Python and Pytorch Tutorial

Introduction

- Pytorch is the Python implementation of the Torch computing package.
- Suitable for machine learning applications:
  - Readily integrates Graphical Processing Units (GPUs)
  - Employs an automatic differentiation (AD) library needed for back propagation
- Pytorch is considered more low-level than its frequently-compared-with counterpart Tensorflow
  - This makes it sometimes faster and more flexible.
- Examples of flexibility-promoting options that you can apply: per-layer training rates,

Example: `optimizer =  optim.Adam([{'params': net.fc2.parameters(), 'lr': 1e-3}, {'params': net.fc3.parameters(), 'lr': 1e-1}], betas=(0.9, 0.999), lr=1e-2)`

Installation

- There are several ways to start using Pytorch.
- In Google Colab, it can be imported directly using `import torch`

pip3 install torch

Installation

- Alternatively, the following steps are suggested to use Pytorch locally on your PC:

Using Anaconda

- Install Anaconda
- Create a virtual environment from the Anaconda CMD

```
conda create –name Example
conda activate Example
```

- Install Pytorch

```
conda install pytorch
```

Using PIP

```
pip install torch
```

Python Editors

PC

kite

SPYDER

5-minute Introduction to Python

- Python syntax is readability-oriented, using English-like language with math.
- Python uses indentation (as compared to curly brackets in c/c++ for example) to define scopes of loops, functions, if-statements…etc.
- Python uses new lines to complete a command (as compared to semicolons for example)

5-minute Introduction to Python

- Assigning values is like other languages, and there are different data types. Example:
  - x = 3
  - A = ‘hello’
  - B = “hello”
  - x, A, B = 3, ‘hello’, ‘world’

```
Example: A='Hello Mam, he said'
         print(A)
         A="Hello Mam, he said"
         print(A)
         A='"Hello Mam", he said'
         print(A)
         A=""Hello Mam", he said"      Not correct
```

- These are global variables, and can be used everywhere, including inside functions.
- Speaking of functions,

  here is how to define one:

```
def Recursion(k):
 if(k > 0):
  result = k + Recursion(k - 1)
  print(result)
 else:
  result = 0
 return result

print("Recursion Example Results")
Recursion(3)
```

5-minute Introduction to Python

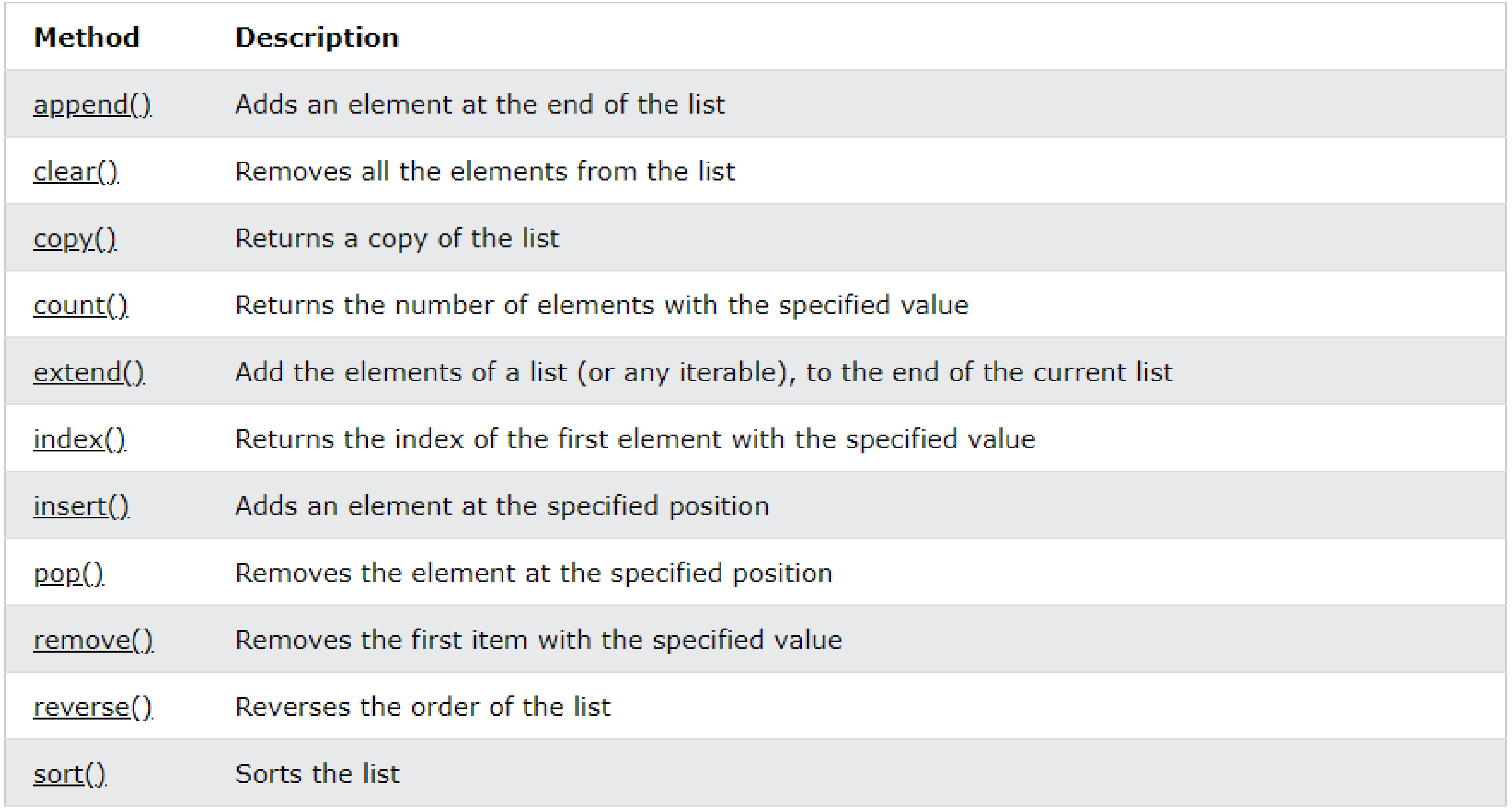
- As we are aiming to use Pytorch, we pay special attention to Tensors.
- Tensors are not native in Python, but they can be created out of arrays.
- Arrays are also not native in Python, but can be created using other libraries such as numpy.
- The closest there is to an array in Python are lists.

