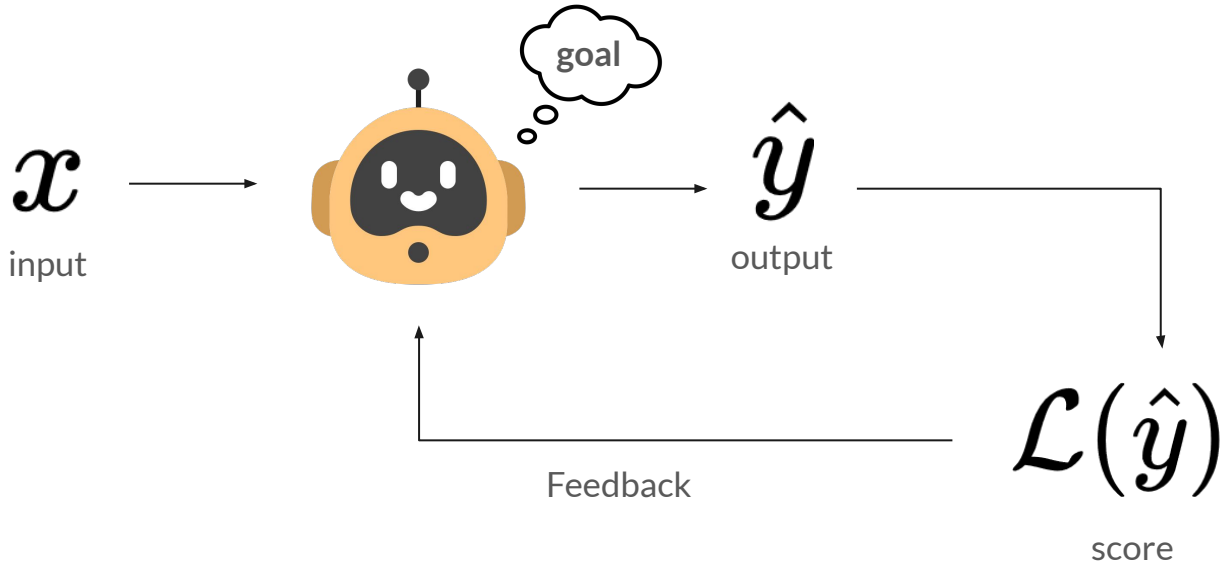
Introductory Seminar of PyTorch for Deep Learning

Daniele Angioni, Cagliari Digital Lab 2024 - Day 2

Machine Learning Foundations

Machine Learning Pipeline



Pattern Classification

Goal: assign a label to a pattern

A pattern is a description of an object through a set of measurement called **features**

Example

Seagull or Flamingo?

How can ML recognize the difference?

Getting the features

Let’s use a single feature for now, for example the **length of the legs**.

We call this quantity x

In general, flamingo’s legs are longer than the seagulls's.
Then, we can set a decision rule, for example:

Getting the features

Let's use a single feature for now, for example the **length of the legs**.

We call this quantity x

In general, flamingo's legs are longer than the seagulls's.
Then, we can set a decision rule, for example:

if $x > x^*$, it's a flamingo

if $x <= x^*$, it's a seagull

But how can we estimate x^*?

Training Dataset

We need a set of **labeled** examples to compute statistics on the two classes.

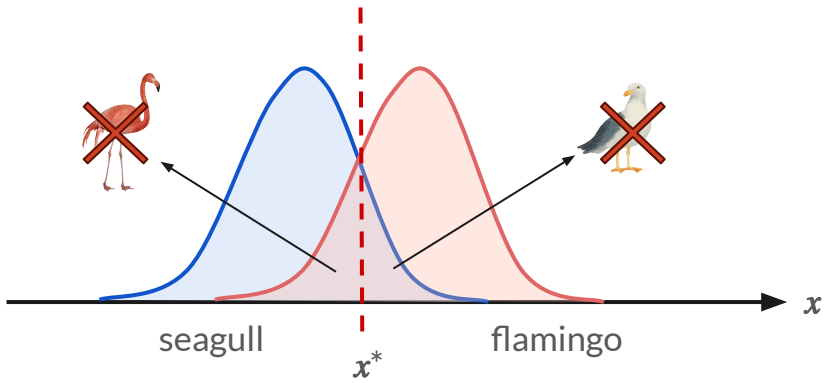
For example, we can estimate the average length of legs of all the flamingos and all the seagulls

$$D = \{(x_1, y_1), (x2, y_2), \ldots, (x_n, y_n)]\}$$

Training Dataset

However, a single feature is not sufficient to properly classify these two animals.

How can we improve the performances?

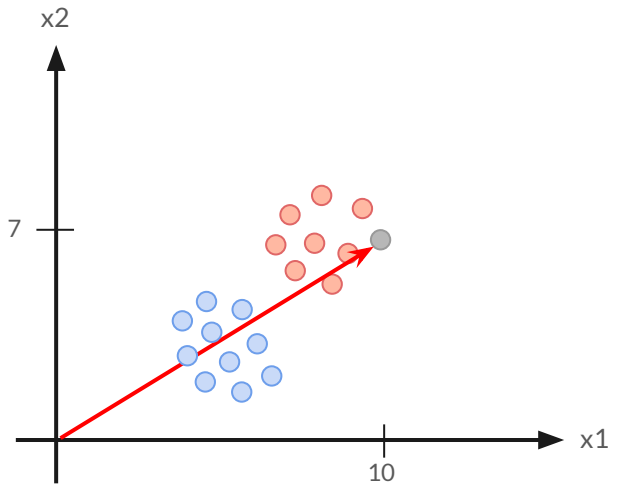


Training Dataset

We can use an additional feature. Let's use the **wingspan**. Now we can represent each subject with two values:

$$\boldsymbol{x_i} = [x_{i,1}, x_{i,2}]$$

$$\boldsymbol{x_i} = [10, 7]$$



Exercise 2.1

Exercise

Let's create a dataset like this ourself with PyTorch.

Imagine that seagulls and flamingos features (*lenght of the legs*, *wingspan*) are sampled from a gaussian distribution $N(\mu,\sigma)$, where $\mu_{flamingo} = [8, 7]^T$, $\mu_{seagull} = [5, 3]^T$ and $\sigma_{flamingo} = \sigma_{seagull} = 1.5$.

Now we can sample, say 10 seagull and 10 flamingo and we can visualize them in a 2-dimensional space.

Exercise

[39]
```
n_samples_per_class = 10
n_features = 2
std = 1.5

torch.manual_seed(0)    # needed for reproducibility
mean_flamingos = torch.tensor([8, 7])
mean_seagulls = torch.tensor([5, 3])

# randn is the normal distribution with zero mean and std=1
# from it we first scale the samples based on std and we add the mean of each class
X_flamingos = torch.randn(size=(n_samples_per_class, n_features)) * std + mean_flamingos
X_seagulls = torch.randn(size=(n_samples_per_class, n_features)) * std + mean_seagulls

# we encode flamingos as the label y=0, while seagulls with label y=1
X = torch.cat([X_flamingos, X_seagulls])
y = torch.cat([torch.zeros(size=(n_samples_per_class,)),
               torch.ones(size=(n_samples_per_class,))])
```

Exercise

```
[43]  1  print(X)
      2  print(X.shape)
```

```
tensor([[ 6.3112,  5.2715],
        [ 7.6241,  6.3492],
        [ 8.8983,  4.6674],
        [ 7.4880,  9.7795],
        [ 8.7021,  6.7634],
        [10.1655,  7.3991],
        [10.0840,  9.3795],
        [ 9.4194,  5.7345],
        [ 9.3977,  8.8885],
        [11.0075,  7.0806],
        [ 5.6595,  3.1686],
        [ 5.9612,  3.6617],
        [ 5.3083,  2.3245],
        [ 4.1404,  2.1670],
        [ 5.8915,  5.3129],
        [ 5.7610,  2.1135],
        [ 3.0120,  3.2828],
        [ 4.8964,  2.2576],
        [ 2.7561,  2.7092],
        [ 5.6683,  4.9879]])
torch.Size([20, 2])
```

```
[44]  1  print(y)
      2  print(y.shape)
```

```
tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1.])
torch.Size([20])
```

Exercise

```
[45]  1  print('### X_flamingos ###\n', X[y == 0, :])
      2  print(X[y == 0, :].shape)
```

```
### X_flamingos ###
 tensor([[ 6.3112,  5.2715],
         [ 7.6241,  6.3492],
         [ 8.8983,  4.6674],
         [ 7.4880,  9.7795],
         [ 8.7021,  6.7634],
         [10.1655,  7.3991],
         [10.0840,  9.3795],
         [ 9.4194,  5.7345],
         [ 9.3977,  8.8885],
         [11.0075,  7.0806]])
torch.Size([10, 2])
```

```
[46]  1  print('### X_seagulls ###\n', X[y == 1, :])
      2  print(X[y == 1, :].shape)
```

```
### X_seagulls ###
 tensor([[5.6595, 3.1686],
         [5.9612, 3.6617],
         [5.3083, 2.3245],
         [4.1404, 2.1670],
         [5.8915, 5.3129],
         [5.7610, 2.1135],
         [3.0120, 3.2828],
         [4.8964, 2.2576],
         [2.7561, 2.7092],
         [5.6683, 4.9879]])
torch.Size([10, 2])
```

Exercise

```
fig, ax = plt.subplots()
ax.scatter(X[y == 0, :][:, 0],
           X[y == 0, :][:, 1],
           label='flamingo',
           color='red')
ax.scatter(X[y == 1, :][:, 0],
           X[y == 1, :][:, 1],
           label='seagull',
           color='blue')
ax.set_xlabel('lenght of the legs')
ax.set_ylabel('wingspan')
ax.legend()
fig.show()
```

```
### X_flamingos ###
 tensor( [ 6.3112   5.2715],
         [ 7.6241,  6.3492],
         [ 8.8983,  4.6674],
         [ 7.4880,  9.7795],
         [ 8.7021,  6.7634],
         [10.1655,  7.3991],
         [10.0840,  9.3795],
         [ 9.4194,  5.7345],
         [ 9.3977,  8.8885],
         [11.0075,  7.0806]])
```

Exercise

```
fig, ax = plt.subplots()
ax.scatter(X[y == 0, :][:, 0],
           X[y == 0, :][:, 1],
           label='flamingo',
           color='red')
ax.scatter(X[y == 1, :][:, 0],
           X[y == 1, :][:, 1],
           label='seagull',
           color='blue')
ax.set_xlabel('lenght of the legs')
ax.set_ylabel('wingspan')
ax.legend()
fig.show()
```

```
### X_flamingos ###
 tensor([[ 6.3112,  5.2715],
         [ 7.6241,  6.3492],
         [ 8.8983,  4.6674],
         [ 7.4880,  9.7795],
         [ 8.7021,  6.7634],
         [10.1655,  7.3991],
         [10.0840,  9.3795],
         [ 9.4194,  5.7345],
         [ 9.3977,  8.8885],
         [11.0075,  7.0806]])
```

Exercise

```python
fig, ax = plt.subplots()
ax.scatter(X[y == 0, :][:, 0],
           X[y == 0, :][:, 1],
           label='flamingo',
           color='red')
ax.scatter(X[y == 1, :][:, 0],
           X[y == 1, :][:, 1],
           label='seagull',
           color='blue')
ax.set_xlabel('lenght of the legs')
ax.set_ylabel('wingspan')
ax.legend()
fig.show()
```

```
### X_seagulls ###
 tensor([[5.6595, 3.1686],
         [5.9612, 3.6617],
         [5.3083, 2.3245],
         [4.1404, 2.1670],
         [5.8915, 3.3129],
         [5.7610, 2.1135],
         [3.0120, 3.2828],
         [4.8964, 2.2576],
         [2.7561, 2.7092],
         [5.6683, 4.9879]])
```

Exercise

```
1  fig, ax = plt.subplots()
2  ax.scatter(X[y == 0, :][:, 0],
3             X[y == 0, :][:, 1],
4             label='flamingo',
5             color='red')
6  ax.scatter(X[y == 1, :][:, 0],
7             X[y == 1, :][:, 1],
8             label='seagull',
9             color='blue')
10 ax.set_xlabel('lenght of the legs')
11 ax.set_ylabel('wingspan')
12 ax.legend()
13 fig.show()
```

```
### X_seagulls ###
 tensor([[5.6595, 3.1686],
         [5.9612, 3.6617],
         [5.3083, 2.3245],
         [4.1404, 2.1670],
         [5.8915, 5.3129],
         [5.7610, 2.1135],
         [3.0120, 3.2828],
         [4.8964, 2.2576],
         [2.7561, 2.7092],
         [5.6683, 4.9879]])
```

Exercise

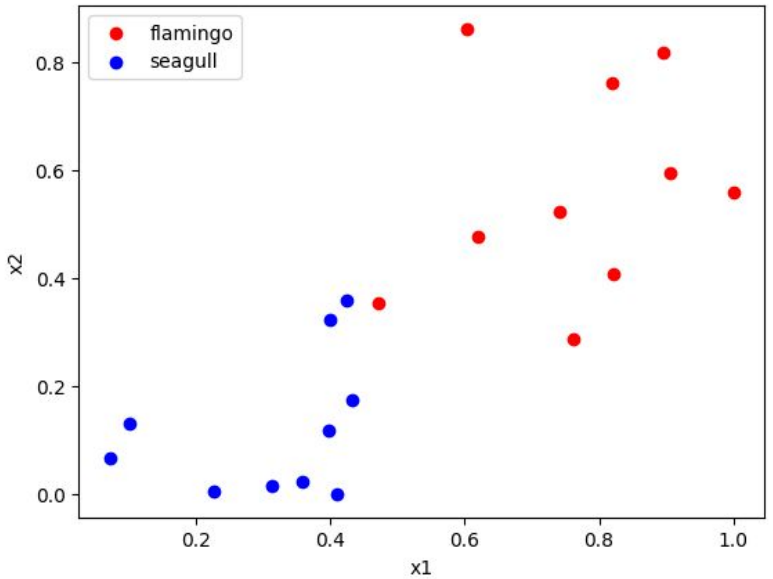
Since data can presents very different scales, in ML we usually prefer to normalize it, say in the range [0,1] .

```
1  X = (X - X.min()) / X.max()
2  print(X)
```

```
tensor([[0.4720, 0.3551],
        [0.6196, 0.4762],
        [0.7629, 0.2871],
        [0.6043, 0.8619],
        [0.7408, 0.5228],
        [0.9053, 0.5943],
        [0.8962, 0.8170],
        [0.8215, 0.4071],
        [0.8190, 0.7618],
        [1.0000, 0.5585],
        [0.3987, 0.1186],
```

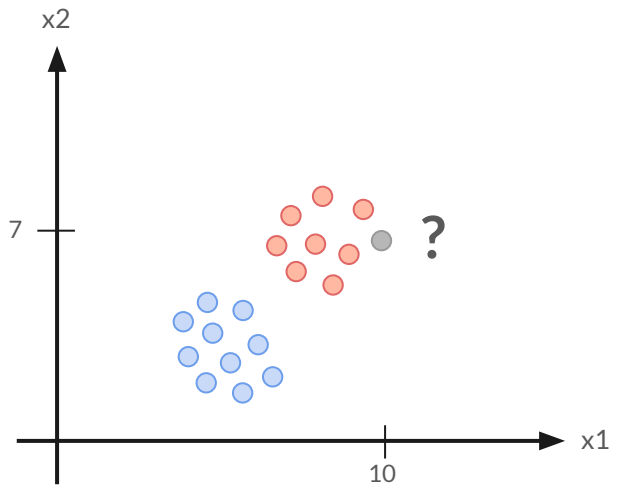
Exercise

```
fig, ax = plt.subplots()
ax.scatter(X[y == 0, :][:, 0],
           X[y == 0, :][:, 1],
           label='flamingo',
           color='red')
ax.scatter(X[y == 1, :][:, 0],
           X[y == 1, :][:, 1],
           label='seagull',
           color='blue')
ax.set_xlabel('lenght of the legs')
ax.set_ylabel('wingspan')
ax.legend()
fig.show()
```

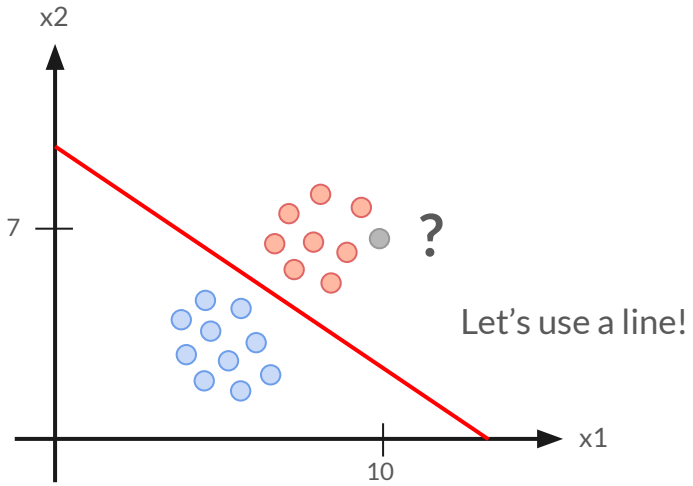


How can we classify?

How can we classify now?



How can we classify now?



