-batch gradient descent

- We can be more efficient by computing the average of only m gradient matrices.
- Mini-batch gradient descent:

$$W = W - \text{lr}\,\frac{\partial \mathcal{L}}{\partial W}$$

$$\frac{\partial \mathcal{L}}{\partial W} \approx \underbrace{\frac{1}{m}\sum_{i \in B} \frac{\partial \ell^{(i)}}{\partial W}}$$

Random approximation of the gradient $\frac{\partial \mathcal{L}}{\partial W}$ using a batch B of m indices chosen at random:

$$B = \{i_1, i_2, i_3, \ldots, i_m\}$$

Mini-batch gradient descent

- Advantages of mini-batch gradient descent:
  - The gradient is quickly computed: No need to go through the whole training set (which can be huge) to update the weights.
  - Datasets can be redundant : Many data are similar, a subset of the data will give almost the same gradient than using all data points.

$$W = W - \text{lr}\,\frac{\partial \mathcal{L}}{\partial W} \qquad \begin{cases} \dfrac{\partial \mathcal{L}}{\partial W} = \dfrac{1}{N}\displaystyle\sum_{i=1}^{N} \dfrac{\partial \ell^{(i)}}{\partial W} & \text{Full batch gradient} \\[2ex] \dfrac{\partial \mathcal{L}}{\partial W} \approx \dfrac{1}{m}\displaystyle\sum_{i \in B} \dfrac{\partial \ell^{(i)}}{\partial W} & \text{Mini batch gradient} \end{cases}$$

Mini-batch gradient descent

- Batch size :
  - No batch (single data): batch size is 1. This is called stochastic gradient descent (SGD). It approximates the gradient by averaging over 1 data point. Best for generalization but not as fast as mini-batch GD.
  - Mini-batch: batch size is 200. This is called mini-batch GD. It approximates the gradient by averaging over 200 data point. Good for generalization and fastest GD.
  - Full batch (all training data): batch size is 60,000. This is called gradient descent (GD). It provides the exact gradient by averaging over all data points. Weak for generalization and slowest GD.

In-lecture question

- How does the choice of batch size in gradient descent affect a neural network's generalization ability and computational speed? Compare Stochastic Gradient Descent (SGD), Mini-batch Gradient Descent, and Full-batch Gradient Descent.
- In Slack #lectures
  - Identify the question and Reply in thread with a short response
- Answer : The choice of batch size involves a direct trade-off between computational efficiency and the network's ability to generalize to new data:

  1) Stochastic Gradient Descent (SGD): Uses a batch size of 1. It is considered the best for generalization because the high noise in the gradient updates helps the model avoid local minima, but it is not as fast as mini-batch methods due to the lack of parallelization.

  2) Mini-batch Gradient Descent: Typically uses a moderate batch size (e.g. 200). It is the fastest version of gradient descent because it balances the computational benefits of matrix operations with enough data variety for good generalization.

  3) Full-batch Gradient Descent: Uses all training data points (e.g. 60,000). While it provides the exact gradient, it is the slowest computationally and is considered weak for generalization.

Connection to template matching

- Mini-batch GD = template update formula ! (No proof given)
  - The mini-batch gradient of the cross-entropy loss turns out to be exactly the same formula for template matching. (No coincidence)

$$\frac{\partial \mathcal{L}^B}{\partial W} = -\frac{1}{m} EX^T$$

- The template update scheme reduces to Mini-batch GD. (See next slide)
- When we did template matching update, we actually minimized the cross-entropy loss by Mini-batch GD.

Connection to template matching

$$\begin{bmatrix} .17 & .12 \\ .42 & .32 \\ .94 & .74 \\ .87 & .65 \\ \vdots & \vdots \\ \vdots & \vdots \\ \vdots & \vdots \\ \vdots & \vdots \\ \vdots & \vdots \\ .32 & .42 \\ .27 & .18 \end{bmatrix}$$

$$\begin{bmatrix} -.91 & .75 & .76 & \cdots & -.83 & -.93 \\ .89 & -.75 & -.67 & \cdots & +.54 & .79 \\ -.83 & -.86 & .25 & \cdots & .64 & -.88 \\ -.73 & -.78 & -.89 & \cdots & .97 & -.56 \end{bmatrix}$$

$$\begin{bmatrix} .02 & -.01 \\ -.01 & .03 \\ .01 & .01 \\ .01 & -.02 \end{bmatrix}$$

Soft max

$$\begin{bmatrix} 0.30 & 0.25 \\ 0.20 & 0.30 \\ 0.20 & 0.26 \\ 0.25 & 0.19 \end{bmatrix}$$

$$\begin{bmatrix} 0.00 & 1.00 \\ 0.00 & 0.00 \\ 1.00 & 0.00 \\ 0.00 & 0.00 \end{bmatrix}$$

$$\begin{bmatrix} -0.30 & +0.75 \\ -0.20 & -0.30 \\ +0.80 & -0.26 \\ -0.25 & -0.19 \end{bmatrix}$$

$$\frac{1}{2} \begin{bmatrix} -0.30 & +0.75 \\ -0.20 & -0.30 \\ +0.80 & -0.26 \\ -0.25 & -0.19 \end{bmatrix} \begin{bmatrix} .17 & .42 & .94 & .87 & \cdots & \cdots & \cdots & \cdots & .32 & .27 \\ .12 & .32 & .74 & .65 & \cdots & \cdots & \cdots & \cdots & .42 & .18 \end{bmatrix}$$

$$W = W - \mathrm{lr} \frac{\partial \mathcal{L}^B}{\partial W}$$

$$\frac{\partial \mathcal{L}^B}{\partial W} = -\frac{1}{m} E X^T$$

Outline

- Quality of neural network
- Maximizing the quality by gradient ascent
- Learning rate strategy
- Neural network loss
- Minimizing the loss by gradient descent
- Mini-batch/stochastic gradient descent
- **Summary**

Summary

Cross-entropy loss : $$\mathcal{L}(w_{11}, w_{12}, \ldots) = \frac{1}{N} \sum_{i=1}^{N} \underbrace{\ell^{(i)}(w_{11}, w_{12}, \ldots)}_{\text{Each data point has a loss describing how well it has been classified by the network.}}$$

Gradient Descent : $$\frac{\partial \mathcal{L}}{\partial W} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \ell^{(i)}}{\partial W}$$

Mini-batch Gradient Descent : $$\frac{\partial \mathcal{L}^B}{\partial W} = \underbrace{\frac{1}{m} \sum_{i \in B} \frac{\partial \ell^{(i)}}{\partial W}}_{\text{Random approximation of the gradient using a batch } B}$$

$$B = \{i_1, i_2, i_3, \ldots, i_m\}$$

Doing Mini-batch GD is exactly the same thing than doing template matching with mini-batch.

Doing GD is exactly the same thing than doing template matching with full batch.

Lab 07

- Cross-entropy loss:

jupyter cross_entropy_demo Last Checkpoint: a few seconds ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Lab 01 : Cross-entropy loss -- demo

In [1]:
```
import torch
import torch.nn as nn
import utils
```

Make a Cross Entropy Criterion and call it mycrit

In [2]:
```
mycrit=nn.CrossEntropyLoss()
print(mycrit)
```

```
CrossEntropyLoss()
```

Make a batch of labels

In [3]:
```
labels=torch.LongTensor([2,3])

print(labels)
```

```
tensor([2, 3])
```

Make a batch of scores

jupyter cross_entropy_exercise Last Checkpoint: a few seconds ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Lab 01 : Cross-entropy loss -- exercise

In [ ]:
```
import torch
import torch.nn as nn
import utils
```

Make a Cross Entropy Criterion and call it criterion. The command is nn.CrossEntropyLoss().

In [ ]:
```
criterion=  # COMPLETE HERE

print(criterion)
```

Suppose that there only two classes (class 0 and class 1).

Suppose we have a batch of three data points:

$\mathbf{x}^{(0)}$ **belongs to class 0**

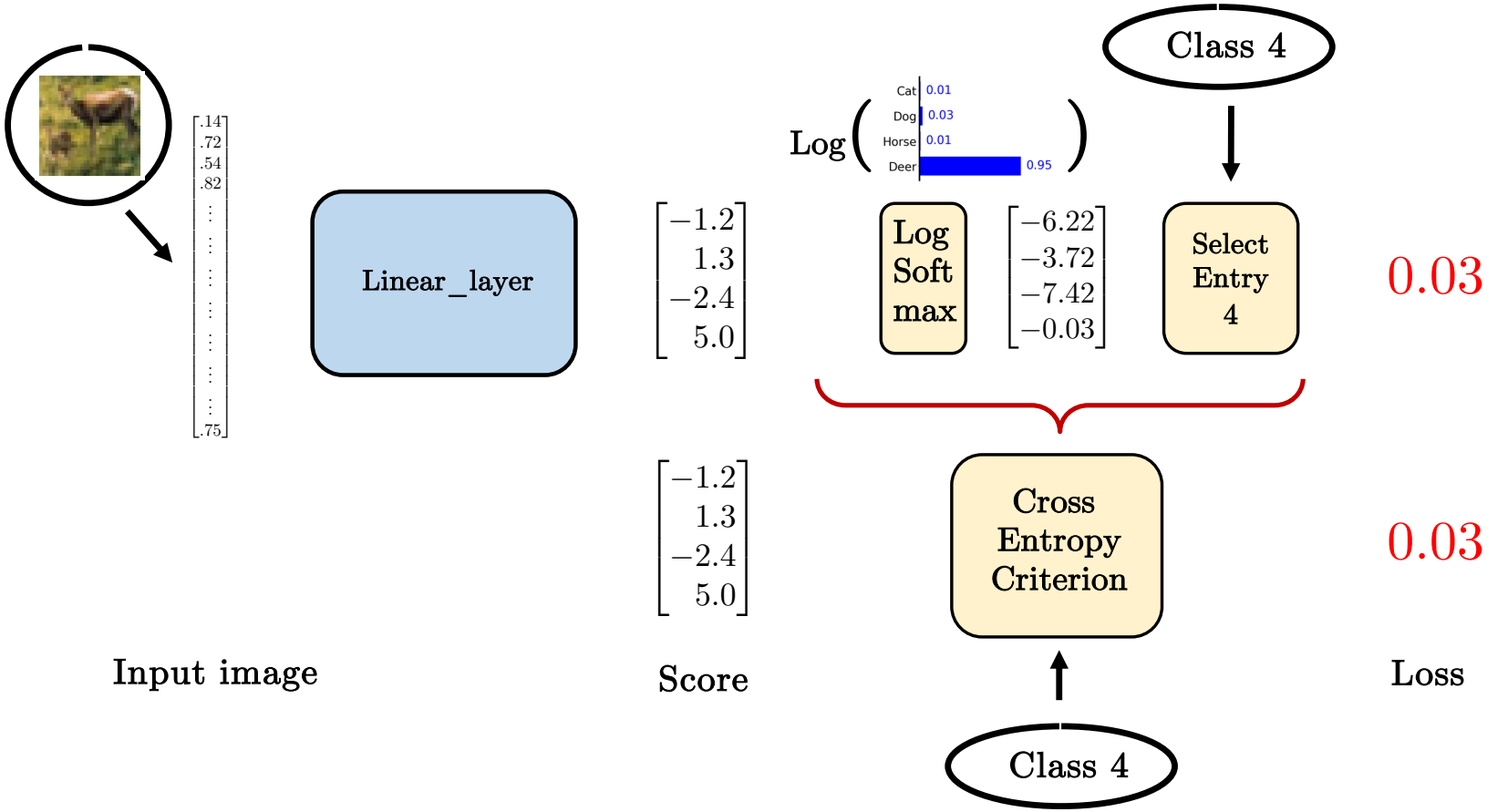
$\mathbf{x}^{(1)}$ **belongs to class 1**

$\mathbf{x}^{(2)}$ **belongs to class 1**

Put the labels of each of these point a LongTensor:

Lab 07

- **Function nn.CrossEntropyLoss() :**



Questions?

CS5242 : Neural Networks and Deep Learning

Lecture 4 : Multi-Layer Perceptron
Part 1 : Inference and Learning

Semester 2 2025/26

Xavier Bresson

https://twitter.com/xbresson

Department of Computer Science
National University of Singapore (NUS)

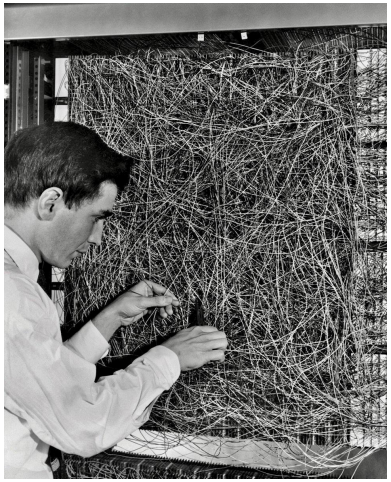
Outline

- Multi-Layer Perceptron
- Loss and gradient descent
- Backpropagation
- PyTorch implementation
- Training with epochs
- Monitoring the loss
- Test set evaluation
- Final code

Outline

- **Multi-Layer Perceptron**
- Loss and gradient descent
- Backpropagation
- PyTorch implementation
- Training with epochs
- Monitoring the loss
- Test set evaluation
- Final code

MLP



- Introduced in Frank Rosenblatt in 1957.

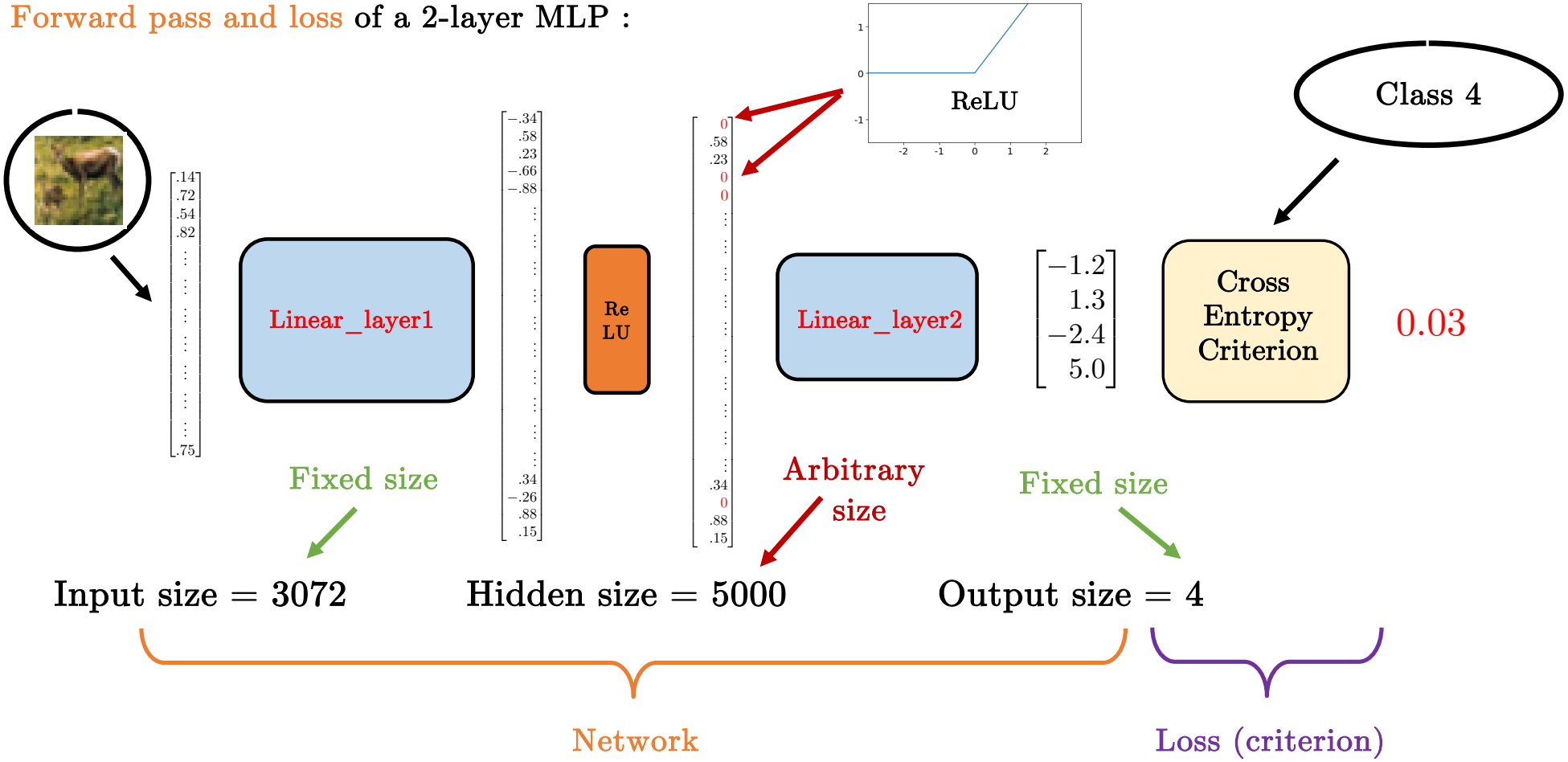
- What is a Multi-Layer Perceptron (MLP)?
  - A MLP is the simplest feedforward neural networks (no cycle connection).
  - A MLP is composed by a series of linear layers + non-linear activations.
  - A MLP is a.k.a. Fully Connected (FC) neural networks.

- Example: Two-layer MLP
  - Consider a two-layer network with only 4 classes:

    Cat, Dog, Horse, Deer

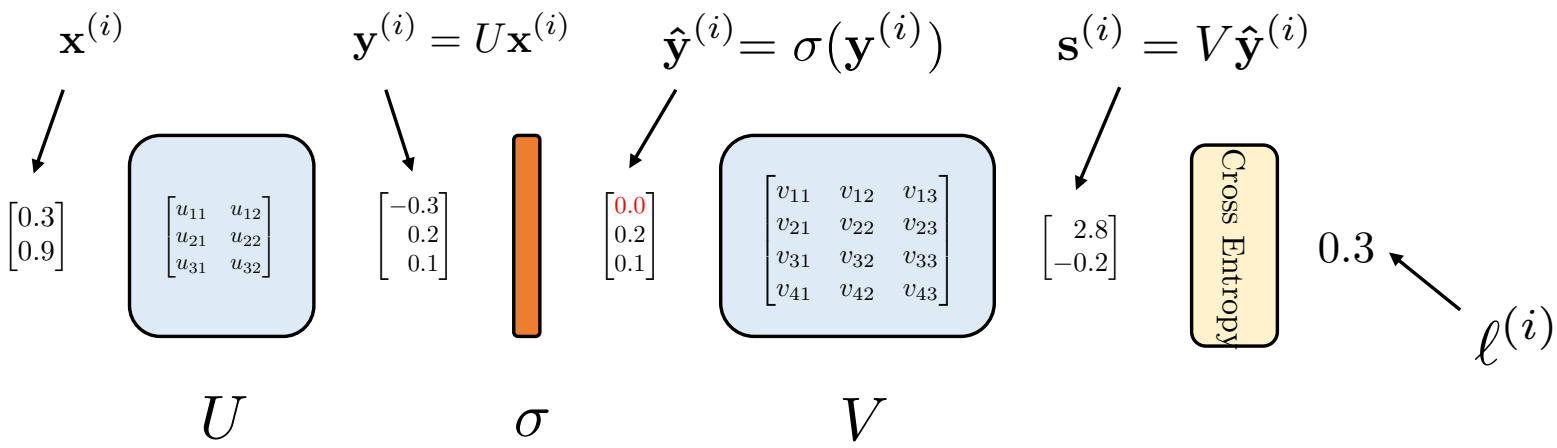
Two-layer MLP

- Forward pass and loss of a 2-layer MLP :



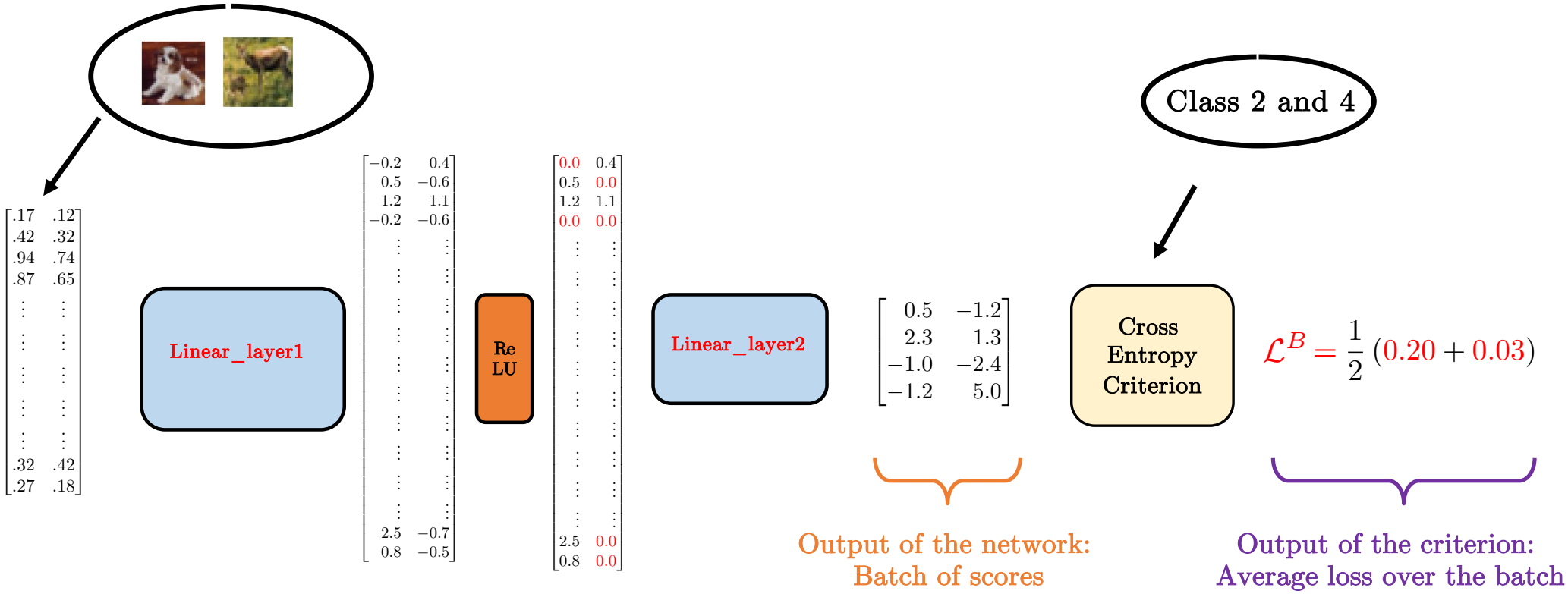
Two-layer MLP

- **Forward pass and loss of a 2-layer MLP with mathematical variables :**



Batch mode

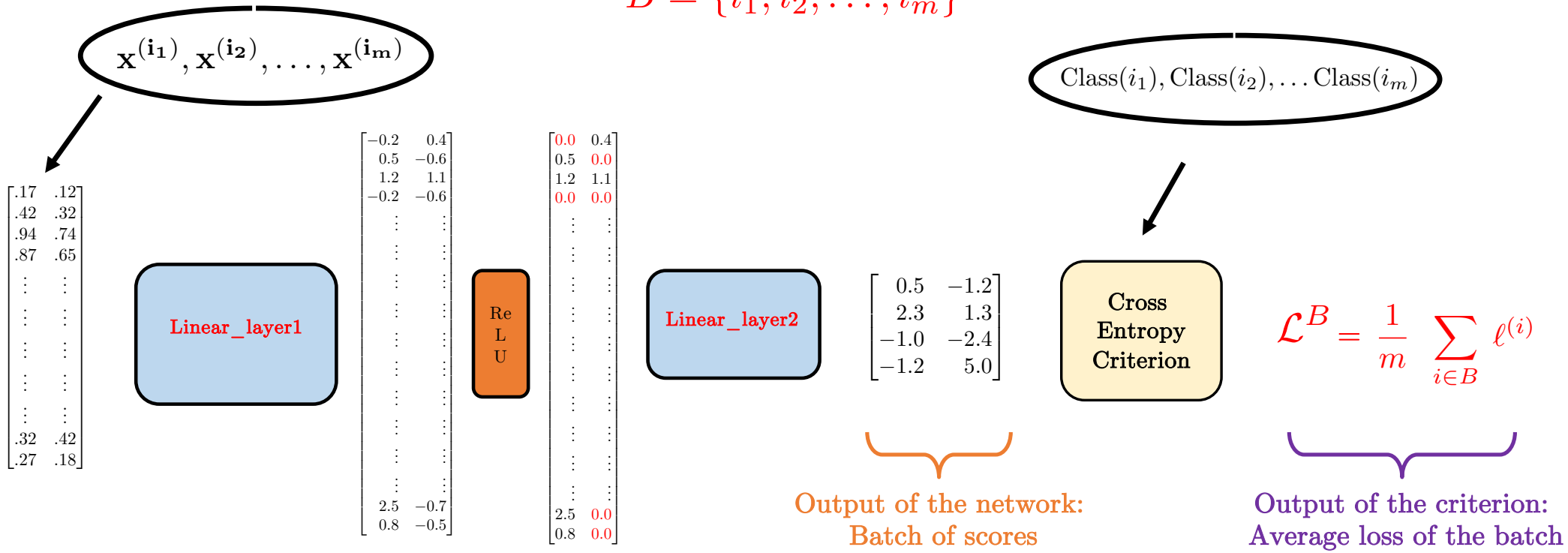
- Forward pass and loss of a 2-layer MLP for two input images :



Batch mode

- Batch size is arbitrary :
  - Sample randomly B indices from the training set :

$$B = \{i_1, i_2, \ldots, i_m\}$$

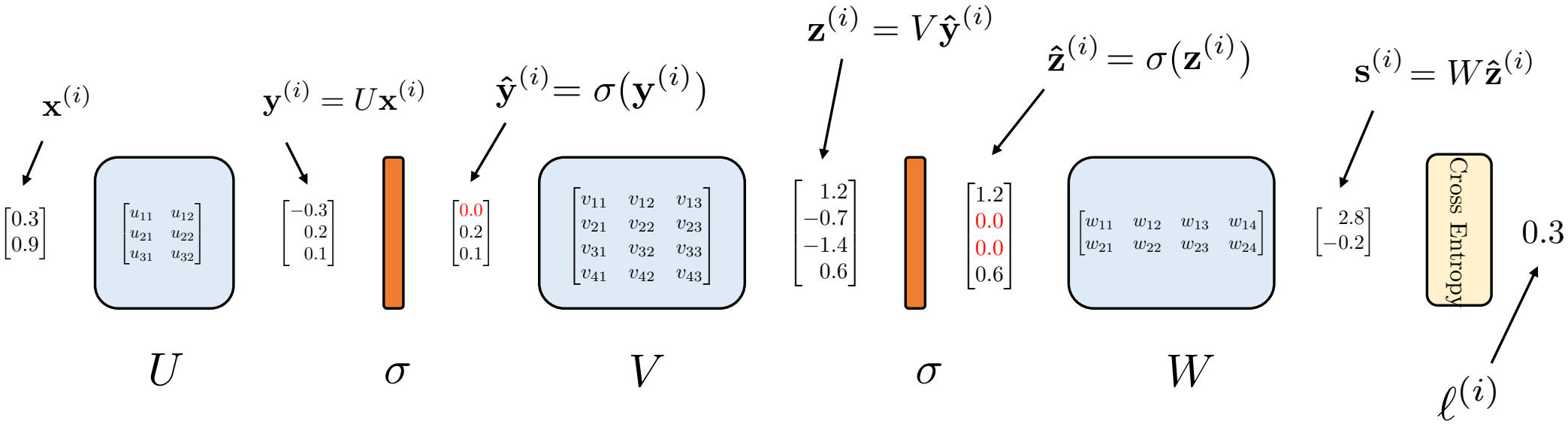


Outline

- Multi-Layer Perceptron
- Loss and gradient descent
- Backpropagation
- PyTorch implementation
- Training with epochs
- Monitoring the loss
- Test set evaluation
- Final code

Three-layer MLP

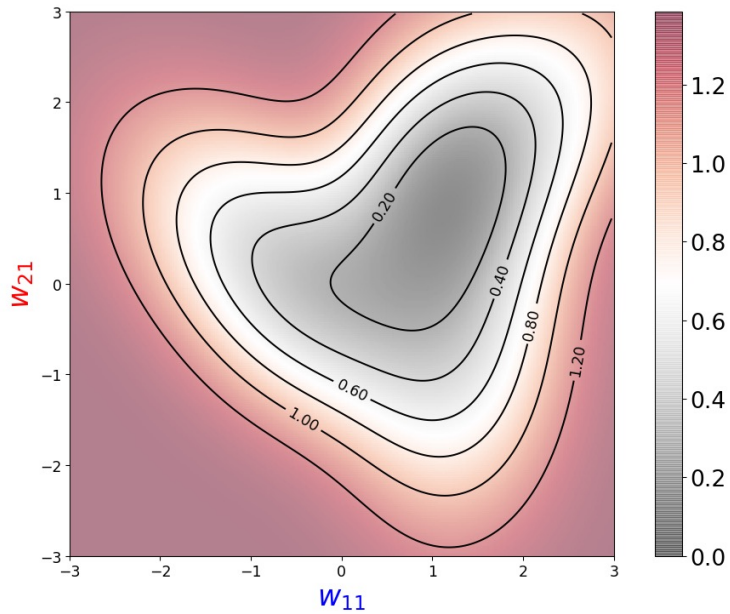
- We consider a three-layer MLP :



MLP loss

- We want to find the parameters that minimize the cross-entropy loss:

$$\mathcal{L}(u_{11}, u_{12}, \ldots) = \underbrace{\frac{1}{N} \sum_{i=1}^{N} \underbrace{\ell^{(i)}(u_{11}, u_{12}, \ldots)}_{\text{Loss of the } i^{\text{th}} \text{ data point}}}_{\text{Average of the losses over all data points}}$$



Mini-batch MLP loss

- Approximated mini-batch loss L^B :
  - Average of the losses of the data points in each batch B :

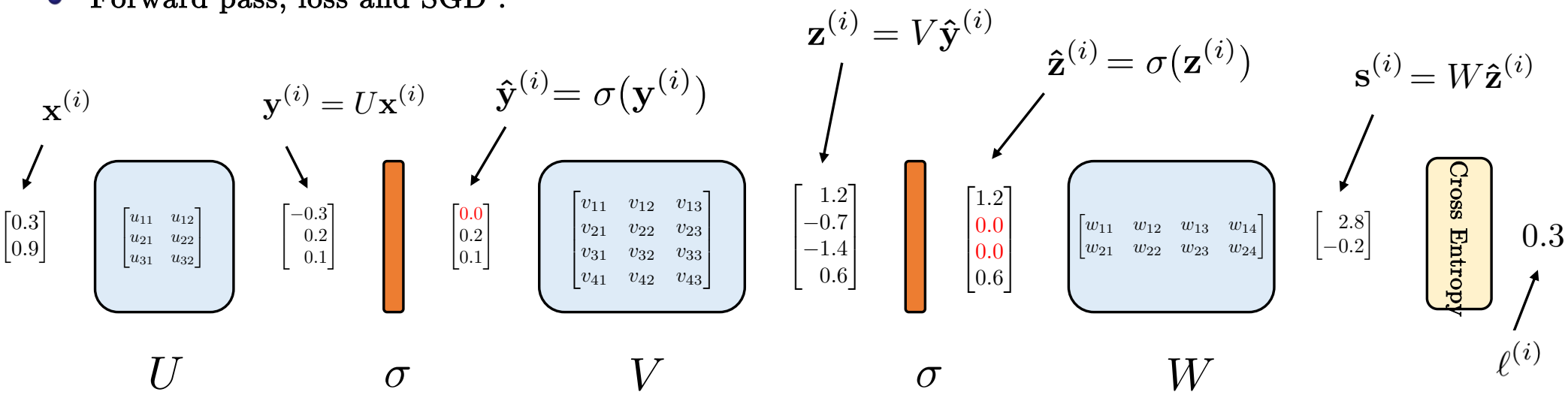
$$\mathcal{L}^B(u_{11}, u_{12}, \ldots) = \frac{1}{m} \sum_{i \in B} \ell^{(i)}(u_{11}, u_{12}, \ldots)$$

$$B = \{i_1, i_2, \ldots, i_m\}$$

Mini-batch B

Stochastic Gradient Descent

- **Forward pass, loss and SGD :**



$$U = U - \text{lr}\frac{\partial \mathcal{L}^B}{\partial U} \qquad V = V - \text{lr}\frac{\partial \mathcal{L}^B}{\partial V} \qquad W = W - \text{lr}\frac{\partial \mathcal{L}^B}{\partial W}$$

But how do we compute the gradients of the network parameters?

Outline

- Multi-Layer Perceptron
- Loss and gradient descent
- **Backpropagation**
- PyTorch implementation
- Training with epochs
- Monitoring the loss
- Test set evaluation
- Final code

Backpropagation

- Backpropagation:
  - Forward pass: Compute the average of the losses $\mathcal{L}^B$ of the data points in the batch by forwarding the batch through the net.
  - Backward pass: Get the derivatives of $\mathcal{L}^B$ w.r.t. the net parameters by back-propagating the loss value (error of prediction) through the net.
  - Update weight: Use the SGD formula to get the new weight values that make the network (a bit) better at prediction:

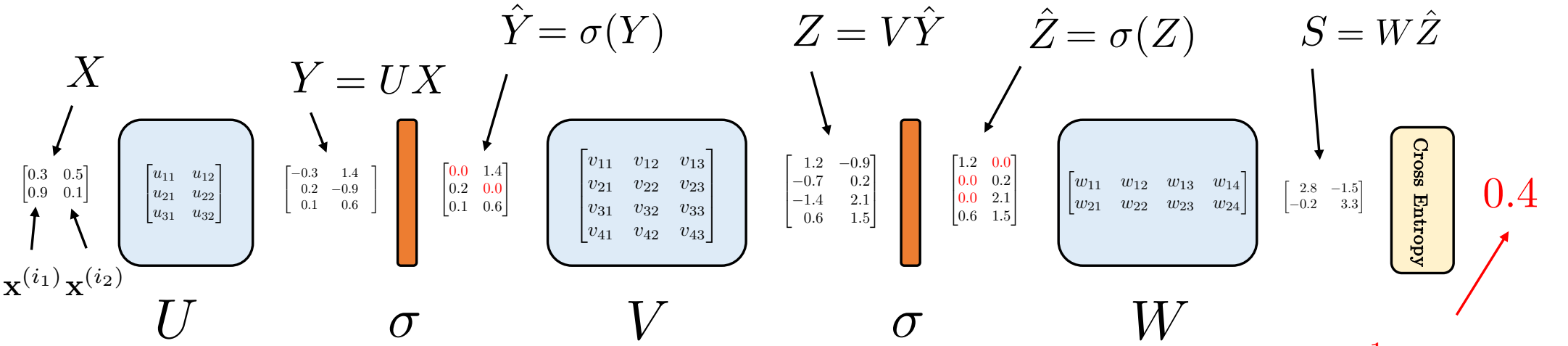
$$U = U - \text{lr} \frac{\partial \mathcal{L}^B}{\partial U}$$

$$V = V - \text{lr} \frac{\partial \mathcal{L}^B}{\partial V}$$

$$W = W - \text{lr} \frac{\partial \mathcal{L}^B}{\partial W}$$

Forward pass

- **Forward pass and loss :**



$$\mathcal{L}^B = \frac{1}{m} \sum_{i \in B} \ell^{(i)}$$

Average of the losses
of the data points in the batch

Forward pass

Backward pass

- How do we compute the gradients of the loss w.r.t. network parameters?

$$\frac{\partial \mathcal{L}^B}{\partial U} \quad \frac{\partial \mathcal{L}^B}{\partial V} \quad \text{and} \quad \frac{\partial \mathcal{L}^B}{\partial W}$$

- Let us introduce the backpropagation formula for a linear layer (that performs pattern matching).

