A Comparative Introduction to Deep Learning Frameworks: TensorFlow, PyTorch and JAX

Introduction to PyTorch

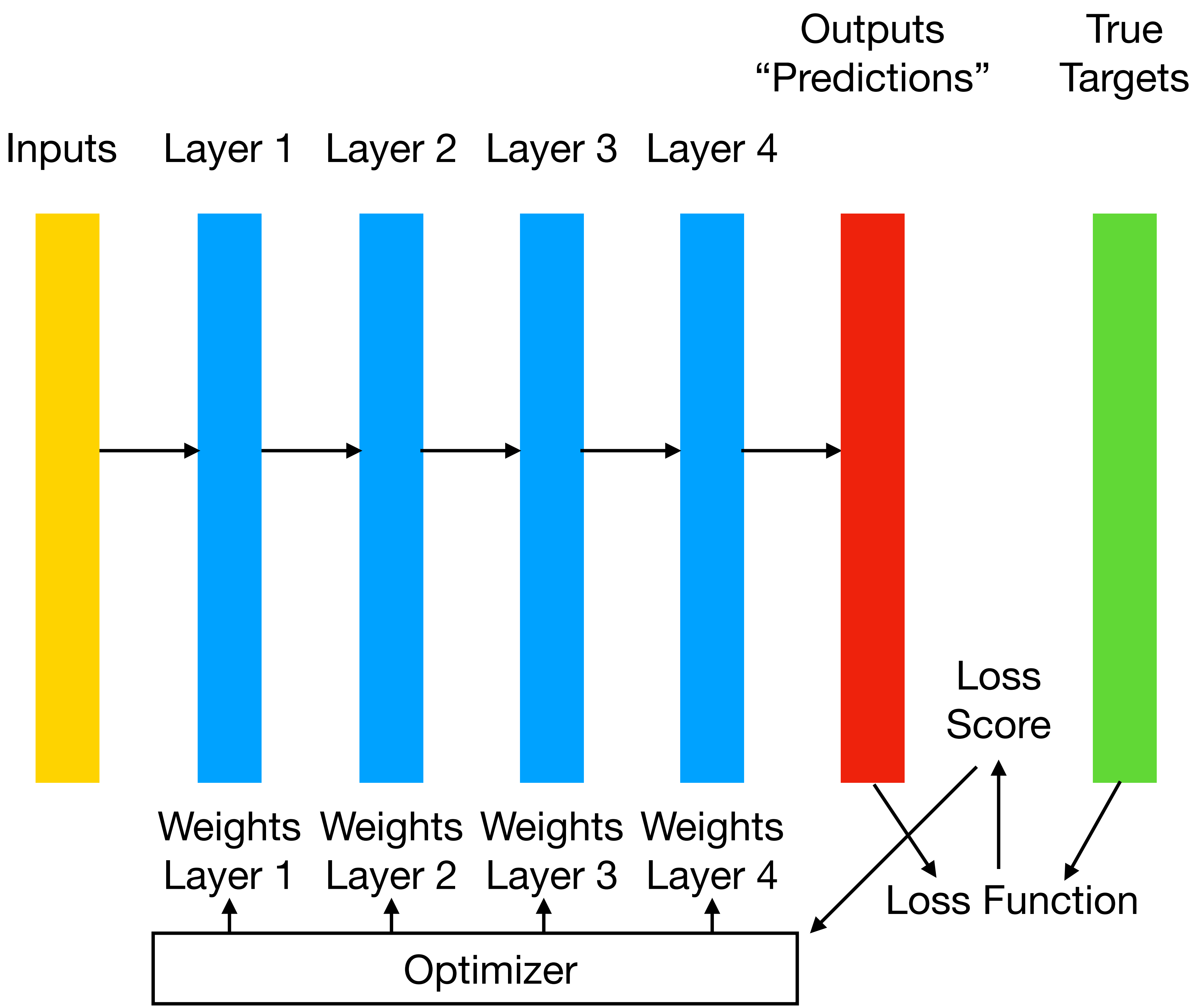
Mirco Musolesi

mircomusolesi@acm.org

PyTorch

- PyTorch is a deep learning framework based on the Torch library.
  - Torch is a machine learning library, which uses the scripting language Lua (on the LuaJIT - Lua Just In Time runtime for the Lua language). Lua itself is implemented in C.
- Current stable release is 1.13.0.
- Originally developed by Meta AI, now part of the Linux Foundation.

