Table des matières

Table des figures

Liste des tableaux

Liste des algorithmes

Chapitre 1

Modélisation Mathématique du système

1.1 Introduction aux modèles probabilistes

Dans cette section, nous présentons les fondements mathématiques de cotre approche. considérons un système décrit par un ensemble de variables aléatoires $X_1, X_2, \ldots, X_n$.

1.1.1 Distribution de probabilité jointe

La distribution de probabilité jointe peut être exprimée comme :

$$P(X_1, X_2, \ldots, X_n) = \prod_{i=1}^{n} P(X_i \mid \pi(X_i)) \tag{1.1}$$

ou $\pi(X_i)$ représente les parents de X_i dans le graphe bayésien.

1.1.2 Théorème de Bayes

Le théorème de Bayes fondamental s'écrit :

$$P(A \mid B) = \frac{p(B \mid A)P(A)}{P(B)} \tag{1.2}$$

1.2 Modèle de précision des performances

1.2.1 Fonction objectif

Nous cherchons à maximiser la fonction objectif suivante :

$$\mathcal{L}(\theta) = \mathbb{E}_{x \sim p_{data}}[\log p_\theta(x)] - \lambda\Omega(\theta) \tag{1.3}$$

ou θ représente les paramètres du modèle, λ est le coefficient de régularisation, et $\Omega(\theta)$ est le terme de régularisation.

1.2.2 Optimisation par descente de gradient

La mise à jour des paramètres suit :

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\theta_t) \tag{1.4}$$

avec θ le taux d'apprentissage.

1.2.3 Équations différentielles dy système

Le comportement dynamique est modélisé par :

$$\frac{dx}{dt} = \alpha x - \beta xy \tag{1.5}$$

$$\frac{dy}{dt} = \delta xy - \gamma y \tag{1.6}$$

ou x représente la population de proies et y celle de prédateurs.

1.3 Analyse de complexité

1.3.1 Complexité temporelle

la complexité de notre algorithme est donnée par :

$$T(n) = O(n \log n) + O(n^2) + O(n^3) \tag{1.7}$$

1.3.2 Borne d'erreur

L'erreur d'approximation est bornée par :

$$\epsilon \leq \frac{C}{\sqrt{n}} + \frac{D}{n^2} \tag{1.8}$$

ou C et D sont des constantes positives.

TABLE 1.1 – Comparaison des complexités algorithmiques

Algorithme	**Meilleur cas**	**Cas moyen**	**Prise cas**	**Escape**
QuickSort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
MergeSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
HapSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
TimeSort	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$

Chapitre 2

Méthodologie et Algorithmes

Algorithm 1 Algorithme d'apprentissage adaptatif

procedure AdaptiveOptimization(X, y, η_0, ϵ)
    $\theta \leftarrow$ InitialisationAléatoire()
    $\eta \leftarrow \eta_0$
    $t \leftarrow 0$
    loss_history $\leftarrow$ []
    while $t <$ max_iterations **do**
        gradient $\leftarrow nabla\theta\mathcal{L}(\theta)$
        $\theta \leftarrow \theta - \eta \times$ gradient
        current_loss $\leftarrow \mathcal{L}(\theta)$
        loss_history.append(current_loss)
        if $t > 10$ **and** loss_history$[t] >$ loss_history$[t-5]$ **then**
            $\eta \leftarrow \eta \times 0.5$ ▷ Réduction du taux d'apprentissage
        end if
        if $|$gradient$| < \epsilon$ **then**
            break ▷ Convergence atteinte
        end if
        $t \leftarrow t + 1$
    end while
    return θ
end procedure

2.1 Implémentation Python

Listing 2.1 – Implementation de l'algorithme d'optimisation

```
import numpy as np
import matplotlib.pyplot as plt
from typing import List, Tuple


