1. Introduction to Deep Learning with PyTorch

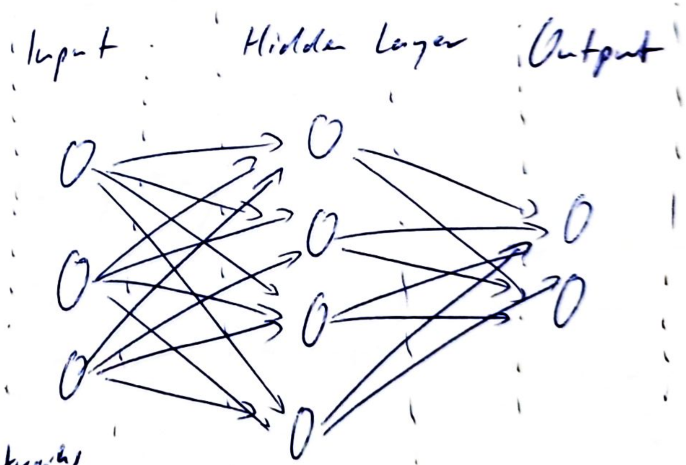
1.1. Introduction to PyTorch, a Deep Learning LIBRARY

What is deep learning?

- Deep learning is everywhere
  - Language translation
  - Self-driving cars
  - Medical diagnostics
  - Chatbots

- Used on multiple data types: images, text and audio (unstructured data)
- Traditional ML: Relies on hand-crafted feature engineering
- Deep learning: Enables feature learning from raw data

$\boxed{\text{DEEP LEARNING IS A SUBSET OF MACHINE LEARNING}}$



- The fundamental model structure is a network of inputs, hidden layers and outputs (figure right)
- A network can also have more than one hidden layer!
- The original intuition behind deep learning was to create models inspired by how the <u>human brain</u> learns → <u>Neural Networks</u>
- Models require large amount of data!

PyTorch : A deep learning framework

- one of the most popular deep learning frameworks
- intuitive and user friendly
- has much in common with NumPy

PyTorch is in Python

```
import torch
```

it supports

- image data with torchvision
- audio data with torchaudio
- text data with torchtext

The fundamental data structure in PyTorch is called a **tensor**

Build a tensor from a List	Build a tensor from a NumPy Array
`list = [[1,2,3], [4,5,6]]`	`np_array = np.array(array)`
`tensor = torch.tensor(list)`	`np_tensor = torch.from_numpy(np_array)`

tensors are <u>multidimensional representations</u> of their elements!

Scalar → Vector → Matrix → Tensor

$[1]$ $\quad \begin{bmatrix} 1 \\ 2 \end{bmatrix}$ $\quad \begin{bmatrix} 1 & 5 \\ 2 & 6 \end{bmatrix}$ $\quad \left[\begin{bmatrix} 1 & 5 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} 1 & 5 \\ 2 & 6 \end{bmatrix} \right]$

Tensor attributes:

- Tensor shape

  list = [[1,2,3], [4,5,6]]

  tensor = torch.tensor(list)

  tensor.shape

  >> torch.Size([2,3])

- Tensor data type

  tensor.dtype

  >> torch.int64

- Tensor device

  tensor.device

  >> device(type='cpu')

! Deep learning often requires a GPU, it can offer:

- parallel computing
- better performance
- faster training times!

Tensor operations:

a = torch.tensor([[1,1],[2,2]])

b = torch.tensor([[2,2], [3,3]])

Addition/subtraction

a+b

>> tensor([[3,3],[5,5]])

ERROR: For incompatible shapes!

Element-wise multiplication:

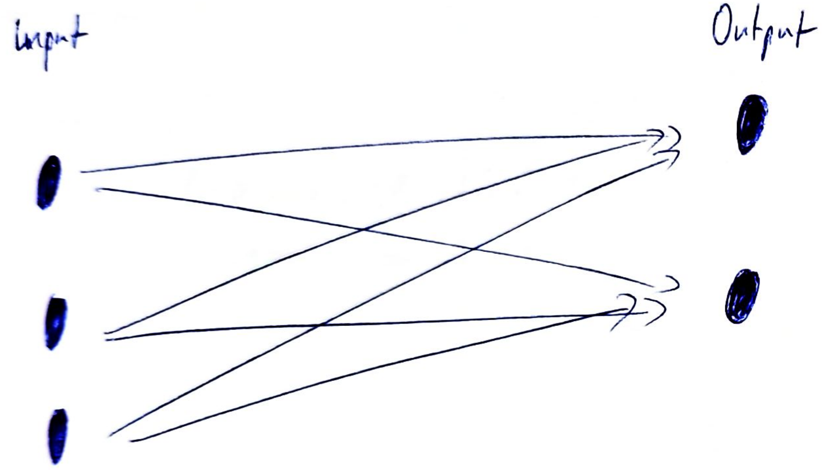
a * b

>> tensor([[2,2],[6,6]])

- ... and much more
  - Transposition
  - Matrix multiplication
  - Concatenation

