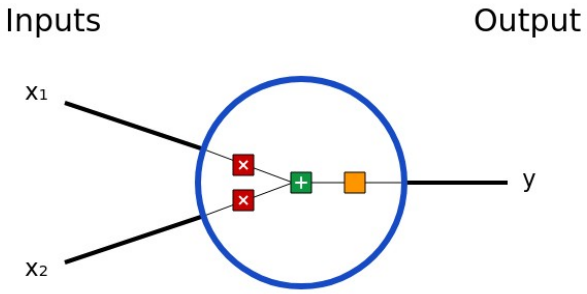
Lab 0 - Machine Learning - une introduction

Ce laboratoire est une introduction aux réseaux de neurones, une explication simple de leur fonctionnement et comment en implémenter un à partir de zéro en Python.

0.1 Un Neurone

Tout d'abord, nous devons parler des neurones, l'unité de base d'un réseau de neurones. Un **neurone** prend des entrées, fait des calculs avec elles et produit une sortie.

Voici à quoi ressemble un neurone à 2 entrées :



`(x1*w1)+(x2*w2)+b`

3 choses se passent ici. Tout d'abord, chaque **entrée** est multipliée par un **poids** :

x1 : x1∗w1 et x2 : x2∗w2

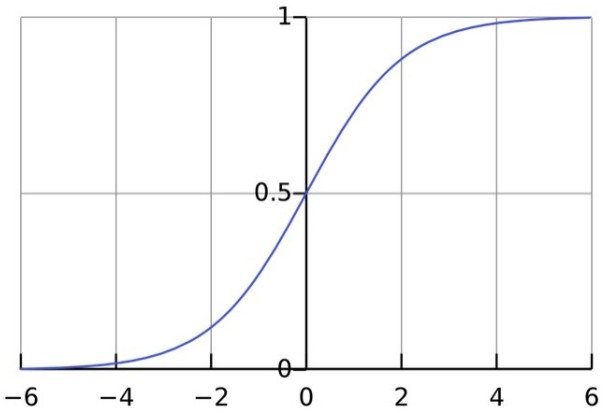
Ensuite, toutes les entrées pondérées sont additionnées avec un **biais** *b* :

(x1∗w1)+(x2∗w2)+b

Enfin, la somme passe par une **fonction d'activation** :

y=f(x1∗w1+x2∗w2+b)

La fonction d'activation est utilisée pour transformer une entrée illimitée en une sortie qui a une forme agréable et prévisible. Une **fonction d'activation** couramment utilisée est la fonction **sigmoïde** :



La **fonction sigmoïde** ne sort que des nombres dans la plage (**0,1**). Vous pouvez considérer cela comme une compression (−∞,+∞) en (0,1) - grands nombres négatifs devient ~0, et les grands nombres positifs deviennent ~1.

0.1.1 Un exemple simple¶

Supposons que nous ayons un neurone à 2 entrées qui utilise la fonction d'activation **sigmoïde** et possède les paramètres suivants :

w=[0,1] et **b=4**

w=[0,1] est juste une façon d'écrire **w1=0, w2=1** sous forme vectorielle.

Maintenant, donnons au neurone une entrée de **x=[2,3]**. Nous utiliserons le **produit scalaire** pour écrire les choses de manière plus concise :

$$(w \cdot x)+b=((w1*x1)+(w2*x2))+b=0*2+1*3+4=7$$

$$y=f(w \cdot x+b)=f(7)=0.999$$

Le neuron dérivé sort **0,9990** étant donné les entrées **x=[2,3]**.

C'est ça! Ce processus de transmission d'entrées pour obtenir une sortie est connu sous le nom de ***feedforward***.