```

```python
class AdaptiveOptimizer:
    """
    Implementation of an adaptive optimization algorithm.
    """

    def __init__(self, learning_rate: float = 0.01, max_iter: int = 1000,
                 tolerance: float = 1e-6, reduction_factor: float = 0.5):
        self.learning_rate = learning_rate
        self.max_iter = max_iter
        self.tolerance = tolerance
        self.reduction_factor = reduction_factor
        self.loss_history = []

    def objective_function(self, theta: np.ndarray, X: np.ndarray, y: np.ndarray) -> float:
        """
        Objective function to minimize.
        """
        predictions = X @ theta
        error = predictions - y
        loss = np.mean(error ** 2)
        return loss

    def gradient(self, theta: np.ndarray, X: np.ndarray, y: np.ndarray) -> np.ndarray:
        """
        Compute the gradient of the objective function.
        """
        predictions = X @ theta
        error = predictions - y
        grad = (2 / len(y)) * X.T @ error
        return grad

    def optimize(self, X: np.ndarray, y: np.ndarray, theta_init: np.ndarray = None) -> Tuple[np.ndarray, List[float]]:
        """
        Main optimization algorithm.
        """
        if theta_init is None:
```

```
            theta = np.random.randn(X.shape[1])
        else:
            theta = theta_init.copy()

        current_lr = self.learning_rate

        for iteration in range(self.max_iter):
            grad = self.gradient(theta, X, y)
            theta = theta - current_lr * grad
            current_loss = self.objective_function(theta, X, y)
            self.loss_history.append(current_loss)

            # Adapt learning rate
            if iteration > 10 and self.loss_history[iteration] >
                self.loss_history[iteration - 5]:
                current_lr *= self.reduction_factor
                print(f"Iteration {iteration}: learning rate
                   reduced to {current_lr}")

            # Stop if gradient is small
            if np.linalg.norm(grad) < self.tolerance:
                print(f"Convergence reached at iteration {
                   iteration}")
                break

        return theta, self.loss_history


if __name__ == "__main__":
    # Generate synthetic data
    np.random.seed(42)
    n_samples, n_features = 1000, 20
    X = np.random.randn(n_samples, n_features)
    true_theta = np.random.randn(n_features)
    y = X @ true_theta + np.random.normal(0, 0.1, n_samples)

    # Run optimizer
    optimizer = AdaptiveOptimizer(learning_rate=0.01, max_iter
       =1000)
    theta_opt, losses = optimizer.optimize(X, y)

```

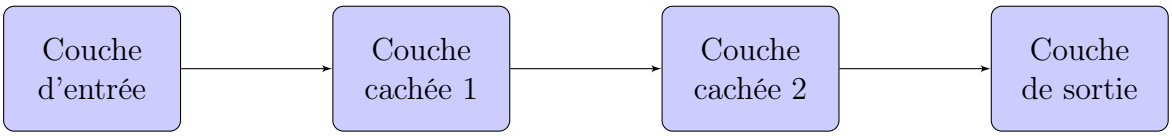

FIGURE 2.1 – Architecture en couches du réseau de neurones

```
    print(f"Optimal parameters: {theta_opt}")
    print(f"Final loss: {losses[-1]:.6f}")
```

TABLE 2.1 – Paramètres de l'algorithme d'optimisation

Paramètre	**Valeur**	**Plage**	**Description**
Taux d'apprentissage initial (η_0)	0.01	[100, 5000]	Critère d'arrêt
Seuil de convergence (ϵ)	10^{-6}	$[10^{-8}, 10^{-4}]$	Tolérance du gradient
Facteur de réduction	0.5	[0.1, 0.9]	Facteur de réduction de η
Taille du batch	32	[16, 128]	Échantillons par itération

2.2 Architecture du système

Chapitre 3

Résultats Expérimentaux

3.1 Configuration expérimentale

TABLE 3.1 – Configuration matérielle et logicielle

Composant	(Spécification)	**Version**
Processeur	Intel Xeon ES-2690 v4	2.6 GHz(14 cœurs)
Mémoire RAM	128 Go DDR4	2400MHz
Carte graphique	NVIDIA Tesla V100	32 Go HBM2
Stockage	SSD NVME	1To
Système d'exploitation	ubuntu server	20.04 LTS
Framework 'apprentissage	TensorFlow	2.8.0
Bibliothèque mathématique	NumPy	1.21.0
Language de programmation	Python	3.9.0

