UNIVERSITÉ DE TECHNOLOGIE DE BELFORT-MONTBÉLIARD

Département Informatique

Formation d'Ingénieur par Alternance (FISA)

SYSTÈME D'APPRENTISSAGE PAR RÉSEAUX DE NEURONES

Rapport de Projet

Étudiants : Ouelhazi Naoures
Hajar Lyoubi
Belarbi Nada

Encadrant : Pr. Abderrafiaa KOUKAM

Année : 2024-2025

Janvier 2025

Table des matières

1 Introduction

Ce rapport présente le développement d'un système interactif d'apprentissage supervisé basé sur les réseaux de neurones. L'objectif principal était de concevoir et implémenter un système permettant aux utilisateurs de créer, paramétrer et tester des algorithmes d'apprentissage en considérant les réseaux de neurones comme des composants logiciels réutilisables.

Le projet met l'accent sur l'application rigoureuse des principes du génie logiciel dans le domaine de l'intelligence artificielle, démontrant comment les bonnes pratiques de développement logiciel peuvent améliorer la conception et l'implémentation des systèmes d'IA.

2 Analyse des Exigences

2.1 Objectifs du Projet

Conformément à l'énoncé, les objectifs principaux étaient :

1. **Développer un système interactif** permettant la création et le paramétrage d'algorithmes d'apprentissage supervisés
2. **Considérer les réseaux de neurones comme des composants logiciels** réutilisables et paramétrables
3. **Fournir un modèle objet** complet d'un réseau de neurones
4. **Proposer des fonctionnalités** pour paramétrer et tester les algorithmes sur des jeux de données
5. **Appliquer les principes du génie logiciel** tout au long du développement

2.2 Concepts Clés Implémentés

Le système implémente fidèlement les trois concepts fondamentaux définis dans l'énoncé :
— **Neurone** : Unité de base avec poids, biais et fonction d'activation
— **Couche** : Groupe de neurones effectuant la même opération
— **Réseau** : Ensemble de couches interconnectées pour l'apprentissage

3 Architecture du Système

3.1 Vue d'Ensemble

L'architecture du système suit le pattern Model-View-Controller (MVC) pour une séparation claire des responsabilités :

```
projet_geni_logiciel/
 src/
    models/          # Modèles (Neuron, Layer, Network)
    controllers/     # Contrôleurs (NetworkBuilder, TrainingController)
    views/          # Vues (GUI, CLI)
    data/           # Gestion des données
    utils/          # Utilitaires (visualisation)
 tests/              # Tests unitaires
 examples/           # Exemples d'utilisation
 datasets/           # Jeux de données CSV
```

FIGURE 1 – Structure du projet

3.2 Modèle Objet des Réseaux de Neurones

3.2.1 Classe Neuron

La classe `Neuron` implémente l'unité de base conformément à l'énoncé :

```python
class Neuron:
    def __init__(self, n_inputs: int, activation: ActivationFunction):
        self.weights = np.random.randn(n_inputs) * 0.1
        self.bias = 0.0
        self.activation = activation
        self.inputs = None
        self.output = None
        self.delta = None

    def forward(self, inputs: np.ndarray) -> float:
        """Propagation avant : somme pond r e + activation"""
        self.inputs = inputs
        z = np.dot(inputs, self.weights) + self.bias
        self.output = self.activation.forward(z)
        return self.output
```

Listing 1 – Implémentation du neurone

3.2.2 Classe Layer

La classe `Layer` gère un groupe de neurones :

```python
class Layer:
    def __init__(self, n_neurons: int, n_inputs: int,
                 activation: Union[str, ActivationFunction]):
        self.neurons = [
            Neuron(n_inputs, create_activation(activation))
            for _ in range(n_neurons)
        ]
        self.outputs = None

    def forward(self, inputs: np.ndarray) -> np.ndarray:
        """Propagation avant pour tous les neurones"""
        self.outputs = np.array([
            neuron.forward(inputs) for neuron in self.neurons
        ])
```

```
        return self.outputs
```

Listing 2 – Implémentation de la couche

3.2.3 Classe NeuralNetwork

La classe `NeuralNetwork` assemble les couches pour former un réseau complet :

```
class NeuralNetwork:
    def __init__(self):
        self.layers = []
        self.training_history = {
            'epochs': [], 'loss': [], 'accuracy': [],
            'val_loss': [], 'val_accuracy': []
        }

    def add_layer(self, layer: Layer) -> None:
        """Ajoute une couche au r seau"""
        self.layers.append(layer)

    def predict(self, inputs: np.ndarray) -> np.ndarray:
        """Effectue une pr diction"""
        output = inputs
        for layer in self.layers:
            output = layer.forward(output)
        return output
```