0.1.2 Codage d'un neurone¶

Il est temps d'implanter un neurone ! Nous utiliserons **NumPy**, une bibliothèque informatique populaire et puissante pour Python, pour nous aider à faire des mathématiques :

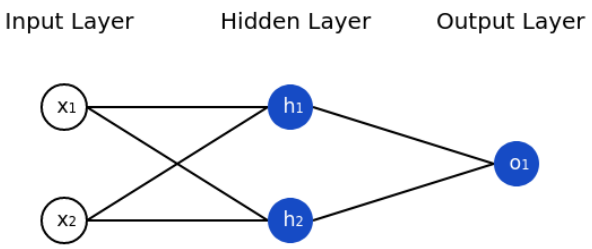
```
import numpy as np

def sigmoid(x):
  # Our activation function: f(x) = 1 / (1 + e^(-x))
  return 1 / (1 + np.exp(-x))

class Neuron:
  def __init__(self, weights, bias):
    self.weights = weights
    self.bias = bias

  def feedforward(self, inputs):
    # Weight inputs, add bias, then use the activation function
    total = np.dot(self.weights, inputs) + self.bias
    return sigmoid(total)

weights = np.array([0, 1]) # w1 = 0, w2 = 1
bias = 4                   # b = 4
n = Neuron(weights, bias)

x = np.array([2, 3])       # x1 = 2, x2 = 3
print(n.feedforward(x))    # 0.9990889488055994
0.9990889488055994
```

Reconnaîssez ces chiffres ? C'est l'exemple que nous venons de faire à la main!

Nous obtenons la même réponse de **0,999**.

0.2 Combinaison de neurones dans un réseau de neurones (NN)¶

Un réseau de neurones n'est rien de plus qu'un groupe de neurones connectés entre eux. Voici à quoi pourrait ressembler un simple réseau de neurones :



Ce réseau a 2 entrées, une couche cachée avec 2 neurones (**h1** et **h2**), et une couche de sortie avec 1 neurone (**o1**). Notez que les entrées de **o1** sont les sorties de **h1** et **h2**- c'est ce qui en fait un réseau.

Une couche cachée est une couche entre la couche d'entrée (première) et la couche de sortie (dernière). Il peut y avoir plusieurs calques cachés !

0.2.1 Un exemple : Feedforward¶

Utilisons le réseau illustré ci-dessus et supposons que tous les neurones ont les mêmes poids **w=[0,1]**, le même biais **b=0** et le même fonction d'activation sigmoïde. Soit **h1,h2,o1** les sorties des neurones qu'ils représentent.

Que se passe-t-il si on passe l'entrée **x=[2,3]** ?

$$h1=h2=f(w\cdot x+b)=f((0*2)+(1*3)+0)=f(3)=0.9526$$

$$o1=f(w\cdot[h1,h2]+b)=f((0*h1)+(1*h2)+0)=f(0.9526)=0.7216$$

La sortie du réseau neuronal pour l'entrée **x=[2,3]** est de **0,72160**.

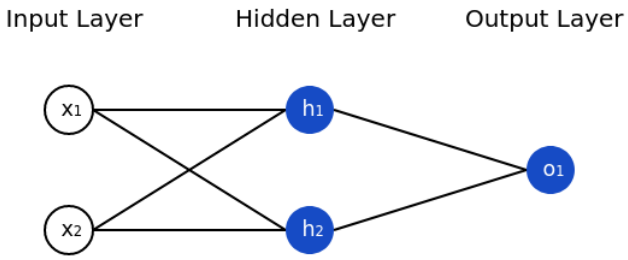
Assez simple, non?

Un réseau de neurones peut avoir n'importe quel nombre de couches avec n'importe quel nombre de neurones dans ces couches. L'idée de base reste la même : transmettre la ou les entrées à travers les neurones du réseau pour obtenir la ou les sorties à la fin.

Par souci de simplicité, nous continuerons à utiliser le réseau illustré ci-dessus pour le reste de ce laboratoire.