Deep Neural Networks



PyTorch Tensors

- Data structures are key aspects of deep learning frameworks.
- Python arrays are inefficient.
  - Vanilla Python arrays are stored in non-continuous memory.
- We use instead specialised data structures, i.e., PyTorch tensors.
  - As NumPy arrays, they are based on contiguous memory cells.

PyTorch Tensors

- Example (a vector of ones):

```
>>> import torch

>>> a = torch.ones (3)

>>> a

tensor([1., 1., 1.])

>>> a[1]

tensor(1.)

>>> float(a[1])

1.0
```

PyTorch Tensors

- We can modify the values of the elements of a Tensor as follows (given the example in the previous slide):

```
>>> a[2] = 2.0

>>> a

tensor([1., 1., 2.])
```

Initialisation of Tensors

- More in general, you can initialise the tensors in different ways.

- One way is directly from data:

```
>>> data = [[10,15],[23, 42]]

>>> x_data = torch.tensor(data)

>>> x_data

tensor([[10, 15],

        [23, 42]])
```

Initialisation of Tensors

- Another way is to create them from NumPy arrays:

```
>>> np_array = np.array(data)

>>> x_np = torch.from_numpy(np_array)

>>> x_np

tensor([[10, 15],

        [23, 42]])
```

- You can also initialise them from existing tensors as well (see documentation).

Initialisation of Tensors

- Or you can initialise them with ones (`torch.ones()`) or zeros (`torch.zeros()`) or random values ((`torch.rand()`).

- For example:

```
>>> import torch

>>> shape = (2, 3, )

>>> random_tensor = torch.rand(shape)

>>> random_tensor

tensor([[0.0685, 0.3877, 0.0179],

        [0.1773, 0.6916, 0.4333]])
```

Tensor Size and Shape

- The shape of a tensor is given by `torch.tensor.size(dim = None)`.
- This method returns the size of the tensor itself.
- If the dimension dim is not specified, the returned value is an object of class `torch.size`, which is a subclass of standard Python tuple.
- Example:

```
>>> t.size()

torch.Size([4, 5, 10])

>>> t.size(dim=1)

5
```

Dataloaders

- Pytorch provides two primitives:
  - `torch.utils.data.DataLoader`
  - `torch.utils.data.Dataset`
- These primitives allow to use pre-loaded datasets as well as user-defined data.
- `Dataset` stores samples and the corresponding labels.
- `DataLoader` is a wrapper of an iterable around `Dataset`.
- PyTorch provides a number of pre-loaded datasets that subclass `torch.utils.data.Dataset` and implements functions that are specific to that dataset.

Dataset Loading

```
import torch
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt


training_data = datasets.MNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.MNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)
```

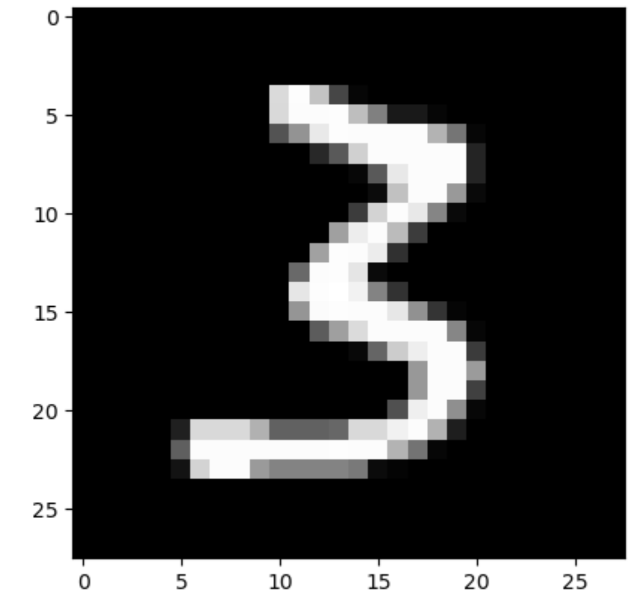
Dataset Loading

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64,
shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64,
shuffle=True)

# Display image and label.
train_features, train_labels = next(iter(train_dataloader))
print(f"Feature batch shape: {train_features.size()}")
print(f"Labels batch shape: {train_labels.size()}")
img = train_features[0].squeeze()
label = train_labels[0]
plt.imshow(img, cmap="gray")
plt.show()
print(f"Label: {label}")
```

Dataset Loading

- This would be the output of the previous slide:

