Machine learning for vision and multimedia
(01URPOV)

Lab 01 – Introduction to Pytorch
Francesco Manigrasso

2025 – 2026

Politecnico di Torino

Why Pytorch

- Pythonic Nature
  - Follows standard Python conventions
- Easy to learn
  - Intuitive syntax
  - Similar to numpy
- Strong Community

DL framework in a nutshell

- The backbone of a Pytorch/Tensorflow program is the **computational graph**
- Graphs are **data structures** composed by a series of **nodes** or **operations**, connected to each other by edges, which represent the units of data that flow between operations
- All data is stored in the form of **Tensors**
- Each node in the graph performs either an **operation** (or **op** for short), e.g., a math operation, or generates a tensor, like variables and constants
- Graphs can be saved, run, and restored without the original Python code

Recap – automatic differentiation

- Many machine learning approaches – and predominantly deep learning – rely on computing the derivatives of the loss w.r.t. the weights
- Key question: how can we compute, efficiently and effectively, the derivatives when we have thousands, millions or even billions of weights?
- Implementing backpropagation by hand is not convenient and does not scale: enters **automatic differentiation** (AD or autodiff)
- AD is a set of techniques to compute any derivative or gradient of a function, implemented by a computer program, **automatically** and with machine precision

Autodiff is not...

- **Symbolic differentiation** builds an analytical expression from a repository of basic rules, e.g., the sum rule or the product rule, much like manual computation
- Used in packages like Mathematica
- Pros:
  - Analytical expression
- Cons:
  - Expression swell
  - Inefficient: cannot reuse components
- The goal of autodiff is not a formula, but a procedure for computing derivatives

Autodiff is not...

- **Approximation of derivatives** through Taylor series (finite difference method)
- $f'(x_0) \approx \frac{f(x_0+h)-f(x_0)}{h}$
- Pros:
  - Good approximation
- Cons:
  - Computationally expensive: requires more than one forward pass
  - Numerically unstable and subject to truncation and round-off errors
- Finite differences can be used, in some cases, to test autodiff implementation

A tiny bit of history

AD is not a new concept, and it is not specific to deep learning either, with many applications in scientific computing

Robert Edwin Wengert. *A simple automatic derivative evaluation program*. Communications of the ACM **7***(8)*:463–4, Aug 1964.

A procedure for automatic evaluation of total/partial derivatives of arbitrary algebraic functions is presented. The technique permits computation of numerical values of derivatives without developing analytical expressions for the derivatives. **The key to the method is the decomposition of the given function**, by introduction of intermediate variables, **into a series of elementary functional steps.** A library of elementary function subroutines is provided for the automatic evaluation and differentiation of these new variables. The final step in this process produces the desired function's derivative. The main feature of this approach is its simplicity. It can be used as a quick-reaction tool where the derivation of analytical derivatives is laborious and also as a debugging tool for programs which contain derivatives.

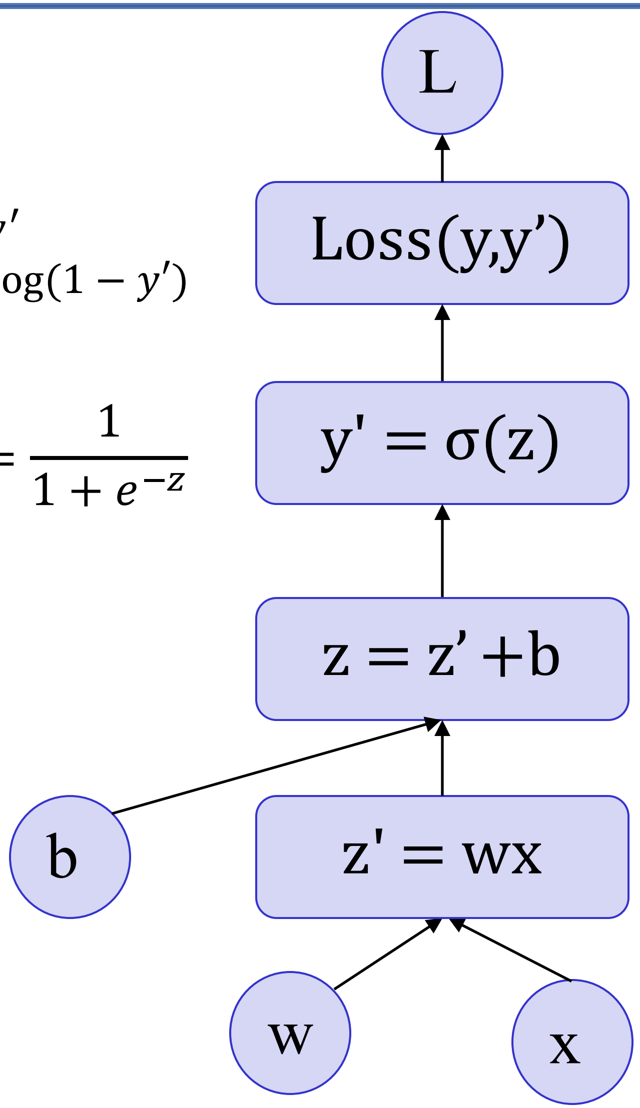
Computational graph

- Deep learning packages (e.g., Pytorch) employ a technique called **reverse mode autodifferentiation**
- It exploits two key constructs:
  - **The computational graph**
  - **The chain rule**
- to calculate automatically and mechanically derivatives as a sequence of primitives/operations
- All intermediate expressions are evaluated as soon as possible

