Chapter 7

Introduction to PyTorch

So far we have studied the fundamentals of neural networks. We implemented and trained the neural network models in Python from scratch. To make a solid understand of the fundamentals, it is essential for us to be able to implement an algorithm from scratch (without relying on package tools). In practice, however, it is very time consuming and error-prone to build everything from scratch when the project (e.g. deep learning) is typically sophisticated in the real world. Fortunately, there are some frameworks or package tools available to accelerate the development of machine learning products, such as Scikit-learn, Tensorflow, and PyTorch. These packages include functions/classes to abstract commonly used modules or functions from designer's codes. We just need to call those functions/classes instead of writing the codes ourselves, so that we can concentrate on the overall project.

Before we proceed to the topic of deep learning, we introduce PyTorch which you will use to develop deep learning project in subsequent chapters. Appendix A gives a tutorial of Jupyter Notebook and PyTorch installation. In this chapter, you will learn:

- The framework of PyTorch
- Basics of tensors in PyTorch
- Data representation in tenors
- Autograd and optimizers in PyTorch
- Example of linear regression using PyTorch
- Example of neural network for image classification using Pytorch

7.1 Why PyTorch?

Deep learning allows us to carry out a very wide range of complicated tasks, like speech recognition, playing strategy games (e.g. Alpha Go), or identifying objects in cluttered scenes. In practice, we need tools that are flexible, so that they can be adapted to such a wide range of problems, and efficient to allow training to occur over large amounts of data in reasonable times; and we need the trained model to perform correctly in the presence of variability in the inputs.

PyTorch is a framework for Python programs that facilitates building deep learning projects. PyTorch's clear syntax, streamlined API, and easy debugging make it an excellent choice for implementing deep learning projects. PyTorch has been proven to be fully qualified for use in professional contexts for real-world, high-profile work. PyTorch provides a core data structure, the *tensor*, which is a multidimensional array that shares many similarities with NumPy arrays. Compared to NumPy arrays, PyTorch tensors have a few superpowers, such as the ability to perform very fast operations on graphical processing units (GPUs), distribute operations on multiple devices or machines, and keep track of the graph of computations. These are all important features when implementing a modern deep learning framework.

PyTorch offers two things that make it particularly relevant for deep learning. First, it provides accelerated computation using graphical processing units (GPUs), often yielding speedups in the range of 50x over doing the same calculation on a CPU. Second, PyTorch provides facilities that support numerical optimization on generic mathematical expressions, which deep learning uses for training. Note that both features are useful for scientific computing in general, not exclusively for deep learning. In fact, PyTorch can be viewed as a high-performance library with optimization support for scientific computing in Python.

Fig.7.1 shows how PyTorch supports a deep learning project. The diagram consists of three layers: physical layer, Python layer, and PyTorch layer. The physical layer is the hardware platform on which the project gets trained and deployed. In our context, we only need to pay attention to Python layer and PyTorch layer. To train a neural network, first we need to physically get the data, most often from some sort of storage as the data source. Then we need to convert each sample from our data into a tensor. The tensors are usually assembled into batches for mini-batch process. PyTorch provides classes *Dataset* and *DataLoader* in *torch.utils.data* package for this purpose. With a selected (untrained) model and batch tensors, a training loop will be implemented in CPUs or GPUs to fit the model, i.e., to minimize the defined loss function. PyTorch packages, torch.optim and torch.nn, provide various classes to support auto-computation of gradients, optimization and construction of neural network layers.