0.2.2 Codage d'un réseau de neurones : feedforward¶

Implémentons le **feedforward** pour notre réseau de neurones. Voici à nouveau l'image du réseau pour référence. Notons que les 3 neurones sont initialisés avec les mêmes poids: **w=[0,1]** et le même biais: **b=0**.




```
import numpy as np
```

```
# ... code from previous section here

class OurNeuralNetwork:
  '''
  A neural network with:
    - 2 inputs
    - a hidden layer with 2 neurons (h1, h2)
    - an output layer with 1 neuron (o1)
  Each neuron has the same weights and bias:
    - w = [0, 1]
    - b = 0
  '''
  def __init__(self):  # no arguments, only internal values
    weights = np.array([0, 1])
    bias = 0
    # The Neuron class here is from the previous section
    self.h1 = Neuron(weights, bias)
    self.h2 = Neuron(weights, bias)
    self.o1 = Neuron(weights, bias)

  def feedforward(self, x):
    out_h1 = self.h1.feedforward(x)
    out_h2 = self.h2.feedforward(x)
    # The inputs for o1 are the outputs from h1 and h2
    out_o1 = self.o1.feedforward(np.array([out_h1, out_h2]))

    return out_o1

network = OurNeuralNetwork()
x = np.array([2, 3])
print(network.feedforward(x)) # 0.7216325609518421
```

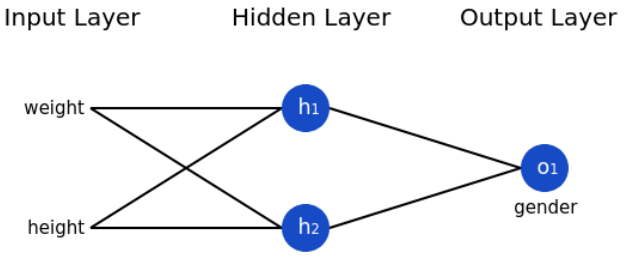
```
0.7216325609518421
```

Nous avons à nouveau **0,7216** ! On dirait que ça marche.

0.3 Formation d'un réseau de neurones, partie 1

Disons que nous avons les mesures suivantes :

Name	Weight (lb)	Height (in)	Gender
Alice	133	65	F
Bob	160	72	M
Charlie	152	70	M
Diana	120	60	F



0.3.1 Perte

Avant de former notre réseau, nous avons d'abord besoin d'un moyen de quantifier à quel point il se comporte « bien » afin qu'il puisse essayer de faire « mieux ». C'est ça la **perte**.

Nous utiliserons la perte **erreur quadratique moyenne** (**MSE**) :

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_{true} - y_{pred})^2$$

Décomposons ceci :

- **n** est le nombre d'échantillons, qui est de **4** (Alice, Bob, Charlie, Diana).
- **y** représente la variable prédite, qui est le sexe.
- **ytruey** est la vraie valeur de la variable (la "bonne réponse"). Par exemple, **ytruey** pour Alice serait 1 (**Femme**).
- **ypredy** est la valeur prédite de la variable. C'est quelle que soit la sortie de notre réseau.
- **(ytrue−ypred)^2** est connu comme l'erreur quadratique. Notre fonction de perte prend simplement la moyenne de toutes les erreurs au carré (d'où le nom **erreur quadratique moyenne**).

Plus nos prédictions sont bonnes, plus notre perte sera faible !

Former un réseau => essayer de minimiser sa perte.

0.3.2 Un exemple de calcul de perte¶

Disons que notre réseau affiche toujours **0** - en d'autres termes, il est sûr que **tous les humains sont des hommes** . Quelle serait notre perte ?

Name	y_{true}	y_{pred}	$(y_{true} - y_{pred})^2$
Alice	1	0	1
Bob	0	0	0
Charlie	0	0	0
Diana	1	0	1

$$\text{MSE} = \frac{1}{4}(1 + 0 + 0 + 1) = \boxed{0.5}$$

0.3.3 Code : Perte MSE

Voici un code pour calculer la perte pour nous:

```
import numpy as np

def mse_loss(y_true, y_pred):
  # y_true and y_pred are numpy arrays of the same length.
  return ((y_true - y_pred) ** 2).mean()

y_true = np.array([1, 0, 0, 1])
y_pred = np.array([0, 0, 0, 0])

print(mse_loss(y_true, y_pred)) # 0.5
0.5
```

C'est parfait. Continuons !

