Travaux Pratique : Classification binaire avec un réseau de neurones

Pr. M. BENADDY
Masters : IMSD & IAA

2024-2025

1 Objectif du TP

L'objectif de ce travaux pratique est de concevoir et d'implémenter une classe Python `NeuralNetwork` pour un réseau de neurones multicouche (MLP) effectuant une classification binaire (diabétique ou non) sur la base de données Pima Indians Diabetes. Le nombre de couches cachées est paramétrable. L'implémentation est réalisée sans framework de deep learning, en codant manuellement la propagation avant, la rétropropagation et l'optimisation par descente de gradient stochastique (SGD) en codant les formules mathématiques pour chaque étape.
Travail à rendre : Un document sous forme d'un article scientifique rédigé en anglais sous Latex selon le template article avec un titre, auteur, abstract, introduction, mthodes, résultats et discussion (IMRAD), avec le code à inclure dans github et fournir le lien dans l'article. Bien sur vous devez aussi inclure les références.
Faites une recherche sur la structure IMRAD pour plus d'informations IMRAD (https://en.wikipedia.org/wiki/IMRAD).

2 Description de la base de données

- **Source** : Pima Indians Diabetes Database (https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database/data).
- **Caractéristiques** : 8 variables (Pregnancies, Glucose, BloodPressure, SkinThickness, Insulin, BMI, DiabetesPedigreeFunction, Age).
- **Cible** : Outcome (0 = non diabétique, 1 = diabétique).
- **Taille** : 768 échantillons.
- **Remarque** : Présence de valeurs manquantes (zéros non valides dans `Glucose`, `BloodPressure`, `SkinThickness`, `Insulin`, `BMI`).

3 Formules mathématiques

3.1 Propagation avant

Pour une couche l, les calculs sont :

- **Entrée pondérée** :
$$Z^{[l]} = A^{[l-1]}W^{[l]} + b^{[l]}$$
où $A^{[l-1]}$ est l'activation de la couche précédente, $W^{[l]}$ est la matrice des poids, et $b^{[l]}$ est le biais.

- **Activation** :
  - Couches cachées (ReLU) :
$$A^{[l]} = \text{ReLU}(Z^{[l]}) = \max(0, Z^{[l]})$$
  - Couche de sortie (sigmoïde) :
$$A^{[L]} = \sigma(Z^{[L]}) = \frac{1}{1 + e^{-Z^{[L]}}}$$

3.2 Fonction de perte

La perte est la **Binary Cross-Entropy** :

$$J = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

où m est le nombre d'exemples, y_i est la vraie étiquette, et $\hat{y}_i = A^{[L]}$ est la prédiction.

3.3 Rétropropagation

- **Couche de sortie** :
$$dZ^{[L]} = A^{[L]} - y$$
$$dW^{[L]} = \frac{1}{m}(A^{[L-1]})^T dZ^{[L]}$$
$$db^{[L]} = \frac{1}{m}\sum_{i=1}^{m} dZ_i^{[L]}$$

- **Couches cachées** ($l = L-1, \ldots, 1$) :
$$dZ^{[l]} = (dZ^{[l+1]}(W^{[l+1]})^T) \cdot \text{ReLU}'(Z^{[l]})$$
où $\text{ReLU}'(z) = 1$ si $z > 0$, sinon 0.
$$dW^{[l]} = \frac{1}{m}(A^{[l-1]})^T dZ^{[l]}$$
$$db^{[l]} = \frac{1}{m}\sum_{i=1}^{m} dZ_i^{[l]}$$

3.4 Mise à jour des poids

$$W^{[l]} \leftarrow W^{[l]} - \eta dW^{[l]}$$
$$b^{[l]} \leftarrow b^{[l]} - \eta db^{[l]}$$

où η est le taux d'apprentissage.

4 Spécifications de la classe `NeuralNetwork`

- **Attributs** :
  - `layer_sizes` : Liste des tailles des couches (entrée, cachées, sortie).
  - `weights` : Liste des matrices $W^{[l]}$.
  - `biases` : Liste des vecteurs $b^{[l]}$.
  - `learning_rate` : Taux d'apprentissage η.
- **Méthodes** :
  - `__init__(layer_sizes, learning_rate)` : Initialise les poids et biais.
  - `forward(X)` : Calcule $A^{[L]}$.
  - `compute_loss(y_true, y_pred)` : Calcule J.
  - `backward(X, y, outputs)` : Calcule les gradients.
  - `train(X, y, epochs, batch_size)` : Entraîne le modèle.
  - `predict(X)` : Prédit les classes (0 ou 1).

