COMPARAISON DU TEMPS D'INFERENCE ENTRE PYTHON ET C++ EN CPU APRES L'ENTRAINEMENT D'UN CNN SUR MNIST

- systemes numeriques C++-

Nom : KHACEF

Prenom : Melisa

M1 ESTEL

Encadrant : Valery Leroy

1. Introduction

Dans un contexte où l'efficacité des systèmes embarqués devient cruciale, l'optimisation des temps de traitement des modèles de deep learning est une priorité. Ces modèles, bien que très performants, sont souvent limités par la capacité de calcul et la consommation énergétique des unités de traitement embarquées. Afin d'adresser ce défi, il est essentiel de comparer les différents environnements d'exécution disponibles pour ces modèles.

Dans ce projet, nous avons choisi de travailler sur le dataset MNIST, un ensemble de données largement utilisé pour la reconnaissance de chiffres manuscrits. Nous avons conçu et entraîné un réseau de neurones convolutifs (CNN) en Python à l'aide de PyTorch. Ce modèle a ensuite été exporté vers le format TorchScript pour permettre son exécution en C++ via la bibliothèque LibTorch.

L'objectif principal de cette étude est de comparer les temps d'inférence entre Python et C++ en utilisant un CPU. Pour ce faire, nous avons mesuré le temps requis pour charger le modèle, préparer les données et effectuer l'inférence sur les 10 000 exemples du jeu de test de MNIST. Cette comparaison permettra d'évaluer les avantages potentiels de l'utilisation de C++ pour des applications à contrainte de temps ou de ressources.

Dans les sections suivantes, nous présenterons la méthodologie détaillée, les résultats obtenus et une discussion des différences observées entre les deux environnements.

2-Méthodologie

2.1. Environnement et Outils :

L'entraînement du modèle a été réalisé dans un environnement Python utilisant la bibliothèque **PyTorch**. Le modèle a été exécuté sur un processeur central (CPU), faute de GPU disponible pour cette tâche. Bien que PyTorch soit également compatible avec les unités de traitement graphique (GPU), le choix du CPU a permis de tester l'efficacité du modèle dans un environnement de calcul plus accessible, souvent utilisé dans des systèmes embarqués ou des dispositifs avec ressources limitées.

Les outils principaux utilisés sont :

- **PyTorch** : bibliothèque de calculs scientifiques utilisée pour définir, entraîner et évaluer le modèle de réseau de neurones convolutif (CNN).
- **TorchVision** : bibliothèque complémentaire à PyTorch, utilisée pour le traitement des images et le chargement des jeux de données comme MNIST.

2-2. Jeu de Données MNIST :

Le jeu de données **MNIST** (Modified National Institute of Standards and Technology) est un ensemble de données très utilisé pour la classification d'images, en particulier pour les tâches d'apprentissage supervisé en reconnaissance de chiffres manuscrits. Il se compose de 60 000 images d'entraînement et de 10 000 images de test, représentant des chiffres de 0 à 9, chacun étant une image en niveaux de gris de taille 28x28 pixels.

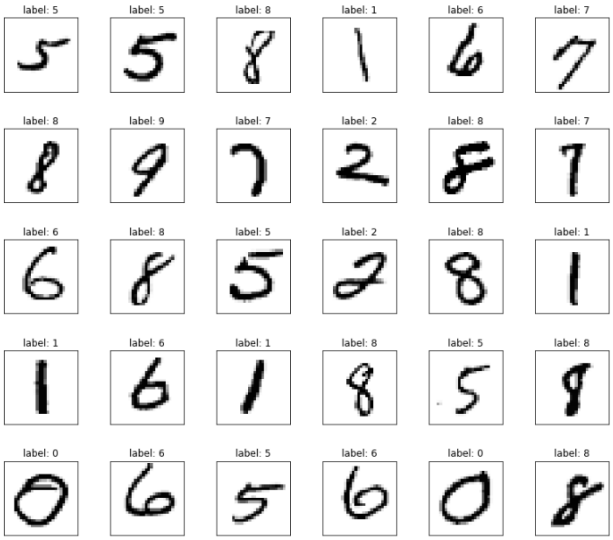

Figure1-MNIST dataset

Les étapes suivantes ont été appliquées au jeu de données :

- **Transformation des données** : Chaque image du jeu de données a été convertie en un tenseur à l'aide de la fonction ToTensor() de PyTorch. De plus, les pixels des images ont été normalisés pour avoir une moyenne de 0.5 et un écart type de 0.5 à l'aide de la transformation Normalize((0.5,), (0.5,)). Cela aide à accélérer la convergence pendant l'entraînement en ayant des données centrées autour de 0.
- **Chargement du jeu de données** : Le jeu de données d'entraînement a été chargé à l'aide de torchvision.datasets.MNIST et distribué en batchs de 64 images à l'aide de DataLoader. Ce dernier permet également de mélanger les données (avec shuffle=True), assurant ainsi que l'ordre des images ne biaise pas l'entraînement.