Arrays/lists in Python

- An array or a list can hold many variables under a single name, on which you can loop, and perform many operations.
- Array/list methods:

Example:

```
Ar1=[2, 3, 5, 2, 11, 2, 7]
Ar2=[1, 4, 2]
x= [10, 7, 2, 5, 2, 11]
Ar2.append(x)
print(Ar2)
Ar1.extend(x)
print(Ar1)
count = Ar1.count(2)
print('Count of 2:', count)
index = Ar1.index(2)
print(index)
```



Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Tensors

- Tensors are data structures that are the generalization of matrices
  - A matrix is a 2D tensor
- Since much of neural network computation can be modeled as matrix operations, such data structures are suitable.
- Pytorch enables running tensor operations on the GPU

Tensors with Pytorch

- Tensors can be created out of:
  - List arrays: a= [[1., -1.], [1., -1.]] → a_torch= torch.tensor(a)
  - NumPy arrays: a_np=numpy.array(a) → a_torch= torch.from_numpy(a_np)
  - Random arrays: a_rand =torch.rand((2,2))
  - Ones: a_ones =torch.ones((2,2))
  - Zeros: a_zeros =torch.zeros((2,2))

Example:

```
import torch
import numpy as np
a= [[1., -1.], [1., -1.]]
a_torch= torch.tensor(a)                torch.float32
print(a_torch, a_torch.dtype)
a_np=np.array(a), print(a_np)
a_torch= torch.from_numpy(a_np)    torch.float64
print(a_torch, a_torch.dtype)
a_np=a_torch.numpy(), print(a_np)
```

Machine Learning with pytorch

- To develop a machine learning project with pytorch, one must learn to:
    1. Build a NN
    2. Get the needed data
    3. Prepare the NN for training
    4. Train the NN, and evaluate its performance.
- The coming slides explain the machine learning pipeline considering the above settings.

Building a Simple Neural Network

- torch.nn is a torch package that contains most of the things one might need when dealing with NNs
- To build a NN, we can define a class of type nn.Module, which typically contains two functions:
  - _ _ init _ _() to define the different layers used, and any other variable of different types that you know you will implement in your network.
  - forward() to make use of the defined layers, order them in a model, and define what happens in a forward pass, including the different activations, special orders…etc.
  - As for the backward pass, it is automatically defined using autograd (will be discussed later)

Building a Simple Neural Network

- Example:

```
import torch.nn as nn
import torch.nn.functional as F


class ExampleNN(nn.Module):
   def __init__(self):
      super().__init__()
      self.conv1 = nn.Conv2d(3, 6, 5)
      self.pool = nn.MaxPool2d(2, 2)
      self.conv2 = nn.Conv2d(6, 16, 5)
      self.fc1 = nn.Linear(16 * 5 * 5, 120)
      self.fc2 = nn.Linear(120, 84)
      self.fc3 = nn.Linear(84, 10)

   def forward(self, x):
      x = self.pool(F.relu(self.conv1(x)))
      x = self.pool(F.relu(self.conv2(x)))
      x = torch.flatten(x, 1) # flatten all dimensions except batch
      x = F.relu(self.fc1(x))
      x = F.relu(self.fc2(x))
      x = self.fc3(x)
      return x
net = ExampleNN()
```