```
Feature batch shape: torch.Size([64, 1, 28, 28])

Labels batch shape: torch.Size([64])
```



The MNIST Dataset in PyTorch

- We consider the MNIST Dataset with the following parameters:
  - `root` is the path where the train/test data is stored.
  - `train` specifies if it is a training or a test dataset.
  - `download=True` downloads the data if it is not available at root.
  - `transform` and `target_transform` specify the feature and the label transformations, which we might want to apply to the data.

Dataset Loading

```
import torch
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda

ds = datasets.MNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
    target_transform=Lambda(lambda y: torch.zeros(10,
dtype=torch.float).scatter_(0, torch.tensor(y),
value=1))
)
```

DataLoaders and Batches

- `Dataset` retrieves the features of the dataset and the labels one sample at a time.
- Instead, we are typically interested in passing mini batches, and possibly reshuffle the data at every epoch to reduce overfitting.
- `DataLoader` is an iterable that abstracts this complexity.
- It also hides the complexity related to the use of `multiprocessing`.
- Example:

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

Definition of the Neural Network

- In PyTorch, a neural network is defined by subclassing `nn.Module`.
- There are two fundamental methods that we need to overwrite:
  - `__init__()`: it is used to initialise the neural network.
  - `forward()`: it is used to implement the "forward pass" of the network.

Creating the Neural Network

- The `torch.nn` namespace provides all the building blocks for building the network.
- Every module in PyTorch subclasses the class `nn.Module`.
- A neural network is a module itself that consists of other modules (our layers).

PyTorch Devices

- We want to train the model on a GPU if available.

- By default, the tensors are generated and managed on the CPU. The model itself is initialised on the CPU. It is necessary to explicitly set the device to GPU.

- It is possible to check if a device is present, for example we can check if CUDA support is available through:

```
torch.cuda.is_available()
```

Devices

```
import os
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms


device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using {device} device”)
```

- The output on a device without a GPU will be:

```
Using cpu device
```

Definition of the Neural Network

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__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):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

Definition of the Neural Network