Listing 3 – Implémentation du réseau

3.3 Composants Logiciels Réutilisables

3.3.1 Pattern Builder pour la Construction

Le `NetworkBuilder` implémente le pattern Builder pour une construction flexible :

```
class NetworkBuilder:
    def __init__(self, input_size: int):
        self.input_size = input_size
        self.network = NeuralNetwork()
        self.last_layer_size = input_size

    def add_layer(self, n_neurons: int, activation: str = 'sigmoid'):
        layer = Layer(n_neurons, self.last_layer_size, activation)
        self.network.add_layer(layer)
        self.last_layer_size = n_neurons
        return self  # Fluent interface

    def build(self) -> NeuralNetwork:
        return self.network
```

Listing 4 – Pattern Builder

3.3.2 Pattern Strategy pour les Fonctions d'Activation

Les fonctions d'activation sont implémentées via le pattern Strategy :

```python
class ActivationFunction(ABC):
    @abstractmethod
    def forward(self, x: np.ndarray) -> np.ndarray:
        pass

    @abstractmethod
    def derivative(self, x: np.ndarray) -> np.ndarray:
        pass

class Sigmoid(ActivationFunction):
    def forward(self, x: np.ndarray) -> np.ndarray:
        return 1 / (1 + np.exp(-np.clip(x, -500, 500)))

    def derivative(self, x: np.ndarray) -> np.ndarray:
        fx = self.forward(x)
        return fx * (1 - fx)
```

Listing 5 – Pattern Strategy

4 Fonctionnalités Implémentées

4.1 Interface Graphique (GUI)

L'interface graphique Tkinter offre une expérience utilisateur complète :

- **Onglet Architecture** : Construction visuelle du réseau
  - Définition de la taille d'entrée
  - Ajout/suppression de couches
  - Choix du nombre de neurones et de la fonction d'activation
- **Onglet Entraînement** : Configuration et suivi
  - Chargement de données (CSV, JSON)
  - Paramètres d'apprentissage (learning rate, epochs, batch size)
  - Barre de progression et log en temps réel
- **Onglet Test** : Évaluation du modèle
  - Test sur jeu de données complet
  - Prédiction sur entrées individuelles
  - Affichage des métriques (loss, accuracy)
- **Onglet Visualisation** : Outils graphiques
  - Architecture du réseau
  - Historique d'entraînement
  - Distribution des poids
  - Frontières de décision (pour problèmes 2D)

4.2 Interface en Ligne de Commande (CLI)

Une interface CLI complète est également disponible :

```
# Construction du r seau
> build
Input size: 2
> add_layer
Number of neurons: 4
Activation function: relu
> add_layer
```

```
Number of neurons: 1
Activation function: sigmoid
> show
Network architecture: [2, 4, 1]

# Entra nement
> train
Learning rate: 0.01
Epochs: 100
Training completed. Final loss: 0.0234
```

Listing 6 – Exemple d'utilisation CLI

4.3 Gestion des Données

Le système offre plusieurs méthodes pour charger et générer des données :
— **Chargement CSV** : Support des fichiers avec features et targets
— **Chargement JSON** : Format structuré pour données complexes
— **Génération synthétique** : Création de datasets pour tests
— **Normalisation** : Min-Max et standardisation

5 Application des Principes du Génie Logiciel

5.1 Principes SOLID

5.1.1 Single Responsibility Principle (SRP)

Chaque classe a une responsabilité unique :
— `Neuron` : Calcul de la sortie d'un neurone
— `Layer` : Gestion d'un groupe de neurones
— `NetworkBuilder` : Construction de réseaux
— `TrainingController` : Gestion de l'entraînement

5.1.2 Open/Closed Principle (OCP)

Le système est extensible sans modification :
— Nouvelles fonctions d'activation via héritage
— Nouveaux formats de données via interfaces
— Nouvelles visualisations via modules séparés

5.1.3 Liskov Substitution Principle (LSP)

Toutes les fonctions d'activation sont interchangeables grâce à l'interface commune.

5.1.4 Interface Segregation Principle (ISP)

Les interfaces sont spécifiques :
— `ActivationFunction` pour les activations
— `DataLoader` pour le chargement de données

5.1.5 Dependency Inversion Principle (DIP)

Les modules de haut niveau dépendent d'abstractions, pas d'implémentations concrètes.