Creating our first neural network:

```
import torch.nn as nn
```



(here: no hidden layer!)

```
# Create input_tensor with 3 features
input_tensor = torch.tensor([[0.3471,
                              0.4547,
                              -0.2356]])
```

```
# Define linear layer
linear_layer = nn.layer(in_features=3,
                        out_features=2)
```

(applies a linear function to the input)

```
# Pass input trough linear layer
output = linear_layer(input_tensor)
print(output)
```

Output: `tensor([[-0.2415, -0.1604]], grad_fn=<AddmmBackward0>)`

Getting to know linear operations:

Each linear layer has a weight and a bias:

```
linear_layer.weight
>> tensor([[-0.4799, 0.4996, 0.1123],
           [-0.0365, -0.1855, 0.0432]],
           requires_grad=True)
```

```
linear_layer.bias
>> tensor([0.0310, 0.1537],
           requires_grad=True)
```

For input X, weights W0 and a bias b0, the linear layer performs

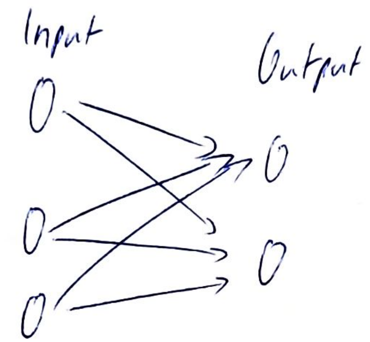
$$\boxed{y_0 = W_0 \cdot X + b_0}$$

In PyTorch: output = W0 @ input + b_0

- Initially, when we call nn.Linear(), weights and biases are initialized randomly, so they are not yet useful
- By tuning these parameters, our linear operation output is meaningful

Our two-layer network summary:

- Input dimensions : 1x3
- linear layer arguments :
  - in_features = 3
  - out_features = 2
- output dimensions : 1x2



- Networks with only linear layers are called „fully connected"

Stacking layers with nn.Sequential()

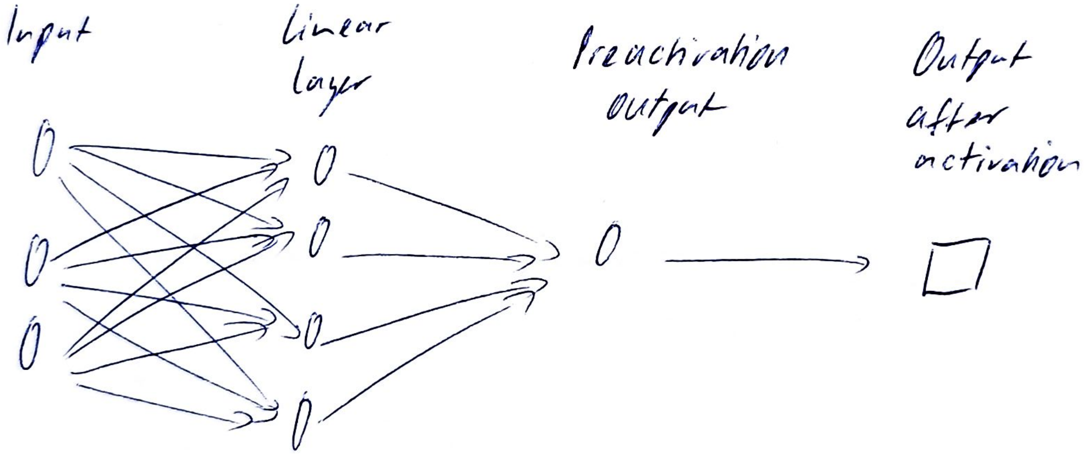
```
#Three linear layers
model = nn.Sequential(
    nn.Linear(10, 18),    → Input with 10 features and outputs a tensor with 18 features
    nn.Linear(18, 20),    → takes an input of size 18 and output is a tensor of size 20
    nn.Linear(20, 5)      ...
)
```

Activation functions:

- Now we add non-linearity to our models using activation functions
- Even with multiple stacked linear layers, output still has linear relationship with input

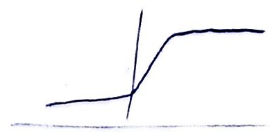
Why do we need activation functions?

- A model can learn more complex relationships with non-linearity
- The output will no longer be a linear function of the input