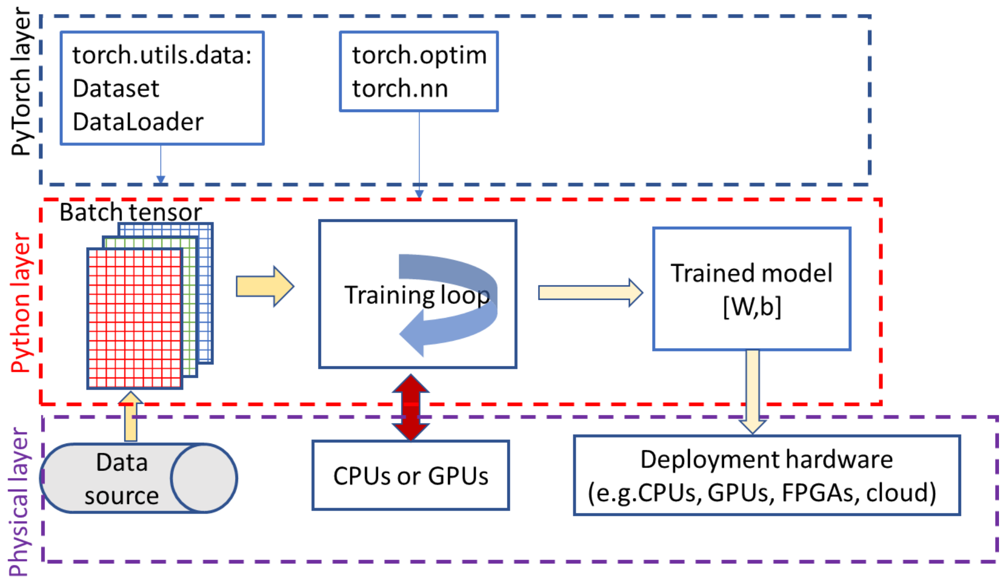


Fig.7.1 PyTorch framework for deep learning

7.2 Tensors

7.2.1 Tensor: multidimensional array

Like arrays in NumPy, tensors are the fundamental data structure in PyTorch. A tensor is an array: that is, a data structure that stores a collection of numbers that are accessible individually using an index, and that can be indexed with multiple indices. PyTorch provides many functions for operating on these tensors. Behind the scenes, tensors can keep track of a computational graph and gradients, PyTorch tensors can be converted to NumPy arrays and vice versa very efficiently. Tensors can be understood as the generalization of vectors and matrices to an arbitrary number of dimensions, illustrated in Fig.7.2.

$$\begin{bmatrix}0.5\\0.4\\0.7\end{bmatrix} \qquad \begin{bmatrix}0.5 & 0.3\\0.4 & 0.1\\0.7 & 0.4\end{bmatrix} \qquad \begin{bmatrix}\begin{bmatrix}0.5 & 0.3\\0.4 & 0.1\\0.7 & 0.4\end{bmatrix}, \begin{bmatrix}0.7 & 0.9\\0.4 & 0.3\\0.2 & 0.1\end{bmatrix}, \begin{bmatrix}0.2 & 0.1\\0.6 & 0.7\\0.9 & 0.5\end{bmatrix}\end{bmatrix}$$

1D tensor, size (3)   2D tensor, size (3,2)   3D tensor, size (3,3,2)

Fig.7.2 Tensor data structure

7.2.2 Indexing and operations on tensors

We will encounter some frequently used tensor operations as we proceed with the book. The complete description of all operations associated with tensors can be found online (https://pytorch.org/docs/stable/index.html). In this section, we will demonstrate some basic operations on tensors by examples through Jupyter Notebook.

1) Create a tensor

We can create a tensor in different ways: from Numpy array, list, random numbers, filling 0s or 1s, a range, linear or log scale space, and byte.