5.2 Design Patterns Utilisés

1. **Builder Pattern** : Construction flexible des réseaux
2. **Strategy Pattern** : Fonctions d'activation interchangeables
3. **Factory Pattern** : Création des fonctions d'activation
4. **Observer Pattern** : Callbacks pendant l'entraînement
5. **MVC Pattern** : Séparation modèle-vue-contrôleur

5.3 Tests et Qualité

5.3.1 Tests Unitaires

Couverture complète des composants critiques :

```python
def test_neuron_forward():
    """Test la propagation avant d'un neurone"""
    neuron = Neuron(n_inputs=2, activation=Sigmoid())
    inputs = np.array([0.5, -0.3])
    output = neuron.forward(inputs)

    assert 0 <= output <= 1  # Sigmoid output range
    assert isinstance(output, float)
```

Listing 7 – Exemple de test unitaire

5.3.2 Documentation

— Docstrings pour toutes les classes et méthodes
— Guide utilisateur complet
— README détaillé avec exemples
— Guide d'utilisation des datasets CSV

6 Jeux de Données et Tests

6.1 Datasets Créés

Onze jeux de données ont été générés pour tester différents scénarios :

Dataset	Type	Complexité
linear_classification.csv	Classification binaire	Simple
and_gate.csv	Porte logique AND	Simple
or_gate.csv	Porte logique OR	Simple
moons_classification.csv	Formes de lunes	Moyenne
circles_classification.csv	Cercles concentriques	Moyenne
blobs_classification.csv	Clusters multiples	Moyenne
multiclass_classification.csv	3 classes	Moyenne
complex_classification.csv	5 features	Élevée
spiral_classification.csv	Spirale	Très élevée
regression.csv	Régression linéaire	Simple
sine_regression.csv	Fonction sinus	Moyenne

TABLE 1 – Jeux de données disponibles

6.2 Résultats de Performance

Les tests montrent que le système peut :
— Atteindre >95% de précision sur les problèmes linéaires
— Résoudre des problèmes non-linéaires complexes (spirales, cercles)
— Gérer la classification multi-classes
— Effectuer des régressions linéaires et non-linéaires

7 Réponse aux Exigences de l'Énoncé

7.1 Système Interactif

Exigence : "Développer un système interactif permettant à l'utilisateur de créer et de paramétrer des algorithmes d'apprentissage"

Réalisation :
— Interface graphique complète avec 4 onglets spécialisés
— Interface CLI pour utilisation en ligne de commande
— Feedback en temps réel pendant l'entraînement
— Visualisations interactives

7.2 Composants Réutilisables

Exigence : "Considérer les réseaux de neurones comme des composants logiciels réutilisables"

Réalisation :
— Classes modulaires (Neuron, Layer, Network)
— Pattern Builder pour construction flexible
— Fonctions d'activation interchangeables
— Sauvegarde/chargement des modèles

7.3 Paramétrage Complet

Exigence : "Paramétrés par : nombre de couches, nombre de neurones, nombre de sorties et d'entrées"

Réalisation :

— Configuration dynamique de l'architecture
— Choix du nombre de neurones par couche
— Sélection des fonctions d'activation
— Paramètres d'entraînement ajustables

7.4 Test sur Jeux de Données

Exigence : "Tester sur des jeux de données"

Réalisation :

— 11 datasets pré-générés couvrant divers scénarios
— Support CSV et JSON
— Séparation automatique train/test
— Métriques de performance détaillées

8 Conclusion

Ce projet démontre avec succès comment les principes du génie logiciel peuvent être appliqués au développement de systèmes d'intelligence artificielle. En considérant les réseaux de neurones comme des composants logiciels réutilisables, nous avons créé un système :

— **Modulaire** : Chaque composant peut être utilisé indépendamment
— **Extensible** : Nouvelles fonctionnalités facilement ajoutables
— **Maintenable** : Code bien structuré et documenté
— **Testable** : Suite de tests complète
— **Utilisable** : Interfaces intuitives pour tous niveaux

Le système répond intégralement aux exigences de l'énoncé en fournissant :

1. Un modèle objet complet des réseaux de neurones
2. Des composants réutilisables et paramétrables
3. Des interfaces interactives pour la création et le test
4. Une application rigoureuse des principes du génie logiciel

Ce projet illustre parfaitement comment l'application des bonnes pratiques du génie logiciel améliore significativement la qualité, la maintenabilité et l'utilisabilité des systèmes d'IA.

9 Annexes

9.1 Guide d'Installation

```
# Cloner le projet
git clone [repository_url]
cd projet_geni_logiciel

# Installer les d pendances
```

```
pip install -r requirements.txt

# G n rer les datasets
python generate_datasets.py

# Lancer l'interface graphique
python main_gui.py

# Ou lancer l'interface CLI
python main_cli.py
```

9.2 Exemple d'Utilisation

```
# Construction d'un r seau
from src.controllers import NetworkBuilder

builder = NetworkBuilder(input_size=2)
network = builder.add_layer(4, 'relu') \
                 .add_layer(1, 'sigmoid') \
                 .build()

# Entra nement
from src.data import DataLoader

features, targets = DataLoader.load_from_csv('datasets/
    moons_classification.csv')
dataset = DataLoader.create_dataset(features, targets)
train_data, test_data = DataLoader.split_dataset(dataset, train_ratio
    =0.8)

history = network.train(train_data, epochs=100, learning_rate=0.01)

#   valuation
loss, accuracy = network.evaluate(test_data)
print(f"Test accuracy: {accuracy:.2%}")
```