Handwritten digits recognition - Maxine LAMA | Marwan MEZROUI

Introduction

Les neurones peuvent être modélisés mathématiquement pour créer des réseaux de neurones artificiels.

Ce sont des systèmes de traitement de l'information qui peuvent être utilisés pour effectuer des tâches telles que la classification, la reconnaissance de formes, la prédiction, etc.

Les neurones artificiels sont généralement organisés en couches, chaque couche étant composée de plusieurs neurones qui communiquent entre eux.

Les réseaux de neurones sont entraînés en utilisant des données d'entraînement pour ajuster les poids et les biais des neurones. l'optimisation est utilisée pour minimiser une fonction de coût.

Ils ont leur importance dans le domaine de l'apprentissage en profondeur. Les réseaux de neurones profonds, qui sont des réseaux de neurones avec plusieurs couches, ont été utilisés pour résoudre des problèmes de plus en plus complexes, tels que la reconnaissance de la parole et la vision par ordinateur.

Le but de ce projet est de faire usage du livre intitulé "Neural Networks and Deep learning" écrit par Michael Nielsen afin d'avoir une première approche et compréhension des réseaux de neurones et de leur apprentissage profond. En effet, grâce à ce livre nous étudions les fondamentaux des réseaux de neurones.

Présentement, nous cherchons à mettre en place un réseau capable de déterminer la valeur des chiffres écrits de manière manuscrite. Pour ce faire, nous allons expérimenter en utilisant le code fournit par Michael Nielsen.

In [1]:
```
# import générique
import numpy as np
```

Perceptron de base

On va donc dans un premier temps expérimenter avec le fichier nommé "network.py". Ce dernier met en place un perceptron dit de base, c'est-à-dire possédant 3 couches (input layer, hidden layer et output layer). Pour entraîner notre réseau de neurones, nous ferons usage de la base de données fournit par MNIST.

In [2]:

```
############################################################
# BEGIN : expérimentation du perceptron dit "de base" #
############################################################

# import module needed for experiment
import network
import mnist_loader

# settings var
training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
net = network.Network([784, 30, 10]) # Network([input_layer, hidden_layer, outpu

# training our network
# SGD(training_data, epoch, batch_size, learning_rate, test_data)

print("############# BEGIN - EXPERIMENT ON LEARNING RATE #############\n\n")
print("------------- BEGIN : FIRST TEST [epoch: 30, batch_size: 10, learning_rat
net.SGD(training_data, 30, 10, 0.01, test_data=test_data)
print("------------- END : FIRST TEST [epoch: 30, batch_size: 10, learning_rate:

print("------------- BEGIN : SECOND TEST [epoch: 30, batch_size: 10, learning_ra
net = network.Network([784, 30, 10])
net.SGD(training_data, 30, 10, 100.0, test_data=test_data)
print("------------- END : SECOND TEST [epoch: 30, batch_size: 10, learning_rate

print("------------- BEGIN : THIRD TEST [epoch: 30, batch_size: 10, learning_rat
net = network.Network([784, 30, 10])
net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
print("------------- END : THIRD TEST [epoch: 30, batch_size: 10, learning_rate:

print("############# END - EXPERIMENT ON LEARNING RATE #############")
```

```
############## BEGIN - EXPERIMENT ON LEARNING RATE ##############


------------- BEGIN : FIRST TEST [epoch: 30, batch_size: 10, learning_rate: 0.0
1] -------------

Epoch 0: 1734 / 10000
Epoch 1: 2297 / 10000
Epoch 2: 2621 / 10000
Epoch 3: 2872 / 10000
Epoch 4: 3053 / 10000
Epoch 5: 3262 / 10000
Epoch 6: 3580 / 10000
Epoch 7: 3886 / 10000
Epoch 8: 4129 / 10000
Epoch 9: 4343 / 10000
Epoch 10: 4514 / 10000
Epoch 11: 4669 / 10000
Epoch 12: 4811 / 10000
Epoch 13: 4942 / 10000
Epoch 14: 5052 / 10000
Epoch 15: 5168 / 10000
Epoch 16: 5245 / 10000
Epoch 17: 5318 / 10000
Epoch 18: 5380 / 10000
Epoch 19: 5488 / 10000
Epoch 20: 5606 / 10000
Epoch 21: 5717 / 10000
Epoch 22: 5833 / 10000
Epoch 23: 6006 / 10000
Epoch 24: 6188 / 10000
Epoch 25: 6372 / 10000
Epoch 26: 6592 / 10000
Epoch 27: 6814 / 10000
Epoch 28: 7072 / 10000
Epoch 29: 7240 / 10000
------------- END : FIRST TEST [epoch: 30, batch_size: 10, learning_rate: 0.01]
-------------

------------- BEGIN : SECOND TEST [epoch: 30, batch_size: 10, learning_rate: 10
0.0] -------------

Epoch 0: 1032 / 10000
Epoch 1: 1032 / 10000
Epoch 2: 1032 / 10000
Epoch 3: 1032 / 10000
Epoch 4: 1032 / 10000
Epoch 5: 1032 / 10000
Epoch 6: 1032 / 10000
Epoch 7: 1032 / 10000
Epoch 8: 1032 / 10000
Epoch 9: 1032 / 10000
Epoch 10: 1032 / 10000
Epoch 11: 1032 / 10000
Epoch 12: 1032 / 10000
Epoch 13: 1032 / 10000
Epoch 14: 1032 / 10000
Epoch 15: 1032 / 10000
Epoch 16: 1032 / 10000
Epoch 17: 1032 / 10000
Epoch 18: 1032 / 10000
Epoch 19: 1032 / 10000
Epoch 20: 1032 / 10000
```

```
Epoch 21: 1032 / 10000
Epoch 22: 1032 / 10000
Epoch 23: 1032 / 10000
Epoch 24: 1032 / 10000
Epoch 25: 1032 / 10000
Epoch 26: 1032 / 10000
Epoch 27: 1032 / 10000
Epoch 28: 1032 / 10000
Epoch 29: 1032 / 10000
------------- END : SECOND TEST [epoch: 30, batch_size: 10, learning_rate: 100.
0] -------------


------------- BEGIN : THIRD TEST [epoch: 30, batch_size: 10, learning_rate: 3.0]
-------------

Epoch 0: 9044 / 10000
Epoch 1: 9197 / 10000
Epoch 2: 9279 / 10000
Epoch 3: 9331 / 10000
Epoch 4: 9359 / 10000
Epoch 5: 9363 / 10000
Epoch 6: 9402 / 10000
Epoch 7: 9373 / 10000
Epoch 8: 9413 / 10000
Epoch 9: 9399 / 10000
Epoch 10: 9419 / 10000
Epoch 11: 9457 / 10000
Epoch 12: 9450 / 10000
Epoch 13: 9439 / 10000
Epoch 14: 9444 / 10000
Epoch 15: 9476 / 10000
Epoch 16: 9466 / 10000
Epoch 17: 9471 / 10000
Epoch 18: 9470 / 10000
Epoch 19: 9467 / 10000
Epoch 20: 9478 / 10000
Epoch 21: 9485 / 10000
Epoch 22: 9458 / 10000
Epoch 23: 9479 / 10000
Epoch 24: 9478 / 10000
Epoch 25: 9478 / 10000
Epoch 26: 9462 / 10000
Epoch 27: 9485 / 10000
Epoch 28: 9482 / 10000
Epoch 29: 9476 / 10000
------------- END : THIRD TEST [epoch: 30, batch_size: 10, learning_rate: 3.0]
-------------

############## END - EXPERIMENT ON LEARNING RATE ##############
```