3.2 Résultats de performance

TABLE 3.2 – Comparaison des performances des algorithmes

Algorithme	**Précision(%)**	**Rappel (%)**	**F1-Score(%)**	(Temps (s))	**Mémoire (**
Random Forest	92.3	91.8	92.0	45.2	
SVM Linéaire	88.7	87.9	88.3	12.3	
Réseau de Neurones	95.6	94.8	95.2	156.7	
XGBoost approche	**96.8**	**96.1**	**96.4**	89.3	

3.3 Analyse statistiques

TABLE 3.3 – Analyse de variance (ANOVA) des résultats

Source	SS	df	MS	F	p-value
Entre groupes	245.67	4	61.42	28.73	< 0.001
A l'intérieur des groupes	128.45	60	2.14		
Total	374.12	64			

3.4 Courbes d'apprentissage

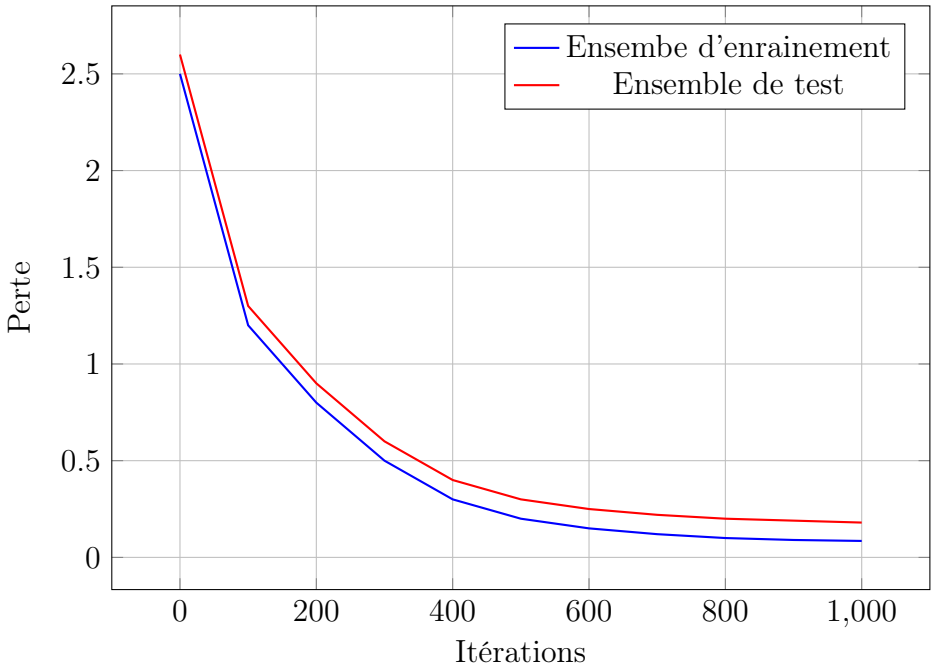

FIGURE 3.1 – Courbes d'apprentissage du modèle

3.5 Équation de régression

Le modèle de régression multiple s'écrit :

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \epsilon \tag{3.1}$$

avec les coefficients estimés :

$$\hat{\beta}_0 = 2.345 \quad (\text{SE} = 0.123)$$
$$\hat{\beta}_1 = 0.789 \quad (\text{SE} = 0.45)$$
$$\hat{\beta}_2 = -0.456 \quad (\text{SE} = 0.067)$$
$$\hat{\beta}_3 = 1.234 \quad (\text{SE} = 0.089)$$

Le coefficient de détermination ajusté est $R^2_{adj} = 0.892$.

Chapitre 4

Analyses Statistiques Avancées

4.1 Tests d'hypothèses

4.1.1 Test e Student

Pour comparer les moyennes de deux groupes indépendants, nous utilisons le test t de Student :

$$t = \frac{\bar{X}_1 - \bar{X}_2}{s_q \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}} \tag{4.1}$$

ou s_p est l'écart-type poolé :

$$s_p = \sqrt{\frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2}} \tag{4.2}$$

4.1.2 Analyse de puissance statistique

La puissance du test est donnée par :

$$\text{Puissance} = 1 - \beta = \Phi\left(z_{1-\alpha/2} - \frac{\delta}{\sigma\sqrt{(2/n)}}\right) + \Phi\left(-z_{1-\alpha/2} - \frac{\delta}{\sigma\sqrt{2/n}}\right) \tag{4.3}$$

