Neural Networks

Quan Minh Phan & Ngoc Hoang Luong

May 18, 2021

Table of Contents

Multi Layer Perceptron

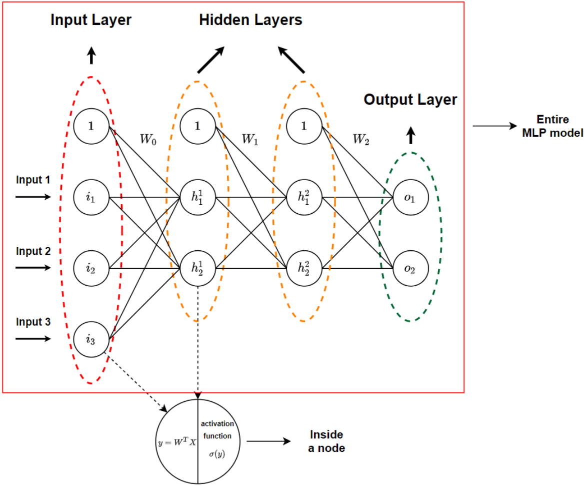
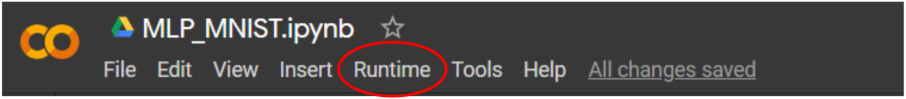


Figure: Example of a MLP.

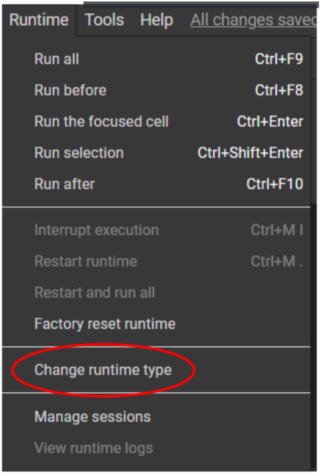
Setup for using GPU

- Click the 'Runtime' button



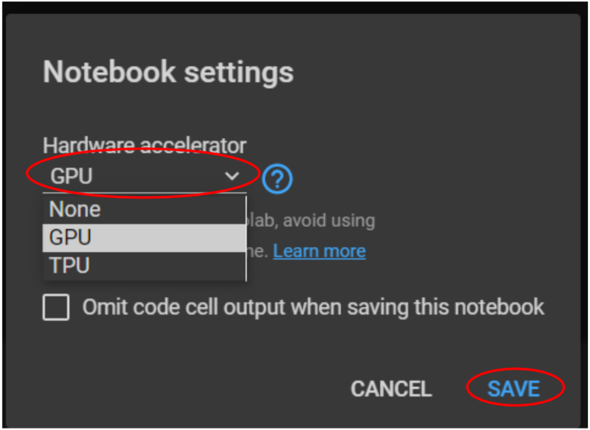
Setup for using GPU

- Click the 'Change runtime type' button



Setup for using GPU

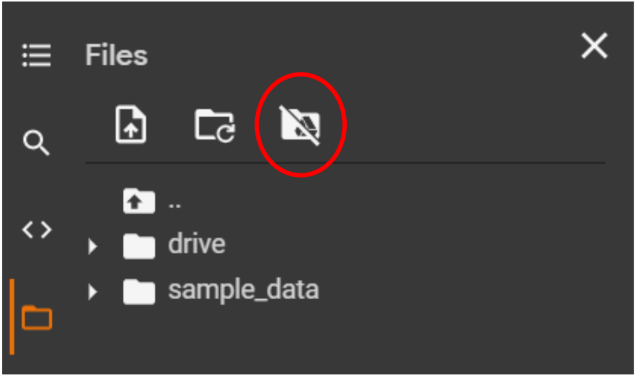
Step 1:
Choose 'GPU'



Step 2:
Click 'SAVE'

Connect to Google Drive

Connect to your Google Drive



Import necessary libraries

```
import random
import numpy as np
import matplotlib.pyplot as plt

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader
import torchvision.datasets as datasets
import torchvision.transforms as transforms
```

Build model

Training strategy

Split data

Load data

Data preprocessing

Check whether we are using 'GPU' or 'CPU'

```
>> device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
   print(device)
>> 'cuda'
```

Setup for getting the reproducible results

```
# Setup for getting the reproducibility of results
>> random.seed(1)
   np.random.seed(1)
   torch.manual_seed(1)
   torch.backends.cudnn.deterministic = True
   torch.backends.cudnn.benchmark = False
```

MNIST dataset

Some informations:

- Grayscale images.
- Each image is stored as a matrix has 28 x 28 size.
- 10 classes $(0, 1, \dots)$.
- $60,000$ samples in training set.
- $10,000$ samples in testing set.