Observation : learning rate

Dans le perceptron de base, On a d'abord expérimenté l'influence du paramètre "learning rate".

Dans un premier temps, on a expérimenté avec un learning rate assez faible, égal à 0.01. 0n a pu observer que notre réseau a évoluer de 1734 à 7240, ce qui indique qu'il a beaucoup appris.

Du coup, on a décidé d'utiliser une valeur plus élevée, soit 100.0. À notre grande surprise, cela s'est révélé contre-productif. En effet, nos résultats stagnent à 1032.

Par conséquent, on a décidé de réduire la valeur de ce paramètre tout en ayant une valeur supérieure à 0.01 en le fixant à 3.0. On observe alors des résultats très satisfaisants, dépassant systématiquement les 9000.

In [7]:
```
# redefine var
training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
net = network.Network([784, 30, 10]) # Network([input_layer, hidden_layer, outpu

print("############## BEGIN - EXPERIMENT ON BATCH SIZE ##############\n\n")
print("------------- BEGIN : FIRST TEST [epoch: 30, batch_size: 10, learning_rat
net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
print("------------- END : FIRST TEST [epoch: 30, batch_size: 10, learning_rate:

print("------------- BEGIN : SECOND TEST [epoch: 30, batch_size: 1000, learning_
net = network.Network([784, 30, 10])
net.SGD(training_data, 30, 1000, 3.0, test_data=test_data)
print("------------- END : SECOND TEST [epoch: 30, batch_size: 1000, learning_ra

print("------------- BEGIN : THIRD TEST [epoch: 30, batch_size: 100, learning_ra
net = network.Network([784, 30, 10])
net.SGD(training_data, 30, 100, 3.0, test_data=test_data)
print("------------- END : THIRD TEST [epoch: 30, batch_size: 100, learning_rate
print("############## END - EXPERIMENT ON BATCH SIZE ##############\n\n")

#########################################################
# END : expérimentation du perceptron dit "de base"   #
#########################################################
```

```
############## BEGIN - EXPERIMENT ON BATCH SIZE #############


------------- BEGIN : FIRST TEST [epoch: 30, batch_size: 10, learning_rate: 3.0]
-------------

Epoch 0: 9107 / 10000
Epoch 1: 9229 / 10000
Epoch 2: 9335 / 10000
Epoch 3: 9346 / 10000
Epoch 4: 9420 / 10000
Epoch 5: 9385 / 10000
Epoch 6: 9422 / 10000
Epoch 7: 9449 / 10000
Epoch 8: 9427 / 10000
Epoch 9: 9432 / 10000
Epoch 10: 9461 / 10000
Epoch 11: 9436 / 10000
Epoch 12: 9457 / 10000
Epoch 13: 9486 / 10000
Epoch 14: 9492 / 10000
Epoch 15: 9466 / 10000
Epoch 16: 9447 / 10000
Epoch 17: 9467 / 10000
Epoch 18: 9473 / 10000
Epoch 19: 9488 / 10000
Epoch 20: 9479 / 10000
Epoch 21: 9493 / 10000
Epoch 22: 9480 / 10000
Epoch 23: 9475 / 10000
Epoch 24: 9497 / 10000
Epoch 25: 9485 / 10000
Epoch 26: 9500 / 10000
Epoch 27: 9488 / 10000
Epoch 28: 9510 / 10000
Epoch 29: 9480 / 10000
------------- END : FIRST TEST [epoch: 30, batch_size: 10, learning_rate: 3.0]
-------------

------------- BEGIN : SECOND TEST [epoch: 30, batch_size: 1000, learning_rate:
3.0] -------------

Epoch 0: 3181 / 10000
Epoch 1: 4590 / 10000
Epoch 2: 5586 / 10000
Epoch 3: 6274 / 10000
Epoch 4: 6682 / 10000
Epoch 5: 7246 / 10000
Epoch 6: 7596 / 10000
Epoch 7: 7797 / 10000
Epoch 8: 7979 / 10000
Epoch 9: 8107 / 10000
Epoch 10: 8203 / 10000
Epoch 11: 8290 / 10000
Epoch 12: 8370 / 10000
Epoch 13: 8428 / 10000
Epoch 14: 8472 / 10000
Epoch 15: 8516 / 10000
Epoch 16: 8557 / 10000
Epoch 17: 8589 / 10000
Epoch 18: 8615 / 10000
Epoch 19: 8641 / 10000
Epoch 20: 8662 / 10000
```

```
Epoch 21: 8692 / 10000
Epoch 22: 8711 / 10000
Epoch 23: 8725 / 10000
Epoch 24: 8747 / 10000
Epoch 25: 8768 / 10000
Epoch 26: 8778 / 10000
Epoch 27: 8799 / 10000
Epoch 28: 8816 / 10000
Epoch 29: 8825 / 10000
------------- END : SECOND TEST [epoch: 30, batch_size: 1000, learning_rate: 3.
0] -------------

------------- BEGIN : THIRD TEST [epoch: 30, batch_size: 100, learning_rate: 3.
0] -------------

Epoch 0: 7299 / 10000
Epoch 1: 8557 / 10000
Epoch 2: 8733 / 10000
Epoch 3: 8845 / 10000
Epoch 4: 8942 / 10000
Epoch 5: 8994 / 10000
Epoch 6: 9040 / 10000
Epoch 7: 9079 / 10000
Epoch 8: 9115 / 10000
Epoch 9: 9136 / 10000
Epoch 10: 9158 / 10000
Epoch 11: 9179 / 10000
Epoch 12: 9198 / 10000
Epoch 13: 9220 / 10000
Epoch 14: 9247 / 10000
Epoch 15: 9256 / 10000
Epoch 16: 9280 / 10000
Epoch 17: 9285 / 10000
Epoch 18: 9287 / 10000
Epoch 19: 9298 / 10000
Epoch 20: 9313 / 10000
Epoch 21: 9316 / 10000
Epoch 22: 9330 / 10000
Epoch 23: 9331 / 10000
Epoch 24: 9343 / 10000
Epoch 25: 9342 / 10000
Epoch 26: 9339 / 10000
Epoch 27: 9351 / 10000
Epoch 28: 9352 / 10000
Epoch 29: 9366 / 10000
------------- END : THIRD TEST [epoch: 30, batch_size: 100, learning_rate: 3.0]
-------------

############# END - EXPERIMENT ON BATCH SIZE #############
```