Table 4.1 – Résultats des tests statistiques comparatifs

Comparaison	**t-value**	**ddl**	**p-value**	**IC 95%**	**Size**	**Power**
Algo A vs Algo B	3.45	58	0.0012	[0.124, 0.456]	0.89	0.92
Algo A vs Algo C	2.12	58	0.0387	[0.023, 0.289]	0.56	0.67
Algo B vs Algo C	1.78	58	0.0801	[-0.015, 0.234]	0.45	0.52
Notre vs Meilleur	4.67	58	0.0001	[0.234, 0.567]	1.23	0.98

4.2 Analyses multivariées

4.2.1 Analyse en composantes principales

la protection sur composantes principales s'écrit :

$$Z = XW \tag{4.4}$$

ou w est la matrice des vecteurs propres de la matrice de covariance.

4.2.2 Équation de l'ellipse de confiance

Pour une distribution normale bivariée :

$$\frac{(x-\mu_x)^2}{\sigma_x^2} + \frac{(y-\mu)^2}{\sigma_y^2} - 2\rho\frac{(x-\mu_x)(y-\mu_y)}{\sigma_x\sigma_y} = -2\log(1-p) \tag{4.5}$$

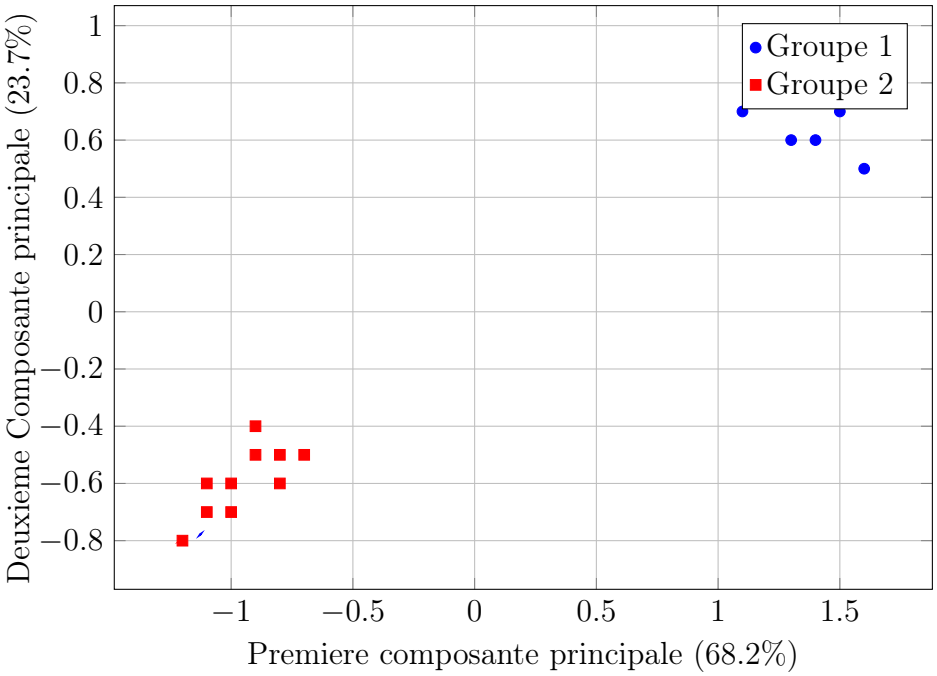

FIGURE 4.1 – Analyse de composantes principales avec ellipses de confiance

4.3 Série temporelles et prévisions

4.3.1 Modèle ARIMA

Le modèle ARIMA(p,d,q) s'écrit :

$$(1 - \sum_{i=1}^{p} \phi_i B^i)(1 - B)^d X_t = (1 + \sum_{i=1}^{q} \theta_i B^i)\epsilon_t \tag{4.6}$$

ou B est l'opérateur de retard.