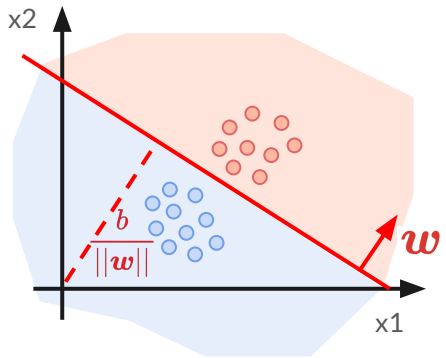
Linear Classifier

A possible solution is to use a line that separates two regions (called **decision regions**).

A line can be expressed in general as:

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + b = \sum_{j=1}^{d} w_j \cdot x_j + b$$

In our case d = 2, but when d = 3 this function expresses a plane in a 3-D space, while for d > 3 we have an hyper-plane.



Linear Classifier

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + b = \sum_{j=1}^{d} w_j \cdot x_j + b$$

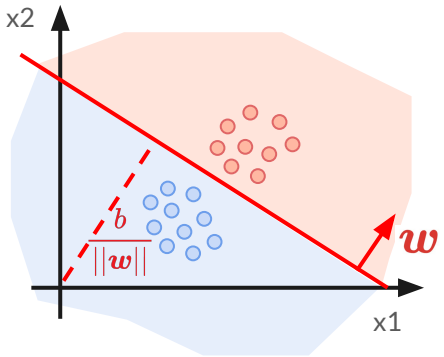
Then we can use this decision rule:

IF $f(\boldsymbol{x}) > 0$:

y = *'flamingo'*

ELSE :

y=*'seagull'*



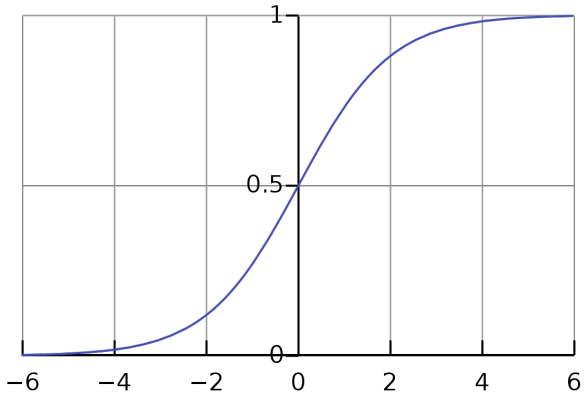
Getting probabilities as output

What if we want a probability as output? In many applications is very important to know not only the predicted label, but also the **level of confidence** that the classifier has with its prediction.

To do so we can use the score assigned by $f(\boldsymbol{x})$ as input to a non-linear **activation function** that squeeze the values in a [0, 1] range.

This function is called the **sigmoid** and it's defined as:

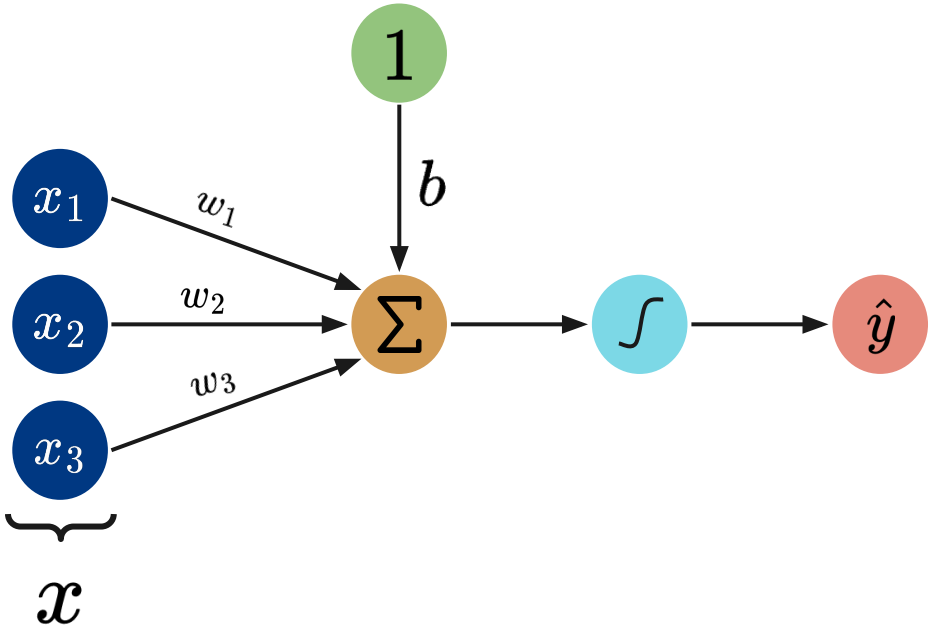
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



Sounds familiar...?

Slides from “Introductory Seminar on Artificial Intelligence and Machine Learning”

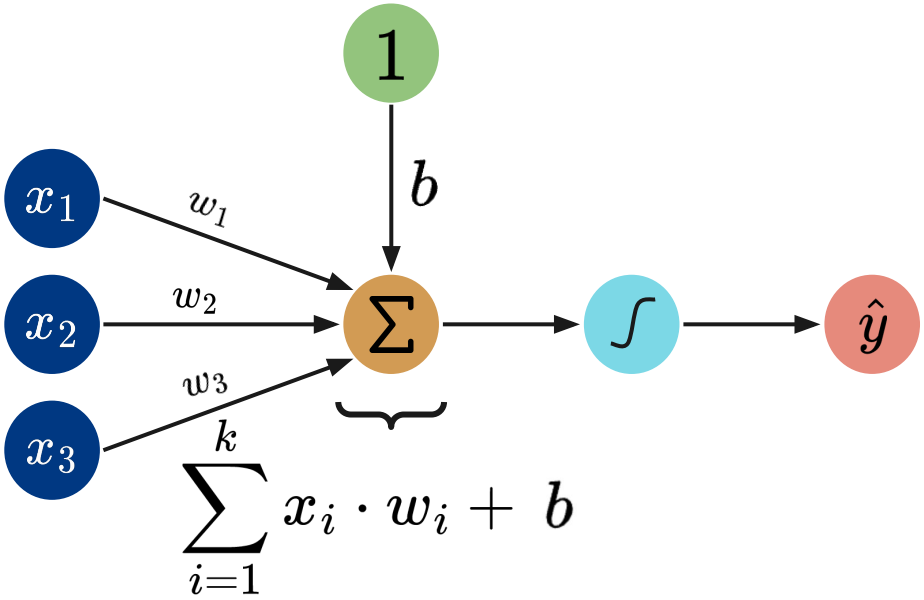
Perceptron



At first, we have the **input x**, characterized by **k** values (**k features**)

Slides from “Introductory Seminar on Artificial Intelligence and Machine Learning”

Perceptron

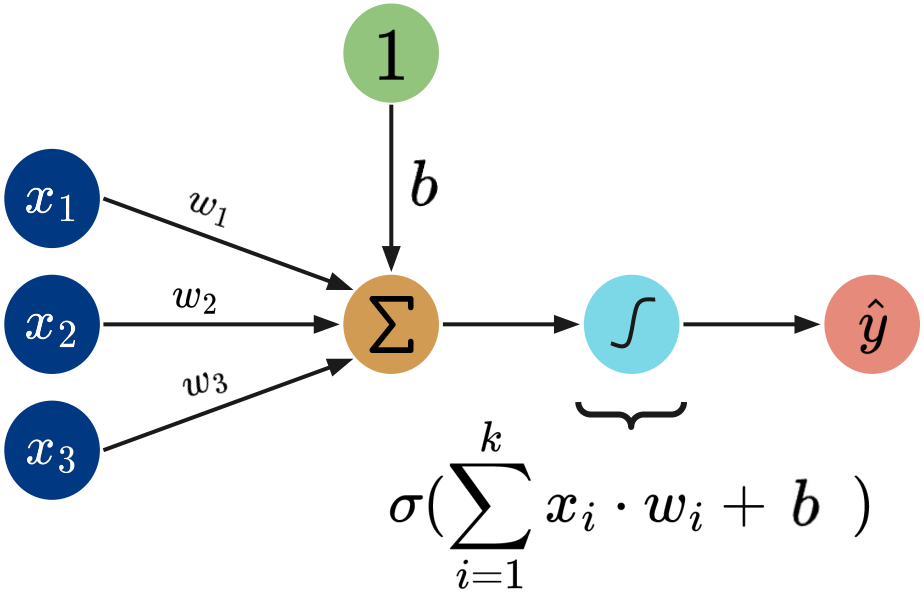


We **multiply** the input values for their relative **weights "w,"** which are then **summed up**

We also add **b**, which is called the **bias**

Slides from “Introductory Seminar on Artificial Intelligence and Machine Learning”

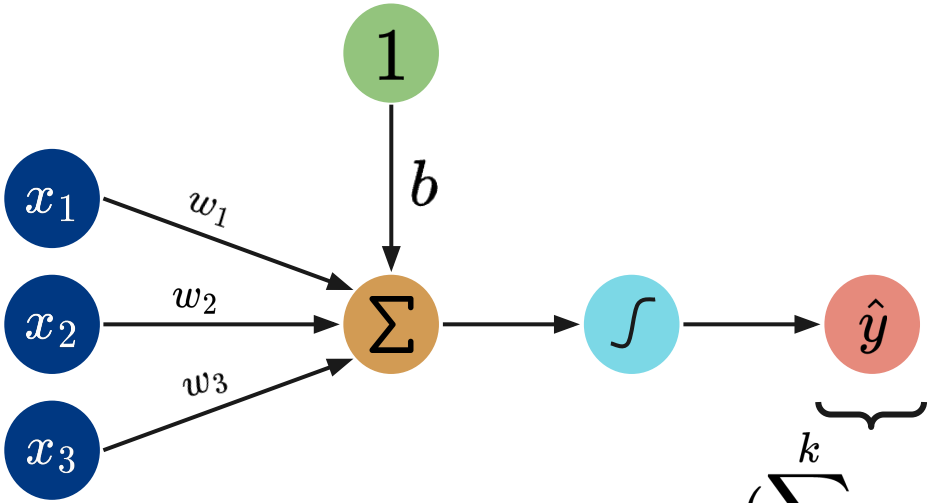
Perceptron



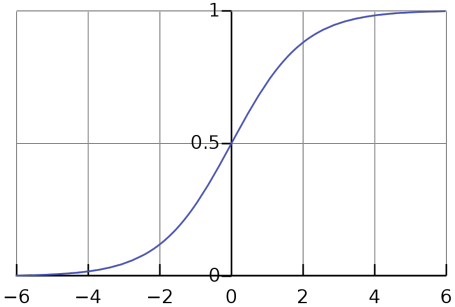
Successively, we apply a nonlinear function called the **"activation function"**

Slides from “Introductory Seminar on Artificial Intelligence and Machine Learning”

Perceptron



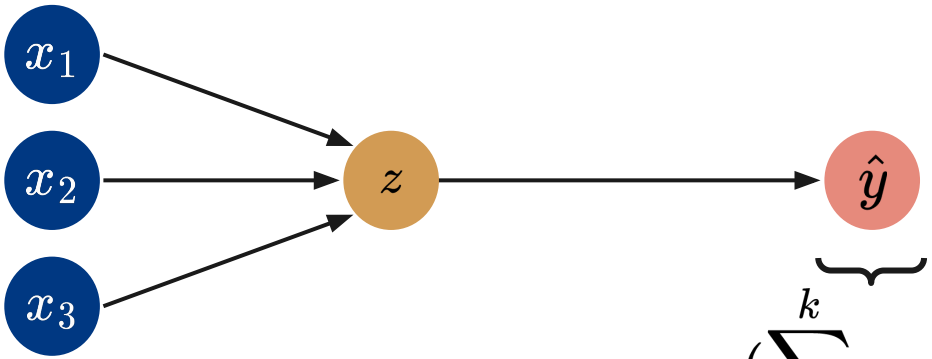
Finally, we obtain the **output** probability



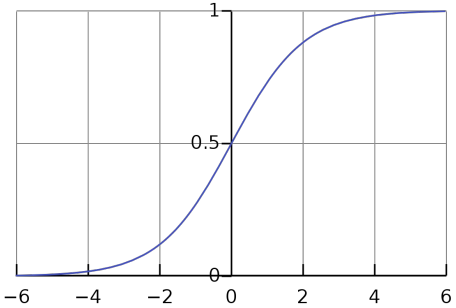
$$\sigma\left(\sum_{i=1}^{k} x_i \cdot w_i + b\right) \qquad \sigma(x) = \frac{1}{1+e^{-x}}$$

Slides from “Introductory Seminar on Artificial Intelligence and Machine Learning”

Perceptron



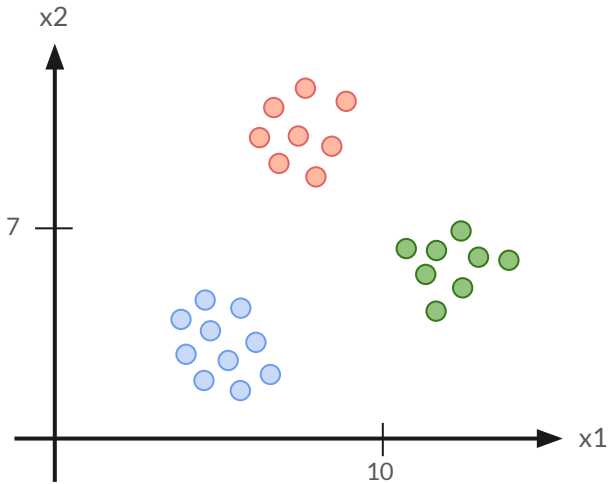
Finally, we obtain the **output** probability



$$\sigma\Big(\sum_{i=1}^{k} x_i \cdot w_i + b\Big) \qquad \sigma(x) = \frac{1}{1+e^{-x}}$$

Multi-class Problems

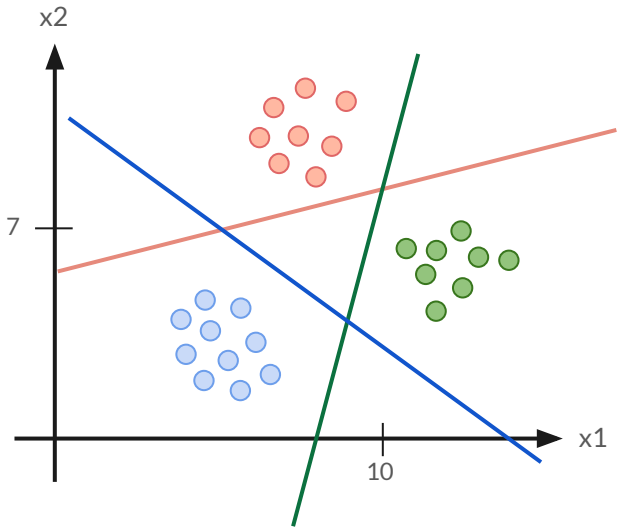
What if the classes are more than two?



Multi-class Problems

What if the classes are more than two?

In this case, the most used strategy is named one-versus-all (OVA), in which we use a classifier to separate each class from the rest.

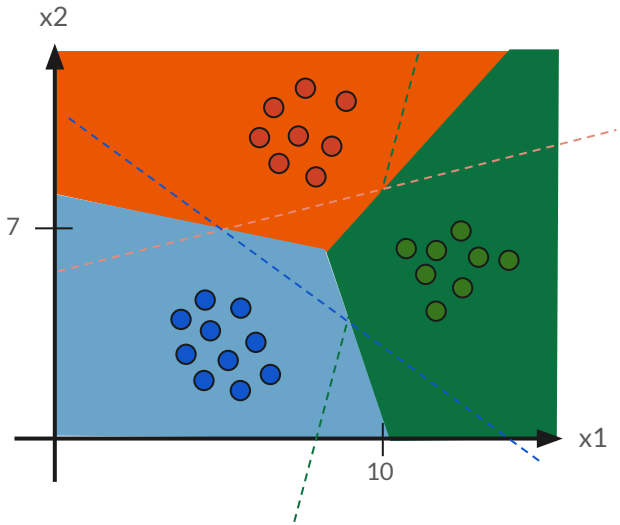


Multi-class Problems

What if the classes are more than two?

In this case, the most used strategy is named one-versus-all (OVA), in which we use a classifier to separate each class from the rest.

To classify a sample one can pick the label corresponding to the highest score among all the classifiers.



