Deep Learning Lab - Lecture 2

Last modified: Saturday 21st September, 2024, 14:50

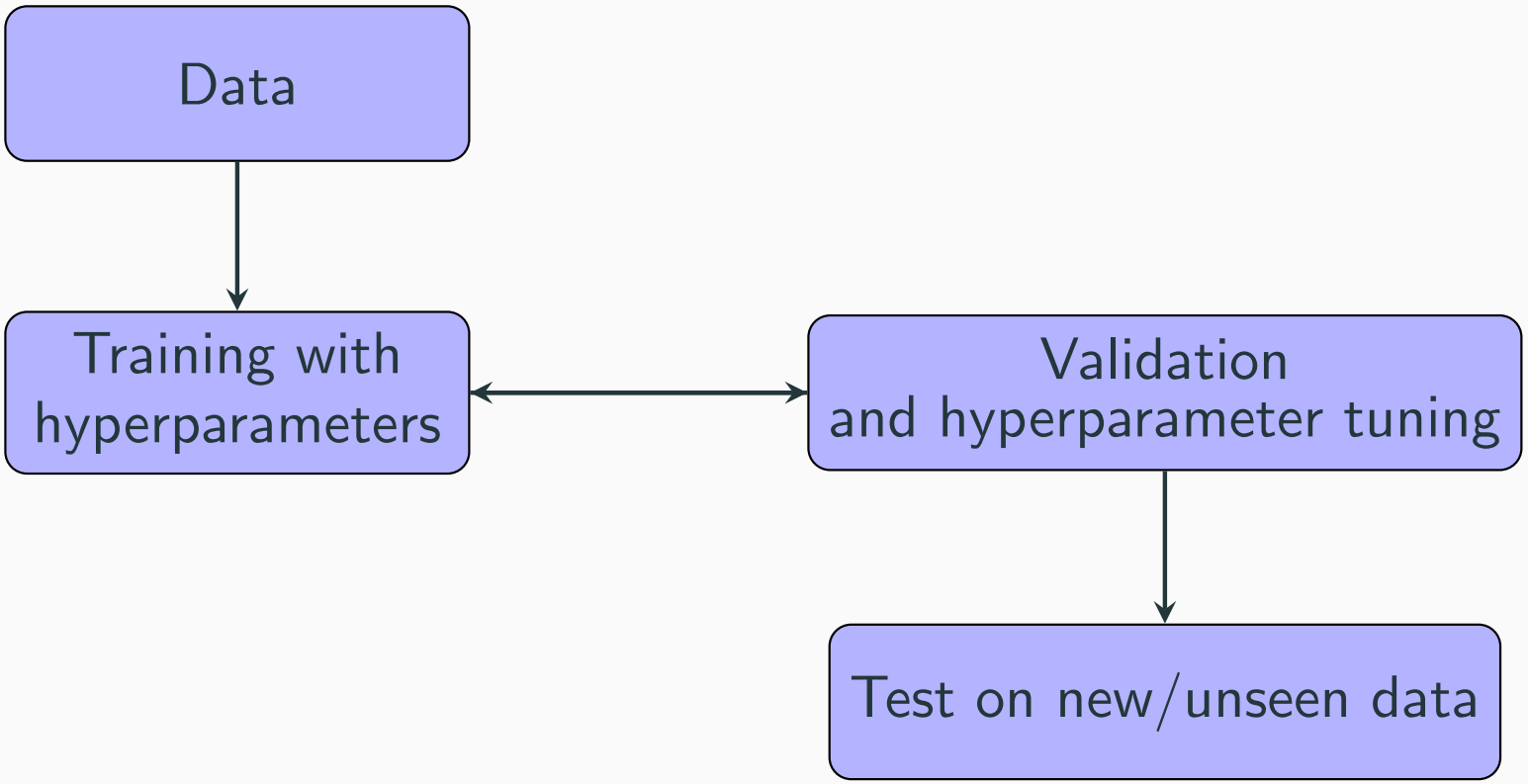
Instructor: E. Vercesi
TAs: A. Dei Rossi, G. Dominici, S. Huber
vercee@usi.ch - alvise.dei.rossi@usi.ch - gabriele.dominici@usi.ch - stefano.huber@usi.ch

Saturday 21st September, 2024

Today plan/objectives

- Learn the high-level framework of Deep Learning (DL)
- Learn some jargon
- Learn the basics of PyTorch
- Exercise 2 with Stefano
- Answering questions on Exercise 1?

High level pipeline



More specifically

- You should decide which type of data and a task

More specifically

- You should decide which type of data and a task
- You should define a *model*

More specifically

- You should decide which type of data and a task
- You should define a *model*
- You should decide how to train it (*optimization method & loss function)*

More specifically

- You should decide which type of data and a task
- You should define a *model*
- You should decide how to train it (*optimization method & loss function)*
- You should decide how to evaluate the performances of your model

Data

Data

- Learn from examples: if we want to distinguish dogs from cats, collect images of dogs, labeled as "dog", and cats, labeled as "cat".

Take-home messages:

- **Define** the task and collect **suitable data**
- Collect some basic info on your data *before* starting the learning process (number of classes, density of each class, ...)

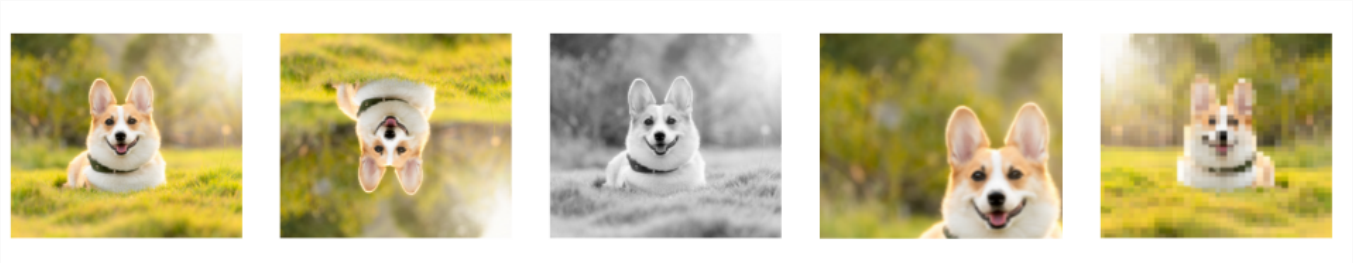
Open issues:

- How much data is enough?
- Are the data I have representative of the true distribution?