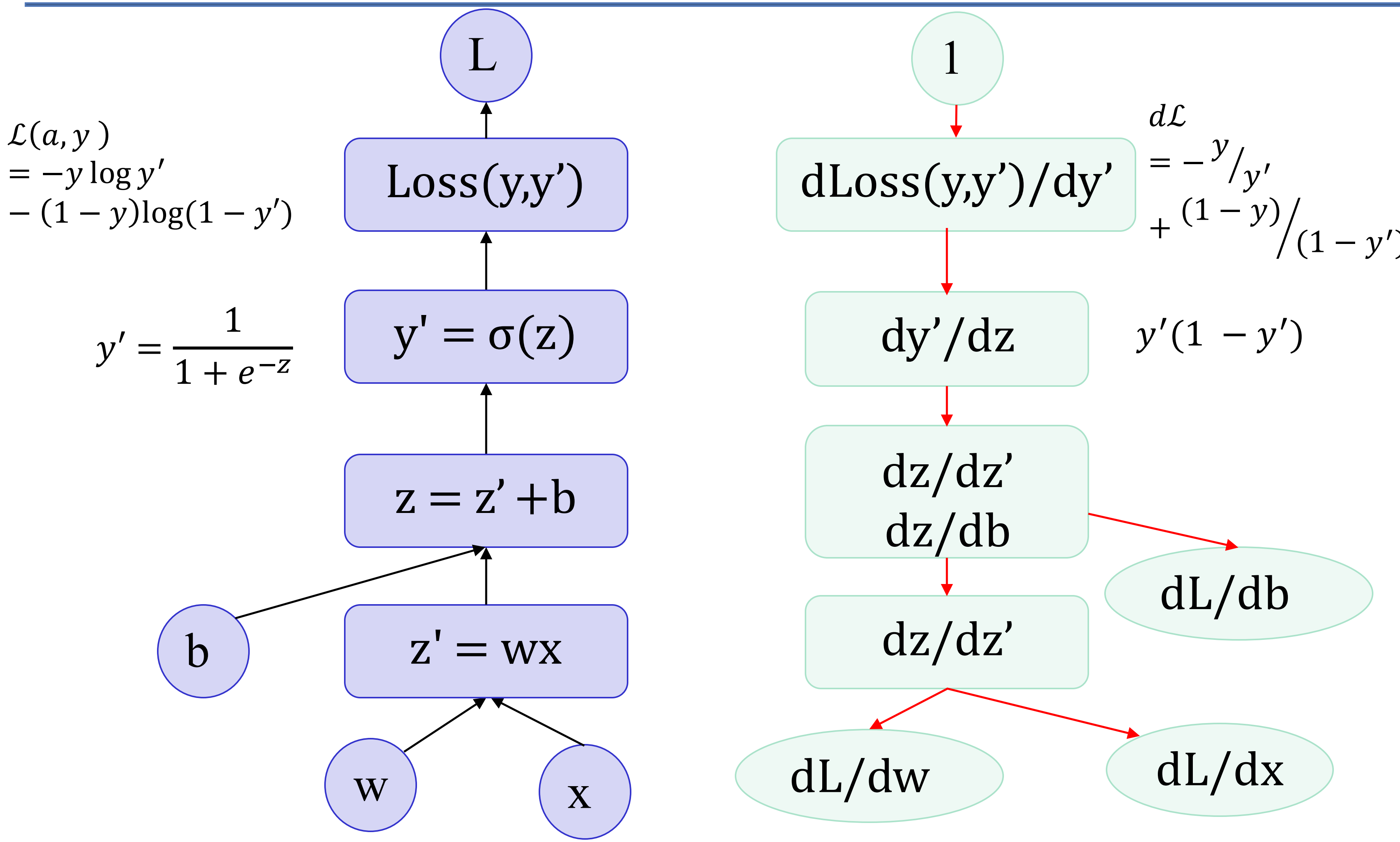
Computational graph

$$\mathcal{L}(a, y) = -y \log y' - (1 - y)\log(1 - y')$$

$$y' = \frac{1}{1 + e^{-z}}$$



Computational graph



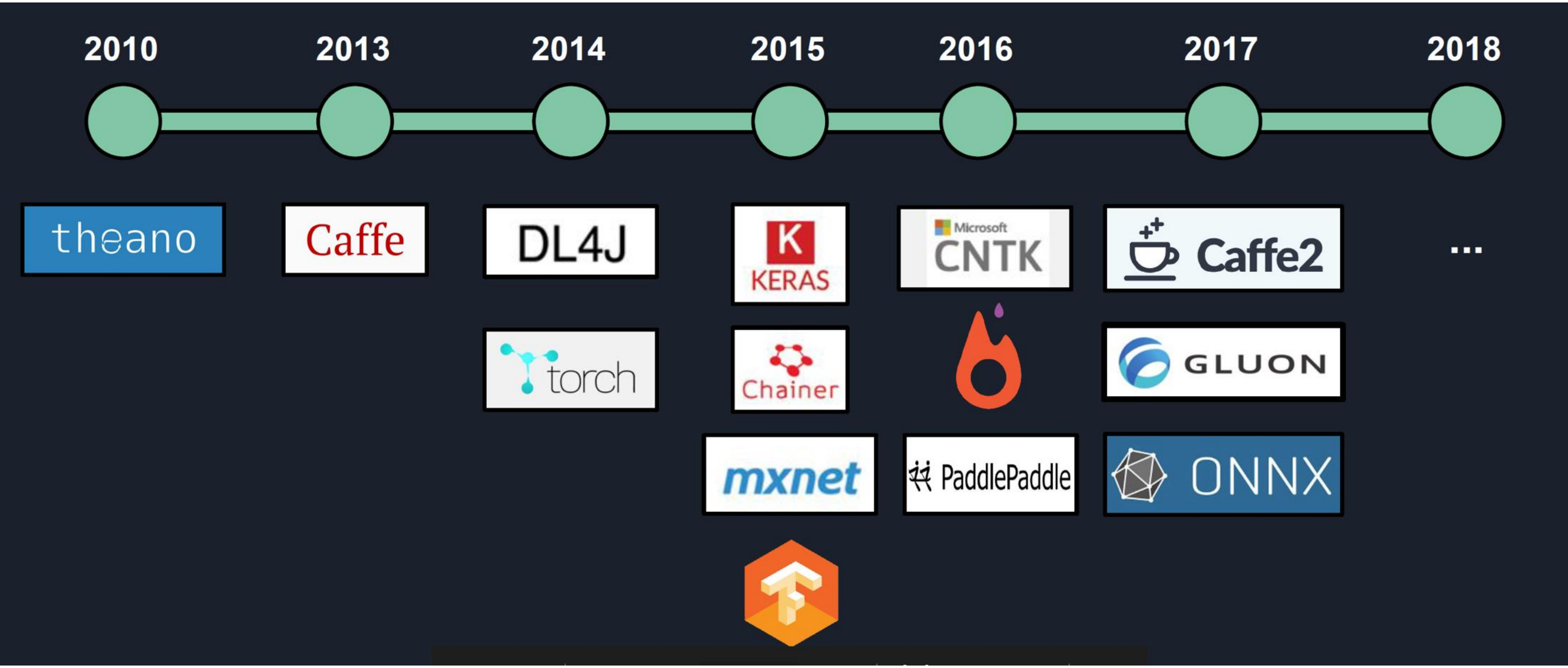
Chain rule

- To compute the derivative of an element of the graph C w.r.t. another element I:
  - Find the path in the computation graph from I to C
  - Backtrack from C to I, and for each operation on the backward path add a node to the graph
  - Compose the partial gradients along the backwards path using the chain rule
- In our regression example:
  - $\frac{d\mathcal{L}}{db} = \frac{d\mathcal{L}}{dy'}\frac{dy'}{dz}\frac{dz}{db}$
  - $\frac{d\mathcal{L}}{dw} = \frac{d\mathcal{L}}{dy'}\frac{dy'}{dz}\frac{dz}{dz'}\frac{dz'}{dw}$

Deep learning stacks

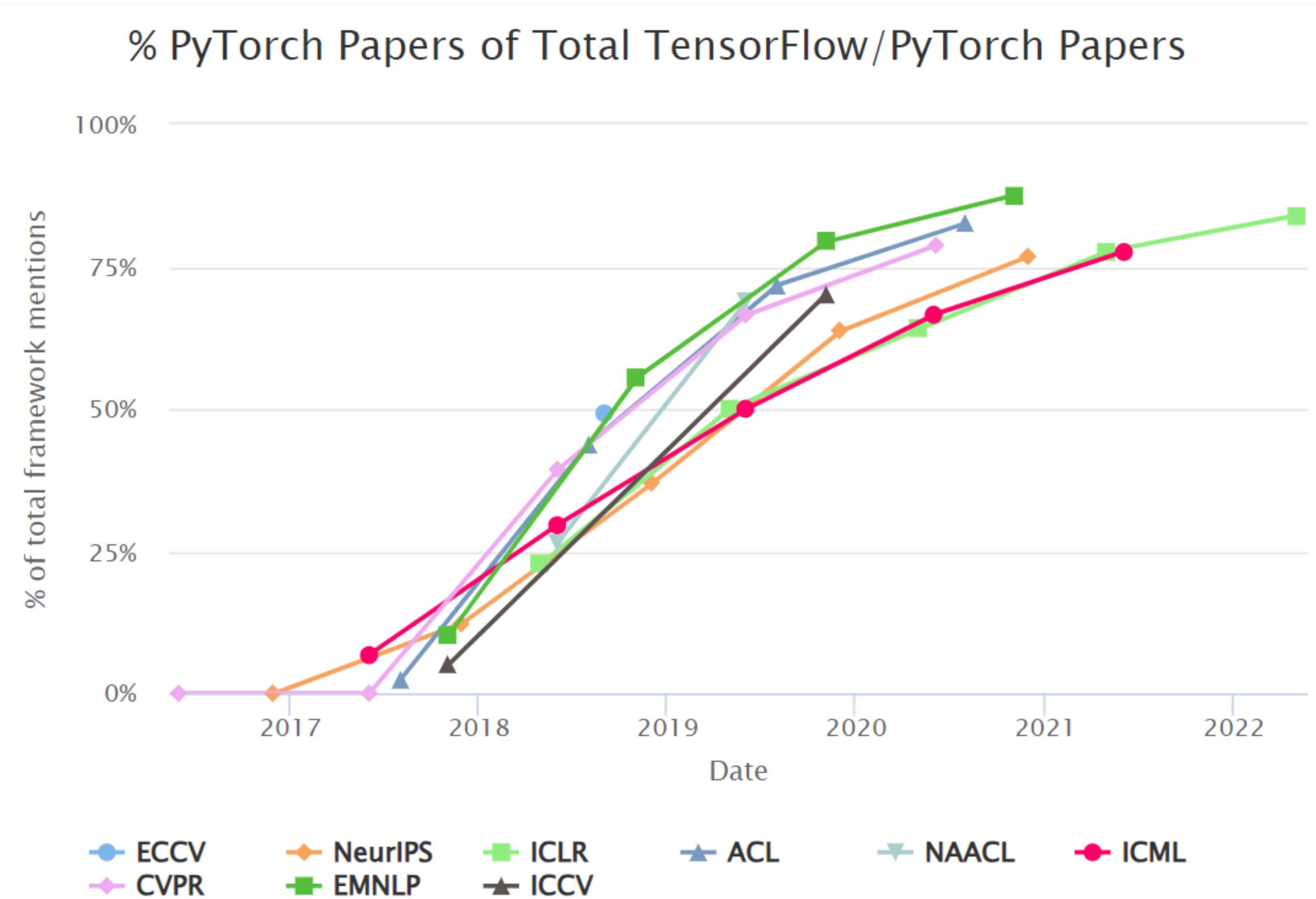
Level	Functionalities	Options
Analysis	Graphical representation Interaction	Tensorbard, DIGITS, …
Deep learning workflow management library	High-level network definition Dataset management	Keras, torch.vision, Lasagne, …
Computational graph management	Forward/backpropagation Gradient computation	Tensorflow, PyTorch, Caffe, Caffe2, …
Deep learning primitives	Low-level operations GPU optimization	cuDNN, CUDA
Hardware		CPU, GPU, TPU