Formally...

$$f_1(\boldsymbol{x}) = \boldsymbol{w}_1^T \cdot \boldsymbol{x} + b_1 \qquad \begin{bmatrix} w_{1,1} & w_{1,2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b_1 = s_1$$

Formally...

$$f_1(\boldsymbol{x}) = \boldsymbol{w}_1^T \cdot \boldsymbol{x} + b_1 \qquad \begin{bmatrix} w_{1,1} & w_{1,2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b_1 = s_1$$

$$f_2(\boldsymbol{x}) = \boldsymbol{w}_2^T \cdot \boldsymbol{x} + b_2 \qquad \begin{bmatrix} w_{2,1} & w_{2,2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b_2 = s_2$$

Formally...

$$f_1(\boldsymbol{x}) = \boldsymbol{w}_1^T \cdot \boldsymbol{x} + b_1 \qquad \begin{bmatrix} w_{1,1} & w_{1,2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b_1 = s_1$$

$$f_2(\boldsymbol{x}) = \boldsymbol{w}_2^T \cdot \boldsymbol{x} + b_2 \qquad \begin{bmatrix} w_{2,1} & w_{2,2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b_2 = s_2$$

$$f_3(\boldsymbol{x}) = \boldsymbol{w}_3^T \cdot \boldsymbol{x} + b_3 \qquad \begin{bmatrix} w_{3,1} & w_{3,2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b_3 = s_3$$

Formally…

argmax?

$$f_1(\boldsymbol{x}) = \boldsymbol{w}_1^T \cdot \boldsymbol{x} + b_1 \qquad \begin{bmatrix} w_{1,1} & w_{1,2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b_1 = s_1$$

$$f_2(\boldsymbol{x}) = \boldsymbol{w}_2^T \cdot \boldsymbol{x} + b_2 \qquad \begin{bmatrix} w_{2,1} & w_{2,2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b_2 = s_2 \Leftarrow \hat{y} = 2$$

$$f_3(\boldsymbol{x}) = \boldsymbol{w}_3^T \cdot \boldsymbol{x} + b_3 \qquad \begin{bmatrix} w_{3,1} & w_{3,2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b_3 = s_3$$

MORE formally...

$$f(\boldsymbol{x}) = \boldsymbol{W}^T \cdot \boldsymbol{x} + \boldsymbol{b}$$

$$\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

MORE formally...

$$f(\boldsymbol{x}) = \boldsymbol{W}^T \cdot \boldsymbol{x} + \boldsymbol{b}$$

$$f_1(\boldsymbol{x}) = \boldsymbol{w}_1^T \cdot \boldsymbol{x} + b_1$$

$$\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} s_1 \\ \\ \end{bmatrix}$$

MORE formally...

$$f(\boldsymbol{x}) = \boldsymbol{W}^T \cdot \boldsymbol{x} + \boldsymbol{b}$$

$$f_2(\boldsymbol{x}) = \boldsymbol{w}_2^T \cdot \boldsymbol{x} + b_2 \quad \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} s_1 \\ s_2 \\ \end{bmatrix}$$

MORE formally...

$$f(\boldsymbol{x}) = \boldsymbol{W}^T \cdot \boldsymbol{x} + \boldsymbol{b}$$

$$f_3(\boldsymbol{x}) = \boldsymbol{w}_3^T \cdot \boldsymbol{x} + b_3$$

$$\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix}$$

MORE formally...

$$f(\boldsymbol{x}) = \boldsymbol{W}^T \cdot \boldsymbol{x} + \boldsymbol{b}$$

argmax?

$$\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix} \Leftarrow \hat{y} = 2$$

The Softmax function

As the sigmoid function provide a probability estimate for a single perceptron, for multi-class problem we can use the softmax function to transform the output vector in a discrete probability distribution:

$$\sigma_k(\boldsymbol{s}) = \frac{e^{-s_k}}{\sum_{j=1}^{c} e^{-s_j}}$$

Here the probability of a class also depend on the score assigned to the other classes.

Another ML problem: Regression

Up to now we put a lot of effort on understanding how classification problems works, but if we just want to build a model that tells the weight of a flamingo based on the length of the legs?

In this case we are dealing with a **regression problem**, in which the goal is not to assign a discrete label among a predefined set of classes, but to assign the correct continuous value.

Another ML problem: Regression

A linear function comes to help again, and we can use it to set this simple decision rule:

$$\boldsymbol{x}_2 = f(x)$$

With $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b = \sum_{j=1}^{d} w_j \cdot x_j + b$

Ps. this time we don't need activation function.

Exercise 2.2

Exercise

Let's take the dataset of flamingos and seagulls from the last exercise.

Consider the linear function $f(\mathbf{x}) = \mathbf{w}^T \cdot \mathbf{x} + b$, setting $\mathbf{w} = [1,2]^T$ and $b = 6$.

Let's compute for each sample $\mathbf{x}$ in the dataset:

- the score s = $f(\mathbf{x})$
- the predicted label
- the probability estimate using the sigmoid activation function

Solution

```python
def f(x, w, b):
    # f = (x1 * w1) + (x2 * w2) + b
    return x @ w + b

def sigmoid(s):
    return 1 / (1 + torch.exp(-s))

w = torch.tensor([-1., -2.])
b = torch.tensor(10)
scores = f(X, w, b)
y_pred = (scores > 0).float()
probabilities = sigmoid(scores)
print("### Scores ###\n", scores)
print("### Pred. ###\n", y_pred)
print("### Probabilities ###\n", probabilities)
```

Solution

```
### Scores ###
 tensor([-6.8542e+00, -1.0322e+01, -8.2330e+00, -1.7047e+01, -1.2229e+01,
        -1.4964e+01, -1.8843e+01, -1.0888e+01, -1.7175e+01, -1.5169e+01,
        -1.9968e+00, -3.2847e+00,  4.2701e-02,  1.5257e+00, -6.5173e+00,
         1.2098e-02,  4.2233e-01,  5.8838e-01,  1.8254e+00, -5.6441e+00])
### Pred. ###
 tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 0., 1., 1., 1.,
        1., 0.])
### Probabilities ###
 tensor([1.0539e-03, 3.2884e-05, 2.6567e-04, 3.9500e-08, 4.8866e-06, 3.1723e-07,
        6.5549e-09, 1.8673e-05, 3.4761e-08, 2.5842e-07, 1.1954e-01, 3.6101e-02,
        5.1067e-01, 8.2137e-01, 1.4755e-03, 5.0302e-01, 6.0404e-01, 6.4299e-01,
        8.6121e-01, 3.5259e-03])
```

```
[40]  1  print(torch.sigmoid(scores))    # built-in PyTorch function
```

```
tensor([1.0539e-03, 3.2884e-05, 2.6567e-04, 3.9500e-08, 4.8866e-06, 3.1723e-07,
        6.5549e-09, 1.8673e-05, 3.4761e-08, 2.5842e-07, 1.1954e-01, 3.6101e-02,
        5.1067e-01, 8.2137e-01, 1.4755e-03, 5.0302e-01, 6.0404e-01, 6.4299e-01,
        8.6121e-01, 3.5259e-03])
```

Exercise 2.3

Exercise

Now consider the sample $\mathbf{x} = \begin{bmatrix} 0.5 & 1 \end{bmatrix}^T$

We have to assign this sample to one of three classes $y \in \{0,1,2\}$ using the function $f(\mathbf{x}) = \mathbf{W} \cdot \mathbf{x} + \mathbf{b}$ where:

$$\mathbf{W} = \begin{bmatrix} 0.2 & 0.5 \\ 2.1 & -1.3 \\ -3 & -2.2 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 0.5 & -0.2 & 4.1 \end{bmatrix}^T$$

Exercise

Compute:

- The score vector $\mathbf{s} = f(\mathbf{x})$
- The discrete probability distribution using the softmax function $\sigma_k(\mathbf{s}) = \frac{e^{-s_k}}{\sum_{j=1}^{c} e^{-s_j}}$
- The predicted labels

Exercise

```
[5]  1  x = torch.tensor([[0.5, 1]],
     2                   dtype=torch.float).T   # torch.Size([2, 1])
     3  W = torch.tensor([[0.2, 0.5],
     4                    [2.1, -1.3],
     5                    [-3, -2.2]],
     6                   dtype=torch.float)     # torch.Size([3, 2])
     7  b = torch.tensor([[0.5, -0.2, 4.1]],
     8                   dtype=torch.float).T   # torch.Size([3, 1])
     9  s = W @ x + b
    10  y_pred = s.argmax(dim=1)
    11  print(s)
    12  print(y_pred)
```

```
tensor([[ 1.1000],
        [-0.4500],
        [ 0.4000]])
tensor([0, 0, 0])
```

Exercise

[60]
```
def softmax(s):
    exps = torch.exp(s)
    return  exps/ exps.sum()

probs = softmax(s.T)
print(probs)
```

```
tensor([[0.5852, 0.1242, 0.2906]])
```

[62]
```
from torch.nn import Softmax
probs = Softmax()(s.T)  # built-in PyTorch function
print(probs)
```

```
tensor([[0.5852, 0.1242, 0.2906]])
```

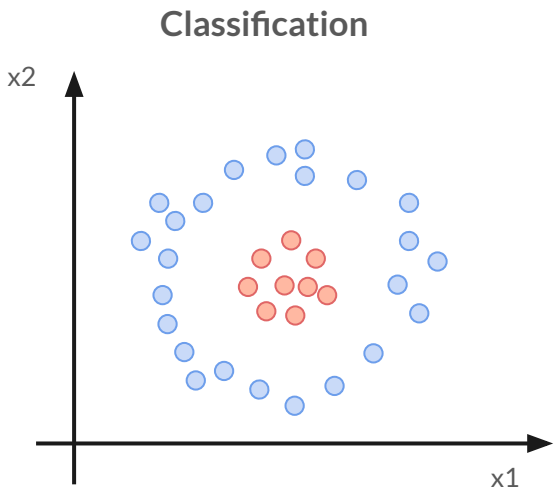
Artificial Neural Networks

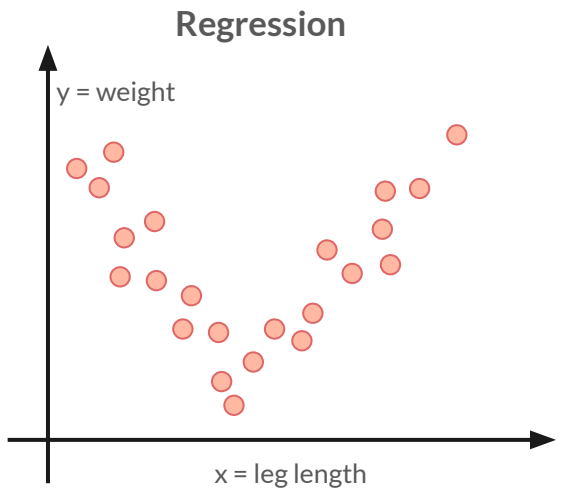
Non-linearities in the data

Now we have seen some examples of ML problems for which a linear function is sufficient to describe the data.

But what if the data present non-linearities?

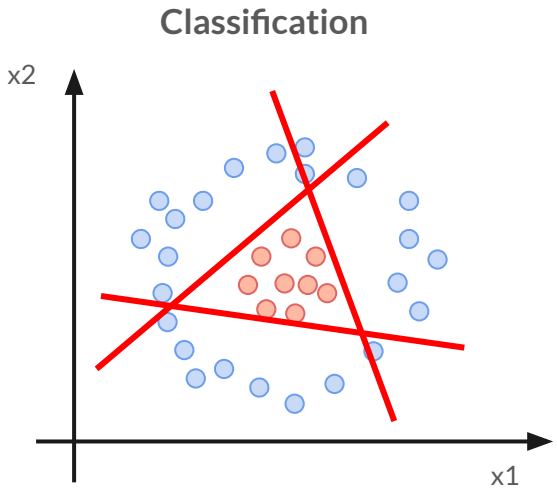
Non-linearities in the data

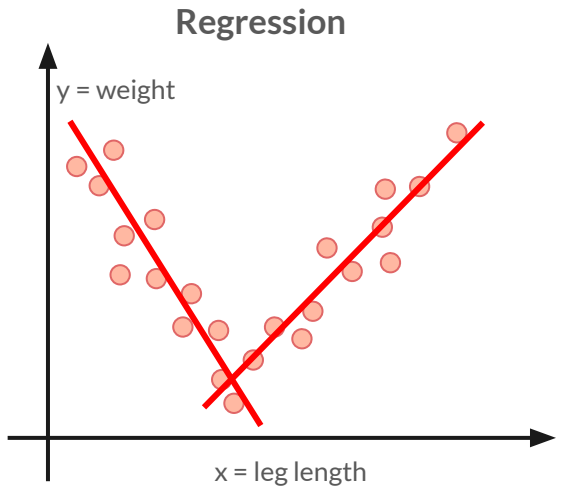




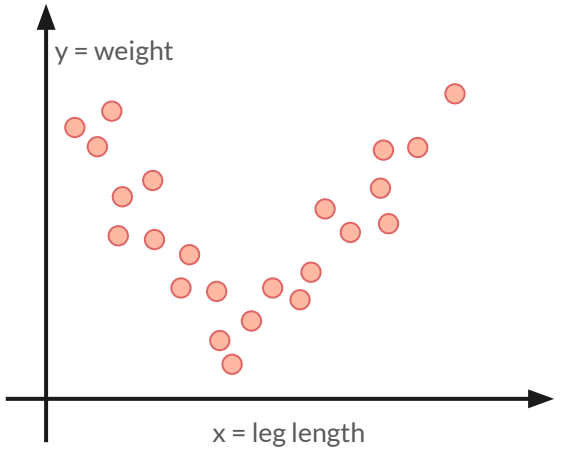
Non-linearities in the data

Combination of linear functions!



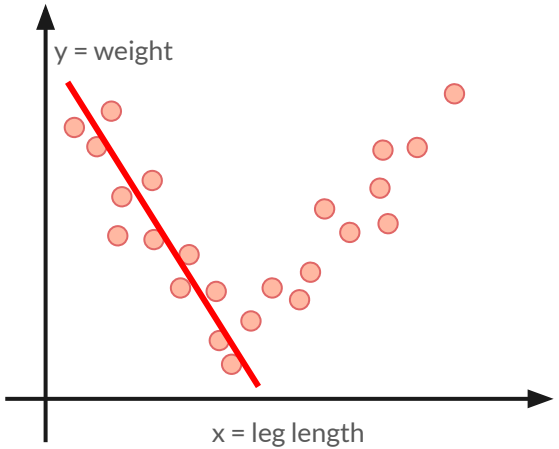


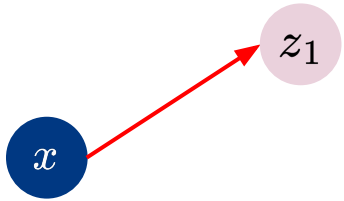
Combining linear functions



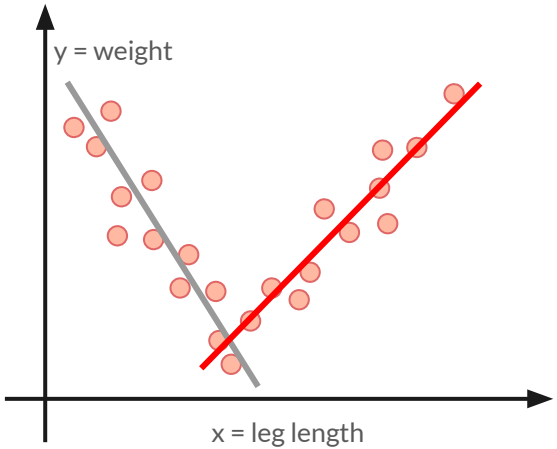
x

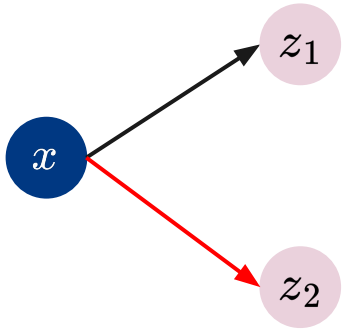
Combining linear functions



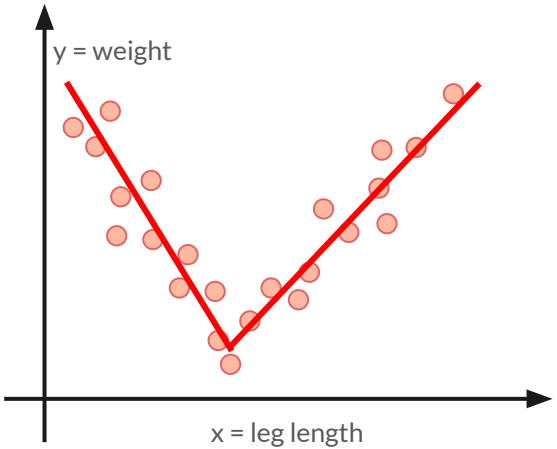


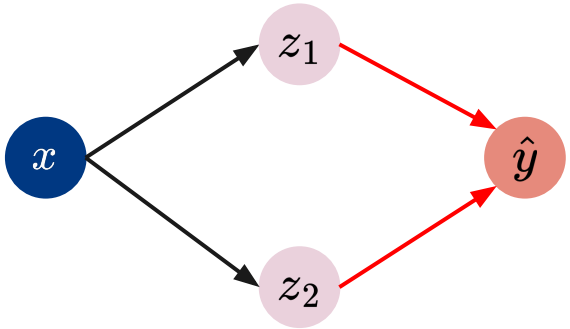
Combining linear functions





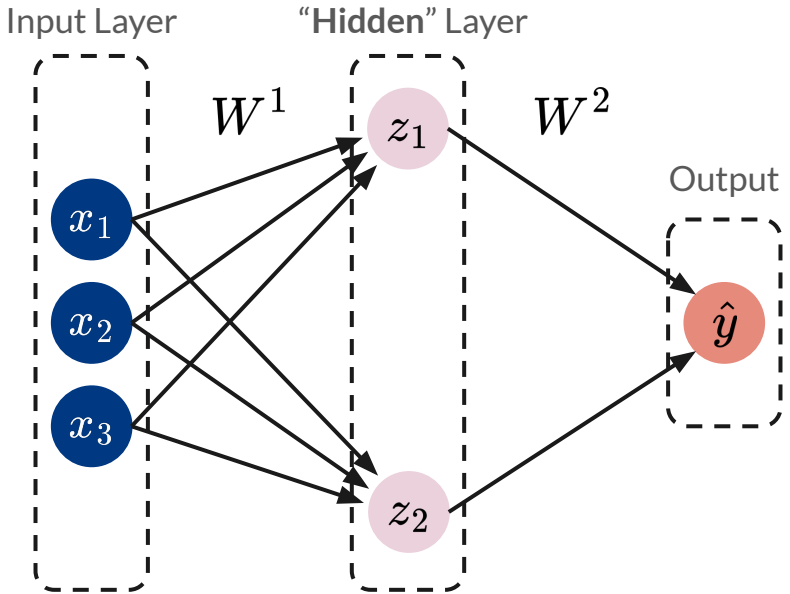
Combining linear functions





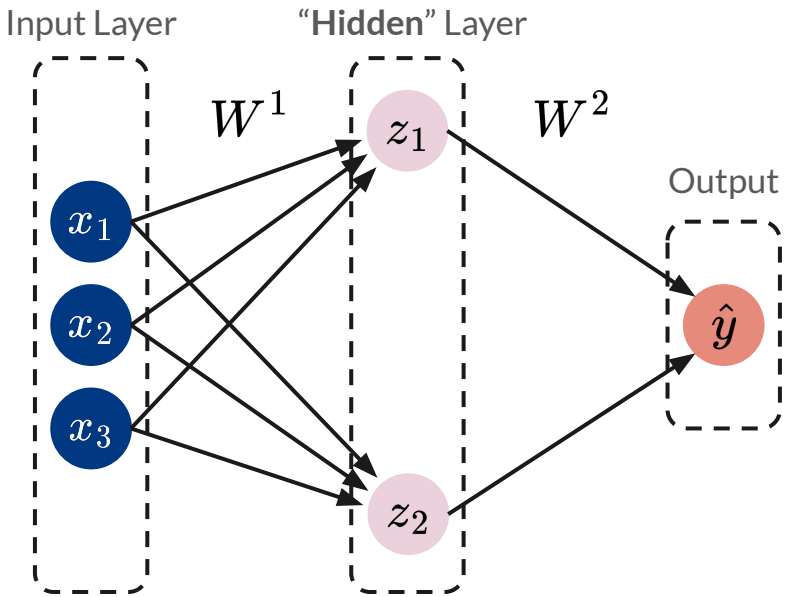
Slides from “Introductory Seminar on Artificial Intelligence and Machine Learning”