5 Étapes du TP

5.1 Préparation des données

1. Charger `diabetes.csv` avec `pandas`.
2. Remplacer les zéros non valides par la médiane.
3. Standardiser : $X \leftarrow \frac{X-\mu}{\sigma}$.
4. Diviser en ensembles d'entraînement (80 %) et de test (20 %) avec stratification.

5.2 Implémentation et entraînement

- Configurer un réseau avec 2 couches cachées (16 et 8 neurones).
- Entraîner sur 100 époques avec une taille de mini-lot de 32.

5.3 Évaluation

- Calculer l'accuracy, la précision, le rappel et le F1-score.
- Générer une matrice de confusion.

6 Code d'implémentation

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix,
    classification_report
import matplotlib.pyplot as plt
import seaborn as sns

# Fonctions d'activation
def relu(x):
    """
    ReLU activation: max(0, x)
    """
    assert isinstance(x, np.ndarray), "Input to ReLU must be a
        numpy array"
    # TODO
    assert np.all(result >= 0), "ReLU output must be non-negative"
    return result

def relu_derivative(x):
    """
    Derivative of ReLU: 1 if x > 0, else 0
    """
    assert isinstance(x, np.ndarray), "Input to ReLU derivative
        must be a numpy array"
    # TODO
    assert np.all((result == 0) | (result == 1)), "ReLU derivative
        must be 0 or 1"
    return result

def sigmoid(x):
    """
    Sigmoid activation: 1 / (1 + exp(-x))
    """
    assert isinstance(x, np.ndarray), "Input to sigmoid must be a
        numpy array"
    # TODO
    assert np.all((result >= 0) & (result <= 1)), "Sigmoid output
        must be in [0, 1]"
    return result

def sigmoid_derivative(x):
    """
    Derivative of sigmoid: sigmoid(x) * (1 - sigmoid(x))
    """
    assert isinstance(x, np.ndarray), "Input to sigmoid derivative
        must be a numpy array"
    # TODO
```

```
42      assert np.all((result >= 0) & (result <= 0.25)), "Sigmoid
            derivative must be in [0, 0.25]"
43      return result
44
45  # Classe NeuralNetwork
46  class NeuralNetwork:
47      def __init__(self, layer_sizes, learning_rate=0.01):
48          """
49          Initialize the neural network with given layer sizes and
                learning rate.
50          layer_sizes: List of integers [input_size, hidden1_size,
                ..., output_size]
51          """
52          assert isinstance(layer_sizes, list) and len(layer_sizes)
                >= 2, "layer_sizes must be a list with at least 2
                elements"
53          assert all(isinstance(size, int) and size > 0 for size in
                layer_sizes), "All layer sizes must be positive
                integers"
54          assert isinstance(learning_rate, (int, float)) and
                learning_rate > 0, "Learning rate must be a positive
                number"
55
56          self.layer_sizes = layer_sizes
57          self.learning_rate = learning_rate
58          self.weights = []
59          self.biases = []
60
61          # Initialisation des poids et biais
62          np.random.seed(42)
63          for i in range(len(layer_sizes) - 1):
64              # TODO
65              assert w.shape == (layer_sizes[i], layer_sizes[i+1]),
                    f"Weight matrix {i+1} has incorrect shape"
66              assert b.shape == (1, layer_sizes[i+1]), f"Bias vector
                     {i+1} has incorrect shape"
67              self.weights.append(w)
68              self.biases.append(b)
69
70      def forward(self, X):
71          """
72          Forward propagation: Z^{[l]} = A^{[l-1]} W^{[l]} + b^{[l
                ]}, A^{[l]} = g(Z^{[l]})
73          """
74          assert isinstance(X, np.ndarray), "Input X must be a numpy
                 array"
75          assert X.shape[1] == self.layer_sizes[0], f"Input
                dimension ({X.shape[1]}) must match input layer size ({
                self.layer_sizes[0]})"
76
77          self.activations = [X]
```

```
        self.z_values = []

        for i in range(len(self.weights) - 1):
            # TODO
            assert z.shape == (X.shape[0], self.layer_sizes[i+1]),
                f"Z^{[i+1]} has incorrect shape"
            # TODO

