[Lab] Introduction to PyTorch

Jae Yun JUN KIM[*]

November 17, 2020

Source:

- Pradeepta Mishra, PyTorch recipes: a problem solution approach, Apress, 2019

1 Introduction

PyTorch is a library for Python programs that facilitates building deep learning projects. PyTorch has grown into one of the most prominent deep learning tools across a broad range of applications. It has been proven to be fully qualified for use in professional contexts for real-world, high-profile work. Its clear syntax, streamlined API, and easy debugging make it an excellent choice for introducing deep learning.

The deep learning machine is a rather complex mathematical function mapping inputs to an output. To facilitate expressing this function, PyTorch provides a core data structure, the *tensor*, which is a multidimensional array that shares many similarities with NumPy arrays. PyTorch comes with features to perform accelerated mathematical operations on dedicated hardware, which makes it convenient to design neural network architectures and train them on individual machines or parallel computing resources.

2 Installation

2.1 Through Anaconda

1. Open the Anaconda navigator and go to the environment page as shown below and open a terminal

2. Once an Anaconda terminal is opened, then type

```
conda install -c peterjc123 pytorch
```

3. Launch Jupyter (for instance) and open the IPython Notebook.

4. Type the following command to check whether the the PyTorch is installed or not.

```
from __future__ import print_function
import torch
```

5. Check the version of the PyTorch.

[*]ECE Paris Graduate School of Engineering, 37 quai de Grenelle 75015 Paris, France; jae-yun.jun-kim@ece.fr

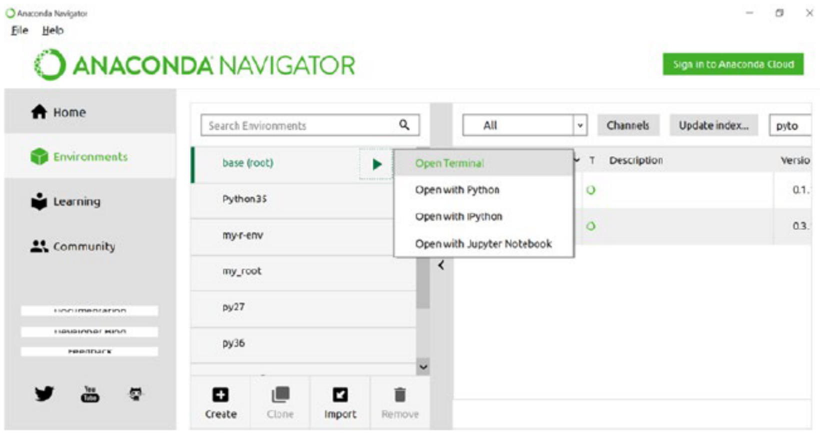


Figure 1: Anaconda navigator

```
torch.version.__version__
```

The other way to install PyTorch (besides the Conda (Anaconda) library management) is using the **Pip3** pacakge management framework. Also, the installation procedure differs between for local systems (such as MaOS, Windows, or Linux) and for cloud machines (such as Microsoft Azure, AWS, and GCP). For more details, follow the instructions given at https://PyTorch.org/get-started/cloud-partners/.

PyTorch has various components:

- Torch has functionalities similar to NumPy with GPU support.
- **Autograd**'s torch.autograd provides classes, methods, and functions for implementing automatic differentiation of arbitrary scalar valued functions. It requires minimal changes to the existing code. You only need to declare **class:'Tensor's**, for which gradients should be computed with the **requires_grad** = **True** keyword.
- **NN** is a neural network library in PyTorch.
- **Optim** provides optimization algorithms that are used for the minimization and maximization of functions.
- **Multiprocessing** is a useful library for memory sharing between multiple tensors.
- **Utils** has utility functions to load data: it has also other functions.

3 PyTorch essential

3.1 Using Tensors

The data structure used in PyTorch is based on **graph** and **tensor**.
We can check whether an object in Python is a tensor object using the **is_tensor** function. We use **is_storage** function to check whether an object is stored as a tensor object. Type the following Python instructions as an example:

```
x = [12,23,34,45,56,67,78]
print(torch.is_tensor(x))
print(torch.is_storage(x))
```

Observe that **False** values are returned.

Let us create an object that contains random numbers from Torch similar to Numpy library. Then, let us check whether it is a tensor object and is stored as it is.

```
y = torch.randn(1,2,3,4,5)
print(torch.is_tensor(y))
print(torch.is_storage(y))
print(torch.numel(y)) # the total number of elements in the input Tensor
```

We should observe that y object is a tensor, but it is not stored. We can check the total number of elements in the tensor object using the **numel()** function.

We can create a 2D tensor with zero values as follows:

```
print(torch.zeros(4,4))
print(torch.numel(torch.zeros(4,4)))
```

We can create a diagonal matrix using the **eye()** function as we usually do with the NumPy library:

```
print(torch.eye(3,4))
print(torch.eye(5,4))
```

We can create a linear space and points between the linear space using tensor operations.

```
import numpy as np
x1 = np.array(x)
print(x1)
print(torch.from_numpy(x1))
```

Let us create a linear space with 25 points starting from value 2 and ending with 10:

```
torch.linspace(2,10,steps=25) # linear spacing
```

Like linear spacing, logarithmic spacing can be craeted.

```
torch.logspace(start=-10,end=10,steps=15) # logarithmic spacing
```

Like NumPy functions, random numbers can be generated as

```
print(torch.rand(10))  # 10 uniformly-distributed random values ranging between 0 and 1
print(torch.rand(4,5)) # 4x5 uniformly-distributed random values ranging between 0 and 1
print(torch.randn(10)) # 10 normally-distributed random values with mean=0 and standard deviation=1
print(torch.randn(4,5)) # 4x5 normally-distributed random values with mean=0 and standard deviation=1
```

We can create a sequence of values ranging from a start value to an end value with certain step size as

```
print(torch.arange(10,40)) # default step size = 1
print(torch.arange(10,40,2)) # step size = 2
```

To select values from a range using random permutation

```
torch.randperm(10) # randomly permute from 0 to 10.
```

For a given tensor, to find the index of the minimum value or of the maximum value along certain dimension we can do

```
d = torch.randn(4,5)
print(d)
print(torch.argmin(d,dim=1))
print(torch.argmax(d,dim=1))
```

We can create tensors with zero values

```
print(torch.zeros(4,5))  # create a 2D tensor filled with zero values.
print(torch.zeros(10))  # create a 1D tensor filled with zero values.
```

We can concatenate two tensors

```
x = torch.randn(4,5)
a = torch.cat((x,x)) # by default, dim=0, along rows
b = torch.cat((x,x),dim=1) # along columns
print(a)
print(a.shape)
print(b)
print(b.shape)
```

We can concatenate in 3D, and there are two different options; the third dimension can be extended over rows or columns:

```
a = torch.cat((x,x,x),0) # concatenate x n times over row
b = torch.cat((x,x,x),1) # concatenate x n times over column
print(a)
print(a.shape)
print(b)
print(b.shape)
```

We can split a tensor between multiple chunks, which can be created along rows or columns:

```
a = torch.randn(4,4)
print(a)
print(torch.chunk(a,2))
print(torch.chunk(a,2,0))
print(torch.chunk(a,2,1))
```

The **gather()** function collects elements from a tensor and places it in another tensor using an index argument. The index position is determined by the **LongTensor()** function in PyTorch.

```
b = torch.Tensor([[11,12],[23,24]])
print(b)
print(torch.gather(b, 1, torch.LongTensor([[0,0],[1,0]])))
```

The **LongTensor** function or the **index_select** function can be used to fetch relevant values from a tensor. One can select along rows (the corresponding argument being 0) or along columns (the corresponding argument being 1).

```
a = torch.randn(4,4)
print(a)
indices = torch.LongTensor([0,2])
print(indices)
print(torch.index_select(a,0,indices))
print(torch.index_select(a,1,indices))
```

It is a common practice to check non-missing values in a tensor, the objective is to identify non-zero elements in a large tensor.

```
# identify null input tensors using nonzero function
torch.nonzero(torch.tensor([10,0,23,0,0.0]))
```

Reconstructing the input tensors into smaller tensors not only fastens the calculation process, but also helps in distributed computing. The **split()** function splits a long tensor into smaller tensors:

```
# splitting the tensor into two small chunks.
print(torch.split(torch.tensor([12,21,34,32,4,54,56,65]),2))

# splitting the tensor into three small chunks.
print(torch.split(torch.tensor([12,21,34,32,4,54,56,65]),3))
```

We can transpose a tensor either using **.t** or **.transpose** functions.

```
x = torch.randn(4,5)
print(x)
print(x.t()) # transpose
print(x.transpose(1,0)) # transpose
```