0.4 Formation d'un réseau de neurones, partie 2

Nous avons maintenant un objectif clair : minimiser la perte du réseau de neurones.

Nous savons que nous pouvons modifier les pondérations et les biais du réseau pour influencer ses prévisions, mais comment le faire de manière à réduire les pertes ?

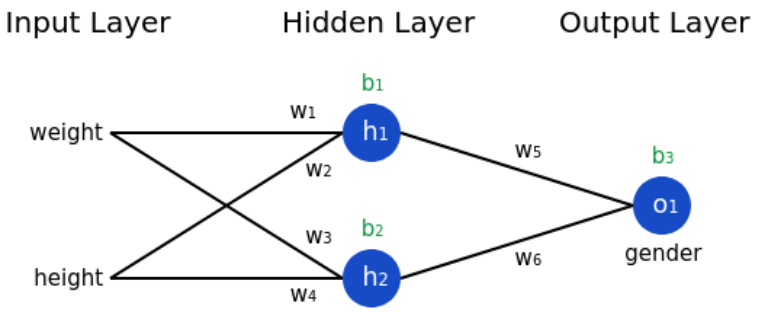
Pour simplifier, supposons que nous n'ayons qu'Alice dans notre ensemble de données :

Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1

Dans ce cas, la perte d'erreur quadratique moyenne n'est que l'**erreur quadratique d'Alice = 1** :

$$\begin{aligned} \text{MSE} &= \frac{1}{1}\sum_{i=1}^{1}(y_{true} - y_{pred})^2 \\ &= (y_{true} - y_{pred})^2 \\ &= (1 - y_{pred})^2 \end{aligned}$$

Une autre façon de penser à la perte est en fonction des poids et des biais. Étiquetons chaque poids et biais de notre réseau :



Ensuite, nous pouvons écrire la perte sous la forme d'une fonction multivariable :

$$L(w_1, w_2, w_3, w_4, w_5, w_6, b_1, b_2, b_3)$$

Imaginez que nous voulions modifier **w1**. Comment la perte **L** changerait-elle si on changeait **w1** ?

C'est une question à laquelle la dérivée partielle **∂L/∂w1** peut répondre. Comment le calcule-t-on ?

Pour commencer, réécrivons la dérivée partielle en termes de **∂ypred/∂w1** à la place :

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial w_1}$$

Nous pouvons calculer **∂L∂ypred** car nous avons calculé **L=(1−ypred)^2** ci-dessus :

$$\frac{\partial L}{\partial y_{pred}} = \frac{\partial (1 - y_{pred})^2}{\partial y_{pred}} = \boxed{-2(1 - y_{pred})}$$

Voyons maintenant quoi faire avec **∂ypred∂w1**. Tout comme avant, soit **h1,h2,o1** les sorties des neurones qu'ils représentent. Puis:

$$y_{pred} = o_1 = f(w_5 h_1 + w_6 h_2 + b_3)$$

f is the sigmoid activation function, remember?

Puisque **w1** n'affecte que **h1** (pas **h2**), nous pouvons écrire:

$$\frac{\partial y_{pred}}{\partial w_1} = \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

$$\frac{\partial y_{pred}}{\partial h_1} = \boxed{w_5 * f'(w_5 h_1 + w_6 h_2 + b_3)}$$

Nous faisons la même chose pour **∂h1∂w1** :

$$h_1 = f(w_1 x_1 + w_2 x_2 + b_1)$$

$$\frac{\partial h_1}{\partial w_1} = \boxed{x_1 * f'(w_1 x_1 + w_2 x_2 + b_1)}$$

x1 ici est le **poids** et **x2** est la **taille**.

