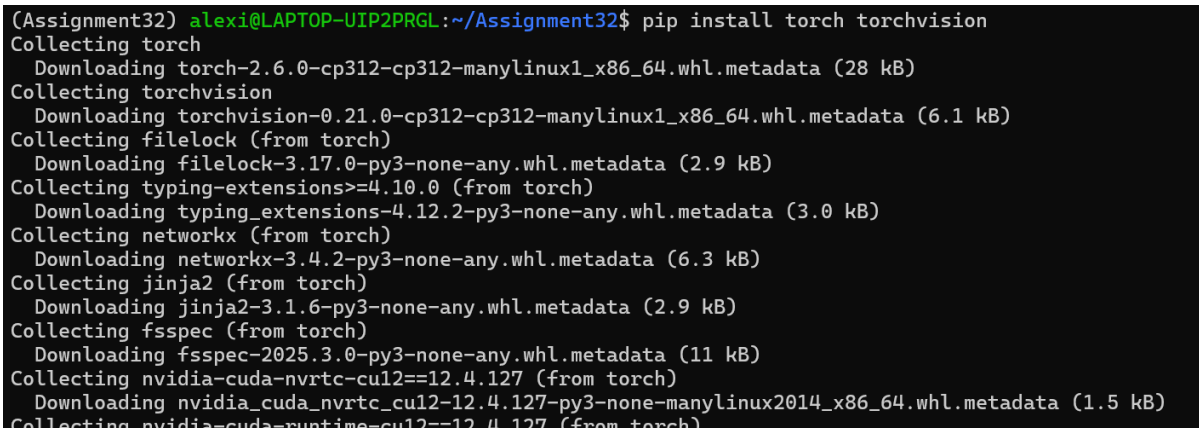
Astral Algorithmists: Assignment 3.2

Table of Contents

I- Environment Setup & Mini Project

For this assignment we have to create an environment for deep learning, in order to discover the use of this new AI solution. Therefore, we first had to choose the framework to use for this solution. Hence, we chose to use PyTorch in order to discover it. Indeed, previously we used Tensorflow. On the other hand, the use of PyTorch's dynamic graphs could be a valuable asset for our AI solution. In fact, we will be freer when it comes to creating and customizing the architecture of our future deep learning solution. So, we had to install PyTorch before we could start creating our working environment. Here is a snippet of the installation of PyTorch:



PyTorch installation

As for the creation of the virtual environment, this was done quite easily as you can see the commands below:

Creation of the environment

Activation of the environment

In addition, as most of the members of the group have already used Github before, the creation of a collaborative directory has been simplified. This directory will allow in the future the management of the different versions of our AI. Here is the link of this latter: https://github.com/Vorpalin/AI_Assignment

However, the GPU Check recognition caused us the most problem. Indeed, as first we did not understand why CUDA was not available and we thought that was due to a bad installation of PyTorch on our part. After several hours of hard research, we finally understood that this was not a problem itself and that it was normal since we did not have a GPU adapted to CUDA.

Finally, thanks to the code that was provided for this assignment, we were able to test the creation and the results of our model, which works as expected. Here is a snippet of the code performing these tests:

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np


# Check GPU availability
print("CUDA Available:", torch.cuda.is_available())
device = torch.device("cuda" if torch.cuda.is_available() else

# Define a simple model
class SimpleModel(nn.Module):
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.fc1 = nn.Linear(20, 32)
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.softmax(x, dim=1)

model = SimpleModel().to(device)
print(model)  # Print model summary-like info

# Generate dummy data
```

The python script

```
CUDA Available: False
SimpleModel(
  (fc1): Linear(in_features=20, out_features=32, bias=True)
  (fc2): Linear(in_features=32, out_features=2, bias=True)
)
Predictions: tensor([[0.5388, 0.4612],
        [0.5264, 0.4736],
        [0.5439, 0.4561],
        [0.5543, 0.4457],
        [0.5102, 0.4898],
        [0.5089, 0.4911]
```

Its result

To summarize, this homework familiarizes us with the structure and creation of a deep learning model as well as the PyTorch Framework and project management. These various skills will be useful to us both for our current term project but also for all our future projects.

II- Practical Training Pipeline & Mini Project

The goal of this part is to set up a training pipeline for CNN, with data loading and preprocessing, data augmentation and a training loop.

2.1) Practical Training Pipeline

2.1.1) Data Pipeline

The first action to take when you want to create an AI solution is to load data. This time, instead of using the panda library as usual, we used functions specific to PyTorch, allowing us on the one hand to use transformers to increase the size of our data by adding new images derived from those already present and to perform preprocessing, but also to load datasets already present in the library. Indeed, this week we preferred to use a dataset (the MNIST for digit recognition) whose quality and balance we are sure of, which was not necessarily the case with our previous image datasets. Here is a snippet of code performing the preprocessing, the data loading and augmentation:

```
transform_augmentation = transforms.Compose([
    #transforms.Grayscale(num_output_channels=3),  # C
    transforms.RandomRotation(20),
    transforms.Resize((28, 28)),
    transforms.RandomResizedCrop(28),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

transform_classic = transforms.Compose([
   # transforms.Grayscale(num_output_channels=3),  # C
    transforms.Resize((28, 28)),  # Ensure size matche
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

transform_test = transforms.Compose([
   # transforms.Grayscale(num_output_channels=3),  # C
    transforms.RandomRotation(15),
    transforms.Resize((28, 28)),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
```

Transformers

```
train_dataset = torchvision.datasets.MNIST(root='./data',
train_dataset_2 = torchvision.datasets.MNIST(root='./data'
test_data = torchvision.datasets.MNIST(root='./data', trai

train_data = torch.utils.data.ConcatDataset([train_dataset
train_loader = torch.utils.data.DataLoader(train_data, bat
test_loader = torch.utils.data.DataLoader(test_data, batch
```

Data loading and augmentation

2.1.2) Training loop and validation

Next, to create our AI solution, we need to define its architecture, which we did by creating the CNN class, inheriting from Module. Our class therefore consists of three alternations between convolution and maxpooling layers with two dropouts to prevent overfitting ending up on two linear layers. This description is found in the forwarding function of the class. Here is a snippet of the class CNN:

```
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, paddin
        self.pool = nn.MaxPool2d(2,2)
        self.dropout1 = nn.Dropout(0.25)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, paddi
        self.pool2 = nn.MaxPool2d(2, 2)
        self.dropout2 = nn.Dropout(0.25)

        self.fc1 = nn.Linear(10368, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.dropout1(x)

        x = self.pool2(F.relu(self.conv3(x)))
        x = self.dropout2(x)

        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

CNN class

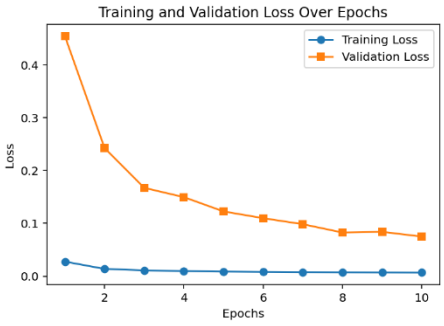
As for AI training, we implemented it outside of the class. To create this training loop, we must first choose the number of epochs. Then for each batch of images (64 images) we apply the forwarding function, we compute the loss, and we update the Artificial Neuron Network with the gradient descent as well as the parameter with our optimizer. We also added an early stopping so that our training stops earlier if our program detects the start of overfitting, so our model does not improve for 5 epochs in a row. Here is a snippet of the code performing this action:

```
all_epoch = []
epoch_l = []
test_epoch = []
best_val_loss = float('inf')
patience = 5
epochs_without_improvement = 0
# training with early stopping
for epoch in range(1,nb_epoch+1):

    running_loss = 0.0

    for i, data in enumerate(train_loader, 0):
        # get the inputs and the label to the gpu
        inputs, labels = data[0].to(device), data[1].to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward
        outputs = cnn(inputs)

        # compute loss
        loss = criterion(outputs, labels)

        # backward
        loss.backward()

        # optimize
        optimizer.step()
        # print statistics
        running_loss += loss.item()

    print(f'[Epoch {epoch}] loss: {running_loss / len(train_data
```

The training loop

```
How many epoch do you want to execute: 10
[Epoch 1] loss: 0.027
```

Its result



Loss over training

Finally, all that remains is to validate our model by testing it on our test data and see the results of this process. Here is snippet of the code performing this action:

```
correct_pred = {i: 0 for i in range(10)}
total_pred = {i: 0 for i in range(10)}
correct = 0
total = 0
running_loss = 0.0
# test the data
with torch.no_grad():
    for data in test_loader:
        images, labels = data[0].to(device), data[1].to(device)
        outputs = cnn(images)
        _, predictions = torch.max(outputs,1)
        loss = criterion(outputs, labels)
        running_loss += loss.item()
        # if the prediction is good we add one to the counter
        if torch.tensor([labels]) == predictions:
            correct_pred[labels.item()] += 1
            correct += 1
        total_pred[labels.item()] += 1
        total += 1


# print accuracy for each class
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname
    print(f'Accuracy for class: {classname} is {accuracy:.1f} %'

print(f'Accuracy: {correct*100/total:.1f}%')
print(f'Loss: {running_loss/len(test_loader)}')
```

The testing loop

```
Accuracy for class: 0 is 99.4 %
Accuracy for class: 1 is 99.2 %
Accuracy for class: 2 is 97.6 %
Accuracy for class: 3 is 98.9 %
Accuracy for class: 4 is 98.7 %
Accuracy for class: 5 is 97.3 %
Accuracy for class: 6 is 96.7 %
Accuracy for class: 7 is 97.0 %
Accuracy for class: 8 is 96.3 %
Accuracy for class: 9 is 94.8 %
Accuracy: 97.6%
Loss: 0.07481043768905996
```

Its result

As we can see with the metrics obtained by the graph on the previous page and the accuracy by number just above, our model works very well without being victim of overfitting.