The **unbind()** function removes a dimension from a tensor. To remove the row dimension, we pass 0 value to the function. To remove the column dimension, we pass 1.

```
x = torch.randn(4,5)
print(x)
print(torch.unbind(x,1)) # Remove the column dimension
print(torch.unbind(x)) # Remove the row dimension
```

The scalar multiplications and additions with 1D tensors can be done using the **add()** and **mul()** functions:

```
x = torch.randn(4,5)
print(x)
print(torch.add(x,20)) # scalar addition
print(torch.mul(x,2)) # scalar multiplication

z = torch.randn(2,2)
print(z)
beta = 0.7456
intercept = torch.randn(1)
print(intercept)
y = torch.add(intercept, torch.mul(z,beta)) # y = intercept + (beta * z)
print(y)
```

Like NumPy operations, the tensor values can be rounded up either by the **ceil()** or **floor()** functions.

```
torch.manual_seed(1234)
print(torch.randn(5,5))

torch.manual_seed(1234)
print(torch.ceil(torch.randn(5,5)))

torch.manual_seed(1234)
print(torch.floor(torch.randn(5,5)))
```

We can limit the values of any tensor within a certain range using the **clamp()** function.

```
torch.manual_seed(1234)
print(torch.clamp(torch.floor(torch.randn(5,5)), min=-0.3, max=0.4))

torch.manual_seed(1234)
print(torch.clamp(torch.floor(torch.randn(5,5)), min=-0.3))

torch.manual_seed(1234)
print(torch.clamp(torch.floor(torch.randn(5,5)), max=0.4))
```

We can apply **exp()**, **log()**, **pow()** functions to tensors.

```
torch.manual_seed(1234)
a = torch.randn(5,5)
print(a)
print(torch.exp(a))

torch.manual_seed(1234)
a = torch.randn(5,5)
print(a)
print(torch.log(a))

torch.manual_seed(1234)
```

```
a = torch.randn(5,5)
print(a)
print(torch.pow(a,2))
```

We can get the fractional portion of each tensor.

```
torch.manual_seed(1234)
x = torch.randn(5,5)
print(x)
print(torch.frac(torch.add(x,10)))
```

To compute the transformation functions such as **sigmoid**, **hyperbolic tangent**, **radial basis function** using PyTorch, we first need to construct tensors.

```
torch.manual_seed(1234)
a = torch.randn(5,5)
print(a)
print(torch.sigmoid(a))

torch.manual_seed(1234)
a = torch.randn(5,5)
print(a)
print(torch.sqrt(a)) # finding the square root of the values.
```

3.2 Probability distributions using PyTorch

We can sample tensors following various types of probability distributions using PyTorch as follows:

```
import torch

# Uni-variate standard Gaussian
torch.manual_seed(1234)
print(torch.randn(4,4))

# random number from uniform distribution
torch.manual_seed(1234)
print(torch.Tensor(4,4).uniform_(0,1))

# Bernoulli by considering uniformly-distributed random values
torch.manual_seed(1234)
print(torch.bernoulli(torch.Tensor(4,4).uniform_(0,1)))

# Uni-variate Gaussian
print(torch.normal(mean=torch.arange(1., 11.), std=torch.arange(1,0,-0.1)))
print(torch.normal(mean=0.5, std=torch.arange(1.,6.)))
print(torch.normal(mean=0.5, std=torch.arange(0.2,0.6)))
```

We can also generate a multinomial distribution random values. We can choose the mode of with replacement or the mode of without replacement. The default one is the multinomial distribution without replacement.

```
a = torch.tensor([10., 10., 13., 10.,
                  34., 45., 65., 67.,
                  87., 89., 87., 34.])
print(a)
print(torch.multinomial(a,3))
print(torch.multinomial(a,5, replacement=True))
```

3.3 Variable tensors

In PyTorch, the algorithms are represented as a computational graph. A variable is considered as a representation around the tensor object, corresponding gradients, and a reference to the function from where it was created.

Basically, a PyTorch variable is a node in a computational graph, which stores data and gradients.
Figure ?? explains how the linear regression equation is deployed under the hood using a neural network model in the PyTorch framework.
When training a neural network model, after each iteration, we need to compute the gradient of the loss function with respect to the parameters of the model, such as weights and biases. After that, we usually update the weights using the gradient descent algorithm.
In a computational graph structure, the sequencing and ordering of tasks is very important.
In Figure 3.3, the one-dimensional tensors are X, Y, W, and alpha. The direction of the arrows change when we implement backpropagation to update the weights and biases to match with Y, so that the error or loss function between Y and predicted Y can be minimized.

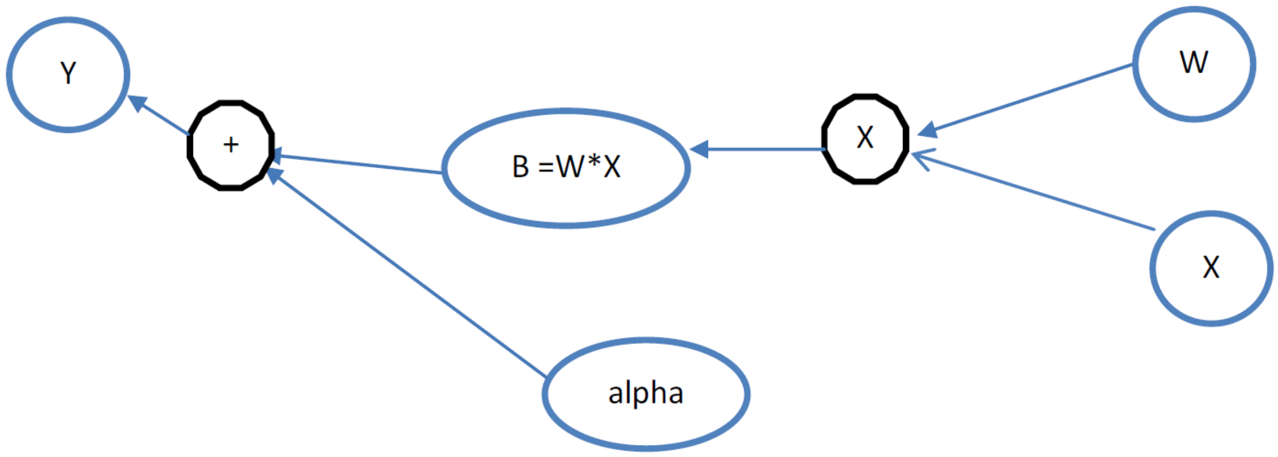


Figure 2: PyTorch as a computational graph

Consider the following example for how a computational graph can be created using variables:

```
from torch.autograd import Variable
a, b = 12, 23
x1 = Variable(torch.randn(a,b), requires_grad = True)
x2 = Variable(torch.randn(a,b), requires_grad = True)
x3 = Variable(torch.randn(a,b), requires_grad = True)

c = x1 * x2
```

```
d = c + x3
e = torch.sum(d)

e.backward()

print(e)
```

In this example, three variable objects around tensors **x1**, **x2**, and **x3**, with random values generated between 12 and 23. This computational graph involves only multiplication and addition, and the final result with the gradient.
On the other hand,the partial derivative of the loss function with respect to the weights and biases in a neural network model is achieved by PyTorch using the **Autograd** module. Variables are specifically designed to hold the changed values while running a backpropagation in a neural network model when the parameters of the model change. The variable type is just a wrapper around the tensor. It has three properties: **data**, **grad**, and **function**.