```
model = NeuralNetwork().to(device)
print(model)
```

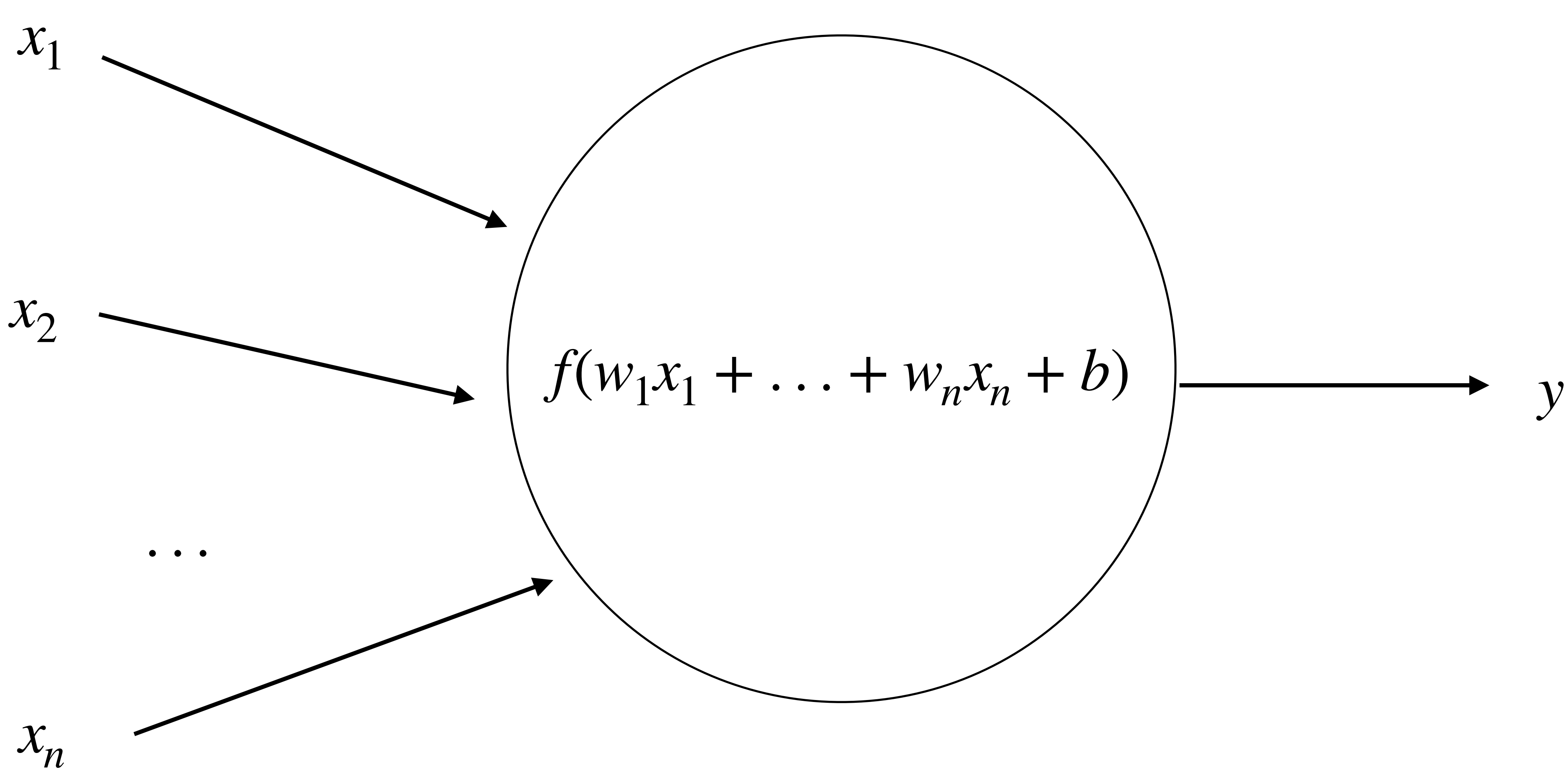
- The output will be:

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

PyTorch Layers

- Let us consider the different modules/layers in detail:
  - `nn.Sequential` is an ordered container of modules. The data is passed through the modules in the same order as they are defined.
  - `nn.flatten` converts each 2D image (28x28 pixels) into an array of 784 pixels.
  - `nn.linear` applies a linear transformation on the input using the stored weights and biases, literally $y_i = \sum_j (w_{j,i} + b_i)$.
  - `nn.relu` is a non-linear activation (it introduces the non-linearity necessary for guaranteeing universal approximation). Please note the separation of the linear and non linear component “per layer”. Also compare with the TensorFlow typical design pattern.

Nodes/Units/Neurons



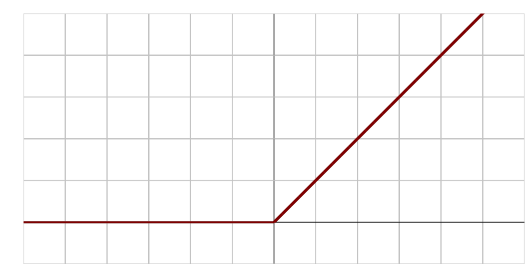
f is called the activation function, b is usually called the bias

Activations Functions

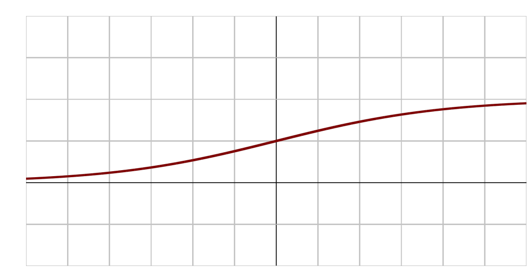
- They are generally used to add non-linearity.

- Examples:

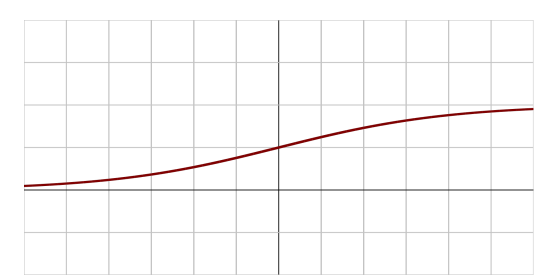
  - *Rectified Linear Unit*: it returns the max between 0 and the value in input. In other words, given the value z in input it returns $max(0,z)$.



  - *Logistic sigmoid*: given the value in input z, it returns $\dfrac{1}{1+e^{z}}$.

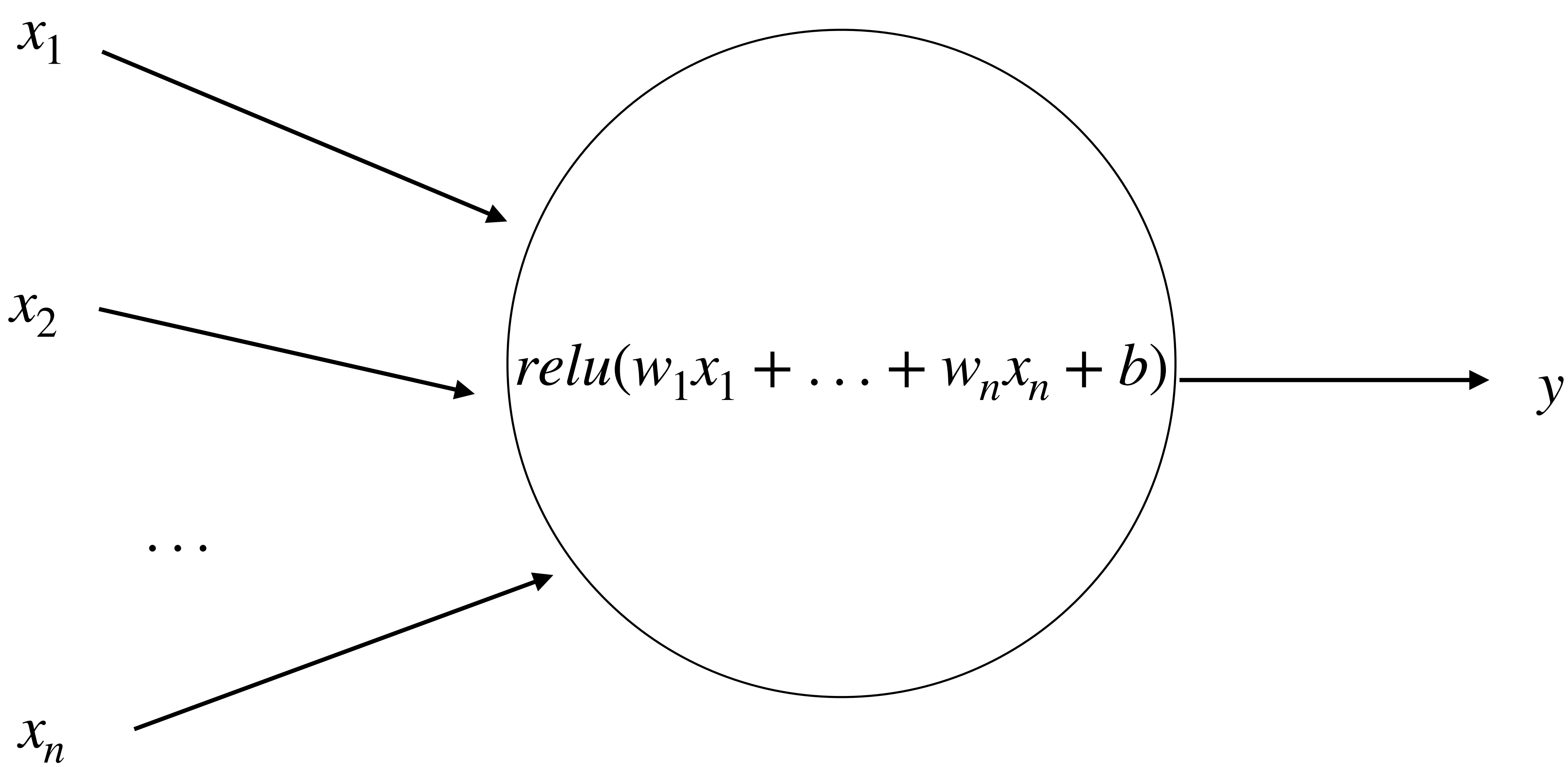


  - *Arctan*: given the value in input z, it returns $tan^{-1}(z)$.



Credit: Wikimedia

Nodes/Units/Neurons



Note that here the function in input of relu is 1-dimensional.

Softmax Layer

- The output of the last linear layer of the neural network returns logits.