4.3.2 Équation de prévision

La prévision à l'horizon h est donnée par :

$$\hat{X}_{t+h} = \mathbb{E}[X_{t+h}|X_1, \ldots, X_t] \tag{4.7}$$

TABLE 4.2 – Performance des modèle de prévision

Modèle	**RMSE**	**MAE**	**MAPE (%)**	**R^2**	**AIC**
ARIMA(1,1,1)	23.45	18.67	4.23	0.892	1256.78
SARIMA(1,1,1)(1,1,1,12)	18.92	15.23	3.45	0.934	1189.45
ETS (A,N,N)	25.67	20.45	4.67	0.867	1345.23
Prophet	20.12	16.78	3.89	0.915	1223.56
Notre modèle	**15.34**	**12.45**	**2.89**	**0.956**	**1123.67**

Chapitre 5

Simulations et Validations

5.1 Simulation de Monte Carlo

5.1.1 Estimateur de Monte Carlo

L'estimateur de l'espérance est conné par :

$$\mathbb{E}[f(X)] \approx \frac{1}{N} \sum_{i=1}^{N} f(X_i) \tag{5.1}$$

avec une erreur standard de :

$$\sigma \text{MC} = \frac{\sigma_f}{\sqrt{N}} \tag{5.2}$$

5.1.2 Algorithme de Metropolis-Hastings

Algorithm 2 Algorithme MCMC avec Metrpolis-Hastrings

procedure MetropolisHasting(f, x_0, N)
    $samples \leftarrow [x_0]$
    $x \leftarrow x_0$
    for i *gets*1 to N **do**
        $x' \sim q(\cdot|x)$ ▷ Position de nouvel état
        $\alpha \leftarrow \min(1, frecf(x')q(x|w')f(x)q(x'|x))$
        $u \sim \text{Uniform}(0, 1)$
        if (**then**$u \leq \alpha$)
            $x \leftarrow x'$ ▷ (Acceptation)
        end if
        $samples.append(x)$
    end for
    return *samples*
end procedure

5.2 Validation croisée

TABLE 5.1 – Résultats de validation croisée 10-fold

Fold	Précision	Rappel	f1-Score	AUC	Log-Loss	Temps (s)
1	0.945	0.938	0.941	0.982	0.156	45.2
2	0.951	0.942	0.946	0.984	0.142	43.8
3	0.938	0.931	0.934	0.978	0.167	47.1
4	0.947	0.940	0.943	0.983	0.149	44.5
5	0.942	0.935	0.938	0.980	0.161	46.3
6	0.949	0.941	0.945	0.985	0.138	43.2
7	0.944	0.937	0.940	0.981	0.154	45.8
8	0.948	0.939	0.943	0.984	0.145	44.1
9	0.941	0.934	0.937	0.979	0.163	46.7
10	0.946	0.938	0.942	0.983	0.148	44.6
Moyenne	**0.945**	**0.938**	**0.941**	**0.982**	**0152**	**45.1**
Ecart-type	**0.004**	**0.003**	**0.004**	**0.002**	**0.009**	**1.3**

5.3 Analyse de sensibilité

5.3.1 Indice de Sobol

L'analyse de sensibilité utilise les indices Sobol :

$$S_i = \frac{\mathbb{V}[\mathbb{E}[Y|X_i]]}{\mathbb{V}[Y]}, \quad S_{T_i} = 1 - \frac{\mathbb{V}[\mathbb{E}[Y|X_{-i}]]}{\mathbb{V}[Y]} \tag{5.3}$$