Sigmoid Activation function

$$\mathrm{sig}(x) = \sigma(x) = \frac{1}{1+\exp(-x)}$$

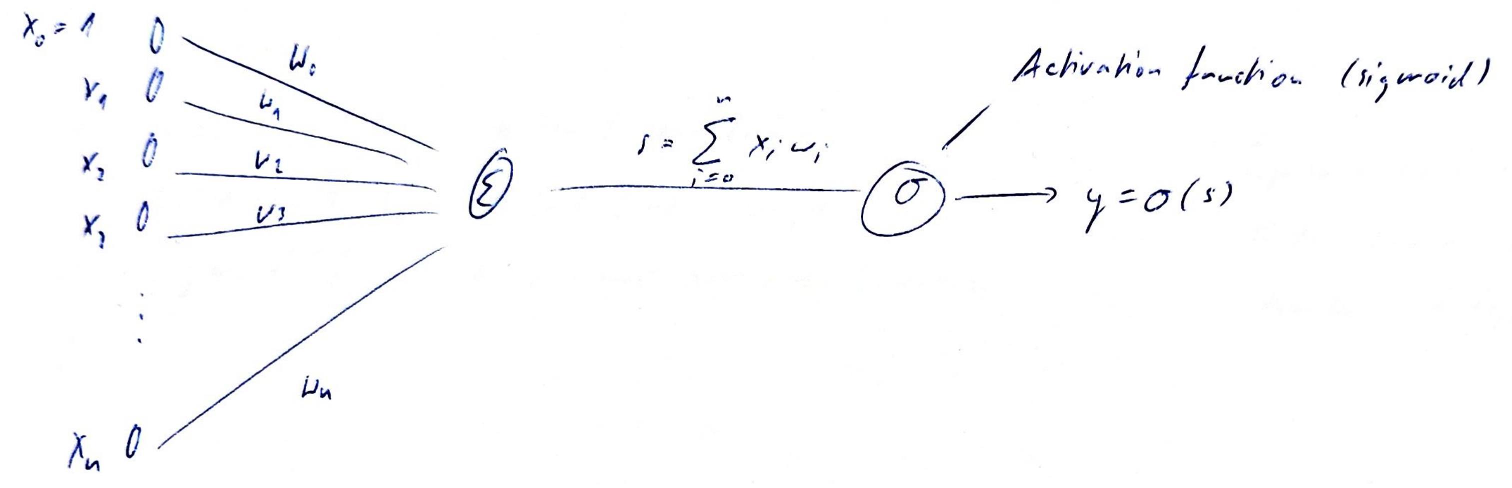


! The function is differentiable everywhere !

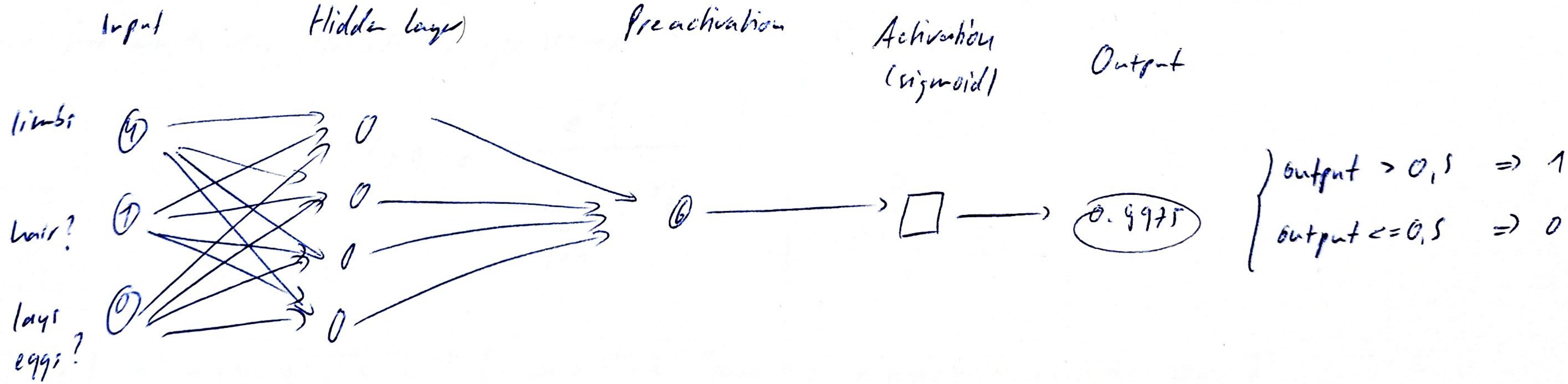
Domain : $(-\infty, +\infty)$ $\quad$ $\sigma(0) = 0{,}5$

Range : $(0, +1)$ $\quad$ $\sigma'(x) = \sigma(x)\,(1-\sigma(x))$

In general:



Binary classification task:



→ Predict whether animal is 1 (mammal) or 0 (not mammal)

We take the pre-activation (6), pass it to to sigmoid, and obtain a value between 0 and 1.

Activation function as the last layer:

```
model = nn.Sequential(
    nn.Linear(6,4),     # 1st linear layer
    nn.Linear(4,1),     # 2nd linear layer
    nn.Sigmoid()        # Sigmoid activation function
)
```

! Sigmoid as last step in network of linear layers is equivalent to traditional logistic regression !

Multiclass classification - softmax()

- takes N-element vector as input and outputs vector of same size [nn.Softmax()]
- used for multi-class classification-problems

$$S(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

- output is a probability distribution: Each element is a probability (between 0 & 1)
- for example $N = 3$ classes: bird = 0, mammal = 1, reptile = 2