Multi Layer Perceptron



Slides from “Introductory Seminar on Artificial Intelligence and Machine Learning”

Multi Layer Perceptron



$$W^1 = \begin{bmatrix} b_0^1 & b_1^1 \\ w_{0,1}^1 & w_{1,1}^1 \\ w_{0,2}^1 & w_{1,2}^1 \\ w_{0,3}^1 & w_{1,3}^1 \end{bmatrix} \quad W^2 = \begin{bmatrix} b_0^2 \\ w_{0,1}^2 \\ w_{0,3}^2 \end{bmatrix}$$

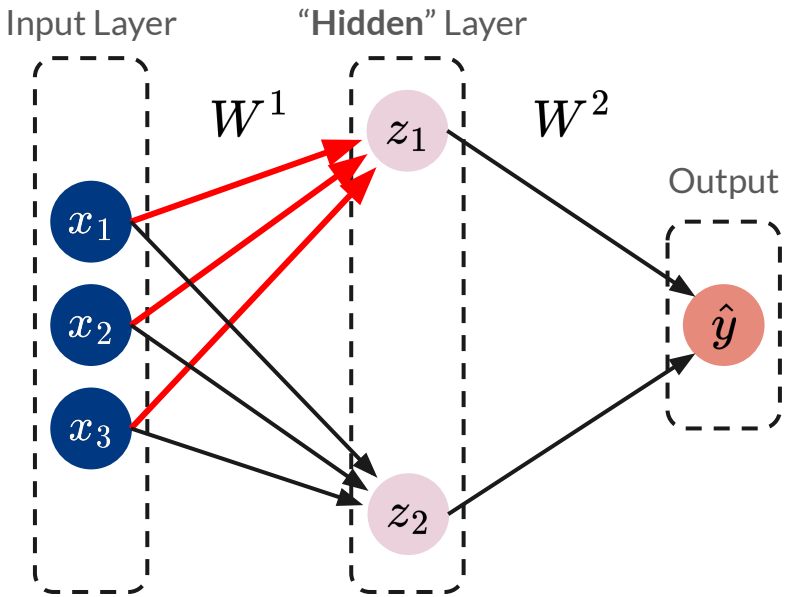
$$x = [x_1, x_2, x_3]$$

$$z = x \cdot W^1$$

$$\hat{y} = z \cdot W^2$$

Slides from “Introductory Seminar on Artificial Intelligence and Machine Learning”

Multi Layer Perceptron



$$W^1 = \begin{bmatrix} b_0^1 & b_1^1 \\ w_{0,1}^1 & w_{1,1}^1 \\ w_{0,2}^1 & w_{1,2}^1 \\ w_{0,3}^1 & w_{1,3}^1 \end{bmatrix} \quad W^2 = \begin{bmatrix} b_0^2 \\ w_{0,1}^2 \\ w_{0,3}^2 \end{bmatrix}$$

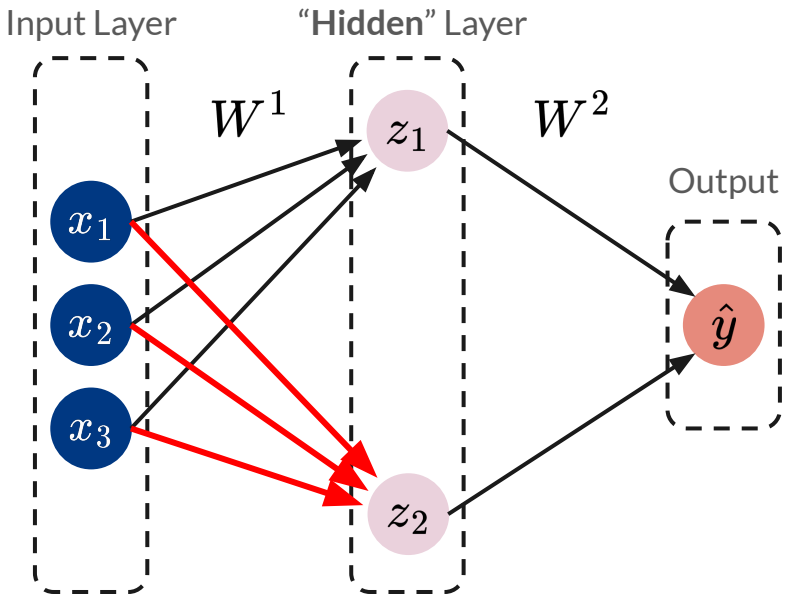
$$x = [x_1, x_2, x_3]$$

$$z = x \cdot W^1$$

$$\hat{y} = z \cdot W^2$$

Slides from “Introductory Seminar on Artificial Intelligence and Machine Learning”

Multi Layer Perceptron



$$W^1 = \begin{bmatrix} b_0^1 & b_1^1 \\ w_{0,1}^1 & w_{1,1}^1 \\ w_{0,2}^1 & w_{1,2}^1 \\ w_{0,3}^1 & w_{1,3}^1 \end{bmatrix} \quad W^2 = \begin{bmatrix} b_0^2 \\ w_{0,1}^2 \\ w_{0,3}^2 \end{bmatrix}$$

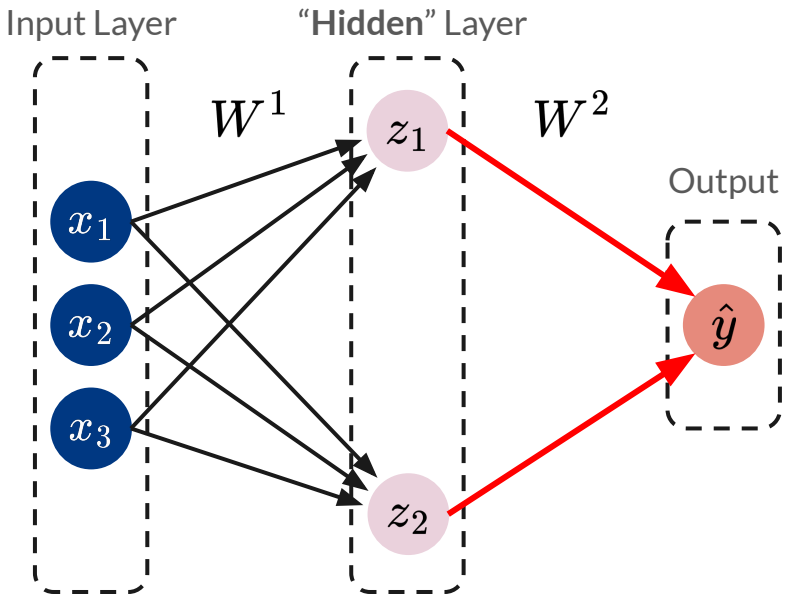
$$x = [x_1, x_2, x_3]$$

$$z = x \cdot W^1$$

$$\hat{y} = z \cdot W^2$$

Slides from “Introductory Seminar on Artificial Intelligence and Machine Learning”

Multi Layer Perceptron



$$W^1 = \begin{bmatrix} b_0^1 & b_1^1 \\ w_{0,1}^1 & w_{1,1}^1 \\ w_{0,2}^1 & w_{1,2}^1 \\ w_{0,3}^1 & w_{1,3}^1 \end{bmatrix} \quad W^2 = \begin{bmatrix} b_0^2 \\ w_{0,1}^2 \\ w_{0,3}^2 \end{bmatrix}$$

$$x = [x_1, x_2, x_3]$$

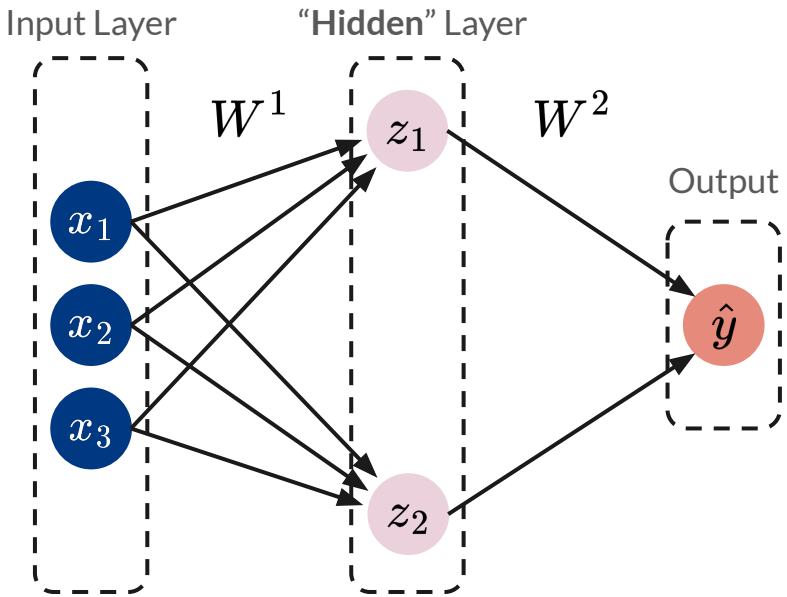
$$z = x \cdot W^1$$

$$\hat{y} = z \cdot W^2$$

Slides from “Introductory Seminar on Artificial Intelligence and Machine Learning”

Multi Layer Perceptron

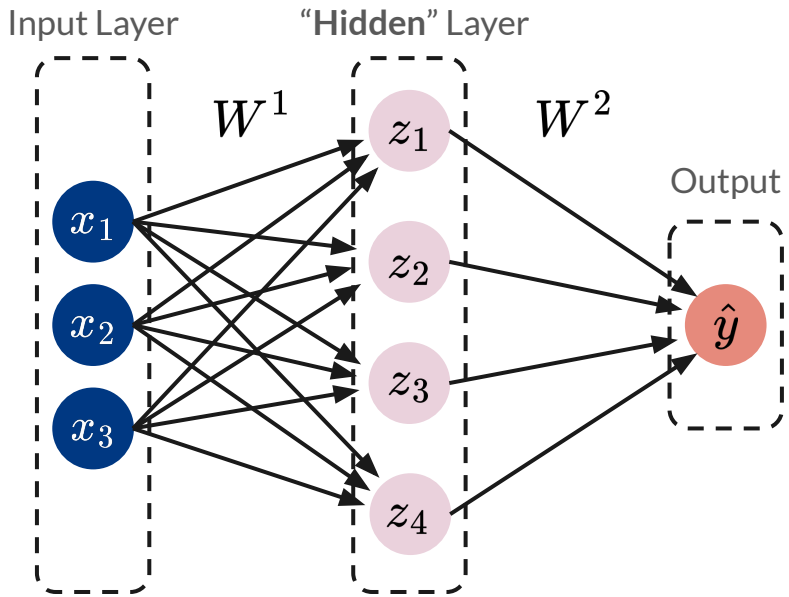
- The Multi Layer Perceptron (MLP) make it possible to build **deep neural network**



Slides from “Introductory Seminar on Artificial Intelligence and Machine Learning”

Multi Layer Perceptron

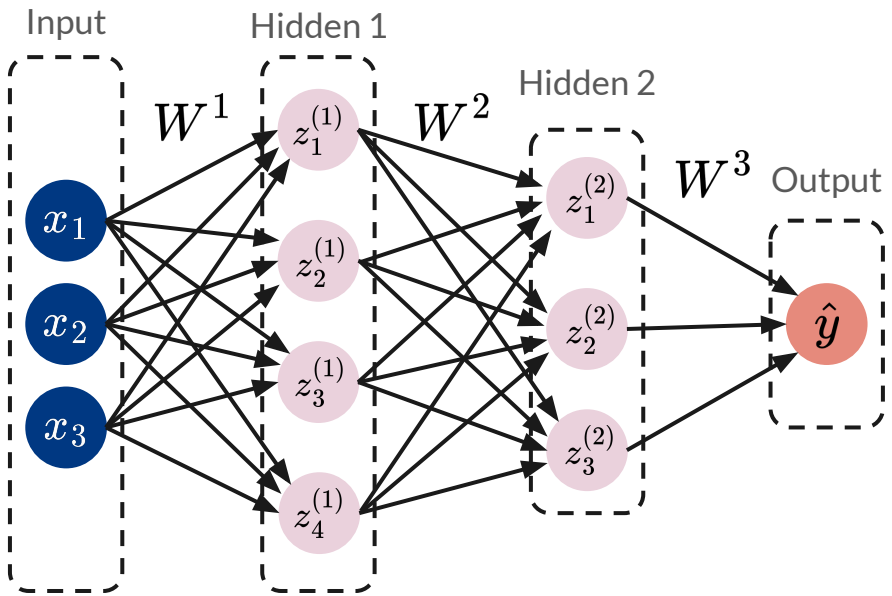
- The Multi Layer Perceptron (MLP) make it possible to build **deep neural network**
- Adding **neurons**…



Slides from “Introductory Seminar on Artificial Intelligence and Machine Learning”

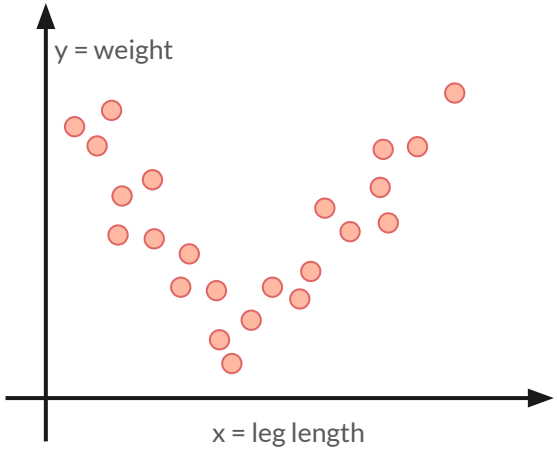
Multi Layer Perceptron

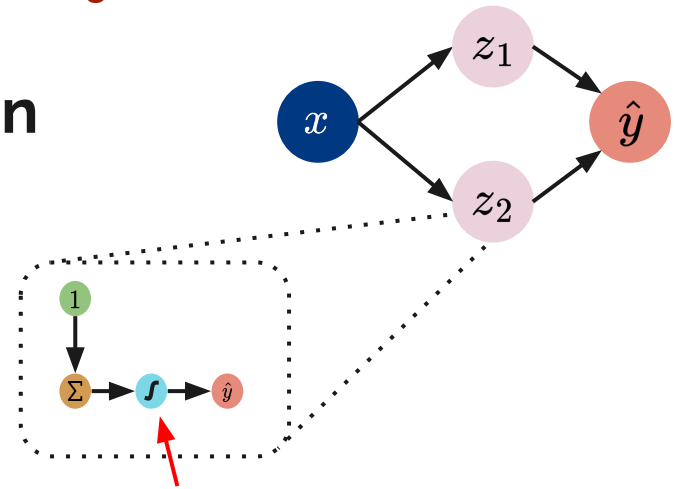
- The Multi Layer Perceptron (MLP) make it possible to build **deep neural network**
- Adding **neurons**…
- … as well as **layers**



Slides from “Introductory Seminar on Artificial Intelligence and Machine Learning”

Non-Linear Activation Function

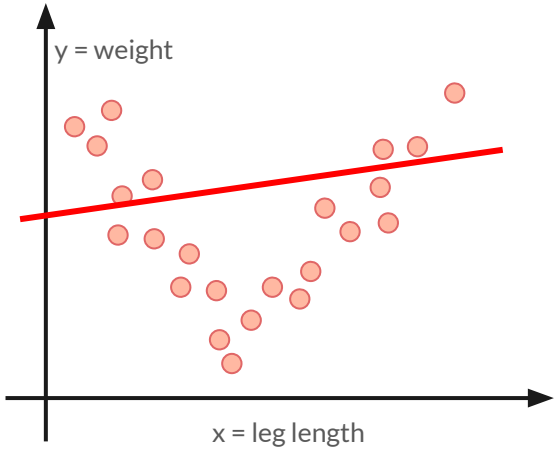


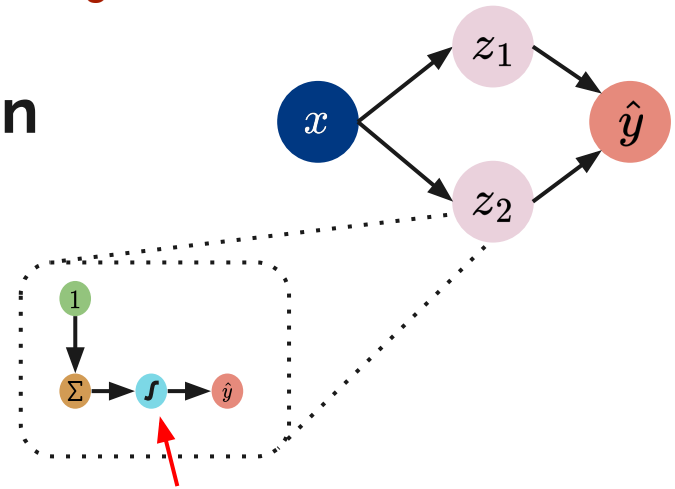


- But why do we need a **non-linear activation function**?