How much data is enough?

- **"10 times" rule of thumb:** the amount of input data (i.e., the number of examples) should be ten times more than the number of parameters in your data set.
- If we distinguish cat images from images of dogs based on 1000 parameters, you need 10000 pictures to train the model
- This only works for small models
- If data are not enough, you can use *Data Augmentation:* a quick overview in the framework of images (*image augmentation*)
- Crop, reduce, rotate, modify brightness, colors

Example of Image Augmentation



- Expand the number of items in the training set
- Allows the model to rely less on certain (positional) attributes
- Not recommended during tests

to the first image, u apply some modifications and u are going to get several images.
in practise works, but u don't have to abuse it.

Further readings:

- Section 14.1 of Zhang et al. [2021]
- Data Augmentation for audio
- Data Augmentation for text

Models / Architectures

Models

- **Note:** architecture depends on the task
- Remember the "Universal theorem":

Theorem

A feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation function (such as the logistic sigmoid activation function) can approximate any "good" function from one finite-dimensional space to another with any desired non-zero amount of error, provided that the network is given enough hidden units.

It does not say **how to find this network**!

- Training = find a good "approximator" of the true function based on data

Reminders on Neural Networks

- Parameterized function, parameters are unknown and learned from data
- Transform a vector to another vector
- Result of the composition of multiple functions of (basically) two types:
  - *layers*: one step forward (we will see it in detail)
  - *activation functions*: add non linearity
- Lot of layers = *Deep*

One-layer feed-forward neural network

Let $d_{in}, d_{out} \in \mathbb{N}$ respectively the dimension of the input and the output (We will see time by time the meaning.)

Let $g_{\boldsymbol{W},\boldsymbol{b}}(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$ an *affine function*

The function we want to learn in this case is

$$f : \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{out}}$$

$$f_{\boldsymbol{W},\boldsymbol{b}}(\boldsymbol{x}) := g_{\boldsymbol{W},\boldsymbol{b}}(\boldsymbol{x})$$

where $\boldsymbol{W}, \boldsymbol{b}$ are *unknown* and should be learned *from data*.

One-layer feed-forward neural network

Let $d_{in}, d_{out} \in \mathbb{N}$ respectively the dimension of the input and the output (We will see time by time the meaning.)

Let $g_{\boldsymbol{W},\boldsymbol{b}}(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$ an *affine function*

The function we want to learn in this case is

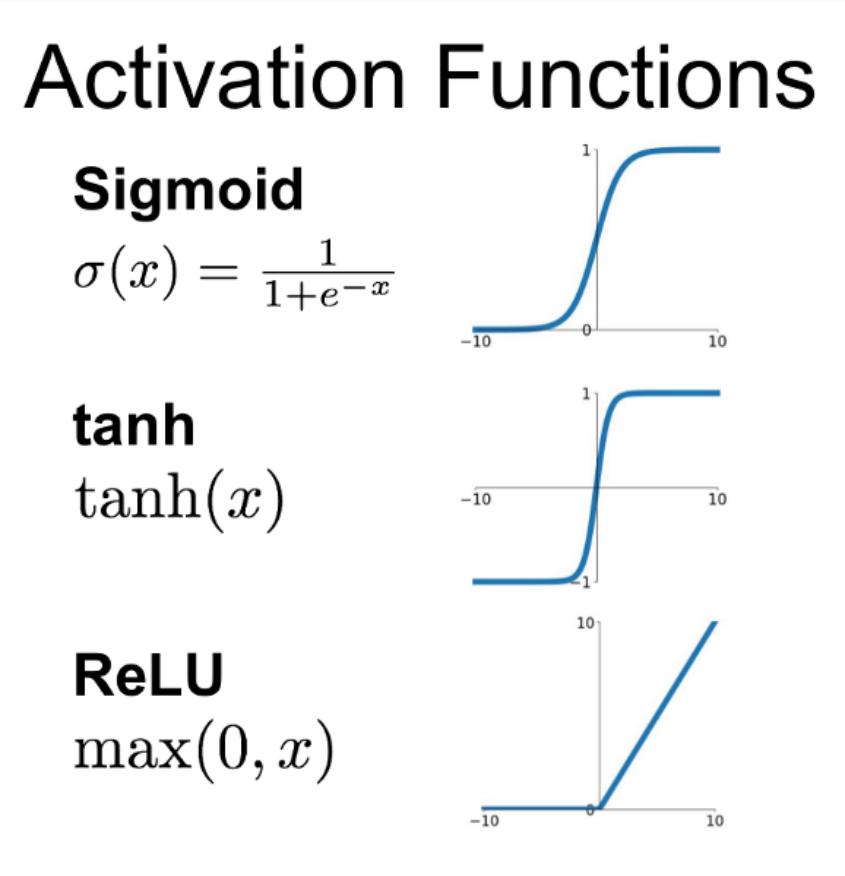
$$f : \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{out}}$$

$$f_{\boldsymbol{W},\boldsymbol{b}}(\boldsymbol{x}) := g_{\boldsymbol{W},\boldsymbol{b}}(\boldsymbol{x})$$

where $\boldsymbol{W}, \boldsymbol{b}$ are *unknown* and should be learned *from data*. Yes, this is basically a linear regression...

Activation functions

- Without any other function, concatenating two linear layers would lead to another linear layer
- Here is when *activation functions* come to play. The most common ones:



Activation functions

- Most recommended: ReLU
- Piecewise linear $\sim$ linear
- Preserve many properties that make models easy to optimize (=training) and generalize well
- The layers between input and output are called *hidden layers*

📖 Further readings:

Nice overview on activation functions.

Shall I change the activation function between layers or keep it the same?

