Question 1 :Un premier modèle

```python
from sklearn.datasets import fetch_openml
from sklearn.metrics import log_loss
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import numpy as np

# Charger les données MNIST
X, y = fetch_openml("mnist_784", version=1, return_X_y=True,
as_frame=False)

print("Dimensions de X :", X.shape)
print("Dimensions de y :", y.shape)
```

```
Dimensions de X : (70000, 784)
Dimensions de y : (70000,)
```

```python
y = y.astype(int)
X = X / 255.0
# Extraire uniquement les images des classes 0 et 1
indices = np.where((y == 0) | (y == 1))
X = X[indices]
y = y[indices]

# Diviser les données en ensembles d'entraînement et de test
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.1, random_state=42)

def binary_cross_entropy(X, y, w):
    m = X.shape[0]
    z = np.dot(X, w)
    y_pred = 1 / (1 + np.exp(-z))  # Fonction sigmoïde

    # la fonction d'entropie binaire croisée
    bce_fun = -np.mean(y * np.log(y_pred + 1e-10) + (1 - y) * np.log(1
- y_pred + 1e-10)) # 1e-10 Pour éviter log(0)
    bce_grad = np.dot(X.T, (y_pred - y)) / m # gradient

    return bce_fun, bce_grad

def optimisation(w_init, eta, X, y):
    w = w_init
    max_iter = 1000
    for _ in range(max_iter):
        bce_fun, bce_grad = binary_cross_entropy(X, y, w)
        w -= eta * bce_grad  # Mise à jour des poids

    def taux_classification(X, y, w):
        z = np.dot(X, w)
        y_pred = 1 / (1 + np.exp(-z))
        y_pred = (y_pred >= 0.5).astype(int)
        return np.mean(y_pred == y) * 100

    taux_train = taux_classification(X_train, y_train, w)
    taux_test = taux_classification(X_test, y_test, w)

    return w, taux_train, taux_test

w_init = np.zeros(X_train.shape[1])  # Initialisation des poids à zéro
eta = 0.1  # Taux d'apprentissage

w_opt, taux_train, taux_test = optimisation(w_init, eta, X_train,
y_train)
print("Taux de classification sur l'ensemble d'entraînement :",
taux_train)
print("Taux de classification sur l'ensemble de test :", taux_test)
```

```
Taux de classification sur l'ensemble d'entraînement :
99.85716433619005
Taux de classification sur l'ensemble de test : 99.93234100135318
```

Question 2 : Un petit réseau convolutif

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten,
Dense
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split
from sklearn.datasets import fetch_openml

# Charger les données MNIST
X, y = fetch_openml("mnist_784", version=1, return_X_y=True,
as_frame=False)
y = y.astype(int)

# Filtrer uniquement les images des classes 0 et 1
indices = (y == 0) | (y == 1)
X = X[indices]
y = y[indices]

# Normaliser les données
X = X / 255.0

# Reshaper les données en format image pour CNN
X = X.reshape(-1, 28, 28, 1)

# Encoder les étiquettes en one-hot (pour 2 classes)
y = to_categorical(y, num_classes=2)
```

```
# Diviser les données en ensembles d'entraînement (90%) et de test
(10%)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.1, random_state=42)

# Construire le modèle CNN
model = Sequential()

# Ajouter des couches convolutionnelles et de pooling
# premier couche
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28,
1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
# 2ème couche
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Aplatier les sorties des couches précédentes
model.add(Flatten())

# Ajouter une couche dense entièrement connectée
model.add(Dense(128, activation='relu'))  # Couche cachée avec 128
neurones
model.add(Dense(2, activation='softmax'))  # Couche de sortie (2
classes)
```

```
C:\Users\HP\anaconda3\Lib\site-packages\keras\src\layers\
convolutional\base_conv.py:107: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)
```

```
# Compiler le modèle
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Entraîner le modèle
history = model.fit(X_train, y_train,
                    epochs=10,
                    batch_size=32,
                    validation_data=(X_test, y_test),
                    verbose=1)
```

```
Epoch 1/10
416/416 ━━━━━━━━━━━━━━━━━━━━ 12s 19ms/step - accuracy: 0.9858 - loss:
0.0438 - val_accuracy: 0.9986 - val_loss: 0.0067
Epoch 2/10
```