Deep learning frameworks



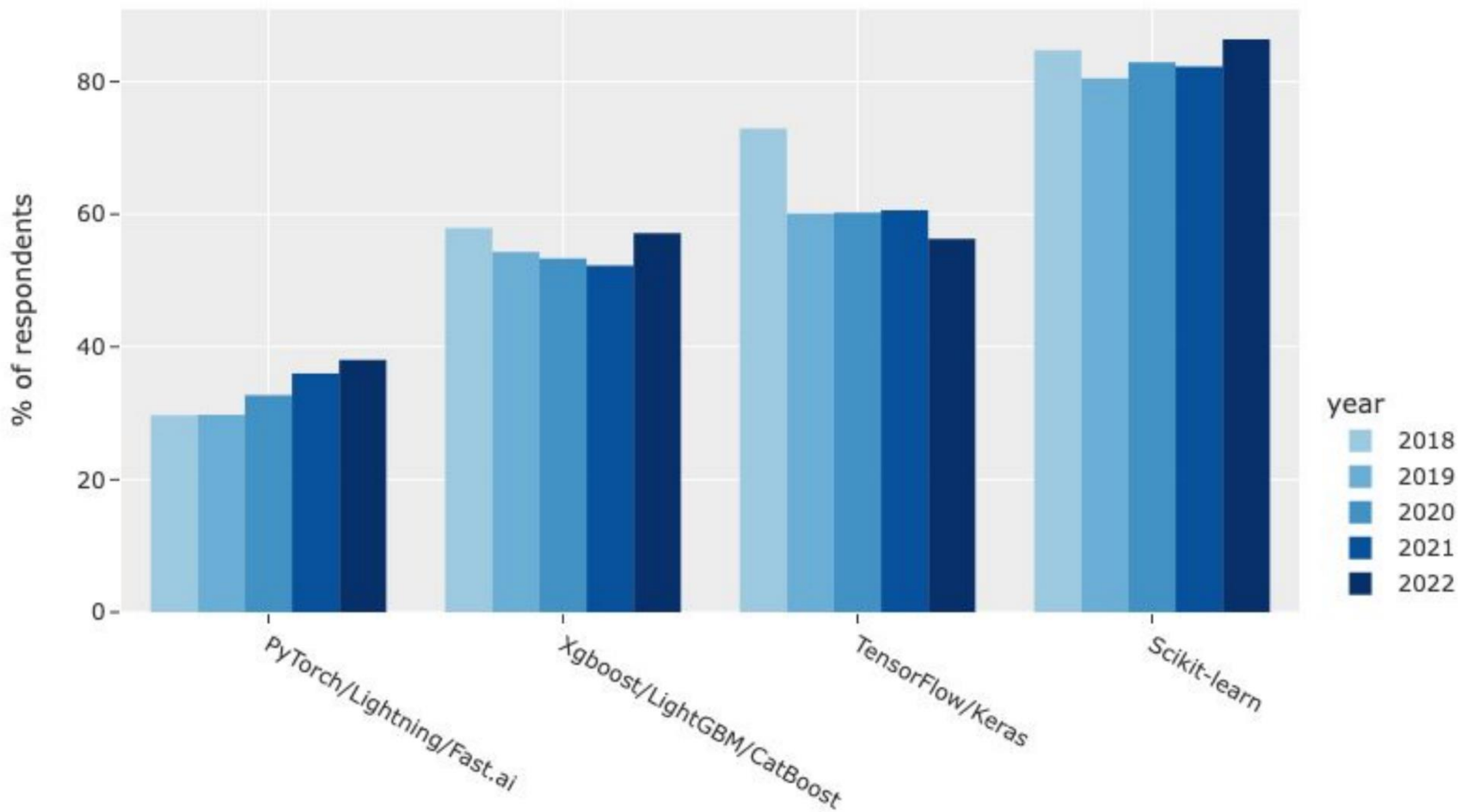
https://www.slideshare.net/VincenzoLomonaco/opensource-frameworks-for-deep-learning-an-overview

Deep learning frameworks (II)



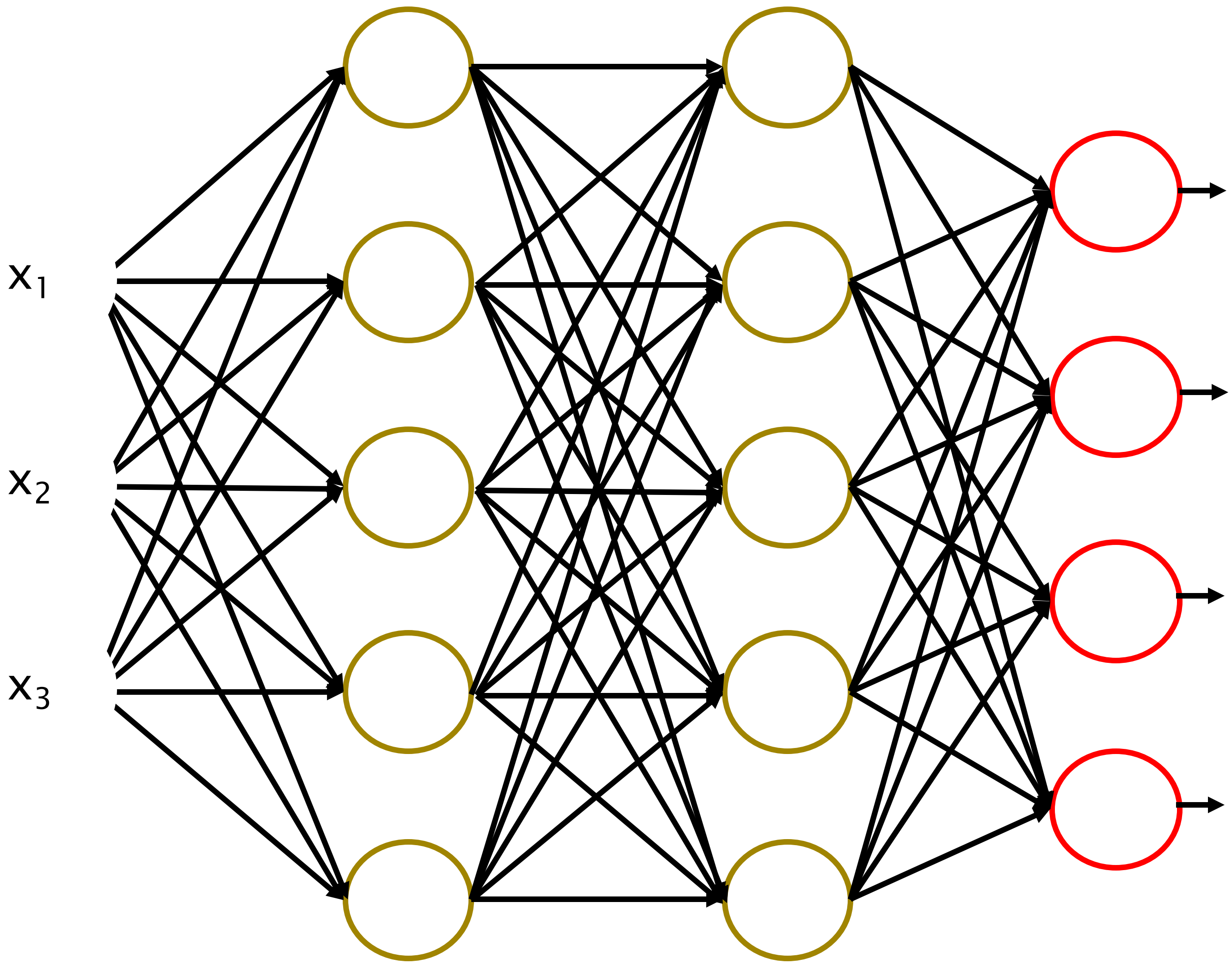
https://miguelgfierro.com/blog/2022/an-analysis-of-the-adoption-of-top-deep-learning-frameworks/

Deep learning frameworks (III)