Slides from “Introductory Seminar on Artificial Intelligence and Machine Learning”

Non-Linear Activation Function

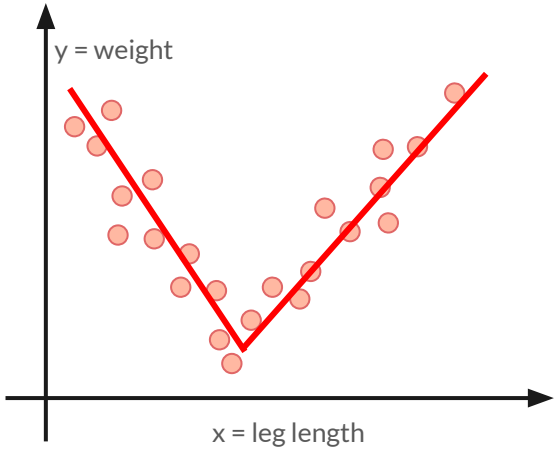


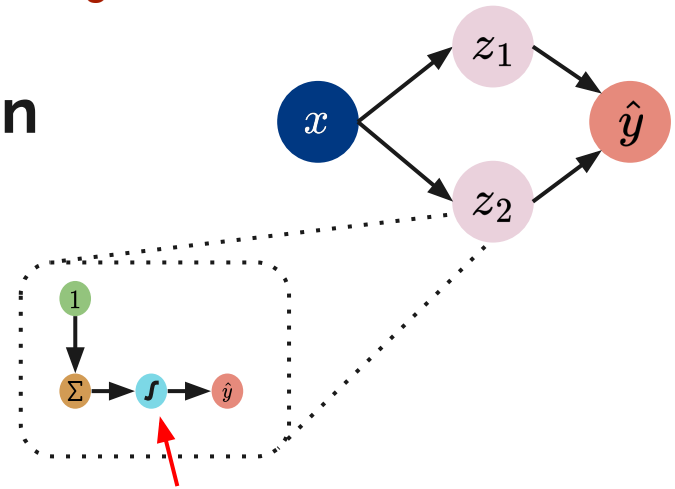


- But why do we need a **non-linear activation function**?
- We can construct **only linear classifiers** using linear activations, no matter how deep the network is!

Slides from “Introductory Seminar on Artificial Intelligence and Machine Learning”

Non-Linear Activation Function



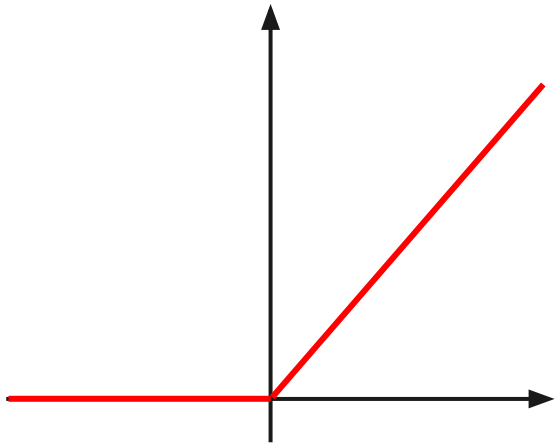


- But why do we need a **non-linear activation function**?
- We can construct **only linear classifiers** using linear activations, no matter how deep the network is!
- The non-linearity of the activation function also lets us **construct non-linear classifiers**

The ReLU Activation Function

While for treating the outputs we use activation function to obtain probability estimates, in the hidden layers we use non-linear activations to build a non-linear function.

One of the most used type of activation is the **ReLU activation** (Rectified Linear Unit), which is very efficient and simple to implement.

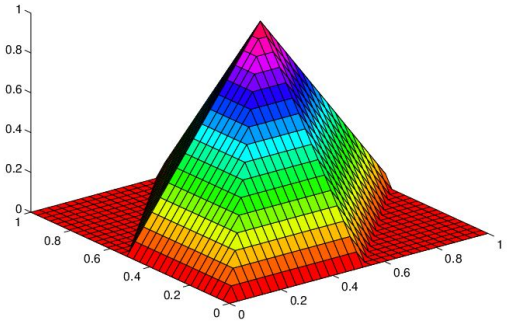


ReLU

$$\max(0, x)$$

The ReLU Activation Function

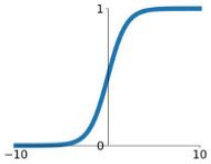
By stacking together linear layers and ReLU activation the output function becomes a **piecewise-linear function**.



Other Activation Functions

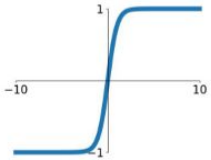
Sigmoid

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



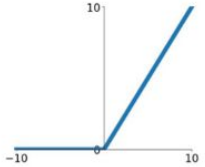
tanh

$$\tanh(x)$$



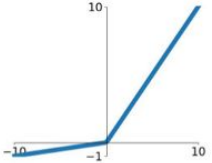
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

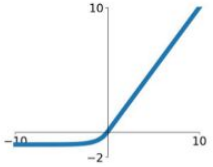


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Built-in modules in PyTorch

PyTorch has a whole submodule dedicated to neural networks, called **torch.nn**.

It contains the building blocks needed to create all sorts of neural network architectures.

Those building blocks are called modules. A PyTorch module is a Python class deriving from the **nn.Module** base class.

A module can have one or more Parameter instances as attributes, which are tensors whose values are optimized during the training process.

A module can also have one or more submodules (subclasses of `nn.Module`) as attributes, and it will be able to track their parameters as well.

Linear function in PyTorch

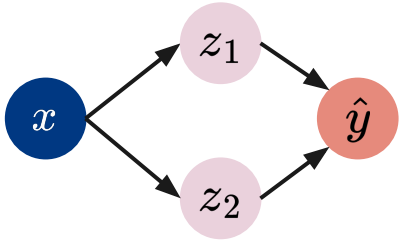
In PyTorch we can instantiate a linear function using the **nn.Linear** class that inherits from nn.Module.

```
from torch import nn
n_features = 2
n_classes = 3
linear = nn.Linear(n_features, n_classes)
x = torch.tensor([1., 2.])
out = linear(x)
```

In general, the first argument indicate the **input size**, while the second the **output size**.

This will help us understanding how to move from linear functions to neural networks

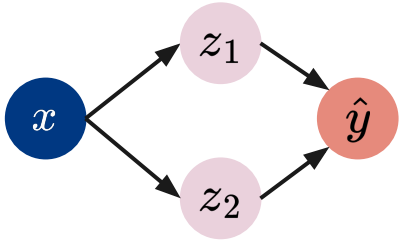
MLP in PyTorch




```python
import torch
from torch import nn

class MLP(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_dim):
        super().__init__()
        self.linear1 = nn.Linear(input_dim, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, output_dim)
        self.relu = nn.ReLU()

    def forward(self, x):
        z = self.relu(self.linear1(x))
        out = self.relu(self.linear2(z))
        return out
```

- Inherit the nn.Module

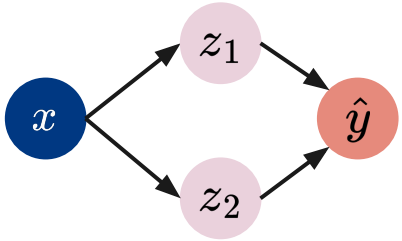
MLP in PyTorch




```python
import torch
from torch import nn

class MLP(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_dim):
        super().__init__()
        self.linear1 = nn.Linear(input_dim, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, output_dim)
        self.relu = nn.ReLU()

    def forward(self, x):
        z = self.relu(self.linear1(x))
        out = self.relu(self.linear2(z))
        return out
```

- Inherit the nn.Module
- Instantiate the basic components (layers and activations)

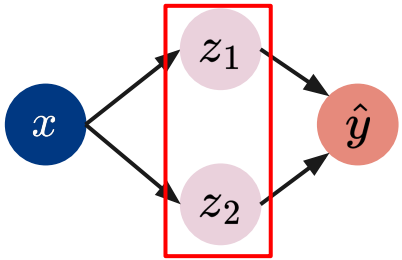
MLP in PyTorch




```python
import torch
from torch import nn

class MLP(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_dim):
        super().__init__()
        self.linear1 = nn.Linear(input_dim, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, output_dim)
        self.relu = nn.ReLU()

    def forward(self, x):
        z = self.relu(self.linear1(x))
        out = self.relu(self.linear2(z))
        return out
```

- Inherit the nn.Module
- Instantiate the basic components (layers and activations)
- Specify the sequence of operations in the **forward method**

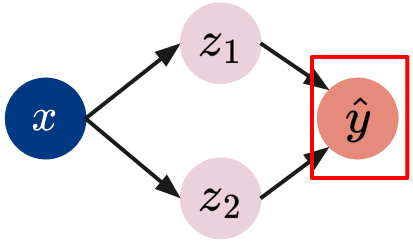
MLP in PyTorch




```python
import torch
from torch import nn

class MLP(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_dim):
        super().__init__()
        self.linear1 = nn.Linear(input_dim, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, output_dim)
        self.relu = nn.ReLU()

    def forward(self, x):
        z = self.relu(self.linear1(x))
        out = self.relu(self.linear2(z))
        return out
```

- Inherit the nn.Module
- Instantiate the basic components (layers and activations)
- Specify the sequence of operations in the **forward method**

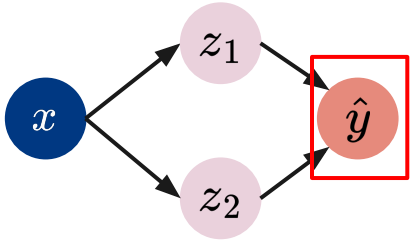
MLP in PyTorch




```python
import torch
from torch import nn

class MLP(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_dim):
        super().__init__()
        self.linear1 = nn.Linear(input_dim, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, output_dim)
        self.relu = nn.ReLU()

    def forward(self, x):
        z = self.relu(self.linear1(x))
        out = self.relu(self.linear2(z))
        return out
```

- Inherit the nn.Module
- Instantiate the basic components (layers and activations)
- Specify the sequence of operations in the **forward method**

MLP in PyTorch




```python
import torch
from torch import nn

class MLP(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_dim):
        super().__init__()
        self.linear1 = nn.Linear(input_dim, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, output_dim)
        self.relu = nn.ReLU()

    def forward(self, x):
        z = self.relu(self.linear1(x))
        out = self.relu(self.linear2(z))
        return out
```

```python
model = MLP()
out = model(x)
```

- Inherit the nn.Module
- Instantiate the basic components (layers and activations)
- Specify the sequence of operations in the **forward method**
- **Instantiate the model class and produce the output**

Using __call__ rather than forward

All PyTorch-provided subclasses of nn.Module have their __call__ method defined

Calling an instance of nn.Module produces the same output of calling the forward

```
y = model(x)
y = model.forward(x)
```

However, there is a silent error here, for which it is always recommended to use __call__

Using __call__ rather than forward

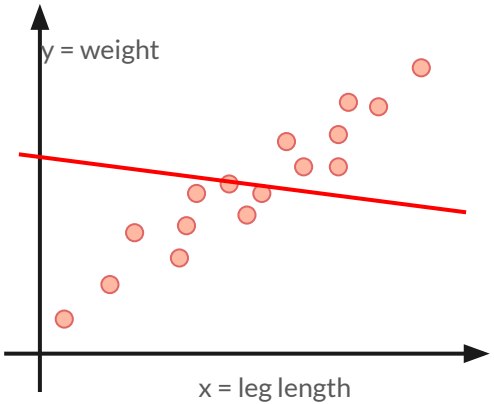
There are hidden functions that are called inside the __call__ before the forward, and calling the forward alone would skip all these operations

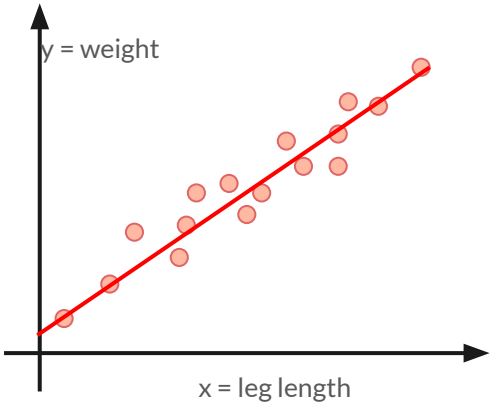
Hooks are used to customize the call with user-defined functions, e.g., logging functions

```
def __call__(self, *input, **kwargs):
    for hook in self._forward_pre_hooks.values():
        hook(self, input)

    result = self.forward(*input, **kwargs)

    for hook in self._forward_hooks.values():
        hook_result = hook(self, input, result)
        ...
    for hook in self._backward_hooks.values():
        ...
    return results
```

Training a Neural Network

How can we obtain useful weights?

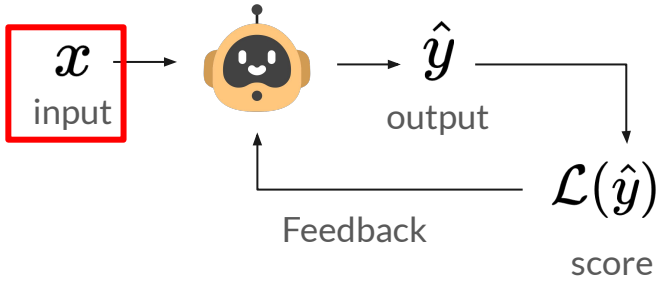




How can we obtain useful weights?

Until now we learned about:

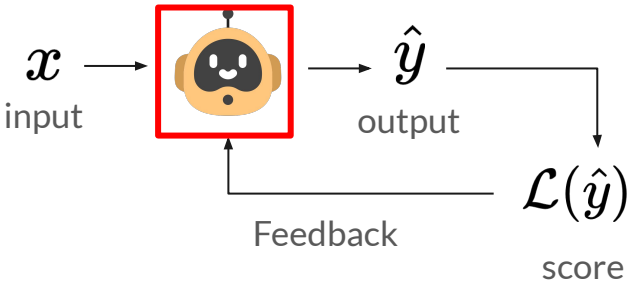
- **how to represent the data**



How can we obtain useful weights?

Until now we learned about:

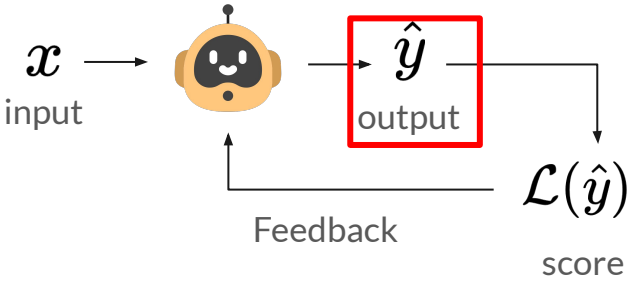
- how to represent the data
- **what function to use as our model**



How can we obtain useful weights?

Until now we learned about:

- how to represent the data
- what function to use as our model
- **how to treat the output**

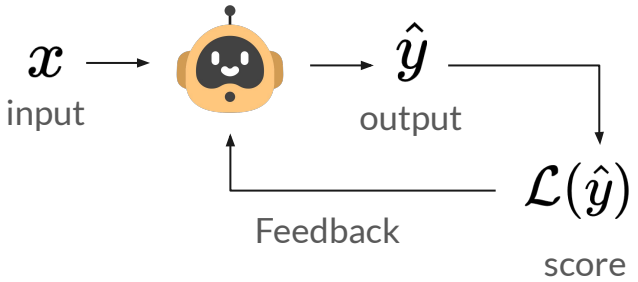


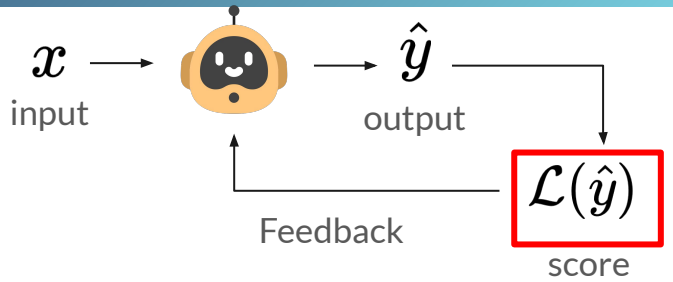
How can we obtain useful weights?