- Again, no consensus!
- It depends on the task, but adding different activation functions makes things more complicated, e.g. in terms of differentiation
- It will take a while until we can "put things together and see what's going on", hence ...
- Demo with Tensorflow playgorund to show the impact of the activation function. You can try:
  - You can try with circular data the difference between ReLU and sigmoid
  - You can see the difference between ReLU and TanH in the grid data
  - ...

Task

Classification or Regression?

- **Classification** is a type of supervised machine-learning problem where the goal is to assign input data points to predefined categories or classes. Example: given some pictures, describe the content of each picture
- **Regression** A regression problem is a supervised machine learning problem where the goal is to predict a continuous numeric value or quantity. Example: Given some features, predict tomorrow's temperature.

Depending on the framework, different precautions or considerations must be adopted.

Training

Training

we want to reduce the penalty to reduce the "mistakes".

- Optimization process to find values for the model parameters that better resemble data
- Intuitively: "*Optimize* = reduce the penalty of having a bad prediction"
  - Before Training: parameters are random
  - After training: parameters should have values that allow the model to solve the task
- Two core elements of the training process
  - **Loss function:** function of the model parameter that should be minimized
  - **Specification of optimization algorithm**

Training data & learning strategy

- **Training data** is a collection of tuple $\{(x^i, y^i)\}_{i=1}^N$, where x^i represents the *features* of the ith sample, while y^i is called *target* or *output*.
- Assume that there is a true relation f between x^i, y^i, and we want to learn it based on data
- We want to train our model $f_{\boldsymbol{\theta}}$, where $\boldsymbol{\theta} \in \mathbb{R}^P$ are *learnable parameters*
- Loss functions

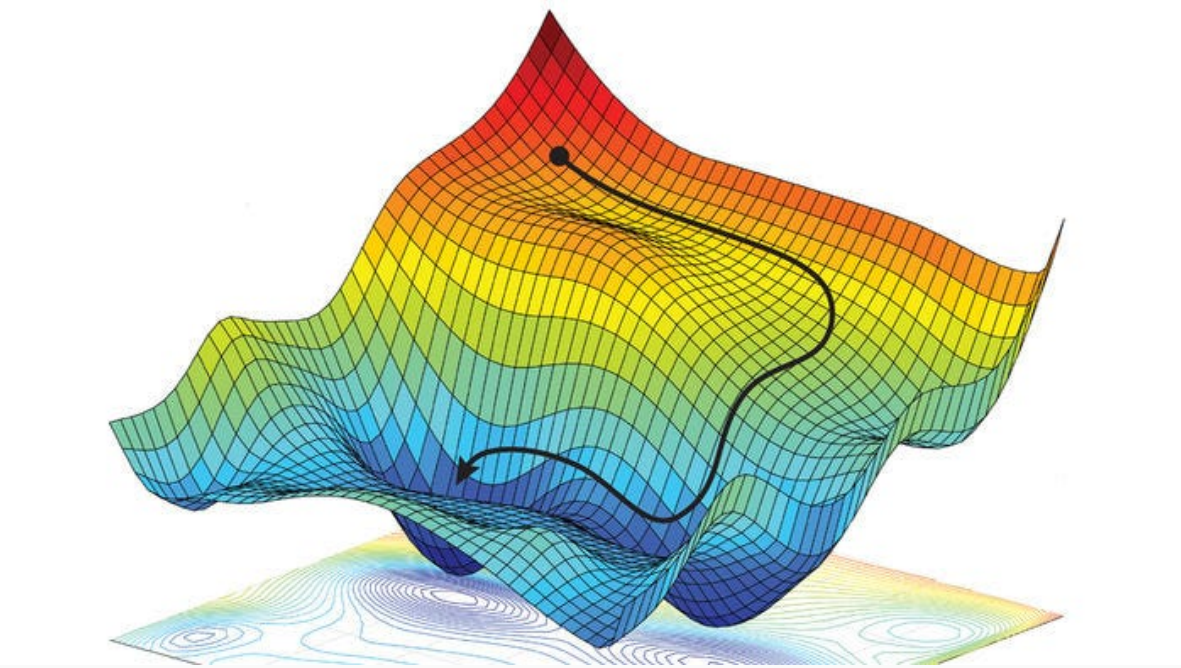
$$\boxed{\mathcal{L}(\boldsymbol{\theta}) := \frac{1}{N} \sum_{i=1}^{N} \ell(f_{\boldsymbol{\theta}}(x^i), y^i)}$$

A loss function measures how good a neural network model is in performing a certain task,

where ℓ compares the model output on x^i with the true value y^i

Minimizing loss

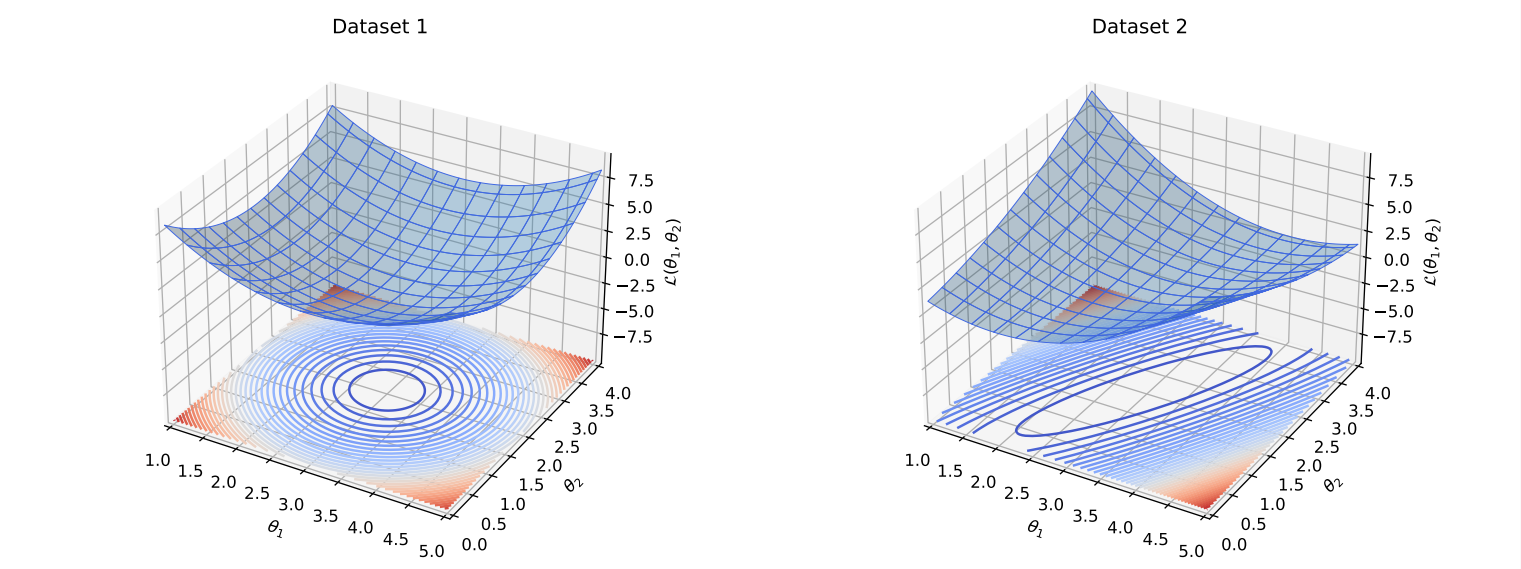
The picture you will always find online . . .



