## **📄 Présentation du Projet NAPCAS**

### **🧩 1. Contexte**

Le projet **NAPCAS** (Neurone Adaptatif Parcimonieux avec Connexions et Similitude) est un framework de deep learning écrit en **C++**, exposé à Python via **Pybind11**, qui vise à reproduire les fonctionnalités d’un framework comme **PyTorch**.

Il permet :

* La création de réseaux de neurones.
* L'entraînement via la descente de gradient.
* La gestion automatique des gradients.
* Le support de couches linéaires, de fonctions d'activation, de pertes, d'optimiseurs, etc.
* L’intégration de connexions adaptatives et de mémoire de chemins.

### **🎯 2. Objectifs du projet**

#### **✅ Objectif principal**

Créer un framework complet de deep learning en C++, avec une interface Python pour faciliter son utilisation dans des applications de machine learning.

#### **✅ Objectifs secondaires**

* Implémenter le modèle **NAPCAS**.
* Gérer les données via DataLoader.
* Générer des gradients via Autograd.
* Utiliser pybind10 pour exposer les classes C++ en Python.
* Permettre l’entraînement de modèles simples (MLP, CNN).
* Tester avec des datasets comme **MNIST** ou **CIFAR-10**.

### **🔧 3. Méthodologie**

#### **✅ Architecture du projet**

napcas-framework/  
├── cpp/  
│ ├── include/  
│ │ ├── linear.h  
│ │ ├── conv2d.h  
│ │ ├── activation.h  
│ │ ├── loss.h  
│ │ ├── optimizer.h  
│ │ ├── data\_loader.h  
│ │ ├── autograd.h  
│ │ ├── module.h  
│ │ ├── nncell.h  
│ │ ├── tensor.h  
│ │ └── napcas.h  
│ └── src/  
│ ├── python\_bindings.cpp  
│ ├── linear.cpp  
│ ├── conv2d.cpp  
│ ├── activation.cpp  
│ ├── loss.cpp  
│ ├── optimizer.cpp  
│ ├── data\_loader.cpp  
│ ├── autograd.cpp  
│ ├── module.cpp  
│ ├── nncell.cpp  
│ ├── tensor.cpp  
│ └── napcas.cpp  
├── python/  
│ ├── setup.py  
│ ├── napcas.py  
│ └── modules/  
│ ├── \_\_init\_\_.py  
│ ├── linear.py  
│ ├── activation.py  
│ ├── loss.py  
│ ├── optimizer.py  
│ ├── data\_loader.py  
│ └── autograd.py  
└── CMakeLists.txt

#### **✅ Technologies utilisées**

* **C++** : pour la performance.
* **Pybind11** : pour l’exposition des classes C++ en Python.
* **CMake** : pour la compilation.
* **Python** : pour l’interface utilisateur.
* **Tensor** : classe principale pour gérer les tenseurs.
* **Module** : classe abstraite pour les couches.
* **Optimiseurs** : SGD, Adam.
* **Fonctions de perte** : MSE, CrossEntropyLoss.
* **Modules d'activation** : ReLU, Sigmoid, Tanh.
* **DataLoader** : pour charger des données.
* **Autograd** : pour la gestion des gradients.

### **🛠️ 4. Problèmes rencontrés et solutions apportées**

#### **✅ Erreur : Classes non déclarées (Linear, Conv2d, ReLU, etc.)**

* **Cause** : Les classes C++ n’étaient pas correctement incluses ou définies.
* **Solution** : Ajout des fichiers .h et .cpp pour chaque classe, avec les bonnes déclarations.

#### **✅ Erreur : Fichier manquant python\_bindings.cpp**

* **Cause** : Le fichier python\_bindings.cpp était absent ou mal configuré.
* **Solution** : Création d’un fichier complet pour les bindings.

#### **✅ Erreur : Chemins incorrects dans CMakeLists.txt**

* **Cause** : Des chemins absolus (/cpp/src/...) étaient utilisés.
* **Solution** : Utilisation de chemins relatifs (src/...) dans CMakeLists.txt.

#### **✅ Erreur : ‘Linear’ was not declared in this scope**

* **Cause** : Le fichier python\_bindings.cpp ne trouvait pas la classe Linear.
* **Solution** : Ajout des #include "include/linear.h" dans python\_bindings.cpp.

#### **✅ Erreur : ‘py’ has not been declared**

* **Cause** : Le namespace py n’était pas déclaré.
* **Solution** : Ajout de namespace py = pybind11; au début du fichier python\_bindings.cpp.

### **🧠 5. Modèle NAPCAS**

#### **🔹 Définition**

Le modèle **NAPCAS** est inspiré de l'idée de **neurones adaptatifs** avec des **connexions dynamiques** et une **mémoire de chemins**.

#### **🔹 Caractéristiques**

* **Connexions adaptatives** entre les neurones.
* **Gestion de la similitude** entre les chemins.
* **Pruning dynamique** des connexions non utiles.
* **Mise à jour conditionnelle** des paramètres selon les besoins.

#### **🔹 Avantages**

* Réduction de la complexité du réseau.
* Optimisation des ressources.
* Flexibilité dans la gestion des connexions.