3.4 Basic statistics

How do we compute basic statistics such as mean, median, mode and so forth from a Torch tensor?
We can compute mean of tensors as follows:

```
# computing the descriptive statistics: mean
print(torch.mean(torch.tensor([10., 10., 13., 10., 34., 45.,
                         65., 67., 87., 89., 87., 34.])))

# mean across rows and across columns
d = torch.randn(4,5)
print(d)
print(torch.mean(d,dim=0)) # across rows
print(torch.mean(d,dim=1)) # across columns
```

We can compute median as follows:

```
# median across rows and across columns
d = torch.randn(4,5)
print(d)
print(torch.median(d,dim=0)) # across rows
print(torch.median(d,dim=1)) # across columns
```

We can compute mode as follows:

```
# mode across rows and across columns
d = torch.randn(4,5)
print(d)
print(torch.mode(d))
print(torch.mode(d,dim=0)) # across rows
print(torch.mode(d,dim=1)) # across columns
```

We can also compute standard deviation (variance), which measures the deviation of the data from the central tendency. It shows whether there is enough fluctuation in data or not.

```
# standard deviation across rows and across columns
d = torch.randn(4,5)
print(d)
print(torch.std(d))
print(torch.std(d,dim=0)) # across rows
print(torch.std(d,dim=1)) # across columns

# compute variance across rows and across columns
d = torch.randn(4,5)
print(d)
print(torch.var(d))
print(torch.var(d,dim=0))
print(torch.var(d,dim=1))
```

3.5 Gradient computation

How do we compute basic gradients from the sample tensors using PyTorch?
In this section, we consider a dataset with two variables (x and y). Withe some given initial weight values, we compute the gradients in each iteration as shown in the following example:

```
# using forward pass
def forward(x):
  return x*w

def loss(x,y):
  y_pred = forward(x)
  return (y_pred - y) * (y_pred - y)

import torch
from torch.autograd import Variable

x_data = [11.0, 22.0, 33.0]
y_data = [21.0, 14.0, 64.0]

w = Variable(torch.Tensor([1.0]), requires_grad = True)

# before training
print("predict (before training)", 4, forward(4).data[0])

# run the training loop
for epoch in range(10):
  for x_val, y_val in zip(x_data, y_data):
    l = loss(x_val, y_val)
    l.backward()
    print("\tgrad: ", x_val, y_val, w.grad.data[0])
    w.data = w.data - 0.01 * w.grad.data

    # manually set the gradients to zero after updating weights
    w.grad.data.zero_()

  print("progress:", epoch, l.data[0])

# after training
print("predict (after training)", 4, forward(4).data[0])
```

The following program shows how to compute the gradients from a loss function using the variable method on the tensor.

```
from torch import FloatTensor
from torch.autograd import Variable

a = Variable(FloatTensor([5]))

weights = [Variable(FloatTensor([i]), requires_grad=True) for i in (12, 53, 91,73)]

w1, w2, w3, w4 = weights

b = w1 * a
c = w2 * a
d = w3 * b + w4 * c
Loss = (10 - d)

Loss.backward()

for index, weight in enumerate(weights, start=1):
  gradient, *_ = weight.grad.data
  print(f"Gradient of w{index} w.r.t. Loss: {gradient}")
```

3.6 Tensor operations

How do we compute or perform operations based on variables?
Recall that tensors are wrapped within the variable, which has three properties: grad, and gradient.
Let us create a variable and extract the properties of the variable:

```
x = Variable(torch.Tensor(4,4).uniform_(-4,5))
y = Variable(torch.Tensor(4,4).uniform_(-3,2))

# matrix multiplication
z = torch.mm(x,y)
print(z.size())
print('Requires Gradient : %s ' % (z.requires_grad))
print('Gradient: %s ' % (z.grad))
print(z.data)
```

```
from torch import FloatTensor
from torch.autograd import Variable

# tensors' definition
mat1 = torch.FloatTensor(4,4).uniform_(0,1)
print(mat1)

mat2 = torch.FloatTensor(5,4).uniform_(0,1)
print(mat2)

vec1 = torch.FloatTensor(4).uniform_(0,1)
print(vec1)

# scalar addition
print(mat1 + 10.5)

# scalar subtraction
print(mat2 - 0.20)

# vector and matrix addition
print(mat1 + vec1)
print(mat2 + vec1)

# matrix product
print(mat1 * mat1)
```

3.7 Distributions

The knowledge of statistical distributions is essential for weight normalization, weight initialization, and computation of gradients in neural network-based operations using PyTorch. Let us see these distribution through the following examples:

```
# about Bernoulli distribution
from torch.distributions.bernoulli import Bernoulli
dist = Bernoulli(torch.tensor([0.3,0.6,0.9]))
print(dist)
dist.sample() # sample is binary, it takes 1 with p and 0 with 1-p
```

The **beta distribution** is a family of continuous random variables defined in the range of 0 and 1. This distribution is typically used for Bayesian inference analysis

```
from torch.distributions.beta import Beta
dist = Beta(torch.tensor([0.3]), torch.tensor([0.5]))
print(dist)
dist.sample()
```

The **binomial distribution** is applicable when the outcome is twofold and the experiment is repetitive. It belongs to the family of discrete probability distribution, where the probability of success is defined as 1 and the probability of failure is 0. The binomial distribution is used to model the number of successful events overs many trials.

```
from torch.distributions.binomial import Binomial

# count of trials: 100
# 0, 0.2, 0.8, and 1 are event probabilities.
dist = Binomial(100, torch.tensor([0, .2, .8, 1]))
print(dist)
dist.sample()
```

A categorical distribution can be defined as a **generalized Bernoulli distribution**, which is a discrete probability distribution that explains the possible results of any random variable that may take on one of the possible categories, with the probability of each category exclusively specified in the tensor.

```
from torch.distributions.categorical import Categorical

# 0.2, 0.2, 0.2, 0.2, 0.2 are event probabilities.
dist = Categorical(torch.tensor([0.2, 0.2, 0.2, 0.2, 0.2]))
print(dist)
dist.sample()
```

A **Laplacian distribution** is a continuous probability distribution function that is otherwise known as a **double exponential distribution**. A Laplacian distribution is used in speech recognition systems to understand probabilities. It is also useful in Bayesian regression for deciding prior probabilities.

```
from torch.distributions.laplace import Laplace

# Laplace distribution parameterized by 'loc' and 'scale'
dist = Laplace(torch.tensor([10.0]), torch.tensor([0.990]))
print(dist)
dist.sample()
```

A **normal distribution** is very useful because of the property of central limit theorem. It is defined by mean and standard deviations. If we know the mean and standard deviation of the distribution, we can estimate the event probabilities.

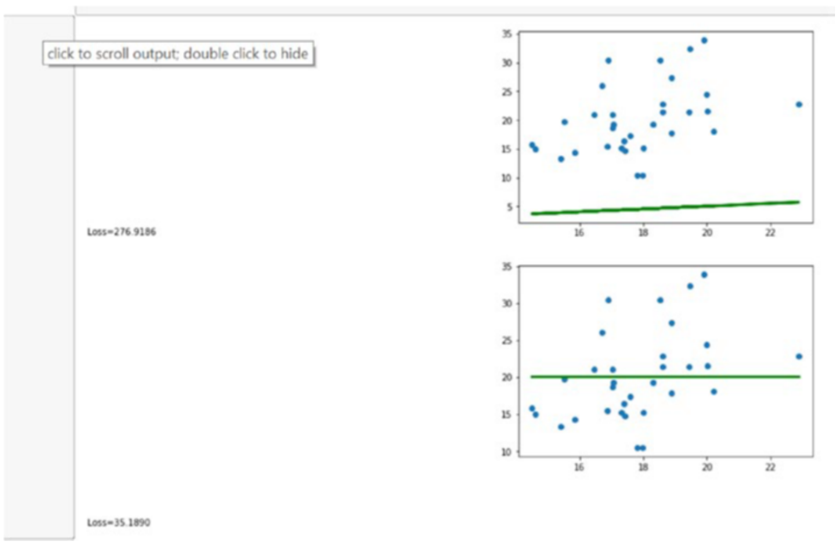
```
from torch.distributions.normal import Normal

# Normal distribution parameterized by 'loc' and 'scale'
dist = Normal(torch.tensor([100.0]), torch.tensor([10.0]))
print(dist)
dist.sample()
```

4 Linear regression

```
import torch
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from torch.autograd import Variable
import torch.nn.functional as F
%matplotlib inline


torch.__version__
df = pd.read_csv("./mtcars.csv")
df.head()

torch.manual_seed(1234)
x = torch.unsqueeze(torch.from_numpy(np.array(df.qsec)),dim=1)
y = torch.unsqueeze(torch.from_numpy(np.array(df.mpg)),dim=1)

class Net(torch.nn.Module):
  def __init__(self, n_feature, n_hidden, n_output):
    super(Net, self).__init__()
    self.hidden = torch.nn.Linear(n_feature, n_hidden) # hidden layer
    self.predict = torch.nn.Linear(n_hidden, n_output) # output layer

  def forward(self, x):
    x = F.relu(self.hidden(x)) # activation function for hidden layer
    x = self.predict(x) # linear output
    return x

net = Net(n_feature=1, n_hidden=20, n_output=1)
net.double()
print(net) # Neural network architecture

optimizer = torch.optim.SGD(net.parameters(), lr=0.2)
loss_func = torch.nn.MSELoss() # this is for regression mean squared loss.
print(optimizer)
print(loss_func)

# turn the interactive mode on
plt.ion()


for t in range(100):
  prediction = net(x) # input x and predict based on x
  loss = loss_func(prediction, y) # must be (1. nn output, 2. target)
  optimizer.zero_grad() # clear gradients for next train
  loss.backward() # backpropagation, compute gradients
  optimizer.step() # apply gradients
```

```
    if t % 50 == 0:
      # plot and show learning process
      plt.cla()
      plt.scatter(x.data.numpy(), y.data.numpy())
      plt.plot(x.data.numpy(), prediction.data.numpy(), 'g-', lw=3)
      plt.text(0.5, 0, 'Loss=%.4f' % loss.data.numpy())
      plt.show()
  plt.ioff()
```