- Understanding the backpropagation for a linear layer is enough to generalize the backpropagation process to any layer, and any network !

Backpropagation for linear layer – scalar case

- **Forward pass:**

Input signal x

Weight parameter w

Linear Layer

$y = wx$ Output signal

- **Backward pass:**

Chain rule on $\mathcal{L}^B(y(w,x))$!

$$\frac{\partial \mathcal{L}^B}{\partial x} = \frac{\partial \mathcal{L}^B}{\partial y} \cdot \frac{\partial y}{\partial x} = e.w$$

$$\frac{\partial \mathcal{L}^B}{\partial w} = \frac{\partial \mathcal{L}^B}{\partial y} \cdot \frac{\partial y}{\partial w} = e.x$$

Output gradients

Linear Layer

$e = \frac{\partial \mathcal{L}^B}{\partial y}$ Input gradient

This is the scalar case.

Backpropagation for linear layer - matrix case

- **Forward pass:**

Never learn before

very hard to differentiate

Input signal X

Weight parameter W

Linear layer

$Y = WX$ Output signal

- **Backward pass:**

computed solely to backprop gradient to prev layer

Chain rule on $\mathcal{L}^B(Y(W, X))$!

$$\frac{\partial \mathcal{L}^B}{\partial X} = \left(\frac{\partial Y}{\partial X}\right)^T \cdot \frac{\partial \mathcal{L}^B}{\partial Y}$$

$$= W^T E$$

Matrix dimension consistency

Output gradients

Linear layer

$E = \frac{\partial \mathcal{L}^B}{\partial Y}$ Input gradient

$$\frac{\partial \mathcal{L}^B}{\partial W} = \frac{\partial \mathcal{L}^B}{\partial Y} \cdot \left(\frac{\partial Y}{\partial W}\right)^T$$

$$= EX^T$$

computed for updating weights

$$A \longrightarrow \boxed{B} \longrightarrow C$$

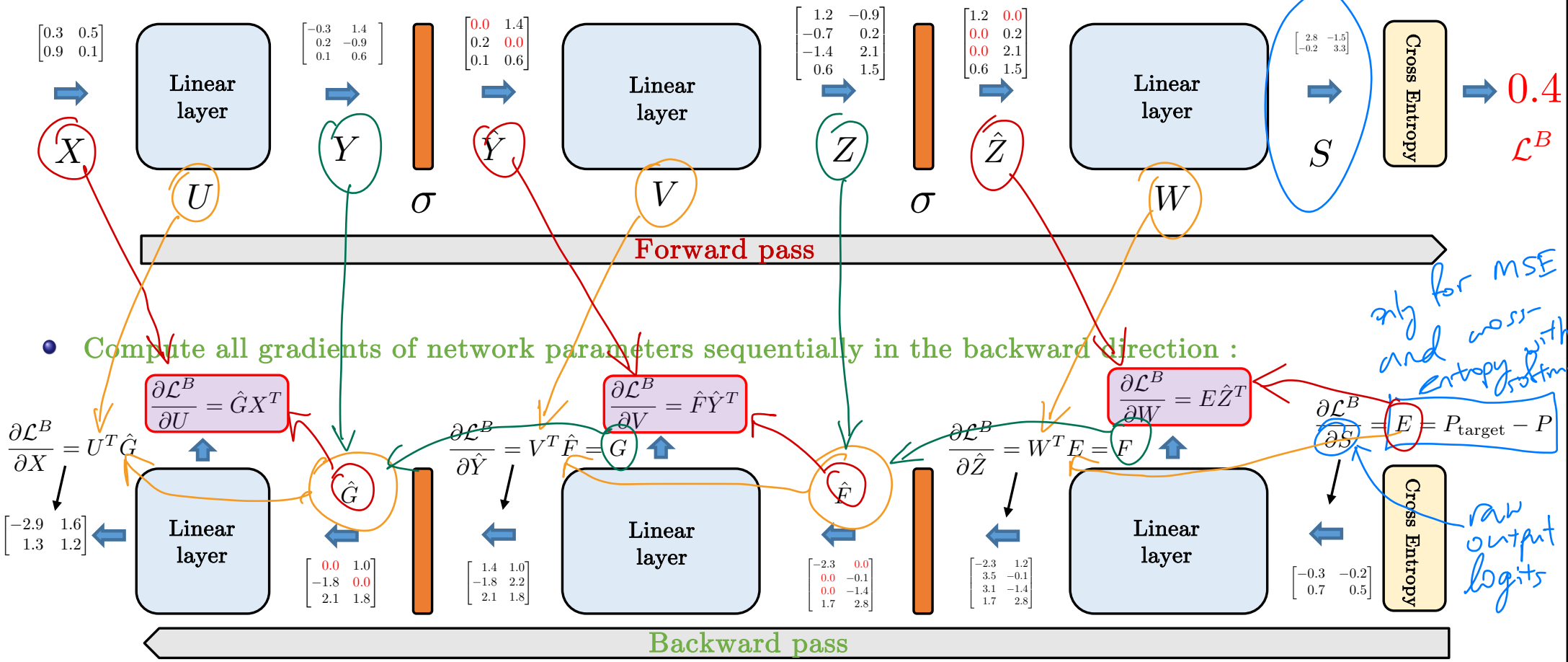
$$\frac{\partial L}{\partial A} = B^T$$

→ gradient propagation (through relu)
→ weight update
→ gradient propagation (thru linear layers)

Backpropagation for MLP

output logits

- **Compute all outputs of the forward pass :**



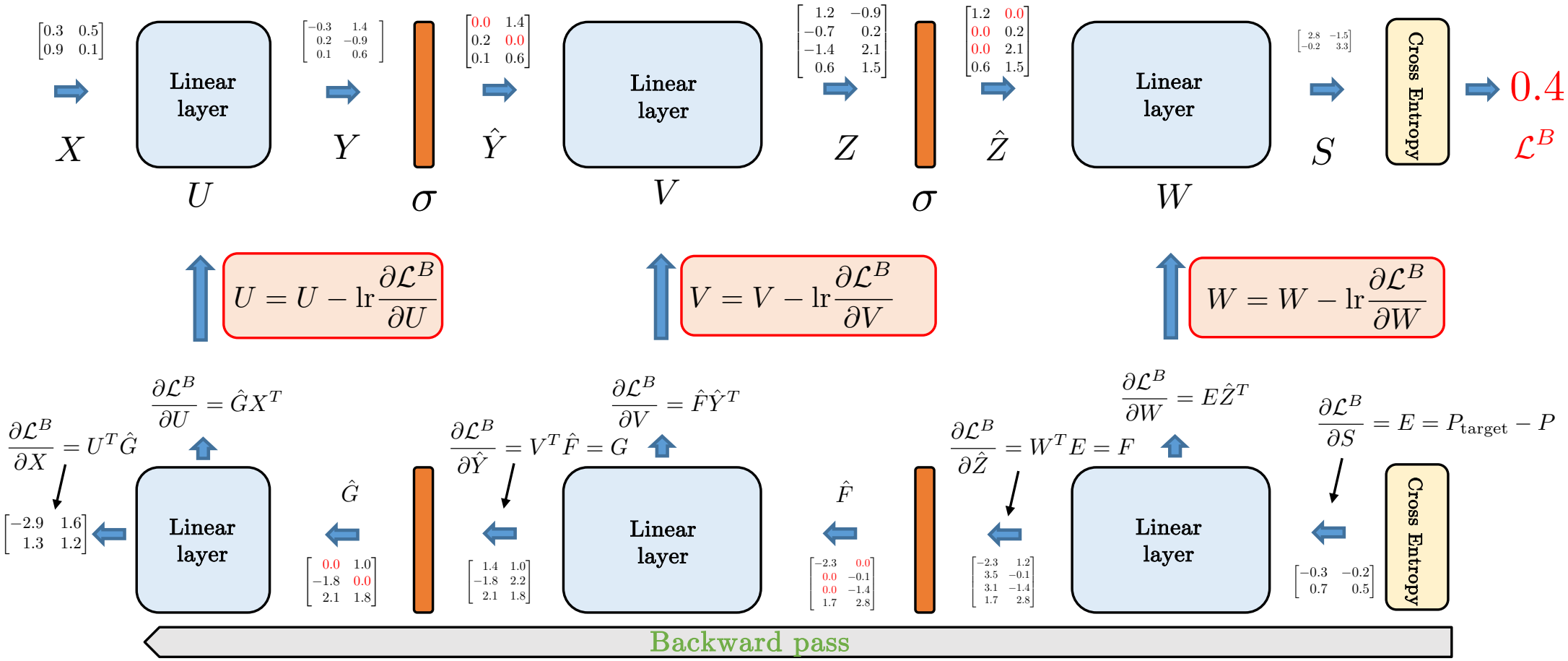
only for MSE and cross-entropy with softmax

raw output logits

- **Compute all gradients of network parameters sequentially in the backward direction :**

Backpropagation for MLP

- Update all network parameters with gradient descent sequentially in the backward direction :

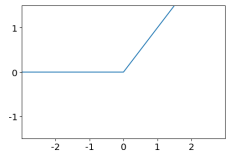


Backpropagation for ReLU layer

- **Forward pass:**

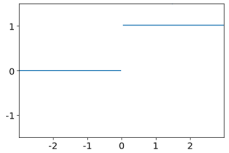
Input signal X → **ReLU layer** → $Y = \max(X, 0)$ Output signal

No weight parameter !



- **Backward pass:**

$$\frac{\partial Y}{\partial X} = \frac{\partial}{\partial X}\max(X,0) = \begin{cases} 1 & \text{if X value is positive} \\ 0 & \text{otherwise} \end{cases} = \text{step}(X)$$



Output gradient $\frac{\partial \mathcal{L}^B}{\partial X} = \left(\frac{\partial Y}{\partial X}\right)^T \cdot \frac{\partial \mathcal{L}^B}{\partial Y} = \text{step}(X)E$ ← **ReLU layer** ← $E = \frac{\partial \mathcal{L}^B}{\partial Y}$ Input gradient

Chain rule on $\mathcal{L}^B(Y(W,X))$!

Generalizing backpropagation for any layer

- Forward pass:

Input signals

x

y

w

Layer
f(x,y,w)

$z = f(x, y, w)$

Output signal

- Backward pass:

Chain rule on $\mathcal{L}^B(f(x, y, w))$!

Output gradients

$$\frac{\partial \mathcal{L}^B}{\partial x} = \frac{\partial \mathcal{L}^B}{\partial z} \cdot \frac{\partial z}{\partial x}$$

$$\frac{\partial \mathcal{L}^B}{\partial y} = \frac{\partial \mathcal{L}^B}{\partial z} \cdot \frac{\partial z}{\partial y}$$

$$\frac{\partial \mathcal{L}^B}{\partial w} = \frac{\partial \mathcal{L}^B}{\partial z} \cdot \frac{\partial z}{\partial w}$$

Layer
f(x,y,z)

$$e = \frac{\partial \mathcal{L}^B}{\partial z}$$

Input gradient

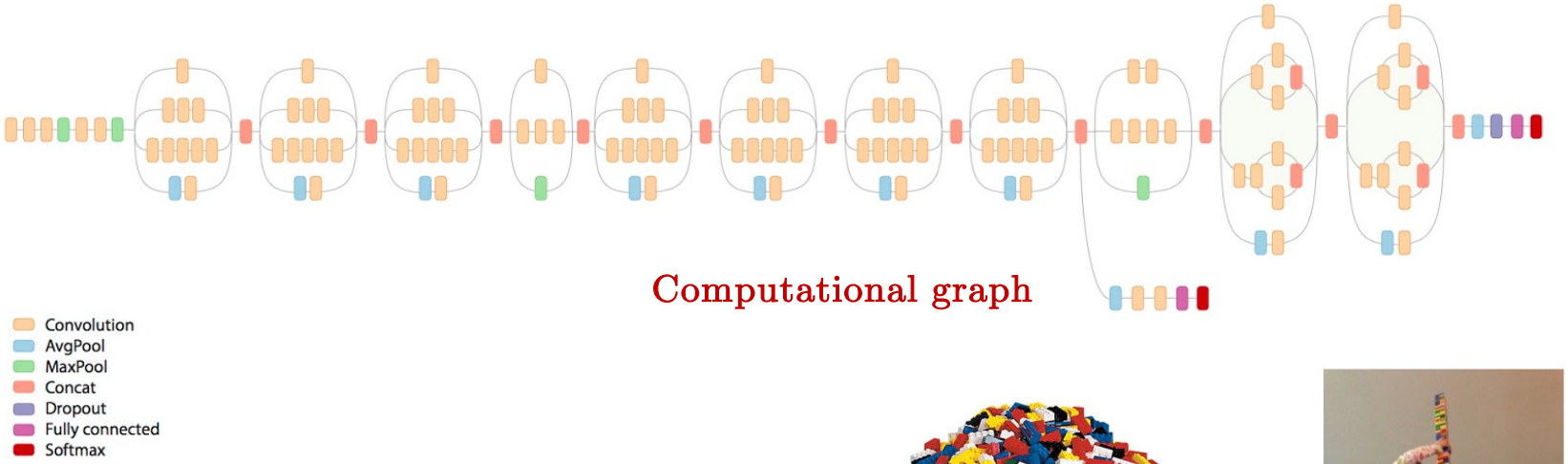
Backpropagation formula: Output gradient = Input gradient . Local gradient !

Backpropagation for any (huge) network

- **Backpropagation is a local rule:**

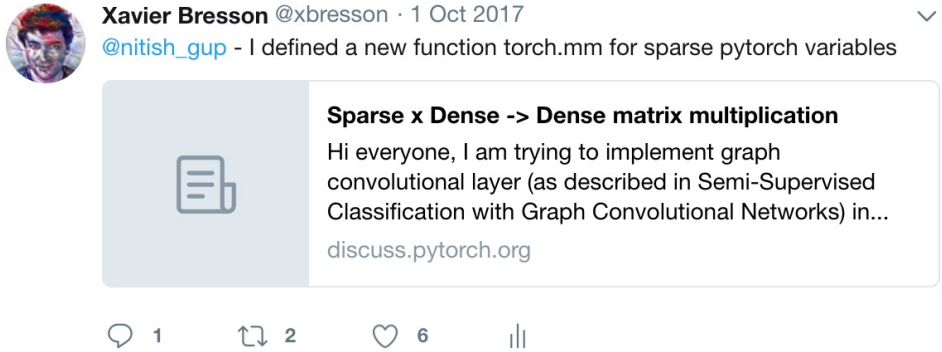
$$\text{Output gradient} = \text{Input gradient} \,.\, \text{Local gradient}$$

- **As such, it can scale up to any number of layers!**



Computational graph

Design your own layer



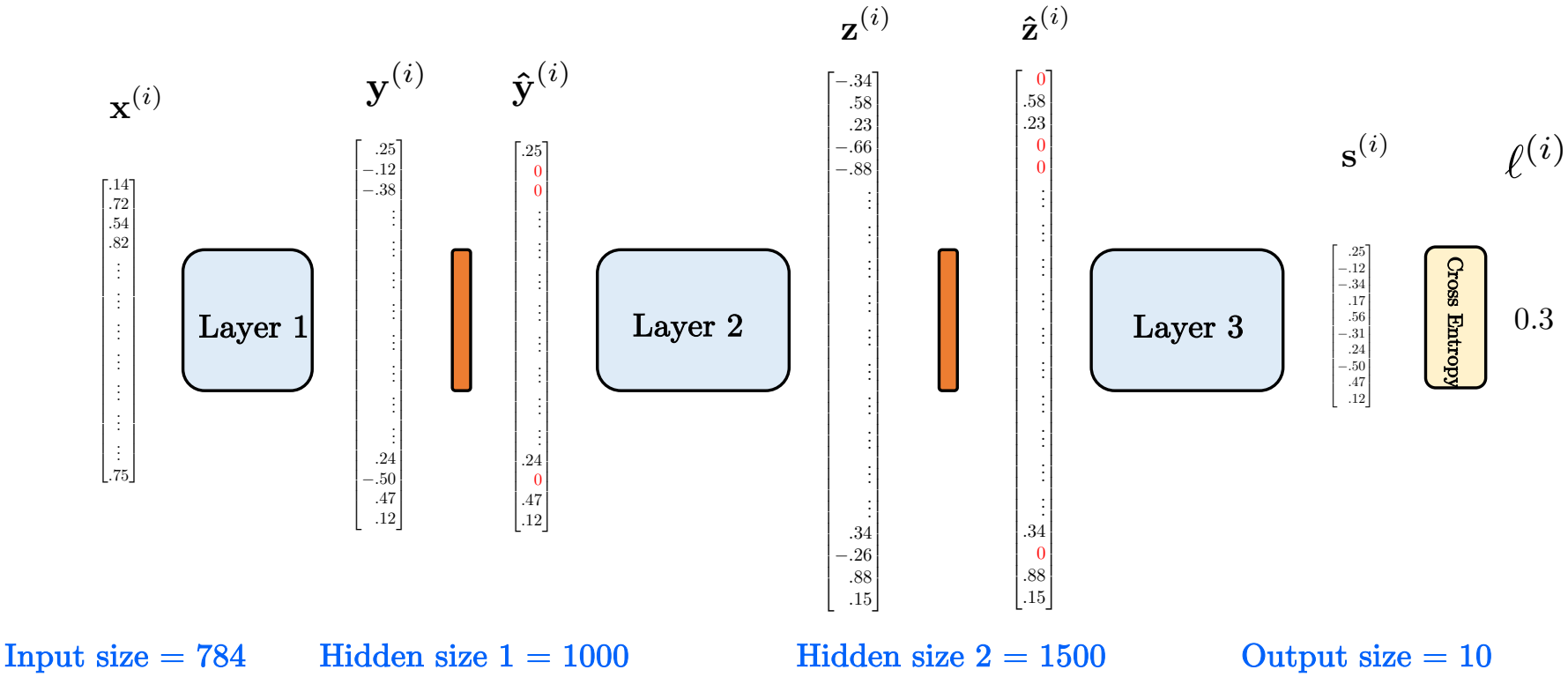

```python
class my_sparse_mm(torch.autograd.Function):
    """
    Implementation of a new autograd function for sparse variables,
    called "my_sparse_mm", by subclassing torch.autograd.Function
    and implementing the forward and backward passes.
    """

    def forward(self, W, x):  # W is SPARSE
        self.save_for_backward(W, x)
        y = torch.mm(W, x)
        return y

    def backward(self, grad_output):
        W, x = self.saved_tensors
        grad_input = grad_output.clone()
        grad_input_dL_dW = torch.mm(grad_input, x.t())
        grad_input_dL_dx = torch.mm(W.t(), grad_input )
        return grad_input_dL_dW, grad_input_dL_dx
```

Outline

- Multi-Layer Perceptron
- Loss and gradient descent
- Backpropagation
- **PyTorch implementation**
- Training with epochs
- Monitoring the loss
- Test set evaluation
- Final code

PyTorch implementation

- Let us implement a 3-layer network for MNIST and the cross-entropy loss :



Network and loss

- Define the network, instantiate it and choose the loss :

Make a class of 3-layer networks

```
class three_layer_net(nn.Module):

    def __init__(self, input_size, hidden_size1, hidden_size2,  output_size):
        super(three_layer_net , self).__init__()

        self.layer1 = nn.Linear( input_size,   hidden_size1 , bias=False)
        self.layer2 = nn.Linear( hidden_size1, hidden_size2 , bias=False)
        self.layer3 = nn.Linear( hidden_size2, output_size  , bias=False)

    def forward(self, x):

        y       = self.layer1(x)
        y_hat   = F.relu(y)
        z       = self.layer2(y_hat)
        z_hat   = F.relu(z)
        scores  = self.layer3(z_hat)

        return scores


net=three_layer_net(784, 1000, 1500, 10)


criterion = nn.CrossEntropyLoss()
```

Create the network

Define the loss/criterion

Hyper-parameters:

- Input dim: 784
- 1st hidden layer: 1000
- 2nd hidden layer: 1500
- Output: 10

Stochastic gradient descent

- Create an optimization function :
  - We create an optimizer object (from the **torch.optim** package):

the technique to minimize the loss

```
optimizer = torch.optim.SGD( net.parameters() , lr=0.01 )
```

optimize over all params (can be a subset of your NN if fine-tuning)

We choose our optimizer to be an SGD (there are other SGD-like optimizers)

The optimizer is given access to the parameters of the 3-layer network

We choose the learning rate to be lr = 0.01

Training loop and backpropagation

Create the minibatch

Forward pass: compute the scores

Compute the cross-entropy criterion:

$$\mathcal{L}^B = \frac{1}{m} \sum_{i \in B} \ell^{(i)}$$

Backward pass: compute the gradients

$$\frac{\partial \mathcal{L}^B}{\partial U} \quad \frac{\partial \mathcal{L}^B}{\partial V} \quad \frac{\partial \mathcal{L}^B}{\partial W}$$

Discussed later.

```
for iter in range(5000):

    optimizer.zero_grad()

    indices = torch.LongTensor(bs).random_(0,60000)
    minibatch_data  =  train_data[indices]
    minibatch_label = train_label[indices]

    inputs = minibatch_data.view(bs,784)

    inputs.requires_grad_()

    scores = net( inputs )

    loss = criterion( scores , minibatch_label )

    loss.backward()

    optimizer.step()
```

Batchsize

sampling

Discussed later.

Reshape

Start recording all operations that will be done to the input tensor.

same as net.forward

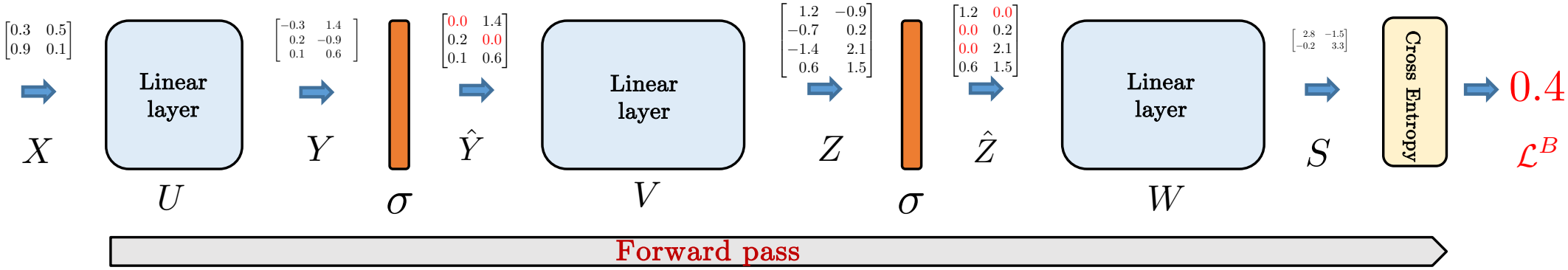
Discussed later.

Do one step of SGD to update the weights

$$U = U - \mathrm{lr}\frac{\partial \mathcal{L}^B}{\partial U} \quad V = V - \mathrm{lr}\frac{\partial \mathcal{L}^B}{\partial V} \quad W = W - \mathrm{lr}\frac{\partial \mathcal{L}^B}{\partial W}$$

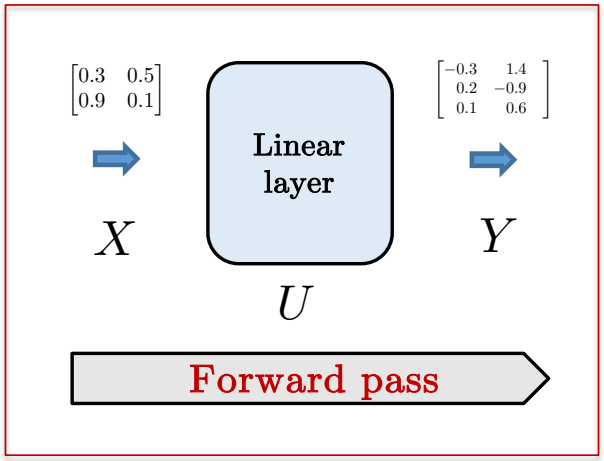
Automatic backpropagation

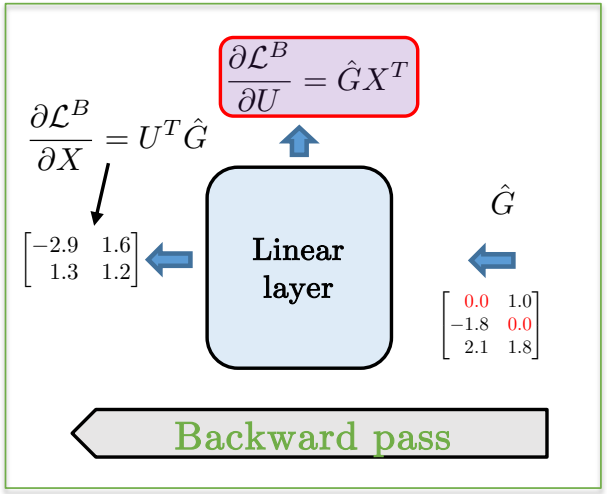
- **Function requires_grad_() :**
  - It tells Pytorch to start recording all operations that will be done to the input tensor X.
  - PyTorch will create the computational graph by stacking the layers and recording the outputs of all layers in sequential order.
  - It will then compute all gradients automatically (autograd) with the backward pass.



Automatic backpropagation

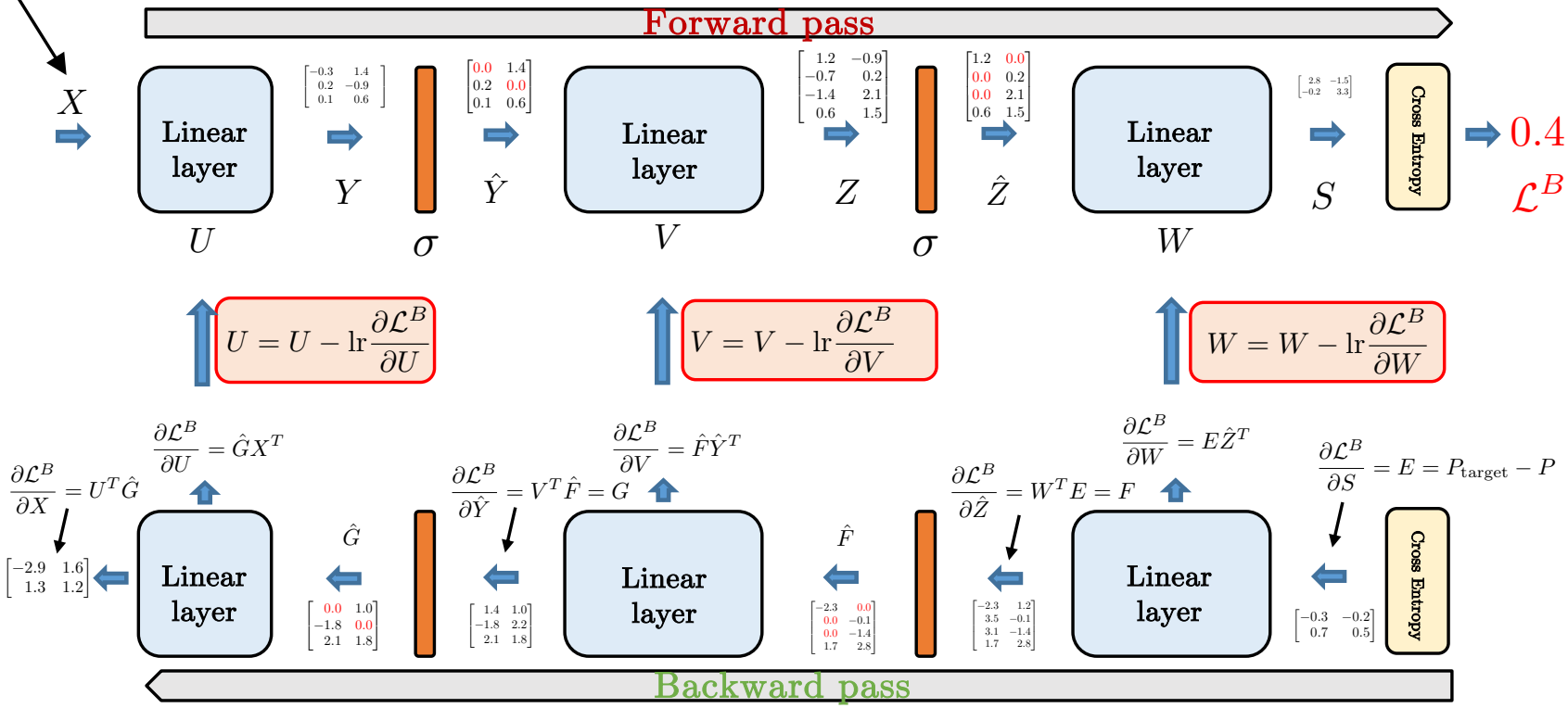
- Function **requires_grad_()** :
  - In Pytorch, each forward operation that can be performed on a tensor has a backward equivalent (already defined or that can be implemented).
  - In deep learning, most layers are composed of simple operations (sum, multiplication, exp(), convolution, etc) that are efficiently implemented with e.g. MKL on CPU or CUDA on GPU. Python functions are just wrappers.





Automatic backpropagation

Put a requires_grad() flag on this tensor and it will record the history of all the operations it goes through.



Then all these operations are replayed in reverse order with **loss.backward()** (the backward version of these operations)

Gradient accumulation

- If **optimizer.zero_grad()** is NOT used then Pytorch adds the gradient computed on this batch to the total gradient: accumulates gradient

set these gradients to 0

$$\frac{\partial \mathcal{L}}{\partial U} = \frac{\partial \mathcal{L}}{\partial U} + \frac{\partial \mathcal{L}^B}{\partial U}\Big|_{\text{computed on this batch}}$$

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial W} + \frac{\partial \mathcal{L}^B}{\partial W}\Big|_{\text{computed on this batch}}$$

$$\frac{\partial \mathcal{L}}{\partial V} = \frac{\partial \mathcal{L}}{\partial V} + \frac{\partial \mathcal{L}^B}{\partial V}\Big|_{\text{computed on this batch}}$$

```python
for iter in range(5000):

    optimizer.zero_grad()

    indices = torch.LongTensor(bs).random_(0,60000)
    minibatch_data  =  train_data[indices]
    minibatch_label = train_label[indices]

    inputs = minibatch_data.view(bs,784)

    inputs.requires_grad_()

    scores = net( inputs )

    loss = criterion( scores , minibatch_label )

    loss.backward()

    optimizer.step()
```

- This is only convenient if you want to do a true gradient where you use all the data points to compute the gradient.

wrt all datapoints

Mini-batch gradient

- But if we want to do a stochastic gradient descent, we need function **optimizer.zero_grad()** :

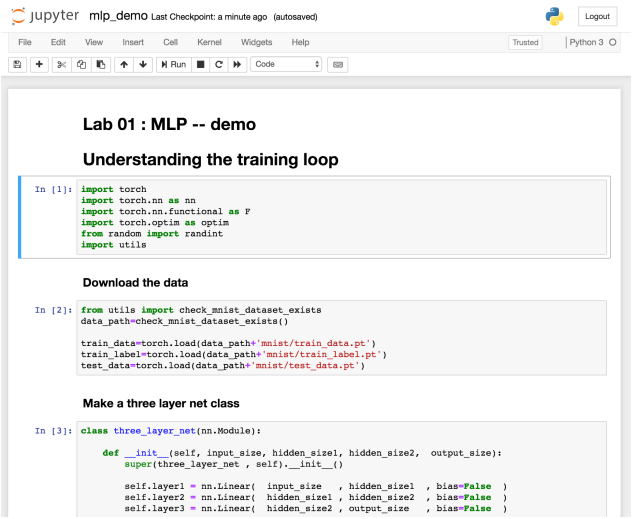
$$\frac{\partial \mathcal{L}}{\partial U} = 0$$
$$\frac{\partial \mathcal{L}}{\partial V} = 0$$
$$\frac{\partial \mathcal{L}}{\partial W} = 0$$

$$\frac{\partial \mathcal{L}}{\partial U} = 0 + \frac{\partial \mathcal{L}^B}{\partial U}\Big|_{\text{computed on this batch}}$$
$$\frac{\partial \mathcal{L}}{\partial W} = 0 + \frac{\partial \mathcal{L}^B}{\partial W}\Big|_{\text{computed on this batch}}$$
$$\frac{\partial \mathcal{L}}{\partial V} = 0 + \frac{\partial \mathcal{L}^B}{\partial V}\Big|_{\text{computed on this batch}}$$

```
for iter in range(5000):

    optimizer.zero_grad()

    indices = torch.LongTensor(bs).random_(0,60000)
    minibatch_data  =  train_data[indices]
    minibatch_label = train_label[indices]

    inputs = minibatch_data.view(bs,784)

    inputs.requires_grad_()

    scores = net( inputs )

    loss = criterion( scores , minibatch_label )

    loss.backward()

    optimizer.step()
```

Lab 01

- Understanding the training loop



jupyter mlp_demo Last Checkpoint: a minute ago (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

Lab 01 : MLP -- demo

Understanding the training loop

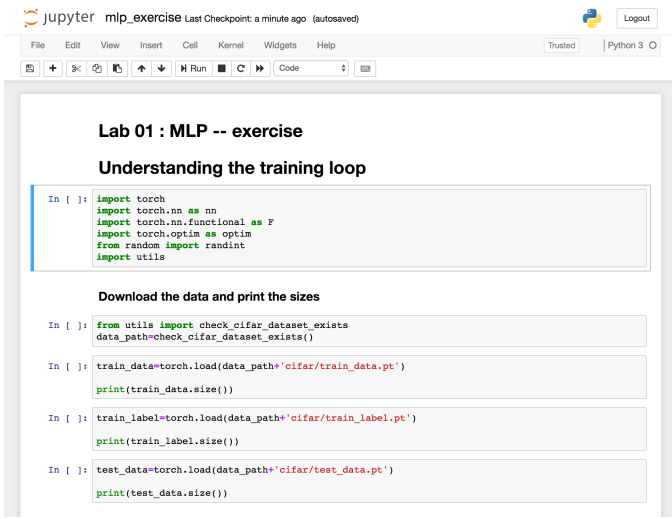
In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from random import randint
import utils
```

Download the data

In [2]:
```
from utils import check_mnist_dataset_exists
data_path=check_mnist_dataset_exists()

train_data=torch.load(data_path+'mnist/train_data.pt')
train_label=torch.load(data_path+'mnist/train_label.pt')
test_data=torch.load(data_path+'mnist/test_data.pt')
```

Make a three layer net class

In [3]:
```
class three_layer_net(nn.Module):

    def __init__(self, input_size, hidden_size1, hidden_size2,  output_size):
        super(three_layer_net , self).__init__()

        self.layer1 = nn.Linear(  input_size   , hidden_size1  , bias=False  )
        self.layer2 = nn.Linear(  hidden_size1 , hidden_size2  , bias=False  )
        self.layer3 = nn.Linear(  hidden_size2 , output_size   , bias=False  )
```



jupyter mlp_exercise Last Checkpoint: a minute ago (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

Lab 01 : MLP -- exercise

Understanding the training loop

In [ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from random import randint
import utils
```

Download the data and print the sizes

In [ ]:
```
from utils import check_cifar_dataset_exists
data_path=check_cifar_dataset_exists()
```

In [ ]:
```
train_data=torch.load(data_path+'cifar/train_data.pt')

print(train_data.size())
```

In [ ]:
```
train_label=torch.load(data_path+'cifar/train_label.pt')

print(train_label.size())
```

In [ ]:
```
test_data=torch.load(data_path+'cifar/test_data.pt')

print(test_data.size())
```

Outline

- Multi-Layer Perceptron
- Loss and gradient descent
- Backpropagation
- PyTorch implementation
- **Training with epochs**
- Monitoring the loss
- Test set evaluation
- Final code

Training with epochs

- Up to now we have been training as follows:
  - Feed 200 randomly selected images to the net (note that the same image can be drawn multiple times).