- The logits are passed to a `nn.Softmax` module.
- Logits are raw values in the [-infinity, +infinity] interval.
- The lights are scaled to values in the [0,1] representing the model's predicted probabilities (calibration might be necessary).

Softmax Layer

- Please note that softmax is not like the activation functions that we discussed before. The activations functions that we discussed before take in input real numbers and returns a real number.

- A softmax function receives in input a vector of real numbers of dimension n and returns a vector of real numbers of dimension n.

- Given a vector of real numbers in input $\mathbf{z}$ of dimension n, a softmax function normalises it into a probability distribution consisting of n probabilities proportional to the exponentials of each element z_i of the vector $\mathbf{z}$. More formally, $softmax(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$ for $i = 1,..n$.

Using the Model

- In order to use the model, we pass the input data.
- This executes the model's `forward()`.
  - `model.forward()` does not have to be call directly.
- In our case, the model returns a 2-dimensional tensor with `dim=0` corresponding to each output of 10 raw predicted values for each class and `dim=1` corresponds to the individual values of each output.
- We obtain the prediction probabilities by passing it through an instance of the `nn.Softmax` module.

Model Parameters

- As we saw, layers are usually parametrised, i.e., they have weights and biases that are optimised during training.

- It is possible to access the models using `parameters()` and `named_parameters()`.

Model Parameters