```
416/416 ━━━━━━━━━━━━━━━━━━━━ 8s 20ms/step - accuracy: 0.9982 - loss:
0.0052 - val_accuracy: 1.0000 - val_loss: 1.1449e-04
Epoch 3/10
416/416 ━━━━━━━━━━━━━━━━━━━━ 8s 18ms/step - accuracy: 0.9993 - loss:
0.0017 - val_accuracy: 0.9986 - val_loss: 0.0047
Epoch 4/10
416/416 ━━━━━━━━━━━━━━━━━━━━ 7s 17ms/step - accuracy: 0.9996 - loss:
0.0011 - val_accuracy: 0.9993 - val_loss: 0.0043
Epoch 5/10
416/416 ━━━━━━━━━━━━━━━━━━━━ 9s 22ms/step - accuracy: 0.9997 - loss:
4.2151e-04 - val_accuracy: 0.9993 - val_loss: 0.0026
Epoch 6/10
416/416 ━━━━━━━━━━━━━━━━━━━━ 7s 18ms/step - accuracy: 1.0000 - loss:
1.6127e-05 - val_accuracy: 0.9993 - val_loss: 0.0036
Epoch 7/10
416/416 ━━━━━━━━━━━━━━━━━━━━ 7s 17ms/step - accuracy: 1.0000 - loss:
7.3679e-06 - val_accuracy: 0.9993 - val_loss: 0.0039
Epoch 8/10
416/416 ━━━━━━━━━━━━━━━━━━━━ 7s 18ms/step - accuracy: 1.0000 - loss:
1.0085e-05 - val_accuracy: 0.9993 - val_loss: 0.0037
Epoch 9/10
416/416 ━━━━━━━━━━━━━━━━━━━━ 7s 17ms/step - accuracy: 1.0000 - loss:
3.0135e-06 - val_accuracy: 0.9993 - val_loss: 0.0039
Epoch 10/10
416/416 ━━━━━━━━━━━━━━━━━━━━ 7s 17ms/step - accuracy: 1.0000 - loss:
1.3536e-06 - val_accuracy: 0.9993 - val_loss: 0.0042

# Évaluation des performances
test_loss, test_accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f"Taux de classification sur l'ensemble de test : {test_accuracy
* 100:.2f}%")

Taux de classification sur l'ensemble de test : 99.93%
```

Question 3 : Astuce du Noyau

```
import numpy as np

def G(x, y, sigma):
    # Norme au carré entre x et y
    distance = np.linalg.norm(x - y)**2
    # Calcul du noyau gaussien
    gaussian_kernel = np.exp(-distance / (2 * sigma**2))

    return gaussian_kernel

x = np.array([1, 2])
y = np.array([2, 3])
sigma = 1.0
print(G(x, y, sigma))  # Renvoie une valeur entre 0 et 1
```

```
0.36787944117144222

def optimize(X_train, y_train, X_test, y_test, sigma, epochs=100,
learning_rate=0.01):
    N = X_train.shape[0]
    lbda = np.zeros(N)  # Initialisation des coefficients lambda à 0

    rate_training = []  # Les taux de classification sur
l'entraînement
    rate_test = []      # Les taux de classification sur le test

    # Matrice du noyau gaussien
    K_train = np.zeros((N, N))
    for i in range(N):
        for j in range(N):
            K_train[i, j] = G(X_train[i], X_train[j], sigma)

    # Descente de gradient
    for epoch in range(epochs):
        # Prédictions sur l'ensemble d'entraînement
        y_pred_train = np.dot(K_train, lbda)

        # Erreur entre prédictions et labels réels
        error = y_pred_train - y_train
        grad = 2 * np.dot(K_train, error)

        # Mise à jour des coefficients lambda
        lbda -= learning_rate * grad

        # Calcul du taux de classification (entraînement)
        y_pred_train_binary = np.sign(y_pred_train)  # Convertir en
prédictions binaires
        accuracy_train = np.mean(y_pred_train_binary == y_train) * 100
        rate_training.append(accuracy_train)