- An epoch is a full pass through the randomly ordered training set.
  - Training with epochs prevents sampling the same image before seeing all other training images.
  - At the beginning of the epoch, the 60,000 images from the training set are randomly ordered.
    - Then:
      - Feed image 0 - 199 to the net and train.
      - Feed image 200 - 399 to the net and train.
      - ...
      - Feed image 59,800 - 59,999 to the net and train.

Training with epochs

- We have been training as follows:

```
for iter in range(5000):

    optimizer.zero_grad()

    indices = torch.LongTensor(bs).random_(0,60000)
    minibatch_data  =  train_data[indices]
    minibatch_label = train_label[indices]

    inputs = minibatch_data.view(bs,784)

    inputs.requires_grad_()

    scores = net( inputs )

    loss = criterion( scores , minibatch_label )

    loss.backward()

    optimizer.step()
```

- Training with epochs :
  - Here we do 8 complete passes through the training set.

```
for epoch in range(8):

    shuffled_indices=torch.randperm(60000)

    for count in range(0,60000,bs):

        optimizer.zero_grad()

        indices=shuffled_indices[count:count+bs]
        minibatch_data =  train_data[indices]
        minibatch_label= train_label[indices]

        inputs = minibatch_data.view(bs,784)

        inputs.requires_grad_()

        scores=net( inputs )

        loss = criterion( scores , minibatch_label)

        loss.backward()

        optimizer.step()
```

Lab 02

- Training with epochs

jupyter epoch_demo Last Checkpoint: a minute ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Lab 02 : Training with epochs -- demo

In [1]:
```
import torch
```

Lets make an artificial training set of 10 images (28x28 pixels)

In [2]:
```
train_data = torch.rand(10,28,28)

print(train_data.size())
```

```
torch.Size([10, 28, 28])
```

Lets define a the random order in which we are going to visit these images:

In [3]:
```
shuffled_indices = torch.randperm(10)
print(shuffled_indices)
```

```
tensor([3, 8, 0, 9, 7, 5, 1, 4, 2, 6])
```

Visit the training set in this random order and do minibatch of size 2

In [4]:
```
bs=2

for count in range(0,10,bs):

    batch_of_indices = shuffled_indices[count:count+bs]

    print(batch_of_indices)

    batch_of_images = train_data[ batch_of_indices  ]
```

jupyter epoch_exercise Last Checkpoint: a minute ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Lab 02: Training with epochs -- exercise

In [ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from random import randint
import utils
```

Download the data and print the sizes

In [ ]:
```
from utils import check_mnist_dataset_exists
data_path=check_mnist_dataset_exists()

train_data=torch.load(data_path+'mnist/train_data.pt')
train_label=torch.load(data_path+'mnist/train_label.pt')
test_data=torch.load(data_path+'mnist/test_data.pt')
```

Make a ONE layer net class.

In [ ]:
```
class one_layer_net(nn.Module):

    def __init__(self, input_size, output_size):
        super(one_layer_net , self).__init__()
        self.linear_layer = nn.Linear( input_size, output_size , bias=False)

    def forward(self, x):
        scores = self.linear_layer(x)
        return scores
```

Outline

- Multi-Layer Perceptron
- Loss and gradient descent
- Backpropagation
- PyTorch implementation
- Training with epochs
- **Monitoring the loss**
- Test set evaluation
- Final code

Monitoring the loss

- Monitoring the loss and the classification error :

Reset to zero at the beginning of a new epoch

running loss += average loss of current mini-batch

running error += average error on current mini-batch

```
for epoch in range(30):

    running_loss=0
    running_error=0
    num_batches=0

    shuffled_indices=torch.randperm(60000)

    for count in range(0,60000,bs):

        optimizer.zero_grad()

        indices=shuffled_indices[count:count+bs]
        minibatch_data =  train_data[indices]
        minibatch_label= train_label[indices]

        inputs = minibatch_data.view(bs,784)

        inputs.requires_grad_()

        scores=net( inputs )

        loss =  criterion( scores , minibatch_label)

        loss.backward()

        optimizer.step()

        # compute and accumulate stats

        running_loss += loss.detach().item()

        error = utils.get_error( scores.detach() , minibatch_label)
        running_error += error.item()

        num_batches+=1

    # compute stats for the full training set
    total_loss = running_loss/num_batches
    total_error = running_error/num_batches

    print('epoch=',epoch, '\t loss=', total_loss , '\t error=', total_error*100 ,'percent')
```

? convert to Python float type

Shouldn't we ÷ N or 60,000 not B ?

Monitoring loss and error

- Running error $= 10\% + 15\% + 12\% + \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots + 15\%$

Error rate on the 1st batch of 200 images

Error rate on the last batch of 200 images

Error rate on the full training set $= \dfrac{\text{running error}}{\text{number of batches}}$

- Same for the running loss

Detach operation

- Observe the code line:
  - **running_loss += loss.detach().item()**

- Function **item()** :
  - Get a Python number from a tensor containing a single value.

- Function **detach()** :
  - The .detach() function is essential because it extracts the numerical value of a node without memorizing this operation in the construction of the computational graph (forward pass).
  - Consequence: When running the backward pass, there is no operation to carry out, like the gradient computation (as monitoring the loss value does not require to compute its gradient) or the update of parameter.

In-lecture question

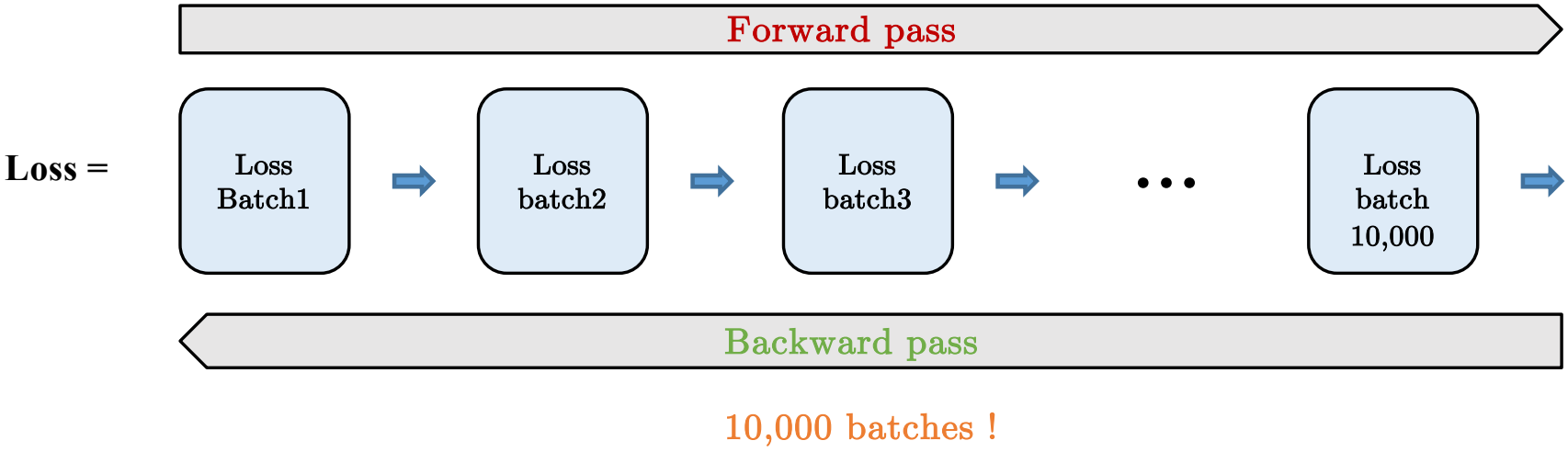
- In the context of training a MLP with PyTorch, why is it essential to use the .detach() function when accumulating a “running loss” for monitoring purposes, and what is the computational risk of omitting it?
- In Slack #lectures
  - Identify the question and Reply in thread with a short response
- Answer : If .detach() is omitted, PyTorch continues to build an increasingly massive computational graph by linking the losses of every single batch processed during an epoch (e.g. potentially 10,000 batches).

  This leads to excessive memory consumption and causes the backward pass (autograd) to take a significantly longer time.

  Since the "running loss" is only used for human monitoring and does not require a gradient to update network weights, there is no need to store the tracking information required for backpropagation for that specific variable.

PyTorch

- Without **detach()** :
  - **running_loss += loss**
  - **loss.backward()**



Lab 03

- Computing the loss and the error rate

jupyter monitoring_loss_demo Last Checkpoint: a minute ago (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

Lab 03 : Loss and error rate -- demo

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from random import randint
import utils
```

Download the data and print the sizes

In [2]:
```
from utils import check_mnist_dataset_exists
data_path=check_mnist_dataset_exists()

train_data=torch.load(data_path+'mnist/train_data.pt')
train_label=torch.load(data_path+'mnist/train_label.pt')
test_data=torch.load(data_path+'mnist/test_data.pt')
```

Make a ONE layer net class.

In [3]:
```
class one_layer_net(nn.Module):

    def __init__(self, input_size, output_size):
        super(one_layer_net , self).__init__()
        self.linear_layer = nn.Linear( input_size, output_size , bias=False)

    def forward(self, x):
        scores = self.linear_layer(x)
        return scores
```

jupyter monitoring_loss_exercise Last Checkpoint: a minute ago (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

Lab 03 : Loss and error rate -- exercise

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from random import randint
import utils
```

Download the CIFAR dataset -- check the size carefully!

In [2]:
```
from utils import check_cifar_dataset_exists
data_path=check_cifar_dataset_exists()

train_data=torch.load(data_path+'cifar/train_data.pt')
train_label=torch.load(data_path+'cifar/train_label.pt')
test_data=torch.load(data_path+'cifar/test_data.pt')

print(train_data.size())
```

```
torch.Size([50000, 3, 32, 32])
```

Make a ONE layer net class.

In [ ]:
```
class one_layer_net(nn.Module):

    def __init__(self, input_size, output_size):
        super(one_layer_net , self).__init__()
        self.linear_layer = nn.Linear( input_size, output_size , bias=True)
```

Outline

- Multi-Layer Perceptron
- Loss and gradient descent
- Backpropagation
- PyTorch implementation
- Training with epochs
- Monitoring the loss
- **Test set evaluation**
- Final code

Generalization performance

- **Evaluation on the test set :**
  - No need to shuffle the test set.
  - Just make a pass through the full test set in the original order.

```
running_error=0
num_batches=0


for i in range(0,10000,bs):

    minibatch_data =  test_data[i:i+bs]
    minibatch_label= test_label[i:i+bs]

    inputs = minibatch_data.view(bs,784)

    scores = net( inputs )

    error = utils.get_error( scores , minibatch_label)

    running_error += error.item()

    num_batches+=1



total_error = running_error/num_batches

print( 'error rate on test set =', total_error*100 ,'percent')
```

// don't actually need to batch
also, no need shuffle

Lab 04

- Test set evaluation

jupyter test_set_demo Last Checkpoint: 3 minutes ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Lab 04 : Test set evaluation -- demo

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from random import randint
import utils
```

Download the data and print the sizes

In [2]:
```
from utils import check_mnist_dataset_exists
data_path=check_mnist_dataset_exists()

train_data=torch.load(data_path+'mnist/train_data.pt')
train_label=torch.load(data_path+'mnist/train_label.pt')
test_data=torch.load(data_path+'mnist/test_data.pt')
test_label=torch.load(data_path+'mnist/test_label.pt')
```

Make a ONE layer net class.

In [3]:
```
class one_layer_net(nn.Module):

    def __init__(self, input_size, output_size):
        super(one_layer_net , self).__init__()
        self.linear_layer = nn.Linear( input_size, output_size , bias=False)

    def forward(self, x):
        scores = self.linear_layer(x)
        return scores
```

jupyter test_set_exercise Last Checkpoint: 3 minutes ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Lab 04 : Test set evaluation -- exercise

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from random import randint
import utils
```

Download the CIFAR dataset -- check the size carefully!

In [2]:
```
from utils import check_cifar_dataset_exists
data_path=check_cifar_dataset_exists()

train_data=torch.load(data_path+'cifar/train_data.pt')
train_label=torch.load(data_path+'cifar/train_label.pt')
test_data=torch.load(data_path+'cifar/test_data.pt')
test_label=torch.load(data_path+'cifar/test_label.pt')

print(train_data.size())
```

```
torch.Size([50000, 3, 32, 32])
```

Make a ONE layer net class.

In [ ]:
```
class one_layer_net(nn.Module):

    def __init__(self, input_size, output_size):
        super(one_layer_net , self).__init__()
        self.linear_layer = nn.Linear( input_size, output_size , bias=True)
```

Outline

- Multi-Layer Perceptron
- Loss and gradient descent
- Backpropagation
- PyTorch implementation
- Training with epochs
- Monitoring the loss
- Test set evaluation
- **Final code**

Final code

- We just make the following improvements to our code:
  - Add a learning rate schedule. Fast lr → slow lr
  - Evaluate on test set every 10 epochs (rather than waiting for the end of the training). evaluate on validation set every epoch
  - Provide some timing information. ← How much time taken to train 1 epoch
  - Count number of parameters in network.
  - Train on test set just before deployment
- This final code is a prototype for any deep learning project:
  - What changes for a new project:
    - New data (collect, prepare, access to batch of training data)
    - New architecture
    - New learning rate schedule

Hacks:

- AT LEAST 3,000 EPOCHS 10,000 is better
- Everything nowadays is transformer architecture. The only changes worth changing are the inputs.

Lab 05

- Demo of the final code :

jupyter final_demo Last Checkpoint: 3 minutes ago (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

Lab 05 : Final code -- demo

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from random import randint
import time
import utils
```

Download the data

In [2]:
```
from utils import check_mnist_dataset_exists
data_path=check_mnist_dataset_exists()

train_data=torch.load(data_path+'mnist/train_data.pt')
train_label=torch.load(data_path+'mnist/train_label.pt')
test_data=torch.load(data_path+'mnist/test_data.pt')
test_label=torch.load(data_path+'mnist/test_label.pt')
```

Make a one layer net class.

In [3]:
```
class one_layer_net(nn.Module):

    def __init__(self, input_size, output_size):
        super(one_layer_net , self).__init__()
        self.linear_layer = nn.Linear( input_size, output_size , bias=False)

    def forward(self, x):
        scores = self.linear_layer(x)
        return scores
```

Questions?

CS5242 : Neural Networks and Deep Learning

Lecture 4 : Multi-Layer Perceptron
Part 2 : Depth

Semester 2 2025/26

Xavier Bresson

https://twitter.com/xbresson

Department of Computer Science
National University of Singapore (NUS)

Outline

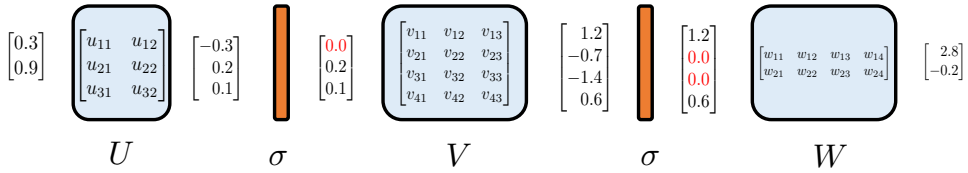
- The three representations of neural networks
- The importance of non-linearities
- The importance of multiple layers
- Labs with MNIST and CIFAR

Outline

- The three representations of neural networks
- The importance of non-linearities
- The importance of multiple layers
- Labs with MNIST and CIFAR

The three representations

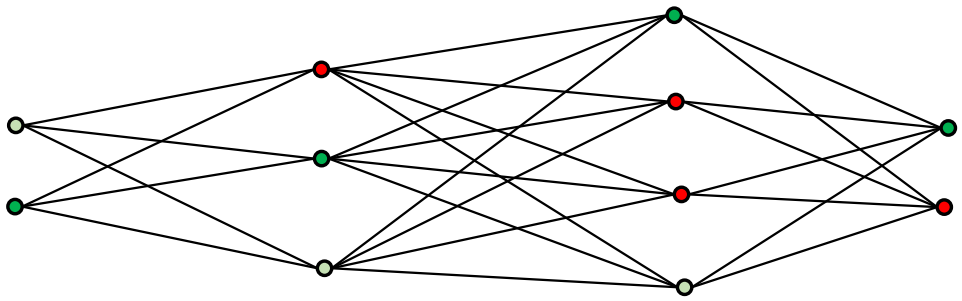
Computer Science



Mathematics

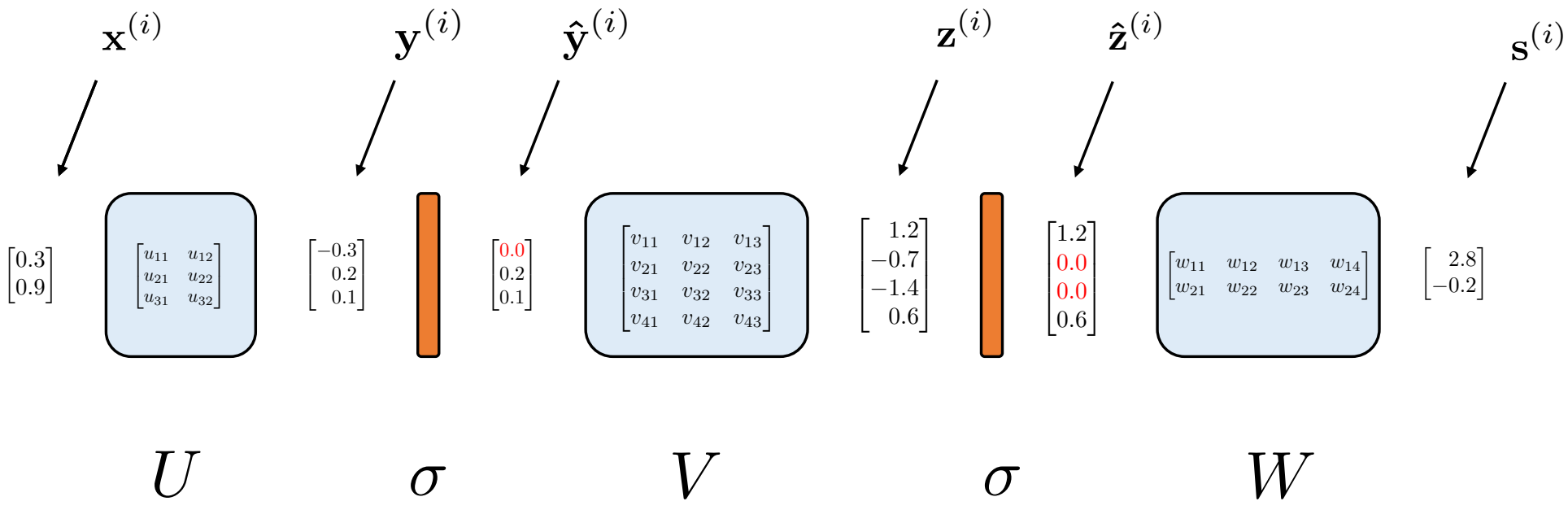
$$\mathbf{s}^{(\mathbf{i})} = W \;\; \sigma(V \;\; \sigma(U\mathbf{x}^{(\mathbf{i})}) \;\;)$$

Biology



Computer science representation

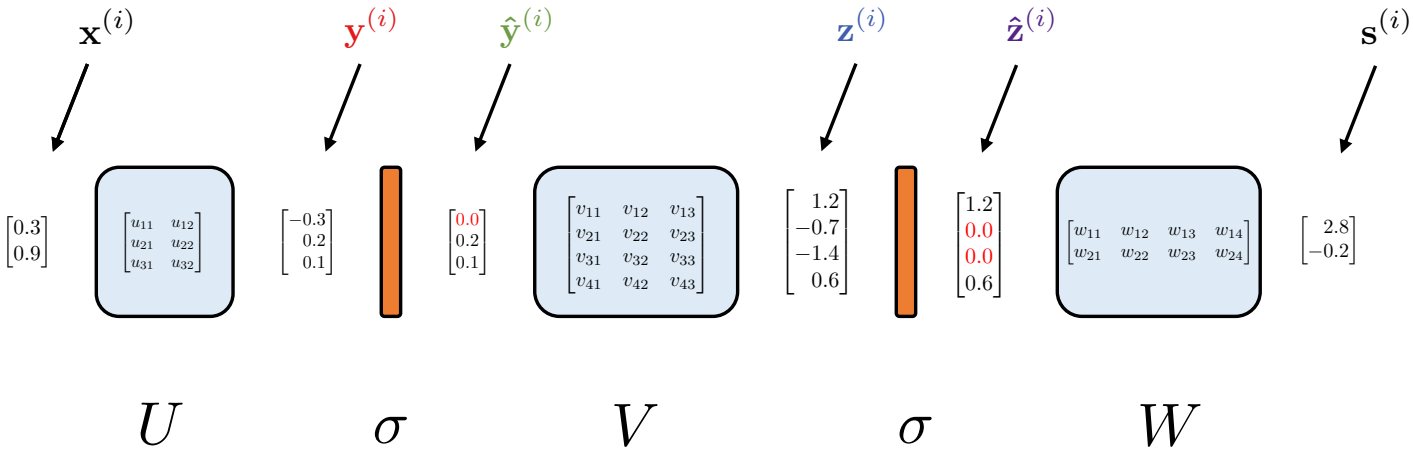
- Computer science visualization of a neural network:



Mathematical representation

- Mathematical formulation of a neural network :

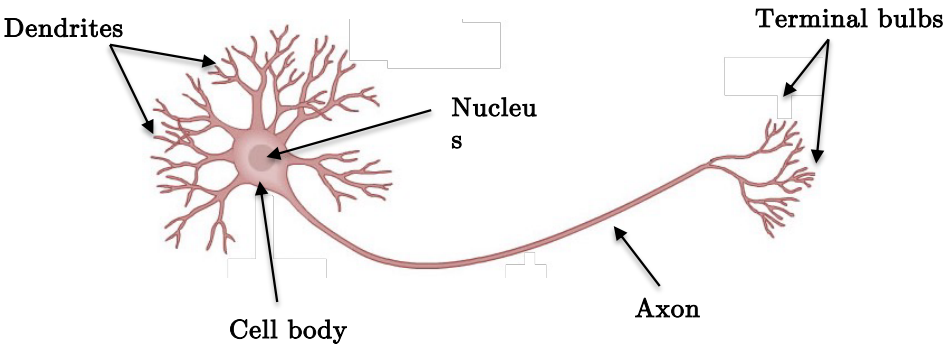
$$\mathbf{s}^{(i)} = W\ \sigma\ (V\ \sigma\ (\underbrace{\underbrace{\underbrace{\underbrace{U\mathbf{x}^{(i)}}_{\mathbf{y}^{(i)}}}_{\hat{\mathbf{y}}^{(i)}}}_{\mathbf{z}^{(i)}}}_{\hat{\mathbf{z}}^{(i)}}\ )\ )$$



A neural net is simply a composition of linear and nonlinear functions !

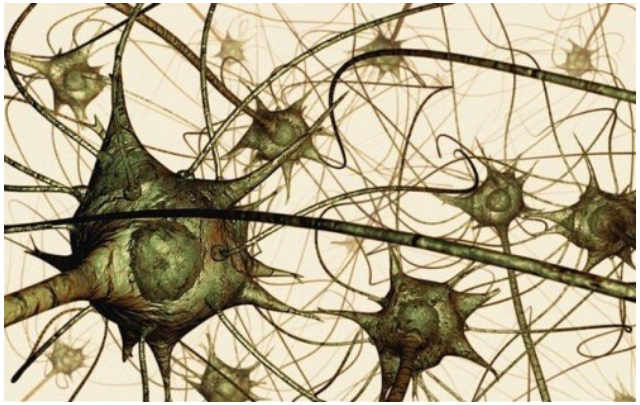
Biological representation

- A basic neuron:



- Brain = neural networks

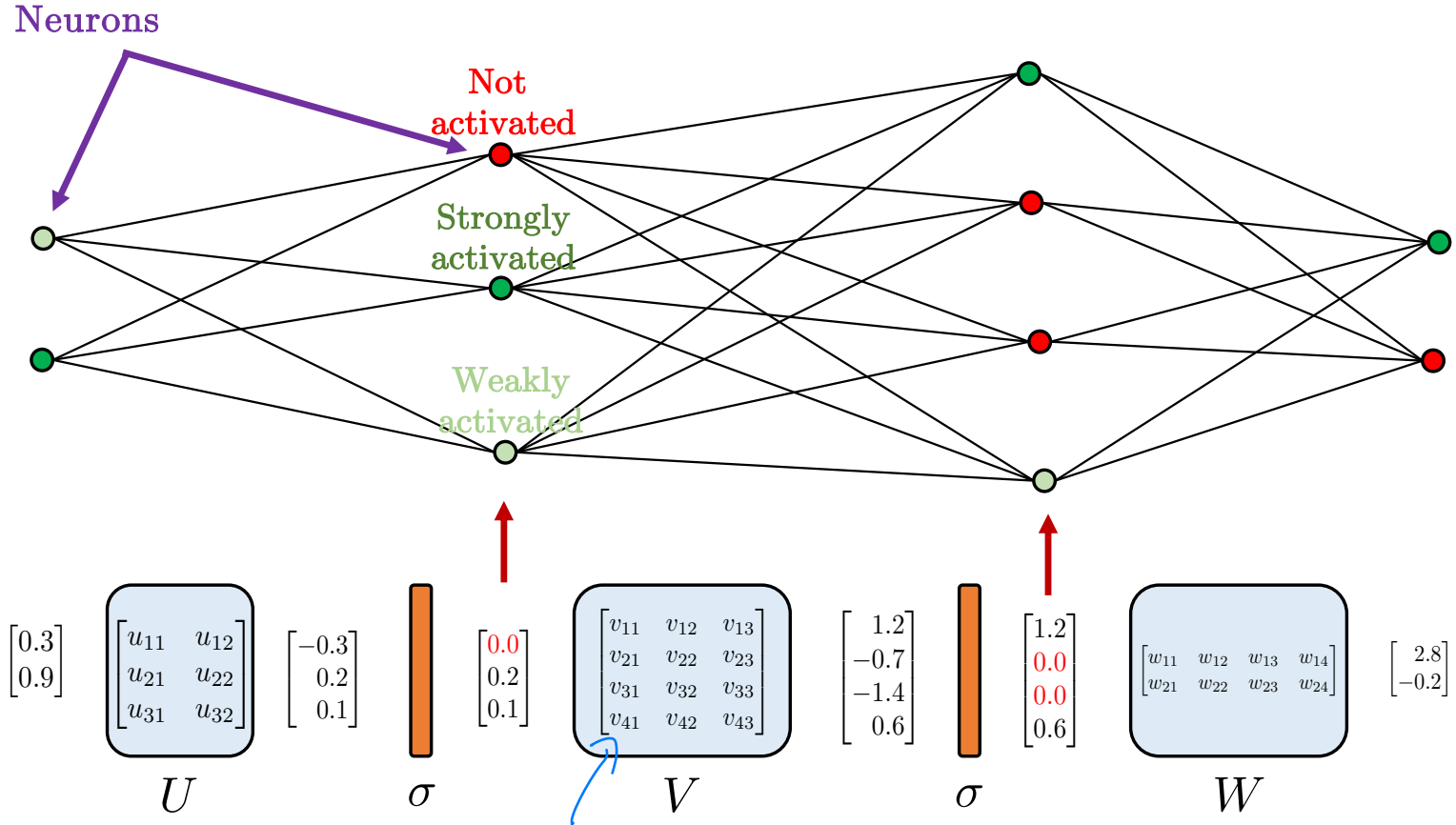
(group of connected neurons)



- A human brain has 10^{11} neurons and 10^{14} axons.

Biological representation

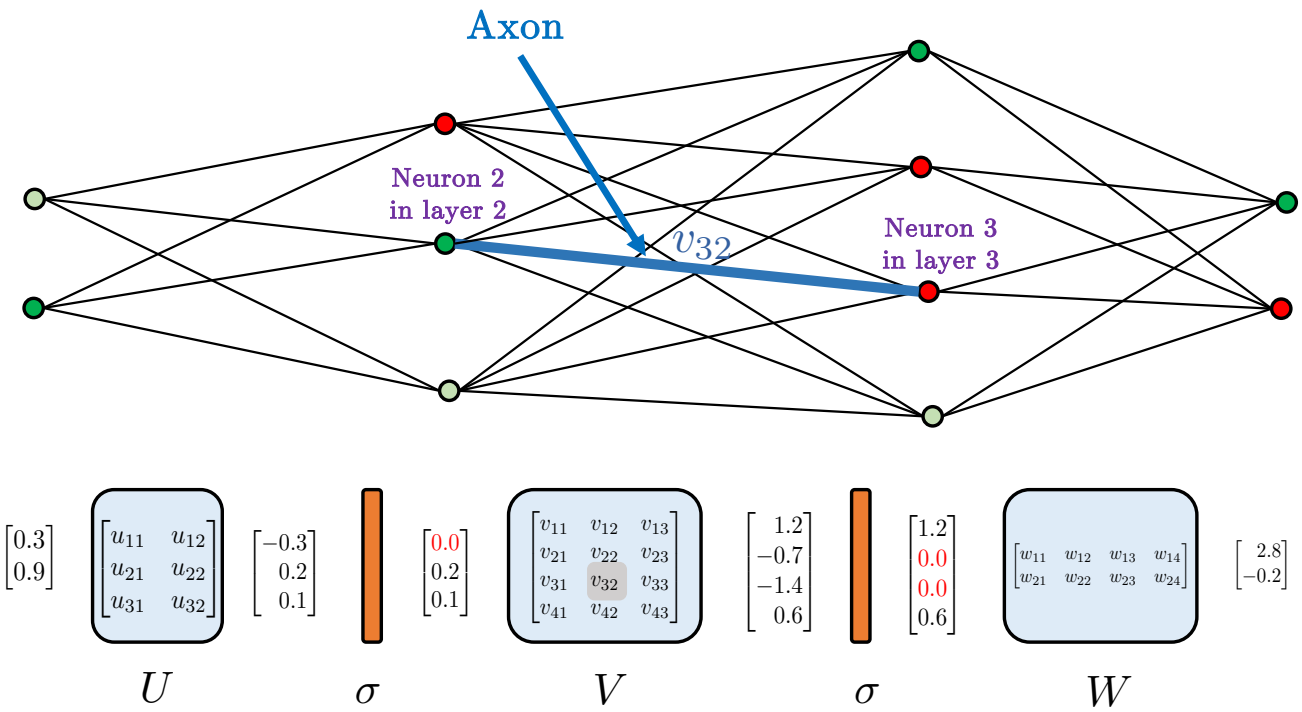
- Biological formulation of a neural network :
  - Neurons



if some weight here is 0, it means there is no connection between the nodes

Biological representation

- Axons :
  - Each weight in the network can be thought as the strength of a neuronal connection.
  - Weight v_{32} = strength of the connection between neuron 2 in layer 2 and neuron 3 in layer 3.



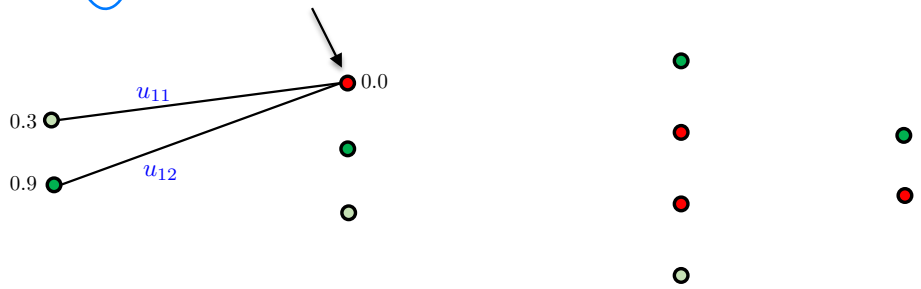
Biological representation

non-linear activation.