2-3. Entraînement du Modèle :

Le modèle utilisé est un réseau de neurones convolutif (CNN) implémenté dans PyTorch. Ce modèle est conçu pour classer les images de chiffres manuscrits du jeu de données MNIST. L'architecture du modèle se compose de deux couches convolutionnelles suivies de couches entièrement connectées. Voici un résumé des étapes d'entraînement :

- **Définition du modèle** : Le modèle CNN est défini avec deux couches de convolution, chacune suivie d'une activation ReLU et d'un pooling max. Ensuite, les caractéristiques extraites sont aplaties et passées à travers deux couches entièrement connectées. La sortie finale du modèle correspond aux 10 classes possibles (les chiffres de 0 à 9).
- **Fonction de perte** : La fonction de perte utilisée pour l'entraînement est la **CrossEntropyLoss**, qui est adaptée pour les tâches de classification multi-classes.
- **Optimiseur** : L'optimiseur choisi est **Adam**. Cet optimiseur est une version améliorée de la descente de gradient, qui ajuste automatiquement les taux d'apprentissage pour chaque paramètre, ce qui le rend particulièrement efficace pour des modèles complexes.
- **Boucle d'entraînement** : L'entraînement a été effectué pendant 5 époques. À chaque époque, les images sont passées dans le modèle par batchs. La sortie du modèle est comparée à la

vérité terrain (les étiquettes des images) pour calculer la perte. Ensuite, l'optimiseur effectue une mise à jour des poids du modèle afin de réduire cette perte. La perte moyenne par époque est calculée et affichée à la fin de chaque époque pour suivre la progression de l'entraînement.

- **Sauvegarde du modèle** : À la fin de l'entraînement, le modèle est converti en format **TorchScript** à l'aide de torch.jit.script(model.cpu()). Cela permet d'optimiser l'exécution du modèle et de le rendre indépendant de l'environnement d'entraînement. Le modèle est ensuite sauvegardé sous le nom "**mnist_model.pt**".

dnn_inference > train_mnist.py > ...

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms

# Define the CNN model
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 64 * 7 * 7)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Prepare MNIST dataset
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
train_dataset = torchvision.datasets.MNIST(root="./data", train=True, download=True, transform=transform)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=64, shuffle=True)

# Training setup
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = CNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
for epoch in range(5):  # Train for 5 epochs
    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print(f"Epoch {epoch+1}, Loss: {running_loss / len(train_loader)}")

# Save the model in TorchScript format
scripted_model = torch.jit.script(model.cpu())
scripted_model.save("mnist_model.pt")
print("Model saved as 'mnist_model.pt'")
```

Figure2- Code de l'entraînement du modèle CNN sur MNIST

2-4. Inférence :

L'inférence est la phase du processus d'apprentissage automatique où un modèle préalablement entraîné est utilisé pour effectuer des prédictions sur de nouvelles données. Contrairement à l'entraînement, où le modèle ajuste ses poids pour minimiser l'erreur, l'inférence utilise des poids fixes (appris pendant l'entraînement) pour générer des prédictions sur des données inconnues.

2-4-1. Inférence avec Python

Le code charge un modèle MNIST préalablement entraîné et sauvé en format TorchScript. Il prépare le dataset de test MNIST, puis effectue des prédictions (inférence) sur les images de test. L'accuracy du modèle est calculée en comparant les prédictions avec les étiquettes réelles. Le nombre total de lots traités est également comptabilisé.

Bibliothèques utilisées :

- **PyTorch** (torch, torchvision) pour gérer le modèle, les transformations de données, et le dataset.
- **TorchScript** pour charger le modèle préalablement entraîné.
- **DataLoader** pour itérer sur le dataset par lots.

dnn_inference > test_mnist_pytorch.py > ...