Summary of functions for creating tensors

Function	**Description**	**Example**
eye	Identity matrix	torch.eye(3)
from_numpy	Convert Numpy array to tensor	torch.from_numpy(a)
linspace	Linear space vector	torch.linspace(1, 10, steps=10)
logspace	Log scale space vector	torch.logspace(start=-10, end=10, steps=5)
ones	Filling with ones	torch.ones(2, 1, 2, 1)
ones_like	Filling with ones	torch.ones_like(eye)
arange	Return a 1-D tensor	torch.arange(1, 2.5, 0.5)
zeros	Filling with zeros	torch.zeros(2, 3)
zeros_like	Filling with zeros for a tensor shape	torch.zeros_like(input)
randn	Filling with random numbers	torch.randn(3,2)

```
import torch
import numpy as np

# creating tensors
# from numpy array
v_np = np.zeros([3,4])
v_tensor = torch.tensor(v_np)
v_tensor
```

```
       tensor([[0., 0., 0., 0.],
               [0., 0., 0., 0.],
               [0., 0., 0., 0.]], dtype=torch.float64)
```

```
# create tensor from list
v = torch.tensor([2,3])      # a tensor initialized with a list, int64
v = torch.Tensor([2,3])      # a tensor initialized with a list, float32
# create tensor from random numbers
torch.manual_seed(1)
points=torch.randn(3,2)   # standard normal distribution
print(points)
points.shape
```

```
       tensor([[ 0.6614,  0.2669],

               [ 0.0617,  0.6213],

               [-0.4519, -0.1661]])


       torch.Size([3, 2])
```

```
# create by filling with 1 or 0
eye = torch.eye(3)             # Create an identity 3x3 tensor
v = torch.ones(10)             # A tensor of size 10 containing all ones
v = torch.ones(2, 1, 2, 1)     # fill with 1, Size 2x1x2x1
v = torch.ones_like(eye)       # A tensor with same shape as eye.Fill it with 1.
v = torch.zeros(10)            # A tensor of size 10 containing all zeros

# create by arange
v = torch.arange(9)
print(v)
v = v.view(3, 3)
print(v)
       tensor([0, 1, 2, 3, 4, 5, 6, 7, 8])
       tensor([[0, 1, 2],
               [3, 4, 5],
               [6, 7, 8]])

v = torch.linspace(1, 10, steps=10)
print(v)
# Create a Tensor with 10 linear points for (1, 10) inclusively
v = torch.logspace(start=-10, end=10, steps=5)
# Size 5: 1.0e-10 1.0e-05 1.0e+00, 1.0e+05, 1.0e+10
print(v)
       tensor([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
       tensor([1.0000e-10, 1.0000e-05, 1.0000e+00, 1.0000e+05, 1.0000e+10])

c = torch.ByteTensor([0, 1, 1, 0])
print(c)
       tensor([0, 1, 1, 0], dtype=torch.uint8)
```

The objects within a tensor must all be numbers of the same type, and PyTorch must keep track of this numeric type. The *dtype* argument to tensor constructors (e.g. *tensor, zeros, ones*) specifies the numerical data (d) type that will be contained in the tensor. Here’s a list of the possible values for the dtype argument:

- torch.float32 or torch.float: 32-bit floating-point
- torch.float64 or torch.double: 64-bit, double-precision floating-point
- torch.float16 or torch.half: 16-bit, half-precision floating-point
- torch.int8: signed 8-bit integers
- torch.uint8: unsigned 8-bit integers
- torch.int16 or torch.short: signed 16-bit integers
- torch.int32 or torch.int: signed 32-bit integers
- torch.int64 or torch.long: signed 64-bit integers
- torch.bool: Boolean

2) Indexing, Slicing, Joining, Mutating Ops

The commonly used functions for indexing, slicing, joining, and mutating are summarized in the table below.

Function	**Description**	**Example**
torch.cat	Concatenation along a specified dimension	y=torch.cat([v,v,v,v],1)
x.view	Re-organize the tensor shape	v.view(3,3)
torch.stack	Add a dimension by stacking	y=torch.stack((v,v))
torch.gather	Gather values along an axis specified by dim.	t = torch.tensor([[1, 2], [3, 4]]) torch.gather(t, 0, torch.tensor([[0, 0], [1, 0]]))
torch.transpose	Transpose a tensor	y = torch.transpose(x, 0, 1)
torch.squeeze	Remove dimensions of size one	y = torch.squeeze(x)
torch.unsqueeze	Add a dimension with size one	y=torch.unsqueeze(x, 0)
torch.chunk	torch.chunk(input, chunks, dim=0) → List of Tensors: Attempts to split a tensor into the specified number of chunks.	y=torch.chunk(x,3,1)
torch.split	torch.split(tensor, split_size, dim=0)	y= torch.split(x, [1,2], 1) # [1,2] is the size of each section
torch.index_select	torch.index_select(input, dim, index, *, out=None) → Tensor	indices = torch.tensor([0, 2]) torch.index_select(x, 1, indices)