#### **🔹 Inconvénients**

* Complexité d’implémentation.
* Besoin d’une gestion rigoureuse des gradients.
* Difficulté de mise à l’échelle pour des modèles très complexes.

### **🔍 6. Analyse SWOT du projet NAPCAS**

|  |  |  |
| --- | --- | --- |
| **Forces** | | **Faiblesses** |
| - Code performant en C++ - Interface Python simple - Modèle NAPCAS innovant - Support de la gestion des gradients | | - Complexité d’implémentation - Difficulté de maintenance - Absence de documentation complète |
| **Opportunités** | **Menaces** | |
| - Intégration possible avec des datasets réels - Possibilité de développement futur - Réutilisation dans des projets de recherche | - Concurrence avec des frameworks existants - Difficulté de compatibilité avec les versions Python - Risque de bugs liés à la gestion des classes C++ dans Python | |

### **🧪 7. Résultats obtenus**

* Compilation réussie du projet après correction des erreurs.
* Exposition correcte des classes C++ via Pybind11.
* Tests unitaires réussis avec des classes comme Linear, ReLU, Sigmoid, Tanh, MSELoss, SGD, Adam, DataLoader, Autograd, NAPCAS.
* Le projet est maintenant prêt pour des tests plus avancés.

### **🧪 8. Résultats attendus**

* Un framework complet, fonctionnel et compatible avec des réseaux de taille moyenne.
* Une interface Python simple et intuitive.
* Une gestion optimisée des connexions et des poids.
* Une utilisation de la bibliothèque pybind11 pour l’exposition des classes C++ en Python.

### **🧪 9. Tests effectués**

* Test de la classe Tensor (méthodes forward, backward, update, reshape, fill, zero\_grad).
* Test de la classe Linear (initialisation, forward, backward, update).
* Test de Sigmoid et Tanh (forward, backward).
* Test de MSELoss, CrossEntropyLoss, SGD, Adam, DataLoader, Autograd, NAPCAS.

### **🧪 10. Tests à faire**

* ✅ Tester avec **MNIST**.
* ✅ Tester avec **CIFAR-10**.
* ✅ Vérifier la gestion des gradients.
* ✅ Comparer les performances avec **PyTorch**.
* ✅ Ajouter des outils de visualisation (matplotlib, numpy).

### **🧩 11. Structure finale du projet**

napcas-framework/  
├── cpp/  
│ ├── include/  
│ │ ├── linear.h  
│ │ ├── conv2d.h  
│ │ ├── activation.h  
│ │ ├── loss.h  
│ │ ├── optimizer.h  
│ │ ├── data\_loader.h  
│ │ ├── autograd.h  
│ │ ├── module.h  
│ │ ├── nncell.h  
│ │ ├── tensor.h  
│ │ └── napcas.h  
│ └── src/  
│ ├── python\_bindings.cpp  
│ ├── linear.cpp  
│ ├── conv2d.cpp  
│ ├── activation.cpp  
│ ├── loss.cpp  
│ ├── optimizer.cpp  
│ ├── data\_loader.cpp  
│ ├── autograd.cpp  
│ ├── module.cpp  
│ ├── nncell.cpp  
│ ├── tensor.cpp  
│ └── napcas.cpp  
├── python/  
│ ├── setup.py  
│ ├── napcas.py  
│ └── modules/  
│ ├── \_\_init\_\_.py  
│ ├── linear.py  
│ ├── activation.py  
│ ├── loss.py  
│ ├── optimizer.py  
│ ├── data\_loader.py  
│ └── autograd.py  
└── CMakeLists.txt

### **🧩 12. Exemple de code Python**

from modules import Linear, ReLU, MSELoss, SGD, DataLoader  
  
model = [  
 Linear(784, 128),  
 ReLU(),  
 Linear(128, 10)  
]  
  
criterion = MSELoss()  
optimizer = SGD(model)  
  
# Entraînement  
for epoch in range(epochs):  
 for inputs, targets in train\_loader:  
 outputs = model[0].forward(inputs)  
 outputs = model[1].forward(outputs)  
 outputs = model[2].forward(outputs)  
  
 loss = criterion.forward(outputs, targets)  
 grad = criterion.backward(outputs, targets)  
  
 model[2].backward(grad)  
 model[1].backward(grad)  
 model[0].backward(grad)  
  
 optimizer.step()

### **🧩 13. Conclusion**

Le projet **NAPCAS** est maintenant **compilé et fonctionnel**, mais il reste encore plusieurs fonctionnalités à implémenter :

* ✅ Implémentation de Conv2d.
* ✅ Tests avec des datasets réels (MNIST, CIFAR-10).
* ✅ Documentation complète.
* ✅ Gestion des réseaux plus complexes (CNN, RNN, LSTM).

### **🧩 14. Prochaines étapes**

* ✅ Implémenter les couches convolutionnelles (Conv2d).
* ✅ Ajouter des exemples complets de réseaux.
* ✅ Ajouter des outils de visualisation.
* ✅ Finaliser la documentation technique.
* ✅ Ajouter des tests supplémentaires.

### **🧩 15. Liens utiles**

* [Documentation de Pybind11](https://pybind11.readthedocs.io/en/stable/)
* [Documentation de CMake](https://cmake.org/documentation/)
* [Documentation de PyTorch](https://pytorch.org/docs/stable/index.html)