The super().__init__() is important such that the class ExampleNN() can be subclassed by other classes (i.e. for the network to be used by pytorch later)

Building a Simple Neural Network

- Calling the model and initiating a forward pass can be then done by:

```
NN_example= ExampleNN()
input_ex=torch.rand(1,3,32,32)
input_ex = (input_ex-
torch.mean(input_ex))/torch.std(input_ex)
out=NN_example(input_ex)
print(out)
```

- It is important to zero-out the gradient records, and randomly initialize the backprop buffers:

```
NN_example.zero_grad()
```

Automatic Differentiation (AD)

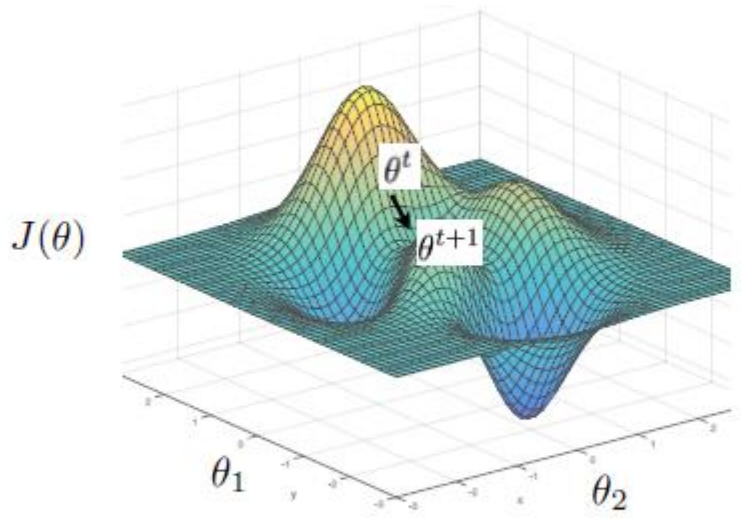
- AD is essential for neural network training. In pytorch, this is done through the engine torch.autograd
- All tensors, their operations, and the resulting tensors from these operations are recorded with autograd in a graph (specifically, a directed acyclic graph).
- The leaves of the graph are the input tensors, and the roots are the output tensors.

```
import autograd.numpy as au              # Thinly-wrapped numpy
from autograd import grad  # The only autograd function you may ever need
def exp_reducerNP(x):
  # Define a function
  y = au.exp(-2.0 * x)
  return y  #
grad_tanh = grad(exp_reducerNP)       # Obtain its gradient function
g= grad_tanh(1.0)                # Evaluate the gradient at x = 1.0
c=(exp_reducerNP(1.0001) -
 exp_reducerNP(0.9999)) / 0.0002  # Compare to finite differences

# Torch version
import torch
def exp_reducer(x):
   y=-2.0*x
   return y.exp()
a_torch = torch.tensor(1.0)
c_torch=torch.autograd.functional.jacobian(exp_reducer, a_torch)
print(g, c, c_torch)
```

AD

- To train a NN using gradient descent, we need to compute the gradient of the cost function w.r.t the model parameters.



$$\boxed{\theta^* = \arg\min_\theta \underbrace{\sum_{i=1}^N \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)}_{J(\theta)}}$$

One iteration of gradient descent:

$$\theta^{t+1} = \theta^t - \underset{\text{learning rate}}{\eta_t} \frac{\partial J(\theta)}{\partial \theta}\bigg|_{\theta=\theta^t}$$

AD

- The backpropagation algorithm is usually used, making use of the chain rule to get said gradient.
- Thus, there is a need to compute partial derivatives of each layer output, w.r.t to its parameters and input.
- Since all input tensors, parameters, operations, and output tensors are recorded in a graph, gradients from roots to leaves can be computed automatically (hence the name automatic differentiation)

AD

- In the forward pass, a NN makes a guess on the input, using the current state of the parameters. This produces an error which is used to correct the parameters.
- In the backward pass, autograd.backward() is called which computes all needed gradients, accumulates and saves them, and uses the chain rule to propagate the gradient all the way to the leaf tensor.

Backpropagation

- Having built a simple NN, and after understanding the basic idea behind torch.autograd, we now explain how the backpropagation algorithm can be implemented.
- Note that the torch.nn package has many different loss functions. A loss function takes a pair (output, target), and computes the resulting error.
- For example, the following applies the mean squared error (MSE) loss:

```
target = torch.empty(1, dtype=torch.long).random_(5)
NN_example = ExampleNN()
input_ex = torch.rand(1,3,32,32)
input_ex = (input_ex-torch.mean(input_ex))/torch.std(input_ex)
out = NN_example(input_ex)
loss_function = nn.CrossEntropyLoss()     # nn.MSELoss()   for regression problem
                                          # nn.CrossEntropyLoss()   for classification problem
loss = loss_function(out, target)
```