        # TODO
        assert z.shape == (X.shape[0], self.layer_sizes[-1]), "
           Output Z has incorrect shape"
        self.z_values.append(z)
        output = sigmoid(z)
        assert output.shape == (X.shape[0], self.layer_sizes[-1]),
            "Output A has incorrect shape"
        # TODO

        return self.activations[-1]

    def compute_loss(self, y_true, y_pred):
        """
        Binary Cross-Entropy: J = -1/m * sum(y * log(y_pred) + (1-
           y) * log(1-y_pred))
        """
        assert isinstance(y_true, np.ndarray) and isinstance(
           y_pred, np.ndarray), "Inputs to loss must be numpy
           arrays"
        assert y_true.shape == y_pred.shape, "y_true and y_pred
           must have the same shape"
        assert np.all((y_true == 0) | (y_true == 1)), "y_true must
            contain only 0s and 1s"

        # TODO
        assert not np.isnan(loss), "Loss computation resulted in
           NaN"
        return loss

    def compute_accuracy(self, y_true, y_pred):
        """
        Compute accuracy: proportion of correct predictions
        """
        assert isinstance(y_true, np.ndarray) and isinstance(
           y_pred, np.ndarray), "Inputs to accuracy must be numpy
           arrays"
        assert y_true.shape == y_pred.shape, "y_true and y_pred
           must have the same shape"

        # TODO
        assert 0 <= accuracy <= 1, "Accuracy must be between 0 and
            1"
        return accuracy
```

```
116
117     def backward(self, X, y, outputs):
118         """
119         Backpropagation: compute dW^{[l]}, db^{[l]} for each layer
120         """
121         assert isinstance(X, np.ndarray) and isinstance(y, np.
                ndarray) and isinstance(outputs, np.ndarray), "Inputs
                to backward must be numpy arrays"
122         assert X.shape[1] == self.layer_sizes[0], f"Input
                dimension ({X.shape[1]}) must match input layer size ({
                self.layer_sizes[0]})"
123         assert y.shape == outputs.shape, "y and outputs must have
                the same shape"
124
125         m = X.shape[0]
126         self.d_weights = [np.zeros_like(w) for w in self.weights]
127         self.d_biases = [np.zeros_like(b) for b in self.biases]
128
129         dZ = outputs - y
130         assert dZ.shape == outputs.shape, "dZ for output layer has
                incorrect shape"
131         self.d_weights[-1] = (self.activations[-2].T @ dZ) / m
132         self.d_biases[-1] = np.sum(dZ, axis=0, keepdims=True) / m
133
134         for i in range(len(self.weights) - 2, -1, -1):
135             # TODO
136
137         # TODO: Ajouter une r gularisation L2 aux gradients des
                poids
138         # dW^{[l]} += lambda * W^{[l]} / m,  o   lambda est le
                coefficient de r gularisation
139
140         for i in range(len(self.weights)):
141             # TODO
142
143     def train(self, X, y, X_val, y_val, epochs, batch_size):
144         """
145         Train the neural network using mini-batch SGD, with
                validation
146         """
147         assert isinstance(X, np.ndarray) and isinstance(y, np.
                ndarray), "X and y must be numpy arrays"
148         assert isinstance(X_val, np.ndarray) and isinstance(y_val,
                np.ndarray), "X_val and y_val must be numpy arrays"
149         assert X.shape[1] == self.layer_sizes[0], f"Input
                dimension ({X.shape[1]}) must match input layer size ({
                self.layer_sizes[0]})"
150         assert y.shape[1] == self.layer_sizes[-1], f"Output
                dimension ({y.shape[1]}) must match output layer size
                ({self.layer_sizes[-1]})"
```

```
151         assert X_val.shape[1] == self.layer_sizes[0], f"Validation
                input dimension ({X_val.shape[1]}) must match input
               layer size ({self.layer_sizes[0]})"
152         assert y_val.shape[1] == self.layer_sizes[-1], f"
               Validation output dimension ({y_val.shape[1]}) must
               match output layer size ({self.layer_sizes[-1]})"
153         assert isinstance(epochs, int) and epochs > 0, "Epochs
               must be a positive integer"
154         assert isinstance(batch_size, int) and batch_size > 0, "
               Batch size must be a positive integer"
155
156         train_losses = []
157         val_losses = []
158         train_accuracies = []
159         val_accuracies = []
160
161         for epoch in range(epochs):
162             indices = np.random.permutation(X.shape[0])
163             # TODO
164
165             epoch_loss = 0
166             for i in range(0, X.shape[0], batch_size):
167                 # TODO
168
169                 outputs = self.forward(X_batch)
170                 epoch_loss += self.compute_loss(y_batch, outputs)
171                 self.backward(X_batch, y_batch, outputs)
172
173             # Calculer les pertes et accuracies pour l'
                   entra nement et la validation
174             # TODO
175
176             train_losses.append(train_loss)
177             val_losses.append(val_loss)
178             train_accuracies.append(train_accuracy)
179             val_accuracies.append(val_accuracy)
180
181             if epoch % 10 == 0:
182                 print(f"Epoch {epoch}, Train Loss: {train_loss:.4f
                       }, Val Loss: {val_loss:.4f}, "
183                       f"Train Acc: {train_accuracy:.4f}, Val Acc:
                           {val_accuracy:.4f}")
184
185         return train_losses, val_losses, train_accuracies,
               val_accuracies
186
187     def predict(self, X):
188         """
189         Predict class labels (0 or 1)
190         """
```