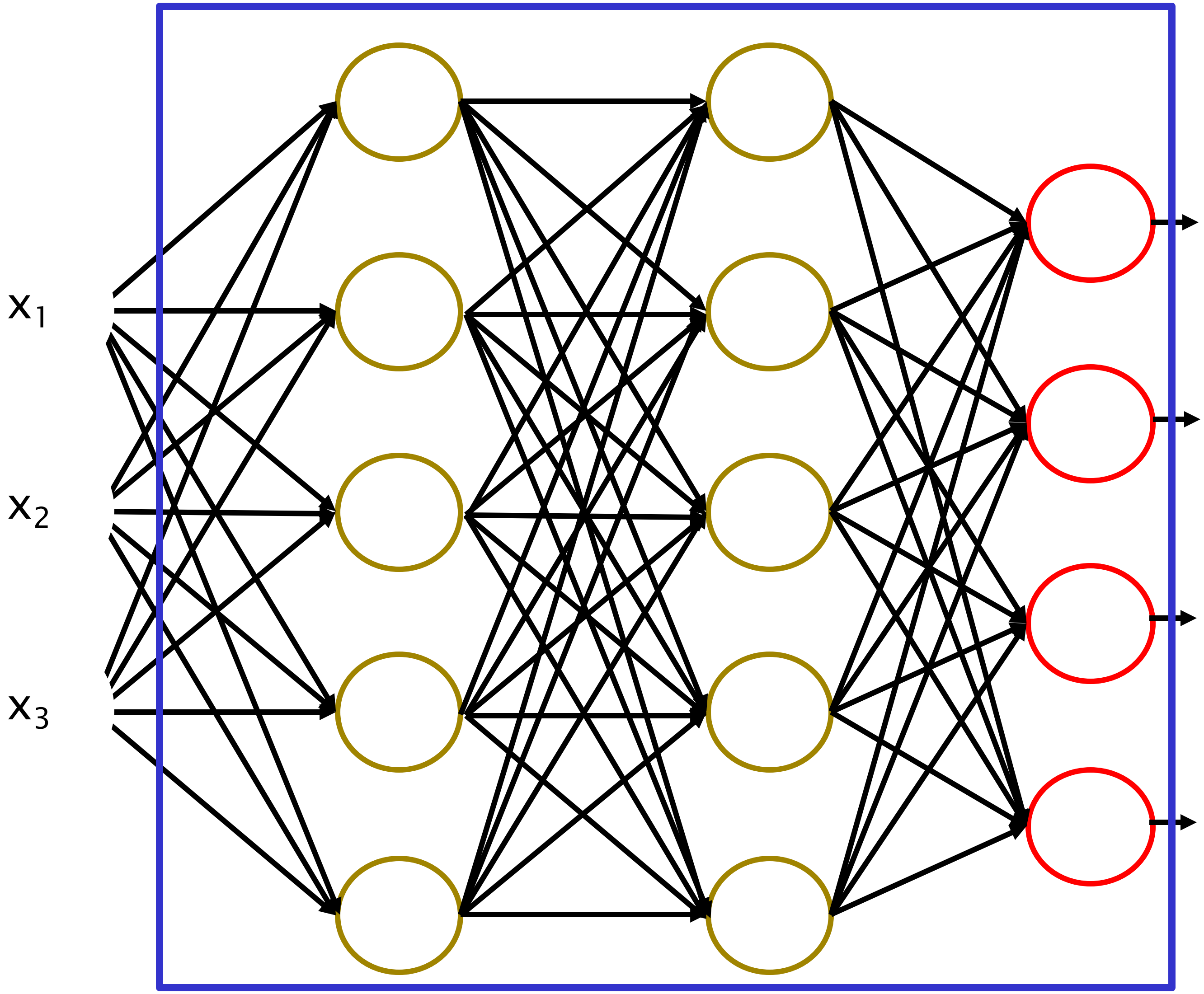


Source: https://www.kaggle.com/kaggle-survey-2022

Abstraction Layers



Abstraction Layers

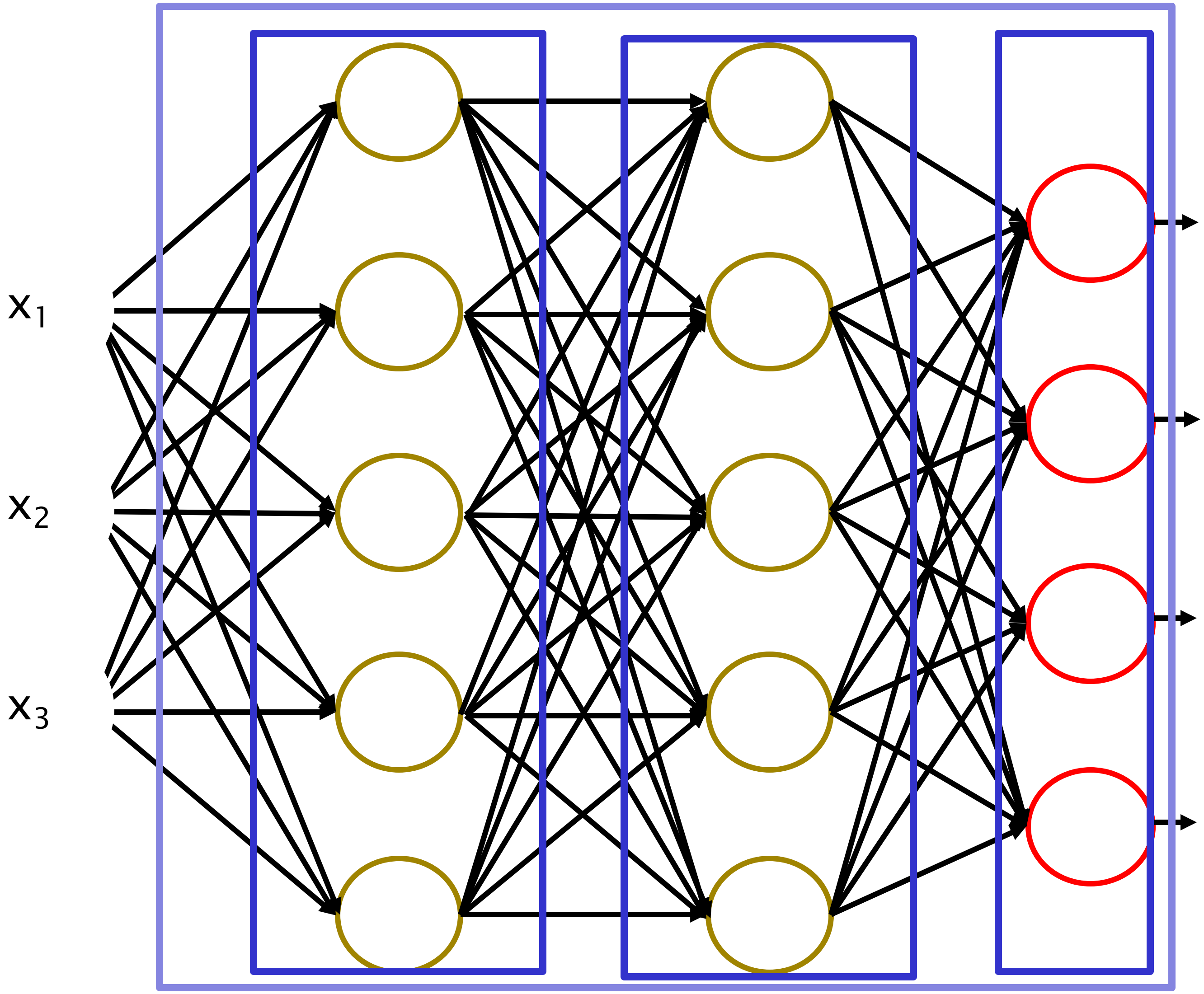


Model

$Y = f(x)$

The architecture (connectivity pattern) determines the space of functions that the model can learn

Abstraction Layers



Layer

$Y_{j+1} = f^{(j)}(y_j)$

Defines forward and backward ops

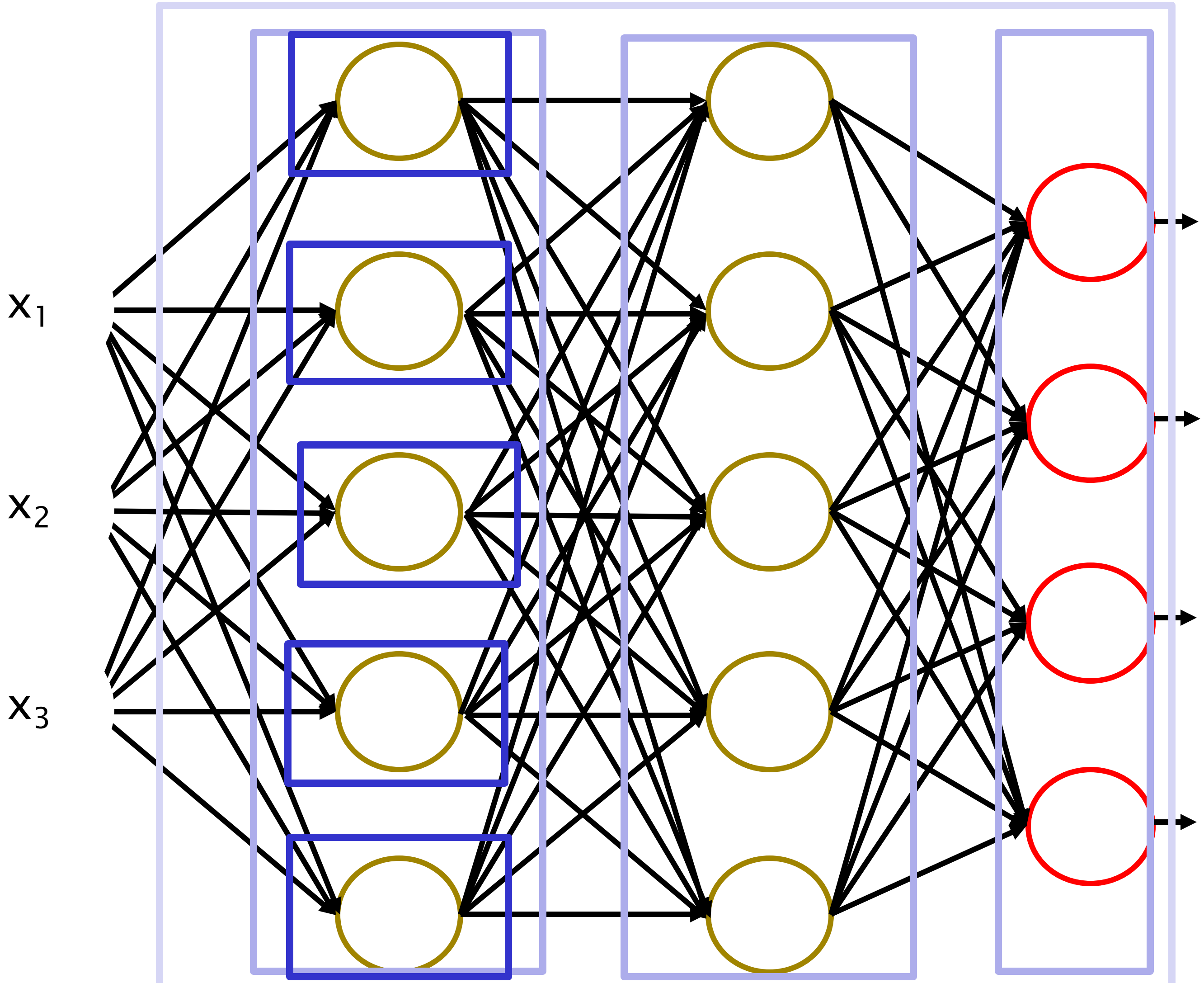
7-150 layers in a typical model

High-level abstraction

Rapid prototypes

Ability to create custom components

Abstraction Layers



Unit or op

Implementation of single operations within each layer (weight multiplication, activation function, etc.)