Recall: For $\sigma = \mathrm{Relu}(\cdot)$, σ acts as a "minimum activation threshold"

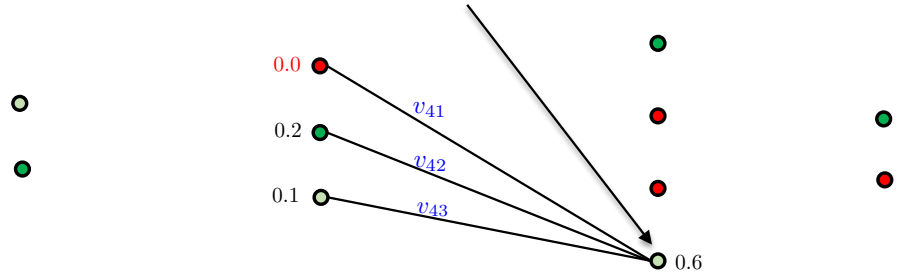
- **Biological representation of an activation :**

$$\sigma\,(\; u_{11}\; 0.3 \;+\; u_{12}\; 0.9 \;) \;=\; 0.0$$



$$\begin{bmatrix}0.3\\0.9\end{bmatrix} \quad \begin{bmatrix}u_{11} & u_{12}\\ u_{21} & u_{22}\\ u_{31} & u_{32}\end{bmatrix} \quad \begin{bmatrix}-0.3\\0.2\\0.1\end{bmatrix} \quad \sigma \quad \begin{bmatrix}0.0\\0.2\\0.1\end{bmatrix} \quad \begin{bmatrix}v_{11} & v_{12} & v_{13}\\ v_{21} & v_{22} & v_{23}\\ v_{31} & v_{32} & v_{33}\\ v_{41} & v_{42} & v_{43}\end{bmatrix} \quad \begin{bmatrix}1.2\\-0.7\\-1.4\\0.6\end{bmatrix} \quad \sigma \quad \begin{bmatrix}1.2\\0.0\\0.0\\0.6\end{bmatrix} \quad \begin{bmatrix}w_{11} & w_{12} & w_{13} & w_{14}\\ w_{21} & w_{22} & w_{23} & w_{24}\end{bmatrix} \quad \begin{bmatrix}2.8\\-0.2\end{bmatrix}$$

$$\sigma\,(\; v_{41}\; 0.0 \;+\; v_{42}\; 0.2 \;+\; v_{43}\; 0.1 \;) \;=\; 0.6$$

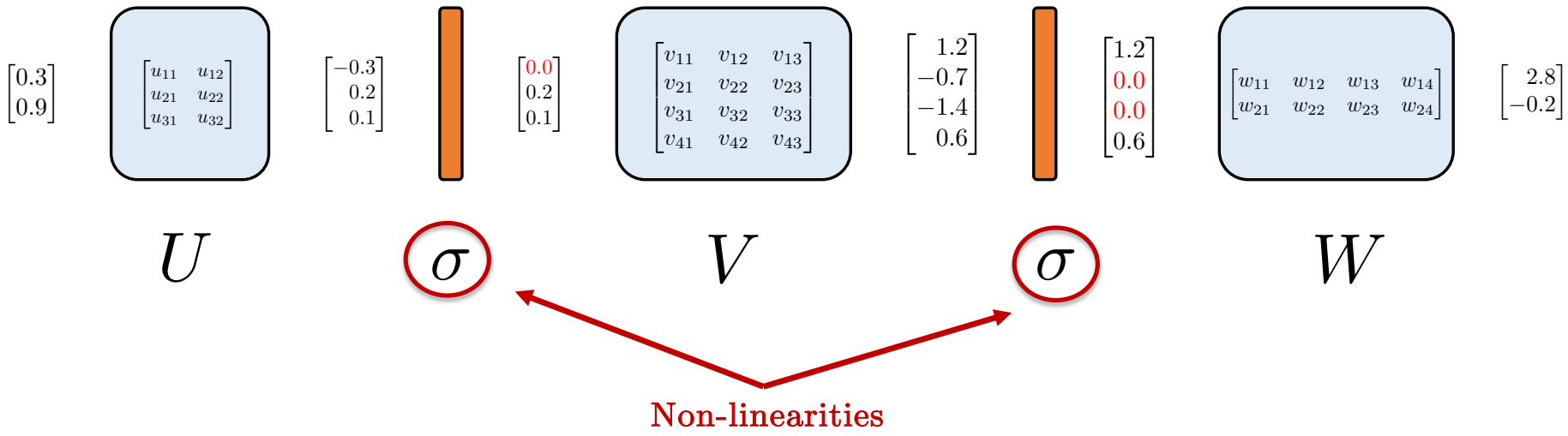


$$\begin{bmatrix}0.3\\0.9\end{bmatrix} \quad \begin{bmatrix}u_{11} & u_{12}\\ u_{21} & u_{22}\\ u_{31} & u_{32}\end{bmatrix} \quad \begin{bmatrix}-0.3\\0.2\\0.1\end{bmatrix} \quad \sigma \quad \begin{bmatrix}0.0\\0.2\\0.1\end{bmatrix} \quad \begin{bmatrix}v_{11} & v_{12} & v_{13}\\ v_{21} & v_{22} & v_{23}\\ v_{31} & v_{32} & v_{33}\\ v_{41} & v_{42} & v_{43}\end{bmatrix} \quad \begin{bmatrix}1.2\\-0.7\\-1.4\\0.6\end{bmatrix} \quad \sigma \quad \begin{bmatrix}1.2\\0.0\\0.0\\0.6\end{bmatrix} \quad \begin{bmatrix}w_{11} & w_{12} & w_{13} & w_{14}\\ w_{21} & w_{22} & w_{23} & w_{24}\end{bmatrix} \quad \begin{bmatrix}2.8\\-0.2\end{bmatrix}$$

Outline

- The three representations of neural networks
- **The importance of non-linearities**
- The importance of multiple layers
- Labs with MNIST and CIFAR

The need of non-linearities



- Why non-linearities are introduced between layers?
  - Biological motivation
  - Mathematical motivation

Mathematical motivation

- Without non-linearities, the network degenerates as a linear model.

With nonlinearities: $$\mathbf{s}^{(\mathbf{i})} = W\sigma(\ V\sigma(\ U\mathbf{x}^{(\mathbf{i})}\ )\ )$$

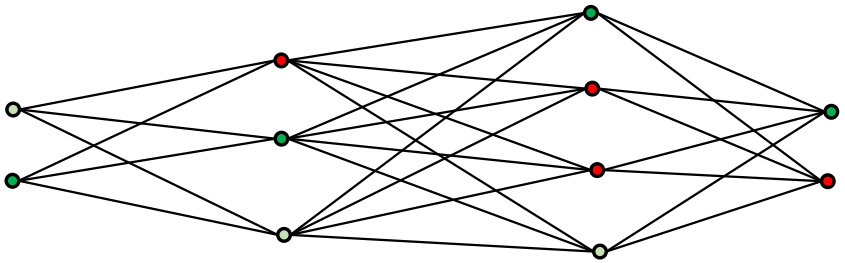
Without nonlinearities: $$\mathbf{s}^{(\mathbf{i})} = \underbrace{W \quad V \quad U}_{A}\mathbf{x}^{(\mathbf{i})}$$

A product of matrices is just another matrix :

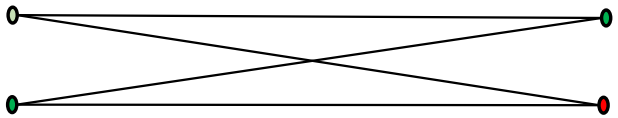
So the network collapses to a one-layer network !

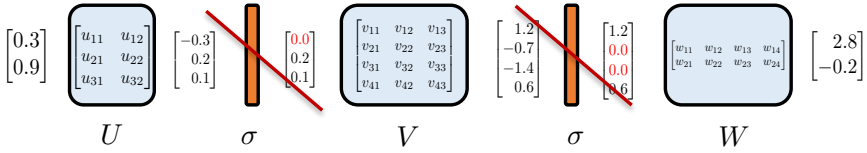
Mathematical motivation

- Without non-linearities:

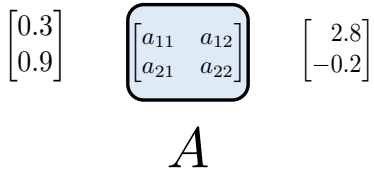


$\Longrightarrow$





$\Longrightarrow$



$$\mathbf{s}^{(\mathbf{i})} = W \cancel{\sigma}(V \cancel{\sigma}(U\mathbf{x}^{(\mathbf{i})}) \;)$$

$\Longrightarrow$

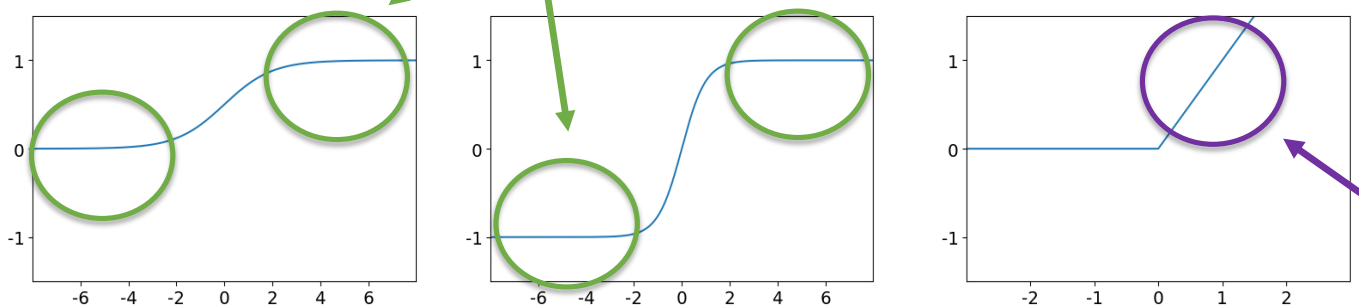
$$\mathbf{s}^{(\mathbf{i})} = A\mathbf{x}^{(\mathbf{i})}$$

Limitations of sigmoid and tanh

- **Issues with Sigmoid and Tanh:**
  - They are responsible for the vanishing gradient problem ⇒ Weights will not be updated.

Gradient is zero.

Propagated gradients even for high activations will be 0.



Gradient is not zero.

Sigmoid

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

Tanh

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

ReLU

$$\sigma(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

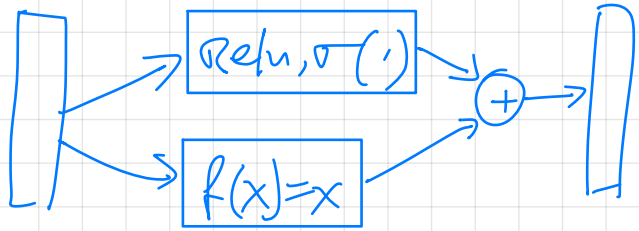
ReLU properties:

1) Non linear → Expressivity of NN is not reduced to 1 layer

2) Gradient = 0 on the left: ∴ Gradients for negative activations can never backpropagate
→ ∴ Weights for these activations will <u>NOT</u> update

Fact: Any non-linear function will have vanishing gradient problem

Fix: Just add a linear function to your non-linear function.
↳ a.k.a. skip/residual connections



? Why

Isn't LeakyReLU non-vanishing?

Outline

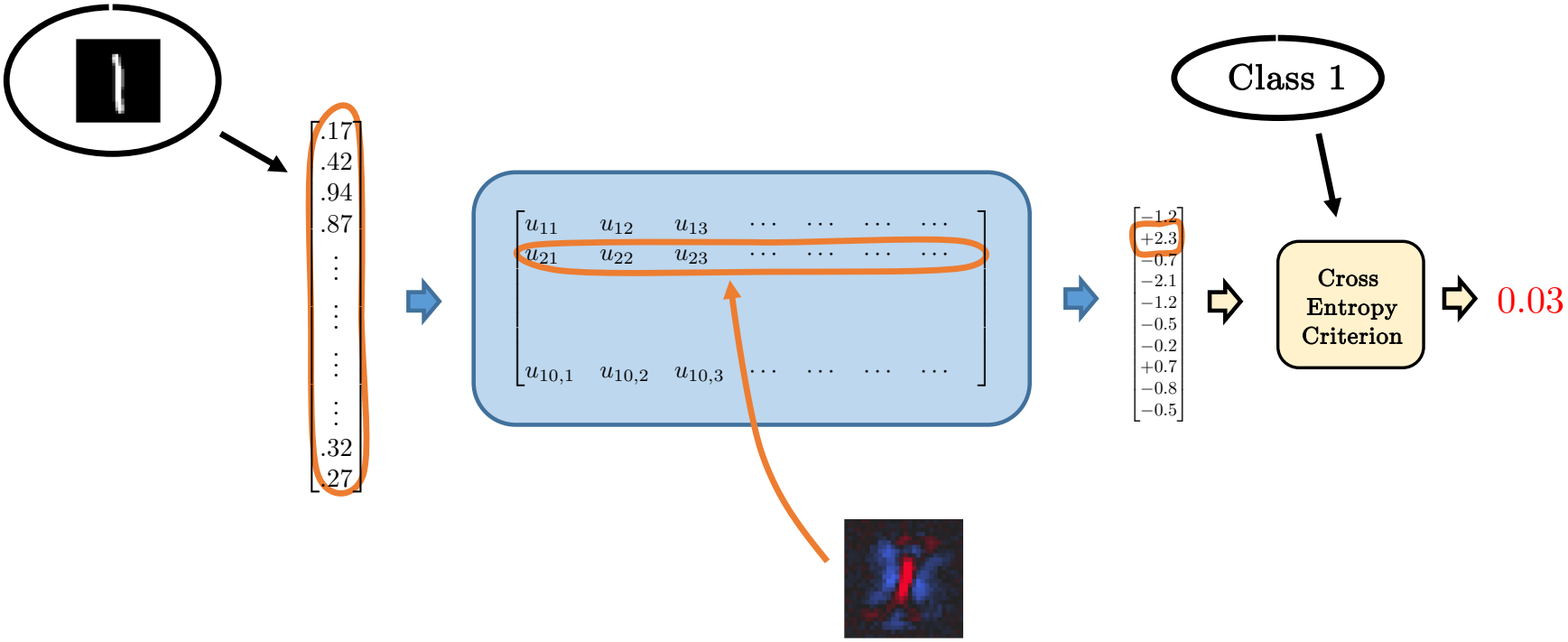
- The three representations of neural networks
- The importance of non-linearities
- **The importance of multiple layers**
- Labs with MNIST and CIFAR

An example

- Let us consider two classification problems to demonstrate the need of multiple layers :
  - An easy classification problem
  - A "hard" classification problem

An easy classification problem

- Classify the digit numbers 0,1,2...9 :
  - One-layer net with 10 classes is enough to solve this classification problem.

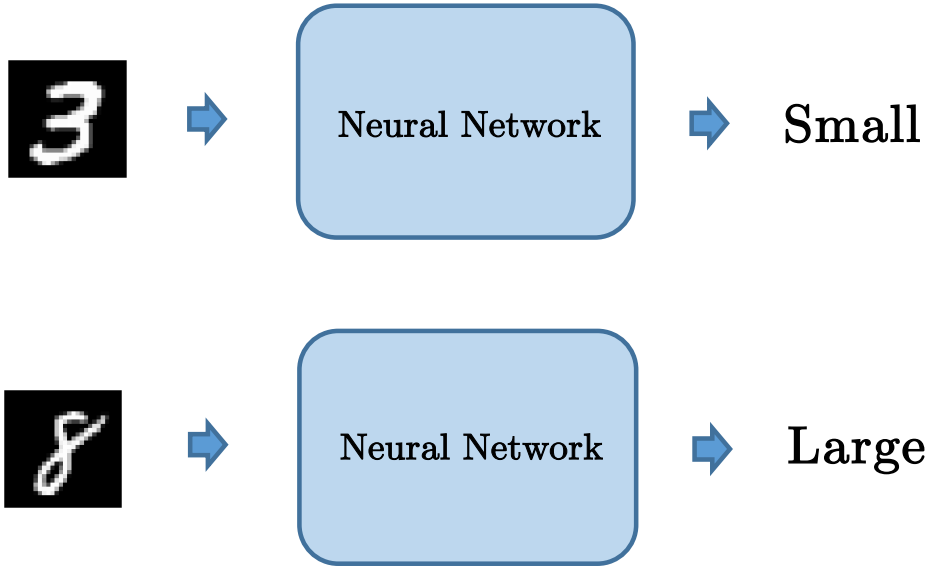


A hard classification problem

- A harder classification problem:
  - Classify small vs large digits into 2 categories

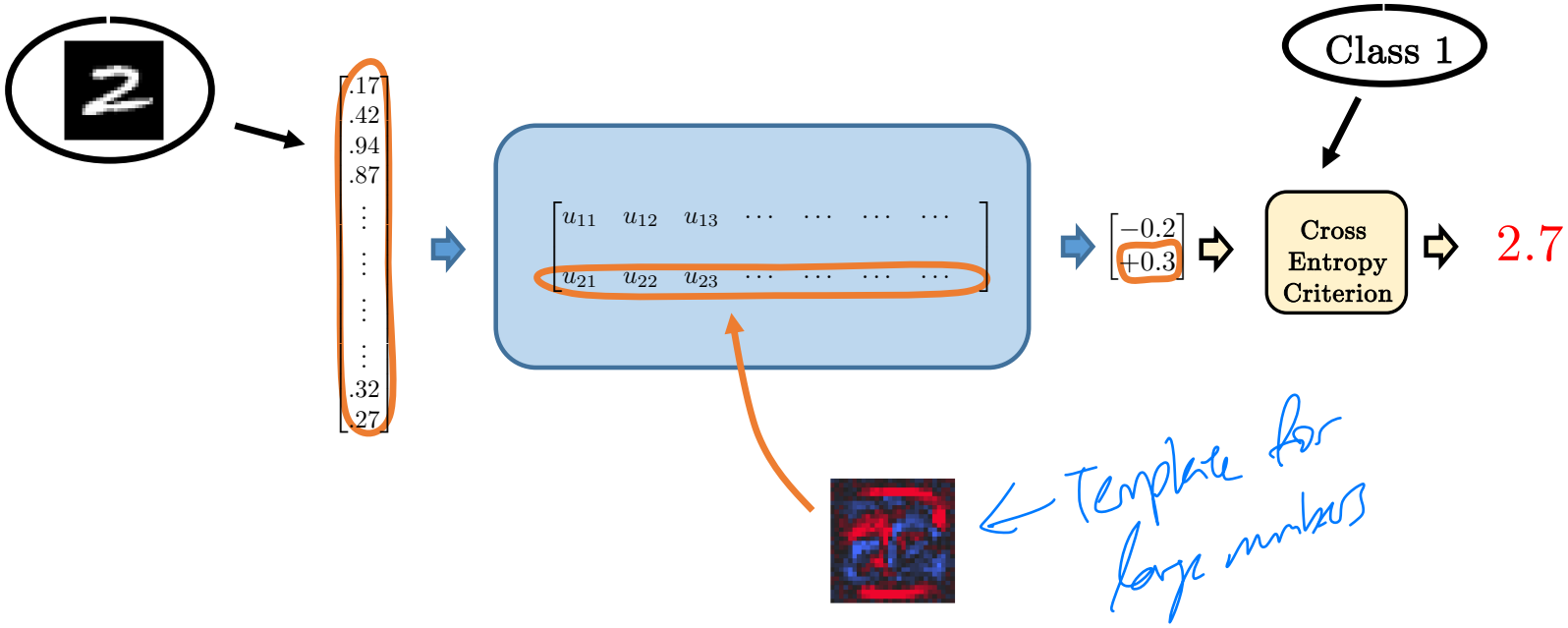
Class 1: Small (0,1,2,3,4)

Class 2: Large (5,6,7,8,9)



A hard classification problem

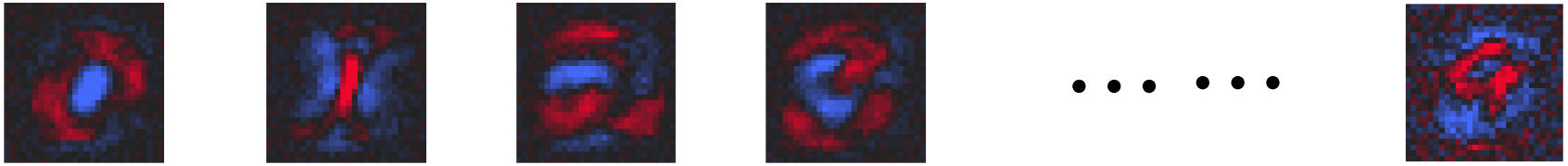
- Classify small vs large digits into 2 categories :
  - One-layer net with 2 classes is not enough to solve this classification problem.



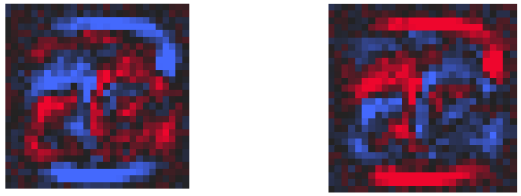
This pattern tries to represent digits 5-9 simultaneously with no success.

Template matching

- It is easy to find good templates for zero, one, two, three, ..., nine :

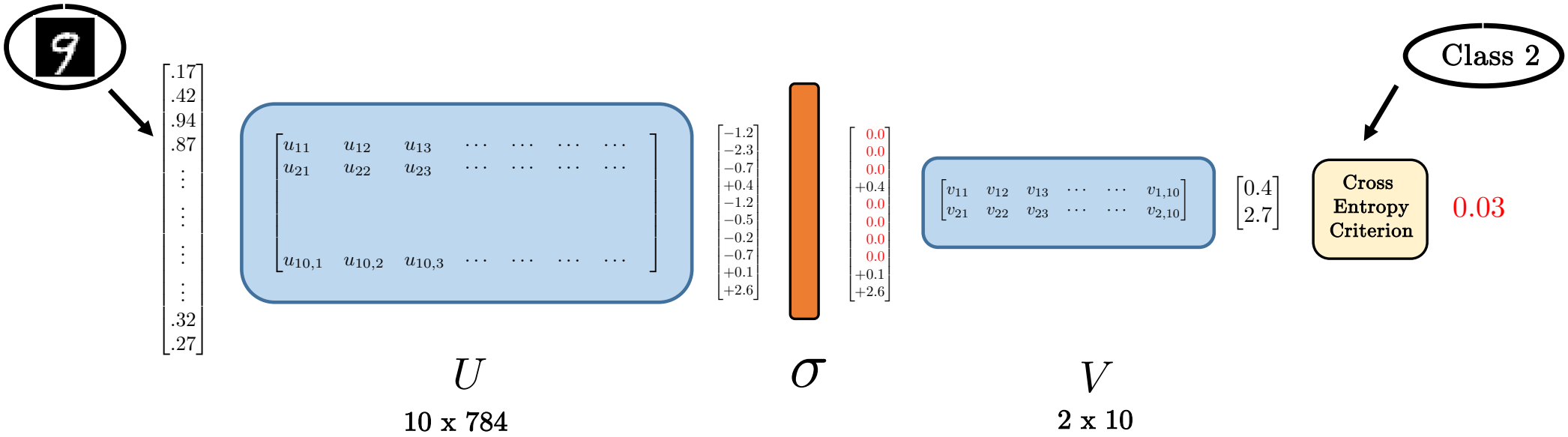


- It is hard to find good templates for small versus large numbers :



A hard classification problem

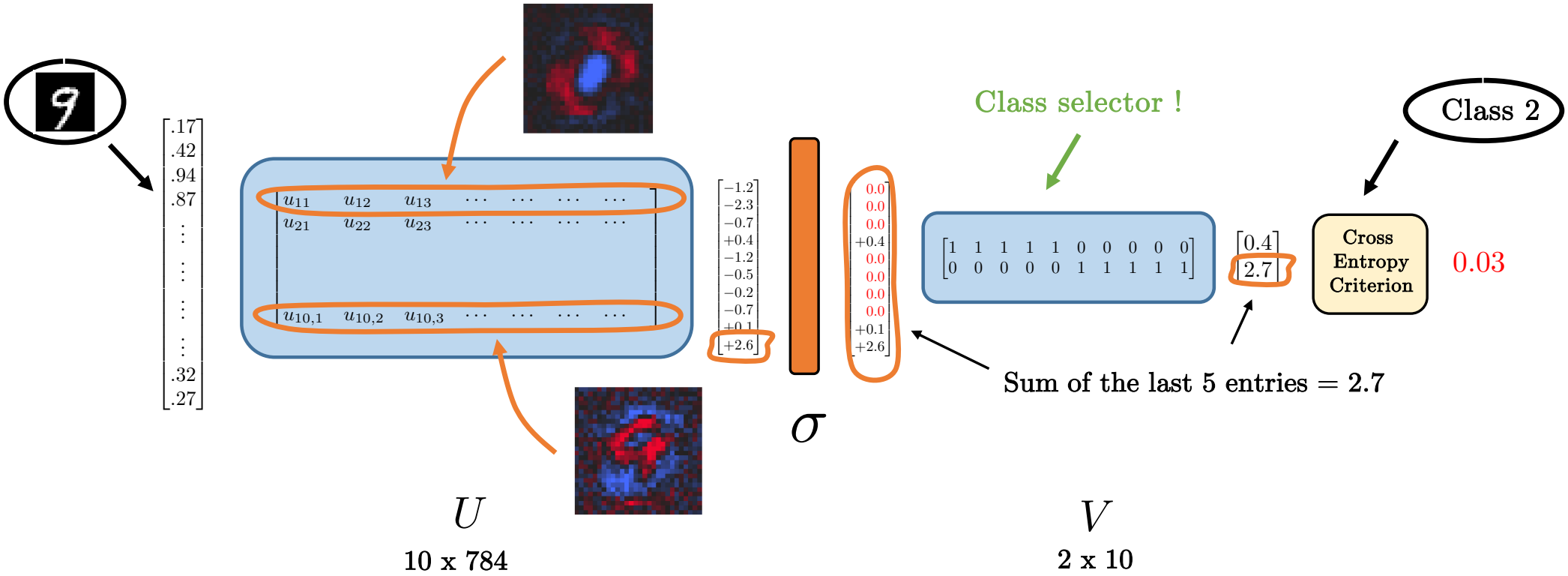
- Let's consider a two-layer network rather than just one-layer :



Input Size = 784 Hidden Size = 10 Output Size =2

A hard classification problem

- After training, the network parameters U and V will become as follows :



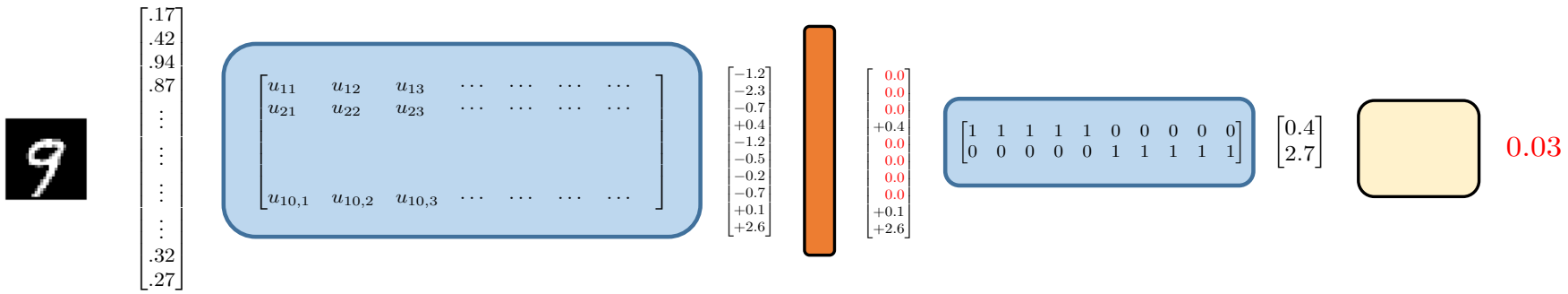
This network will solve this classification problem.

In-lecture question

- Explain why a one-layer neural network is insufficient for classifying digits as “small” (0, 1, 2, 3, 4) versus “large” (5, 6, 7, 8, 9), whereas a multiple-layer network can solve this problem.
- Answer :
- Limitation of one-layer networks: A one-layer network typically relies on “template matching”. In this ”complex” problem, it fails because it cannot create 2 single patterns that represent digits 0-4 and 5-9 simultaneously.
- Multiple-step reasoning: Multiple layers allow the network to break the problem into stages:
- First Layer: This layer is used to identify the specific type of digit (0, 1, 2, ..., 9).
- Second Layer: This layer takes the classifications from the first layer and processes them further by “pooling” together digits 0–4 into one group and 5–9 into another.
- Conclusion: By adding depth, the network can learn more relevant and complex features through hierarchical stages, enabling it to solve “hard” classification problems that a single layer cannot handle.

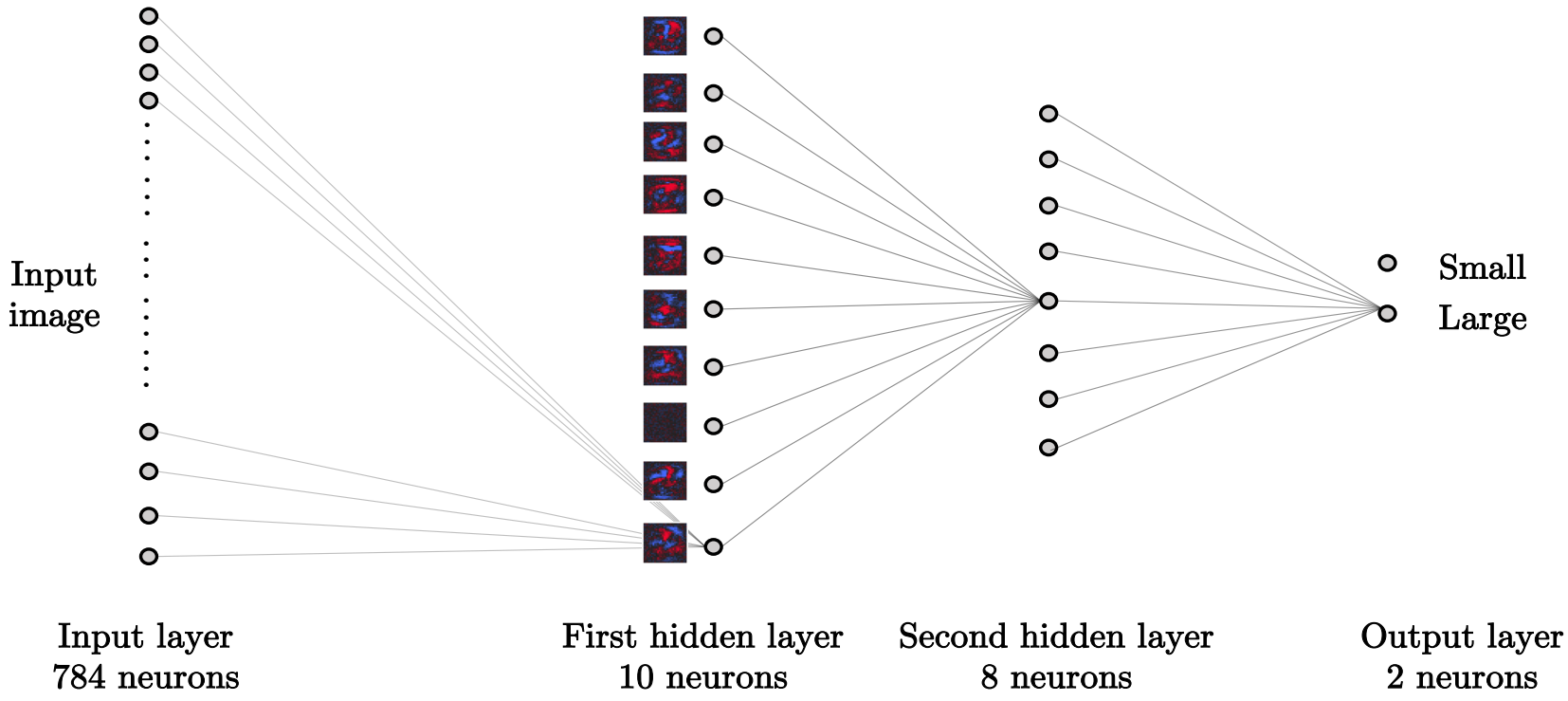
Multiple-step reasoning

- Multiple layers provide multiple-step reasoning :
  - The first layer is used to identify the type of digits
  - The second layer uses the classification made by the first layer, and processes it by pooling together 0-4 into one group, and 5-9 into another group.



Multiple-step reasoning

- What would happen with more layers?
  - The new network would also succeed.
  - The network would learn relevant features and do multiple stage reasoning.
  - The new features will be less/no interpretable (it is fine).