Nice plot, but does not explain the whole story!

Minimizing loss

- The shape of the loss function is provided by *data*
- Consider a loss function having two parameters, that is $\boldsymbol{\theta} = [\theta_1, \theta_2]$
- Consider two different datasets having as true parameter values $\boldsymbol{\theta}^* = [3, 2]$, same loss function, same architecture of the network
- Here are two losses: different shapes, with the minimum in the same place



Regression Task

- $y^i \in \mathbb{R}^d$ for some $d \in \mathbb{N}_+$
- $f_{\boldsymbol{\theta}}(x^i) \in \mathbb{R}^d$

$\Rightarrow$ choose $\ell(u, v) = ||u - v||_2^2$, hence

Mean Squared Error (MSE) loss: $\mathcal{L}(\boldsymbol{\theta}) := \frac{1}{N} \sum_{i=1}^{N} ||f_{\boldsymbol{\theta}}(x^i) - \boldsymbol{y}^i||_2^2$

Intuition: distance between the true function and the estimated one

Classification Task

Let C denote the number of output classes

- $y^i \in \{1, \ldots, C\}$
- Model output: a vector $f_{\boldsymbol{\theta}}(x^i) \in \mathbb{R}^C$ defining probability over the class labels:

$$f_{\boldsymbol{\theta}}(x^i) = [p_{\boldsymbol{\theta}}(1|x^i), p_{\boldsymbol{\theta}}(2|x^i), \ldots, p_{\boldsymbol{\theta}}(C|x^i)]$$

where

$$p_{\boldsymbol{\theta}}(k|x^i) = \text{ Probability that sample } x^i \text{ belongs to class } k$$

❓ Is there any way to compute the distance between probability distributions?

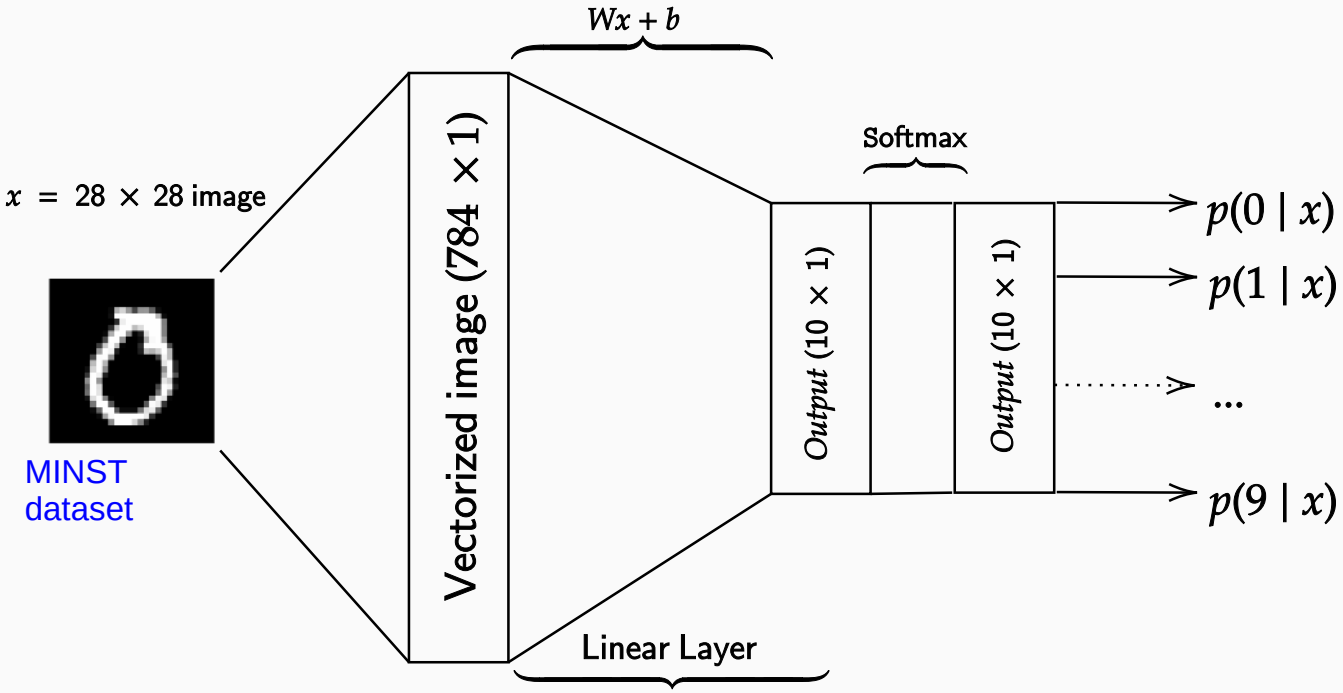
Normalized output with softmax in classification task