As an output of the above code you should get the plots shown in Figure 4.



```
Loss=276.9186
```

```
Loss=35.1890
```

Figure 3: Solving a regression problem with a neural network

5 Classification

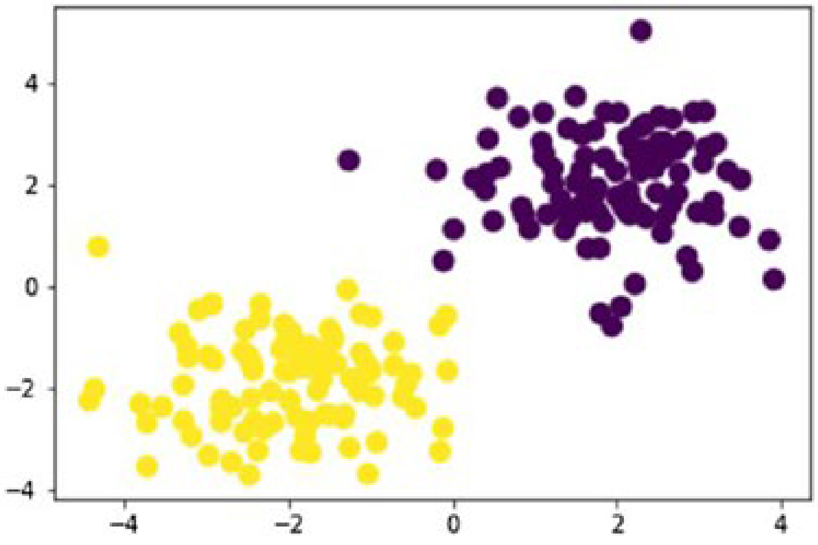


Figure 4: A classification problem

```
import torch
import matplotlib.pyplot as plt
from torch.autograd import Variable
import torch.nn.functional as F
%matplotlib inline

torch.manual_seed(1) # reproducible

# sample data preparation
n_data = torch.ones(100,2)
x0 = torch.normal(2*n_data,1)
y0 = torch.zeros(100)
x1 = torch.normal(-2*n_data, 1)
y1 = torch.ones(100)
x = torch.cat((x0,x1),0).type(torch.FloatTensor)
y = torch.cat((y0,y1),).type(torch.LongTensor)

# torch need to train on Variable, so convert sample features to Variable
x, y = Variable(x), Variable(y)

plt.scatter(x.data.numpy()[:,0], x.data.numpy()[:,1], c=y.data.numpy(), s=100)
plt.show()


class Net(torch.nn.Module):
  def __init__(self, n_feature, n_hidden, n_output):
    super(Net, self).__init__()
    self.hidden = torch.nn.Linear(n_feature, n_hidden) # hidden layer
    self.out = torch.nn.Linear(n_hidden, n_output) # output layer
  def forward(self, x):
    x = torch.sigmoid(self.hidden(x)) # activation function for hidden layer
    # x = F.sigmoid(self.hidden(x)) # activation function for hidden layer
    x = self.out(x)
    return x

net = Net(n_feature=2, n_hidden=10, n_output=2) # define the network
# net.double()
print(net) # Neural network architecture

# Loss and optimizer
# Softmax is internally computed.
# Set parameters to be updated
optimizer = torch.optim.SGD(net.parameters(), lr=0.02)
loss_func = torch.nn.CrossEntropyLoss() # the target label is not an one-hotted.
print(optimizer)
print(loss_func)

# turn the interactive mode on
plt.ion()


for t in range(100):
  out = net(x) # input x and predict based on x
  loss = loss_func(out, y) # must be (1. nn output, 2. target)
  optimizer.zero_grad() # clear gradients for next train
  loss.backward() # backpropagation, compute gradients
  optimizer.step() # apply gradients

  if t % 10 == 0 or t in [3,6]:
    # plot and show learning process
    plt.cla()
    _, prediction = torch.max(F.softmax(out),1)
    pred_y = prediction.data.numpy().squeeze()
    target_y = y.data.numpy()
    plt.scatter(x.data.numpy()[:,0],
                x.data.numpy()[:,1],
                c=pred_y, s=100, lw=0)
    accuracy = sum(pred_y == target_y)/200.
    plt.text(1.5,-4, 'Accuracy=%.2f' % accuracy,
             fontdict={'size': 20, 'color': 'blue'})
    plt.show()
plt.ioff()
```

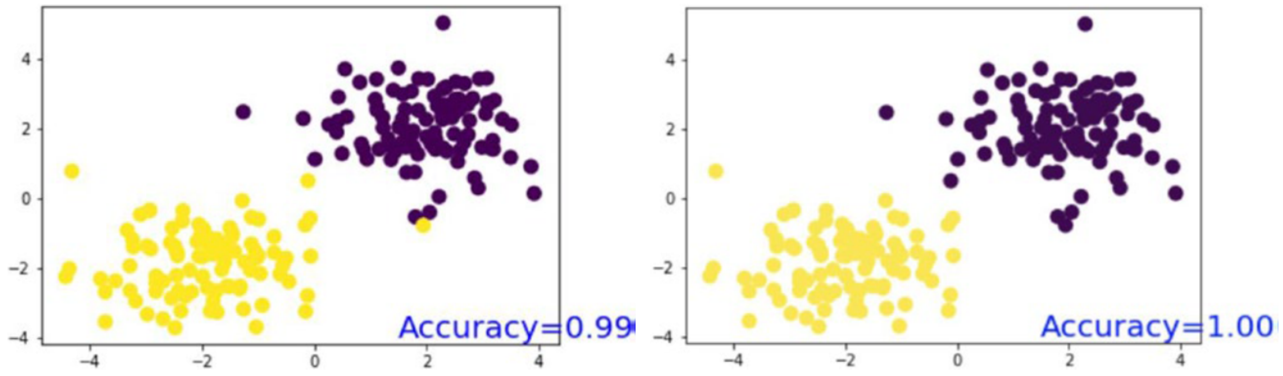


Figure 5: Solving a classification problem with a neural network

6 Fine-tuning deep learning models using PyTorch

Deep learning models have roots in the way that biological neurons are connected and in the way that they transmit information from one node to another one in the neural network model.

When the data dimension is very large (in the order of 1000's or more), then standard machine learning algorithms

- may fail to predict the outcome variable.
- may not be computationally efficient.
- may consume a very large number of resources and may never converge.

Some prominent applications that may be solved using deep learning algorithms are:

- object detection (places a bounding box around specific objects)
- image classification (identifies what is in an image)
- image segmentation (provides the outline of an object within an image in the level of pixel-by-pixel)

The most commonly used deep learning algorithms are:

- **Convolutional neural network**: mostly suitable for highly sparse datasets, image classification, image detection, etc.
- **Recurrent neural network**: applicable to processing sequential information, if there is an internal sequential structure. This includes music, natural language, audio, and video, where the information is consumed in a sequence.
- **Deep neural network**: typically applicable when a single layer of a machine learning algorithm cannot predict correctly. There are three variants:
  - **Deep network**: where the number of neurons present in each hidden layer is usually more than the previous layer.
  - **Wide network**, where the number of hidden layers are more than a usual neural network model.

- **Both deep and wide network**: where the number of neurons and the number of layers in the network is very high.

In this section, we discuss how to fine-tune hyperparameters in deep learning models.

6.1 Building sequential neural networks

Is there a way to build sequential neural network models (as we do with Keras)?

If we declare the entire neural network model, line by line, with the number of neurons, number of hidden layers and iterations, choice of loss functions, optimization functions, and the selection of weight distribution, and so forth, it will be extremely cumbersome to scale the model. To avoid the issues in declaring the entire model line by line, we can use a high-level function that assumes certain default parameters in the back end and returns the result to the user with minimum hyperparameters.

In the Torch library, the neural network module contains functional APIs (application programming interfaces) that define various activation functions. For this, we need to declare **class Net**, declaring features, hidden neurons, and activation functions. See the following example:

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

# we can replace the following class code with an easy sequential network
class Net(torch.nn.Module):
  def __init__(self, n_feature, n_hidden, n_output):
    super(Net, self).__init__()
    self.hidden = torch.nn.Linear(n_feature, n_hidden) # hidden layer
    self.predict = torch.nn.Linear (n_hidden, n_output) # output layer

  def forward(self, x):
    x = F.relu(self.hidden(x)) # activation function for hidden layer
    x = self.predict(x)        # linear output
    return x

net1 = Net(1, 100, 1)
print(net1)
```

Instead of using the above script (which creates a neural network using functional APIs), we can create a neural network using sequential APIs as follows:

```
# easy and fast way to build a neural network
net2 = torch.nn.Sequential(torch.nn.Linear(1, 100),
                           torch.nn.ReLU(),
                           torch.nn.Linear(100,1))
print(net2)
```

If we print both the *net1* and *net2* model architectures, we should observe that they do the same tasks

```
print(net1) # net1 architecture created using functional APIs
print(net2) # net2 architecture created using sequential APIs
```

6.2 Deciding the batch size

How do we perform batch data training for a deep learning model using PyTorch?