Vectorized implementation

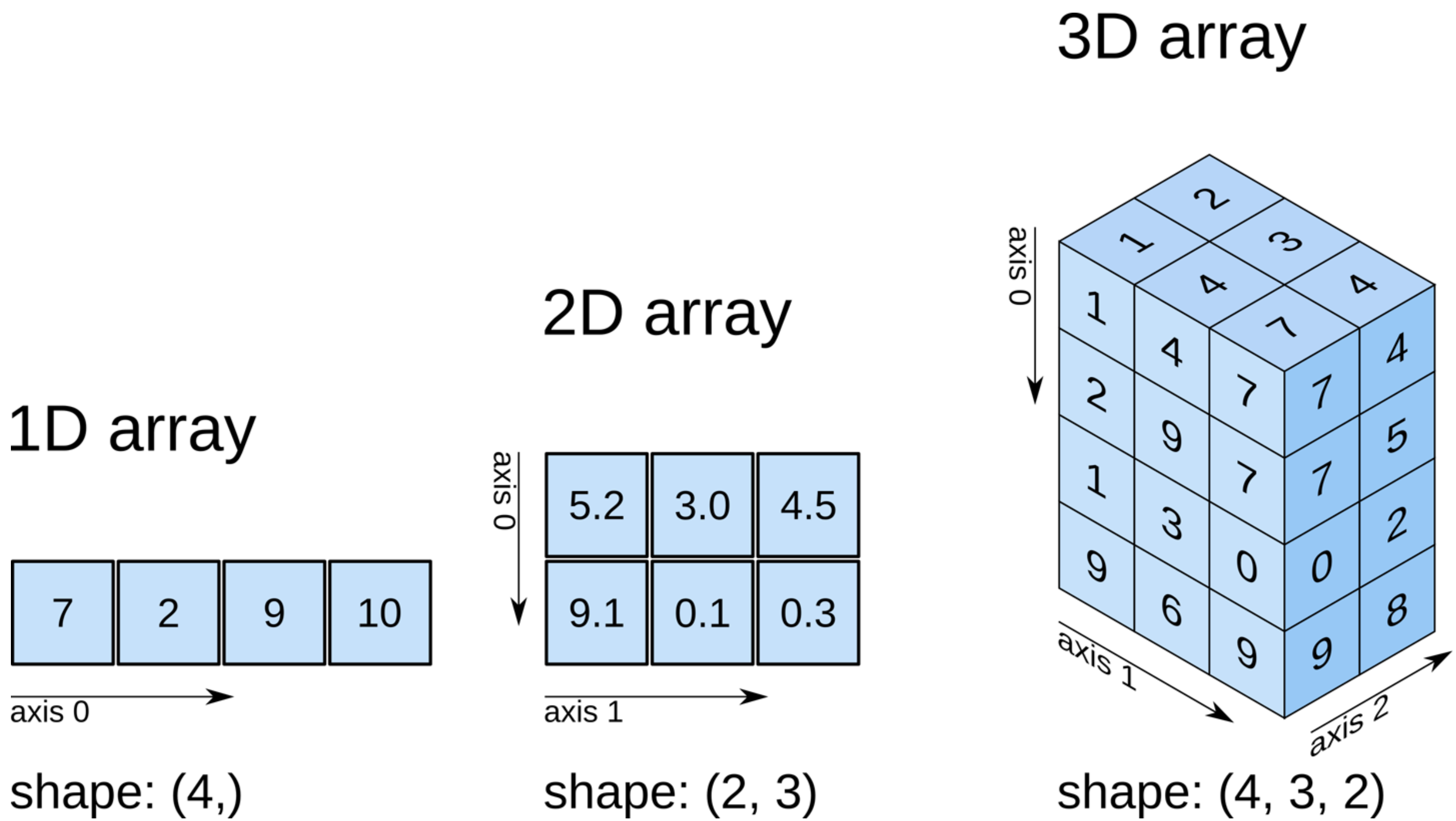
Hardware-independent

PYTORCH BASICS

Tensors

- Machine/deep learning libraries operate on **tensors**, a generalization of arrays and matrices to N-dimension
  - *"In the general case, an array of numbers arranged on a regular grid with a variable number of axes is known as a tensor" (I. Goodfellow, Deep learning, p.33)*
- An array is a grid of values, all of the same type, indexed by nonnegative integers
  - The number of dimensions is the **rank** of the array
  - The **shape** of an array is a tuple of integers giving the size of the array along each dimension
- Can be seen as counterparts of Numpy arrays

Tensors/Numpy Arrays



https://www.oreilly.com/library/view/elegant-scipy/9781491922927/ch01.html

Tensors

- All computation takes place on Tensors
  - Can be seen as counterparts of Numpy arrays
- Tensors encode
  - input data to process (images, text, etc.)
  - parameters (weights) of the network
  - output of the internal computations (activations)

```
#                          R     G     B
img = tensor([[[ 48,   80,   79],
               [175, 104, 207],
               [162,   24, 224],
               [ 97,   27,   28],
               [ 51, 137,   60],
               [124, 214, 249]],
                ...
            ]])
t.size() ==> [256, 256, 3]
t.device ==> gpu:0
t.dtype ==> torch.float32
```

Tensors

- Properties of a tensor
  - shape – dimensions of tensor t (equivalent to t.size() method)
  - dtype – data type of values (float, double, int, etc.)
  - requires_grad – a Boolean value indicating whether the tensor requires gradients to be computed during backpropagation
  - device –the location where the tensor is stored, e.g., CPU or GPU
- The rank and shape of output tensors are defined by the mathematical operations performed

Tensor creation

- Create a Tensor t with shape (3, 5)
- **init with zeros**
  - t = torch.zeros(size=(3, 5), dtype=torch.float32)
- **init with ones**
  - t = torch.ones(size=(3, 5), dtype=torch.float32)
- **init randomly in the range** (0,1)
  - t = torch.rand(size=(3, 5), dtype=torch.float32)

```
In [4]: torch.zeros(size=(3, 5), dtype=torch.float32)
Out[4]:
tensor([[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]])
```

```
In [5]: torch.ones(size=(3, 5), dtype=torch.float32)
Out[5]:
tensor([[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]])
```

```
In [7]: torch.rand(size=(3, 5), dtype=torch.float32)
Out[7]:
tensor([[0.2781, 0.1713, 0.3152, 0.6436, 0.3709],
        [0.2789, 0.4106, 0.1298, 0.6884, 0.5782],
        [0.0018, 0.7707, 0.4201, 0.9484, 0.5120]])
```

Tensor reshape

- **Change shape** of a tensor while **keeping** the **number** of elements
- `Tensor.view:`
  - returned tensor share the underlying data with the original tensor
- `Tensor.reshape:`
  - more flexible
  - may allocate a new tensor

```
In [55]: # Create tensor X with shape (N,C,W,H)
    ...: N, C, W, H = 50, 3, 28, 28
    ...: X = torch.randn((N, C, W, H), dtype=torch.float32)
    ...: print('\nOriginal: ', X.shape)
    ...:
    ...: # Reshape tensor with torch.view
    ...: # X: (N, C, W, H) -> X_r: (N, C, W * H)
    ...: X_r = X.view(N, C, 784)
    ...: print('View: ', X_r.shape)
    ...:
    ...: # A small trick
    ...: # torch.view can automatically infer one dim
    ...: # specify dim to guess with -1
    ...: X_r2 = X.view(-1, C, 784).shape
    ...: print('View - 2: ', X_r2)
    ...:
    ...: # Alternatively we can use torch.reshape
    ...: print('Reshape: ', torch.reshape(X, (-1, C, 784)).shape)

Original:  torch.Size([50, 3, 28, 28])
View:  torch.Size([50, 3, 784])
View - 2:  torch.Size([50, 3, 784])
Reshape:  torch.Size([50, 3, 784])
```

- A single dim. may be –1, in which case it is inferred from the remaining dimensions

Tensor device

- Tensors are allocated on a specific device
  - By default, PyTorch tensors are allocated on the CPU
- Move tensor between CPU and GPU
  - `tensor.cuda()` move to the GPU
  - `tensor.cpu()` move to the CPU
  - needs a **CUDA capable** device

```
In [27]: # Create tensor X with shape (N,C,W,H)
    ...: N, C, W, H = 100, 3, 28, 28
    ...: X = torch.randn((N, C, W, H), dtype=torch.float32)
    ...: print()
    ...: print("On creation: ")
    ...: print(f"X: \nshape: {X.shape},\ndtype: {X.dtype},\ndevice: {X.device}\n")
    ...:
    ...: # move tensor X - CPU -> GPU
    ...: X = X.cuda()
    ...: print(f"X to GPU: \nshape: {X.shape},\ndtype: {X.dtype},\ndevice: {X.device}\n")
    ...:
    ...: # move tensor X - GPU -> CPU
    ...: X = X.cpu()
    ...: print(f"X to CPU: \nshape: {X.shape},\ndtype: {X.dtype},\ndevice: {X.device}\n")
    ...:

On creation:
X:
shape: torch.Size([100, 3, 28, 28]),
dtype: torch.float32,
device: cpu

X to GPU:
shape: torch.Size([100, 3, 28, 28]),
dtype: torch.float32,
device: cuda:0

X to CPU:
shape: torch.Size([100, 3, 28, 28]),
dtype: torch.float32,
device: cpu
```

Tensor device (II)

- Once a tensor is allocated, you can do operations on it irrespective of the selected device
  - Results will be always placed on the same device as the tensor
- Cross-device operations are not allowed!!!
  - x1.cpu() + x2.cpu() ⇒ ok
  - x1.cpu() + x2.cuda() ⇒ **error**
  - x1.cuda() + x2.cpu() ⇒ **error**
  - x1.cuda() + x2.cuda() ⇒ ok
- OSS. the same holds also if x1 and x2 are on two different GPUs

Conversion to/from Numpy

- numpy array ⇒ pytorch tensor
  - torch.from_numpy()
- **pytorch tensor ⇒ numpy array**
  - **tensor.numpy()**
  - tensor must be on cpu before invoking numpy()!!!

```
In [80]: x_arr = np.array([1., 41., 19.])  # create numpy ndarray
    ...: x_tensor = torch.tensor([1., 41., 19.])  # create pytorch tensor
    ...: print()
    ...: print("array: ", x_arr)
    ...: print(f"array: shape: {x_arr.shape}, dtype: {x_arr.dtype}")
    ...:
    ...: print()
    ...: print("tensor: ", x_tensor)
    ...: print(f"tensor: shape: {x_tensor.shape}, dtype: {x_tensor.dtype}")
    ...:
    ...: # to and from numpy, pytorch
    ...: print('\nNumpy array to pytorch tensor')
    ...: print("Numpy => Torch: ", torch.from_numpy(x_arr))
    ...:
    ...: print('\nPytorch tensor to numpy array')
    ...: print("Torch => Numpy: ", torch.from_numpy(x_arr))
```

```
array:  [ 1. 41. 19.]
array: shape: (3,), dtype: float64

tensor:  tensor([ 1., 41., 19.])
tensor: shape: torch.Size([3]), dtype: torch.float32

Numpy array to pytorch tensor
Numpy => Torch:  tensor([ 1., 41., 19.], dtype=torch.float64)

Pytorch tensor to numpy array
Torch => Numpy:  tensor([ 1., 41., 19.], dtype=torch.float64)
```

Broadcasting

- Broadcasting allows, under certain conditions, to combine arrays with incompatible shapes during mathematical operations
- The smaller array is “broadcasted” by copying its content across the larger array, thus avoiding:
  - looping in Python (computational inefficiency)
  - creating array copies (memory inefficiency)
- Many PyTorch operations support NumPy’s broadcasting semantics.