- Let C denote a positive integer
- $\boldsymbol{z} \in \mathbb{R}^C$
- Define

$$y_i := \text{Softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{d} e^{z_j}}$$

- The output y_i is such that
  - $0 \leq y_i \leq 1 \; \forall i \in \{1, \dots d\}$
  - $\sum_{i=1}^{C} z_i = 1$
- Probability distribution over classes!
- **Jargon:** The input of the softmax is called *logit*

One-layer feed-forward neural network



Classification Task

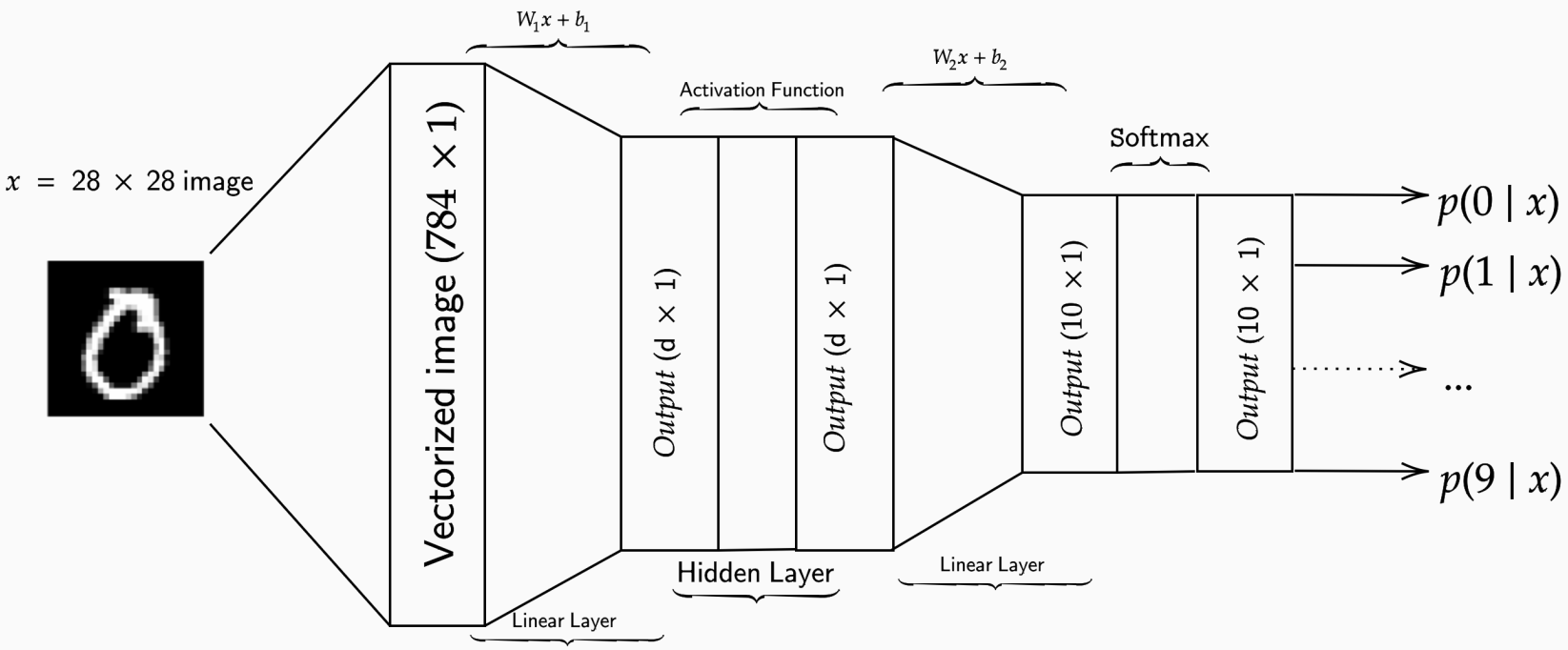
- If x^i belongs to the class k, the true probability vector $\boldsymbol{y}^i$ is the so-called *one hot encoded vector* $\in \mathbb{R}^C$, that is a vector with all the entries 0 and the kth equal to 1.
- Function *Dirac's delta function*

$$\delta_k(j) : \{1, \ldots, C\} \to \{0, 1\}$$

$$\delta_k(j) = \begin{cases} 1 & j = k \\ 0 & \text{otherwise} \end{cases}$$

- This, as well as the output of the softmax, can be associated with a probability distribution:

An illustration of a two layers feed-forward neural network with activation function