```
import torch
import torch.utils.data as Data

torch.manual_seed(1234) # reproducible
batch_size = 5

x = torch.linspace(1, 10, 10) # this is x data (torch tensor)
y = torch.linspace(10, 1, 10) # this is y data (torch tensor)

torch_dataset = Data.TensorDataset(x,y)
loader = Data.DataLoader(
  dataset = torch_dataset, # torch TensorDataset format
  batch_size = batch_size, # mini batch size
  shuffle = True,          # random shuffle for training
  num_workers = 2          # subprocesses for loading data
)

for epoch in range(5): # train entire dataset 5 times
  # for each training step
  for step, (batch_x, batch_y) in enumerate(loader):
    # write your code here to train data in mini-batch
    print('Epoch: ', epoch, '| Step: ', step, '| batch x: ',
          batch_x.numpy(), '| batch y: ', batch_y.numpy())
```

6.3 Deciding the learning rate

How do we identify the best solution based on learning rate and the number of epochs?

The learning rate and the epoch number are associated with model accuracy. To reach the global minimum value of a loss function, we should keep the learning rate to a minimum and the epoch number to a maximum to achieve a minimum loss.

```
import torch
import torch.utils.data as Data
import torch.nn.functional as F
from torch.autograd import Variable
import matplotlib.pyplot as plt
%matplotlib inline

torch.manual_seed(12345) # reproducible
```

```
LR = 0.01
BATCH_SIZE = 32
EPOCH = 12

# sample dataste
x = torch.unsqueeze(torch.linspace(-1, 1, 1000), dim=1)
y = x.pow(2) + 0.3*torch.normal(torch.zeros(*x.size()))

# plot dataset
plt.scatter(x.numpy(), y.numpy())
plt.show()

# load the tensor dataset
torch_dataset = Data.TensorDataset(x,y)
loader = Data.DataLoader(
  dataset = torch_dataset,
  batch_size = BATCH_SIZE,
  shuffle = True,
  num_workers = 2)

# declare and create neural netwokrs using functional APIs
class Net(torch.nn.Module):
  def __init__(self):
    super(Net, self).__init__()
    self.hidden = torch.nn.Linear(1, 20) # hidden layer
    self.predict = torch.nn.Linear (20, 1) # output layer

  def forward(self, x):
    x = F.relu(self.hidden(x)) # activation function for hidden layer
    x = self.predict(x)        # linear output
    return x

net_SGD      = Net()
net_Momentum = Net()
net_RMSprop  = Net()
net_Adam     = Net()
nets = [net_SGD, net_Momentum, net_RMSprop, net_Adam]


# include options for optimization for a better search
opt_SGD      = torch.optim.SGD(net_SGD.parameters(), lr=LR)
opt_Momentum = torch.optim.SGD(net_Momentum.parameters(),
                               lr=LR, momentum=0.8)
opt_RMSprop  = torch.optim.RMSprop(net_RMSprop.parameters(),
                                   lr=LR, alpha=0.9)
opt_Adam     = torch.optim.Adam(net_Adam.parameters(),
                                lr=LR, betas=(0.9,0.99))
optimizers = [opt_SGD, opt_Momentum, opt_RMSprop, opt_Adam]
```

```
loss_func = torch.nn.MSELoss()
losses_his = [[],[],[],[]] # record loss

# training
for epoch in range(EPOCH):
  print('Epoch: ', epoch)
  for step, (batch_x, batch_y) in enumerate(loader):
    b_x = Variable(batch_x)
    b_y = Variable(batch_y)

    for net, opt, l_his in zip(nets, optimizers, losses_his):
      output = net(b_x)              # get otuput for every net
      loss = loss_func(output, b_y)  # compute loss fro every net
      opt.zero_grad()                # clear gradients for next train
      loss.backward()                # backpropagation, compute gradients
      opt.step()                     # apply gradients
      #print(loss.item())
      l_his.append(loss.data)     # loss recorder

labels = ['SGD', 'Momentum', 'RMSprop', 'Adam']
for i, l_his in enumerate(losses_his):
  plt.plot(l_his, label=labels[i])
plt.legend(loc='best')
plt.xlabel('Steps')
plt.ylabel('Loss')
plt.ylim((0,0.2))
plt.show()
```

When we run the above script, we should get the following results:

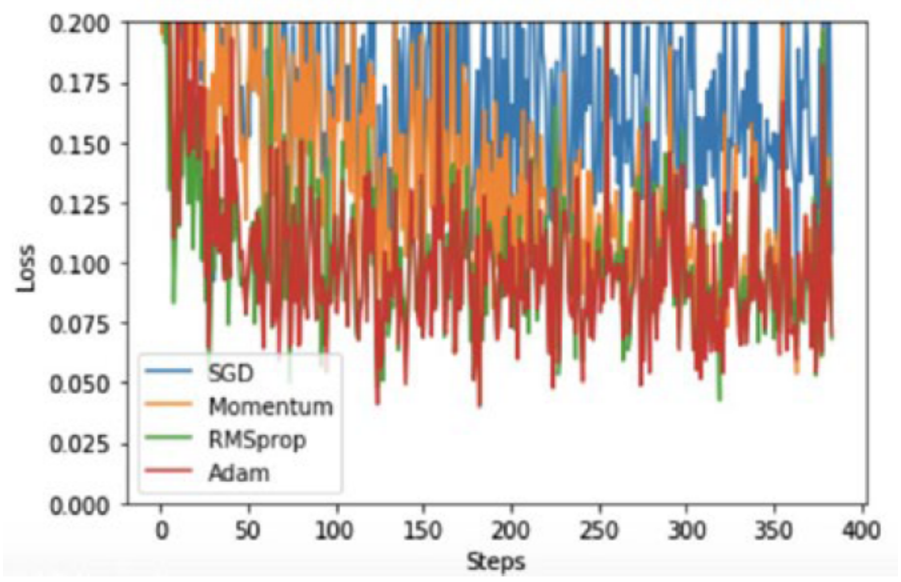


Figure 6: Loss values for various nets

7 Neural networks using PyTorch

7.1 Working with activation functions

What are the activation function? How do they work? How can it be implemented using PyTorch?

All activation functions can be classified as linear functions and nonlinear functions. The PyTorch torch.nn module creates any type of a neural network model.

The core differences between PyTorch and TensorFlow is the way a computational graph is defined, the way the two frameworks perform calculations, and the amount of flexibility we have in changing the script introducing other Python-based libraries in it.

- **TensorFlow**: we need to
  - define the variables and placeholders before we initialize the model.
  - keep track of objects that we need later. For this, we need a placeholder.
  - define the model first, and then compile and run
- **PyTorch**: we need to
  - define the model as we go (do not need to keep placeholders in the code). Hence, PyTorch is a dynamic framework.

7.1.1 Linear and bilinear function

Linear function

- is used in the problems in which the data vary slowly.
- has the output confined to a specific range.
- is used typically in the last hidden layer to the output layer or for linear regression-based tasks.
- has the following formula

$$y = \alpha + \beta x.$$

A **bilinear function** is a simple function typically used to transfer information as

$$y = x_1 \cdot A \cdot x_2 + b$$

The following script shows some examples of a linear function and a bilinear function.