Until now we learned about:

- how to represent the data
- what function to use as our model
- how to treat the output

Now we can focus on the final part of the pipeline…





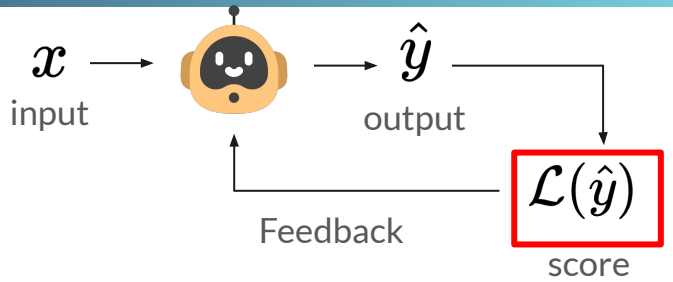
How to quantify the error?

To evaluate how much we are happy about the model's outputs, we define a **loss function** ℓ

$$\ell(y, f^{\theta}(x))$$

Where:

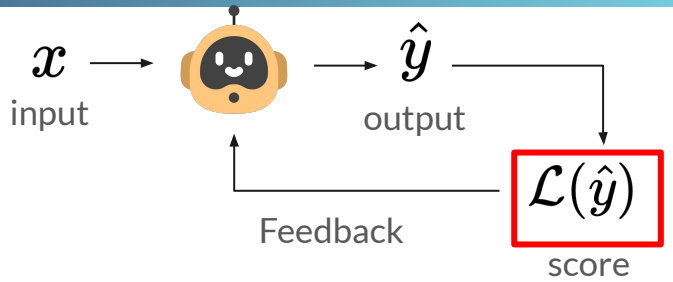
- y is the ground truth (what we want the model to output)
- $f^{\theta}(x)$ is the decision function



How to quantify the error?

By getting the average on the whole training set we get an estimate of the quality of the model

$$\mathcal{L}(D, \theta) = \frac{1}{n} \sum_{i=1}^{n} \ell(y, f^{\theta}(x))$$



How to quantify the error?

By getting the average on the whole training set we get an estimate of the quality of the model

$$\mathcal{L}(D, \theta) = \frac{1}{n} \sum_{i=1}^{n} \ell(y, f^{\theta}(x))$$

But what function do we use as ℓ ?

Cross Entropy

The main idea is the notion of **diversity. How different is my output from the desired one?**
We can express this diversity by using the notion of **Cross-Entropy**

Cross-Entropy measures the **divergence between two categorical distributions**

$$H(\hat{y}, y) = \sum_{k \in \mathcal{Y}} -y_k \cdot log(\hat{y}_k)$$

$$\begin{bmatrix} 0.83 \\ 0.17 \end{bmatrix}$$

Output (as a probability distribution)

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

Label (as a probability distribution)

Cross Entropy - Example

$$H(\hat{y}, y) = -y_0 \log \hat{y}_0 - y_1 \log \hat{y}_1$$

$$H(\hat{y}, y) = -0 \log(0.83) - 1 \log(0.17)$$

$$= -\log(0.17) = \boxed{1.77}$$

$$\begin{bmatrix} 0.83 \\ 0.17 \end{bmatrix}$$

Output (as a probability distribution)

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

Label (as a probability distribution)

Cross Entropy - Example

$$H(\hat{y}, y) = -y_0 \log \hat{y}_0 - y_1 \log \hat{y}_1$$

$$H(\hat{y}, y) = -0 \log(0.21) - 1 \log(0.79)$$

$$= -\log(0.79) = \boxed{0.23}$$

- We can decrease the error by making the prediction closer to the desired output

$$\begin{bmatrix} 0.21 \\ 0.79 \end{bmatrix}$$

Output (as a probability distribution)

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

Label (as a probability distribution)

Mean Square Error

- If we are estimating continuous values we need a proper measure of diversity for real values
- In general, the divergence may be captured by computing the distance between the two numbers
  - Mean Absolute Error (MAE)
- The most common idea is however to weight differently large distances
  - Mean Square Error (MSE)

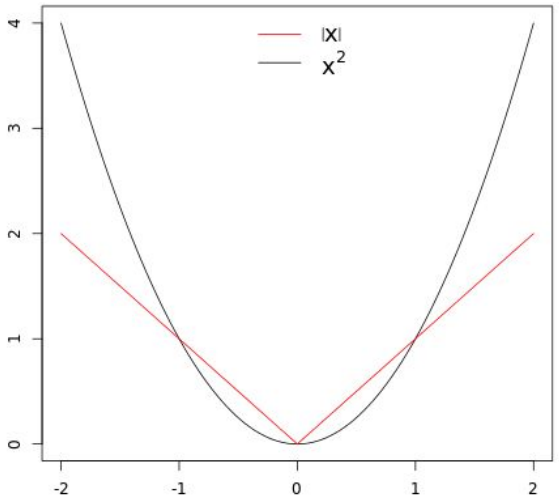
23.5

Output

25

Label

Mean Square Error



$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y})^2$$

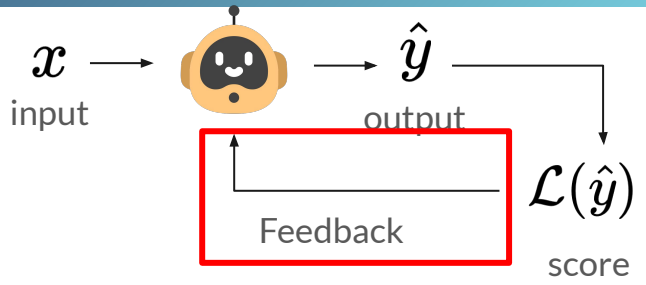
$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}|$$

Loss functions in PyTorch

```
1  loss_fn = torch.nn.CrossEntropyLoss()
2  loss = loss_fn(y_pred, y)
```

```
1  loss_fn = torch.nn.MSELoss()
2  loss = loss_fn(y_pred, y)
```

* Be aware that the Cross-Entropy implementation of PyTorch automatically applies the **softmax function** on the input tensor to ensure it is a discrete probability distribution.



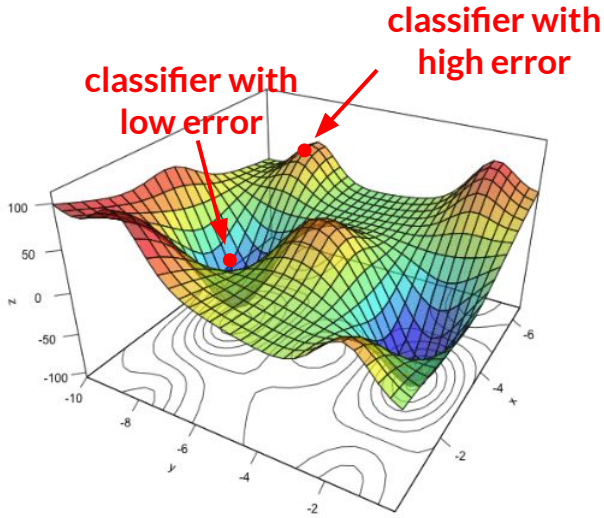
Learning from Errors

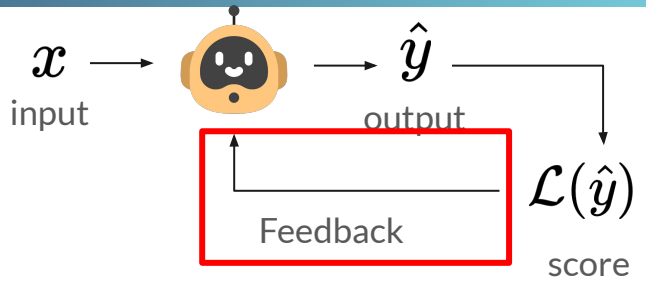
Now we know how to quantify the errors... but how can we **learn** from errors?

Changing **the weights** implies changing **the output...** and changing **the output** implies changing **the error**

The **loss function**, indeed, depends upon the **network parameters**

One can see each possible configuration as a point on a multi-dimensional surface called the **Loss Surface**





Learning from Errors

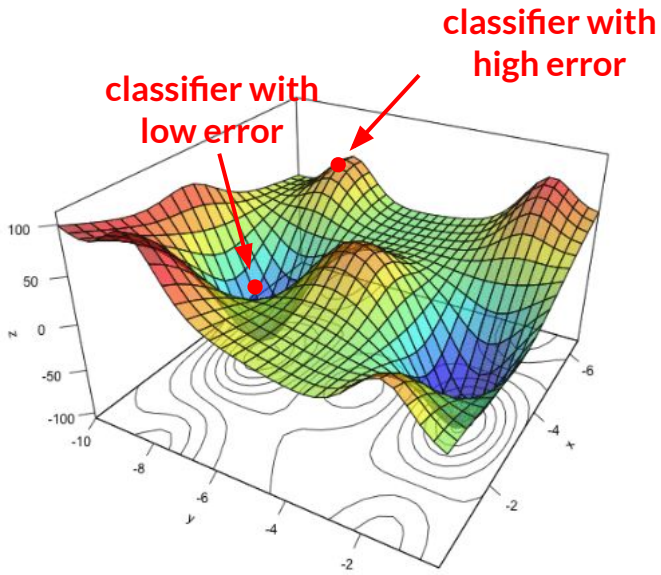
Formally, we define the **learning algorithm** as an optimization problem

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \quad \mathcal{L}(D, \theta)$$

Where θ^* is the set of all model's parameters (all weights and biases!) that lead to the optimum.

Some simple problems have a closed form solution, while for others we have to rely on **solvers** that find an approximate solution.

Following the gradient



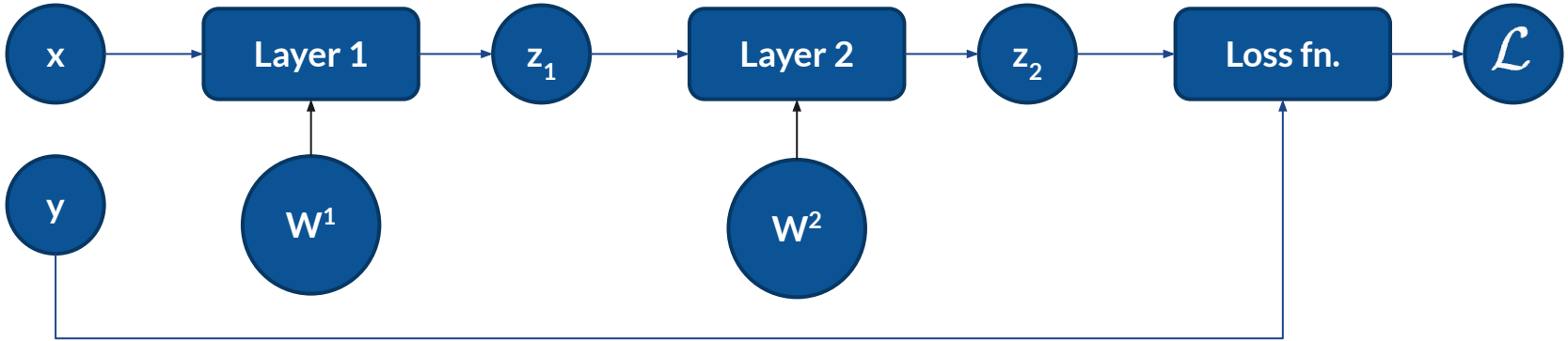
- The idea consists of "following" the **direction leading toward lower regions on the loss surface**
- The **gradient** of the loss function gives us information about the "**direction**" and the "**magnitude**" of the **slope of the surface**.

Backpropagation

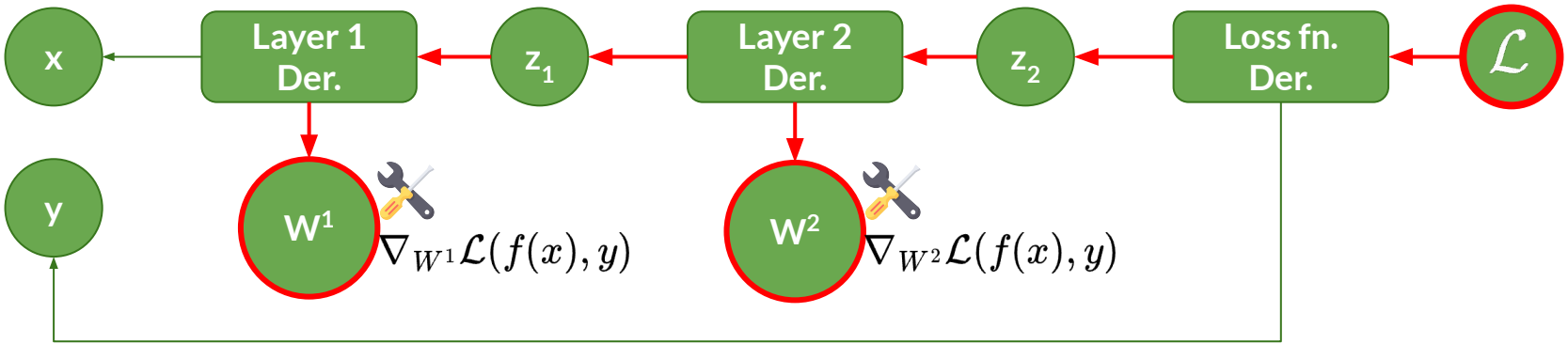
Remember the computational graph from wich we applied the chain rule to obtain the gradients of any function with respect to any variable?

This is exactly what we need to obtain the gradients for updating the parameters!

Forward pass



Backpropagation (Backward pass)



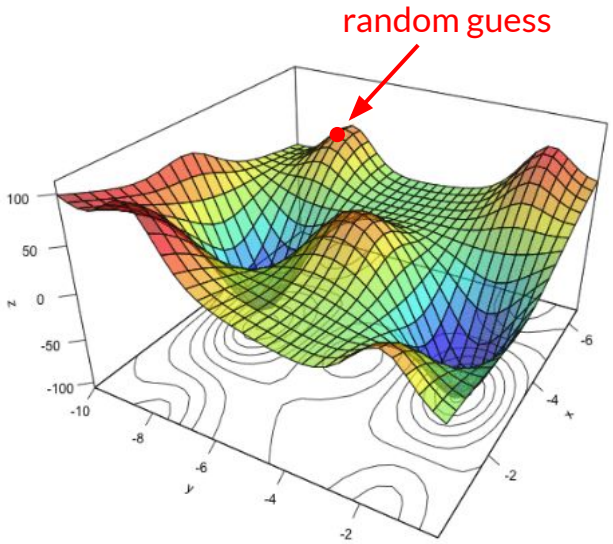
Gradient Descent

To update the parameters we use the gradient descend algorithm:

Gradient Descent

To update the parameters we use the gradient descend algorithm:

1. Randomly initialize the parameters

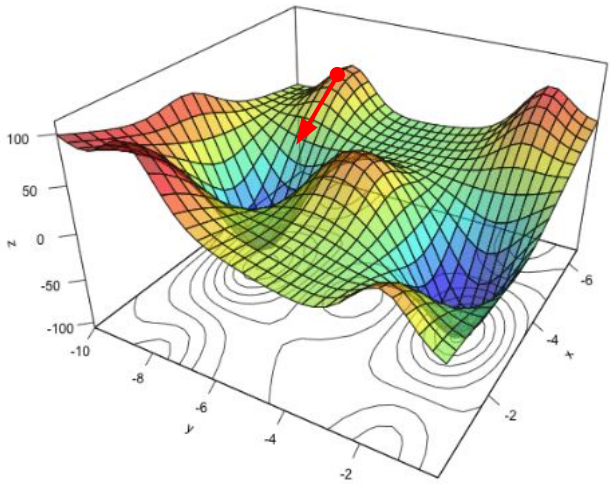


Gradient Descent

To update the parameters we use the gradient descend algorithm:

1. Randomly initialize the parameters
2. Compute the current gradient of the loss function

$$\nabla_W \mathcal{L}(f(x), y)$$

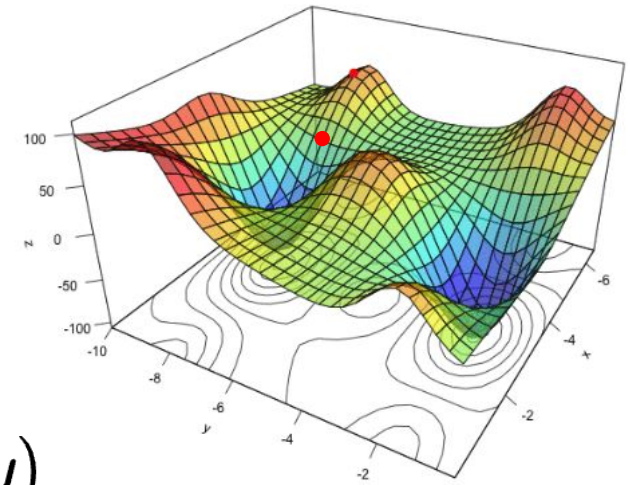


Gradient Descent

To update the parameters we use the gradient descend algorithm:

1. Randomly initialize the parameters
2. Compute the current gradient of the loss function
3. Take a small step opposite to the gradient direction

Updated parameters



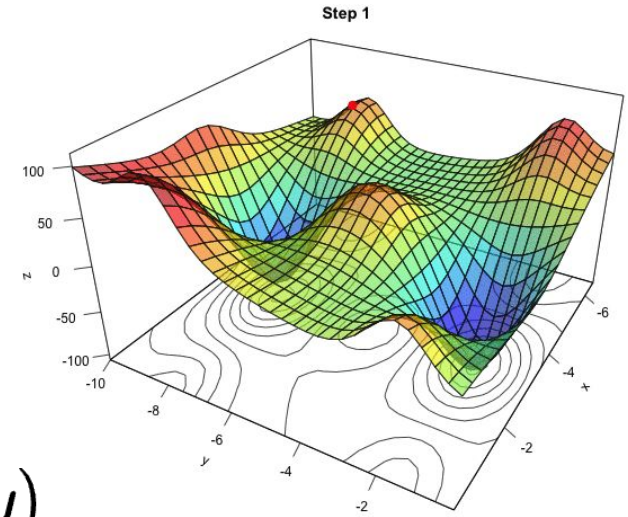
$$W^{(k+1)} = W^{(k)} - \alpha \nabla_W \mathcal{L}(f(x), y)$$

learning rate

Gradient Descent

To update the parameters we use the gradient descend algorithm:

1. Randomly initialize the parameters
2. Compute the current gradient of the loss function
3. Take a small step towards the gradient direction
4. Repeat from 2. until convergence



$$W^{(k+1)} = W^{(k)} - \alpha \nabla_W \mathcal{L}(f(x), y)$$

Choosing the right learning rate

if the learning rate is too small, we need many steps to reach convergence

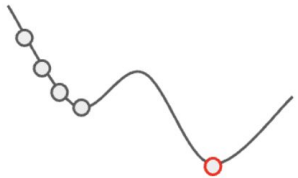
if the learning rate is too big, we might overshoot the optimum

the decay of the learning rate is also important
(how much we reduce the step size during the iterations)

Try out gradient descent with this online tool:
https://fa.bianp.net/teaching/2018/eecs227at/gradient_descent.html

Choosing the right loss function

Some loss function may also present some local minima, and our optimization can get stuck in them.

