Lab Session 5

«Neural network, with and wihtout hiden layer, for regression and classification»

---

1. Introduction

In this lab session, you will build a neural network with or without hidden layers, to perform classification and regression

2. Learning outcome

On successful completion of this lab session you will be able to :

- Build a neural network
- Use hidden layer in neural network
- Do regression with neural network
- Do classification with neural network

3. Ressources

Libraries Documentation

- Python : https://docs.python.org/3/
- NumPy : https://numpy.org/doc/
- SciPy : https://docs.scipy.org/doc/scipy/
- Matplotlib : https://matplotlib.org/3.5.1/
- Panda : https://pandas.pydata.org/docs/
- mglearn : https://libraries.io/pypi/mglearn
- sklearn : https://scikit-learn.org/stable/
- PyTorch : https://pytorch.org/docs/stable/index.html

**to proceed with the lab session, refer to this section*

4. Setup

to install pyTorch in UTBM Desktops :

1. Create and activate your conda environement (refers to moodle)
2. Install pytorch packages in conda environment using the following instruction

```
# !pip3 install torch torchvision torchaudio
```

5. Neural Network (with no hidden layers) applied to Regression

To setup our nn you need to perform the following imports

```
import torch
import torch.nn as nn
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
```

In this example you will use a simple dataset which consists of two arrays (X and y), then separate the datatset as usual

```
X, y = datasets.make_regression(n_samples=150, n_features=1, noise=15, random_state=4)
plt.scatter(X,y)
plt.xlabel("X")
```

```
plt.ylabel("y")
plt.title("Dataset")
```

Remember that NNs takes tensors as input, so always perform a conversion and reshape numpy arrays.

```
X = torch.from_numpy(X.astype(np.float32)).view(-1,1)
y = torch.from_numpy(y.astype(np.float32)).view(-1,1)
```

the size determins the numbers of values

```
input_size = 1
output_size = 1
```

Since we want to train a single layer NN, we execute the following instruction

```
model = nn.Linear(input_size , output_size)
```

To train the model properly, you will need to define the loss fucntion, and an optimizer to reduce the loss.

```
learning_rate = 0.001
l = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr =learning_rate )
```

Now you train the NN.

1. **Forward feed** : in this step, you are calculating the prediction (y_pred) using initial weights and features (X).
2. **Loss phase** : we measure the loss for each epochs using mean squared error.
3. **Backpropagation** : gardients are calculated
4. **Step** : weights are updated for every epoch
5. **Gradient clear** : gradients are cleared to make new ones

```
num_epochs = 2000

for epoch in range(num_epochs):
    #forward feed
    y_pred = model(X.requires_grad_())

    #calculate the loss
    loss= l(y_pred, y)

    #backward propagation: calculate gradients
    loss.backward()

    #update the weights
    optimizer.step()

    #clear out the gradients from the last step loss.backward()
    optimizer.zero_grad()

    print('Epoch {}, loss {}'.format(epoch, loss.item()))
```

predicted values are calculated

```
predicted = model(X).detach().numpy()
```

```
plt.scatter(X.detach().numpy() , y.detach().numpy(), label="Dataset")
plt.plot(X.detach().numpy() , predicted , "red", label="Regression line")
plt.xlabel("X")
plt.ylabel("y")
plt.title('regression using {} epochs'.format(num_epochs))
plt.legend()
plt.show()
```

Try plotting for 100, 500, 1500 and 2000 epochs

Exercise 1

Plot dynamic curve that show the evolution of the regression per epochs using the previous model

6. Neural Network (with hidden layers) applied to Classification

We are gonna create a neural network with hidden layers for classification.
We will work on the MNIST database (handwritten digits).
Let's start by importing the right librairies !

```
import torch
import torchvision # torch package for vision related things
import torch.nn.functional as F  # Parameterless functions, like (some) activation functions
import torchvision.datasets as datasets  # Standard datasets
import torchvision.transforms as transforms  # Transformations we can perform on our dataset for augmentation
from torch import optim  # For optimizers like SGD, Adam, etc.
from torch import nn  # All neural network modules
from torch.utils.data import DataLoader  # Gives easier dataset managment by creating mini batches etc.
from tqdm import tqdm  # For nicer progress bar
```

Exercise
We need to define the following values for the model.

```
input_size = ???     ## Size of the initial input
num_classes = ???    ## How many class do we have ?
learning_rate = ???  ## Size of the step between each iteration
batch_size = ???     ## Number of samples processed before the model is updated
num_epochs = ???     ## number of complete passes through the training dataset.
```

Now, we need to create the model for the neural network.

Exercise
What should be the value of the missing variables ?

```
class NN(nn.Module):
    def __init__(self, input_size, num_classes):
        super(NN, self).__init__()
        self.fc1 = nn.Linear(???, ???) ## What could we use for the first value ? for the second ?
        self.fc2 = nn.Linear(???, ???) ## What values could we use here ?

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

We can use this line of code to use the GPU if it's available and otherwise run on the CPU :

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Then, we will import the database and set up the training and test sets :

```
# Load Training and Test data
train_dataset = datasets.MNIST(root="dataset/", train=True, transform=transforms.ToTensor(), download=True)
test_dataset = datasets.MNIST(root="dataset/", train=False, transform=transforms.ToTensor(), download=True)
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=True)
```

We can now initialize the network and choose our optomizer and loss function :

```
# Initialize network
model = NN(input_size=input_size, num_classes=num_classes).to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

Now it's time to train the network :

```
# Train Network
for epoch in range(num_epochs):
    for batch_idx, (data, targets) in enumerate(tqdm(train_loader)):
        # Get data to cuda if possible
        data = data.to(device=device)
        targets = targets.to(device=device)

        # Get to correct shape
        data = data.reshape(data.shape[0], -1)

        # forward
        scores = model(data)
        loss = criterion(scores, targets)

        # backward
        optimizer.zero_grad()
        loss.backward()

        # gradient descent or adam step
        optimizer.step()
```

We can check the accuracy the following way :

```
# Check accuracy on training & test to see how good our model
def check_accuracy(loader, model):
    num_correct = 0
    num_samples = 0
    model.eval()

    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device)
            y = y.to(device=device)
            x = x.reshape(x.shape[0], -1)

            scores = model(x)
            _, predictions = scores.max(1)
            num_correct += (predictions == y).sum()
            num_samples += predictions.size(0)

    model.train()
    return num_correct/num_samples


print(f"Accuracy on training set: {check_accuracy(train_loader, model)*100:.2f}")
print(f"Accuracy on test set: {check_accuracy(test_loader, model)*100:.2f}")
```

Exercise

What should be the value of the missing variables ?

Choose the right hyper-parameters to get an accuracy on the training and test set above 90 %

Exercise

Add one more hidden layer.