Observation : batch size

Par la suite, on a effectué des tests en modifiant la valeur du paramètre appelé "batch_size".

On a testé ce paramètre avec 3 valeurs : 10, 100 et 1000. On a remarqué que plus la valeur était importante, plus le réseau commetait des erreurs au début, mais apprenait plus rapidement par la suite.

Perceptron à deux couches de poids

In [13]:
```
######################################################################
# BEGIN : expérimentation du perceptron à deux couches de poids #
######################################################################

# settings var
training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
net = network.Network([784, 30, 30, 10]) # Network([input_layer, hidden_layer_1,

# training our network
# SGD(training_data, epoch, batch_size, learning_rate, test_data)

print("############# BEGIN - EXPERIMENT [epoch: 30, batch_size: 10, learning_rat
print("------------- BEGIN : FIRST TEST 30 neurons for each hidden layer -------
net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
print("Exactitude des données de test : {:.2f}%".format(net.evaluate(test_data)
print("\n------------- END : FIRST TEST -------------\n")

print("------------- BEGIN : SECOND TEST 300 neurons & 300 neurons -------------
net = network.Network([784, 300, 300, 10])
net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
print("Exactitude des données de test : {:.2f}%".format(net.evaluate(test_data)
print("\n------------- END : SECOND TEST -------------\n")

print("------------- BEGIN : THIRD TEST 100 neurons & 30 neurons -------------\n
net = network.Network([784, 100, 30, 10])
net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
print("Exactitude des données de test : {:.2f}%".format(net.evaluate(test_data)
print("\n------------- END : THIRD TEST -------------\n")

print("------------- BEGIN : FOURTH TEST 30 neurons & 100 neurons -------------\
net = network.Network([784, 30, 100, 10])
net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
print("Exactitude des données de test : {:.2f}%".format(net.evaluate(test_data)
print("\n------------- END : FOURTH TEST -------------\n")

print("############# END - EXPERIMENT [epoch: 30, batch_size: 10, learning_rate:
######################################################################
# END : expérimentation du perceptron à deux couches de poids   #
######################################################################
```

```
############## BEGIN - EXPERIMENT [epoch: 30, batch_size: 10, learning_rate: 3.0]
##############


------------- BEGIN : FIRST TEST 30 neurons for each hidden layer -------------

Epoch 0: 9044 / 10000
Epoch 1: 9255 / 10000
Epoch 2: 9288 / 10000
Epoch 3: 9388 / 10000
Epoch 4: 9428 / 10000
Epoch 5: 9429 / 10000
Epoch 6: 9432 / 10000
Epoch 7: 9469 / 10000
Epoch 8: 9503 / 10000
Epoch 9: 9489 / 10000
Epoch 10: 9518 / 10000
Epoch 11: 9499 / 10000
Epoch 12: 9481 / 10000
Epoch 13: 9490 / 10000
Epoch 14: 9533 / 10000
Epoch 15: 9482 / 10000
Epoch 16: 9523 / 10000
Epoch 17: 9527 / 10000
Epoch 18: 9532 / 10000
Epoch 19: 9550 / 10000
Epoch 20: 9517 / 10000
Epoch 21: 9553 / 10000
Epoch 22: 9540 / 10000
Epoch 23: 9553 / 10000
Epoch 24: 9528 / 10000
Epoch 25: 9559 / 10000
Epoch 26: 9539 / 10000
Epoch 27: 9538 / 10000
Epoch 28: 9528 / 10000
Epoch 29: 9541 / 10000
Exactitude des données de test : 95.41%

------------- END : FIRST TEST -------------

------------- BEGIN : SECOND TEST 300 neurons & 300 neurons -------------

Epoch 0: 2538 / 10000
Epoch 1: 3567 / 10000
Epoch 2: 4889 / 10000
Epoch 3: 6735 / 10000
Epoch 4: 6962 / 10000
Epoch 5: 7164 / 10000
Epoch 6: 7767 / 10000
Epoch 7: 7808 / 10000
Epoch 8: 7804 / 10000