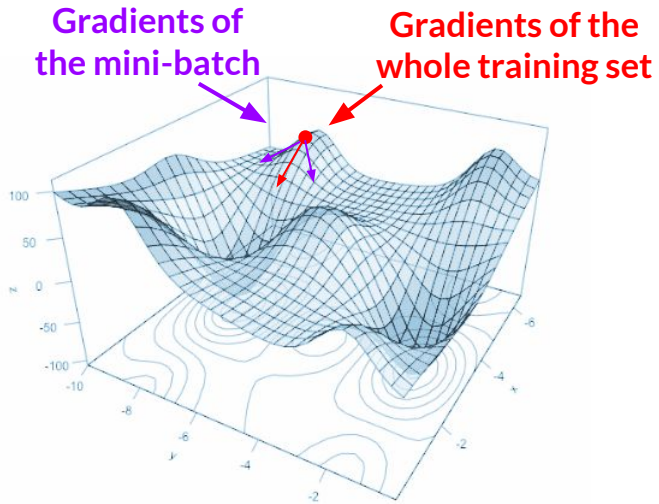


Stochastic Gradient Descent (SGD)

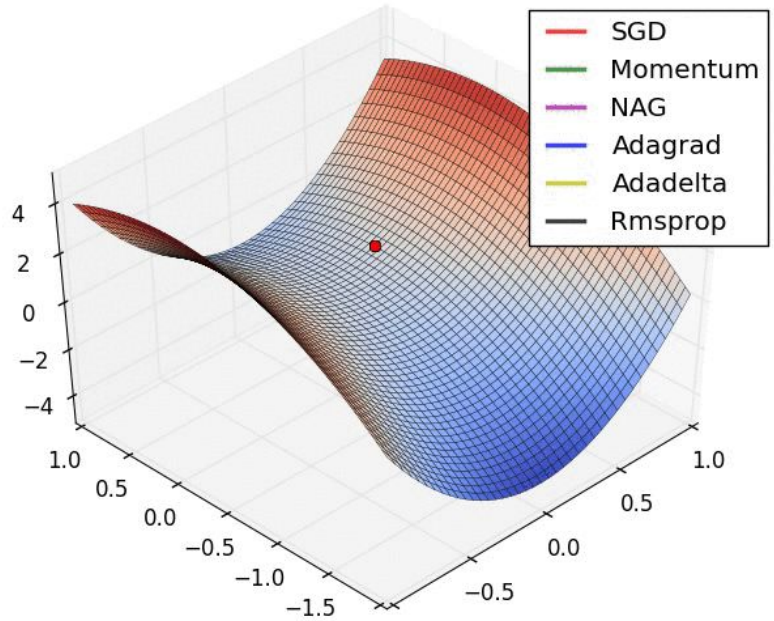
Sometime the dataset is too large to compute the gradient loss for all the training set.

Moreover, as we will see later, low error on the training set is only a necessary but not sufficient condition for obtaining good performance with never-seen examples.

A possible solution is to approximate the gradient with a random subset of samples that we call **mini-batch**.



Can we do better?



- Together with SGD, many optimizers exist that try to avoid the gradient getting stuck in local minima:
  - SGD + Momentum
  - RMSProp
  - Adam
- Standard deep-learning libraries implement many of these optimizers
- Empirically, one should **test many diverse optimizers to choose the best solution for the problem we are solving**.

PyTorch built-in optimizers

Mostly common optimization algorithm are already implemented inside the torch.optim package (link to the documentation)

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

We will see how to use them in a bit.

Performance Evaluation

Elements of performance evaluation

Ok, now we achieved low error on the training set, is this sufficient?

NO! We must prove that our model can **generalize** to sample that were not seen during training!

Usually we dedicate a part of the dataset, called **test set** for testing if this property is achieved

How to evaluate performance?

In case of binary classification problem (only two classes) we can represent the prediction with the **confusion matrix.**

- TP = True Positive
- TN = True Negative
- FP = False Positive
- FN = False Negative

		Predicted	
		Positive	Negative
True Class	Positive	TP	FN
	Negative	FP	TN

How to evaluate performance?

In case of binary classification problem (only two classes) we can represent the prediction with the **confusion matrix.**

- Correct = TP + TN
- Errors = FP + FN

		Predicted	
		Positive	Negative
True Class	Positive	TP	FN
	Negative	FP	TN

How to evaluate performance?

From these basic quantities we can define metrics that measure the error or characterize different kind of errors

$$\textbf{Accuracy} = \frac{(TP+TN)}{(TP+TN+FP+FN)} = \frac{\text{correct}}{\text{all samples}}$$

$$\textbf{Precision} = \frac{TP}{(TP+FP)} = \frac{\text{how many predicted and really positive}}{\text{all predicted positive}}$$

$$\textbf{Recall} = \frac{TP}{(TP+FN)} = \frac{\text{how many predicted and really positive}}{\text{all really positive}}$$

Overfitting

Neural Networks Suffers from high risk of Overfitting!

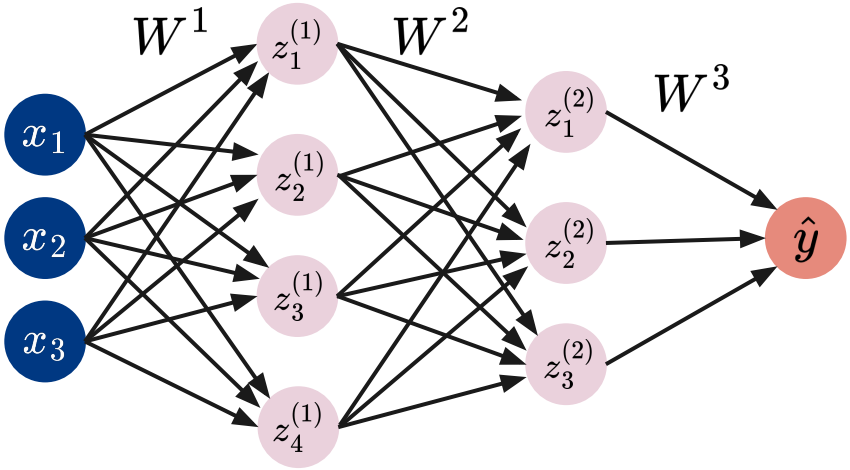
Regularization

Regularization term

$$\mathcal{L}(\hat{y}, y) = H(\hat{y}, y) + \overbrace{\|W\|}^{}_p$$

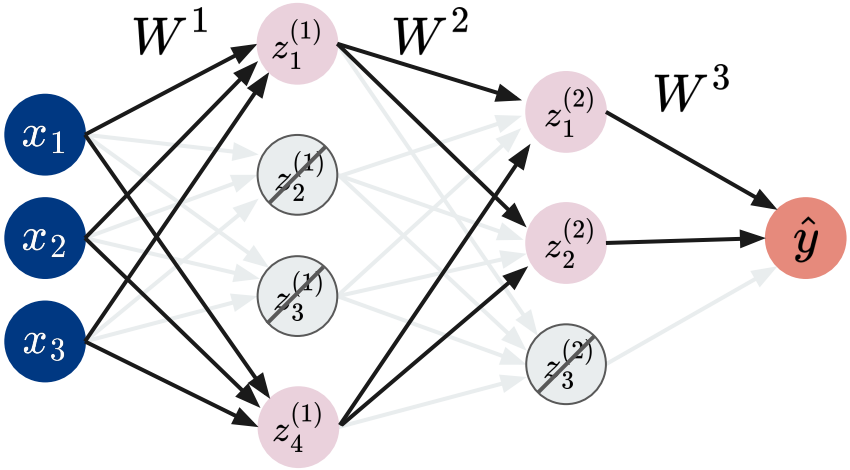
- An **L1** regularization (norm 1) **enforce sparsity** but **penalizes the magnitude** of the weights
- An **L2** regularization (norm 2) **penalizes sparsity** but **enforce the magnitude** of the weights
- In general, the desired effect is to diminish the **models' complexity**

Dropout Regularization



- In standard training we use all the weights during training…

Dropout Regularization

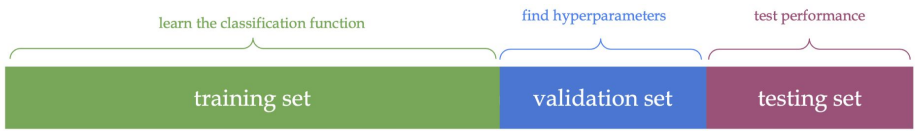


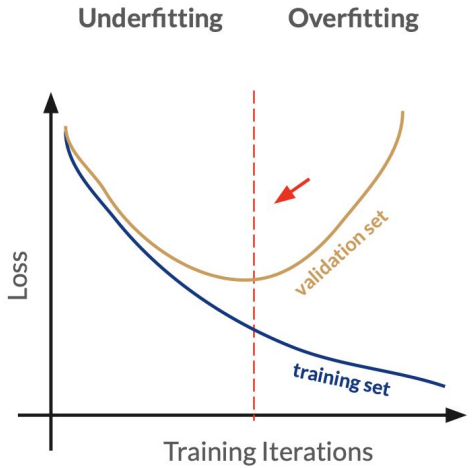
- When training with **dropout**, instead, we "**turn off**" some of the neurons **randomly at each iteration**, in order to reinforce each connection of the network

Early Stopping

We want to keep training until the error on the validation set stop decreasing

- Being the SGD algorithm iterative, we can force an "early stopping" if, at some iteration, the error on the validation increases instead of decreasing.



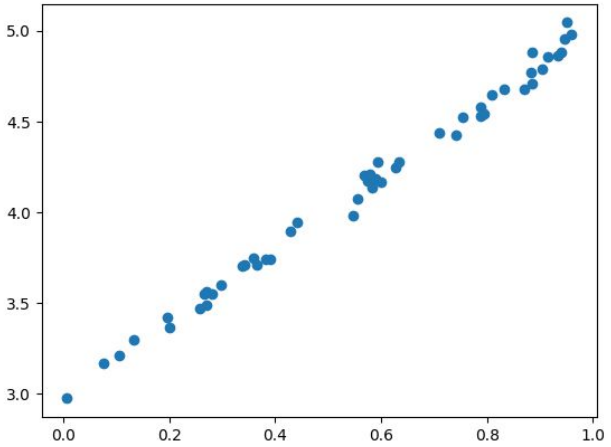


Implementing and Training a Neural Network with PyTorch

link for the colab

A Simple Regression Problem

```
n_samples = 50
random_state = 42
torch.manual_seed(random_state)
samples = torch.rand(size=(n_samples,))
noise = 0.05 * torch.randn(size=(n_samples,))
labels = 2*samples + 3     # underlying function to be learned
labels += noise
plt.scatter(samples, labels)
```



```python
def model(x, w, b):
    return w * x + b

def loss_fn(y_pred, y_true):
    squared_diffs = (y_pred - y_true)**2
    return squared_diffs.mean()

def plot_line(w, b, **kwargs):
    x_axis = torch.linspace(0, 1, 100)
    y_axis = w * x_axis + b
    plt.plot(x_axis.detach().numpy(),
             y_axis.detach().numpy(),
             color='r', **kwargs)

def plot_points(samples, labels):
    plt.scatter(samples, labels)

def plot_module(model, **kwargs):
    x_axis = torch.linspace(0, 1, 100).unsqueeze(dim=0).T
    y_axis = model(x_axis)
    plt.plot(x_axis.detach().numpy(),
             y_axis.detach().numpy(),
             color='r', **kwargs)
```

Linear model

Mean Squared Error Loss

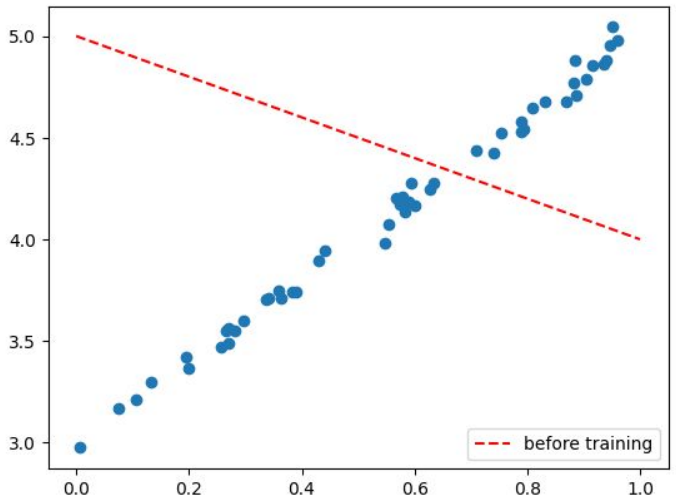
Utility functions to visualize the learned linear functions and the training examples

This will be needed if we have to use a generic model which is not necessarily linear

Initializing the parameters

```
1  torch.manual_seed(random_state)
2  params = torch.tensor([-1., 5.], requires_grad=True)
3  plot_points(samples, labels)
4  plot_line(*params, label='before training', linestyle='dashed')
5  plt.legend()
```

```
params =
torch.randn(size=(2,),requires_grad=True)
```



Gradient Descent

```
1  epochs = 1000
2  learning_rate = 1e-1
3
4  params = torch.tensor([-1., 5.], requires_grad=True)
5  plot_points(samples, labels)
6  plot_line(*params, label='before training', linestyle='dashed')
7
8  for epoch in range(epochs):
9      y_pred = model(samples, *params)
10     loss = loss_fn(y_pred, labels)
11     loss.backward()
12     with torch.no_grad():
13         params -= learning_rate * params.grad
14     params.grad.zero_()
15     if epoch % 100 == 0:
16         print("Epoch: %d, Loss %f" % (epoch, float(loss)))
17
18 plot_line(*params, label='after training')
19 plt.legend()
20 plt.show()
```

1 epochs = 1 iteration over the entire training set

Gradient Descent

```
1  epochs = 1000
2  learning_rate = 1e-1
3
4  params = torch.tensor([-1., 5.], requires_grad=True)
5  plot_points(samples, labels)
6  plot_line(*params, label='before training', linestyle='dashed')
7
8  for epoch in range(epochs):
9      y_pred = model(samples, *params)
10     loss = loss_fn(y_pred, labels)
11     loss.backward()
12     with torch.no_grad():
13         params -= learning_rate * params.grad
14     params.grad.zero_()
15     if epoch % 100 == 0:
16         print("Epoch: %d, Loss %f" % (epoch, float(loss)))
17
18 plot_line(*params, label='after training')
19 plt.legend()
20 plt.show()
```

setting the learning rate

Gradient Descent

```
1  epochs = 1000
2  learning_rate = 1e-1
3
4  params = torch.tensor([-1., 5.], requires_grad=True)
5  plot_points(samples, labels)
6  plot_line(*params, label='before training', linestyle='dashed')
7
8  for epoch in range(epochs):
9      y_pred = model(samples, *params)
10     loss = loss_fn(y_pred, labels)
11     loss.backward()
12     with torch.no_grad():
13         params -= learning_rate * params.grad
14     params.grad.zero_()
15     if epoch % 100 == 0:
16         print("Epoch: %d, Loss %f" % (epoch, float(loss)))
17
18 plot_line(*params, label='after training')
19 plt.legend()
20 plt.show()
```

