Artificial Neural Networks in Pytorch and Tensorflow

contact: *cursifrancesco@gmail.com*

Introduction

Pytorch (https://pytorch.org/) and **TensorFlow** (https://www.tensorflow.org/) are two of the most popular frameworks to build **Artificial Neural Networks** (ANN) in Python.

Here we are going to show an example on how to build an ANN using both frameworks, discussing their main differences:

- ANN with Pytorch (in section -A);
- ANN with Tensorflow -B;
- Comparison -C;

A. *ANN with Pytorch*

```
#Import necessary modules

import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets as dsets
import cv2 as cv
from google.colab.patches import cv2_imshow
import numpy as np
import matplotlib.pyplot as plt
import random

#Check if GPU is available
if (torch.cuda.device_count()):
  print(torch.cuda.device_count())
  print(torch.cuda.get_device_name(0))

#Assign cuda GPU located at location '0' to a variable
  cuda0 = torch.device('cuda:0')
else:
  cuda0 = torch.device('cpu')

#———Neural Network Model
class NetClassifier(nn.Module):
# inputs are:
# - input dimension, nodes in hidden layers, output dimension
  def __init__(self,in_size,hidden_sizes,out_size):
    super(NetClassifier,self).__init__()
    self.in_size = in_size

#activation function
    self.act = torch.nn.ReLU()

#Linear layer
    self.input = nn.Linear(in_size,hidden_sizes[0])

#initialize set of layers
    self.hidden = torch.nn.ModuleList()

    for i in range(len(hidden_sizes)-1):
      hidden = nn.Linear(hidden_sizes[i],hidden_sizes[i+1])
      self.hidden.add(hidden)
```

```python
    self.output = nn.Linear(hidden_sizes[-1],out_size)

    self.softmax = nn.Softmax()

#This function defines the forward pass to compute
#network output
  def forward(self,x):

    x = self.input(x)
    x = self.act(x)

    for h in self.hidden:
      x = h(x)
      x = self.act(x)

    out = self.output(x)
    proba = self.softmax(out)
    _,predicted_class = torch.max(out, 1)

    return out,proba,predicted_class

#--- Function for training the network
def TrainNet(Net,Data_train,Data_val,ConvNet=False,lr=1e-03,Epochs = 1000):
    loss_function = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(Net.parameters(), lr=lr)

    N_batches = len(Data_train)

    for epoch in range(Epochs):
      loss_tot = 0
      for x,y in Data_train:
        x = x.to(cuda0)
        y = y.to(cuda0)
        if not ConvNet:
          y_pred = Net(x.view(-1,Net.in_size))[0]
        else:
          y_pred = Net(x)[0]

        loss = loss_function(y_pred,y)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        loss_tot = loss_tot + loss.item()

      loss_tot = loss_tot/N_batches
      print("*****epoch: ",epoch," loss: ",loss_tot)

      #Validation accuracy:
      correct = 0
      for x,y in Data_val:
        x = x.to(cuda0)
        y = y.to(cuda0)

        if not ConvNet:
          Net_eval = Net.eval()
```

```
            _,_,pred_class = Net_eval(x.view(-1,Net.in_size))
        else:
            _,_,pred_class = Net(x)
        correct = correct+(pred_class==y).sum().item()

      accuracy = correct/(y.shape[0])
      print("*****accuracy: ",accuracy)

#---- get Training Data
train_data = dsets.MNIST(root="",train=True,download=True, transform=transforms.ToTensor())
val_data = dsets.MNIST(root="",train=False,download=True, transform=transforms.ToTensor())

data = train_data[0]
img = np.transpose(data[0].numpy(), (1, 2, 0))
img = img[:,:,0] #for gray scale only "D array needed"
plt.imshow(img,cmap='gray')
print("Value",data[1] )

in_size = img.shape[0]*img.shape[1] #squeeze the image
out_size = 10 #10 classes form 0 to 9

train_loader = torch.utils.data.DataLoader(dataset=train_data,batch_size=100)
val_loader = torch.utils.data.DataLoader(dataset=val_data,batch_size=len(val_data))

#------------------
#--- Initialize network and train
h_sizes = [10] #nodes in each hidden layer
model = NetClassifier(in_size,h_sizes,out_size)
model.to(cuda0)

TrainNet(model,train_loader,val_loader,lr=1e-03,Epochs = 20)
```