Hyperparameters

```
>> input_size = 784
   n_classes = 10
   learning_rate = 0.001
   batch_size = 64
   n_epochs = 5
```

input_size = 784	→ 1 * 28 * 28
n_classes = 10	→ 10 digits 0, 1, 2, ..., 9
learning_rate = 0.001	→ learning rate on gradient descent
batch_size = 64	→ the number of samples in each batch
n_epochs = 5	→ the number of training epochs

Load data from Google Drive

```
>> from torch.utils.data import DataLoader
   import torchvision.datasets as datasets
   import torchvision.transforms as transforms
```

necessary libraries

```
# Load 'MNIST' dataset
>> train_dataset = datasets.MNIST(root='/content/drive/MyDrive/datasets/mnist', train=True,
                                  transform=transforms.ToTensor(), download=True)
   train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)

   test_dataset = datasets.MNIST(root='/content/drive/MyDrive/datasets/mnist', train=False,
                                 transform=transforms.ToTensor(), download=True)
   test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)
```

The dataset which we want to load

datasets
- root: the path which we want to contain the dataset
- train: set 'True' if we want to load training set, 'False' if we want to load testing set.
- trainsform: the set of operators to transform the original data.
- download: download data if we don't have dataset in the root path.

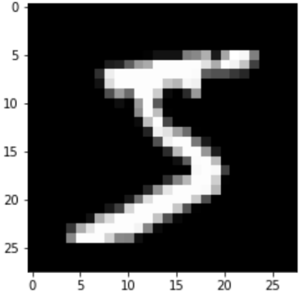
DataLoader
- dataset: the dataset which we want to load
- batch_size: split data to n batches, each batch has 'batch_size' samples.
- shuffle: if set 'True', data is shuffled after we iterate over all batches

Show an image example in training set

```
>> import matplotlib.pyplot as plt
```

```
>> image, label = train_dataset[0]
   plt.imshow(image.squeeze(), cmap='gray')
   print(label)
```

```
>> 5
```



Build a Multi Layer Perceptron model by using Pytorch

Requirements:

- Problem dataset: MNIST
- Build a model has 4 layers: 1 'input' layer, 2 'hidden' layers, 1 'output' layer.
- 'input' layer has 782 nodes (input_size), 'hidden 1' layer has 100 nodes, 'hidden 2' layer has 100 nodes and 'output' layer has 25 nodes.
- The activation of each node in 'input' layer and 'hidden' layers is 'ReLU'.
- The activation of each node in 'output' layer is 'Softmax'.
- 'out_features' value of 'output' layer is 10 (n_classes).

Build a Multi Layer Perceptron model by using Pytorch

Approach 1

```
>> class MLP(nn.Module):
       def __init__(self, input_size, n_classes):
           super().__init__()
           self.input_layer = nn.Linear(input_size, 100)
           self.hidden_layer_1 = nn.Linear(100, 100)
           self.hidden_layer_2 = nn.Linear(100, 25)
           self.output_layer = nn.Linear(25, n_classes)

       def forward(self, X):
           X = self.input_layer(X)
           X = F.relu(X)
           X = self.hidden_layer_1(X)
           X = F.relu(X)
           X = self.hidden_layer_2(X)
           X = F.relu(X)
           X = self.output_layer(X)
           prob = F.softmax(X, dim=1)
           return prob
```

Approach 2

```
>> class MLP(nn.Module):
       def __init__(self, input_size, n_classes):
           super().__init__()
           self.model = nn.Sequential(
               nn.Linear(input_size, 100),
               nn.ReLU(),
               nn.Linear(100, 100),
               nn.ReLU(),
               nn.Linear(100, 25),
               nn.ReLU(),
               nn.Linear(25, n_classes),
               nn.Softmax(dim=1)
           )

       def forward(self, X):
           prob = self.model(X)
           return prob
```

Build a Multi Layer Perceptron model by using Pytorch

```
>> model = MLP(input_size=input_size,
              n_classes=n_classes).to(device)
   print(model)
```

```
>> MLP(
     (model): Sequential(
       (0): Linear(in_features=784, out_features=100,
                  bias=True)
       (1): ReLU()
       (2): Linear(in_features=100, out_features=100,
                  bias=True)
       (3): ReLU()
       (4): Linear(in_features=100, out_features=25,
                  bias=True)
       (5): ReLU()
       (6): Linear(in_features=25, out_features=10,
                  bias=True)
       (7): Softmax(dim=1)
      )
)
```

```
>> model = MLP(input_size=input_size,
              n_classes=n_classes).to(device)
   print(model)
```

```
>> MLP(
     (input_layer): Linear(in_features=784,
                           out_features=100,
                           bias=True)
     (hidden_layer_1): Linear(in_features=100,
                              out_feature=100,
                              bias=True)
     (hidden_layer_2): Linear(in_features=100,
                              out_features=25,
                              bias=True)
     (output_layer): Linear(in_features=25,
                            out_features=10,
                            bias=True)
                    bias=True)
)
```