Finally, as an improvement we used the wandb library to preserve the performance of our model, with a backup of our best AI in terms of accuracy. Here is snippet of the code performing this action:

```
wandb.login(key=k)
wandb.init(project="mnist-cnn", name="experiment_1")
```

Initilisation of the wandb library

```
try:
    wandb.log({
"epoch": epoch ,
"train_loss": c/len(train_loader),
"train_accuracy": correct/total,
"test_loss": all_epoch[-1],
"test_accuracy": test_epoch[-1]
    })
except:
    pass
scheduler.step()
# early stopping
if running_loss /len(test_loader) < best_val_loss:
    best_val_loss = running_loss /len(test_loader)
    torch.save(cnn.state_dict(), "model.pth")  # Save model
    print("Best model saved")
else:
    epochs_without_improvement += 1
    if epochs_without_improvement == patience:
        break
```

Data backup

2.2) Individual Reflection

2.2.1) AMIOT Tom

During this assignment, one of the challenges we faced was the newly use of the PyTorch framework, we previously worked on TensorFlow and us adapting to PyTorch's dynamic required some time. Another hard time we had was, like said before, during the training sessions, we were faced with many compatibility problems between layers, data stacking together, etc.

Also, the tuning of CNN's hyperparameters was another small difficulty we encounter. To prevent overfitting, we needed to find the right balance between the layers and the dropout. Throught many experiments, we learned how small adjustments could impact the model's accuracy and risk of overfitting.

With these difficulties, we gain a lot more experience in building and optimizing AI models, future creation in AI will be more effective.

2.2.2) AMRADE Madin

One of the main challenges we encountered in this part was related to data augmentation and normalization. For data augmentation, we had to find the right balance in applying transformations. Applying too many or too aggressive transformations risked altering the images too much, making the training process harder and slower. Our goal was to generalize the data and make the model more robust, while minimizing the loss of important visual information. Similarly, for normalization, we faced the challenge of scaling the data appropriately for the CNN changing its quality. Although normalization helps the network converge faster, it can sometimes cause some information to be lost. So again, we had to carefully tune our preprocessing to make our training process faster while losing a minimum amount of information.

2.2.3) MIALON Alexis

The main challenge that Astral Algorithmists encountered during this assignment was the model training part and the model architecture.

Indeed, during the various training sessions, the group often realized that there were compatibility problems between layers or that data had a tendency to group together in blocks, so all the images corresponding to 0 were together and the other numbers the same. So, we had to revisit our architecture several times and we made sure that our dataset shuffled each training epoch. We also had trouble finding a good architecture, but we overcame this by using architecture already used previously on other projects. Finally, the last complication that the group encountered was the training time, which was long, wasting a lot of time in the event of an

implementation error. To overcome this, we redirected our calculations to our CPU and made sure that our training processes the images in mini batches, further increasing the efficiency of the training.

Solving these problems has allowed the group to gain experience in the field of AI creation, which will facilitate the creation of future models.

2.2.4) MOSSAND Thomas

During this assignment, the **change of environment** from **TensorFlow to PyTorch** and the use of the **wandb library** were two of the challenging moments in our progress.

Unlike TensorFlow, where many operations are automated, PyTorch requires a **more detailed approach.** This represents **a difficulty** at first sight but ultimately enables **greater precision and control** in the training process. Also, the implementation of wandb, which enables deep learning experiments to be monitored and training metrics to be recorded and visualized, was **not a difficulty** in itself. On the other hand, understanding **the structure of the logs**, in particular how to correctly **record and load saved models**, was more **complicated**.

2.2.5) MULOT Antoine

A key challenge in this assignment was transitioning from TensorFlow to PyTorch for model development. While PyTorch's dynamic graphs offered more flexibility, it required adjustments in model implementation and debugging.

Additionally, setting up the training pipeline introduced compatibility issues between layers, and the dataset needed proper shuffling to avoid grouping patterns during training. Long training times also complicated the process, leading to shifting towards mini-batch processing for improved efficiency.

These challenges provided valuable experience in deep learning model optimization and framework adaptation.