	Returns a new tensor which indexes the input tensor along dimension dim using the entries in index	

torch.masked_select	torch.masked_select(input, mask, *, out=None) → Tensor
	Returns a new 1-D tensor which indexes the input tensor according to the boolean mask mask which is a BoolTensor
		mask = x.ge(0)
		y=torch.masked_select(x, mask)

torch.nonzero	returns a 2-D tensor where each row is the index for a nonzero value	y=torch.nonzero(v)
torch.take	torch.take(input, index) → Tensor Returns a new tensor with the elements of input at the given indices.	y=torch.take(x, torch.tensor([0, 2, 5]))

The continued part of Jupyter notebook shows some examples.

```
# indexing
a=points[:, -1]
print(a)
a.shape

       tensor([ 0.2669,  0.6213, -0.1661])

       torch.Size([3])

# re-organize: view, cat, stack, gather, squeeze, unsqueeze

v=torch.arange(9)
v=v.view(3,3)
v
       tensor([[0, 1, 2],
               [3, 4, 5],
               [6, 7, 8]])

y=torch.cat([v,v,v,v],1)
# concatenation along with 1 (column) dimension or 0 (row) dimension
#y.shape
y

       tensor([[0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2],
               [3, 4, 5, 3, 4, 5, 3, 4, 5, 3, 4, 5],
               [6, 7, 8, 6, 7, 8, 6, 7, 8, 6, 7, 8]])

y=torch.stack((v,v))  # add one more dimension
y.shape

       torch.Size([2, 3, 3])

t = torch.tensor([[1, 2], [3, 4]])
torch.gather(t, 1, torch.tensor([[0, 0], [1, 0]])) # gather along with column
(axis 1)

       tensor([[1, 1],
               [4, 3]])

t = torch.tensor([[1, 2], [3, 4]])
torch.gather(t, 0, torch.tensor([[0, 0], [1, 0]]))  # gather along with row
(axis 0)
```

```
       tensor([[1, 2],
               [3, 2]])

x = torch.zeros(2, 1, 2, 1, 2)
print("original", x.size())
y = torch.squeeze(x)                # dimensions with size 1 will be removed
print("after squeeze", y.size())
     original torch.Size([2, 1, 2, 1, 2])
     after squeeze torch.Size([2, 2, 2])

x = torch.tensor([1, 2, 3, 4])
print(x.shape, x)
y=torch.unsqueeze(x, 0)
# Returns a new tensor with a dimension of size one
# inserted at the specified position.
print(y.shape, y)
y=torch.unsqueeze(x, 1)
print(y.shape, y)
       torch.Size([4]) tensor([1, 2, 3, 4])
       torch.Size([1, 4]) tensor([[1, 2, 3, 4]])
       torch.Size([4, 1]) tensor([[1],
               [2],
               [3],
               [4]])

# torch.transpose(input, dim0, dim1)
# Returns a tensor that is a transposed version of input.
# The given dimensions dim0 and dim1 are swapped.
x = torch.randn(2,3)
y = torch.transpose(x, 0, 1)
print(x)
print(y)
       tensor([[-1.5228,  0.3817, -1.0276],
               [-0.5631, -0.8923, -0.0583]])
       tensor([[-1.5228, -0.5631],
               [ 0.3817, -0.8923],
               [-1.0276, -0.0583]])

y=torch.chunk(x,3,1)   # 3 is the number of chunks
y= torch.split(x, [1,2], 1) # [1,2] is the size of each section
y[1]

       tensor([[ 0.3817, -1.0276],
               [-0.8923, -0.0583]])
indices = torch.tensor([0, 2])
torch.index_select(x, 1, indices) # select along dimension 1 by indices

       tensor([[-1.5228, -1.0276],
               [-0.5631, -0.0583]])

mask = x.ge(0)
print(mask)
       tensor([[False,  True, False],
               [False, False, False]])

y=torch.masked_select(x, mask)
print(y.shape, y)
       torch.Size([1]) tensor([0.3817])

y=torch.nonzero(v) # return the index for non-zero elements in v
print(y.shape, y)
       torch.Size([8, 2]) tensor([[0, 1],
                [0, 2],
                [1, 0],
                [1, 1],
                [1, 2],
                [2, 0],
                [2, 1],
                [2, 2]])

y=torch.take(x, torch.tensor([0, 2, 5]))
print(y)
       tensor([-1.5228, -1.0276, -0.0583])
```