```
print(f"Model structure: {model}\n\n")

for name, param in model.named_parameters():
    print(f"Layer: {name} | Size: {param.size()} |
Values : {param[:2]} \n")
```

Model Parameters

- The output will be:

```
(flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)


Layer: linear_relu_stack.0.weight | Size: torch.Size([512, 784]) | Values : tensor([[-0.0326,  0.0231, -0.0234,  ..., -0.0043,
-0.0072,  0.0234],
        [-0.0068,  0.0255,  0.0012,  ..., -0.0176,  0.0071,  0.0073]],
       grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.0.bias | Size: torch.Size([512]) | Values : tensor([0.0126, 0.0055], grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.2.weight | Size: torch.Size([512, 512]) | Values : tensor([[ 4.2491e-02, -2.7992e-02,
-3.4629e-02,  ...,  2.7492e-02,
          4.2681e-02,  2.0021e-03],
        [-3.5574e-03, -4.6985e-05, -3.4182e-02,  ..., -3.8956e-02,
          3.4745e-02, -1.6162e-03]], grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.2.bias | Size: torch.Size([512]) | Values : tensor([0.0132, 0.0310], grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.4.weight | Size: torch.Size([10, 512]) | Values : tensor([[ 0.0180, -0.0450, -0.0341,  ...,  0.0254,
0.0009,  0.1083],
        [-0.0237,  0.0091,  0.0851,  ...,  0.0052,  0.1011, -0.0933]],
       grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.4.bias | Size: torch.Size([10]) | Values : tensor([-0.1172,  0.1277], grad_fn=<SliceBackward0>)
```

Using the Model

```
X = torch.rand(1, 28, 28, device=device)
logits = model(X)
pred_probab = nn.Softmax(dim=1)(logits)
y_pred = pred_probab.argmax(1)
print(f"Predicted class: {y_pred}”)
```

- The output will be:

```
Predicted class: tensor([1])
```

Parameters and Optimisers

```
learning_rate = 1e-3
batch_size = 64
epochs = 5
```

```
# Initialize the loss function
loss_fn = nn.CrossEntropyLoss()
```

```
optimizer = torch.optim.SGD(model.parameters(),
lr=learning_rate)
```

Training and Validation/Test Loop

- Once the hyper parameter are set, we can train and optimise with an optimisation loop.
- Each iteration of the optimisation loop is called an epoch.
- We have two parts:
  - The *train loop*, which iterates over the training dataset and converge to optimal parameter.
  - The *validation/test loop*, which iterates over the test dataset to check if the the model performance is improving.

Training and Validation/Test Loop

- Inside the training loop, optimisation happens in three steps:
  - The call of `optimizer.zero_grad()`, which is used to reset the gradients of model parameters.Gradients by default add up. We must explicitly zero them at each iteration.
  - The call of `loss.backward()`, which back-propagates the prediction loss. PyTorch deposits the gradient loss with respect to each parameter. This call is explicit!
  - The call of `optimizer.step()`, which adjusts the parameters by the gradients collected in the backward pass above.

Train Loop

```
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/
{size:>5d}]")
```

Test Loop

```
def test_loop(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) ==
y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%,
Avg loss: {test_loss:>8f} \n")
```

Execution of Train Loop and Test Loop

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
lr=learning_rate)

epochs = 10
for t in range(epochs):
    print(f"Epoch
{t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn,
optimizer)
    test_loop(test_dataloader, model, loss_fn)
```

References

- Eli Stevens, Luca Antinga, Thomas Viehmann. Deep Learning with PyTorch. Manning. 2020.

- PyTorch Documentation.

- Some of the material in these slides has been taken from the official PyTorch documentation.