Broadcasting example

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 6 & 10 \end{bmatrix} * [5] = \begin{bmatrix} 1 & 2 & 3 \\ 5 & 6 & 10 \end{bmatrix} * \begin{bmatrix} 5 & 5 & 5 \\ 5 & 5 & 5 \end{bmatrix} = \begin{bmatrix} 5 & 10 & 15 \\ 25 & 30 & 50 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 6 & 10 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 5 & 6 & 10 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 5 & 0 & 0 \end{bmatrix}$$

TRAINING A DEEP MODEL IN PYTORCH

Training a deep learning model

- Find the best model parameters θ^* that minimize the loss functions over a given dataset

$$\theta^* = \arg\min_{\theta} \sum_{(x,y)\in \mathcal{D}} \mathcal{L}\left(f_\theta(x) , y\right)$$

Training a deep learning model

- Find the best model parameters θ^* that minimize the loss functions over a given dataset

$$\theta^* = \arg\min_{\theta} \sum_{(x,y)\in \mathcal{D}} \mathcal{L}\left(\boxed{f_\theta}(x) \, , \, y \right)$$

Neural network (architecture)

Training a deep learning model

- find the best model parameters θ^* that minimize the loss functions over a given dataset

Labels

$$\theta^* = \arg\min_{\theta} \sum_{(x,y)\in \mathcal{D}} \boxed{\mathcal{L}}\left(f_\theta(x) , \boxed{y}\right)$$

Loss function

Training a deep learning model

- Find the best model parameters θ^* that minimize the loss functions over a given dataset

$$\theta^* = \arg\min_{\theta} \sum_{\boxed{(x,y)\in \mathcal{D}}} \mathcal{L}\left(f_\theta(x) , y \right)$$

Training set
(data loader)

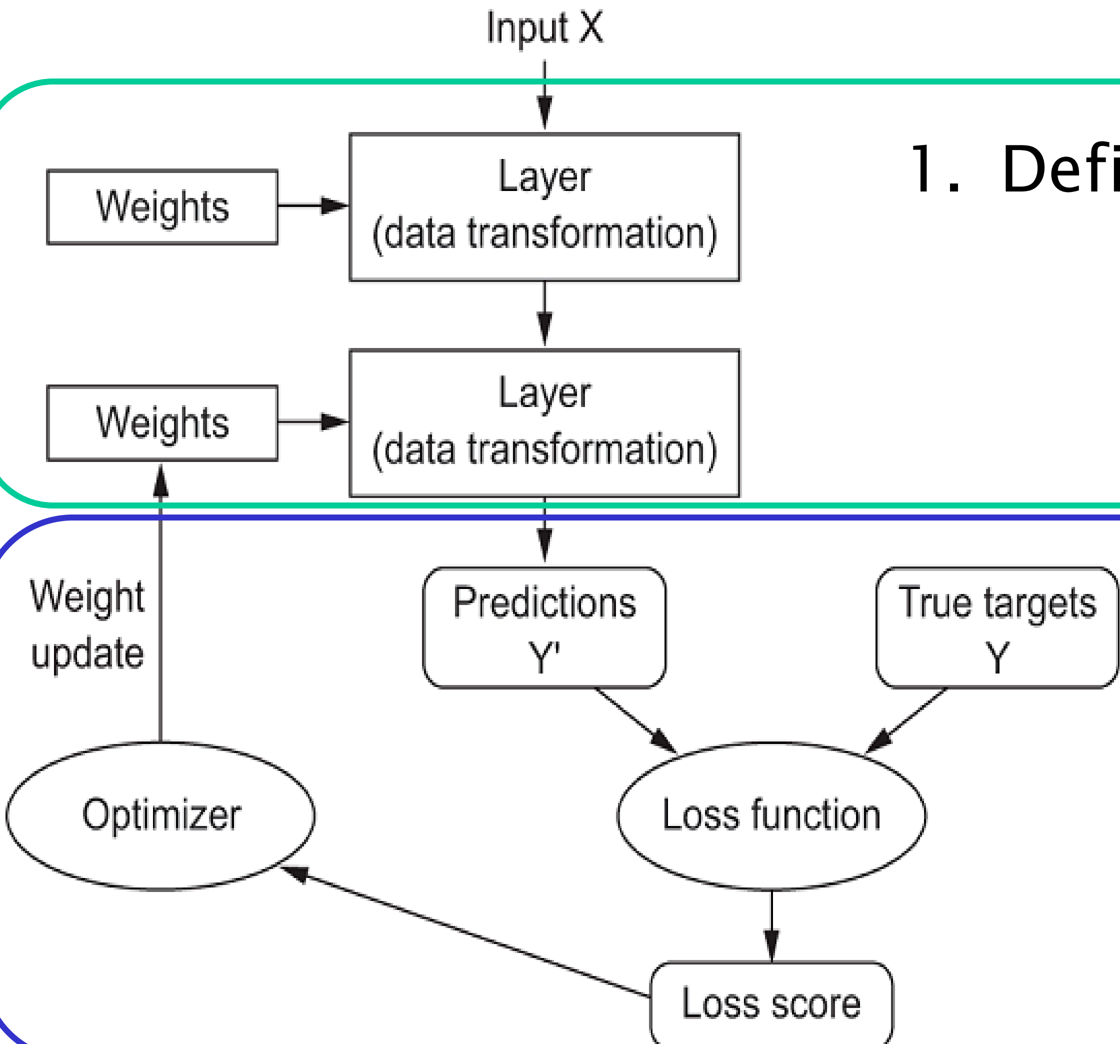
Training a deep learning model

- find the best model parameters θ^* that minimize the loss functions over a given dataset

$$\theta^* = \boxed{\arg\min_{\theta}} \sum_{(x,y)\in\mathcal{D}} \mathcal{L}\left(f_\theta(x) \, , \; y \right)$$

Gradient (computed by autodiff)
Descent (optimizer)

In practice…



1. Define the **model**

2. Define the **training loop**

a. Forward pass
b. Compute Loss
c. Backward pass
d. Optimization and weight update

Repeat until convergence…..

Training loop

Step	Implementation
Forward pass	– Defined in the neural network model – Standard layers are available as Pytorch element
Compute loss	– Most common losses are already implemented in Pytorch – Custom losses can be defined
Backward pass	– Gradients are automatically computed by Pytorch Autograd (no need to define the backward pass)
Optimization	– Most common optimizers are already implemented in Pytorch

Pytorch basic libraries

- **torch.nn.Module**
  - losses and network component
- **torch.optim**
  - Optimizers to update network parameters
- **torch.utils.data.Dataset**
  - to define a dataset classes
- **torch.utils.data.Dataloader**
  - to efficiently load batches of samples from the dataset

Batch Processing

- Functions and modules from torch.nn process batches of inputs stored in a tensor whose first dimension indexes them
  - The batch size is the number of samples processed before the model is updated
- Given a training dataset with **N data samples** and **batch size B**
  - We feed to the network a batch of B data samples at a time
  - Loss + optimization is performed for each training batch
  - A complete pass through the whole training dataset is called **epoch**
  - (typically) an epoch is N/B batches