Epoch 9: 7822 / 10000
Epoch 10: 7799 / 10000
Epoch 11: 7799 / 10000
Epoch 12: 7851 / 10000
Epoch 13: 7842 / 10000
Epoch 14: 7832 / 10000
Epoch 15: 8004 / 10000
Epoch 16: 7985 / 10000
Epoch 17: 8112 / 10000
Epoch 18: 8043 / 10000
Epoch 19: 7928 / 10000
Epoch 20: 7861 / 10000
```

```
Epoch 21: 8031 / 10000
Epoch 22: 7960 / 10000
Epoch 23: 7932 / 10000
Epoch 24: 7919 / 10000
Epoch 25: 8001 / 10000
Epoch 26: 7975 / 10000
Epoch 27: 7941 / 10000
Epoch 28: 7968 / 10000
Epoch 29: 7988 / 10000
Exactitude des données de test : 79.88%

------------- END : SECOND TEST -------------

------------- BEGIN : THIRD TEST 100 neurons & 30 neurons -------------

Epoch 0: 9158 / 10000
Epoch 1: 9317 / 10000
Epoch 2: 9430 / 10000
Epoch 3: 9426 / 10000
Epoch 4: 9475 / 10000
Epoch 5: 9515 / 10000
Epoch 6: 9510 / 10000
Epoch 7: 9566 / 10000
Epoch 8: 9585 / 10000
Epoch 9: 9570 / 10000
Epoch 10: 9577 / 10000
Epoch 11: 9594 / 10000
Epoch 12: 9581 / 10000
Epoch 13: 9595 / 10000
Epoch 14: 9606 / 10000
Epoch 15: 9606 / 10000
Epoch 16: 9627 / 10000
Epoch 17: 9588 / 10000
Epoch 18: 9596 / 10000
Epoch 19: 9608 / 10000
Epoch 20: 9608 / 10000
Epoch 21: 9598 / 10000
Epoch 22: 9591 / 10000
Epoch 23: 9601 / 10000
Epoch 24: 9604 / 10000
Epoch 25: 9629 / 10000
Epoch 26: 9623 / 10000
Epoch 27: 9631 / 10000
Epoch 28: 9628 / 10000
Epoch 29: 9628 / 10000
Exactitude des données de test : 96.28%

------------- END : THIRD TEST -------------

------------- BEGIN : FOURTH TEST 30 neurons & 100 neurons -------------

Epoch 0: 7506 / 10000
Epoch 1: 8251 / 10000
Epoch 2: 8339 / 10000
Epoch 3: 9260 / 10000
Epoch 4: 9316 / 10000
Epoch 5: 9410 / 10000
Epoch 6: 9366 / 10000
Epoch 7: 9407 / 10000
Epoch 8: 9444 / 10000
Epoch 9: 9458 / 10000
Epoch 10: 9454 / 10000
Epoch 11: 9439 / 10000
```

```
Epoch 12: 9434 / 10000
Epoch 13: 9446 / 10000
Epoch 14: 9464 / 10000
Epoch 15: 9528 / 10000
Epoch 16: 9495 / 10000
Epoch 17: 9494 / 10000
Epoch 18: 9527 / 10000
Epoch 19: 9527 / 10000
Epoch 20: 9529 / 10000
Epoch 21: 9530 / 10000
Epoch 22: 9515 / 10000
Epoch 23: 9499 / 10000
Epoch 24: 9537 / 10000
Epoch 25: 9528 / 10000
Epoch 26: 9514 / 10000
Epoch 27: 9522 / 10000
Epoch 28: 9516 / 10000
Epoch 29: 9553 / 10000
Exactitude des données de test : 95.53%

------------- END : FOURTH TEST -------------

############# END - EXPERIMENT [epoch: 30, batch_size: 10, learning_rate: 3.0] #
############
```

Observation : influence du nombre de neurones

On a donc mis en place cette fois-ci un perceptron à deux couches de poids. Ainsi, l'input layer va être traitée deux fois par deux couches de neurones au lieu d'être traitée une seul fois. Cela signifie qu'elle va être filtrée une première fois, nous fournissant une première sortie. Cette sortie va ensuite être filtrée de nouveau pour nous donner le résultat final.

Dans un premier temps, on a cherché à expérimenter sur l'influence de la quantité de neurones à travers les tests 1 et 2. On observe qu'avoir beaucoup de neurones ne garantit pas pour autant les meilleurs résultats. En effet, avec seulement 30 neurones dans chaque couche, on obtient une exactitude de 95,41% tandis qu'avec 300 neurones par couche, on obtient une exactitude inférieur, égale à 79,88%.

Ensuite, on a cherché à tester la variation du nombre de neurones d'une couche à l'autre. On constate qu'avoir une couche avec beaucoup de neurones suivie d'une couche avec un nombre de neurones moins important nous permet d'obtenir de meilleurs résultats, à savoir 96,28% contre 95,53%.

Ainsi, il est intéressant de faire varier le nombre de neurones d'une couche à l'autre, en veillant à ne pas avoir des couches avec un nombre de neurones trop important.