B. ANN with Tensorflow

```
#imort modules
import tensorflow as tf
import tensorflow_probability as tfp
import tensorflow_datasets.public_api as tfds

import cv2 as cv
from google.colab.patches import cv2_imshow
import numpy as np
import matplotlib.pyplot as plt
import random

#--- define Network Model
class MLP(tf.keras.Sequential):
    def __init__(self, input_dim,h_sizes, output_dim,activation):
        super(MLP, self).__init__()

        self.add(tf.keras.layers.Flatten()) #flattens the image size
        input_layer = tf.keras.layers.Input(shape=input_dim)  # instantiate Keras input tensor
        self.add(input_layer)

        for k in range(len(h_sizes)):
            layer = tf.keras.layers.Dense(h_sizes[k], activation=activation)
            self.add(layer)
```

```
        self.add(tf.keras.layers.Dense(output_dim,activation=tf.nn.softmax)) #ouptu layer

        self.loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
        self.optimizer = tf.keras.optimizers.Adam(lr=0.001)


        self.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              optimizer=tf.keras.optimizers.Adam(lr=0.001),
              metrics=['accuracy'])

        #for one-hot-encoed labels
        # self.compile(loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
        #         optimizer=tf.keras.optimizers.Adam(),
        #         metrics=['accuracy'])

        self.input_dim = input_dim
        self.h_sizes = h_sizes
        self.output_dim = output_dim

#------ Training Functon
    def train(self,X_train,Y_train,X_test,Y_test,batch_size,TrainType=0, epochs = 1e03, Err = 1e-06, dErr = 1e-07):

        if TrainType == 0:
          print("Fitting")
          self.fit(X_train,
        Y_train,
        batch_size=batch_size,
        epochs=epochs,
        verbose=1,
        validation_data=(X_test, Y_test))

        elif TrainType == 1:
          epoch_lim = epochs

          batch_loss = 1
          iter = 0

          n_batches = X_train.shape[0]//batch_size

          X_train_batch = tf.split(X_train, num_or_size_splits=n_batches, axis=0)
          Y_train_batch = tf.split(Y_train, num_or_size_splits=n_batches, axis=0)

          for epoch in range(epochs):

            epoch_loss = 0
            correct = 0

            for i in range(len(X_train_batch)):
              x = X_train_batch[i]
              y = Y_train_batch[i]

              batch_loss = self.train_on_batch(x, y)
              epoch_loss = epoch_loss+batch_loss[0]
              correct = correct+batch_loss[1]

            epoch_loss = epoch_loss/len(X_train_batch)
```

```
            correct = correct/len(X_train_batch)
            print("Epoch", epoch, " Loss ", epoch_loss," accuracy", correct)

        elif TrainType == 2:
          n_batches = X_train.shape[0]//batch_size

          X_train_batch = tf.split(X_train, num_or_size_splits=n_batches, axis=0)
          Y_train_batch = tf.split(Y_train, num_or_size_splits=n_batches, axis=0)

          for epoch in range(epochs):
            correct = 0
            i = 0
            loss_epoch = 0
            for i in range(len(X_train_batch)):
              x = X_train_batch[i]
              y = Y_train_batch[i]

              with tf.GradientTape() as tape:

                y_pred = self.call(x)
                loss_value = self.loss(y,y_pred)

                grads = tape.gradient(loss_value, self.trainable_variables)
                self.optimizer.apply_gradients(zip(grads, self.trainable_variables))

                class_pred = np.argmax(y_pred.numpy(),axis=1)
                correct = correct+(y.numpy()==class_pred).sum()/y.shape[0]

                loss_epoch = loss_epoch+loss_value.numpy()

            epoch_accuracy = correct/n_batches
            epoch_loss = loss_epoch/n_batches

            #test set
            y_pred = self.predict(X_test)
            loss_value = self.loss(Y_test,y_pred).numpy()
            class_pred = np.argmax(y_pred,axis=1)
            correct = (Y_test==class_pred).sum()/Y_test.shape[0]

            print("***epoch: ",epoch," train loss epoch:",epoch_loss," train accuracy:",epoch_accuracy)
            print("                    test loss:",loss_value," test accuracy:",correct)


#Load Traning data
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

img = x_train[0]
img = img.reshape((img.shape[0],img.shape[1],1))
img = img[:,:,0] #for gray scale only "D array needed"
plt.imshow(img,cmap='gray')
print("Value",y_train[0] )

input_dim = img.shape[0]*img.shape[1]
output_dim = 10

#---- Train the Network
hidden_sizes = [30]
```

```
Net = MLP(input_dim,hidden_sizes, output_dim,"relu")
# Net = MLP_2((img.shape[0],img.shape[1]),hidden_sizes, output_dim,"relu")

Net.train(x_train,y_train,x_test,y_test,100,TrainType=2, epochs = 100, Err = 1e-06, dErr = 1e-07)
```