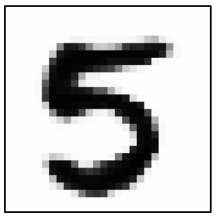


Outline

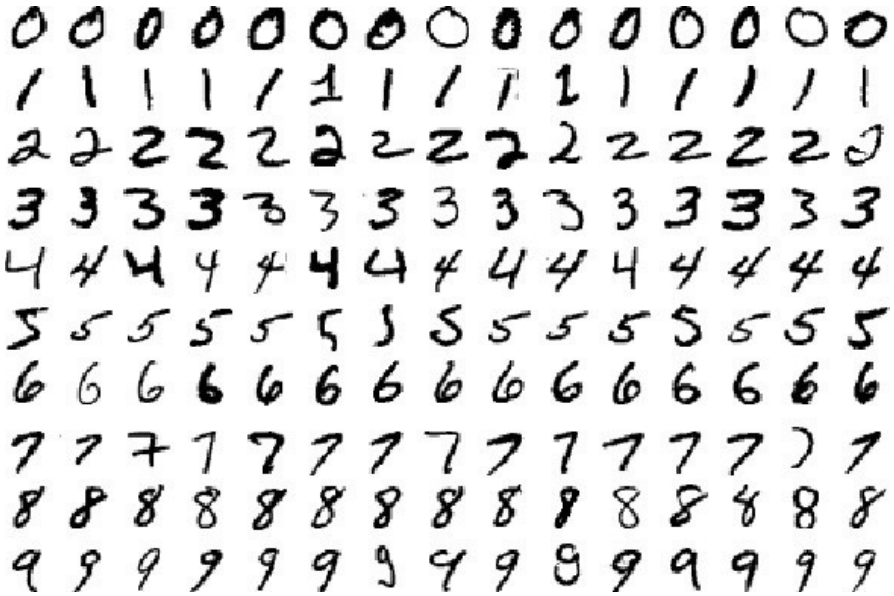
- The three representations of neural networks
- The importance of non-linearities
- The importance of multiple layers
- **Labs with MNIST and CIFAR**

MNIST dataset

- Number of classes: 10
- Training set: 50,000 labelled images
- Test set: 10,000 labelled images



28 x 28 grayscale image



Lab 01

- MNIST classification with MLP

jupyter mnist_multilayer_demo Last Checkpoint: a few seconds ago (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

Lab 01 : MNIST multi-layer -- demo

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from random import randint
import time
import utils
```

Download the data

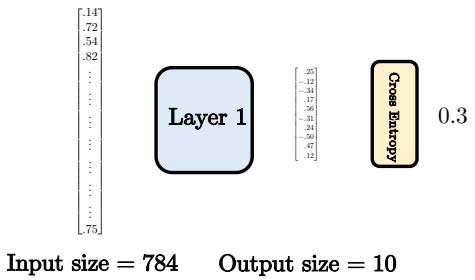
In [2]:
```
from utils import check_mnist_dataset_exists
data_path=check_mnist_dataset_exists()

train_data=torch.load(data_path+'mnist/train_data.pt')
train_label=torch.load(data_path+'mnist/train_label.pt')
test_data=torch.load(data_path+'mnist/test_data.pt')
test_label=torch.load(data_path+'mnist/test_label.pt')
```

Make a two layer net class.

In [3]:
```
class two_layer_net(nn.Module):

    def __init__(self, input_size, hidden_size,  output_size):
        super(two_layer_net , self).__init__()

        self.layer1 = nn.Linear(  input_size   , hidden_size  , bias=False  )
        self.layer2 = nn.Linear(  hidden_size  , output_size   , bias=False  )
```

Lab 01

- Results
  - One-layer network:

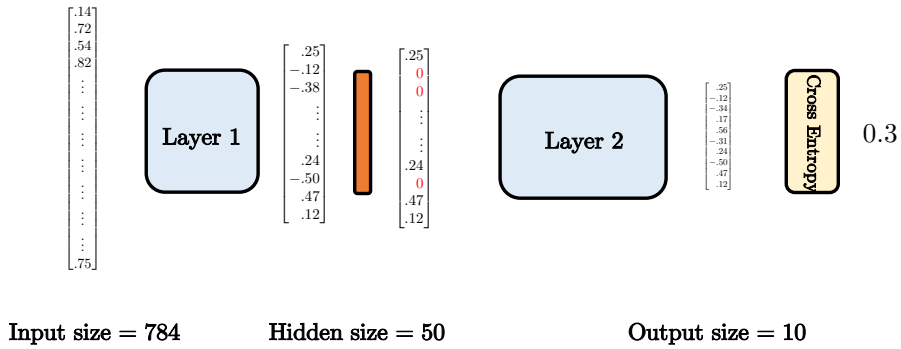


Number of parameters:
784 x 10 = 7,840

Error on train set: 8%

Error on test set: 8%

  - Two-layer network:



Number of parameters:
784 x 50 + 50 x 10 = 39,700

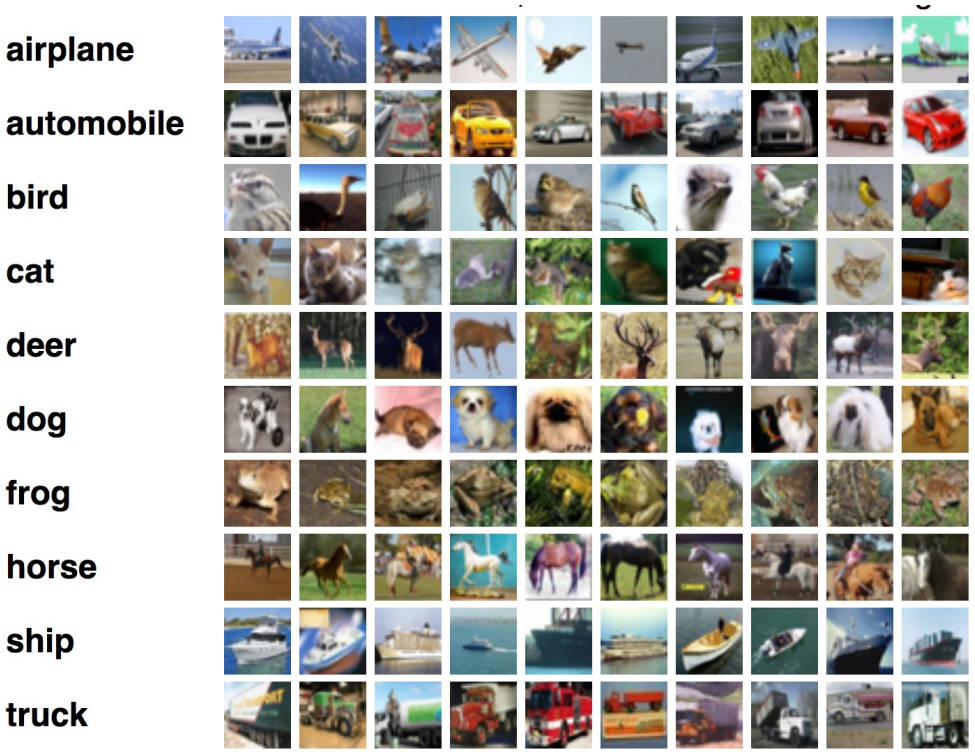
Error on train set: 0%

Error on test set: 2.5%

CIFAR dataset

- Number of classes: 10
- Training set: 50,000 labelled images
- Test set: 10,000 labelled images

32 x 32 RGB image



Lab 02

- CIFAR classification with MLP

jupyter gpu_demo Last Checkpoint: a few seconds ago (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

Lab 02 : GPU demo

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

Set the device variable to be either cuda or cpu

In [9]:
```
device = torch.device("cuda")

#device = torch.device("cpu")

print(device)
```

```
cpu
```

Make a random tensor (by default the tensor is created on the CPU)

In [10]:
```
x=torch.rand(3,3)

print(x)
```

```
tensor([[0.2126, 0.8232, 0.9466],
        [0.0370, 0.5176, 0.7032],
        [0.4544, 0.4457, 0.8515]])
```

Send the tensor to the device (which is going to be be the GPU if the device was set to be the GPU -- otherwise it will remain on the CPU)

jupyter cifar_multilayer_exercise Last Checkpoint: a few seconds ago (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

Lab 02 : CIFAR multi-layer -- exercise

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from random import randint
import utils
import time
```

With or without GPU?

It is recommended to run this code on GPU:

- Time for 1 epoch on CPU : 5 sec
- Time for 1 epoch on GPU : 0.6 sec w/ GeForce GTX 1080 Ti

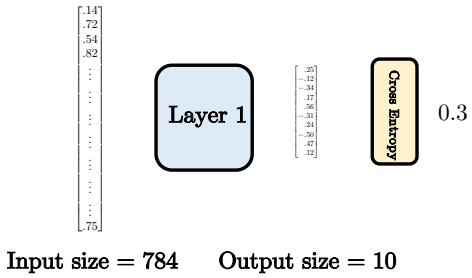
In [ ]:
```
#device= torch.device("cuda")
device= torch.device("cpu")
print(device)
```

Download the CIFAR dataset

In [2]:
```
from utils import check_cifar_dataset_exists
data_path=check_cifar_dataset_exists()

train_data=torch.load(data_path+'cifar/train_data.pt')
train_label=torch.load(data_path+'cifar/train_label.pt')
test_data=torch.load(data_path+'cifar/test_data.pt')
test_label=torch.load(data_path+'cifar/test_label.pt')
```

Lab 02

- Results
  - One-layer network:

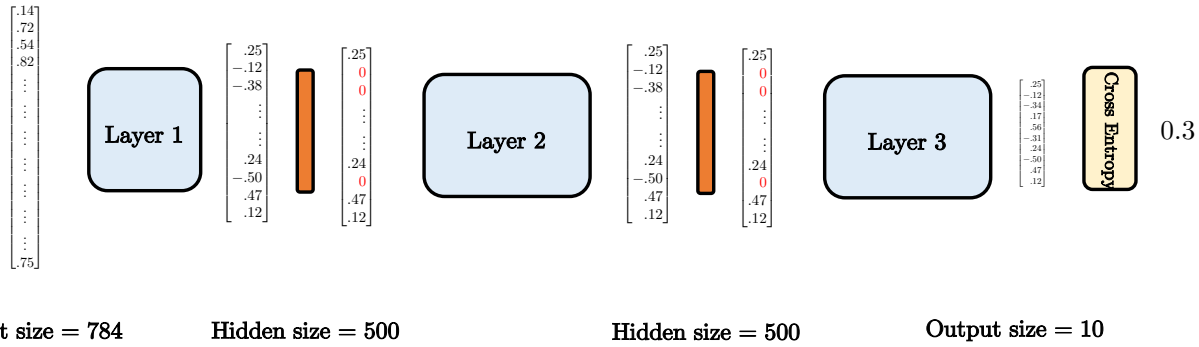


Number of parameters:
3,072 x 10 = 30,720

Error on train set: 59%

Error on test set: 65%

  - Three-layer network:



Number of parameters:
2 millions

Error on train set: 0%

Error on test set: 46%

Questions?

CS5242 : Neural Networks and Deep Learning

Lecture 5 : Convolutional Neural Networks
Part 1 : Introduction

Semester 2 2025/26

Xavier Bresson

https://twitter.com/xbresson

Department of Computer Science
National University of Singapore (NUS)

Outline

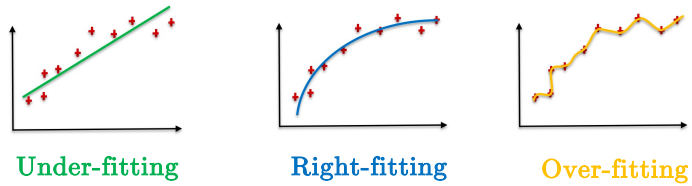
- Data structure
- Local reception fields
- Modeling of hierarchical organization
- One-layer convolutional neural network
- Paradigm shift in computer vision

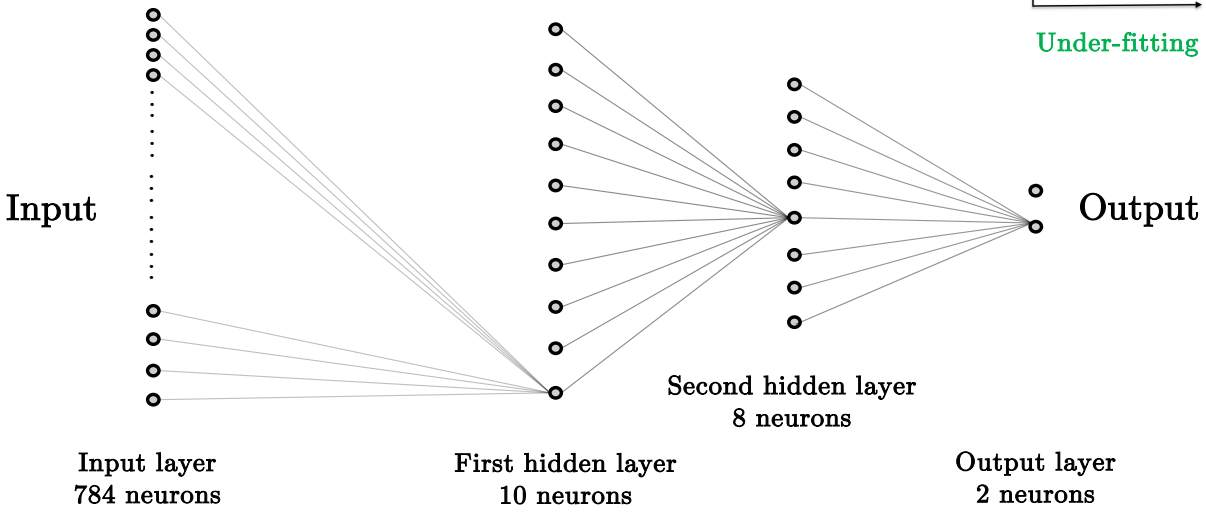
Outline

- Data structure
- Local reception fields
- Modeling of hierarchical organization
- One-layer convolutional neural network
- Paradigm shift in computer vision

Motivation

- MLP (a.k.a. Fully connected networks) :
  - MLP networks do not consider any specific structure of data, only pattern matching.
  - Theoretically, they can learn anything (universal approximation theorem), but they are practically very hard to train (too much parameters, too long).
  - They are pruned to overfitting and do not generalize well.





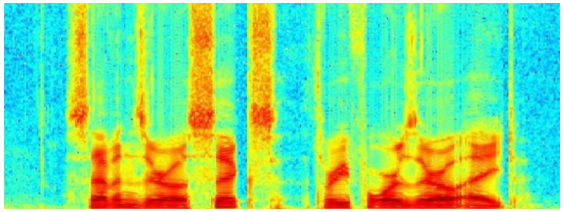
MLP/Fully connected networks

Data structure

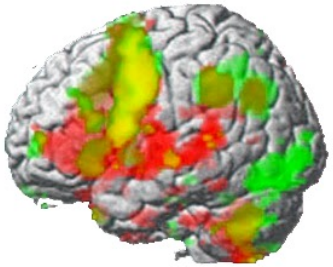
- Observe that data are not random, but exhibit special structures to find and to leverage.
- Invariance/symmetry refer to common and meaningful data structure.
- Identifying fundamental, minimal and universal data structures, that can be encoded into a large number of layers, is the best way to design neural networks.

Image

Video



Spectogram



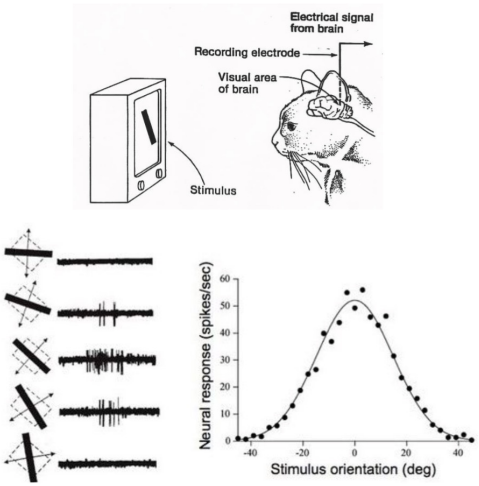
fMRI

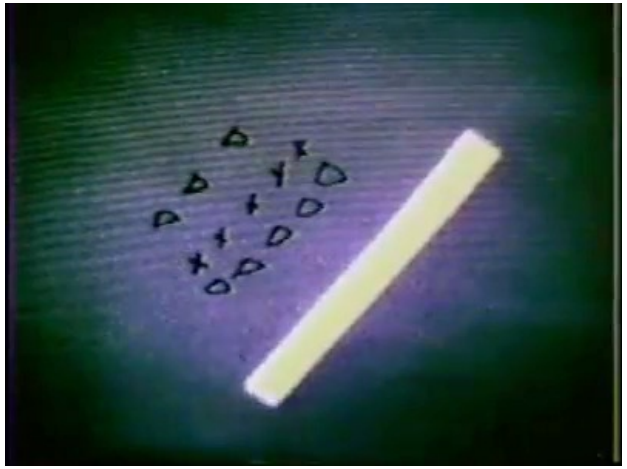
Outline

- Data structures
- Local reception fields
- Modeling of hierarchical organization
- One-layer convolutional neural network
- Paradigm shift in computer vision

Biological reception fields

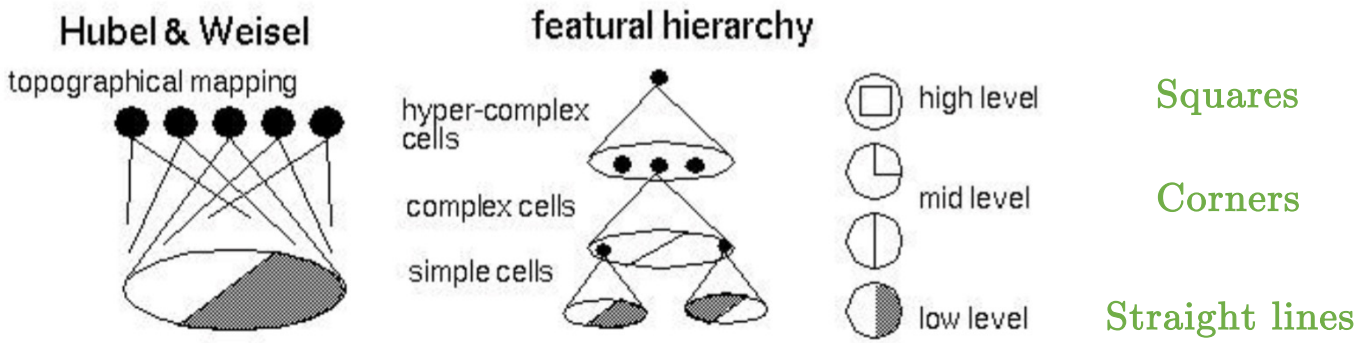
- Hubel and Wiesel 1959 :
  - Nobel Prize in Medicine for the understanding of the primary visual cortex system (first 2 layers):
  - Visual system is composed of receptive fields called V1 cells that are composed of neurons that activate depending on the local orientation.





Biological reception fields

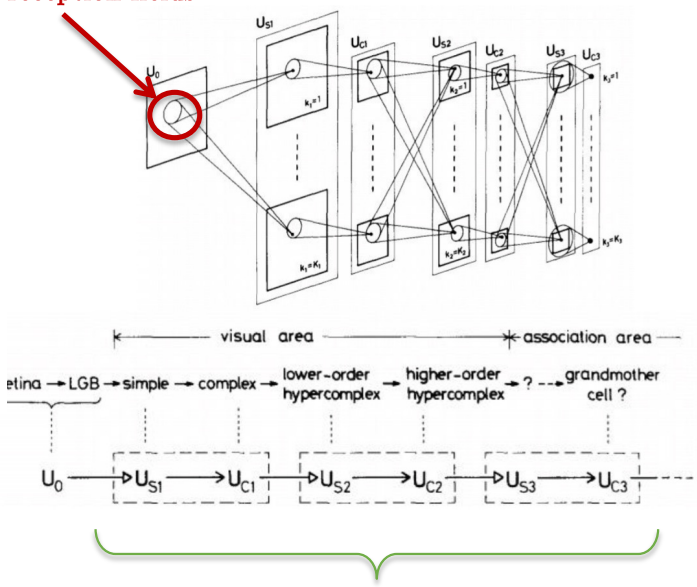
- The second layer of the visual cortex takes and composes outputs of V1 neurons.
  - The second layer neurons are called V2 cells.
  - This forms a hierarchical organization.



Neocognitron

- First algorithm that implements Huber-Wiesel visual recognition system [Fukushima, 1987] :
  - Introduction of concepts of local features to model biological reception fields.
  - Cascade simple and complex cells (V1 and V2 cells) to form hierarchical organization of features
  - Precursor of convolutional neural networks [LeCun, Bottou, Bengio, Haffner, 1998].

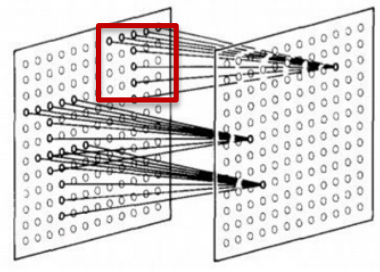
Local reception fields



Stack simple-complex cells

The importance of local features

- Local features model the biological local reception fields :

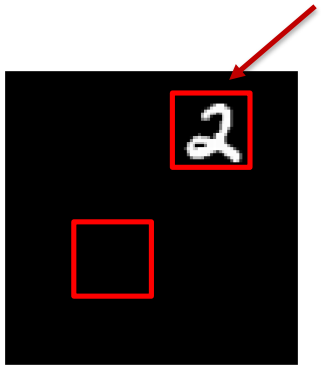


V1 layer V2 layer

Modeling

$\Longrightarrow$

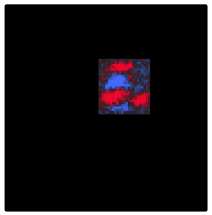
Local reception fields



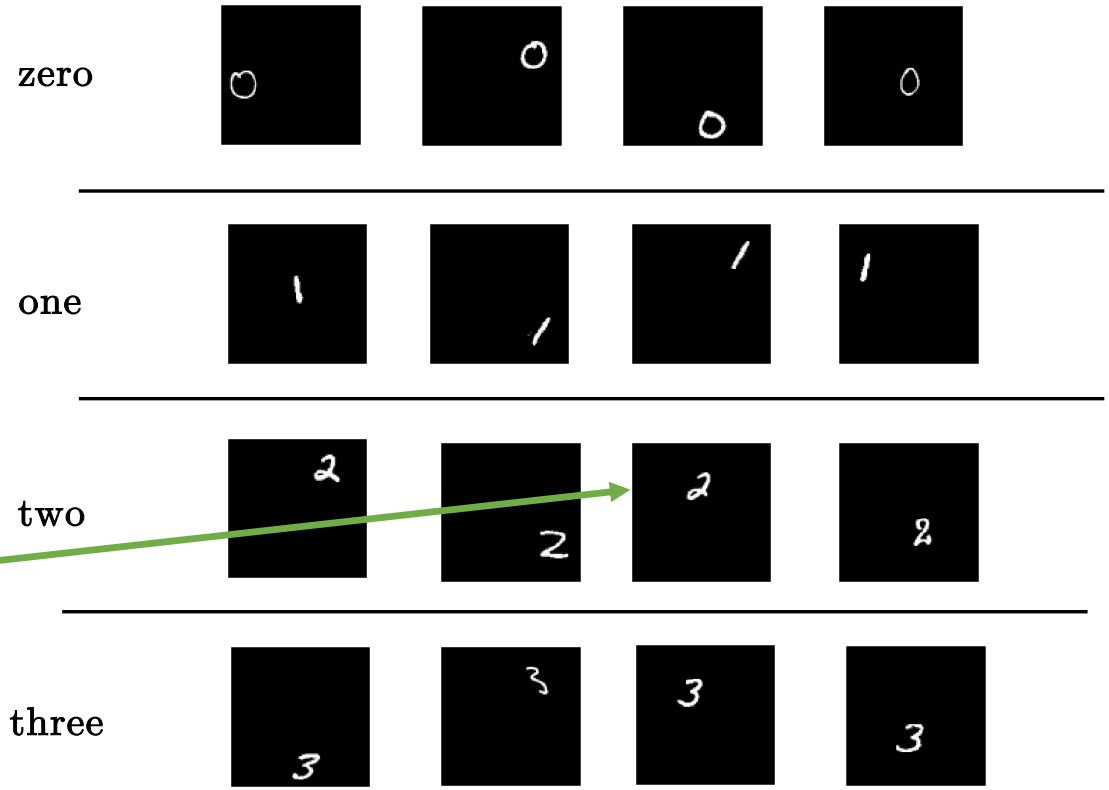
- Why local features are important for image recognition?

The importance of local features

- What if images are not well centered?
  - Suppose the handwritten digits are not in the center of the image.
  - It is not possible to find a single template for all the two's !



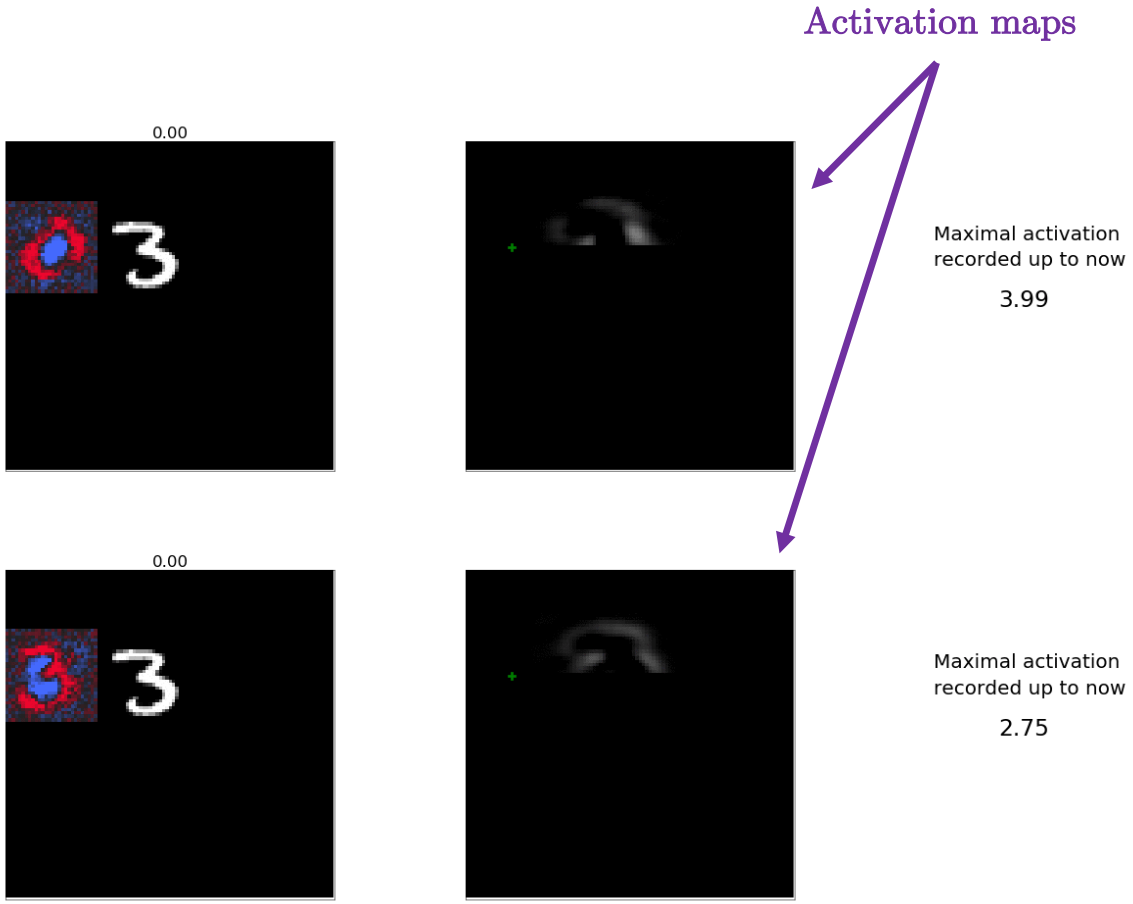
This template will only be useful for this image.

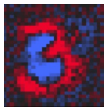


  - It is intractable to design a pattern for all locations in the image.

Detecting local features

- Sliding templates :
  - Define small templates and scan them across the image domain.
  - This operation is known as convolution.





Sliding template

In-lecture question

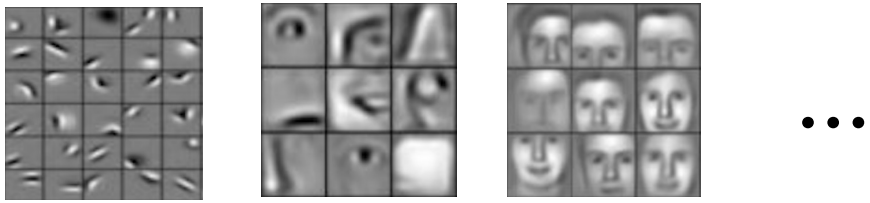
- How do Convolutional Neural Networks (CNNs) improve the limitations of Multi-Layer Perceptrons (MLPs) for image recognition tasks? Specifically, how does convolution address the issue of parameter count?
- Answer : MLPs only consider pattern matching.
  - For MLPs to work for all possible translations in the image domain, it would require a unique template for every possible location in the image.
  - This is intractable as it would require too many parameters, making MLPs practically very hard and time-consuming to train. And even if we have an infinite number of parameters, MLPs will overfit/memorize, and will not generalize well to new data.
- CNNs address these issues through Local Receptive Fields and Convolution:
  - Local Features: Instead of looking at the whole image at once, CNNs use small "sliding templates" (a.k.a. filters) that scan across the image domain to detect local features.
  - Parameter Efficiency: By using small templates (e.g. 3 x 3), the network significantly reduces the number of parameters compared to a MLP.
  - Translation Invariance: This "inductive bias" makes CNNs generalize well as they capture a universal property of visual data.

Outline

- Data structures
- Local reception fields
- **Modeling of hierarchical organization**
- One-layer convolutional neural network
- Paradigm shift in computer vision

Hierarchical features

- Local feature patterns can be composed to form abstract complex patterns :



Layer 1 Layer 2 Layer 3 Layer 4

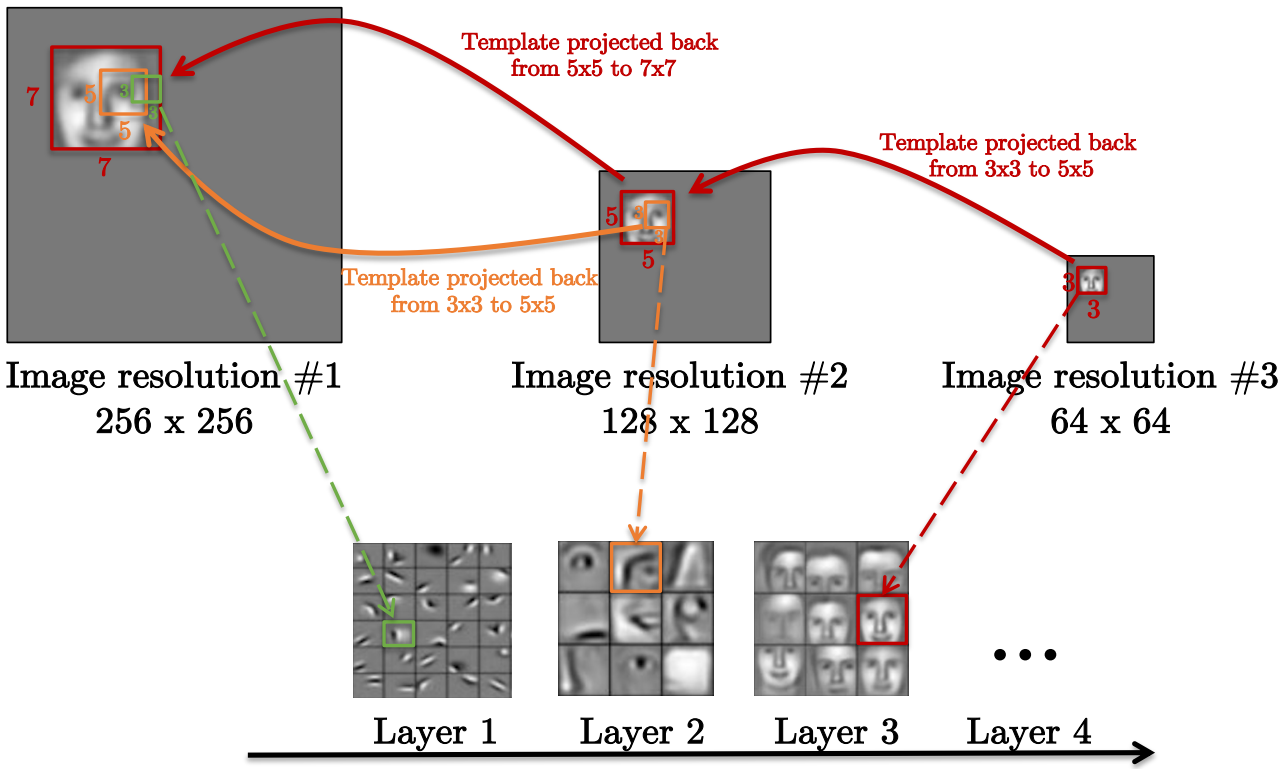
Deep/hierarchical features

(simple to abstract concepts)

- Image data is compositional :
  - It is formed from hierarchical local stationary patterns.

Hierarchical features

- How to extract hierarchical patterns (low-level to high-level features)?
  - Hierarchical patterns are multi-scale features.
    - They are captured by 3x3 templates at each image resolution :



We will only learn 3x3 templates !

At Image resolution #1 : 3 3

At Image resolution #2 : 3 3

At Image resolution #3 : 3 3

Hierarchical features

- How to change image resolution?
  - Down-sampling and
  - Pooling (for examples max pooling, average pooling)

How does gradient pass through ?