C'est la deuxième fois que nous voyons **f'(x)** (le dérivé de la **fonction sigmoïde**) maintenant ! Dérivons-le :

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

$$f'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = f(x) * (1 - f(x))$$

Nous utiliserons cette belle forme pour **f'(x)** plus tard.

Succés. Nous avons réussi à décomposer **∂L∂w1** en plusieurs parties que nous pouvons calculer :

$$\boxed{\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}}$$

Ce système de calcul des dérivées partielles en travaillant à rebours est connu sous le nom de **rétropropagation**, ou «**backprop**».

C'était beaucoup de symboles - ce n'est pas grave si vous êtes encore un peu confus. Faisons un exemple pour voir cela en action !

0.4.1 Exemple : Calcul de la dérivée partielle¶

Nous allons continuer à prétendre que seule Alice est dans notre ensemble de données :

Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1

Initialisons tous les poids à **1** et tous les biais à **0**. Si nous effectuons un *feedforward* pass à travers le réseau, nous obtenons :

$$\begin{aligned} h_1 &= f(w_1 x_1 + w_2 x_2 + b_1) \\ &= f(-2 + -1 + 0) \\ &= 0.0474 \end{aligned}$$

$$h_2 = f(w_3 x_1 + w_4 x_2 + b_2) = 0.0474$$

$$\begin{aligned} o_1 &= f(w_5 h_1 + w_6 h_2 + b_3) \\ &= f(0.0474 + 0.0474 + 0) \\ &= 0.524 \end{aligned}$$

Le réseau génère **ypred=0,524**, ce qui ne favorise pas fortement Male (**0**) ou Female (**1**). Calculons **∂L∂w1** :

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

$$\begin{aligned} \frac{\partial L}{\partial y_{pred}} &= -2(1 - y_{pred}) \\ &= -2(1 - 0.524) \\ &= -0.952 \end{aligned}$$

$$\begin{aligned} \frac{\partial y_{pred}}{\partial h_1} &= w_5 * f'(w_5 h_1 + w_6 h_2 + b_3) \\ &= 1 * f'(0.0474 + 0.0474 + 0) \\ &= f(0.0948) * (1 - f(0.0948)) \\ &= 0.249 \end{aligned}$$

$$\begin{aligned} \frac{\partial h_1}{\partial w_1} &= x_1 * f'(w_1 x_1 + w_2 x_2 + b_1) \\ &= -2 * f'(-2 + -1 + 0) \\ &= -2 * f(-3) * (1 - f(-3)) \\ &= -0.0904 \end{aligned}$$

$$\begin{aligned} \frac{\partial L}{\partial w_1} &= -0.952 * 0.249 * -0.0904 \\ &= \boxed{0.0214} \end{aligned}$$

Nous l'avons fait! Cela nous indique que si nous augmentions **w1**, **L** augmenterait un tout petit peu en conséquence.

0.4.2 Entraînement : Descente de gradient stochastique

Nous avons tous les outils dont nous avons besoin pour former un réseau de neurones maintenant !

Nous utiliserons un algorithme d'optimisation appelé **descente de gradient stochastique (SGD)** qui nous indique comment modifier nos poids et biais pour minimiser les pertes. Il s'agit essentiellement de cette équation de mise à jour :

$$w_1 \leftarrow w_1 - \eta \frac{\partial L}{\partial w_1}$$

η est une constante appelée **taux d'apprentissage** qui contrôle la vitesse à laquelle nous entraînons le réseau.

Tout ce que nous faisons est de soustraire **ηðLðw1** de **w1** :

1. Si **ðLðw1** est positif, w1 diminuera, ce qui fait **diminuer L**.
2. Si **ðLðw1** est négatif, w1 augmentera, ce qui fera **diminuer L**.

Si nous faisons cela pour chaque poids et biais du réseau, la perte diminuera lentement et notre réseau s'améliorera.