There is allows another possible way to build and ANN with TF, which allows concatenating different layers and specifying multiple inputs and outputs:

```
class MLP_2():
  def __init__(self, input_dim,h_sizes, output_dim,activation):
        super(MLP_2, self).__init__()

        input = tf.keras.layers.Input(shape=(input_dim[0],input_dim[1]))
        x = tf.keras.layers.Flatten()(input)

        for k in range(len(h_sizes)):
            x = tf.keras.layers.Dense(h_sizes[k], activation=activation)(x)

        output = tf.keras.layers.Dense(output_dim,activation=tf.nn.softmax)(x) #ouptu layer

        self.model = tf.keras.Model(inputs=input,outputs=output)

        self.loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
        self.optimizer = tf.keras.optimizers.Adam(lr=0.001)

        self.model.compile(loss=self.loss,
              optimizer=self.optimizer,
              metrics=['accuracy'])

  def train(self,X_train,Y_train,X_test,Y_test,batch_size,TrainType=0, epochs = 1e03, Err = 1e-06, dErr = 1e-07):

      if TrainType == 0:
        print("Fitting")
        self.model.fit(X_train,
      Y_train,
      batch_size=batch_size,
      epochs=epochs,
      verbose=1,
      validation_data=(X_test, Y_test))

      elif TrainType == 1:
        epoch_lim = epochs

        batch_loss = 1
        iter = 0

        n_batches = X_train.shape[0]//batch_size

        X_train_batch = tf.split(X_train, num_or_size_splits=n_batches, axis=0)
        Y_train_batch = tf.split(Y_train, num_or_size_splits=n_batches, axis=0)

        for epoch in range(epochs):

          epoch_loss = 0
          correct = 0

          for i in range(len(X_train_batch)):
            x = X_train_batch[i]
```

```
        y = Y_train_batch[i]

        batch_loss = self.model.train_on_batch(x, y)
        epoch_loss = epoch_loss+batch_loss[0]
        correct = correct+batch_loss[1]

      epoch_loss = epoch_loss/len(X_train_batch)
      correct = correct/len(X_train_batch)
      print("Epoch", epoch, " Loss ", epoch_loss," accuracy", correct)

  elif TrainType == 2:
    n_batches = X_train.shape[0]//batch_size

    X_train_batch = tf.split(X_train, num_or_size_splits=n_batches, axis=0)
    Y_train_batch = tf.split(Y_train, num_or_size_splits=n_batches, axis=0)

    for epoch in range(epochs):
      correct = 0
      i = 0
      loss_epoch = 0
      for i in range(len(X_train_batch)):
        x = X_train_batch[i]
        y = Y_train_batch[i]

        with tf.GradientTape() as tape:

          y_pred = self.model.call(x)
          loss_value = self.loss(y,y_pred)

          grads = tape.gradient(loss_value, self.model.trainable_variables)
          self.optimizer.apply_gradients(zip(grads, self.model.trainable_variables))

          class_pred = np.argmax(y_pred.numpy(),axis=1)
          correct = correct+(y.numpy()==class_pred).sum()/y.shape[0]

          loss_epoch = loss_epoch+loss_value.numpy()

      epoch_accuracy = correct/n_batches
      epoch_loss = loss_epoch/n_batches

      #test set
      y_pred = self.model.predict(X_test)
      loss_value = self.loss(Y_test,y_pred).numpy()
      class_pred = np.argmax(y_pred,axis=1)
      correct = (Y_test==class_pred).sum()/Y_test.shape[0]

      print("***epoch: ",epoch," train loss epoch:",epoch_loss," train accuracy:",epoch_accuracy)
      print("                    test loss:",loss_value," test accuracy:",correct)
```

C. Comparison

The main differences between the two frameworks are:

- **Utilizing the GPU**: Pytorch allows much easier way to exploit the GPU by just utilizing few lines of codes. remember that throughout the code, all tensors and networks need to be moved to the same device by means of $.to(device)$.
- **Inputs to ANN**: in Pytorch, all inputs must be converted to tensors by using $torch.tensor()$. TF, instead, allows inputting directly numpy arrays.
- **Outputs**: in both frameworks the model outputs can be converted to numpy by means of $.numpy()$.

- **Layer definition**: in Pytorch the activation function for each layer needs to be specified as an additional object and called every time after each layer. In TF the activation can be directly specified within the layer definition.
- **Multiple-inputs, Multiple-outputs**: in Pytorch you can define your own function for the forward step in $forward()$. Multiple inputs can be passed and multiple outputs obtained. In TF to build a model with multiple inputs and outputs a different structure needs to be used by employing $keras.Model(inputs = [in1, in2], outputs = [out1, out2]$.
- **Model Training**: at each iteration in Pytorch, you need to call your loss function, zero the network gradients with $optimizer.zero\_grad()$, and update the gradients with $optimizer.step()$. In TF there are different ways to train your model. In order to use $.fit()$ and $.train\_on\_batch()$ the model needs to be compiled first. If one wants to have more freedom on the training, a custom training function can be created. In this case, at each iteration $with\ tf.GradientTape()\ as\ tape$ needs to be called, the loss function computed, the gradients computed with $tape.gradient(loss, model_parameters)$, and finally updating the gradients with $optimizer.apply\_gradients()$.