Initialize the parameter and set requires_grad=True for computing their gradients

Gradient Descent

```
1  epochs = 1000
2  learning_rate = 1e-1
3
4  params = torch.tensor([-1., 5.], requires_grad=True)
5  plot_points(samples, labels)
6  plot_line(*params, label='before training', linestyle='dashed')
7
8  for epoch in range(epochs):
9      y_pred = model(samples, *params)
10     loss = loss_fn(y_pred, labels)
11     loss.backward()
12     with torch.no_grad():
13         params -= learning_rate * params.grad
14     params.grad.zero_()
15     if epoch % 100 == 0:
16         print("Epoch: %d, Loss %f" % (epoch, float(loss)))
17
18 plot_line(*params, label='after training')
19 plt.legend()
20 plt.show()
```

Iterate over the training set

Gradient Descent

```
1  epochs = 1000
2  learning_rate = 1e-1
3
4  params = torch.tensor([-1., 5.], requires_grad=True)
5  plot_points(samples, labels)
6  plot_line(*params, label='before training', linestyle='dashed')
7
8  for epoch in range(epochs):
9      y_pred = model(samples, *params)
10     loss = loss_fn(y_pred, labels)
11     loss.backward()
12     with torch.no_grad():
13         params -= learning_rate * params.grad
14     params.grad.zero_()
15     if epoch % 100 == 0:
16         print("Epoch: %d, Loss %f" % (epoch, float(loss)))
17
18 plot_line(*params, label='after training')
19 plt.legend()
20 plt.show()
```

Forward pass: obtain the output of the model and compute the loss by comparing output and label

Gradient Descent

```
1  epochs = 1000
2  learning_rate = 1e-1
3
4  params = torch.tensor([-1., 5.], requires_grad=True)
5  plot_points(samples, labels)
6  plot_line(*params, label='before training', linestyle='dashed')
7
8  for epoch in range(epochs):
9      y_pred = model(samples, *params)
10     loss = loss_fn(y_pred, labels)
11     loss.backward()
12     with torch.no_grad():
13         params -= learning_rate * params.grad
14     params.grad.zero_()
15     if epoch % 100 == 0:
16         print("Epoch: %d, Loss %f" % (epoch, float(loss)))
17
18 plot_line(*params, label='after training')
19 plt.legend()
20 plt.show()
```

Backward pass: backpropagate the gradients from the loss to the model's parameter

Gradient Descent

```
1  epochs = 1000
2  learning_rate = 1e-1
3
4  params = torch.tensor([-1., 5.], requires_grad=True)
5  plot_points(samples, labels)
6  plot_line(*params, label='before training', linestyle='dashed')
7
8  for epoch in range(epochs):
9      y_pred = model(samples, *params)
10     loss = loss_fn(y_pred, labels)
11     loss.backward()
12     with torch.no_grad():
13         params -= learning_rate * params.grad
14     params.grad.zero_()
15     if epoch % 100 == 0:
16         print("Epoch: %d, Loss %f" % (epoch, float(loss)))
17
18 plot_line(*params, label='after training')
19 plt.legend()
20 plt.show()
```

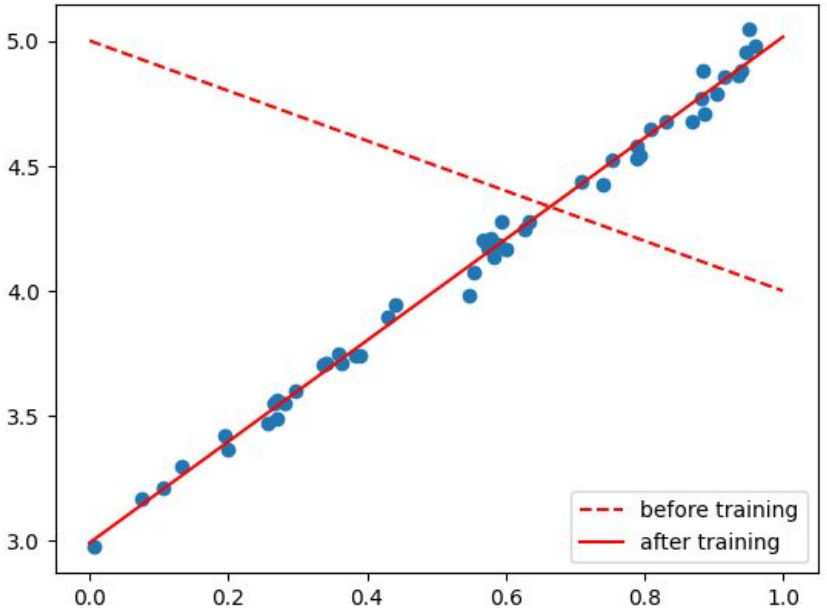
The no_grad block tells PyTorch to stop adding edges to the forward graph.
Inside this block we **update the parameters** using the information contained in params.grad attribute (filled after the .backward method)

Gradient Descent

```
1  epochs = 1000
2  learning_rate = 1e-1
3
4  params = torch.tensor([-1., 5.], requires_grad=True)
5  plot_points(samples, labels)
6  plot_line(*params, label='before training', linestyle='dashed')
7
8  for epoch in range(epochs):
9      y_pred = model(samples, *params)
10     loss = loss_fn(y_pred, labels)
11     loss.backward()
12     with torch.no_grad():
13         params -= learning_rate * params.grad
14     params.grad.zero_()
15     if epoch % 100 == 0:
16         print("Epoch: %d, Loss %f" % (epoch, float(loss)))
17
18 plot_line(*params, label='after training')
19 plt.legend()
20 plt.show()
```

Make the gradients equal to zero as we don't need to accumulate them

```
Epoch: 0, Loss 0.783028
Epoch: 100, Loss 0.078971
Epoch: 200, Loss 0.010152
Epoch: 300, Loss 0.003007
Epoch: 400, Loss 0.002265
Epoch: 500, Loss 0.002188
Epoch: 600, Loss 0.002180
Epoch: 700, Loss 0.002179
Epoch: 800, Loss 0.002179
Epoch: 900, Loss 0.002179
```



Using built-in modules

```
epochs = 1000
learning_rate = 1e-1

model = torch.nn.Linear(1, 1)
optim = torch.optim.SGD(model.parameters(), lr=learning_rate)
loss_fn = torch.nn.MSELoss()

plot_points(samples, labels)
plot_module(model, label='before training', linestyle='dashed')
for epoch in range(epochs):
    y_pred = model(samples.float().unsqueeze(1))
    loss = loss_fn(y_pred, labels.float().unsqueeze(1))
    loss.backward()
    optim.step()
    optim.zero_grad()
    if epoch % 100 == 0:
        print("Epoch: %d, Loss %f" % (epoch, float(loss)))
plot_module(model.to('cpu'), label='after training', linestyle='solid')
plt.legend()
plt.show()
```

Built-in Linear function with in/out size = 1/1
(1 scalar feature and 1 scalar value)
The parameters will be initialized randomly internally

Using built-in modules

```
epochs = 1000
learning_rate = 1e-1

model = torch.nn.Linear(1, 1)
optim = torch.optim.SGD(model.parameters(), lr=learning_rate)
loss_fn = torch.nn.MSELoss()

plot_points(samples, labels)
plot_module(model, label='before training', linestyle='dashed')
for epoch in range(epochs):
    y_pred = model(samples.float().unsqueeze(1))
    loss = loss_fn(y_pred, labels.float().unsqueeze(1))
    loss.backward()
    optim.step()
    optim.zero_grad()
    if epoch % 100 == 0:
        print("Epoch: %d, Loss %f" % (epoch, float(loss)))
plot_module(model.to('cpu'), label='after training', linestyle='solid')
plt.legend()
plt.show()
```

Use the MSELoss class from the torch.nn package

Using built-in modules

```
epochs = 1000
learning_rate = 1e-1

model = torch.nn.Linear(1, 1)
optim = torch.optim.SGD(model.parameters(), lr=learning_rate)
loss_fn = torch.nn.MSELoss()

plot_points(samples, labels)
plot_module(model, label='before training', linestyle='dashed')
for epoch in range(epochs):
    y_pred = model(samples.float().unsqueeze(1))
    loss = loss_fn(y_pred, labels.float().unsqueeze(1))
    loss.backward()
    optim.step()
    optim.zero_grad()
    if epoch % 100 == 0:
        print("Epoch: %d, Loss %f" % (epoch, float(loss)))
plot_module(model.to('cpu'), label='after training', linestyle='solid')
plt.legend()
plt.show()
```

With the .parameters() method we pass all the learnable parameters to the SGD optimizer class.
The optim class also require the learning rate as inputs as it is needed inside its internal update rule

Using built-in modules

```
epochs = 1000
learning_rate = 1e-1

model = torch.nn.Linear(1, 1)
optim = torch.optim.SGD(model.parameters(), lr=learning_rate)
loss_fn = torch.nn.MSELoss()

plot_points(samples, labels)
plot_module(model, label='before training', linestyle='dashed')
for epoch in range(epochs):
    y_pred = model(samples.float().unsqueeze(1))
    loss = loss_fn(y_pred, labels.float().unsqueeze(1))
    loss.backward()
    optim.step()
    optim.zero_grad()
    if epoch % 100 == 0:
        print("Epoch: %d, Loss %f" % (epoch, float(loss)))
plot_module(model.to('cpu'), label='after training', linestyle='solid')
plt.legend()
plt.show()
```

After the backward, the **.step** method tells the optimizer to update the received parameters (during init).
After that, optim.zer_grad() take care of zeroing all gradients parameters

Using built-in modules

```
epochs = 1000
learning_rate = 1e-1

model = torch.nn.Linear(1, 1)
optim = torch.optim.SGD(model.parameters(), lr=learning_rate)
loss_fn = torch.nn.MSELoss()

plot_points(samples, labels)
plot_module(model, label='before training', linestyle='dashed')
for epoch in range(epochs):
    y_pred = model(samples.float().unsqueeze(1))
    loss = loss_fn(y_pred, labels.float().unsqueeze(1))
    loss.backward()
    optim.step()
    optim.zero_grad()
    if epoch % 100 == 0:
        print("Epoch: %d, Loss %f" % (epoch, float(loss)))
plot_module(model.to('cpu'), label='after training', linestyle='solid')
plt.legend()
plt.show()
```

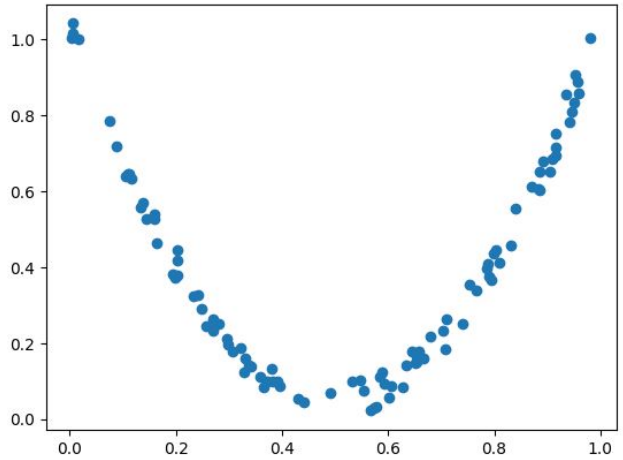
What's this 'unsqueeze(1)'???

nn.Module always expects the first dimension to refer the sample index.

Here we just take 1D vectors and tell PyTorch to add a new dimension of size one in the 2nd position (indices in python starts at zero, thus dim=1 is second dimension)

Non linearities in the data

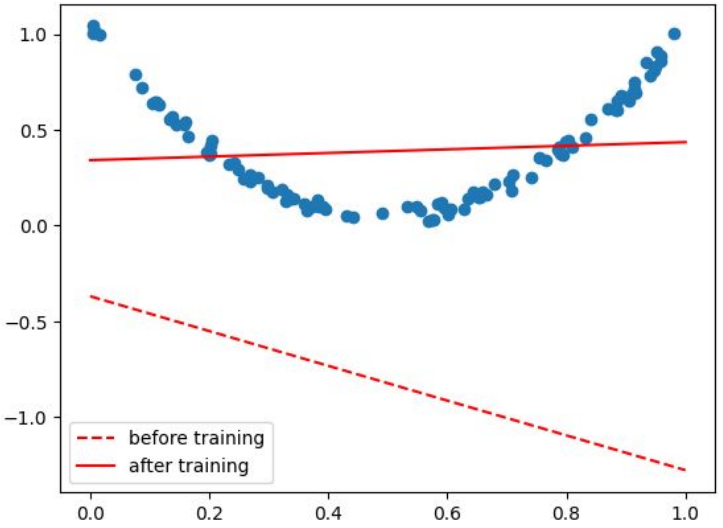
```
1  torch.manual_seed(42)
2  samples = torch.rand(100)
3  noise = torch.rand(samples.shape) * 0.1
4  labels = ((2 * samples - 1) ** 2) + noise
5
6  plt.scatter(samples, labels)
```



Using the Linear model…

```
Epoch: 0, Loss 1.673574
Epoch: 100, Loss 0.080875
Epoch: 200, Loss 0.079673
Epoch: 300, Loss 0.079591
Epoch: 400, Loss 0.079586
Epoch: 500, Loss 0.079586
Epoch: 600, Loss 0.079586
Epoch: 700, Loss 0.079586
Epoch: 800, Loss 0.079586
Epoch: 900, Loss 0.079586
```

```python
epochs = 1000
learning_rate = 1e-1

model = torch.nn.Linear(1, 1)
optim = torch.optim.SGD(model.parameters(), lr=learning_rate)
loss_fn = torch.nn.MSELoss()

plot_points(samples, labels)
plot_module(model, label='before training', linestyle='dashed')
for epoch in range(epochs):
    y_pred = model(samples.float().unsqueeze(1))
    loss = loss_fn(y_pred, labels.float().unsqueeze(1))
    loss.backward()
    optim.step()
    optim.zero_grad()
    if epoch % 100 == 0:
        print("Epoch: %d, Loss %f" % (epoch, float(loss)))
plot_module(model.to('cpu'), label='after training', linestyle='solid')
plt.legend()
plt.show()
```



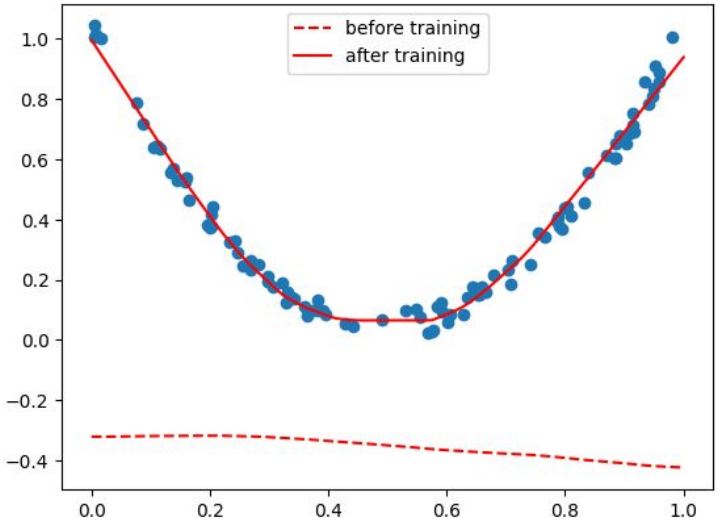
Using an MLP

```python
class MLP(torch.nn.Module):
    def __init__(self, input_dim=1, output_dim=1, hidden_dim=10):
        super().__init__()
        self.linear1 = torch.nn.Linear(input_dim, hidden_dim)
        self.linear2 = torch.nn.Linear(hidden_dim, output_dim)
        self.activation = torch.nn.ReLU()

    def forward(self, x):
        out_hidden = self.linear1(x)
        out_hidden = self.activation(out_hidden)
        out = self.linear2(out_hidden)
        return out

```

Using an MLP

```
1  epochs = 1000
2  learning_rate = 1e-1
3
4  model = MLP(input_dim=1, output_dim=1, hidden_dim=100)
5  optim = torch.optim.SGD(model.parameters(), lr=learning_rate)
6  loss_fn = torch.nn.MSELoss()
7
8  plot_points(samples, labels)
9  plot_module(model, label='before training', linestyle='dashed')
10 for epoch in range(epochs):
11     y_pred = model(samples.float().unsqueeze(1))
12     loss = loss_fn(y_pred, labels.float().unsqueeze(1))
13     loss.backward()
14     optim.step()
15     optim.zero_grad()
16     if epoch % 100 == 0:
17         print("Epoch: %d, Loss %f" % (epoch, float(loss)))
18 plot_module(model.to('cpu'), label='after training', linestyle='solid')
19 plt.legend()
20 plt.show()
```

```
Epoch: 0, Loss 0.646976
Epoch: 100, Loss 0.026540
Epoch: 200, Loss 0.005861
Epoch: 300, Loss 0.002618
Epoch: 400, Loss 0.001986
Epoch: 500, Loss 0.001705
Epoch: 600, Loss 0.001508
Epoch: 700, Loss 0.001357
Epoch: 800, Loss 0.001244
Epoch: 900, Loss 0.001161
```



End of part 2

Summary:

- Linear models for classification and regression
- Multi-class problems
- Neural Networks
- Loss functions
- Optimization with Gradient Descent
- Performance evaluation, overfitting, validation