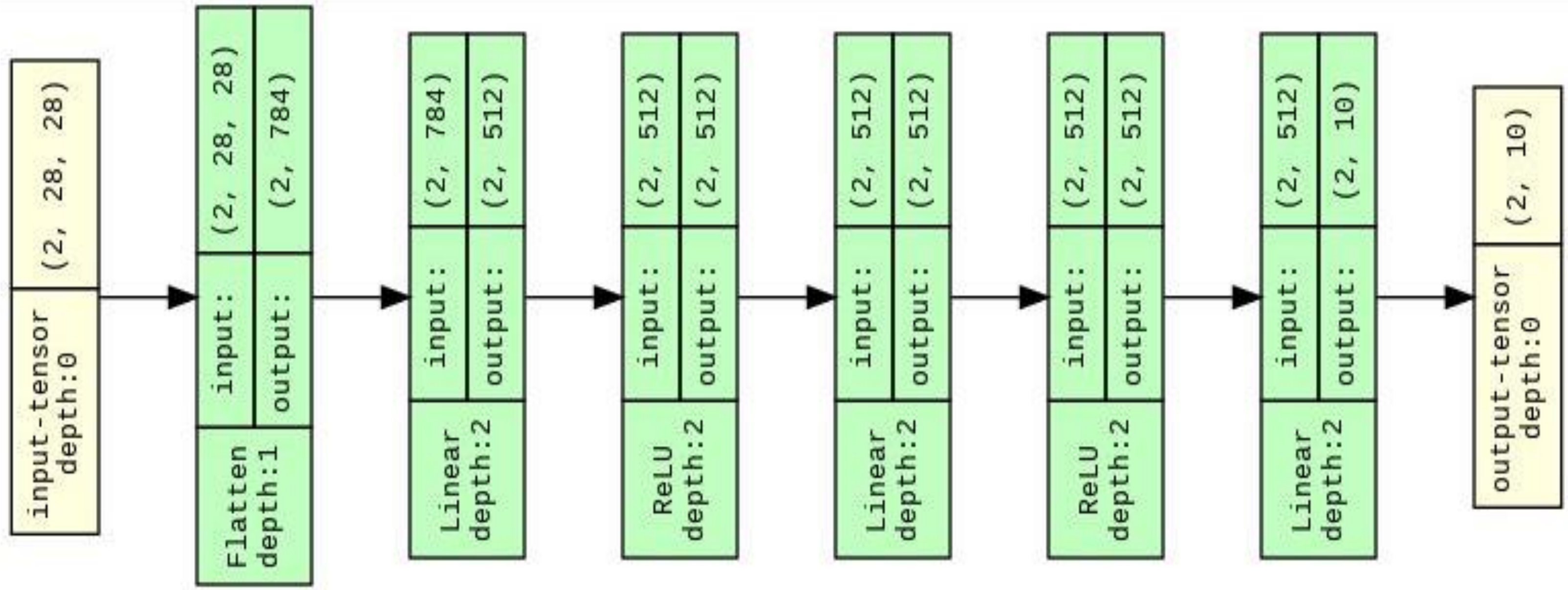
NETWORK DEFINITION

Network definition

- The base mechanism for defining a network architecture is **subclassing**
- Network components in PyTorch are a subclass of the `torch.nn.Module`
  - Base class for all neural network modules
  - Modules can also contain other Modules, allowing to nest them in a tree structure
  - Documentation available online: https://pytorch.org/docs/stable/nn.html#torch.nn.Module
- Every subclass must implement the `__init__` and `forward` methods

Example: classifier

- Let’s start with an example….



Network definition

```
class NeuralNetwork(nn.Module):
    def __init__(self, num_classes):
        super().__init__()
        self.flatten = nn.Flatten()

        self.lin1 = nn.Linear(28*28, 512)
        self.act1 = nn.ReLU()

        self.lin2 = nn.Linear(512, 512)
        self.act2 = nn.ReLU()

        self.output_layer = nn.Linear(512, 10)

    def forward(self, x):
        # (batch_size, 28, 28) => (batch_size, 28*28)
        x = self.flatten(x)

        # first layer (input is x, output is x1)
        x1 = self.lin1(x)
        x1 = self.act1(x1)

        # second layer (input is x1, output is x2)
        x2 = self.lin2(x1)
        x2 = self.act2(x2)

        # third/output layer (input is x2, output is logits)
        logits = self.output_layer(x2)
        return logits
```

Network definition

```
class NeuralNetwork(nn.Module):
    def __init__(self, num_classes):
        super().__init__()
        self.flatten = nn.Flatten()

        self.lin1 = nn.Linear(28*28, 512)
        self.act1 = nn.ReLU()

        self.lin2 = nn.Linear(512, 512)
        self.act2 = nn.ReLU()

        self.output_layer = nn.Linear(512, 10)

    def forward(self, x):
       # (batch_size, 28, 28) => (batch_size, 28*28)
        x = self.flatten(x)

        # first layer (input is x, output is x1)
        x1 = self.lin1(x)
        x1 = self.act1(x1)

        # second layer (input is x1, output is x2)
        x2 = self.lin2(x1)
        x2 = self.act2(x2)

        # third/output layer (input is x2, output is logits)
        logits = self.output_layer(x2)
        return logits
```

init function **instantiates** neural networks blocks (which layers?)

Network definition

```
class NeuralNetwork(nn.Module):
    def __init__(self, num_classes):
        super().__init__()
        self.flatten = nn.Flatten()

        self.lin1 = nn.Linear(28*28, 512)
        self.act1 = nn.ReLU()

        self.lin2 = nn.Linear(512, 512)
        self.act2 = nn.ReLU()

        self.output_layer = nn.Linear(512, 10)

    def forward(self, x):
       # (batch_size, 28, 28) => (batch_size, 28*28)
        x = self.flatten(x)

        # first layer (input is x, output is x1)
        x1 = self.lin1(x)
        x1 = self.act1(x1)

        # second layer (input is x1, output is x2)
        x2 = self.lin2(x1)
        x2 = self.act2(x2)

        # third/output layer (input is x2, output is logits)
        logits = self.output_layer(x2)
        return logits
```

- **Linear** layer computes linear transformation
  - $y = xW^T + b$
- **Activation** must be applied separately
- Many other layers defined… see https://pytorch.org/docs/stable/nn.html

Network definition

```
class NeuralNetwork(nn.Module):
    def __init__(self, num_classes):
        super().__init__()
        self.flatten = nn.Flatten()

        self.lin1 = nn.Linear(28*28, 512)
        self.act1 = nn.ReLU()

        self.lin2 = nn.Linear(512, 512)
        self.act2 = nn.ReLU()

        self.output_layer = nn.Linear(512, 10)

    def forward(self, x):
       # (batch_size, 28, 28) => (batch_size, 28*28)
        x = self.flatten(x)

        # first layer (input is x, output is x1)
        x1 = self.lin1(x)
        x1 = self.act1(x1)

        # second layer (input is x1, output is x2)
        x2 = self.lin2(x1)
        x2 = self.act2(x2)

        # third/output layer (input is x2, output is logits)
        logits = self.output_layer(x2)
        return logits
```

- **forward** function describes the flow of data through the network layers (which function is computed)?
- Input/output variables connect layers
- Notice how some tensors may be overwritten

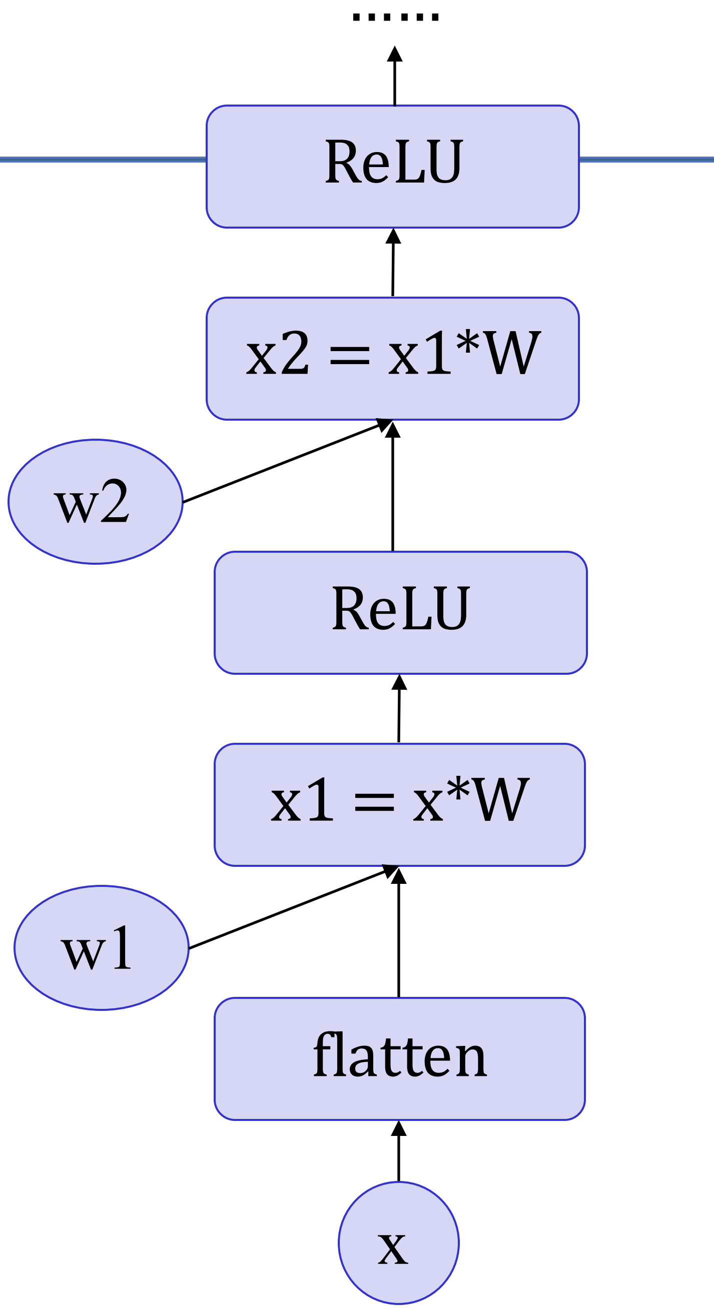
Network definition

```
class NeuralNetwork(nn.Module):
    def __init__(self, num_classes):
        super().__init__()
        self.flatten = nn.Flatten()

        self.lin1 = nn.Linear(28*28, 512)
        self.act1 = nn.ReLU()

        self.lin2 = nn.Linear(512, 512)
        self.act2 = nn.ReLU()

        self.output_layer = nn.Linear(512, 10)

    def forward(self, x):
        # (batch_size, 28, 28) => (batch_size, 28*28)
        x = self.flatten(x)

        # first layer (input is x, output is x1)
        x1 = self.lin1(x)
        x1 = self.act1(x1)

        # second layer (input is x1, output is x2)
        x2 = self.lin2(x1)
        x2 = self.act2(x2)

        # third/output layer (input is x2, output is logits)
        logits = self.output_layer(x2)
        return logits
```

Notice that the last layer does not include the softmax function!