Notre processus de formation ressemblera à ceci :

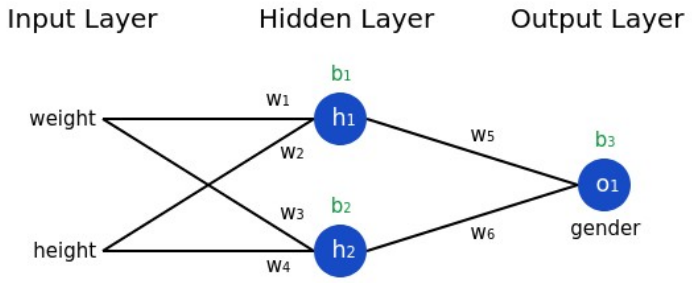
1. Choisissez un échantillon de notre ensemble de données. C'est ce qui en fait une descente de gradient stochastique - nous n'opérons que sur un échantillon à la fois.
2. Calculer toutes les dérivées partielles de la perte par rapport aux poids ou aux biais (par exemple **ðLðw1**, **ðLðw2**, etc.).
3. Utilisez l'équation de mise à jour pour mettre à jour chaque poids et biais.
4. Revenez à l'étape 1.

Voyons-le en action !

0.5 Code : Un réseau de neurones complet

Il est enfin temps de mettre en place un réseau de neurones complet :

Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1
Bob	25	6	0
Charlie	17	4	0
Diana	-15	-6	1




```
import numpy as np

def sigmoid(x):
  # Sigmoid activation function: f(x) = 1 / (1 + e^(-x))
  return 1 / (1 + np.exp(-x))

def deriv_sigmoid(x):
  # Derivative of sigmoid: f'(x) = f(x) * (1 - f(x))
  fx = sigmoid(x)
  return fx * (1 - fx)

def mse_loss(y_true, y_pred):
  # y_true and y_pred are numpy arrays of the same length.
  return ((y_true - y_pred) ** 2).mean()

class OurNeuralNetwork:
  '''
  A neural network with:
    - 2 inputs
    - a hidden layer with 2 neurons (h1, h2)
    - an output layer with 1 neuron (o1)

  *** DISCLAIMER ***:
  The code below is intended to be simple and educational, NOT optimal.
  Real neural net code looks nothing like this. DO NOT use this code.
  Instead, read/run it to understand how this specific network works.
  '''
  def __init__(self):
    # Weights
    self.w1 = np.random.normal()
    self.w2 = np.random.normal()
 self.w3 = np.random.normal()
    self.w4 = np.random.normal()
    self.w5 = np.random.normal()
    self.w6 = np.random.normal()

    # Biases
    self.b1 = np.random.normal()
    self.b2 = np.random.normal()
    self.b3 = np.random.normal()

  def feedforward(self, x):
    # x is a numpy array with 2 elements.
    h1 = sigmoid(self.w1 * x[0] + self.w2 * x[1] + self.b1)
    h2 = sigmoid(self.w3 * x[0] + self.w4 * x[1] + self.b2)
```

```
    o1 = sigmoid(self.w5 * h1 + self.w6 * h2 + self.b3)
    return o1

  def train(self, data, all_y_trues):
    '''
    - data is a (n x 2) numpy array, n = # of samples in the dataset.
    - all_y_trues is a numpy array with n elements.
      Elements in all_y_trues correspond to those in data.
    '''
    learn_rate = 0.1
    epochs = 1000 # number of times to loop through the entire dataset

    for epoch in range(epochs):
      for x, y_true in zip(data, all_y_trues):
        # --- Do a feedforward (we'll need these values later)
        sum_h1 = self.w1 * x[0] + self.w2 * x[1] + self.b1
        h1 = sigmoid(sum_h1)

        sum_h2 = self.w3 * x[0] + self.w4 * x[1] + self.b2
        h2 = sigmoid(sum_h2)

        sum_o1 = self.w5 * h1 + self.w6 * h2 + self.b3
        o1 = sigmoid(sum_o1)
        y_pred = o1

        # --- Calculate partial derivatives.
        # --- Naming: d_L_d_w1 represents "partial L / partial w1"
        d_L_d_ypred = -2 * (y_true - y_pred)

        # Neuron o1
        d_ypred_d_w5 = h1 * deriv_sigmoid(sum_o1)
        d_ypred_d_w6 = h2 * deriv_sigmoid(sum_o1)
        d_ypred_d_b3 = deriv_sigmoid(sum_o1)

        d_ypred_d_h1 = self.w5 * deriv_sigmoid(sum_o1)
        d_ypred_d_h2 = self.w6 * deriv_sigmoid(sum_o1)

        # Neuron h1
        d_h1_d_w1 = x[0] * deriv_sigmoid(sum_h1)
        d_h1_d_w2 = x[1] * deriv_sigmoid(sum_h1)
        d_h1_d_b1 = deriv_sigmoid(sum_h1)

        # Neuron h2
        d_h2_d_w3 = x[0] * deriv_sigmoid(sum_h2)
        d_h2_d_w4 = x[1] * deriv_sigmoid(sum_h2)
        d_h2_d_b2 = deriv_sigmoid(sum_h2)

        # --- Update weights and biases
        # Neuron h1
        self.w1 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_w1
        self.w2 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_w2
        self.b1 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_b1

        # Neuron h2
        self.w3 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_w3
        self.w4 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_w4
        self.b2 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_b2

        # Neuron o1
        self.w5 -= learn_rate * d_L_d_ypred * d_ypred_d_w5
        self.w6 -= learn_rate * d_L_d_ypred * d_ypred_d_w6
        self.b3 -= learn_rate * d_L_d_ypred * d_ypred_d_b3

      # --- Calculate total loss at the end of each epoch
      if epoch % 10 == 0:
        y_preds = np.apply_along_axis(self.feedforward, 1, data)
        loss = mse_loss(all_y_trues, y_preds)
        print("Epoch %d loss: %.3f" % (epoch, loss))

# Define dataset
data = np.array([
  [-2, -1],  # Alice
  [25, 6],   # Bob
  [17, 4],   # Charlie
  [-15, -6], # Diana
])
```

```
all_y_trues = np.array([
  1, # Alice
  0, # Bob
  0, # Charlie
  1, # Diana
])

# Train our neural network!
network = OurNeuralNetwork()
network.train(data, all_y_trues)
```

Nous pouvons maintenant utiliser le réseau pour prédire les sexes :

```
# Make some predictions
emily = np.array([-7, -3]) # 128 pounds, 63 inches
frank = np.array([20, 2])  # 155 pounds, 68 inches
print("Emily: %.3f" % network.feedforward(emily)) # 0.951 - F
print("Frank: %.3f" % network.feedforward(frank)) # 0.039 - M
Emily: 0.964
Frank: 0.039
```

0.6 Résumé¶

C'est fait! Un petit récapitulatif de ce que nous avons fait :

1. Nous avons introduit les neurones, les éléments constitutifs des réseaux de neurones.
2. On a utilisé la fonction d'activation sigmoïde dans nos neurones.
3. Les réseaux de neurones ne sont que des neurones connectés entre eux.
4. La création d'un réseaux - jeu de données avec le poids et la taille comme entrées (ou caractéristiques) et le sexe comme sortie (ou étiquette).
5. Nous avons pris la connaissance des fonctions de perte et la perte d'erreur quadratique moyenne (MSE).
6. Nous avons réalisé que la formation d'un réseau ne fait que minimiser sa perte.
7. La rétro-propagation est utilisée pour calculer les dérivées partielles.
8. Nous avons utilisé la descente de gradient stochastique (SGD) pour entraîner notre réseau.
9. Nous avons fait une prédiction