```
191         assert isinstance(X, np.ndarray), "Input X must be a numpy
                array"
192         assert X.shape[1] == self.layer_sizes[0], f"Input
               dimension ({X.shape[1]}) must match input layer size ({
               self.layer_sizes[0]})"
193
194         # TODO
195         assert predictions.shape == (X.shape[0], self.layer_sizes
               [-1]), "Predictions have incorrect shape"
196         return predictions
197
198 # Charger et pr parer les donn es
199 data = pd.read_csv('diabetes.csv')
200 # TODO
201 assert X.shape[0] == y.shape[0], "Number of samples in X and y
       must match"
202 assert X.shape[1] == 8, "Expected 8 features in input data"
203
204 # Standardisation : X = (X - mu) / sigma
205 # TODO
206
207 # Diviser les donn es en entra nement, validation et test
208 X_temp, X_test, y_temp, y_test = train_test_split(X, y, test_size
       =0.2, stratify=y, random_state=42)
209 X_train, X_val, y_train, y_val = train_test_split(X_temp, y_temp,
       test_size=0.25, stratify=y_temp, random_state=42)  # 0.25 x 0.8
        = 0.2
210 assert X_train.shape[0] + X_val.shape[0] + X_test.shape[0] == X.
       shape[0], "Train-val-test split sizes must sum to total samples
       "
211
212 # Cr er et entra ner le mod le
213 layer_sizes = [X_train.shape[1], 16, 8, 1]
214 nn = NeuralNetwork(layer_sizes, learning_rate=0.01)
215 train_losses, val_losses, train_accuracies, val_accuracies = nn.
       train(X_train, y_train, X_val, y_val, epochs=100, batch_size
       =32)
216
217 # TODO: Ajouter une validation crois e pour  valuer  la
       robustesse du mod le
218 # TODO: Impl menter l'optimiseur Adam pour une meilleure
       convergence
219
220 # Pr dictions et  valuation
221 y_pred = nn.predict(X_test)
222 print("\nRapport de classification (Test set) :")
223 print(classification_report(y_test, y_pred))
224
225 # Matrice de confusion
226 cm = confusion_matrix(y_test, y_pred)
227 # TODO
```

```
plt.show()

# Courbes de perte et d'accuracy
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Courbe de perte
# TO DO
plt.show()
```

Listing 1: Implémentation en Python

7 Résultats attendus

- **Accuracy** : Environ 65–75 % (ou plus selon la configuration).
- **F1-score** : Évalue l'équilibre entre précision et rappel.
- **Courbe de perte** : Convergence progressive.
- **Matrice de confusion** : Visualisation des erreurs.

8 Limites et améliorations

- **Limites** :
  - Sensibilité au taux d'apprentissage η.
  - Déséquilibre des classes (35 % diabétiques).
  - Absence de régularisation.
- **Améliorations** :
  - Ajouter une régularisation L2 : $J = J + \lambda \sum_l ||W^{[l]}||_2^2$.
  - Implémenter Adam.
  - Tester des architectures variées (ex. : `[8, 32, 16, 8, 1]`).

9 Livrables

- Code Python documenté avec formules.
- Rapport selon la structure IMRAD incluant :
  - Prétraitement, architecture, formules.
  - Résultats (accuracy, F1-score, matrice de confusion).
  - Résultats qualitatifs.
  - Analyse et suggestions d'amélioration.

10 Exemple avec 3 couches cachées

```
layer_sizes = [X_train.shape[1], 32, 16, 8, 1]
nn = NeuralNetwork(layer_sizes, learning_rate=0.01)
losses = nn.train(X_train, y_train, epochs=100, batch_size=32)
```

Listing 2: Exemple avec 3 couches cachées