3	3	0	1	1	1
7	2	1	1	5	2
7	9	1	0	5	6
5	6	1	0	3	2
9	7	0	0	0	0
4	7	0	0	0	1

Down-sampling
+ 2x2 max pooling

$\Longrightarrow$

7	1	5
9	1	6
9	0	1

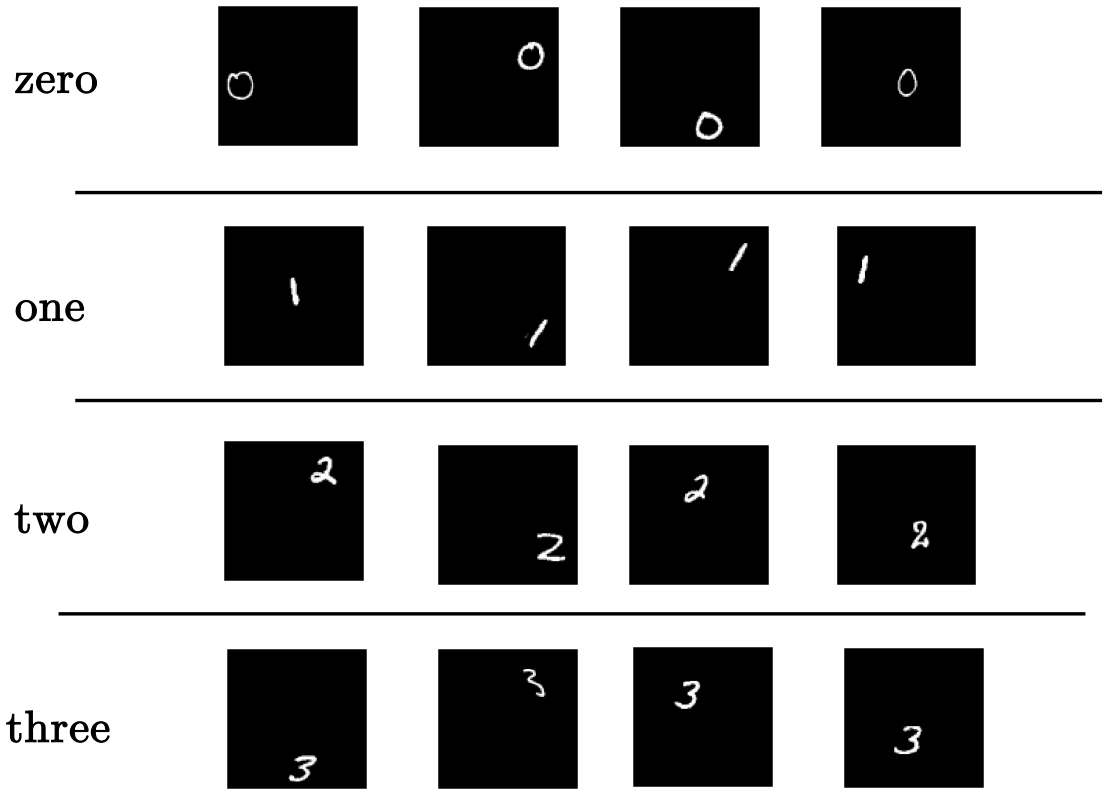
Outline

- Data structures
- Local reception fields
- Modeling of hierarchical organization
- **One-layer convolutional neural network**
- Paradigm shift in computer vision

One-layer ConvNet

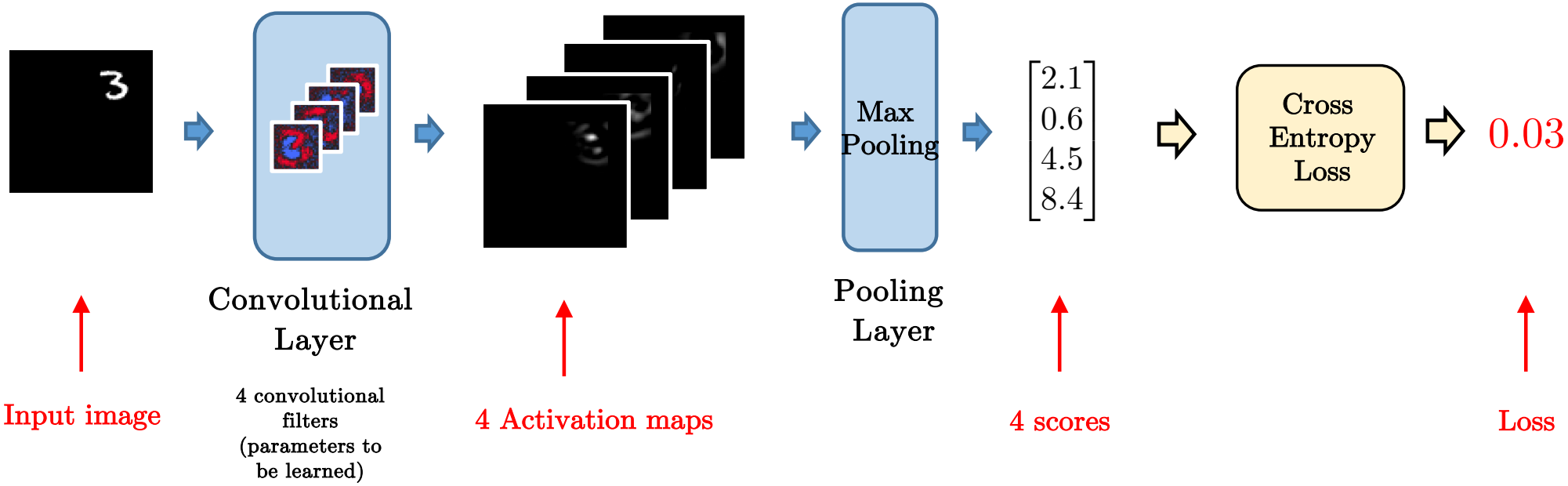
- Assume there are only four classes :

  (0,1,2,3)



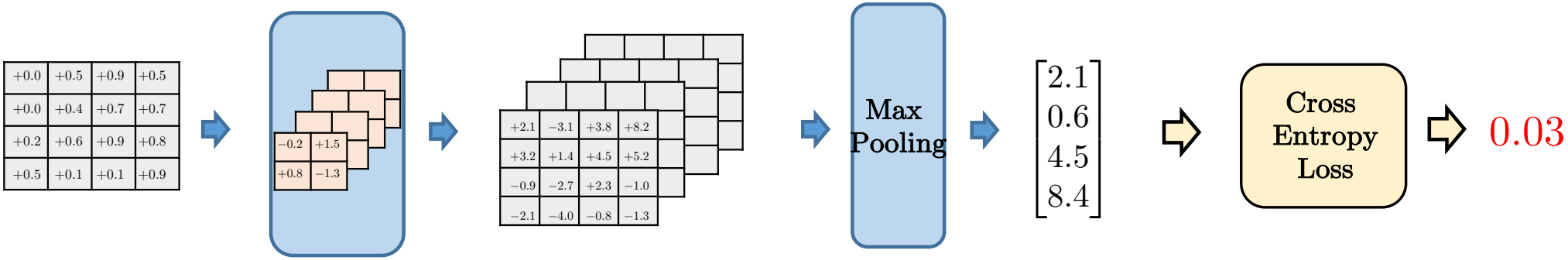
One-layer ConvNet

- Architecture : 1 CL + MP + Loss



One-layer ConvNet

- Tensor representation :



Convolutional Layer

Pooling Layer

Input image

100 x 100 tensor

4 convolutional filters (parameters to be learned)

4 x 28 x 28 tensor

4 Activation maps

4 x 100 x 100 tensor

4 scores

4 x 1 tensor

Loss

scalar

CS5242 : Neural Networks and Deep Learning

Lecture 5 : Convolutional Neural Networks
Part 2 : Implementation

Semester 2 2024/25

Xavier Bresson

https://twitter.com/xbresson

Department of Computer Science
National University of Singapore (NUS)

Outline

- Padding
- Stride
- Multi-dimensional input
- Convolution as linear operation
- Depth
- Convolutional layer with PyTorch
- Pooling layer with PyTorch
- LeNet5 architecture
- VGG architecture
- Case studies

Outline

- **Padding**
- Stride
- Multi-dimensional input
- Convolution as linear operation
- Depth
- Convolutional layer with PyTorch
- Pooling layer with PyTorch
- LeNet5 architecture
- VGG architecture
- Case studies

Padding

- Convolution operation shrinks the size of the image :

$$(3)(0) + (2)(1) + (3)(0) + (0)(0) + (1)(0) + (3)(0) + (2)(1) + (2)(1) + (0)(2) = 6$$

$$\begin{bmatrix} 3 & 2 & 3 & 3 & 3 & 1 \\ 0 & 1 & 3 & 3 & 1 & 2 \\ 2 & 2 & 0 & 0 & 2 & 0 \\ 3 & 0 & 1 & 3 & 3 & 0 \\ 2 & 3 & 2 & 0 & 0 & 1 \\ 2 & 2 & 2 & 3 & 1 & 0 \end{bmatrix} \star \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 6 & 5 & 7 & 5 \\ 6 & 10 & 13 & 7 \\ 11 & 5 & 2 & 4 \\ 8 & 11 & 10 & 7 \end{bmatrix}$$

Input image

6 x 6

Convolutional filter

3 x 3

Output image/
Activation map

4 x 4

Padding

- Padding input image with zeros allows to have the same size for the input and the output.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 2 & 3 & 3 & 3 & 1 & 0 \\ 0 & 0 & 1 & 3 & 3 & 1 & 2 & 0 \\ 0 & 2 & 2 & 0 & 0 & 2 & 0 & 0 \\ 0 & 3 & 0 & 1 & 3 & 3 & 0 & 0 \\ 0 & 2 & 3 & 2 & 0 & 0 & 1 & 0 \\ 0 & 2 & 2 & 2 & 3 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \star \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 7 & 10 & 8 & 8 & 3 \\ 9 & 6 & 5 & 7 & 5 & 3 \\ 3 & 6 & 10 & 13 & 7 & 5 \\ 10 & 11 & 5 & 2 & 4 & 1 \\ 9 & 8 & 11 & 10 & 7 & 1 \\ 2 & 3 & 2 & 0 & 0 & 1 \end{bmatrix}$$

Input image 6 x 6
padded with zeros
8 x 8

Convolutional filter
3 x 3

Activation map/
output
6 x 6

Padding

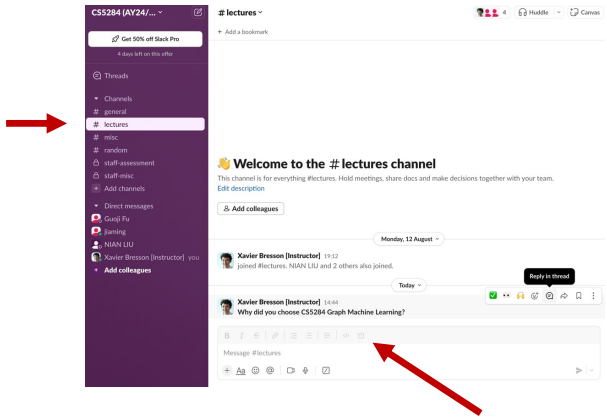
- Padding size depends on the filter size :
  - When using 5x5 filters (radius=2), image must be padded with 2 layers of zeros.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 2 & 3 & 3 & 3 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 3 & 3 & 1 & 2 & 0 & 0 \\ 0 & 0 & 2 & 2 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 1 & 3 & 3 & 0 & 0 & 0 \\ 0 & 0 & 2 & 3 & 2 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 2 & 2 & 3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \star \begin{bmatrix} 1 & 2 & 1 & 1 & 2 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 2 & 1 \\ 1 & 2 & 1 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 7 & 10 & 8 & 8 & 3 \\ 9 & 6 & 5 & 7 & 5 & 3 \\ 3 & 6 & 10 & 13 & 7 & 5 \\ 10 & 11 & 5 & 2 & 4 & 1 \\ 9 & 8 & 11 & 10 & 7 & 1 \\ 2 & 3 & 2 & 0 & 0 & 1 \end{bmatrix}$$

  - When using 7x7 filters (radius=3), image must be padded with 3 layers of zeros.
  - Etc.

In-lecture question

- Suppose I have an input image of 256 x 256 pixels and I want to design a CNN with 100 layers. How would the network perform if the padding was set to zero at each layer?
- In Slack #lectures
  - Identify the question and Reply in thread with a short response
- Answer :



Outline

- Padding
- Stride
- Multi-dimensional input
- Convolution as linear operation
- Depth
- Convolutional layer with PyTorch
- Pooling layer with PyTorch
- LeNet5 architecture
- VGG architecture
- Case studies

Strided convolution

- Strided convolution can be useful for large images (e.g. satellite images 10,000 x 10,000) :

Consider a stride = 2

$$\begin{bmatrix} 3 & 2 & 3 & 3 & 3 & 1 & 2 \\ 0 & 1 & 3 & 3 & 1 & 2 & 1 \\ 2 & 2 & 0 & 0 & 2 & 0 & 0 \\ 3 & 0 & 1 & 3 & 3 & 0 & 3 \\ 2 & 3 & 2 & 0 & 0 & 1 & 1 \\ 2 & 2 & 2 & 3 & 1 & 0 & 1 \\ 0 & 2 & 1 & 1 & 3 & 3 & 1 \end{bmatrix} \star \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 6 & 5 & 7 \\ 6 & 10 & 13 \\ 11 & 5 & 2 \end{bmatrix}$$

Strided convolution

Consider a stride = 2

$$\begin{bmatrix} 3 & 2 & 3 & 3 & 3 & 1 & 2 \\ 0 & 1 & 3 & 3 & 1 & 2 & 1 \\ 2 & 2 & 0 & 0 & 2 & 0 & 0 \\ 3 & 0 & 1 & 3 & 3 & 0 & 3 \\ 2 & 3 & 2 & 0 & 0 & 1 & 1 \\ 2 & 2 & 2 & 3 & 1 & 0 & 1 \\ 0 & 2 & 1 & 1 & 3 & 3 & 1 \end{bmatrix} \star \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 6 & 5 & 7 \\ 6 & 10 & 13 \\ 11 & 5 & 2 \end{bmatrix}$$

$$\begin{bmatrix} 3 & 2 & 3 & 3 & 3 & 1 & 2 \\ 0 & 1 & 3 & 3 & 1 & 2 & 1 \\ 2 & 2 & 0 & 0 & 2 & 0 & 0 \\ 3 & 0 & 1 & 3 & 3 & 0 & 3 \\ 2 & 3 & 2 & 0 & 0 & 1 & 1 \\ 2 & 2 & 2 & 3 & 1 & 0 & 1 \\ 0 & 2 & 1 & 1 & 3 & 3 & 1 \end{bmatrix} \star \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 6 & 5 & 7 \\ 6 & 10 & 13 \\ 11 & 5 & 2 \end{bmatrix}$$

Output size

- The output size of a convolution depends on :
  - Input size : n
  - Convolutional filter size : f
  - Padding : p
  - Stride : s

Integer floor value

$$n \times n \quad \Longrightarrow \quad \lfloor \frac{n+2p-f}{s} + 1 \rfloor \times \lfloor \frac{n+2p-f}{s} + 1 \rfloor$$

Input size

Convolution operation
Parameters: f, p, s

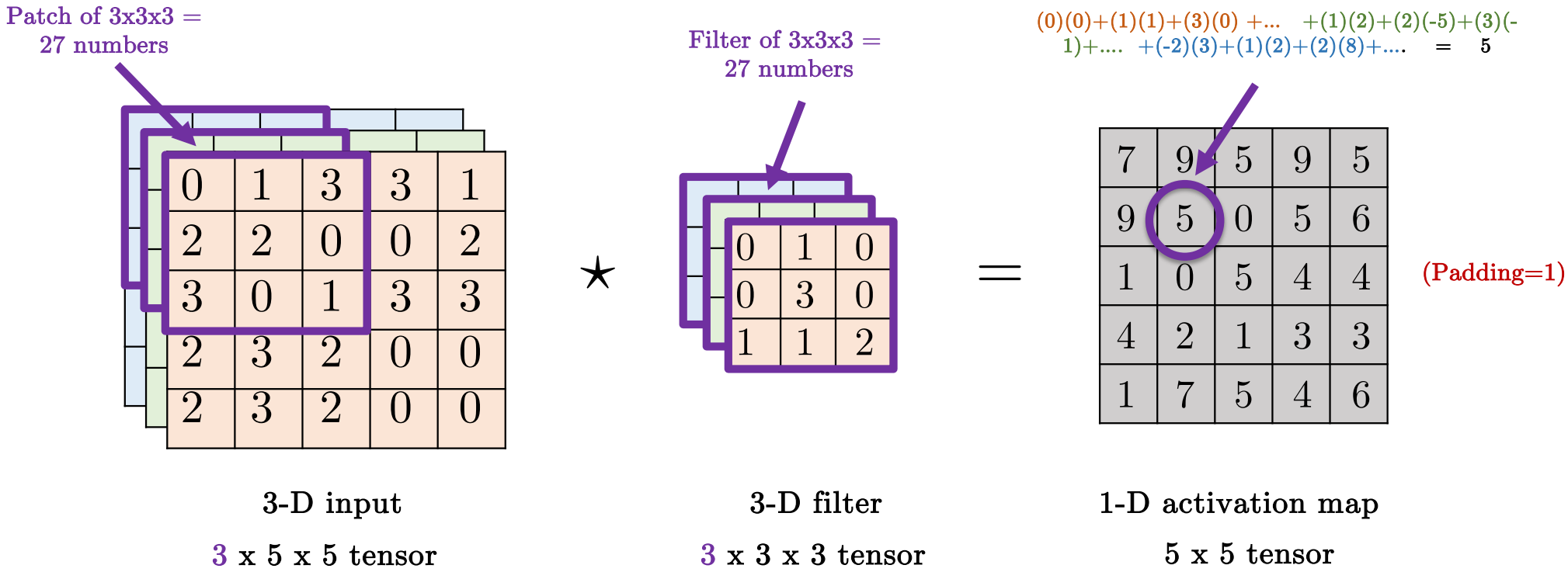
Output size

Outline

- Padding
- Stride
- **Multi-dimensional input**
- Convolution as linear operation
- Depth
- Convolutional layer with PyTorch
- Pooling layer with PyTorch
- LeNet5 architecture
- VGG architecture
- Case studies

Multi-dimensional convolution

- N-D convolution using N-D filters with N-D inputs returns 1-D activation map.
  - Example with 3-D input :



3-D input

3 x 5 x 5 tensor

3-D filter

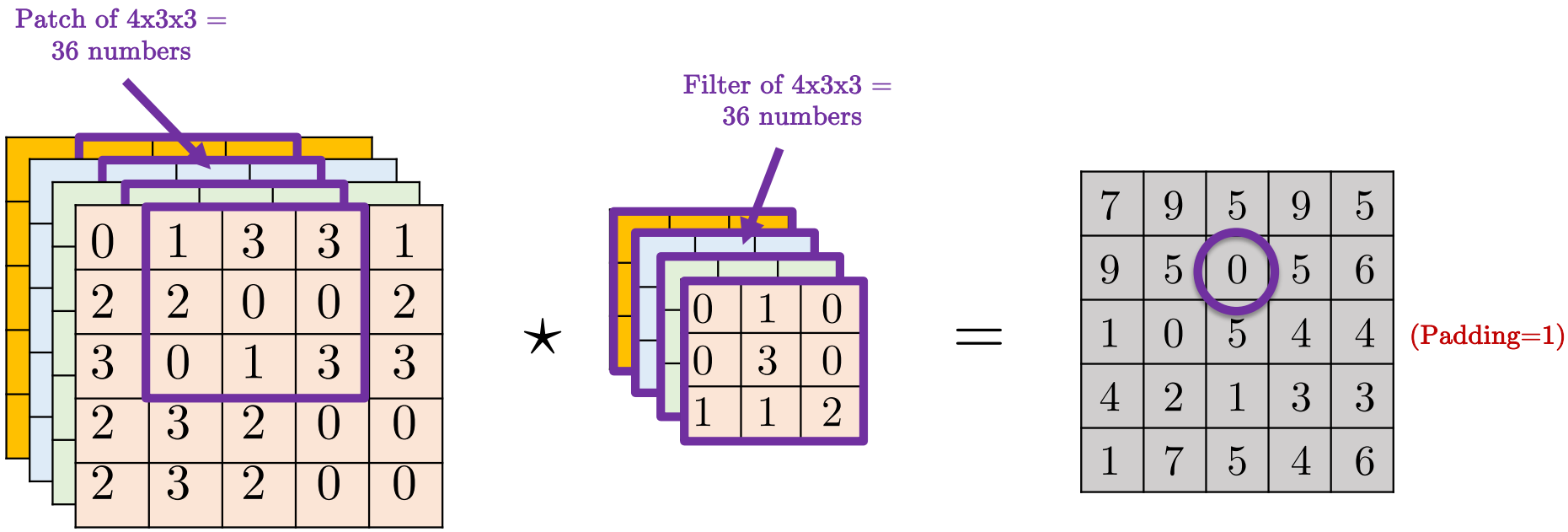
3 x 3 x 3 tensor

1-D activation map

5 x 5 tensor

Multi-dimensional convolution

- N-D convolution using N-D filters with N-D inputs returns 1-D activation map.
  - Example with 4-D input :



4-D input

4 x 5 x 5 tensor

4-D filter

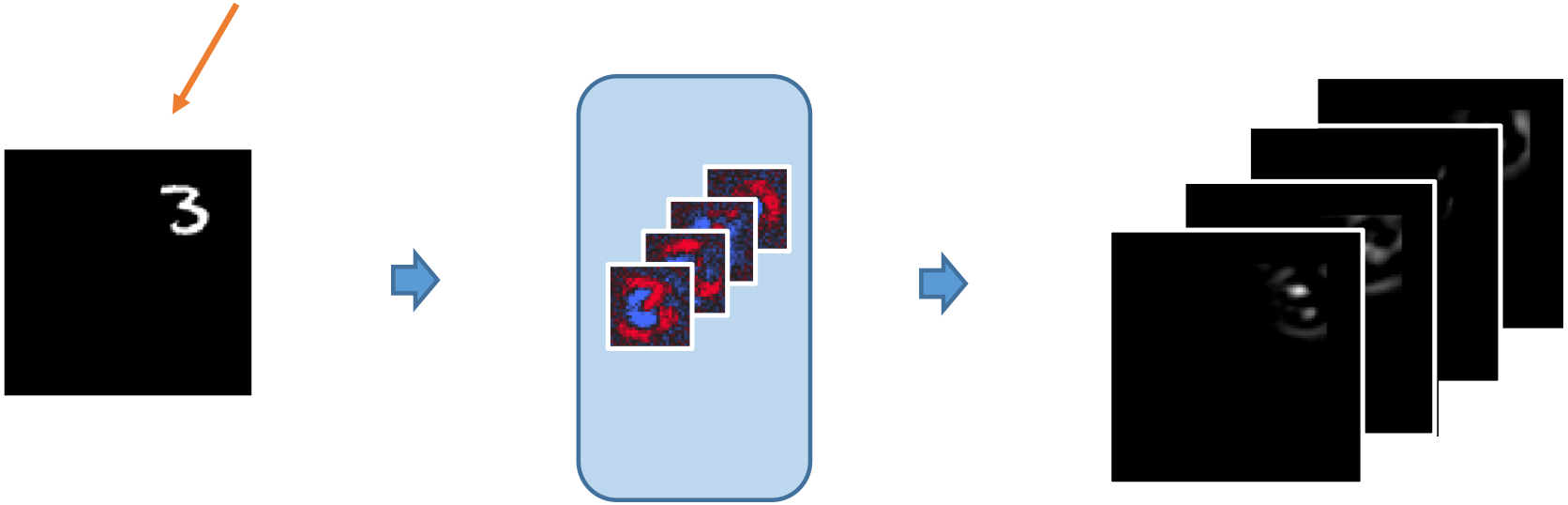
4 x 3 x 3 tensor

1-D activation map

5 x 5 tensor

One-dimensional input

- 1-D input : Gray-scale images



The input has 1 channel → Each filter has 1 channel

128 x 128 tensor

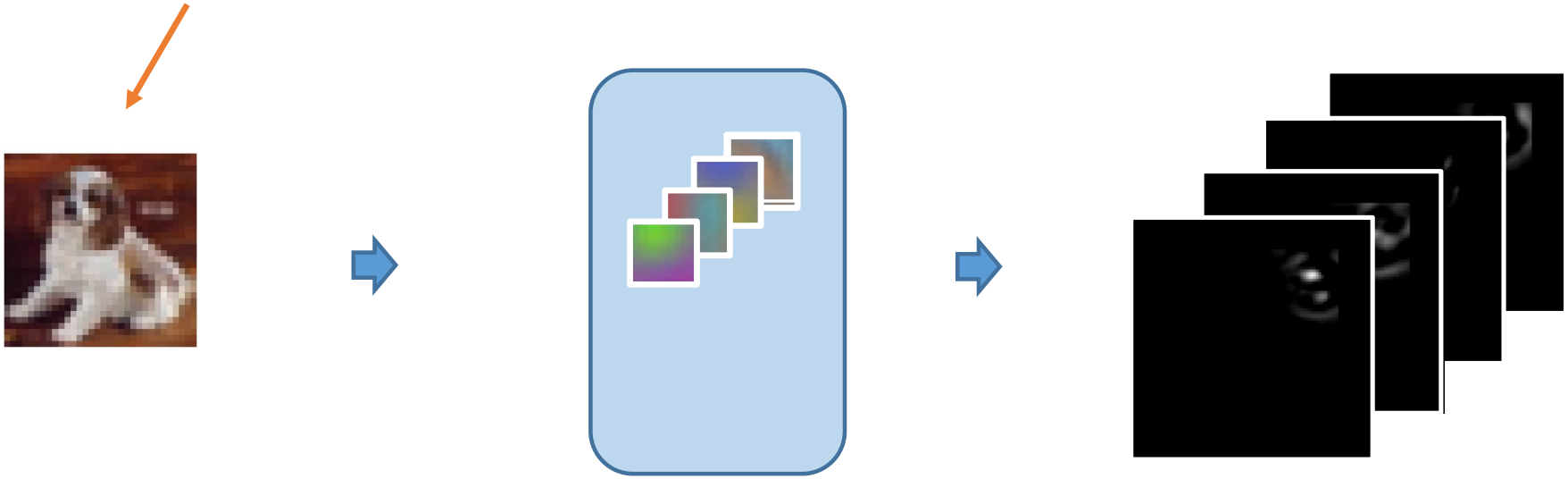
Four filters → Four activation maps

4 x 28 x 28 tensor

4 x 128 x 128 tensor

Three-dimensional input

- 3-D input : RGB images



The input has N=3 channels → Each filter has 3 channels

Four filters → Four activation maps

3 x 28 x 28 tensor

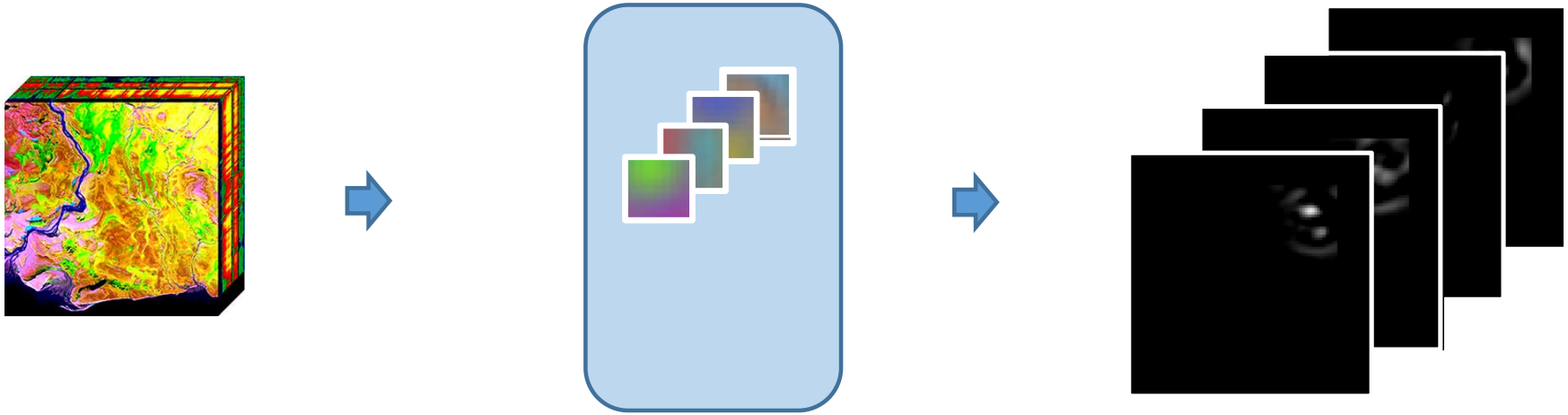
4 x 3 x 3 x 3 tensor

4 x 28 x 28 tensor

Why consider all channels together in a filter

F-D activations map with N-D input

- Generate F activation maps with N-D inputs :
  - N channels, F filters/activation maps, N x f x f filter size



The input has N channels → Each filter has N channels

F filters → F activation maps

N x n x n tensor

F x N x f x f tensor

F x n x n tensor

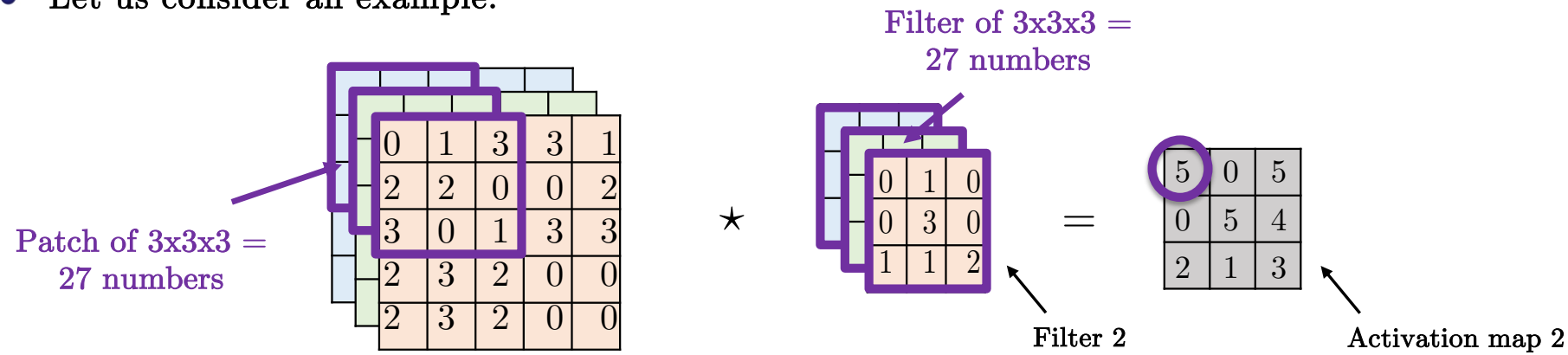
Outline

- Padding
- Stride
- Multi-dimensional input
- **Convolution as linear operation**
- Depth
- Convolutional layer with PyTorch
- Pooling layer with PyTorch
- LeNet5 architecture
- VGG architecture
- Case studies

Convolution as linear operation

- Convolutional layers are not very different from regular linear layers.
  - They also define a linear operator with a special structure.

- Let us consider an example:

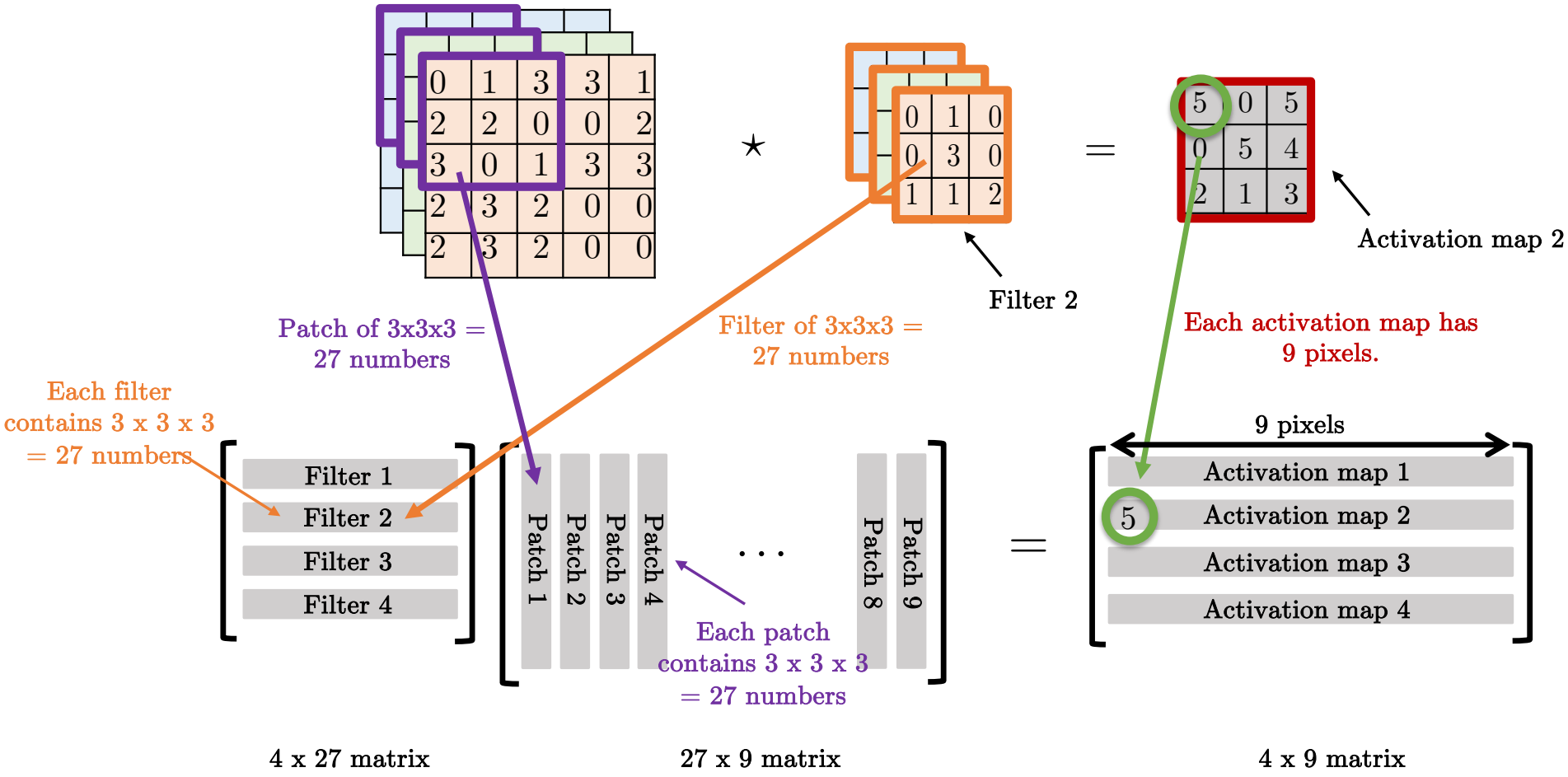


3 x 5 x 5 tensor	4 x 3 x 3 x 3 tensor	4 x 3 x 3 tensor
3 channels (each channel has 25 pixels)	4 filters	4 activation maps (9 pixels)

Key observation : Convolution operations are applied on patches !

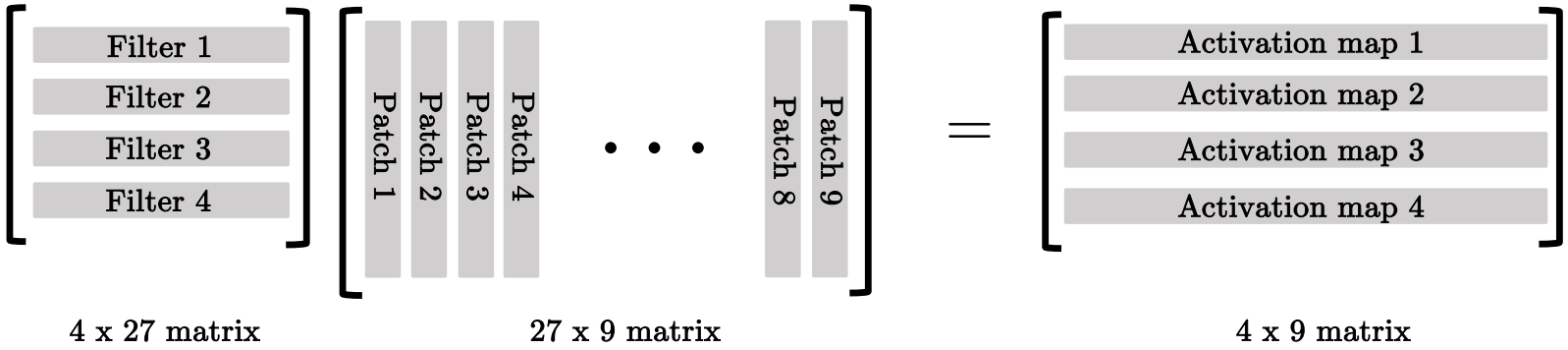
Convolution as linear operation

- Convolutions can be expressed as matrix-matrix multiplications :



Convolution as linear operation

- Convolutions can be expressed as matrix-matrix multiplications :



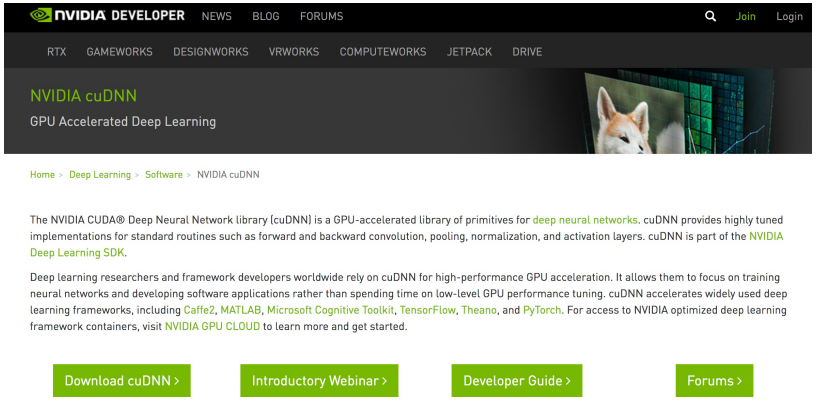
Convolution = matrix - matrix multiplication

Filters vectorized
(one filter per row)

Input pixels
rearranged as vectorized patches
(one patch per column)

Fast convolution

- Convolutions are simply matrix-matrix multiplications.
  - As matrix-matrix multiplications are standard linear algebra operations that have been highly parallelized.
    - CPU: MKL and BLAS libraries
    - GPU: CUDA library
- Fast convolutions on GPUs with **cuDNN** library.



https://developer.nvidia.com/cudnn

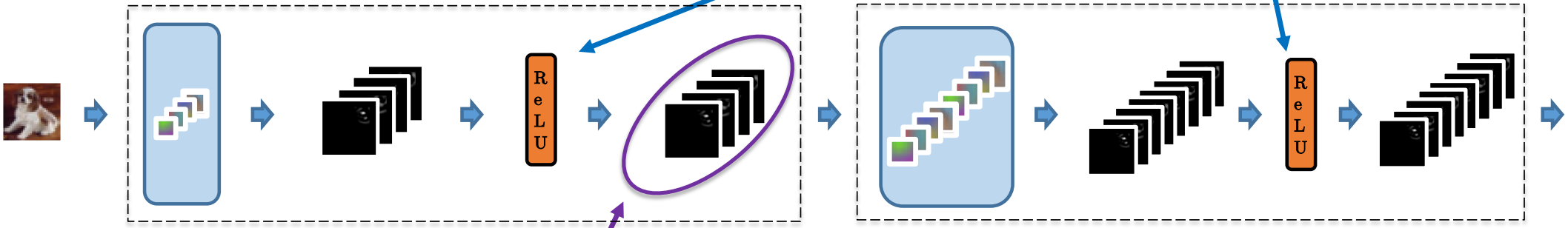
Outline

- Padding
- Stride
- Multi-dimensional input
- Convolution as linear operation
- **Depth**
- Convolutional layer with PyTorch
- Pooling layer with PyTorch
- LeNet5 architecture
- VGG architecture
- Case studies

Depth

- We can put some depth inside the ConvNet architecture :
  - We can stack convolutional layers.