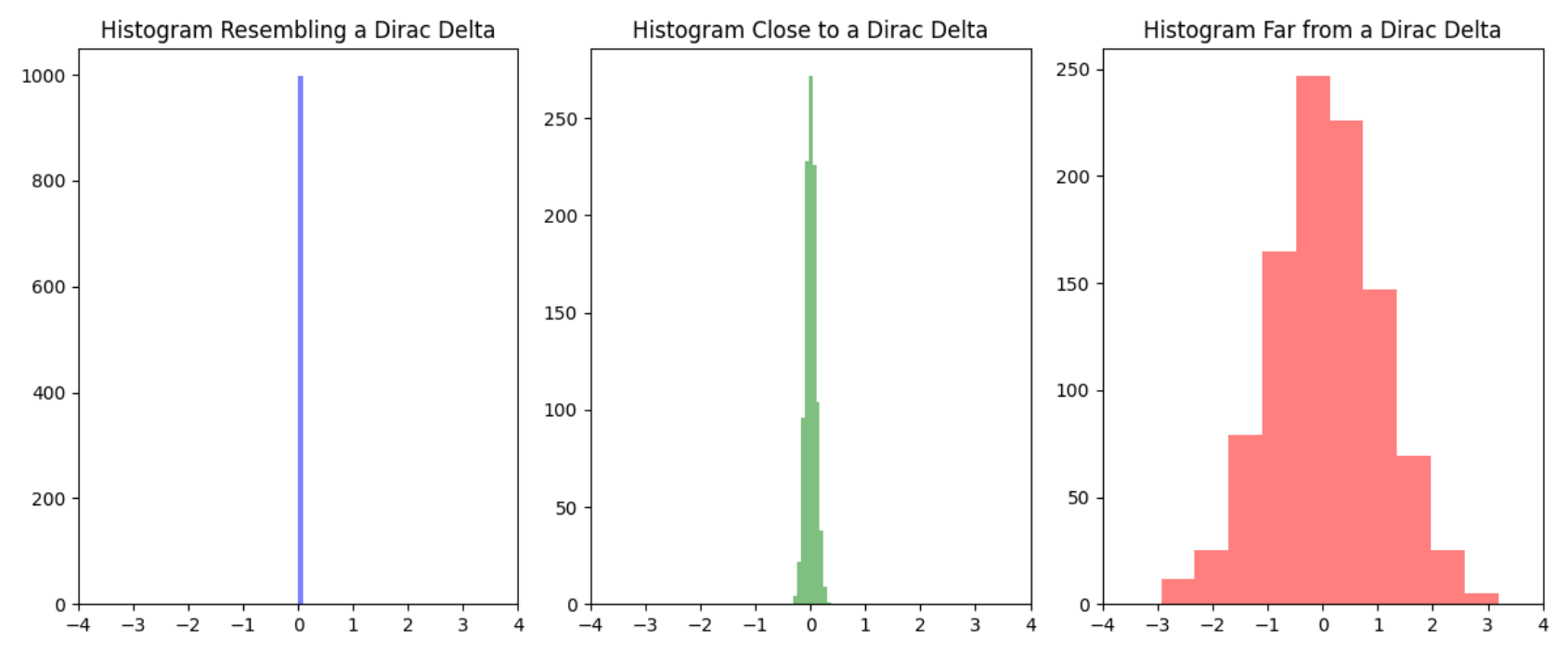


- The resulting function we want to learn is

$$g_{\boldsymbol{W}_1,\boldsymbol{b}_1} \circ \sigma \circ g_{\boldsymbol{W}_2,\boldsymbol{b}_2}$$

- d denotes the dimension of the hidden layer

Some intuitions on the distributions



Classification Task

- How can we compute the distance between $f_{\boldsymbol{\theta}}(x^i)$ and $\boldsymbol{y}^i$?
- A good measure is the *Kullback–Leibler divergence*

$$\ell(p, q) = \sum_{j \in \{1, \dots C\}} p(k) \log \left(\frac{p(k)}{q(k)} \right)$$

- if we replace p with $\boldsymbol{y}^i$ and q with $f_{\boldsymbol{\theta}}(\boldsymbol{x}^i)$, we have

$$\ell(\boldsymbol{y}^i, f_{\boldsymbol{\theta}}(\boldsymbol{x}^i)) = \sum_{j \in \{1, \dots C\}} \boldsymbol{y}^i_j \log \left(\frac{\boldsymbol{y}^i_j}{f_{\boldsymbol{\theta}}(\boldsymbol{x}^i)} \right)$$

Cross-Entropy Loss

- Note that, only $\boldsymbol{y}_k^i$ is not zero, and in particular is 1, and hence

$$\ell(\boldsymbol{y}^i, f_{\boldsymbol{\theta}}(\boldsymbol{x}^i)) = 1 \log \left(\frac{1}{f_{\boldsymbol{\theta}}(\boldsymbol{x}^i)_k} \right) = -\log(f_{\boldsymbol{\theta}}(\boldsymbol{x}^i)_k)$$

- Note that, the kth component of

$$f_{\boldsymbol{\theta}}(x^i) = [p_{\boldsymbol{\theta}}(1|x^i), p_{\boldsymbol{\theta}}(2|x^i), \ldots, p_{\boldsymbol{\theta}}(C|x^i)]$$

is precisely $p_{\boldsymbol{\theta}}(y^i|x^i)$, (that is, the true class), hence

$$\boxed{\textbf{Cross-Entropy Loss: } \mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} -\log p_{\boldsymbol{\theta}}(y^i|x^i)}$$

Other loss functions?

Just a couple of examples:

- For classification: Hinge Loss.
- For Regression: L1 loss function
- No recipe for choosing the best loss function: your experience + other people's experience

there isn't a rule to choose the best loss, but the experience and team working make it easier to understand.

Further readings:

- The PyTorch Documentation will provide a list of available loss functions

MSE > REGRESSION
CROSS > CLASSIFICATION

- This blog provides insights on how to choose it.
- Here a focus on regression tasks

Gradient descent

We start from the top and we go down, during the descent,.
Gradient descent is a method for unconstrained mathematical optimization. It is a first-order iterative algorithm for minimizing a differentiable multivariate function.

- Iterative process: $\boldsymbol{\theta}^n \rightarrow \boldsymbol{\theta}^{n+1}$
- Steps
  1. Compute gradient $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$
  2. Apply one gradient descent step:

$$\boldsymbol{\theta}^{n+1} = \boldsymbol{\theta}^n - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^n)$$

  3. Length of the step: $\alpha \rightarrow$ *learning rate*
- Gradient computed on the whole dataset?

Stochastic gradient descent (SGD)

We compute the loss function not on all datasets, but on a part.

- Alternative: update parameters using gradients computed on *mini-batches* (also *batches*)
- Gradients computed on a few, randomly selected data points
- Number of data points in a batch is the *batch size*
- **Jargon issue**: In some books/papers SGD = size of mini-batch = 1, and the other is mini-batch SGD
- As size mini-batch = 1 is a bit uncommn, we will use SGD for mini-batch SGD.

Change the learning rate at each iteration

- *Adaptive* methods, change the effective learning rate depending on the past gradients of the corresponding parameter.
- Designed to accelerate the optimization process, e.g. decrease the number of function evaluations required to reach the optimum
- "*Adagrad* [Duchi et al., 2011], an algorithm where coordinates that routinely correspond to large gradients are scaled down significantly, whereas others with small gradients receive a much more gentle treatment." (from Zhang et al. [2021])
- General recommendation: Adam or SGD

Which optimizer?