        # Calcul du taux de classification (test)
        y_pred_test = np.array([
            np.sum(lbda * np.array([G(X_train[j], X_test[i], sigma)
for j in range(N)]))
            for i in range(X_test.shape[0])
        ])
        y_pred_test_binary = np.sign(y_pred_test)  # Convertir en
prédictions binaires
        accuracy_test = np.mean(y_pred_test_binary == y_test) * 100
        rate_test.append(accuracy_test)

        if epoch % 10 == 0:
            print(f"Epoch {epoch}: Taux de classification
(entraînement) = {accuracy_train:.2f}% | Test = {accuracy_test:.2f}%")
```

```
    return lbda, rate_training, rate_test

from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
import numpy as np

X, y = fetch_openml("mnist_784", version=1, return_X_y=True,
as_frame=False)
y = y.astype(int)

indices = (y == 0) | (y == 1)
X = X[indices]
y = y[indices]
X = X / 255.0

# 0 devient -1, 1 reste 1
y = np.where(y == 0, -1, 1)

# Sous-échantillonner : 50 images de 0 et 50 images de 1
X0, y0 = X[y == -1][:50], y[y == -1][:50]
X1, y1 = X[y == 1][:50], y[y == 1][:50]
X_subset = np.vstack([X0, X1])
y_subset = np.hstack([y0, y1])

X_train, X_test, y_train, y_test = train_test_split(X_subset,
y_subset, test_size=0.1, random_state=42)

sigma = 1.0
epochs = 100
learning_rate = 0.01


lbda, rate_training, rate_test = optimize(X_train, y_train, X_test,
y_test, sigma, epochs, learning_rate)

print("Lambda optimisé :", lbda)
print("Taux de classification final (entraînement) :", rate_training[-
1])
print("Taux de classification final (test) :", rate_test[-1])
```

```
Epoch 0: Taux de classification (entraînement) = 0.00% | Test =
100.00%
Epoch 10: Taux de classification (entraînement) = 100.00% | Test =
100.00%
Epoch 20: Taux de classification (entraînement) = 100.00% | Test =
100.00%
Epoch 30: Taux de classification (entraînement) = 100.00% | Test =
100.00%
Epoch 40: Taux de classification (entraînement) = 100.00% | Test =
100.00%
```

```
Epoch 50: Taux de classification (entraînement) = 100.00% | Test =
100.00%
Epoch 60: Taux de classification (entraînement) = 100.00% | Test =
100.00%
Epoch 70: Taux de classification (entraînement) = 100.00% | Test =
100.00%
Epoch 80: Taux de classification (entraînement) = 100.00% | Test =
100.00%
Epoch 90: Taux de classification (entraînement) = 100.00% | Test =
100.00%
Lambda optimisé : [-0.86738044 -0.86738044  0.83413468 -0.86738044
0.81153547 -0.86738044
  0.86475017  0.79524098 -0.86725827 -0.86716832  0.86738044
0.86319402
 -0.86738044 -0.86738044  0.85767421 -0.86738043 -0.86716774
0.85954764
 -0.86738044  0.86121932 -0.86738044 -0.86738044  0.85846715 -
0.86738044
  0.82678071 -0.86738044  0.86616108  0.86267095 -0.86738036 -
0.86737975
 -0.86738044 -0.86737975 -0.86738044  0.83394743 -0.86738044 -
0.8672578
  0.86738044 -0.86738044  0.86069219 -0.86738044 -0.86738044 -
0.86738042
 -0.86738044 -0.86737973 -0.86737991  0.8617215  -0.86738044
0.86310087
 -0.86738039  0.86017798  0.83520842  0.86736125  0.71786814 -
0.86738044
  0.73320919  0.81348557 -0.86738044  0.85893879  0.8657984
0.84227242
  0.86117412 -0.86737985  0.86708472 -0.86738044  0.86736409
0.85821968
  0.86092807 -0.86738044  0.85592768  0.86738043  0.86607722
0.86132952
 -0.86738044 -0.86738044 -0.86737977  0.86536588 -0.86738044 -
0.86738043
 -0.86738044  0.85078182  0.74702032  0.86619089  0.85941782
0.86147287
 -0.86738039  0.83968012  0.86738044 -0.86738044  0.74637767
0.86651312]
Taux de classification final (entraînement) : 100.0
Taux de classification final (test) : 100.0
```