```python
import torch
import torchvision
import torchvision.transforms as transforms
import time  # Ajouté pour utiliser time.time()

# Charger le modèle
device = torch.device("cpu")
model = torch.jit.load("mnist_model.pt").to(device)
model.eval()

# Préparer le dataset de test MNIST
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
test_dataset = torchvision.datasets.MNIST(root="./data", train=False, download=True, transform=transform)

# Créer un DataLoader avec la taille du batch de 64
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=64, shuffle=False)

# Tester l'accuracy et mesurer le temps d'inférence
correct = 0
total = 0

# Mesurer le temps d'inférence
start_inference_time = time.time()  # Début du chronométrage pour l'inférence

with torch.no_grad():
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

end_inference_time = time.time()  # Fin du chronométrage pour l'inférence
inference_duration = end_inference_time - start_inference_time

accuracy = 100 * correct / total
```

```python
# Afficher les résultats
print(f"Total inference time: {inference_duration:.4f} seconds")
print(f"Accuracy: {accuracy:.2f}%")

# Vérification du nombre de lots traités
num_batches = 0
with torch.no_grad():
    for images, labels in test_loader:
        num_batches += 1

print(f"Nombre total de lots traités : {num_batches}")

```

Figure3- Code d'inférence sur MNIST avec PyTorch

2-4-2. Inférence avec C++ :

Le code C++ que tu as mis en place est destiné à effectuer l'inférence sur un modèle MNIST entraîné avec PyTorch. Il commence par charger le modèle TorchScript en utilisant torch::jit::load, ce qui permet d'utiliser un modèle pré-entrainé pour l'inférence sans avoir à réentraîner. Ensuite, il charge le dataset MNIST à l'aide de la bibliothèque LibTorch, en appliquant des transformations pour normaliser les images et les convertir en tenseurs. Un DataLoader est utilisé pour charger les données en lots de 64, facilitant ainsi le traitement par le modèle.

Le cœur du processus est l'inférence proprement dite, qui se fait sans calcul de gradients (grâce à torch::NoGradGuard). Le modèle fait des prédictions sur chaque lot d'images, et les résultats sont comparés aux étiquettes réelles pour calculer l'accuracy. Après avoir traité toutes les données, l'accuracy totale est affichée, ainsi que le nombre de lots traités.

Bibliothèques utilisées :

- **<torch/torch.h>** : Bibliothèque principale de LibTorch pour accéder aux fonctionnalités de PyTorch (telles que la gestion des tenseurs, des modèles, etc.).
- **<torch/script.h>** : Utilisée pour charger et exécuter des modèles TorchScript via torch::jit::load.
- **<torch/data/datasets/mnist.h>** : Permet de charger et d'accéder au dataset MNIST dans LibTorch.
- **<iostream>** : Fournit les fonctionnalités d'entrée/sortie pour afficher les résultats et messages dans la console.
- **<fstream>** : Utilisée pour la gestion des fichiers (bien que dans ce code, elle ne soit pas utilisée directement).
- **<chrono>** : Utilisée pour mesurer les durées des différentes étapes (chargement du modèle, traitement des données, etc.), même si tu souhaites l'enlever plus tard.

dnn_inference > test_mnist_libtorch.cpp > ...

```cpp
#ifndef TEST_MNIST_LIBTORCH_H  // Protection d'inclusion pour éviter les redéfinitions
#define TEST_MNIST_LIBTORCH_H
#include <torch/torch.h>
#include <torch/script.h>  // Nécessaire pour torch::jit::load
#include <torch/data/datasets/mnist.h>  // Correctement inclus ici
#include <iostream>
#include <fstream>
#include <chrono>

// Charger le modèle avec torch::jit::load
torch::jit::script::Module load_model(const std::string& model_path) {
    torch::jit::script::Module model;
    try {
        model = torch::jit::load(model_path); // Charger le modèle TorchScript
    } catch (const c10::Error& e) {
        std::cerr << "Erreur lors du chargement du modèle : " << e.msg() << std::endl;
        exit(1);
    }
    return model;
}

int main() {
    // Chemin vers ton modèle sauvegardé
    std::string model_path = "/home/meli/dnn_inference/mnist_model.pt";

    // Charger le modèle
    auto model = load_model(model_path);