Augmentation du nombe de couches de poids

In [5]:

```
#########################################################################
# BEGIN : expérimentation du perceptron à plusieurs couches de poids #
#########################################################################

# import module needed for experiment
import network
import mnist_loader

# settings var
training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
net = network.Network([784, 30, 30, 30, 10]) # Network([input_layer, hidden_laye

# training our network
# SGD(training_data, epoch, batch_size, learning_rate, test_data)

print("############# BEGIN - EXPERIMENT [epoch: 30, batch_size: 10, learning_rat
print("------------- BEGIN : FIRST TEST 3 layers -------------\n")
net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
print("Exactitude des données de test : {:.2f}%".format(net.evaluate(test_data)
print("------------- END : FIRST TEST -------------\n")

print("------------- BEGIN : SECOND TEST 10 layers -------------\n")
net = network.Network([784, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 10])
net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
print("Exactitude des données de test : {:.2f}%".format(net.evaluate(test_data)
print("------------- END : SECOND TEST -------------\n")

print("------------- BEGIN : THIRD TEST 40 layers -------------\n")
net = network.Network([784,
                       30, 30, 30, 30, 30, 30, 30, 30, 30, 30,
                       30, 30, 30, 30, 30, 30, 30, 30, 30, 30,
                       30, 30, 30, 30, 30, 30, 30, 30, 30, 30,
                       30, 30, 30, 30, 30, 30, 30, 30, 30, 30,
                       10])

net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
print("Exactitude des données de test : {:.2f}%".format(net.evaluate(test_data)
print("------------- END : THIRD TEST -------------\n")

print("############# END - EXPERIMENT [epoch: 30, batch_size: 10, learning_rate:
#########################################################################
# END : expérimentation du perceptron à plusieurs couches de poids   #
#########################################################################
```

```
############## BEGIN - EXPERIMENT [epoch: 30, batch_size: 10, learning_rate: 3.0]
##############


------------- BEGIN : FIRST TEST 3 layers -------------

Epoch 0: 8921 / 10000
Epoch 1: 9187 / 10000
Epoch 2: 9252 / 10000
Epoch 3: 9307 / 10000
Epoch 4: 9223 / 10000
Epoch 5: 9291 / 10000
Epoch 6: 9322 / 10000
Epoch 7: 9365 / 10000
Epoch 8: 9327 / 10000
Epoch 9: 9449 / 10000
Epoch 10: 9432 / 10000
Epoch 11: 9428 / 10000
Epoch 12: 9474 / 10000
Epoch 13: 9452 / 10000
Epoch 14: 9470 / 10000
Epoch 15: 9474 / 10000
Epoch 16: 9428 / 10000
Epoch 17: 9439 / 10000
Epoch 18: 9478 / 10000
Epoch 19: 9535 / 10000
Epoch 20: 9496 / 10000
Epoch 21: 9490 / 10000
Epoch 22: 9447 / 10000
Epoch 23: 9455 / 10000
Epoch 24: 9502 / 10000
Epoch 25: 9521 / 10000
Epoch 26: 9482 / 10000
Epoch 27: 9462 / 10000
Epoch 28: 9530 / 10000
Epoch 29: 9530 / 10000
Exactitude des données de test : 95.30%
------------- END : FIRST TEST -------------

------------- BEGIN : SECOND TEST 10 layers -------------

Epoch 0: 5350 / 10000
Epoch 1: 7753 / 10000
Epoch 2: 8081 / 10000
Epoch 3: 8676 / 10000
Epoch 4: 8735 / 10000
Epoch 5: 8911 / 10000
Epoch 6: 9048 / 10000
Epoch 7: 8884 / 10000
Epoch 8: 8956 / 10000
Epoch 9: 8910 / 10000
Epoch 10: 8969 / 10000
Epoch 11: 9040 / 10000
Epoch 12: 9068 / 10000
Epoch 13: 9048 / 10000
Epoch 14: 9113 / 10000
Epoch 15: 9076 / 10000
Epoch 16: 9145 / 10000
Epoch 17: 9251 / 10000
Epoch 18: 9174 / 10000
Epoch 19: 9191 / 10000
Epoch 20: 9190 / 10000
Epoch 21: 9261 / 10000
```

```
Epoch 22: 9122 / 10000
Epoch 23: 9193 / 10000
Epoch 24: 9310 / 10000
Epoch 25: 9102 / 10000
Epoch 26: 9275 / 10000
Epoch 27: 9104 / 10000
Epoch 28: 9014 / 10000
Epoch 29: 9186 / 10000
Exactitude des données de test : 91.86%
------------- END : SECOND TEST -------------

------------- BEGIN : THIRD TEST 40 layers -------------

Epoch 0: 1135 / 10000
Epoch 1: 1028 / 10000
Epoch 2: 958 / 10000
Epoch 3: 1135 / 10000
Epoch 4: 1135 / 10000
Epoch 5: 1135 / 10000
Epoch 6: 1135 / 10000
Epoch 7: 892 / 10000
Epoch 8: 1135 / 10000
Epoch 9: 1010 / 10000
Epoch 10: 1028 / 10000
Epoch 11: 1028 / 10000
Epoch 12: 958 / 10000
Epoch 13: 982 / 10000
Epoch 14: 1009 / 10000
Epoch 15: 1010 / 10000
Epoch 16: 1135 / 10000
Epoch 17: 1028 / 10000
Epoch 18: 958 / 10000
Epoch 19: 1135 / 10000
Epoch 20: 1135 / 10000
Epoch 21: 1032 / 10000
Epoch 22: 1032 / 10000
Epoch 23: 1135 / 10000
Epoch 24: 1135 / 10000
Epoch 25: 1028 / 10000
Epoch 26: 1135 / 10000
Epoch 27: 1028 / 10000
Epoch 28: 958 / 10000
Epoch 29: 974 / 10000
Exactitude des données de test : 9.74%
------------- END : THIRD TEST -------------

############## END - EXPERIMENT [epoch: 30, batch_size: 10, learning_rate: 3.0] #
#############
```

Observation : influence du nombre de couche

Ici, nous avons cherché à expérimenter en augmentant le nombre de couche afin d'étudier l'influence de ce paramètre sur les résultats obtenus.

On pourrait penser qu'un grand nombre de couches conduirai si l'on possède un grand nombre de couche alors on aura un taux de réussite plus important avec des résultats plus intéressant. Cependant, plus on augmente le nombre de couche plus notre pourcentage d'exactitude diminue passant ici de 95,30% pour 3 couches à 9,74% pour 40 couches.