Network definition

```
class NeuralNetwork(nn.Module):
    def __init__(self, num_classes):
        super().__init__()
        self.flatten = nn.Flatten()

        self.lin1 = nn.Linear(28*28, 512)
        self.act1 = nn.ReLU()

        self.lin2 = nn.Linear(512, 512)
        self.act2 = nn.ReLU()

        self.output_layer = nn.Linear(512, 10)

    def forward(self, x):
        # (batch_size, 28, 28) => (batch_size, 28*28)
        x = self.flatten(x)

        # first layer (input is x, output is x1)
        x1 = self.lin1(x)
        x1 = self.act1(x1)

        # second layer (input is x1, output is x2)
        x2 = self.lin2(x1)
        x2 = self.act2(x2)

        # third/output layer (input is x2, output is logits)
        logits = self.output_layer(x2)
        return logits
```



Concise vs. verbose definition

```
class NeuralNetwork(nn.Module):
    def __init__(self, num_classes):
        super().__init__()
        self.flatten = nn.Flatten()

        self.linear_relu_stack=nn.Sequential(
                nn.Linear(28*28, 512),
                nn.ReLU(),
                nn.Linear(512, 512),
                nn.ReLU(),
               nn.Linear(512, 10))

     def forward(self, x):
       # (batch_size, 28, 28) => (batch_size, 28*28)
        x = self.flatten(x)

        # first layer (input is x, output is x1)
        x1 = self.lin1(x)
        x1 = self.act1(x1)

        # second layer (input is x1, output is x2)
        x2 = self.lin2(x1)
        x2 = self.act2(x2)

        # third/output layer (input is x2, output is logits)
        logits = self.output_layer(x2)
        return logits
```

- Sequential module can be used to stack layers on top of each other
- Not all NN architectures can be represented by a stack of layers (more on that later…)

Concise vs. verbose definition

```
class NeuralNetwork(nn.Module):
    def __init__(self, num_classes):
        super().__init__()
        self.flatten = nn.Flatten()

        self.linear_relu_stack=nn.Sequential(
                nn.Linear(28*28, 512),
                nn.ReLU(),
                nn.Linear(512, 512),
                nn.ReLU(),
                 nn.Linear(512, 10))

    def forward(self, x):
        # (batch_size, 28, 28) => (batch_size, 28*28)
        x = self.flatten(x)

        logits = self.linear_relu_stack(x)
        return logits
```

- Compact, modular code
- Computational graph is the same

Forward and backward pass

- Instantiate the neural network object
  - `model = NeuralNetwork()`
- Move model to GPU or CPU
  - `model = model.cuda()`
  - `model = model.cpu()`
- Invoke on input data
  - `net_out = model(x)` calls `model.forward()`
  - network and data must be on the same device!
- Compute backward pass directly from tensor
  - `net_out.backward()`

```
RuntimeError: Expected all tensors to be on the same
device, but found at least two devices, cuda:0 and
cpu! (when checking argument for argument mat1 in
method wrapper_CUDA_addmm)
```

DATA MANAGEMENT

Data management

- Define Preprocessing
  - https://pytorch.org/docs/stable/torchvision/transforms.html
- Create a Dataset class
  - https://pytorch.org/docs/stable/data.html#torch.utils.data.Dataset
  - organize data, return one data sample at a time
- Instantiate a Dataloader
  - https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader
  - fetch batch of data from the dataset

Dataset

- Dataset stores the samples and their corresponding labels
- superclass is torch.utils.data.Dataset
- `__init__`
  - runs once when instantiating the Dataset object
  - initializes the directory containing the images, the annotations file, and both transforms (covered later)
- `__len__`
  - returns the total number of samples in the dataset
- `__getitem__`
  - takes an index as argument
  - loads and returns a sample from the dataset at the given index

DataLoader

- Wraps an iterable around the Dataset to enable easy access to the samples
  - Dataset retrieves our dataset's features and labels one sample at a time through
  - DataLoader efficiently reads from the dataset and returns batches of data
- Key parameters:
  - Batch size: number of samples returned at each iteration
  - Data shuffling

```
train_loader = torch.utils.data.DataLoader(
  torchvision.datasets.MNIST('/files/', train=True, download=True,
                             transform=torchvision.transforms.Compose([
                               torchvision.transforms.ToTensor(),
                               torchvision.transforms.Normalize(
                                 (0.1307,), (0.3081,))
                             ])),
  batch_size=batch_size_train, shuffle=True)
```

LOSS AND TRAINING LOOP

Loss

- torch.nn module has multiple standard loss functions
- Examples are
  - `torch.nn.CrossEntropyLoss` (classification task)
  - `torch.nn.MSELoss` (regression task)
- How to use:
  - `output = loss(input, target)`
  - `output.backward()` computes the gradients of the loss w.r.t. to the input
  - Note: CrossEntropyLoss takes as input the network logits since it encapsulates a softmax op. inside

Optimizer

- Updates network parameters depending on the gradients computed
- Optimizer constructor takes as inputs:
  - **Parameters:** a set of parameters to optimize (e.g. neural network weights)
  - **lr**: learning rate to use in the update rule
- Basic method is `step`
  - when invoked it updates the model parameters w.r.t computed gradients
- call `optimizer.step()` only after `loss.backward()`

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
optimizer = optim.Adam([var1, var2], lr=0.0001)
```

https://pytorch.org/docs/stable/optim.html

Putting everything together

```
def train(model, device, train_loader,
optimizer, epoch):
    model.train()  # model to train mode

    # ITERATE DATALOADER: train_loader
    for batch_idx, (data, target) in
enumerate(train_loader):

        #  SINGLE OPTIMIZATION STEP IS
PERFORMED ON A BATCH!
        data, target = data.to(device),
target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.cross_entropy(output,
target)
        loss.backward()
        optimizer.step()
```

Let's see how a typical training loop looks like…

This function traines the model for one epoch

Putting everything together

```
def train(model, device, train_loader,
optimizer):
    model.train()  # model to train mode

    # ITERATE DATALOADER: train_loader
    for batch_idx, (data, target) in
enumerate(train_loader):

        #  SINGLE OPTIMIZATION STEP IS
PERFORMED ON A BATCH!
        data, target = data.to(device),
target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.cross_entropy(output,
target)
        loss.backward()
        optimizer.step()
```

Inputs:

- Model (defined and instantiated as seen before)
- Device (CPU/GPU)
- DataLoader
- Optimizer

Putting everything together

```
def train(model, device, train_loader,
optimizer, epoch):
    model.train()

    # ITERATE DATALOADER: train_loader
    for batch_idx, (data, target) in
enumerate(train_loader):

        #  SINGLE OPTIMIZATION STEP IS
PERFORMED ON A BATCH!
        data, target = data.to(device),
target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.cross_entropy(output,
target)
        loss.backward()
        optimizer.step()
```

Set model to train mode

Putting everything together

```
def train(model, device, train_loader,
optimizer, epoch):
    model.train()  # model to train mode

    # ITERATE DATALOADER: train_loader
    for batch_idx, (data, target) in
enumerate(train_loader):

        #  SINGLE OPTIMIZATION STEP IS
PERFORMED ON A BATCH!
        data, target = data.to(device),
target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.cross_entropy(output,
target)
        loss.backward()
        optimizer.step()
```

- Iterate once over the entire training set
- (data, target) are (x, y) samples

Putting everything together

```
def train(model, device, train_loader,
optimizer, epoch):
    model.train()  # model to train mode

    # ITERATE DATALOADER: train_loader
    for batch_idx, (data, target) in
enumerate(train_loader):

        #  SINGLE OPTIMIZATION STEP IS
PERFORMED ON A BATCH!
        data, target = data.to(device),
target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.cross_entropy(output,
target)
        loss.backward()
        optimizer.step()
```

- Move data to the device (assuming the network weights are stored on the same device)

Putting everything together

```
def train(model, device, train_loader,
optimizer, epoch):
    model.train()  # model to train mode

    # ITERATE DATALOADER: train_loader
    for batch_idx, (data, target) in
enumerate(train_loader):

        #  SINGLE OPTIMIZATION STEP IS
PERFORMED ON A BATCH!
        data, target = data.to(device),
target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.cross_entropy(output,
target)
        loss.backward()
        optimizer.step()
```

- Clear gradients from previous iterations

Putting everything together

```
def train(model, device, train_loader,
optimizer, epoch):
    model.train()  # model to train mode

    # ITERATE DATALOADER: train_loader
    for batch_idx, (data, target) in
enumerate(train_loader):

        #  SINGLE OPTIMIZATION STEP IS
PERFORMED ON A BATCH!
        data, target = data.to(device),
target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.cross_entropy(output,
target)
        loss.backward()
        optimizer.step()
```

- Compute forward pass

Putting everything together

```
def train(model, device, train_loader,
optimizer, epoch):
    model.train()  # model to train mode

    # ITERATE DATALOADER: train_loader
    for batch_idx, (data, target) in
enumerate(train_loader):

        #  SINGLE OPTIMIZATION STEP IS
PERFORMED ON A BATCH!
        data, target = data.to(device),
target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.cross_entropy(output,
target)
        loss.backward()
        optimizer.step()
```

- Compute loss

Putting everything together

```
def train(model, device, train_loader,
optimizer, epoch):
    model.train()  # model to train mode

    # ITERATE DATALOADER: train_loader
    for batch_idx, (data, target) in
enumerate(train_loader):

        #  SINGLE OPTIMIZATION STEP IS
PERFORMED ON A BATCH!
        data, target = data.to(device),
target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.cross_entropy(output,
target)
        loss.backward()
        optimizer.step()
```

- Compute backward pass
- Pytorch Autograd automatically computes the gradients

Putting everything together

```
def train(model, device, train_loader,
optimizer, epoch):
    model.train()  # model to train mode

    # ITERATE DATALOADER: train_loader
    for batch_idx, (data, target) in
enumerate(train_loader):

        #  SINGLE OPTIMIZATION STEP IS
PERFORMED ON A BATCH!
        data, target = data.to(device),
target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.cross_entropy(output,
target)
        loss.backward()
        optimizer.step()
```

- Update weights