📖 Further readings:

- An overview of gradient descent optimization algorithms.
- Nice reading on the topic, with an overview of the methods, weakness/strength
- More details on Adam

Training Jargon

how much time we spend on training procedure?

- **Epochs:** 1 epoch = 1 run over the whole training data
- **Step/Update:** 1 step/update = 1 gradient update
- These are the definitions that are most commonly used within the DL community. Still, in papers or blogs, you may always come across alternative uses of these definitions - be careful!

Hyper-parameters i

- $\alpha =$?, batch size = ?
- How many layers? What is the size of each hidden layer?
- All of them are Hyperparameters = not learned by the algorithm, but tuned
- Model performances highly depends on the choice of the hyperparameter
- Only recommendations/suggestions, not clear rules

Hyper-parameters ii

Further readings:

- Picture taken from this nice blog on hyperparameter tuning
- There are some ways to tune hyperparameters *technicaly*, here a review,

Second part: objectives

- Learn some basics of **PyTorch 2.4**.
- Have a solid background for the next lectures and assignments.
- **Today**, Stefano will lead Exercise 2, you can find more theory in the appendix of these slides.

PyTorch

PyTorch is super easy, it's pythonic i.e you care about the content and not the code.
Based on the chart u can see that is most used than tensorflow.

Two core aspects

- GPU-friendly tensor
- Automatic differentiation

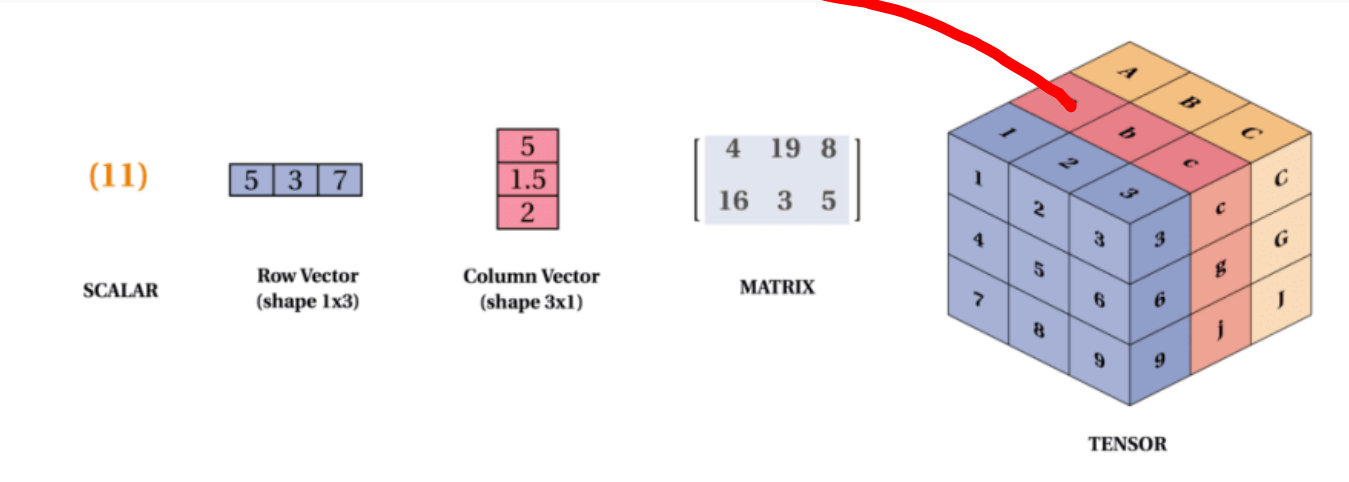
Why PyTorch over other tools?

- Very "Pythonic": you care about the content and not the code
- Most used recently

Nice comparison PyTorch against Tensorflow (You may understand why we chose PyTorch given this blog)

Tensors i

multidimensional matrix



Number of indices to access element = number of dimensions

That is, in the last case, $x_{0,0,0} = 1$

Are 3 dimensions enough?

Tensors ii

- One image in RGB = 3 dimensions needed
- Two images in RGB = 4 dimensions needed!

Corresponding class = `torch.Tensor`

```
> import torch
> x = torch.tensor([[1, 2],[3, 4]])
> print(x)
tensor([[1, 2],
        [3, 4]])
```

GPUs and Their Importance in Deep Learning

Super easy to use with GPUs

Graphics Processing Units (GPUs) have become essential in deep learning.

- Parallel Processing: GPUs excel at parallel tasks, speeding up training.
- Matrix Operations: Deep learning heavily relies on matrix calculations.
- CUDA: NVIDIA's CUDA platform enables efficient GPU programming (and behaves well with PyTorch)
- Even M1/M2 GPUs can be used for DL

Operation in DL are tensor operations

Transform a matrix-vector operation into a matrix-matrix operation!

- N data points, each of them having feature of dimension d_{in} and output of dimension d_{out}.
- Pack them into a single matrix multiplication

$$\boldsymbol{y} = \boldsymbol{W}^T \boldsymbol{x},\ \boldsymbol{x} \in \mathbb{R}^{d_{in}}, \boldsymbol{y} \in \mathbb{R}^{d_{out}}$$

$$\downarrow$$

$$\boldsymbol{Y} = \boldsymbol{W}\boldsymbol{X},\ \boldsymbol{X} \in \mathbb{R}^{N \times d_{in}}, \boldsymbol{Y} \in \mathbb{R}^{N \times d_{out}}$$

Some issues with GPUs

- Be aware of the GPU memory of the machine
- Classic error you will get: `RuntimeError: CUDA out of memory`
- Typical solutions. (a) Reduce the batch size (b) Run on more powerful GPUs

Automatic differentiation i

- Remind that, for the gradient descent method, we have to compute the gradient. In Pytorch, this computation is automatic.
- Every `torch.Tensor` has a `requires_grad` attribute, true if we need to compute the gradient for that tensor, false otherwise.
- Assume you have

$$f(x, y) = x^2 + y$$

- then

$$\nabla f(x, y) = \left[\frac{\partial f}{\partial x}(x, y), \frac{\partial f}{\partial y}(x, y)\right] = [2x, 1]$$