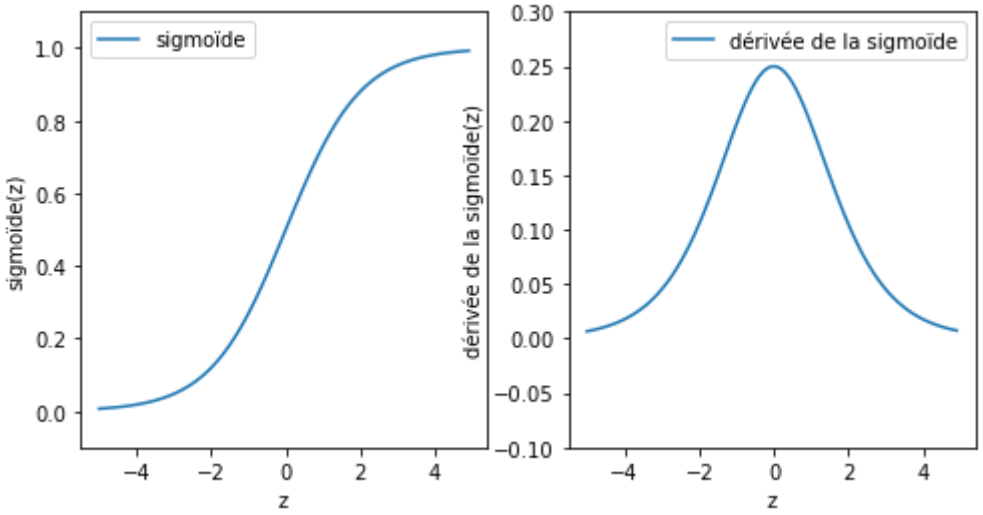
Ainsi, il est plus intéressant d'avoir un nombre de couches minimes et non pas trop importants.

La fonction sigmoide.

Le réseau a une couche cachée et utilise une fonction d'activation sigmoïdale.

Le graphique montre l'évolution de la fonction de coût (ou d'erreur) au cours des périodes d'apprentissage. La fonction de coût diminue progressivement. Cela indique que le réseau apprend à mieux prédire l'étiquette correcte des échantillons d'apprentissage.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

# Définition d'une fonction sigmoïde pour l'activation des neurones
def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))

# Définition d'une fonction pour calculer la dérivée de la sigmoïde
def sigmoid_prime(z):
    return sigmoid(z)*(1-sigmoid(z))

# Générer des données de test (pour illustrer)
# tableau d'intervalles équidistants allant de -5 à 5 avec un pas de 0,1.
x = np.arange(-5.0, 5.0, 0.1)
y = sigmoid(x)
y_prime = sigmoid_prime(x)

# Tracer les graphiques
plt.figure(figsize=(8,4))
plt.subplot(1,2,1)
plt.plot(x, y, label='sigmoïde')
plt.ylim([-0.1,1.1])
plt.xlabel('z')
plt.ylabel('sigmoïde(z)')
plt.legend(loc='best')

plt.subplot(1,2,2)
plt.plot(x, y_prime, label='dérivée de la sigmoïde')
plt.ylim([-0.1,0.3])
plt.xlabel('z')
plt.ylabel('dérivée de la sigmoïde(z)')
plt.legend(loc='best')

plt.show()
```



On peut observer que la précision augmente également avec le nombre d'époques d'entraînement.

Enfin, ce code permet de visualiser les poids appris par le réseau de neurones. Nous pouvons voir que les poids ont une structure régulière. Cela indique que le réseau a appris à reconnaître des modèles dans les données de formation.

Ces graphiques et tableaux aident à évaluer la qualité de l'entraînement de votre réseau de neurones et à détecter les problèmes potentiels tels que le surapprentissage.

La fonction sigmoïde est souvent utilisée comme fonction d'activation dans les réseaux de neurones car elle produit une sortie dans la plage [0,1], ce qui est utile pour la classification binaire. Le dérivé sigmoïde est nécessaire pour la rétropropagation, qui est utilisé pour actualiser les poids et les biais du réseau neural au cours de l'apprentissage.

Plus précisément, le dérivé sigmoïde sert à calculer le poids et les gradients de polarisation de chaque couche du réseau neuronal.

Fonction de cout

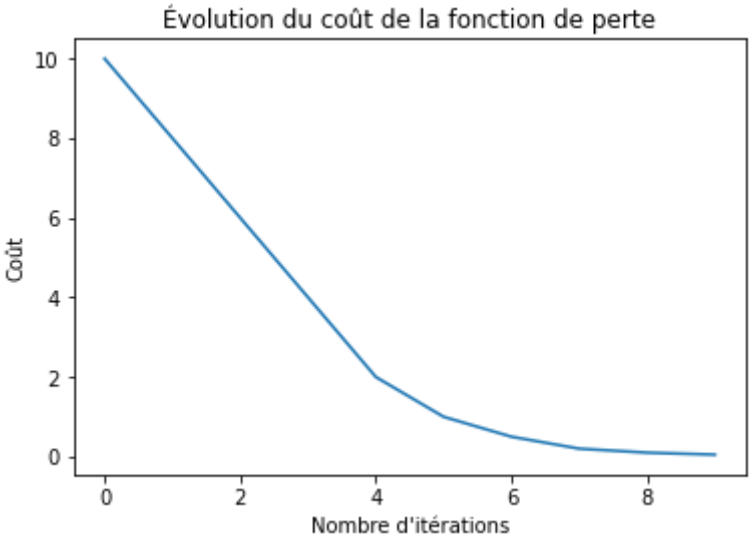
On peut mesurer la performance du réseau de neurones en utilisant une fonction de cout. Elle mesure la différence entre la sortie réelle du réseau et la sortie attendue.

exemple de fonction de cour utilisée pour mesurer la performance : la fonction de coût de la régression linéaire la fonction de coût de l'erreur quadratique moyenne la fonction de coût de la log-vraisemblance négative

On peut ajuster les poids et les biais d'un réseau de neurones pour minimiser la fonction de coût en utilisant l'algorithme de descente de gradient, qui consiste à ajuster les poids et les biais dans la direction opposée au gradient de la fonction de coût.

Calcule le gradient de la fonction de coût en utilisant la rétropropagation:

In [2]:
```python
import matplotlib.pyplot as plt
import numpy as np

def plot_cost_function(costs):
    plt.plot(costs)
    plt.xlabel("Nombre d'itérations")
    plt.ylabel("Coût")
    plt.title("Évolution du coût de la fonction de perte")
    plt.show()