Non-linearity is needed between each layer to prevent the network to collapse into a single convolutional layer.



3 channels Input features

Convolutional layer 1

4 activation maps

4 rectified maps Hidden features

Convolutional layer 2

8 rectified maps Hidden features

Each filter has N=3 channels (RGB) with f=5: 3 x 5 x 5 tensor

CL1 has 4 filters ⇒ #parameters = 4 x 3 x 5 x 5

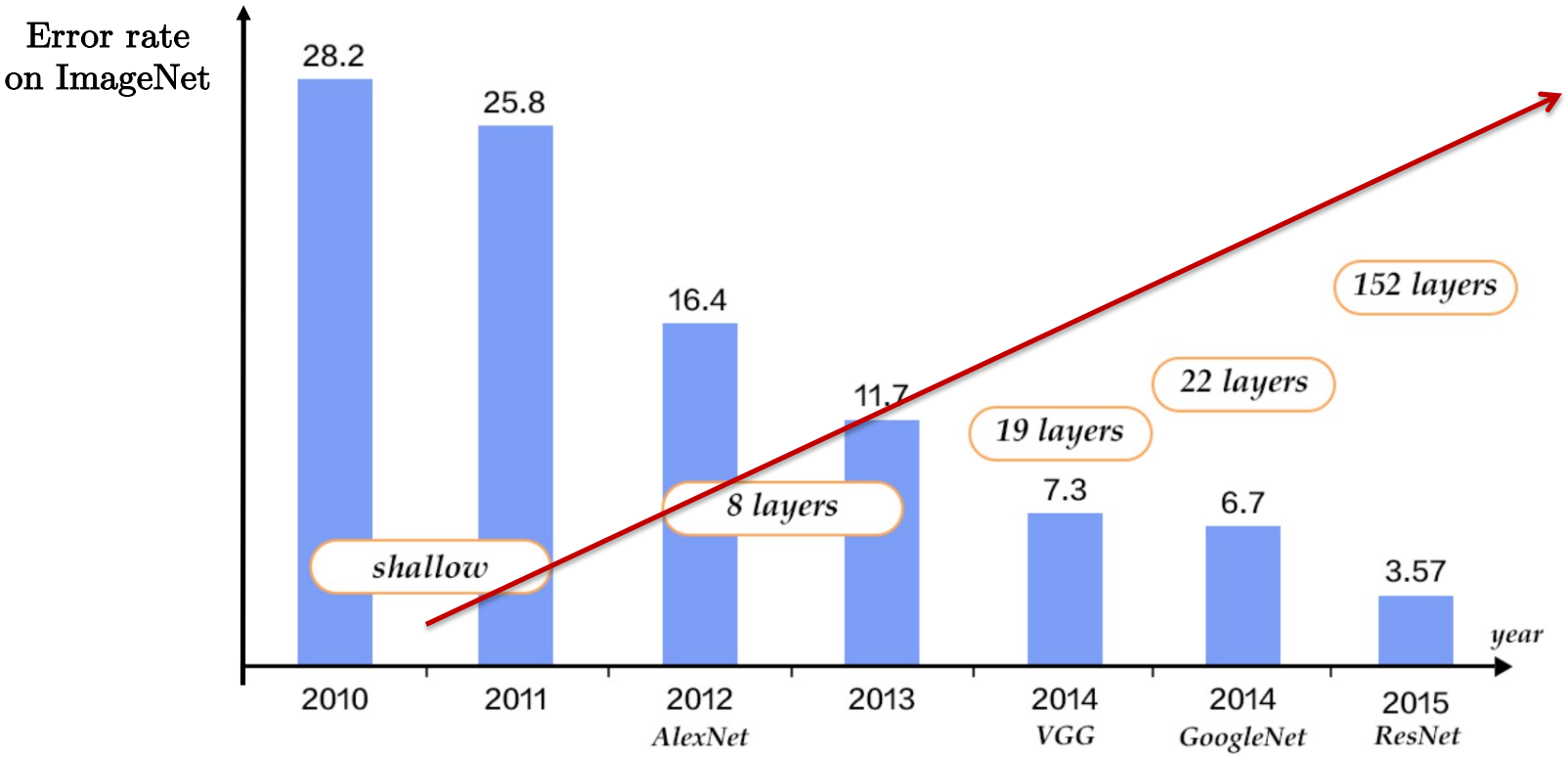
Just think of this as a RGB picture with 4 channels

Each filter has N=4 channels with f=3: 4 x 3 x 3 tensor

CL2 has 4 filters ⇒ #parameters = 8 x 4 x 3 x 3

The deeper the better

- Revolution of depth :



2018
1000s layers

Outline

- Padding
- Stride
- Multi-dimensional input
- Convolution as linear operation
- Depth
- **Convolutional layer with PyTorch**
- Pooling layer with PyTorch
- LeNet5 architecture
- VGG architecture
- Case studies

Convolutional layer in PyTorch

- PyTorch function:

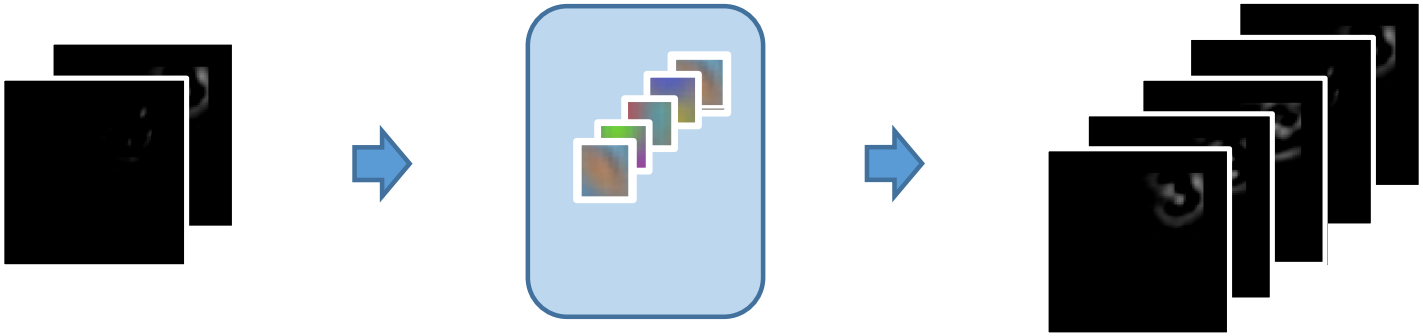
```
mod = nn.Conv2d( 2 , 5 ,  kernel_size=3,  padding=1 )
```

2 input channels

5 filters

Each filter has a size 3x3 filters

5 output channels



Input is a
2 x 6 x 6 tensor

Conv filters are a
5 x 2 x 3 x 3 tensor

Output is a
5 x 6 x 6
tensor

Lab 01

- Convolutional layer

jupyter conv_layer_demo Last Checkpoint: a few seconds ago (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

Lab 01 : Convolutional layer - demo

In [2]:
```
import torch
import torch.nn as nn
```

Make a convolutional module

- inputs: 2 channels
- output: 5 activation maps
- filters are 3x3
- padding with one layer of zero to not shrink anything

In [3]:
```
mod = nn.Conv2d( 2 , 5 ,  kernel_size=3,  padding=1 )
```

Make an input 2 x 6 x 6 (two channels, each one has 6 x 6 pixels)

In [4]:
```
bs=1

x=torch.rand(bs,2,6,6)

print(x)

print(x.size())
```

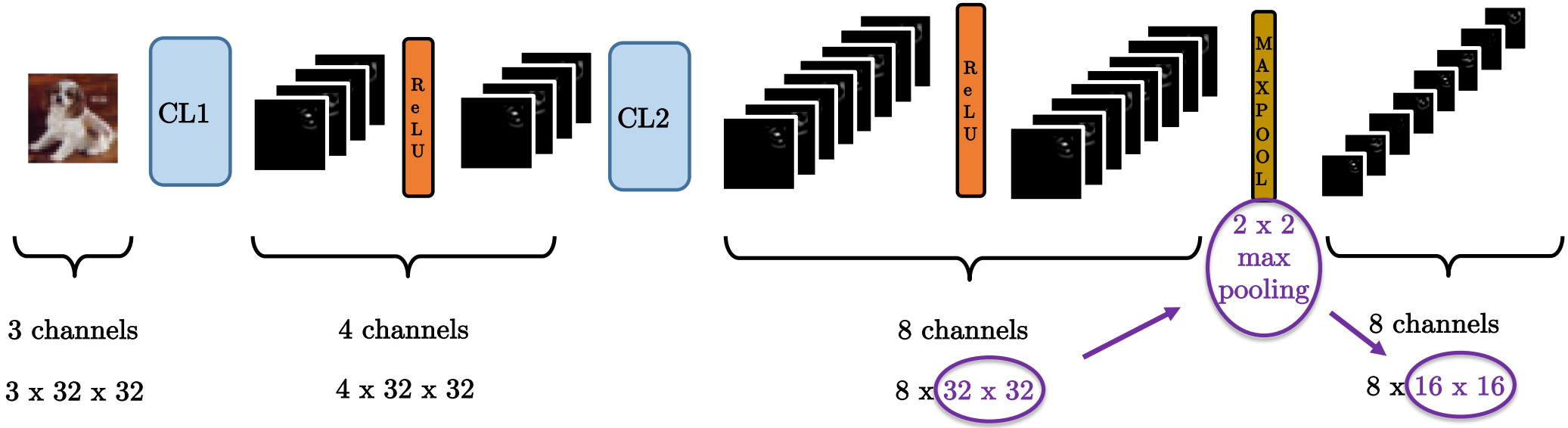
```
tensor([[[[0.9647, 0.6641, 0.1846, 0.0741, 0.2426, 0.0286],
          [0.2926, 0.2690, 0.8945, 0.7917, 0.6449, 0.4339],
          [0.4199, 0.9387, 0.3030, 0.7605, 0.5784, 0.1366],
          [0.0920, 0.2539, 0.1089, 0.0882, 0.4449, 0.9903],
          [0.0181, 0.6897, 0.2516, 0.8399, 0.7515, 0.3681],
          [0.6694, 0.0813, 0.5470, 0.1661, 0.7311, 0.2856]],
```

Outline

- Padding
- Stride
- Multi-dimensional input
- Convolution as linear operation
- Depth
- Convolutional layer with PyTorch
- **Pooling layer with PyTorch**
- LeNet5 architecture
- VGG architecture
- Case studies

How does gradient pass through max pooling ?

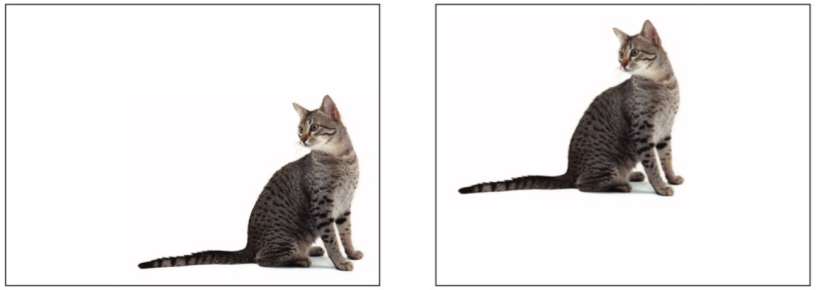
Max pooling



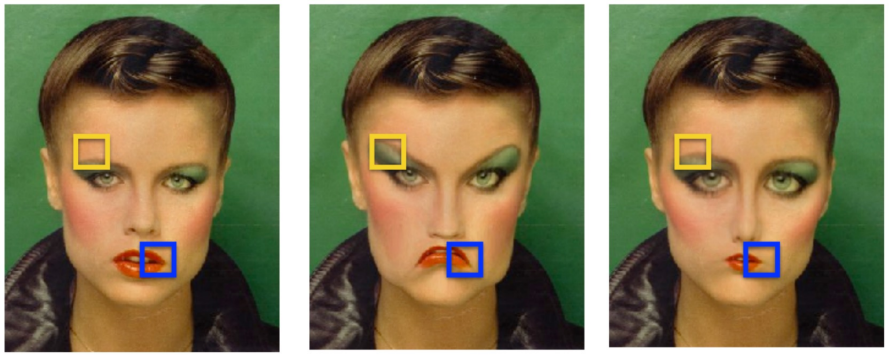
- MP (max pooling) layers are useful for :
  - Trading image resolution with features number (features capture data structures).
  - Controlling the computational complexity (image size /2 and #features x2 per dimension).

Max pooling

- MP (max pooling) layers are also useful for :
  - Creating global invariance w.r.t. translation (same semantic object independent of location).



  - Providing local invariance w.r.t. object deformations (robust to slight object changes).



Pooling layer in PyTorch

- PyTorch function:
  - Max Pooling with window size = 2 x 2

```
mod = nn.MaxPool2d(2,2)
```

3	3	0	1	1	1
7	2	1	1	5	2
7	9	1	0	5	6
5	6	1	0	3	2
9	7	0	0	0	0
4	7	0	0	0	1

6 x 6 activation map

Down-sampling
+ max pooling

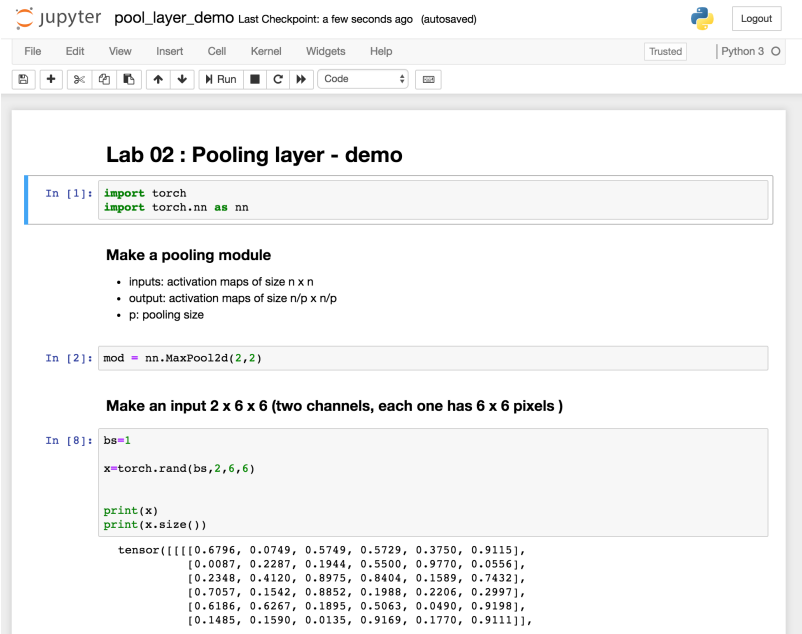
$\Longrightarrow$

7	1	5
9	1	6
9	0	1

3 x 3 activation map

Lab 02

- Pooling layer :

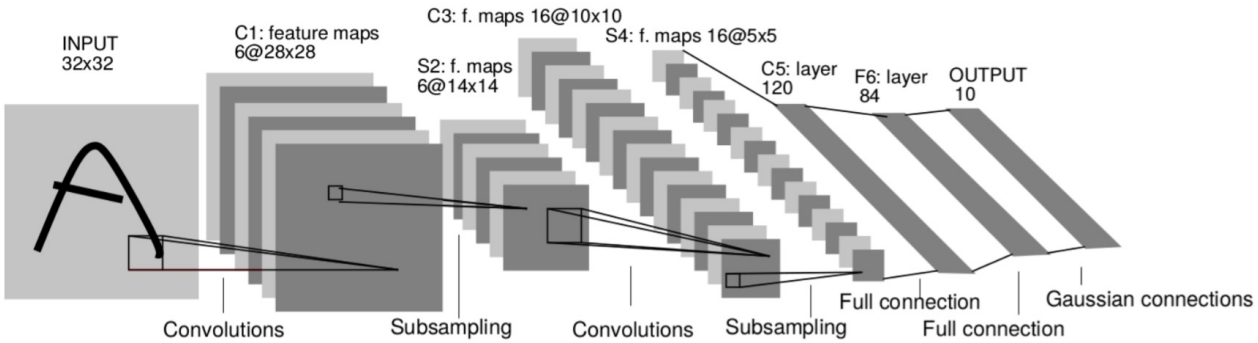


Outline

- Padding
- Stride
- Multi-dimensional input
- Convolution as linear operation
- Depth
- Convolutional layer with PyTorch
- Pooling layer with PyTorch
- LeNet5 architecture
- VGG architecture
- Case studies

LeNet5 architecture

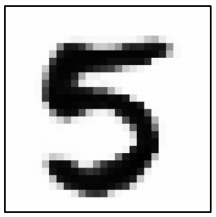
- LeCun, Bottou, Bengio, Haffner, 1998:
  - 1st ConvNet architecture



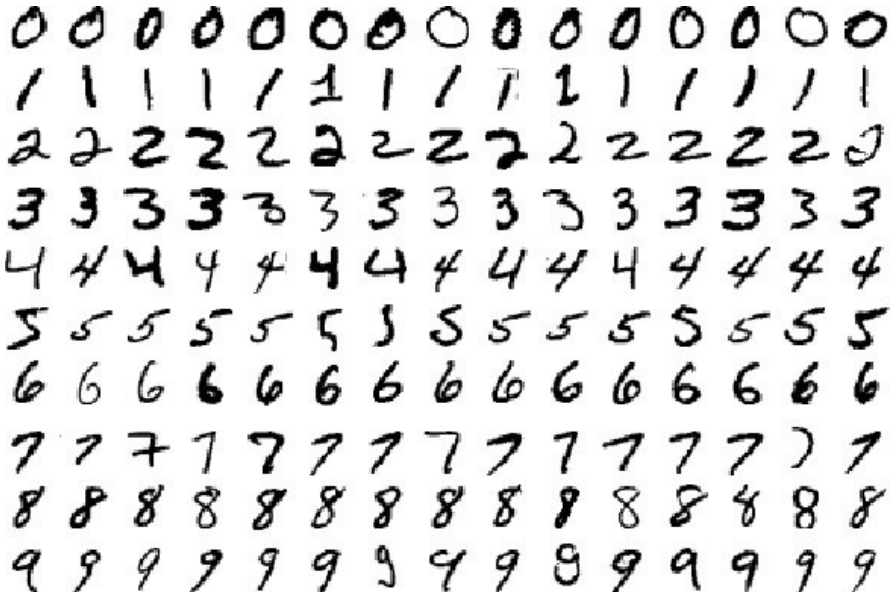
- The dataset:
  - MNIST
- The network:
  - Convolutional/MP layers
  - Linear layers

MNIST dataset

- Number of classes: 10
- Training set: 50,000 labelled images
- Test set: 10,000 labelled images

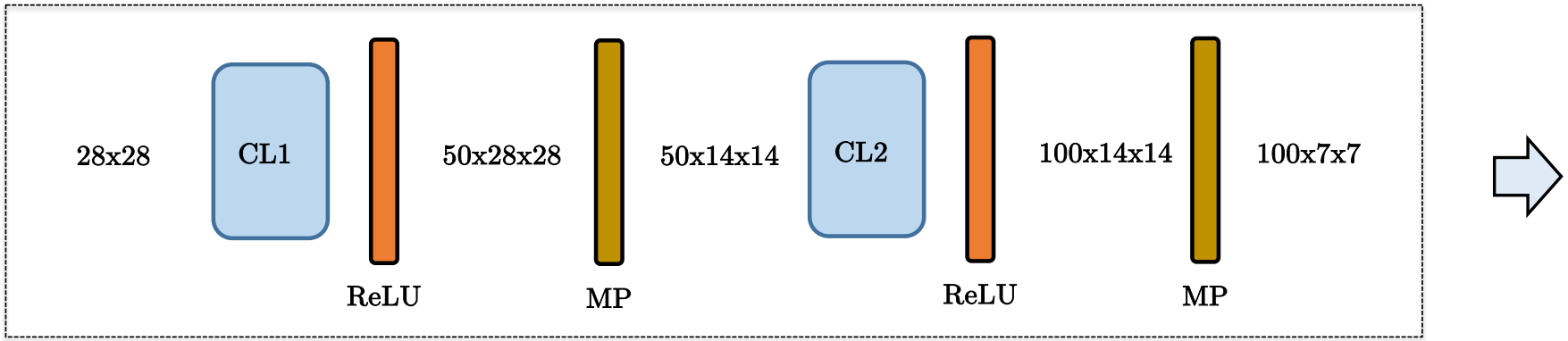


28 x 28 grayscale image

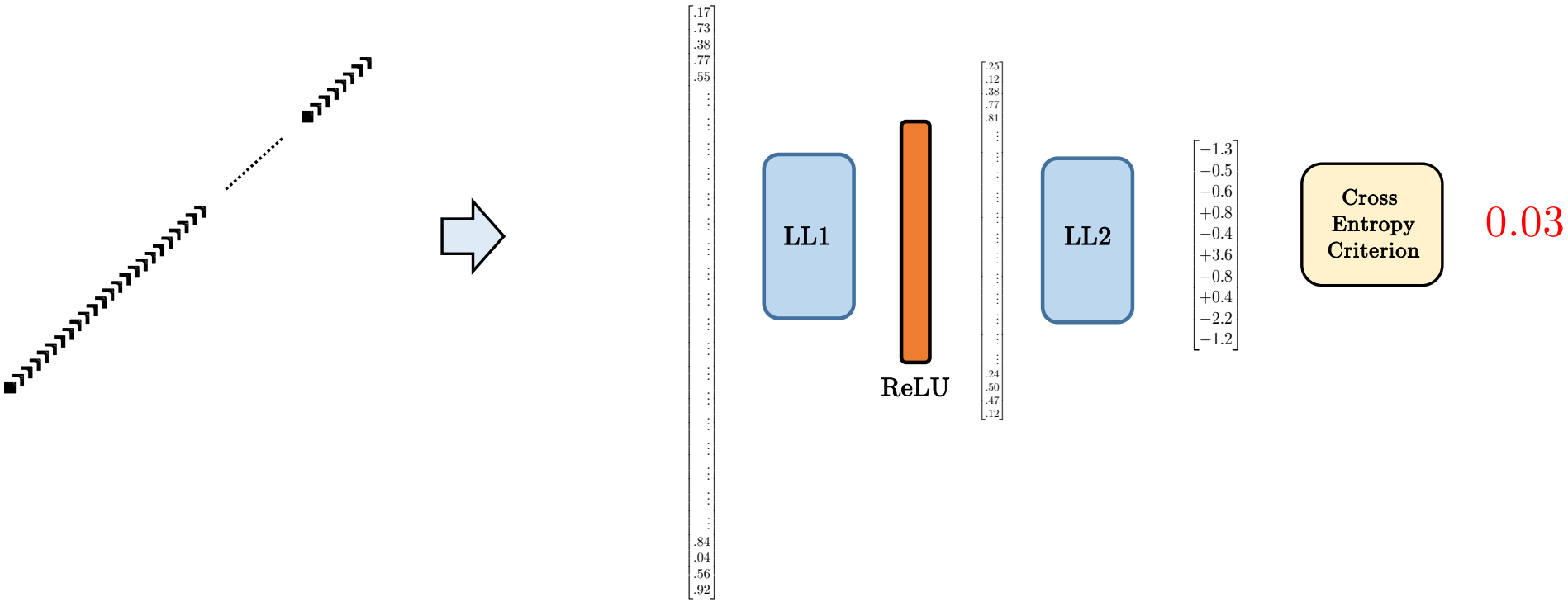


Two convolutional layers

- Convolutional/MP layers:



Linear layers



CL features

100 (small) activation maps

(7 x 7 pixel each)

Vector with 4,900 entries

(100 x 7 x 7 = 4,900)

Vectors with 100 entries

10 Scores

Loss

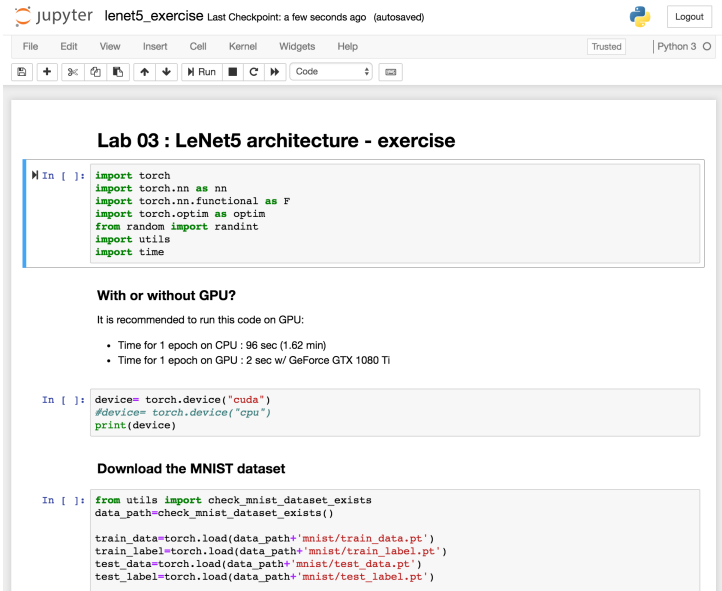
Number of architecture parameters

- First CL:
  - 28x28 ⇒ CL1+ReLU ⇒ 50x28x28
  - 50x28x28 ⇒ MP ⇒ 50x14x14
  - #Parameters = 50x3x3 + 50 = 500
- Second CL:
  - 50x14x14 ⇒ CL1+ReLU ⇒ 100x14x14
  - 100x14x14 ⇒ MP ⇒ 100x7x7
  - #Parameters = 100x50x3x3+100 = 45,100
- First LL:
  - 100x7x7 ⇒ Reshape ⇒ 4,900
  - 4,900 ⇒ LL1 ⇒ 100
  - #Parameters = 4900x100+100 = 490,100
- Second LL:
  - 100 ⇒ LL2 ⇒ 10
  - #Parameters = 100x10+10 =1,010

Total for 4 Layers ⇒ 536,710 / 0.54 Millions of parameters

Lab 03

- LeNet5 on MNIST :
  - Learning rate schedule:
    - Start with lr=0.25
    - Divide by 1.5 every 5 epochs



Results

- Results with a 4-layer ConvNet:
  - Number of parameters: 0.54 million
  - Error on training set: 0%
  - Error on test set: 0.78%

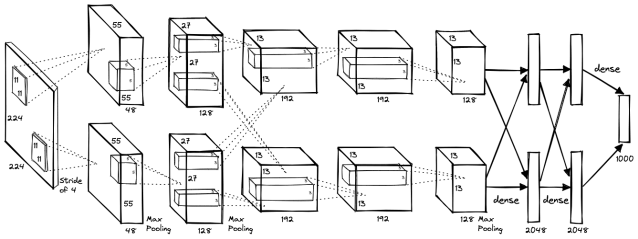
- Timing for one epoch:
  - CPU : 1.6 min
  - Tesla K80 GPU (Google Cloud) : 6 sec
  - GTX 1080 GPU ($600) : 2 sec

Outline

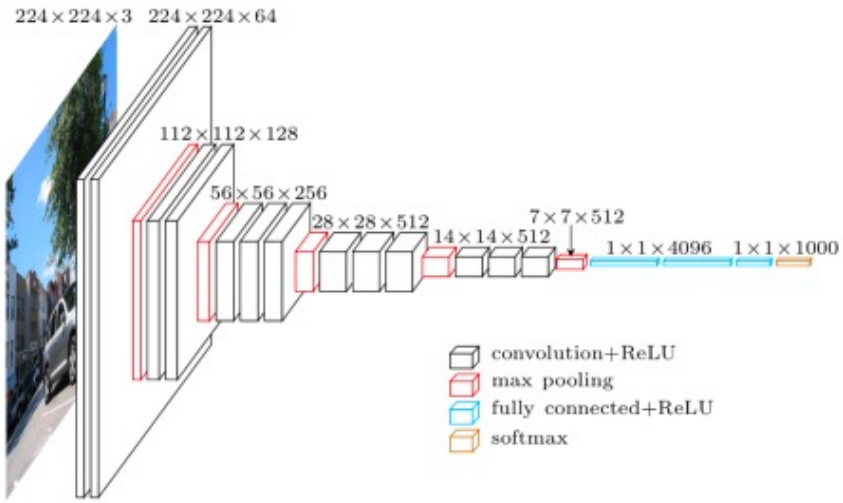
- Padding
- Stride
- Multi-dimensional input
- Convolution as linear operation
- Depth
- Convolutional layer with PyTorch
- Pooling layer with PyTorch
- LeNet5 architecture
- **VGG architecture**
- Case studies

VGG architecture

- Simonyan and Zisserman, 2014 :
  - 2nd place at the ImageNet challenge in 2014
  - All convolutional features are 3x3



AlexNet

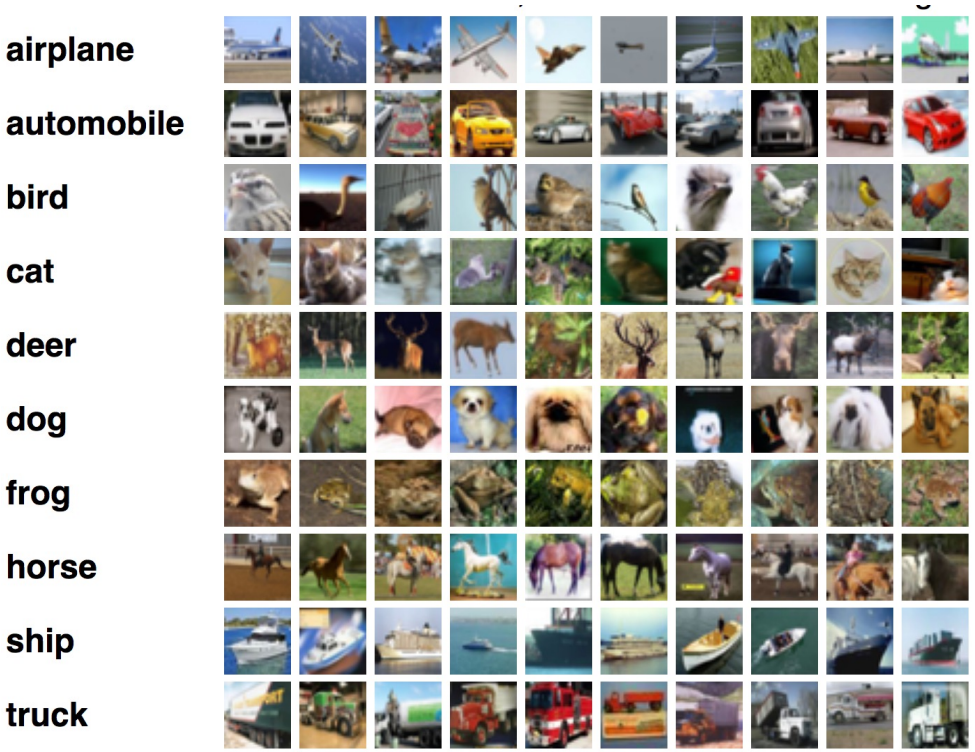


- The dataset:
  - CIFAR
- The network:
  - Convolutional/MP layers
  - Linear layers

CIFAR dataset

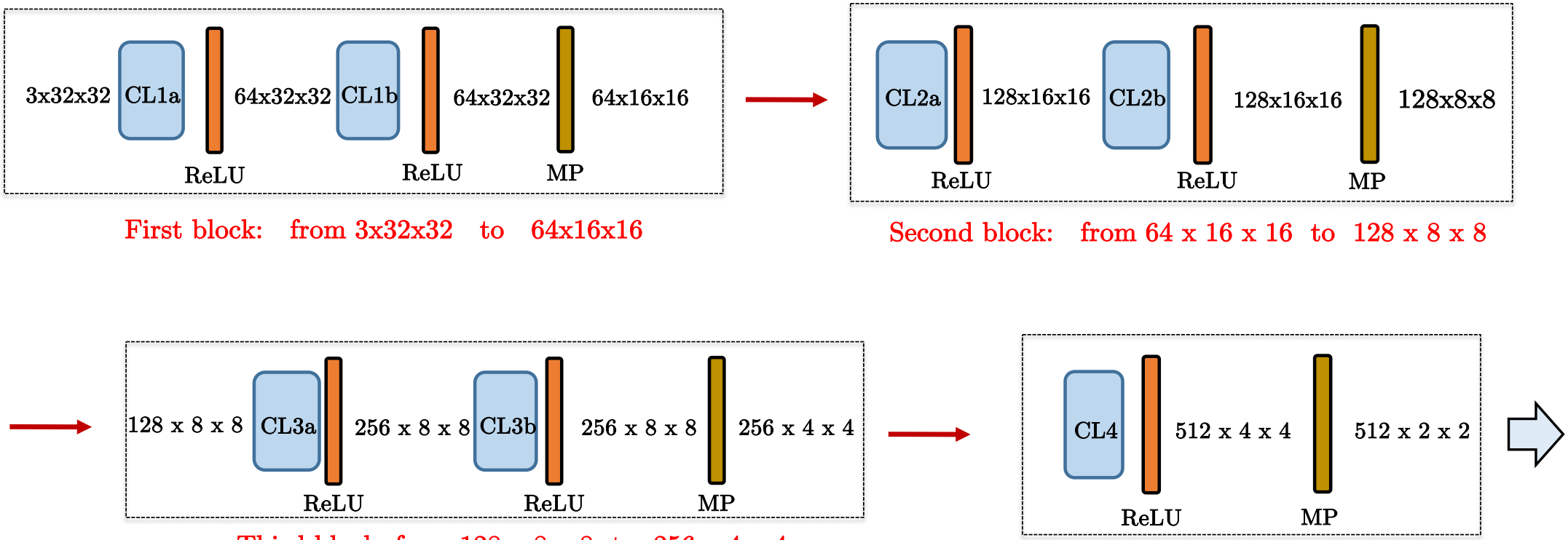
- Number of classes: 10
- Training set: 50,000 labelled images
- Test set: 10,000 labelled images

32 x 32 RGB image



Four convolutional layers

- Convolutional/MP layers:
  - The number of features/channels increases by a factor 2.
  - The resolution of the activation maps decreases by a factor 2.