    // Charger les données MNIST (en utilisant la bibliothèque de LibTorch)
    auto dataset = torch::data::datasets::MNIST("/home/meli/dnn_inference/data/MNIST/raw/MNIST/raw",
     torch::data::datasets::MNIST::Mode::kTest)// Mode test
        .map(torch::data::transforms::Normalize<>{0.5, 0.5})  // Normalisation des données
        .map(torch::data::transforms::Stack<>()); // Conversion en tensor

    // Créer un DataLoader pour charger les données par lot
    auto data_loader = torch::data::make_data_loader(std::move(dataset), 64); // Taille du batch = 64

    size_t correct = 0; // Nombre de prédictions correctes
    size_t total = 0;   // Nombre total d'exemples
    size_t batch_count = 0; // Compteur de lots

    // Désactiver les gradients pour l'inférence
    torch::NoGradGuard no_grad;

    // Mesurer le temps total d'inférence
    auto inference_start = std::chrono::high_resolution_clock::now();

    // Boucle sur les données pour faire l'inférence et calculer l'accuracy
    for (auto& batch : *data_loader) {
        batch_count++; // Incrémenter le compteur de lots

        torch::Tensor inputs = batch.data.to(torch::kFloat32);  // Assurer que les données sont en float32
        torch::Tensor labels = batch.target; // Les vraies étiquettes

        // Faire l'inférence
        torch::Tensor outputs = model.forward({inputs}).toTensor();

        // Prédictions (classes avec la probabilité la plus élevée)
        torch::Tensor predictions = outputs.argmax(1);

        // Calculer les prédictions correctes
        correct += predictions.eq(labels).sum().item<int64_t>();
        total += labels.size(0); // Ajouter le nombre d'exemples du batch
    }

    auto inference_end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<float> inference_duration = inference_end - inference_start;
    std::cout << "Temps total d'inférence : " << inference_duration.count() << " secondes" << std::endl;

    // Calculer l'accuracy
    float accuracy = static_cast<float>(correct) / total * 100.0;

    // Afficher les résultats
    std::cout << "Précision (accuracy) : " << accuracy << "%" << std::endl;
    std::cout << "Nombre total de lots traités : " << batch_count << std::endl; // Afficher le nombre de lot

    return 0;
}

#endif // TEST_MNIST_LIBTORCH_H  // Fermeture de la protection d'inclusion
```

Figure4- Code d'inférence sur MNIST avec LibTorch en C++

Compilation avec CMake :

1. **Fichier CMakeLists.txt** :
    - Le fichier CMakeLists.txt est utilisé pour configurer le projet et faciliter la compilation.
    - CMakeLists.txt spécifie le chemin vers les fichiers de configuration de LibTorch avec set(CMAKE_PREFIX_PATH ...), ce qui permet à CMake de trouver et de lier les bibliothèques nécessaires à l'exécution du modèle.
    - Le projet est configuré en mode Release pour optimiser les performances.
    - Il inclut le fichier source C++ principal (test_mnist_libtorch.cpp) et lie les bibliothèques nécessaires avec target_link_libraries.
    - Le standard C++ utilisé est C++17, spécifié avec set_property(TARGET inference PROPERTY CXX_STANDARD 17).

dnn_inference > CMakeLists.txt

```
cmake_minimum_required(VERSION 3.12)

# Set project name and build type
project(DNNInference)
set(CMAKE_BUILD_TYPE Release)

# Set libtorch paths (mettre à jour les chemins si nécessaire)
set(CMAKE_PREFIX_PATH "/home/meli/dnn_inference/libtorch/share/cmake/Torch")

# Find Torch
find_package(Torch REQUIRED)

# Créer un répertoire de build séparé (optionnel mais recommandé)
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)

# Ajouter les fichiers sources
add_executable(inference /home/meli/dnn_inference/test_mnist_libtorch.cpp)

# Lier les bibliothèques
target_link_libraries(inference "${TORCH_LIBRARIES}")

# Définir le standard C++
set_property(TARGET inference PROPERTY CXX_STANDARD 17)
```

Figure5- Fichier CMakeLists.txt pour la compilation du projet d'inférence en C++

3-Résultats :

3.1- Python :

```
(dnn) meli@DESKTOP-OQK1SQ7:~/dnn_inference$ python3 test_mnist_pytorch.py
Total inference time: 2.3687 seconds
Accuracy: 99.05%
Nombre total de lots traités : 157
```

Figure6-Affichage des résultats d'inférence sur Python (temps et accuracy)

3.2-C++ :