Backpropagation

- One backpropagation iteration can be realized using the aforementioned .backward() function on the loss tensor:

```
loss.backward()
```

- The loss tensor exists at the root of the graph. Calling .backward() on it will change all model parameters leading up to it (weights, biases)

Backpropagation

- As mentioned, the basic gradient descent algorithm updates the weights using

$$\theta^{t+1} = \theta^t - \underset{\text{learning rate}}{\eta_t} \frac{\partial J(\theta)}{\partial \theta}\bigg|_{\theta=\theta^t}$$

- This can be done manually, or one can choose from the available optimizers offered by torch.optim library to employ specific rules to update the weights.
- The optimizer holds the current state of the parameters, and updates them based on the computed gradients
- For example, stochastic gradient descent (SGD) can be used by:

```
import torch.optim as optim
optimizer = optim.SGD(net.parameters(), lr=0.01)
```

Backpropagation

- There are many learning rate optimization algorithms supported by `torch.optim` such as:
  - Adam
  - RMSprop
  - AdamW
  - Adadelta
  - Adagrad
  - SGD
  - And others…

Backpropagation

- To update the weights, optimizer.step() is called right after loss.backward(). This can be put in a loop for a number of iterations, while zeroing_out the gradients at the beginning of each iteration:

```
for i, data in enumerate(trainloader, 0):
    # get the inputs; data is a list of [inputs, labels]
    inputs, labels = data
    # zero the parameter gradients
    optimizer.zero_grad()
    # forward + backward + optimize
    outputs = net(inputs)
    loss = loss_function(outputs, labels)
    loss.backward()
    optimizer.step()
```

Getting the Data

- The torch.torchvision library consists of many datasets that can be readily downloaded and used,
- this is in addition to transform rules (such as normalization, and converting to tensors) that can be implemented using the sub-library torch.torchvision.transforms

Putting it Together: Training a Classifier

- The coming slides run through an example of building and training a classifier on the CIFAR10 dataset, going through all of the visited steps.

Step 1: importing

```
import torch
import torchvision
import torchvision.transforms as transforms
```

```
import torch.nn as nn
import torch.nn.functional as F
```

```
import torch.optim as optim
```

Step 2: data loading

- We prepare the wanted transformations. If a composition of transformations is wanted, transforms.Compose() is used:

```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

- We are setting the mean and standard deviation of all channels (3 in RGB) to 0.5, and 0.5

Step 2: data loading

- We set the batch size, get the training and testing data and save them in their designated folders, and define the different classes.
- We also wrap these datasets in loaders that will be used to feed our NN later. This makes things easier. Note how the batch size is stated in the data loader. See below the usage of the different functions.

```
batch_size = 64
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True, num_workers=2)
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=False, num_workers=2)
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

Step 3: defining the model

```
class ExampleNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
net = ExampleNN()
```

Step 4: defining the training engines

- The loss function is cross entropy:

```
loss_function = nn.CrossEntropyLoss()
```

- The optimizer is SGD:

```
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9)
```

Step 5: the training loop

- The training loop counts the epochs of the training. Remember that the batch size (and hence the needed number of iterations in each epoch) is already known to the data loader, so we do not need to worry about it.
- We can manually keep track of the loss as shown using loss.item(). Useful in case we want to report it.

```
for epoch in range(2):  # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = loss_function(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 200 == 199:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 200))
            running_loss = 0.0
print('Finished Training')
```

Step 6: evaluate the model

- We can now use the model to predict the images in the test data set and get the accuracy of the model

```
correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for our outputs
with torch.no_grad():
    for data in testloader:
        images, labels = data
        # calculate outputs by running images through the network
        outputs = net(images)
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % ( 100 * correct / total))
```

Working with GPUs

- Migrating your model and using it on a GPU is easy with pytorch
- First we define the device:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
```

- Then we migrate the model, along with its parameters and buffers, to the device

```
net.to(device)
```

- Also, in the training loop, the inputs and labels should be sent to the GPU at every iteration

```
inputs, labels = data[0].to(device), data[1].to(device)
```

Working with GPUs

Example: Speed comparison

```
import torch
import time
###GPU
start_time = time.time()
b = torch.ones(400,4000).cuda()
for _ in range(100):
    b += b
elapsed_time = time.time() - start_time
print('GPU time = ',elapsed_time)
###CPU
start_time = time.time()
a = torch.ones(4000,4000)
for _ in range(100):
    a += a
elapsed_time = time.time() - start_time

print('CPU time = ',elapsed_time)
```

```
GPU time = 0.04393911361694336   second
CPU time = 0.8792381286621094     second
```

GPU is 20x faster than CPU
For this example

```
def EndOfLecture(statment):
   return statment
statment = 'Tank you for your listening'
print(EndOfLecture(statment))
```