```
from __future__ import print_function
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.init as init
import torch.nn.functional as F
from torch.autograd import Variable

# torch.nn: Neural networks can be constructed using the torch.nn package.
```

```
x = Variable(torch.randn(100,10))
y = Variable(torch.randn(100,30))

linear = nn.Linear(in_features=10, out_features=5, bias=True)
output_linear = linear(x)
print('Output size : ', output_linear.size())

bilinear = nn.Bilinear(in1_features=10, in2_features=30, out_features=5, bias=True)
output_bilinear = bilinear(x,y)
print('Output size: ', output_bilinear.size())
```

7.1.2 Sigmoid function

A **sigmoid function**

- captures the nonlinearity present in the data.
- has a simple expression for its gradient expression.
- has its output values confined within 0 and 1; therefore, it is mostly used in performing classification-based tasks.
- is often limited by the fact that the parameters optimized are stuck to local minimum.
- has an advantage in the fact that it provides the probability.
- has the following expression

$$f(x) = \frac{1}{1 + e^{-\beta x}}$$

An example is given in the following script:

```
from __future__ import print_function
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.init as init
import torch.nn.functional as F
from torch.autograd import Variable

# torch.nn: Neural networks can be constructed using the torch.nn package.

x = Variable(torch.randn(100,10))
y = Variable(torch.randn(100,30))

func = nn.Sigmoid()
output_x = func(x)
output_y = func(y)
print('Output size : ', output_x.size())
print('Output size : ', output_y.size())
print(x[0])
print(output_x[0])
print(y[0])
print(output_y[0])
```

7.1.3 Hyperbolic tangent function

A **hyperbolic tangent function**

- is a variant of a transformation function.
- is used to transform information from the mapping layer to the hidden layer.
- is typically used between the hidden layers of a neural network model.
- ranges between -1 and 1.
- has the following mathematical expression

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

An example is given in the following script:

```
from __future__ import print_function
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.init as init
import torch.nn.functional as F
from torch.autograd import Variable

# torch.nn: Neural networks can be constructed using the torch.nn package.

x = Variable(torch.randn(100,10))
y = Variable(torch.randn(100,30))

func = nn.Tanh()
output_x = func(x)
output_y = func(y)
print('Output size : ', output_x.size())
print('Output size : ', output_y.size())

print(x[0])
print(output_x[0])
print(y[0])
print(output_y[0])
```

7.1.4 Log sigmoid transfer function

The **log sigmoid transfer function**

- is used in mapping the input layer to the hidden layer.
- is often used when the input dataset has a large number of outliers.
- has the following mathematical expression

$$f(x) = \log\left(\frac{1}{1+e^{-\beta x}}\right)$$

An example is given in the following script:

```
from __future__ import print_function
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.init as init
import torch.nn.functional as F
from torch.autograd import Variable

# torch.nn: Neural networks can be constructed using the torch.nn package.

x = Variable(torch.randn(100,10))
y = Variable(torch.randn(100,30))

func = nn.LogSigmoid()
output_x = func(x)
output_y = func(y)
print('Output size : ', output_x.size())
print('Output size : ', output_y.size())

print(x[0])
print(output_x[0])
print(y[0])
print(output_y[0])
```

7.1.5 ReLU Function

The **rectified linear unit (ReLU)**

- is an activation function.
- is used in transferring information from the input layer to the output layer.
- is mostly used in a convolutional neural network model.
- ranges from 0 to infinity
- is mostly used between different hidden layers in a neural network model.

An example is given in the following script:

```
from __future__ import print_function
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.init as init
import torch.nn.functional as F
from torch.autograd import Variable

# torch.nn: Neural networks can be constructed using the torch.nn package.

x = Variable(torch.randn(100,10))
y = Variable(torch.randn(100,30))

func = nn.ReLU()
output_x = func(x)
output_y = func(y)
print('Output size : ', output_x.size())
print('Output size : ', output_y.size())

print(x[0])
print(output_x[0])
print(y[0])
print(output_y[0])
```

7.1.6 Leaky ReLU

The **leaky ReLU**

- is proposed to avoid the common issue of vanishing gradient present in neural networks.
- allows a small and nonzero gradient even when the unit is not active.

An example is given in the following script:

```
from __future__ import print_function
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.init as init
import torch.nn.functional as F
from torch.autograd import Variable

# torch.nn: Neural networks can be constructed using the torch.nn package.

x = Variable(torch.randn(100,10))
y = Variable(torch.randn(100,30))

func = nn.LeakyReLU()
```

```
output_x = func(x)
output_y = func(y)
print('Output size : ', output_x.size())
print('Output size : ', output_y.size())

print(x[0])
print(output_x[0])
print(y[0])
print(output_y[0])
```

7.1.7 Visualizing the activation functions

```
import torch.nn.functional as F
from torch.autograd import Variable
import matplotlib.pyplot as plt
%matplotlib inline

x = torch.linspace(-10,10, 1500)
x = Variable(x)
x_1 = x.data.numpy() # transforming into numpy

y_relu = F.relu(x).data.numpy()
y_sigmoid = torch.sigmoid(x).data.numpy()
y_tanh = torch.tanh(x).data.numpy()
y_softplus = F.softplus(x).data.numpy()

plt.figure(figsize=(7,4))
plt.plot(x_1, y_relu, c='blue', label='ReLU')
plt.ylim((-1,11))
plt.legend(loc='best')

plt.figure(figsize=(7,4))
plt.plot(x_1, y_sigmoid, c='blue', label='sigmoid')
plt.ylim((-0.2,1.2))
plt.legend(loc='best')

plt.figure(figsize=(7,4))
plt.plot(x_1, y_tanh, c='blue', label='tanh')
plt.ylim((-1.2,1.2))
plt.legend(loc='best')

plt.figure(figsize=(7,4))
plt.plot(x_1, y_softplus, c='blue', label='softplus')
plt.ylim((-0.2,11))
plt.legend(loc='best')
```

7.2 Basic neural network model

How do we build a basic neural network model using PyTorch?

A basic neural network model in PyTorch requires six steps:

- preparing training data
- initializing weights
- creating a basic network model
- calculating the loss function
- selecting the learning rate
- optimizing the loss function with respect to the model's parameters

An example is given in the following script:

```
from __future__ import print_function
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.init as init
import torch.nn.functional as F
from torch.autograd import Variable

import matplotlib.pyplot as plt
%matplotlib inline

def prep_data():
  train_X = np.asarray([13.3,14.4,15.5,16.71,16.93,14.168,19.779,16.182,
                        17.59,12.167,17.042,10.791,15.313,17.997,15.654,
                        19.27,13.1])

  train_Y = np.asarray([11.7,12.76,12.09,13.19,11.694,11.573,13.366,12.596,
                        12.53,11.221,12.827,13.465,11.65,12.904,12.42,12.94,
                        11.3])
  dtype = torch.FloatTensor
  X = Variable(torch.from_numpy(train_X).type(dtype),
               requires_grad=False).view(17,1)
  y = Variable(torch.from_numpy(train_Y).type(dtype),
               requires_grad=False)
  return X,y

def set_weights():
  w = Variable(torch.randn(1), requires_grad = True)
  b = Variable(torch.randn(1), requires_grad = True)
  return w,b

# deploy customized neural network model
```

```
def build_network(x):
  y_pred = torch.matmul(x,w)+b
  return y_pred

# implement in PyTorch
import torch.nn as nn
f = nn.Linear(17,1) # much simpler
print(f)

def loss_calc(y,y_pred):
  loss = (y_pred - y).pow(2).sum()
  for param in [w,b]:
    if param.grad is not None: param.grad.data.zero_()
  loss.backward()
  return loss.data # loss.data[0]

# optimizing results
def optimize(learning_rate):
  w.data -= learning_rate * w.grad.data
  b.data -= learning_rate * b.grad.data

learning_rate = 1e-4

x,y = prep_data() # x: training data, y: target variables
w,b = set_weights() # w,b: parameters
for i in range(5000):
  y_pred = build_network(x) # function that computes wx + b
  loss = loss_calc(y,y_pred) # error calculation
  if i % 1000 == 0:
    print(loss)
  optimize(learning_rate) # minimize the loss w.r.t. w, b.

x_numpy = x.data.numpy()
y_numpy = y.data.numpy()
y_pred = y_pred.data.numpy()
plt.plot(x_numpy,y_numpy,'o')
plt.plot(x_numpy,y_pred,'-')
```

7.3 Tensor differentiation

What is tensor differentiation, and how is it relevant in computational graph execution using the PyTorch framework?

The computational graph network is represented by **nodes** and **edges**. The **edges** represent functions. The **nodes** can be of two types: *dependent* and *independent*. The *dependent nodes* await for results from other nodes as their inputs for a posterior processing. The *independent nodes* are either input tensors or the results from some functions.

Tensor differentiation can be performed very efficiently in PyTorch because it can be done in parallel nodes. **Autograd** is the function that helps perform tensor differentiation. To

apply tensor differentiation, the **nn.backward()** method needs to be called.
An example is given in the following script:

```
from __future__ import print_function
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.init as init
import torch.nn.functional as F
from torch.autograd import Variable

import matplotlib.pyplot as plt
%matplotlib inline

x = Variable(torch.ones(4,4) * 12.5, requires_grad=True)
print(x)

fn = 2 * (x * x) + 5 * x + 6
fn.backward(torch.ones(4,4))
print(x.grad)
```

8 Deep neural network

8.1 Setting up a loss function

How do we set up a loss function and optimize it? Choosing the right loss function increases the chances of model convergence.
An example is given in the following script:

```
import torch
print(torch.__version__)
print(torch.tensor)
t_c = torch.tensor([0.5,14.0,15.0,28.0,11.0,8.0,3.0,-4.0,6.0,13.0,21.0])
t_u = torch.tensor([35.7,55.9,58.2,81.9,56.3,48.9,33.9,21.8,48.4,60.4,68.4])
print(t_c)
print(t_u)

def model(t_u,w,b):
  return w*t_u + b

def loss_fn(t_p, t_c):
  squared_diffs = (t_p - t_c)**2
  return squared_diffs.mean()

w = torch.ones(1)
b = torch.zeros(1)
```

```
t_p = model(t_u, w, b)
print(t_p)

loss = loss_fn(t_p, t_c)
print(loss)

delta = 0.1

# derivative approximate by ((x+deltax) - (x-deltax))/ (2*deltax)
loss_rate_of_change_w = \
  (loss_fn(model(t_u,w+delta,b), t_c) - loss_fn(model(t_u,w-delta,b),t_c)) \
  / (2.0 * delta)

loss_rate_of_change_b = \
  (loss_fn(model(t_u,w,b+delta), t_c) - loss_fn(model(t_u,w,b-delta),t_c)) \
  / (2.0 * delta)

learning_rate = 1e-2

w -= learning_rate * loss_rate_of_change_w

b -= learning_rate * loss_rate_of_change_b
```

Now, we can optimize the model parameters using PyTorch as follows

```
from torch import nn
loss = nn.MSELoss()
input = torch.randn(10,5, requires_grad=True)
traget = torch.randn(10,5)
output = loss(input, target)
output.backward()
print(output)
print(output.grad_fn)
```

8.2 Estimating the derivative of the loss function

How do we estimate the derivative of a loss function?
An example is given in the following script:

```
t_c = torch.tensor([0.5,14.0,15.0,28.0,11.0,8.0,3.0,-4.0,6.0,13.0,21.0])
t_u = torch.tensor([35.7,55.9,58.2,81.9,56.3,48.9,33.9,21.8,48.4,60.4,68.4])
t_un = 0.1*t_u

def dloss_fn(t_p, t_c):
  dsq_diffs = 2 * (t_p - t_c)
  return dsq_diffs

def model(t_u, w, b):
  return w * t_u + b
```

```
def dmodel_dw(t_u, w, b):
  return t_u

def dmodel_db(t_u, w, b):
  return 1.0

def grad_fn(t_u, t_c, t_p, w, b):
  dloss_dw = dloss_fn(t_p,t_c) * dmodel_dw(t_u, w, b)
  dloss_db = dloss_fn(t_p,t_c) * dmodel_db(t_u, w, b)
  return torch.stack([dloss_dw.mean(), dloss_db.mean()])

params = torch.tensor([1.0, 0.0])

nepochs = 100

learning_rate = 1e-4

for epoch in range(nepochs):
  # forward pass
  w, b = params
  t_p = model(t_un, w, b)

  loss = loss_fn(t_p, t_c)
  print('Epoch %d, Loss %f' % (epoch, float(loss)))

  # backward pass
  grad = grad_fn(t_u, t_c, t_p, w, b)

  print('Params: ', params)
  print('Grad: ', grad)

  params -= params - learning_rate * grad

print(params)
```

From the above script, you can change the *learning rate* and the *total number of epochs* to see the quality of the obtained results.

8.3 Fine tuning a model

How do we find the gradients of the loss function by applygin an optimization function to optimize the loss function?

We do so by using the **backward()** function.
See the following example:
An example is given in the following script:

```
loss.backward()
params.grad
```

The **backward()** function calculates the gradients of a function with respect to its parameters. On the other hand, we need to reset the parameter grid. If we do not reset the parameters in an existing session, the error values accumulated from any other session become mixed.

```
t_c = torch.tensor([0.5,14.0,15.0,28.0,11.0,8.0,3.0,-4.0,6.0,13.0,21.0])
t_u = torch.tensor([35.7,55.9,58.2,81.9,56.3,48.9,33.9,21.8,48.4,60.4,68.4])
t_un = 0.1*t_u

if params.grad is not None:
  params.grad.zero_()

def model(t_u, w, b):
  return w * t_u + b

def loss_fn(t_p, t_c):
  sq_diffs = (t_p - t_c)**2
  return sq_diffs.mean()

params = torch.tensor([1.0, 0.0], requires_grad=True)
nepochs = 5000
learning_rate = 1e-2

for epoch in range(nepochs):
  # forward pass
  t_p = model(t_un, *params)
  loss = loss_fn(t_p, t_c)

  print('Epoch %d, Loss %f' % (epoch, float(loss)))

  # backward pass
  if params.grad is not None:
    params.grad.zero_()

  loss.backward()

  # params.grad.clamp_(-1.0, 1.0)
  print(params, params.grad)

  params = (params - learning_rate * params.grad).detach().requires_grad_()

print(params)
```

8.4 Selecting an optimization function

How dow we optimize the gradients?

There exist a number of functions that are embedded in PyTorch. Let us look at them.

```
import torch.optim as optim

dir(optim)
```

- The **Adam** optimizer
  - is a first-order, gradient-based optimization of stochastic objective functions.
  - is based on adaptive estimation of lower-order moments.
  - is computationally efficient for a deployment on large datasets.
- **Adadelta** is another optimizer that is fast enough to work on large datasets. It does not require manual fine-tuning of the learning rate; the algorithm takes care of it internally.

An example for the SGD with a small learning rate value:

```
import torch.optim as optim
t_c = torch.tensor([0.5,14.0,15.0,28.0,11.0,8.0,3.0,-4.0,6.0,13.0,21.0])
t_u = torch.tensor([35.7,55.9,58.2,81.9,56.3,48.9,33.9,21.8,48.4,60.4,68.4])
t_un = 0.1*t_u

params = torch.tensor([1.0,0.0], requires_grad=True)
learning_rate = 1e-5
optimizer = optim.SGD([params], lr=learning_rate)
t_p = model(t_u, *params)
loss = loss_fn(t_p, t_c)
loss.backward()
optimizer.step()
print(params)
```

An example for the SGD with a larger learning rate value:

```
import torch.optim as optim
params = torch.tensor([1.0,0.0], requires_grad=True)
learning_rate = 1e-2
optimizer = optim.SGD([params], lr=learning_rate)
t_p = model(t_un, *params)
loss = loss_fn(t_p, t_c)
loss.backward()
optimizer.step()
print(params)
```

You will notice that the updated parameter values are further from the initial parameter values for the second case than that of the first case.

Let us call now the model and the loss function along with the optimization function.

```
import torch.optim as optim

t_c = torch.tensor([0.5,14.0,15.0,28.0,11.0,8.0,3.0,-4.0,6.0,13.0,21.0])
t_u = torch.tensor([35.7,55.9,58.2,81.9,56.3,48.9,33.9,21.8,48.4,60.4,68.4])
t_un = 0.1*t_u

def model(t_u, w, b):
  return w * t_u + b

def loss_fn(t_p, t_c):
  squared_diffs = (t_p - t_c)**2
  return squared_diffs.mean()

params = torch.tensor([1.0,0.0], requires_grad=True)
nepochs = 5000
learning_rate = 1e-2
optimizer = optim.SGD([params], lr=learning_rate)

for epoch in range(nepochs):
  # forward pass
  t_p = model(t_un, *params)
  loss = loss_fn(t_p, t_c)

  print('Epoch %d, Loss %f' % (epoch, float(loss)))

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

t_p = model(t_un, *params)
print(params)
```

Now, instead of using sigmoid activation function, let us use Adam activation function.