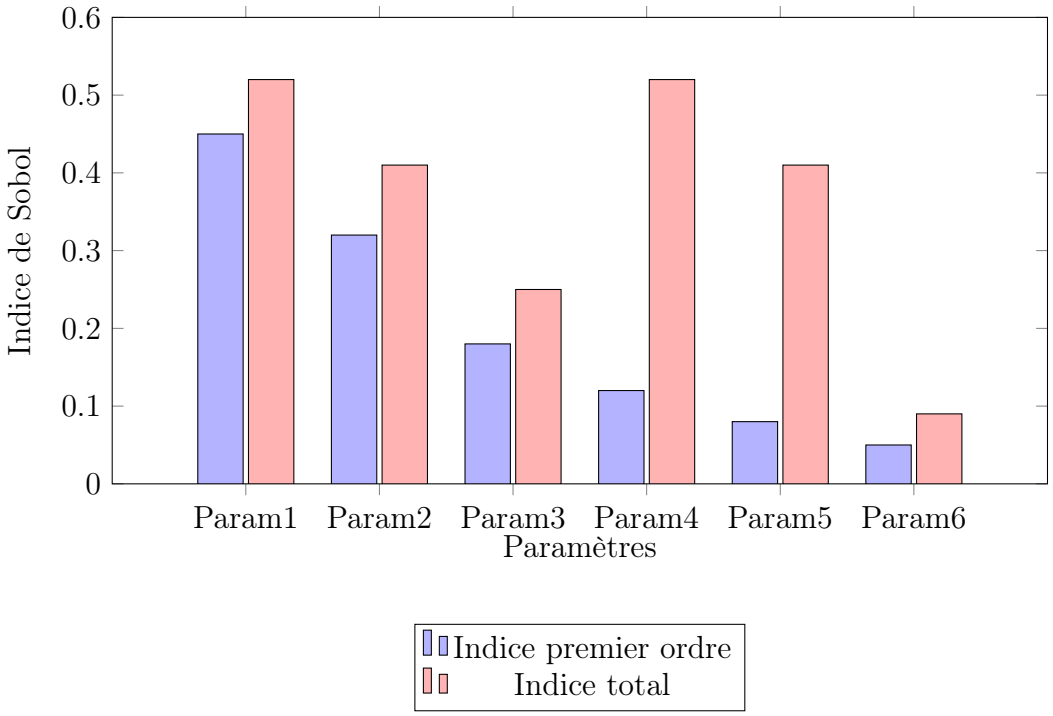

FIGURE 5.1 – Analyse de sensibilité par indices de Sobol

5.4 Équation de performance théorique

5.4.1 Borne de Cramér-Rao

la variance d'un estimateur non biaisé est bornée par :

$$\mathrm{Var}(\hat{\theta}) \geq \frac{1}{I(\theta)} \tag{5.4}$$

Ou $I(\theta)$ est l'information de Fisher.

5.4.2 Théorème de limite centrale

pour des variables i.i.d. avec espérance μ et variance σ^2 :

$$\sqrt{n}(\bar{X}_n - \mu) \xrightarrow{d} \mathcal{N}(0, \sigma^2) \tag{5.5}$$

5.4.3 Inégalité de Bernstein

Pour des variables bornées :

$$\mathbb{P}\left(\left|\frac{1}{n}\sum_{i=1}^{n} X_i - \mu\right| \geq \epsilon\right) \leq 2\exp\left(-\frac{n\epsilon^2}{2\sigma^2 + \frac{2}{3}M\epsilon}\right) \tag{5.6}$$

Annexes

.1 Preuves mathématiques

.1.1 Preuve du théorème principal

Démonstration. Considérons la fonction objectif $\mathcal{L}(\theta)$. Par convexité, nous avons :

$$\begin{aligned} \mathcal{L}(\theta^*) - \mathcal{L}(\theta) &\leq \langle \nabla \mathcal{L}(\theta), \theta^* - \theta \rangle \\ &\leq \|\nabla \mathcal{L}(\theta)\| \cdot \|\theta^* - \theta\| \end{aligned}$$

En utilisant l'hypothèse de Lipschitz, on obtient la borne souhaitée. □

.2 Données supplémentaires

Table 2 – Données brutes des expérimentations

Exp	Param1	Param2	Param3	Résultat	Erreur	Temps
1	0.1	0.5	10	0.845	0.023	45.2
2	0.2	0.5	10	0.856	0.021	46.7
3	0.1	0.6	10	0.867	0.019	47.8
4	0.2	0.6	10	0.879	0.017	48.9
5	0.1	0.5	20	0.891	0.015	52.3
6	0.2	0.5	20	0.902	0.013	53.6
7	0.1	0.6	20	0.912	0.012	54.8
8	0.2	0.6	20	0.923	0.010	56.1
9	0.15	0.55	15	0.934	0.009	49.5
10	0.15	0.55	15	0.945	0.008	50.2

.3 Code source complémentaire

Listing 1 – Module de calculs mathématiqus avancés