Automatic differentiation ii

- If we want to compute the gradient value in the point $(2, 3)$, then we have:

$$\boxed{\nabla f(2,3) = [4, 1]}$$

- Pytorch does exactly what we expect!

```
> x = torch.tensor(2, requires_grad=True,
    dtype=torch.float32)
> y = torch.tensor(3, requires_grad=True,
    dtype=torch.float32)
> z = x * x + y
> z.backward()
> print(x.grad)
> print(y.grad)
tensor(4.)
tensor(1.)
```

Automatic differentiation iii

- Note: gradients are accumulated. Consider the following example:

$$g(x, y) = x(y + 3)$$

- Then,

$$\nabla g(x, y) = \left[\frac{\partial g}{\partial x}(x, y), \frac{\partial g}{\partial y}(x, y)\right] = [(y + 3), x]$$

- If we want to compute in the point $(2, 3)$, then we have:

$$\nabla g(2, 3) = [6, 2]$$

Automatic differentiation iv

- But PyTorch returns instead

$$\left[\frac{\partial f}{\partial x}(2,3) + \frac{\partial g}{\partial x}(2,3), \frac{\partial f}{\partial y}(2,3) + \frac{\partial g}{\partial y}(2,3)\right]$$

As illustrated here

```
> # Continuing from the previous code
> w = x * ( y + 3)
> w.backward()
> print(x.grad)
> print(y.grad)
tensor(10.)
tensor(3.)
```

The gradient will be accomulated, there is a way to prevent in , in the next slide

Automatic differentiation v

- If you want to prevent this, you should use `x.grad.data.zero_()`. Consider the following example:

```
> x = torch.tensor(2, requires_grad=True, dtype=torch.float32)
> y = torch.tensor(3, requires_grad=True, dtype=torch.float32)
> z = x * x + y
> z.backward()
> print(x.grad)
tensor(4.)
> x.grad.data.zero_()
> print(x.grad)
tensor(0.)
> w = x * ( y + 3)
> w.backward()
> print(x.grad)
tensor(6.)
```

Accumulated gradients

- Traditional SGD updates model parameters after each data batch.
- Noisy updates can slow convergence.
- Accumulated gradients average over multiple batches, before updating the model parameters
- This smooth updates, reduces noise and ensures stable convergence.

📖 Further readings:

- Tutorial on accumulated gradient – we will talk about it
- Tutorial on how the PyTorch autograd system works

References i

J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021.

Tensors i

A fundamental attribute is the shape of a tensor, both include the number of dimensions and the size of each dimension. This can be checked with both `x.shape` and `x.size()`

```
>print("This tensor has {} dimensions of {}".format(
    len(x.size()), x.size()))
This tensor has 2 dimensions of torch.Size([2, 2])
```

The number of dimensions can be also obtained with `x.ndim`.

Tensors ii

Each tensor has an attribute, `dtype`, containing the type of the tensor elements. The attribute can be hard-coded:

```
> print(x.dtype)
torch.int64
> z = torch.tensor([1, 2], dtype=torch.float32)
> print(z.dtype)
torch.float32
```

Tensors - manipulating dimensions & access elements

- Get dimension, shape
- Change dimension (= reshape)
- Concatenate tensors
- Insert more dimensions
- Permute dimensions
- Access elements / slicing ...

You will see all of these commands to action in Exercise 2.

Conversion from / to numpy

Trivial:

```
> c = np.asarray([1, 2, 3])
> # From numpy to torch
> d = torch.from_numpy(c)
> # And back...
> f = d.numpy()
> print(d)
> print(f)
tensor([1, 2, 3])
[1 2 3]
```

Tensors: different types of operations

As we have seen in numpy, we have operations that do something *between* tensors (scalar product), and operations that are applied to each element of tensors (activation function on each element), as well as operations that take a tensor as input and return a single number as an output (maximum).

```
> a = torch.randint(0, 10, (6, ))
Z b = torch.randint(0, 10, (6, ))
> print("Scalar product between {} and {}".format(a, b))
> print(torch.dot(a, b))
Scalar product between tensor([2, 7, 6, 4, 6, 5]) and tensor([0, 4, 0, 3, 8, 4])
tensor(108)
> print("ReLu on {}".format(a))
> print(F.relu(a))
ReLu on tensor([2, 7, 6, 4, 6, 5])
tensor([2, 7, 6, 4, 6, 5])
> print("Maximum of the elements of {}".format(b))
> print(torch.max(b))
Maximum of the elements of tensor([0, 4, 0, 3, 8, 4])
tensor(8)
```

Note: be aware that on which operation is doing what to avoid errors!

Tensors on GPU i

- To use GPU, you need to copy tensors to GPU memory.
- Every `torch.Tensor` has the `device` attribute (basically CPU or GPU, tells the device on which the tensor is stored.).
- Here you can see how to transfer devices from CPU to GPU and vice-versa

Tensors on GPU ii

```
> a = torch.randint(0, 10, (2, 3), device='cpu')
> print("Starting device: ", a.device)
Starting device:  CPU
> # Move it to GPU after checking availability
> print("Is GPU available?", torch.cuda.is_available())
Is GPU available? True
> # General framework when you want to use GPUs with torch
> device = torch.device('cuda:0' if torch.cuda.is_available() else "cpu")
> b = a.to(device)
> print("Finally, we have it on", b.device)
Finally, we have it on cuda:0
```

Tensors on GPU iii

For M1/M2 Mac users:

```
> a = torch.randint(0, 10, (2, 3), device='cpu')
> print("Starting device: ", a.device)
Starting device:  CPU
> # Move it to GPU after checking availability
> print("Is GPU available?", torch.backends.mps.is_available())
Is GPU available? True
> # General framework when you want to use GPUs with torch
> device = torch.device('mps' if torch.backends.mps.is_available() else "cpu")
> b = a.to(device)
> print("Finally, we have it on", b.device)
Finally, we have it on mps:0
```

Tutorial about it on PyTorch Documentation