Define the loss and the optimization algorithm

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

Loss function: Cross entropy

Optimizer: Adam

Training model

```
for epoch in range(n_epochs):
    for batch_idx, (data, targets) in enumerate(train_loader):
        # Get data to GPU
        data = data.to(device) # Put our images to the GPU if GPU is available
        targets = targets.to(device) # Put our labels to the GPU as well

        # Change to the correct tensor shape
        # Our data is in the form (batch_size, color_channel, w, h) (64, 1, 28, 28)
        # We need to change it to (batch_size, color_channel * w * h) (64, 784)
        data = data.reshape(data.shape[0], -1)

        # forward pass
        scores = model(data)
        loss = criterion(scores, targets) # compute the loss/cost function J for this batch

        # backward pass
        optimizer.zero_grad() # empty the optimizer first
        loss.backward() # compute the gradient dJ/dw's

        # gradient descent
        optimizer.step()

        if (batch_idx+1) % 100 == 0:
            print(f'Epoch {epoch+1}/{n_epochs}, Batch {batch_idx+1}, Loss: {loss.item():.2f}')
```

Performance Evaluation

```python
def get_accuracy(loader, model):
    if loader.dataset.train:
        print('Getting accuracy on training data.')
    else:
        print('Getting accuracy on testing data.')

    n_corrects = 0
    n_samples = 0
    model.eval() # put our model to evaluation mode

    with torch.no_grad(): # no need to compute gradient here
        for x, y in loader:
            x = x.to(device)
            y = y.to(device)
            x = x.reshape(x.shape[0], -1)

            # forward
            scores = model(x)  # scores 64 x 10
            _, y_pred = scores.max(1)
            n_corrects += (y_pred == y).sum()
            n_samples += y_pred.size(0)

        print(f'We got {n_corrects}/{n_samples} correct. Accuracy = {float(n_corrects)/float(n_samples)*100.0:.2f}')
    model.train() # put our model to train mode again
```

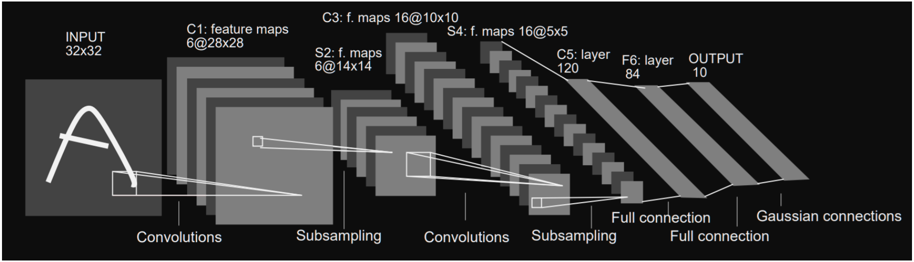
Performance Evaluation

```
>> get_accuracy(train_loader, model)
   get_accuracy(test_loader, model)
```

```
>> Getting accuracy on training data.
   We got 57691/60000 correct. Accuracy = 96.15
   Getting accuracy on testing data.
   We got 9560/10000 correct. Accuracy = 95.60
```

Table of Contents

LeNet-5



Layer	Layer Type	Input Channels	Output Channels	Kernel Size	Stride	Activation function
Input	Image 32 x 32	-	-	-	-	-
C1	Convolution	1	6	5 x 5	1	Tanh
S2	Sub Sampling	6	6	2 x 2	-	-
C3	Convolution	6	16	5 x 5	1	Tanh
S4	Sub Sampling	16	16	2 x 2	-	-
C5	Convolution	16	120	-	-	Tanh
F6	Fully Connected	120	84	-	-	Tanh
Output	Fully Connected	84	10	-	-	Softmax

Table of Contents

Import necessary libraries

```
import random
import numpy as np
import matplotlib.pyplot as plt

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader
import torchvision.datasets as datasets
import torchvision.transforms as transforms
```

Check whether we are using 'GPU' or 'CPU'

```
>> device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
   print(device)
>> 'cuda'
```

Setup for getting the reproducible results

```
# Setup for getting the reproducibility of results
>> random.seed(1)
   np.random.seed(1)
   torch.manual_seed(1)
   torch.backends.cudnn.deterministic = True
   torch.backends.cudnn.benchmark = False
```

Hyperparameters

```
n_classes = 10 # 10 digits 0, 1, 2, ... 9
learning_rate = 0.001
batch_size = 64
n_epochs = 5
```

Some informations

- Each image in MNIST is stored as a matrix has 28 x 28 size but the required input of LeNet-5 is an image 32 x 32

  $\rightarrow$ transform the image

Transformations

```
transforms = transforms.Compose([transforms.Resize((32, 32)),
                                 transforms.ToTensor()])
```

We have 2 transformation. The first is resize the image. The second is convert the data type to Tensor.