First block: from 3x32x32 to 64x16x16

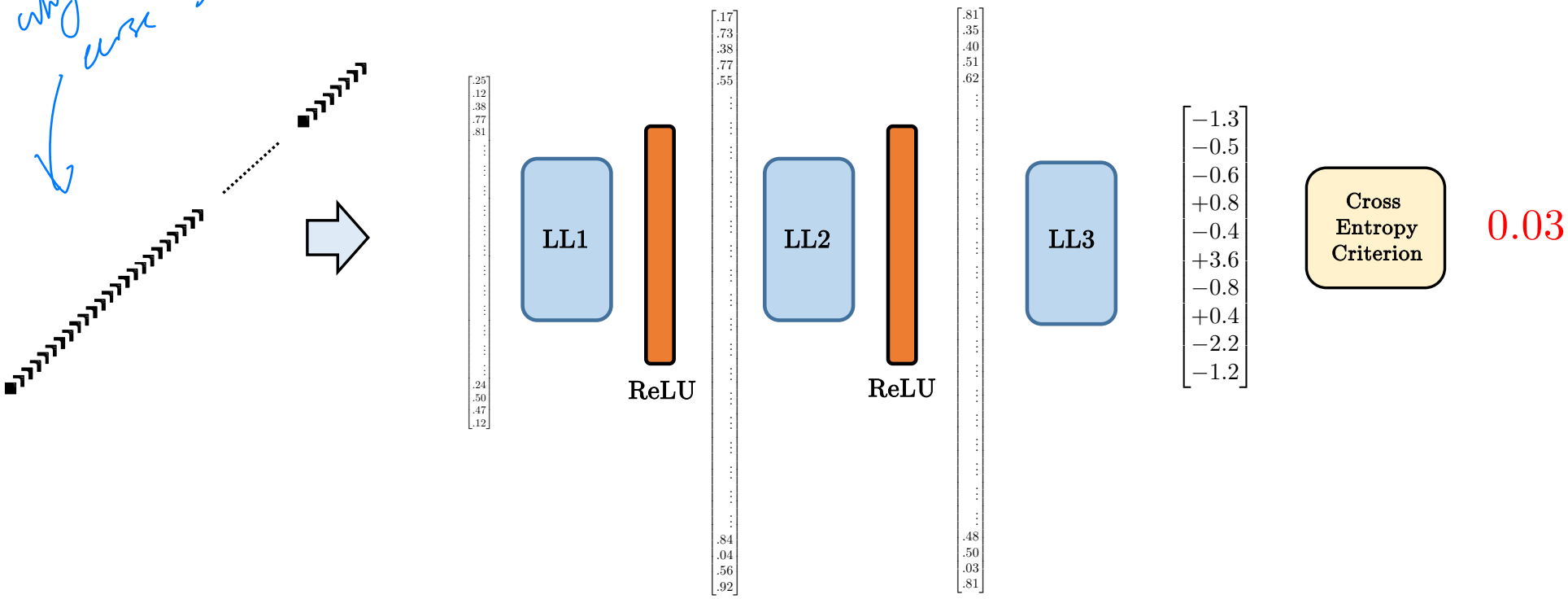
Second block: from 64 x 16 x 16 to 128 x 8 x 8

Third block: from 128 x 8 x 8 to 256 x 4 x 4

Fourth block: from 256 x 4 x 4 to 512 x 2 x 2

why don't NNs suffer from curse of dimensionality ?

Linear layers



CL features

512 (tiny) activation maps

(2 x 2 pixel each)

Vector with 2,048 entries

(512 x 2 x 2 = 2,048)

Vector with 4,096 entries

Vectors with 4,096 entries

10 Scores

Loss

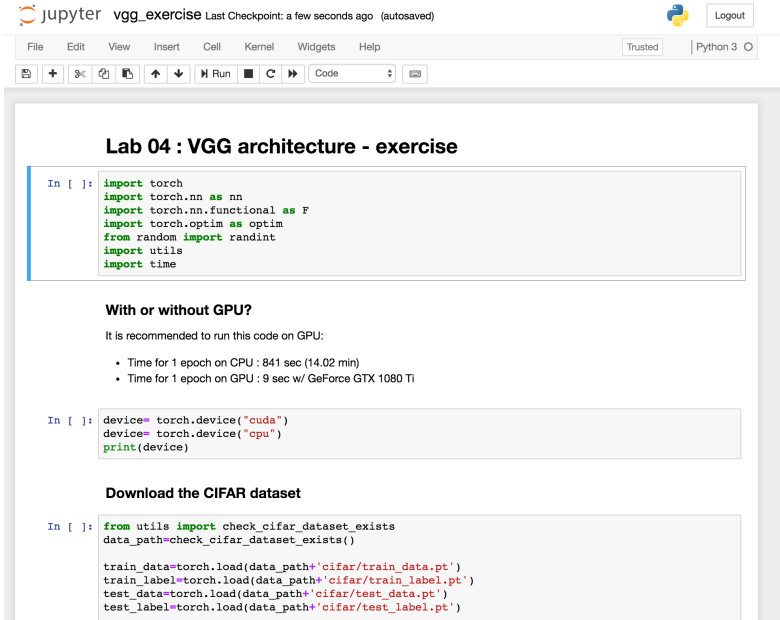
Number of architecture parameters

- First Block:
  - 3x32x32 ⇒ CL1a+ReLU ⇒ 64x32x32
  - 64x32x32 ⇒ CL1b+ReLU ⇒ 64x32x32
  - 64x32x32 ⇒ MP ⇒ 64x16x16
  - #Parameters = 64x3x3x3+64+64x64x3x3+64 = 38K
- Second Block:
  - 64x16x16 ⇒ CL2a+ReLU ⇒ 128x16x16
  - 128x16x16 ⇒ CL2b+ReLU ⇒ 128x16x16
  - 128x16x16 ⇒ MP ⇒ 128x8x8
  - #Parameters = 128x64x3x3+128+128x128x3x3+128 = 221K
- Third Block:
  - 128x8x8 ⇒ CL3a+ReLU ⇒ 256x8x8
  - 256x8x8 ⇒ CL3b+ReLU ⇒ 256x8x8
  - 256x8x8 ⇒ MP ⇒ 256x4x4
  - #Parameters = 256x128x3x3+256+256x256x3x3+256 = 885K
- Fourth Block:
  - 256x4x4 ⇒ CL4+ReLU ⇒ 512x4x4
  - 512x4x4 ⇒ MP ⇒ 512x2x2
  - #Parameters = 512x216x3x3+512+512x512x3x3+512 = 3.3M
- LL:
  - 512x2x2 ⇒ Reshape ⇒ 2,048
  - 2,048 ⇒ LL1+ReLU ⇒ 4,096
  - 4,096 ⇒ LL2+ReLU ⇒ 4,096
  - 4,096 ⇒ LL3 ⇒ 10
  - #Parameters = 2048x4096+4096+4096x4096+4096+4096x10+10 = 25.2M

Total for 10 Layers ⇒ 30 Millions of parameters

Lab 04

- Learning rate schedule:
  - Start with lr=0.25
  - Divide by 2 at epoch 10
  - Divide by 2 again at epoch 14
  - Divide by 2 again at epoch 18



Results

- Results with a 1-layer MLP :
  - Number of parameters: 0.03 million
  - Error on training set: 59%
  - Error on test set: 65%

- Results with a 3-layer MLP :
  - Number of parameters: 2 millions
  - Error on training set: 0%
  - Error on test set: 46%

- Results with a 10-layer ConvNet :
  - Number of parameters: 30 millions
  - Error on training set: 0%
  - Error on test set: 20%

Timing for one epoch:

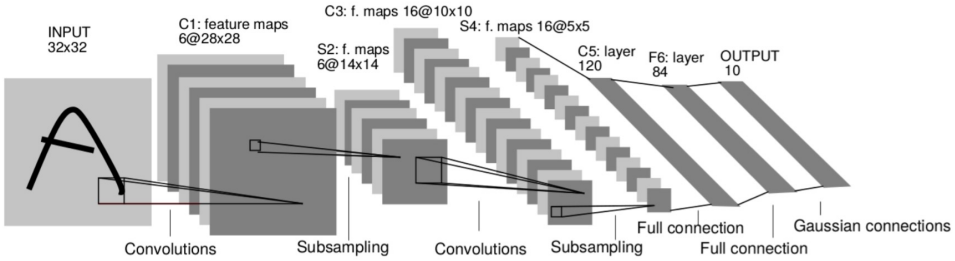
- CPU : 14 min
- Tesla K80 GPU (Google Cloud): 1.2 min
- GTX 1080 GPU ($600) : 9 sec

Outline

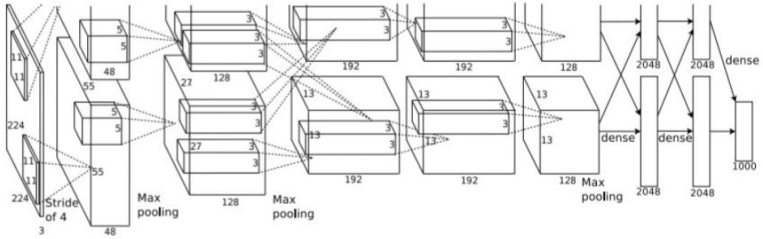
- Padding
- Stride
- Multi-dimensional input
- Convolution as linear operation
- Depth
- Convolutional layer with PyTorch
- Pooling layer with PyTorch
- LeNet5 architecture
- VGG architecture
- Case studies

ConvNet architectures

- LeNet5 [LeCun, Bottou, Bengio, Haffner, 1998]:
  - Input is 28x28
  - Architecture is CL-MP-CL-MP-LL-LL
  - Accuracy on MNIST is 99.6%
  - #Parameters=0.6M



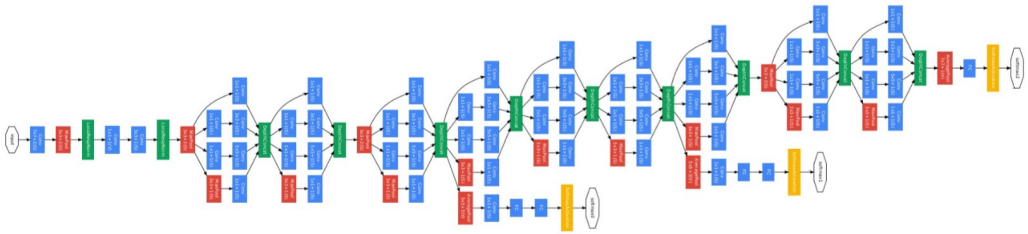
- AlexNet [Krizhevsky, Hinton, Sutskever, 2012]:
  - Input is 227x227x3
  - Architecture is 7CL-3MP-2FC
  - Predication error on ImageNet is 15.4%
  - 1st rank
  - #Parameters=62M



ConvNet architectures

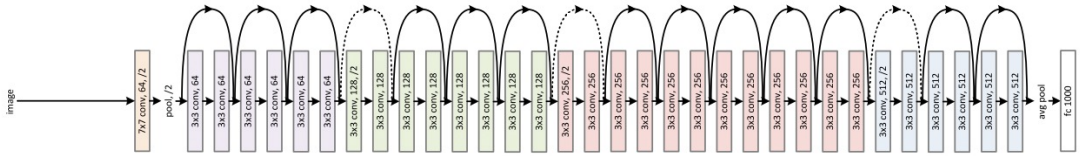
- VGGNet [Simonyan, Zisserman, 2014]:
  - Input is 227x227x3
  - Architecture has 16 layers
  - Predication error on ImageNet is 7.3%
  - 2nd rank
  - #Parameters=138M

- GoogleNet/Inception [Szegedy-et.al, 2014]:
  - Input is 227x227x3
  - Architecture has 22 layers
  - Predication error on ImageNet is 6.7%
  - 1st rank
  - #Parameters=5M



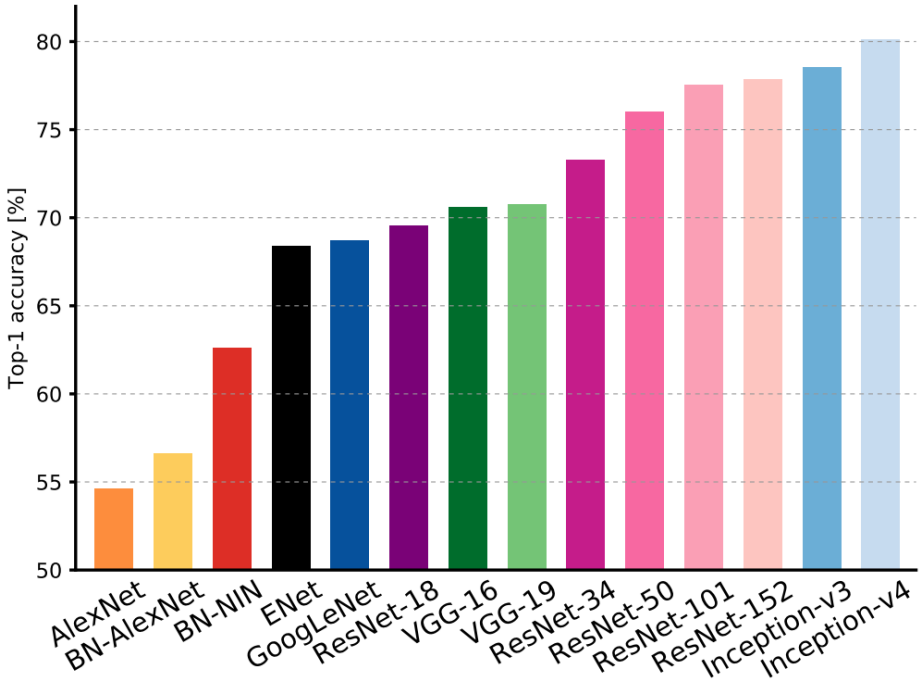
ConvNet architectures

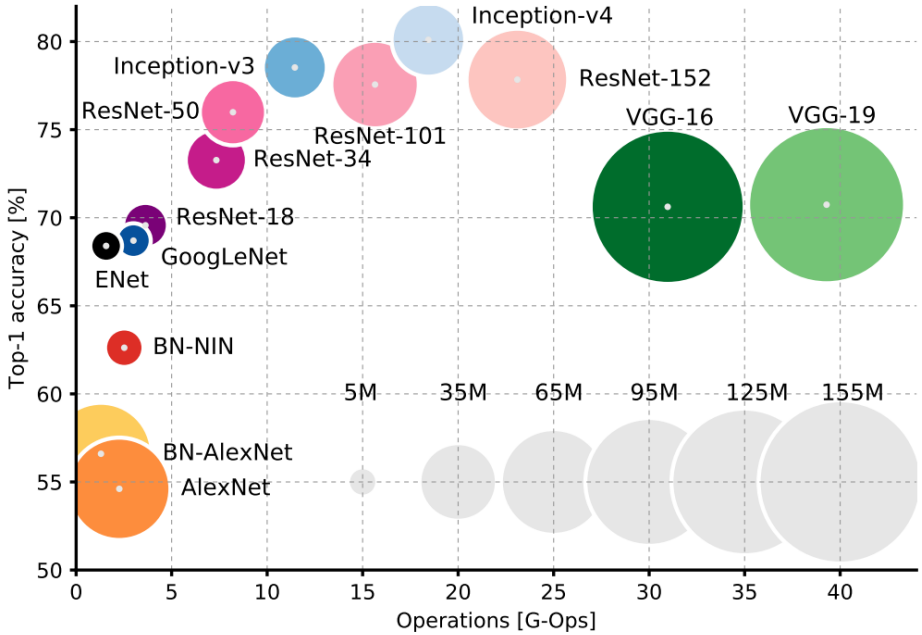
- Microsoft/ResNet [He-et.al'15]:
  - Input is 227x227x3
  - Architecture has 152 layers
  - Predication error on ImageNet is 3.6%
  - 1st rank
  - #Parameters=60M



- Etc.

ConvNet architectures





Questions?

Convolution

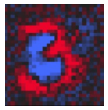
- Exact formula for the convolution layer:
  - Each filter generates an activation map.

$$(0)(-1) + (3)(1) + (2)(2) + (0)(1) = 7$$

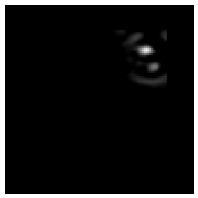
$$\begin{bmatrix} 0 & 3 & 0 & 1 \\ 2 & 0 & 1 & 2 \\ 3 & 0 & 0 & 2 \\ 0 & 1 & 2 & 0 \end{bmatrix} \star \begin{bmatrix} -1 & 1 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 7 & -2 & 5 \\ 4 & 1 & 3 \\ -2 & 4 & 6 \end{bmatrix}$$

Input image

One filter

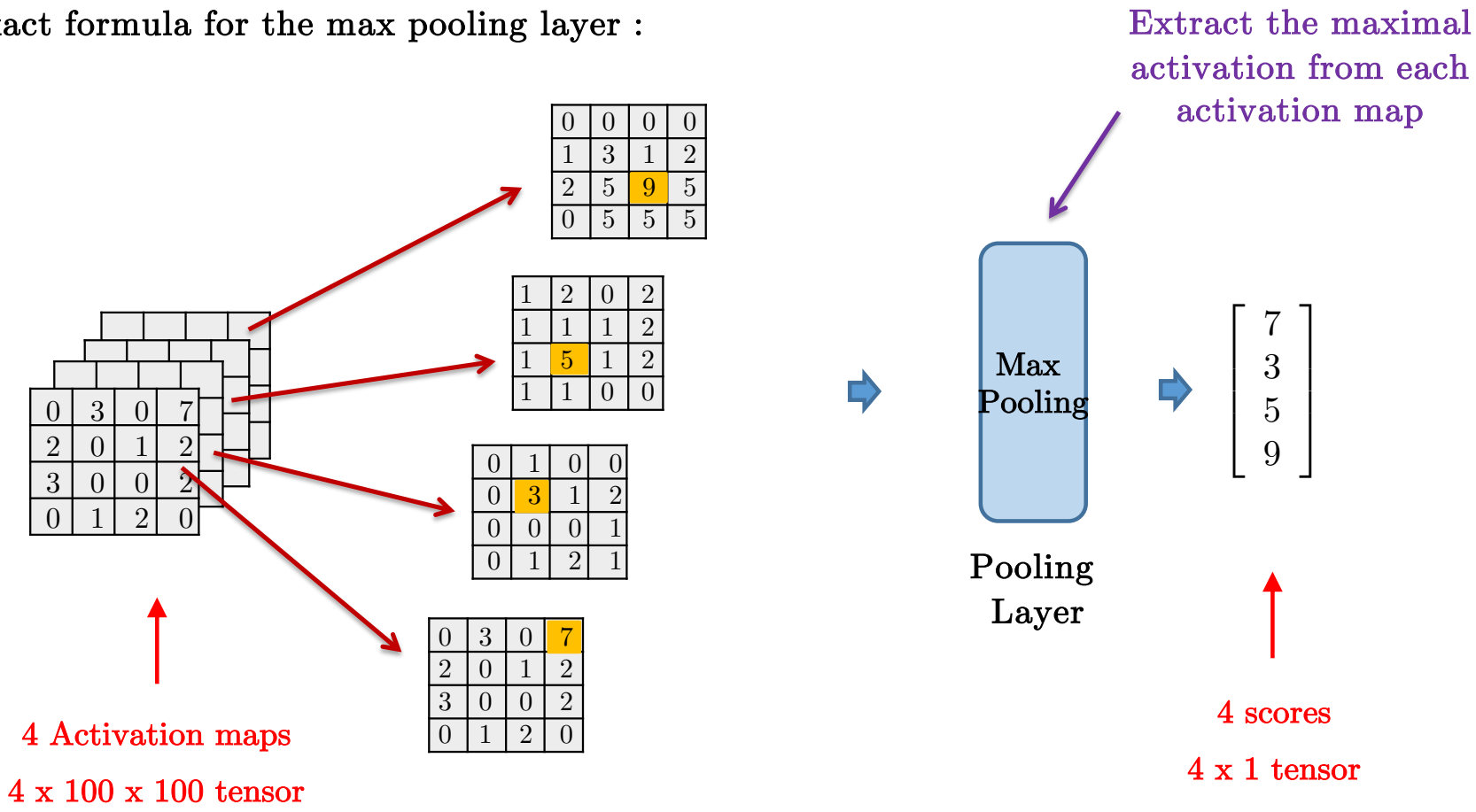


One activation map



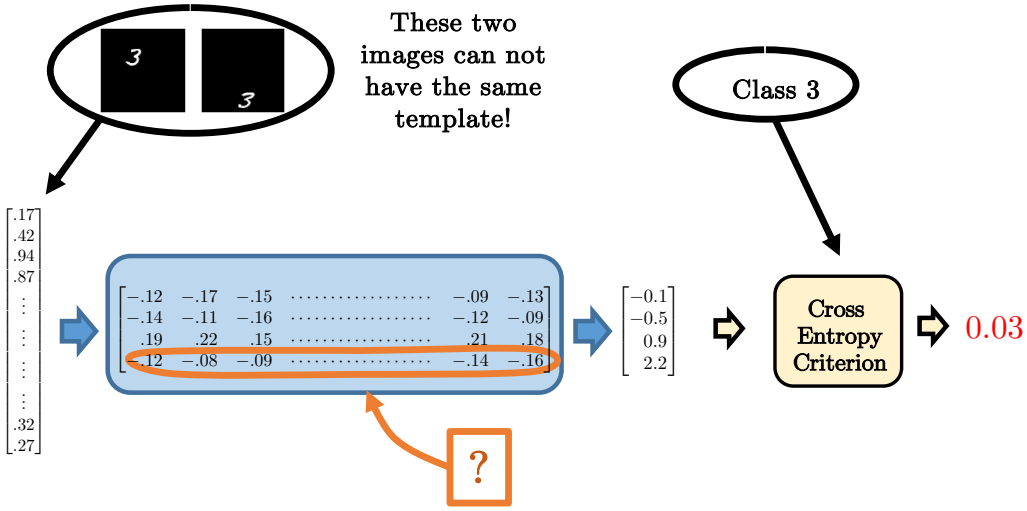
One-layer ConvNet

- Exact formula for the max pooling layer :

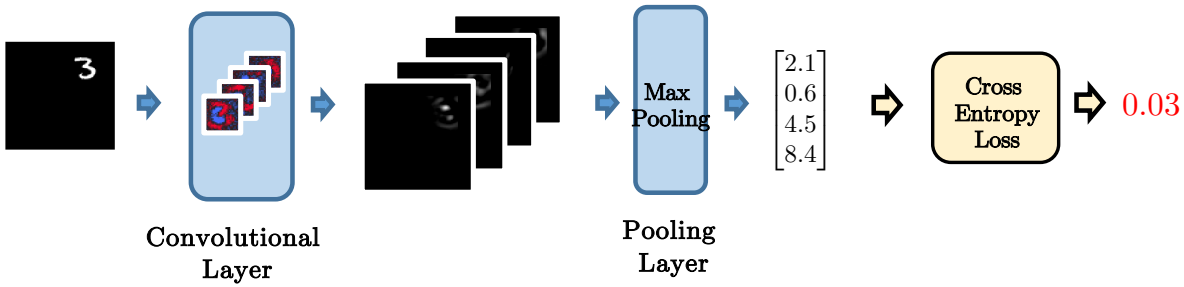


MLP vs ConvNets

- A regular one-layer neural net can not find one template that matches all the threes:



- But a one-layer convnet can find one template that can do it with one convolution layer:



Outline

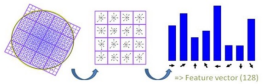
- Data structures
- Local reception fields
- Modeling of hierarchical organization
- One-layer convolutional neural network
- **Paradigm shift in computer vision**

From handcrafted to learned features

- Handcrafted approaches:

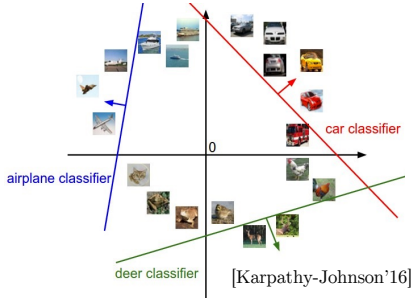
Input image

Handcrafted Features



SIFT, Lowe, 1999
(most cited paper in CV)

SVM classifier



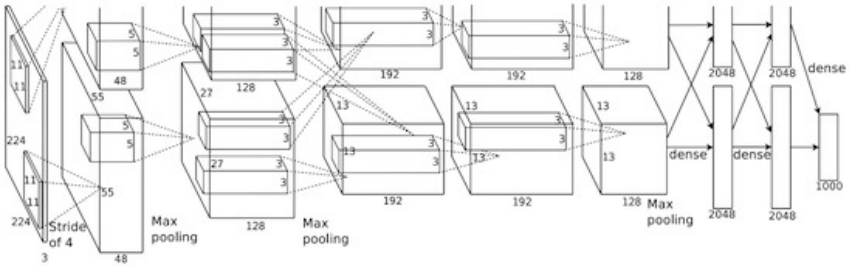
[Karpathy-Johnson'16]

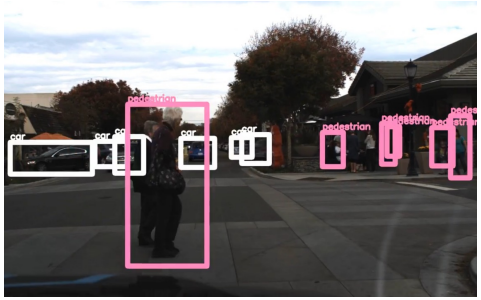
Before deep learning

- ConvNet approaches:
  - AlexNet made a breakthrough in 2012 in image recognition.

Input image

Learned features + Classifier





Deep learning

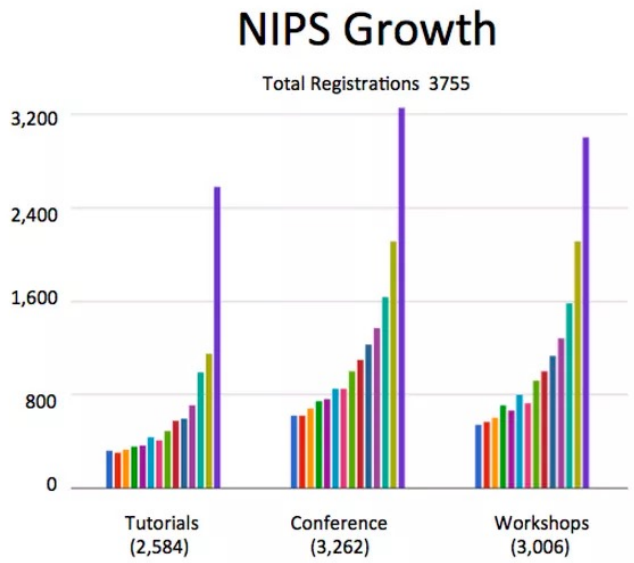
ConvNets

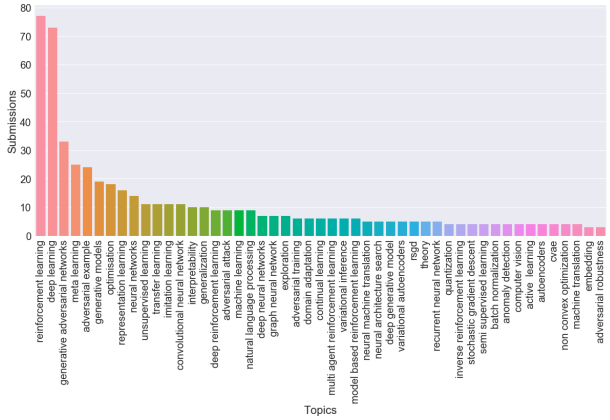
- Deep ConvNets:
  - Stack multiple convolutional+pooling layers
    - LeNet5, AlexNet, VGG, Inception, ResNet, etc

- ConvNets are extremely efficient at extracting compositional statistic patterns in large-scale and high-dimensional image datasets.

- Almost the entire ML, CV, NLP communities have moved to deep learning techniques and industry is investing considerably in deep learning.

- ICLR'19 topics

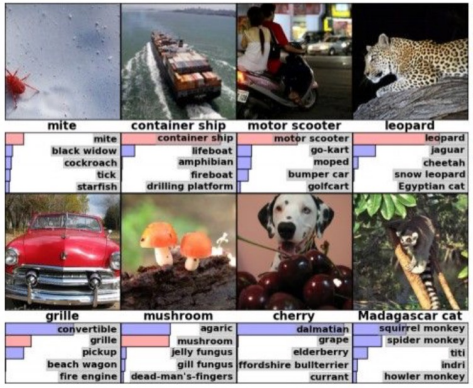




ConvNet applications

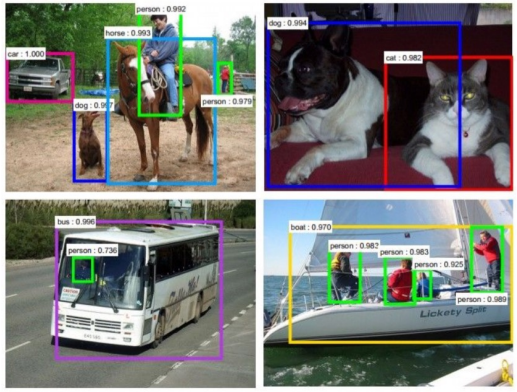
- ConvNets are today ubiquitous in computer vision !

Classification

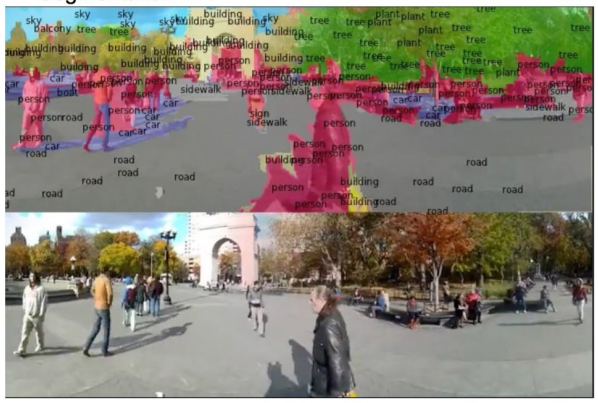


Retrieval

Detection

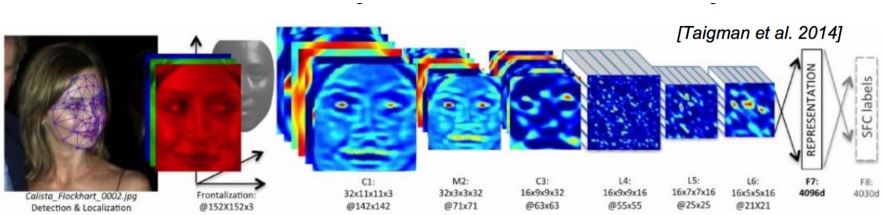


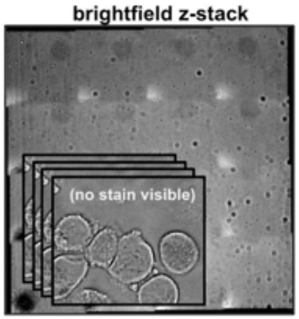
Segmentation

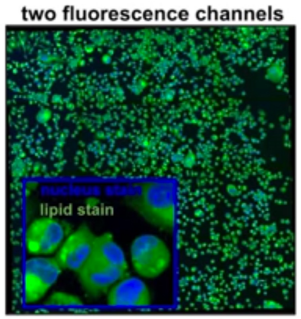


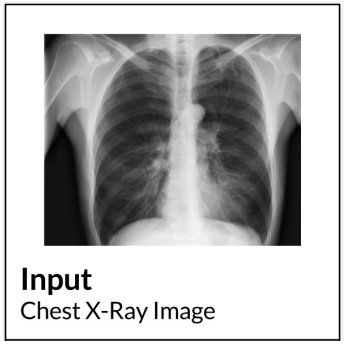
ConvNet applications

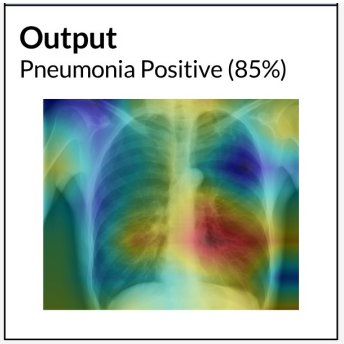
- ConvNets are today ubiquitous in computer vision !

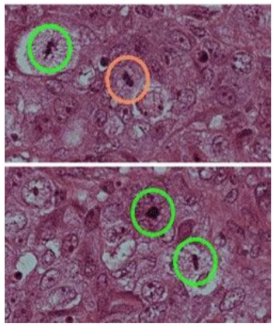


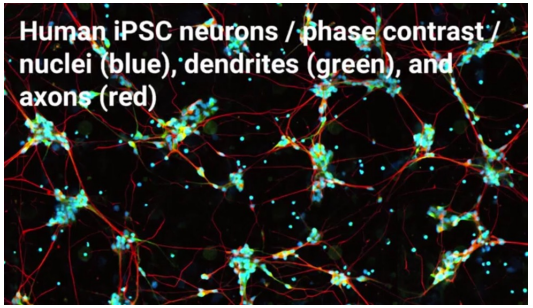








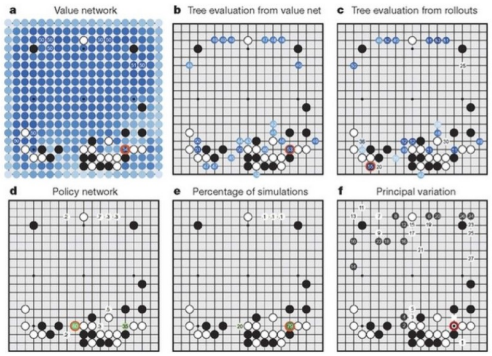




ConvNet applications

- ConvNets are today ubiquitous in computer vision !

self-driving cars



Questions?