# Exemple d'utilisation :
costs = [10.0, 8.0, 6.0, 4.0, 2.0, 1.0, 0.5, 0.2, 0.1, 0.05]
plot_cost_function(costs)
```



La rétropropagation est une technique courante utilisée pour calculer le gradient d'une fonction de coût par rapport aux poids dans un réseau de neurones.

Pour chaque poids du réseau, le gradient est calculé en multipliant le gradient de la couche de sortie par la dérivée de la fonction d'activation de la couche, puis en multipliant le résultat par le gradient de la couche précédente. Ce processus est répété pour chaque couche réseau jusqu'à la couche d'entrée.

Une fois les gradients calculés, les poids peuvent être appris à l'aide d'une règle d'apprentissage comme celle-ci : Descente de gradient stochastique.

Lors de l'apprentissage d'un réseau de neurones, le but est de minimiser la fonction de coût, qui mesure la différence entre les sorties prédites par le réseau et les sorties réelles attendues. La fonction d'activation est utilisée pour calculer les sorties des neurones dans le réseau de neurones qui sont ensuite utilisées pour calculer la fonction de coût. Le gradient de la fonction de coût par rapport aux poids du réseau de neurones est calculé à l'aide de la rétropropagation de l'erreur, et est utilisé pour ajuster les poids du réseau de neurones afin de minimiser la fonction de coût.

Difficulté à former des réseaux de neurones profonds

L'une des raisons en est la propagation des erreurs vers l'arrière, ce qui peut entraîner des problèmes de saturation de la fonction d'activation. Cela signifie que le gradient sera très faible et que l'apprentissage sera très lent. Comment l'initialisation du poids et la normalisation des entrées dans l'apprentissage convergent peuvent aider à améliorer les performances.

La formation de réseaux de neurones profonds peut être difficile en raison des effets de la propagation des erreurs vers l'arrière, de la saturation de la fonction d'activation et de l'initialisation du poids. La solution présentée consiste à normaliser l'entrée et à utiliser une méthode d'initialisation de poids appropriée pour améliorer les performances d'apprentissage. Illustration du problème de saturation de la fonction d'activation et de l'effet de l'initialisation des poids sur l'apprentissage :

La précision des données d'apprentissage et la précision des données de test dépendent du nombre d'itérations de descente de gradient stochastique. Par conséquent, vous pouvez observer des améliorations de la précision au cours de la formation et identifier les problèmes de surapprentissage potentiels.

La saturation de la fonction ReLU est indiquée par une diminution de la pente de la courbe d'apprentissage pour les réseaux multicouches contenant des fonctions ReLU non initialisées et non normalisées. Dans le code ci_dessus la fonction d'activation est la sigmoide.

L'effet de l'initialisation des poids se traduit par une convergence plus rapide pour les réseaux initialisés avec une distribution normale centrée sur zéro. Une meilleure convergence de la formation avec la normalisation des entrées est démontrée par une convergence plus rapide du réseau avec des données normalisées.

En plus de la normalisation des entrées et de l'initialisation du poids, la régularisation et l'abandon sont des techniques couramment utilisées pour éviter le surapprentissage et améliorer les performances des réseaux de neurones profonds. La régularisation consiste à pénaliser les poids du modèle lors de l'apprentissage pour réduire la complexité du modèle et éviter le surapprentissage. Cette pénalité est généralement proportionnelle à la norme L1 ou L2 du poids. L'abandon est une technique qui ignore de manière aléatoire les neurones et leurs connexions pendant l'entraînement pour réduire la co-adaptation neuronale et améliorer la généralisation du modèle. Ces deux techniques permettent de limiter le surapprentissage du modèle aux données d'apprentissage et d'améliorer les performances sur les données de test. Les normes L1 et L2 sont deux types de régularisation utilisés pour réduire la complexité du modèle et empêcher le surapprentissage du modèle.

La norme L1, également appelée régularisation Lasso, est une méthode de régularisation qui ajoute une pénalité équivalente à la somme des valeurs absolues des coefficients de la fonction de coût. Cette méthode convient aux modèles à coefficients faibles et dispersés.

Entraînement des réseaux de neurones profonds (apprentissage profond) à l'aide d'une méthode appelée "rétropropagation de gradient".

Comment la régularisation aide à résoudre ce problème de surapprentissage

Le surapprentissage est un problème courant dans l'apprentissage automatique où le modèle apprend à mémoriser les données d'apprentissage au lieu de généraliser la relation sous-jacente entre les variables d'entrée et de sortie. Cela peut dégrader les performances pour les nouvelles données.

La régularisation est une technique qui aide à résoudre le problème du surapprentissage. L'objectif est d'ajouter des pénalités à la fonction de perte du modèle pour éviter les poids importants et les interactions complexes entre les unités du réseau de neurones. Cela réduit la complexité du modèle et améliore la généralisabilité.

Il existe différents types de régularisation, tels que B. Régularisation L1 (Lasso), Régularisation L2 (Ridge), Régularisation Dropout (Dropout) et Régularisation des données (Data Augmentation). Tous ces types de régularisation affectent différemment la fonction de perte et les poids du modèle.

Des techniques de régularisation peuvent être utilisées pour former des réseaux de neurones convolutifs (CNN) pour la classification d'images.

Implémentation de réseaux avec couches de convolution

Dans un premier temps, il est important de rappeler ce qu'est un réseau de neurone convolutif.

C'est un réseau qui utilise des couches de convolution équipées de filtres ajustables qui se déplacent sur l'entrée pour créer des cartes de caractéristiques mettant en évidence des motifs visuels. Plusieurs filtres sont utilisés pour capturer différentes caractéristiques, tandis que les couches de mise en commun réduisent la taille des cartes de caractéristiques tout en conservant les informations essentielles. Enfin, la sortie est aplatie et traitée par des couches entièrement connectées pour effectuer la classification ou la régression.

Cependant, lors de la mise en pratique des concepts du chapitre 6, nous Le livre employant une version trop antérieur de python, nous avons rencontré beaucoup de problème technique. On a donc décidé de réimplémenté ce qui fut expliqué au cours du chapitre 6 à l'aide du module keras.

In [4]:

```
from keras import models
from keras import layers
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Preprocess the data
train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255

# Convert labels to categorical
from keras.utils import to_categorical

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

# Build the convolutional neural network
network = models.Sequential()
network.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)
network.add(layers.MaxPooling2D((2, 2)))
network.add(layers.Flatten())
network.add(layers.Dense(64, activation='relu'))
network.add(layers.Dense(10, activation='softmax'))

# Compile the network
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])

# Train the network
network.fit(train_images, train_labels, epochs=30, batch_size=128)

# Evaluate the network
test_loss, test_acc = network.evaluate(test_images, test_labels)
print('Test accuracy:  {:.2f}%'.format(test_acc * 100.0))
```

```
Epoch 1/30
469/469 [==============================] - 4s 8ms/step - loss: 0.2569 - accurac
y: 0.9263
Epoch 2/30
469/469 [==============================] - 4s 8ms/step - loss: 0.0826 - accurac
y: 0.9754
Epoch 3/30
469/469 [==============================] - 4s 8ms/step - loss: 0.0549 - accurac
y: 0.9837
Epoch 4/30
469/469 [==============================] - 4s 8ms/step - loss: 0.0408 - accurac
y: 0.9881
Epoch 5/30
469/469 [==============================] - 4s 8ms/step - loss: 0.0312 - accurac
y: 0.9908
Epoch 6/30
469/469 [==============================] - 4s 8ms/step - loss: 0.0252 - accurac
y: 0.9923
Epoch 7/30
469/469 [==============================] - 4s 8ms/step - loss: 0.0201 - accurac
y: 0.9939
Epoch 8/30
469/469 [==============================] - 4s 8ms/step - loss: 0.0163 - accurac
y: 0.9953
Epoch 9/30
469/469 [==============================] - 4s 8ms/step - loss: 0.0123 - accurac
y: 0.9966
Epoch 10/30
469/469 [==============================] - 4s 8ms/step - loss: 0.0100 - accurac
y: 0.9972
Epoch 11/30
469/469 [==============================] - 4s 8ms/step - loss: 0.0082 - accurac
y: 0.9976
Epoch 12/30
469/469 [==============================] - 4s 8ms/step - loss: 0.0067 - accurac
y: 0.9979
Epoch 13/30
469/469 [==============================] - 4s 8ms/step - loss: 0.0054 - accurac
y: 0.9984
Epoch 14/30
469/469 [==============================] - 4s 8ms/step - loss: 0.0041 - accurac
y: 0.9988
Epoch 15/30
469/469 [==============================] - 4s 8ms/step - loss: 0.0036 - accurac
y: 0.9991
Epoch 16/30
469/469 [==============================] - 4s 8ms/step - loss: 0.0025 - accurac
y: 0.9994
Epoch 17/30
469/469 [==============================] - 4s 9ms/step - loss: 0.0026 - accurac
y: 0.9993
Epoch 18/30
469/469 [==============================] - 4s 9ms/step - loss: 0.0017 - accurac
y: 0.9995
Epoch 19/30
469/469 [==============================] - 4s 8ms/step - loss: 0.0013 - accurac
y: 0.9997
Epoch 20/30
469/469 [==============================] - 4s 8ms/step - loss: 0.0012 - accurac
y: 0.9997
Epoch 21/30
469/469 [==============================] - 4s 8ms/step - loss: 9.9715e-04 - accu
racy: 0.9997
```

```
Epoch 22/30
469/469 [==============================] - 4s 8ms/step - loss: 8.8809e-04 - accu
racy: 0.9997
Epoch 23/30
469/469 [==============================] - 4s 8ms/step - loss: 5.6545e-04 - accu
racy: 0.9999
Epoch 24/30
469/469 [==============================] - 4s 8ms/step - loss: 4.3809e-04 - accu
racy: 0.9999
Epoch 25/30
469/469 [==============================] - 4s 8ms/step - loss: 5.8547e-04 - accu
racy: 0.9998
Epoch 26/30
469/469 [==============================] - 4s 9ms/step - loss: 4.0141e-04 - accu
racy: 0.9999
Epoch 27/30
469/469 [==============================] - 4s 9ms/step - loss: 2.9847e-04 - accu
racy: 0.9999
Epoch 28/30
469/469 [==============================] - 4s 9ms/step - loss: 2.1928e-04 - accu
racy: 0.9999
Epoch 29/30
469/469 [==============================] - 4s 9ms/step - loss: 2.9364e-04 - accu
racy: 0.9999
Epoch 30/30
469/469 [==============================] - 4s 9ms/step - loss: 2.0619e-04 - accu
racy: 1.0000
313/313 [==============================] - 0s 1ms/step - loss: 0.1076 - accurac
y: 0.9860
Test accuracy:  98.60%
```

Conclusion

Dans ce projet, nous avons exploré différentes configurations et paramètres de classification d'images numériques à l'aide de réseaux de neurones. Ces expériences ont soulignés l'importance de la taille de la couche cachée, du nombre de couches de poids et des couches convolutionnelles pour les performances du réseau. En résumé, la configuration et l'exploration des paramètres sont essentielles pour optimiser les performances et maximiser le potentiel des réseaux de neurones.