Load data from Google Drive

```
# Load data to our Google Drive
train_dataset = datasets.MNIST(root='/content/drive/MyDrive/datasets/mnist', train=True,
                               transform=transforms, download=True)
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_dataset = datasets.MNIST(root='/content/drive/MyDrive/datasets/mnist', train=False,
                              transform=transforms, download=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)
```

We put above transformations here

Build LeNet-5 by using PyTorch

```python
class LeNet5(nn.Module):
    def __init__(self, n_classes):
        super().__init__()
        self.model = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, stride=1),
            nn.Tanh(),
            nn.AvgPool2d(kernel_size=2),
            nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, stride=1),
            nn.Tanh(),
            nn.AvgPool2d(kernel_size=2),
            nn.Conv2d(in_channels=16, out_channels=120, kernel_size=5, stride=1),
            nn.Tanh(),
            nn.Flatten(),
            nn.Linear(in_features=120, out_features=84),
            nn.Tanh(),
            nn.Linear(in_features=84, out_features=n_classes),
            nn.Softmax(dim=1)
        )

    def forward(self, X):
        prob = self.model(X)
        return prob
```

Build LeNet-5 by using PyTorch

```
model = LeNet5(n_classes=n_classes).to(device)
print(model)
```

```
LeNet5(
  (model): Sequential(
    (0): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
    (1): Tanh()
    (2): AvgPool2d(kernel_size=2, stride=2, padding=0)
    (3): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
    (4): Tanh()
    (5): AvgPool2d(kernel_size=2, stride=2, padding=0)
    (6): Conv2d(16, 120, kernel_size=(5, 5), stride=(1, 1))
    (7): Tanh()
    (8): Flatten(start_dim=1, end_dim=-1)
    (9): Linear(in_features=120, out_features=84, bias=True)
    (10): Tanh()
    (11): Linear(in_features=84, out_features=10, bias=True)
    (12): Softmax(dim=1)
  )
)
```

Define the loss and the optimization algorithm

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

→ **Loss function: Cross entropy**

→ **Optimizer: Adam**

Training model

```python
# Train our Logistic Regression model
for epoch in range(n_epochs):
    for batch_idx, (data, targets) in enumerate(train_loader):
        # Get data to cuda
        data = data.to(device) # Put our images to the GPU if GPU is available
        targets = targets.to(device) # Put our labels to the GPU as well

        # We don't need to reshape the image to vector format
        # because the input of LeNet-5 is an image 32 x 32.
        # data = data.reshape(data.shape[0], -1)

        # forward pass
        scores = model(data)
        loss = criterion(scores, targets) # compute the loss/cost function J for this batch

        # backward pass
        optimizer.zero_grad() # empty the optimizer first
        loss.backward() # compute the gradient dJ/dw's

        # gradient descent
        optimizer.step()

        if (batch_idx+1) % 100 == 0:
            print(f'Epoch {epoch+1}/{n_epochs}, Batch {batch_idx+1}, Loss: {loss.item():.2f}')
```

Performance Evaluation

```python
def get_accuracy(loader, model):
    if loader.dataset.train:
        print('Getting accuracy on training data.')
    else:
        print('Getting accuracy on testing data.')

    n_corrects = 0
    n_samples = 0
    model.eval() # put our model into evaluation mode

    with torch.no_grad(): # no need to compute gradient here
        for x, y in loader:
            x = x.to(device)
            y = y.to(device)
            # x = x.reshape(x.shape[0], -1)

            # forward
            scores = model(x)  # scores 64 x 10
            _, y_pred = scores.max(1)
            n_corrects += (y_pred == y).sum()
            n_samples += y_pred.size(0)

        print(f'We got {n_corrects}/{n_samples} correct. Accuracy = {float(n_corrects)/float(n_samples)*100.0:.2f}')
    model.train() # put our model to train mode again
```

Performance Evaluation

```
>> get_accuracy(train_loader, model)
   get_accuracy(test_loader, model)
```

```
>> Getting accuracy on training data.
   We got 59142/60000 correct. Accuracy = 98.57
   Getting accuracy on testing data.
   We got 9821/10000 correct. Accuracy = 98.21
```