```
import torch.optim as optim

t_c = torch.tensor([0.5,14.0,15.0,28.0,11.0,8.0,3.0,-4.0,6.0,13.0,21.0])
t_u = torch.tensor([35.7,55.9,58.2,81.9,56.3,48.9,33.9,21.8,48.4,60.4,68.4])
t_un = 0.1*t_u

def model(t_u, w, b):
  return w * t_u + b

def loss_fn(t_p, t_c):
  squared_diffs = (t_p - t_c)**2
  return squared_diffs.mean()

```

```
params = torch.tensor([1.0,0.0], requires_grad=True)
nepochs = 5000
learning_rate = 1e-1
optimizer = optim.Adam([params], lr=learning_rate)

for epoch in range(nepochs):
  # forward pass
  t_p = model(t_un, *params)
  loss = loss_fn(t_p, t_c)

  print('Epoch %d, Loss %f' % (epoch, float(loss)))

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

t_p = model(t_un, *params)
print(params)
```

Let us visualize the sample data in graphical form using the actual and predicted tensors

```
from matplotlib import pyplot as plt
%matplotlib inline
plt.plot(0.1 * t_u.numpy(), t_p.detach().numpy())
plt.plot(0.1 * t_u.numpy(), t_c.numpy(), 'o')
```

Then, you should get the following results

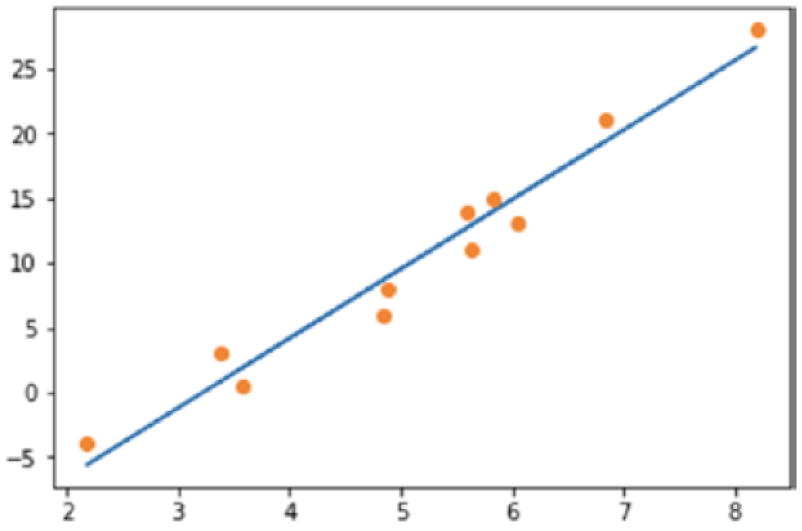


Figure 7: Optimization results

8.5 Further optimizing the function

How do we optimize the training set and test it with a validation set using random samples? An example is shown in the following script:

```
import torch.optim as optim

t_c = torch.tensor([0.5,14.0,15.0,28.0,11.0,8.0,3.0,-4.0,6.0,13.0,21.0])
t_u = torch.tensor([35.7,55.9,58.2,81.9,56.3,48.9,33.9,21.8,48.4,60.4,68.4])
t_un = 0.1*t_u

n_samples = t_u.shape[0]
n_val = int(0.2 * n_samples)

shuffled_indices = torch.randperm(n_samples)

train_indices = shuffled_indices[:-n_val]
val_indices = shuffled_indices[-n_val:]

print(train_indices, val_indices)

t_u_train = t_u[train_indices]
t_c_train = t_c[train_indices]

t_u_val = t_u[val_indices]
t_c_val = t_c[val_indices]

def model(t_u, w, b):
  return w * t_u + b

def loss_fn(t_p, t_c):
  squared_diffs = (t_p - t_c)**2
  return squared_diffs.mean()

params = torch.tensor([1.0,0.0], requires_grad=True)
nepochs = 5000
learning_rate = 1e-2
optimizer = optim.SGD([params], lr=learning_rate)

t_un_val = 0.1 * t_u_val
t_un_train = 0.1 * t_u_train

for epoch in range(nepochs):
  # forward pass
  t_p_train = model(t_un_train, *params)
  loss_train = loss_fn(t_p_train, t_c_train)

  # for validation we do not need backpropagation
  # doing in this way we reduce memory consumption.
```

```
    with torch.no_grad():
      t_p_val = model(t_un_val, *params)
      loss_val = loss_fn(t_p_val, t_c_val)

    print('Epoch %d, Train loss %f, Validation loss %f'
          % (epoch, float(loss_train), float(loss_val)))

    # backward pass
    optimizer.zero_grad()
    loss_train.backward()
    optimizer.step()

t_p = model(t_un, *params)
print(params)
```

9 Implementing a convolutional neural network (CNN)

How do we implement a convolutional neural network (CNN) using PyTorch?

Let us consider a dataset available on *torchvision* (MNIST) and build a CNN model as follows:

- AS a first step, we set up the hyperparameters.
- The second step is to set up the architecture.
- The last step is to train the model and make predictions.

Before you proceed to write and execute the following script, make sure that you have installed the **torchvision** module. If not, install it by opening a terminal and typing the following command

```
conda install torchvision -c pytorch
```

or

```
pip install torchvision
```

Then, restart the kernel from your Jupyter notebook.

```
import torch
import torch.nn as nn
from torch.autograd import Variable
import torch.utils.data as Data
import torchvision
import matplotlib.pyplot as plt
%matplotlib inline

torch.manual_seed(1) # reproducible

# Hyper parameters
# train the input data n times, to save time, we just train 1 epoch
EPOCH = 1
```

```
# 50 samples at a time to pass through the epoch
BATCH_SIZE = 50
# learning rate
LR = 0.001
# set to false if you have downloaded
DOWNLOAD_MNIST = True

# MNIST digits dataset
train_data = torchvision.datasets.MNIST(
  root='./mnist/',
  # this is training data
  train=True,
  # torch.FloatTensor of shape (Color x Height x Width) and
  # normalize in the range [0.0, 1.0]
  transform = torchvision.transforms.ToTensor(),
  # download it if you don't have it
  download = DOWNLOAD_MNIST,
)

# plot one example
print(train_data.train_data.size())   # (60000, 28, 28)
print(train_data.train_labels.size()) # (60000)
plt.imshow(train_data.train_data[0].numpy(), cmap='gray')
plt.title('%i' % train_data.train_labels[0])
plt.show()

# data loader for easy mini-batch return in training,
# the image batch shape will be (50, 1, 28, 28)
train_loader = Data.DataLoader(dataset=train_data,batch_size=BATCH_SIZE,shuffle=True)

# convert test data into Variable, pick 2000 samples to speed up testing
test_data = torchvision.datasets.MNIST(root='./mnist/', train=False)
test_x = Variable(torch.unsqueeze(test_data.test_data, dim=1)).type(torch.FloatTensor)
#shape from (2000, 28, 28) to (2000,1,28,28), value in range(0,1)
test_y = test_data.test_labels[:2000]

class CNN(nn.Module):
  def __init__(self):
    super(CNN, self).__init__()
    self.conv1 = nn.Sequential( # input shape (1,28,28)
      nn.Conv2d(
        in_channels = 1,
        out_channels = 16,
        kernel_size = 5,
        stride = 1,
        padding = 2
      ),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=2),
    )
    self.conv2 = nn.Sequential(
      nn.Conv2d(16, 32, 5, 1, 2),
      nn.ReLU(),
      nn.MaxPool2d(2)
    )
    self.out = nn.Linear(32 * 7 * 7, 10)

  def forward(self, x):
    x = self.conv1(x)
    x = self.conv2(x)
    x = x.view(x.size(0), -1)
    output = self.out(x)
    return output, x

cnn = CNN()

optimizer = torch.optim.Adam(cnn.parameters(), lr=LR)
loss_func = nn.CrossEntropyLoss()

from matplotlib import cm
try: from sklearn.manifold import TSNE; HAS_SK = True
except: HAS_SK = False; print('Please install sklearn')

def plot_with_labels(lowDWeights, labels):
  plt.cla()
  X, Y = lowDWeights[:,0], lowDWeights[:,1]
  for x, y, s in zip(X,Y,labels):
    c = cm.rainbow(int(255 * s /9));
    plt.text(x,y,s,backgroundcolor=c, fontsize=9)
  plt.xlim(X.min(), X.max());
  plt.ylim(Y.min(), Y.max());
  plt.title('Visualize last layer')

plt.ion()

for epoch in range(EPOCH):
  for step, (x,y) in enumerate(train_loader):
      # gives batch data, normalize x when iterate train_loader
      b_x = Variable(x) # batch x
      b_y = Variable(y) # batch y

      output = cnn(b_x)[0]
      loss = loss_func(output, b_y)
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()
      if step % 100 == 0:
        test_output, last_layer = cnn(test_x)
        pred_y = torch.max(test_output, 1)[1].data.squeeze()
        accuracy = (pred_y == test_y).sum().item() / float(test_y.size(0))
        print('Epoch: ',epoch, '| train_loss: %.4f' % loss.data,
              '| test accuracy: %.2f' % accuracy)
        #print('Epoch: ',epoch, '| train_loss: %.4f' % loss.data[0],
        #       '| test accuracy: %.2f' % accuracy)

        if HAS_SK:
          # visualization of trained flatten layer (T-SNE)
          tsne = TSNE(perplexity=30, n_components=2, init='pca', n_iter=5000)
          plot_only = 500
          low_dim_embs = tsne.fit_transform(last_layer.data.numpy()[:plot_only, :])
          labels = test_y.numpy()[:plot_only]
          plot_with_labels(low_dim_embs,labels)

plt.ioff()

# print 10 predictions from test data
test_output, _ = cnn(test_x[:10])
pred_y = torch.max(test_output,1)[1].data.numpy().squeeze()
print(pred_y, 'prediction number')
print(test_y[:10].numpy(), 'real number')
```