```
(dnn) meli@DESKTOP-OQK1SQ7:~/dnn_inference/build$ cmake ..
-- Configuring done (2.0s)
-- Generating done (0.0s)
-- Build files have been written to: /home/meli/dnn_inference/build
(dnn) meli@DESKTOP-OQK1SQ7:~/dnn_inference/build$ make
[100%] Built target inference
(dnn) meli@DESKTOP-OQK1SQ7:~/dnn_inference/build$ ./bin/inference
Temps total d'inférence : 1.23274 secondes
Précision (accuracy) : 99.05%
Nombre total de lots traités : 157
```

Figure 7- Affichage des résultats d'inférence sur C++ (temps et accuracy)

4-Discussion des Résultats d'Inférence :

Dans cette étude, nous avons comparé les performances de l'inférence d'un modèle de réseau de neurones convolutifs (CNN) entraîné sur le jeu de données MNIST en utilisant Python et C++.

4.2-Nombre de lots traités :

- **Nombre de lots traités en Python : 157**
- **Nombre de lots traités en C++ : 157**

Dans les deux langages, nous avons traité le même nombre de lots, avec une taille de lot (batch size) identique de 64 exemples. Cela garantit que l'évaluation a été effectuée de manière équivalente dans les deux cas, assurant ainsi la comparabilité des performances.

4.2-Précision du modèle :

- **Précision (accuracy) en Python : 99.05%**
- **Précision (accuracy) en C++ : 99.05%**

Il est important de souligner que la **précision du modèle** est exactement la même dans les deux langages, à savoir **99.05%**. Cette constance de l'accuracy démontre que le modèle de réseau de neurones, lorsqu'il est chargé et utilisé pour effectuer des prédictions, produit des résultats identiques, indépendamment du langage de programmation utilisé. Le modèle, les poids et les fonctions d'activation étant les mêmes dans les deux environnements, la précision n'est pas affectée par le choix du langage pour l'inférence.

4.3-Temps d'inférence

- **Temps d'inférence total en Python : 2.3687 secondes**
- **Temps d'inférence total en C++ : 1.23274 secondes**

Le **temps d'inférence** entre Python et C++ varie de manière significative, avec **C++ étant presque deux fois plus rapide** que Python. Cette différence peut être attribuée à plusieurs facteurs, notamment :

1. **Optimisation du langage C++ :** C++ étant un langage compilé, il bénéficie d'une exécution plus rapide comparée à Python, qui est un langage interprété. En C++, le code est directement converti en code machine, ce qui permet une gestion plus efficace des opérations de bas niveau et donc des temps d'exécution plus courts.
2. **LibTorch vs PyTorch :** Bien que PyTorch en Python et LibTorch en C++ soient construits sur la même base, la version C++ est généralement mieux optimisée pour les tâches de calcul intensif. En Python, PyTorch peut introduire un overhead dû à l'interprétation du code et à des abstractions supplémentaires, tandis que LibTorch est plus proche du matériel et plus rapide dans l'exécution des tâches d'inférence.
3. **Contrôle des ressources en C++ :** C++ permet un contrôle plus fin des ressources matérielles, comme la gestion de la mémoire et des threads. Cela peut contribuer à une meilleure gestion des opérations parallèles et à une réduction des délais d'exécution.

4.4-Importance de la Rapidité d'Exécution en C++ :

La différence de temps d'inférence entre Python et C++ n'est pas négligeable et peut avoir des implications importantes pour des applications dans des environnements exigeants en termes de performance. Par exemple, dans des systèmes embarqués ou des applications en temps réel, la rapidité d'exécution peut être cruciale, et le temps d'inférence réduit de **C++** pourrait permettre de traiter plus de données en moins de temps, offrant ainsi une meilleure réactivité et une utilisation plus efficace des ressources matérielles. Cela souligne l'importance de choisir le langage le plus adapté en fonction des contraintes de performance spécifiques aux cas d'usage.

5-Conclusion :

Les résultats montrent que, bien que Python et C++ offrent la même précision dans les prédictions du modèle, le **C++ se distingue par une rapidité d'inférence deux fois plus élevée**. Ce gain de performance est crucial dans des contextes où la rapidité d'exécution est essentielle, comme dans les systèmes embarqués ou les applications en temps réel.