3) Point-wise operations

Pointwise (or element-wise) operations operate on each element in the tensor in the same way simultaneously. Commonly used point-wise operations are summarized in the table below.

Function	**Description**	**Example**
abs	Absolute value	torch.abs(x)
acos	arc cosine	torch.acos(x)
add	Return: a+10*b	torch.add(a,10, b)
addcmul	t1+s*(t2*t3)	torch.addcmul(t1, t2, t3, value=s)
asin, atan, atan2	arcsine and arctangent	torch.asin(x)
ceil	Round up to the integer	torch.ceil(x)
clamp	Clamp values in a tensor	torch.clamp(x, min=-0.5, max=0.5)
cos, cosh	Cosine, hyperbolic cosine	
div	Element-wise divide	torch.div(x, x2)
erf	Compute Gaussian error function	
erfinv	Inverse of erf	
exp	Exponential function	torch.exp(x)
expm1	Computes exp(x)-1, but provides greater precision than exp(x) - 1 for small values of x.	torch.expm1(x)
floor	Round down to integer	torch.floor(x)
fmod	Remainder of division	
frac	Get fraction part	torch.frac(x)
lerp	Linear interpolation	torch.lerp(start, end, 0.5)
log	Natural log	torch.log(x)
logp1	Log(x+1)	torch.log(x)
mul	Element-wise multiply	torch.mul(x,x2)
neg	Return negative of elements	torch.neg(x)
pow	Take power of each element	torch.pow(x,x2)

reciprocal	1/x	torch.reciprocal(x)
remainder	Remainder of division	torch.remainder(x,x2)
round	Rounds elements of input to the nearest integer.	torch.round(x)
sigmoid	Sigmoid function	torch.sigmoid(x)
sign	Returns a new tensor with the signs of the elements of input. (1 for positive, 0 for 0, -1 for negative)	torch.sign(x)
sin, sinh	sine, hyperbolic sine	torch.sin(x)
sqrt	Square root	torch.sqrt(x)
tan, tanh	Tangent, hyperbolic tangent	torch.tan(x)
trunc	Returns a new tensor with the truncated integer values of the elements of input.	torch.trunc(x)

Examples are shown in the following codes. For more detailed information, one can refer to Pytorch official website (https://pytorch.org/docs/stable/).

```
# point-wise operations
y=torch.abs(x)
print(y)
tensor([[1.5228, 0.3817, 1.0276],
        [0.5631, 0.8923, 0.0583]])

x2=torch.randn(2,3)
print(x2)
tensor([[-0.1955, -0.9656,  0.4224],
        [ 0.2673, -0.4212, -0.5107]])

y=torch.add(x,10) # add 10 to all elements
print(y)
tensor([[ 8.4772, 10.3817,  8.9724],
        [ 9.4369,  9.1077,  9.9417]])

y=torch.add(x,10,x2) # x+10*x2
print(y)
tensor([[-3.4779, -9.2747,  3.1965],
        [ 2.1101, -5.1042, -5.1652]])

y=torch.clamp(x2,min=-0.5, max=0.5)
print(y)
tensor([[-0.1955, -0.5000,  0.4224],
        [ 0.2673, -0.4212, -0.5000]])

y=torch.div(x, x2)
y=torch.mul(x,x2)
y=torch.acos(x)
y=torch.ceil(x)
y=torch.pow(x,x2)
y=torch.reciprocal(x)
y=torch.sign(x)
y=torch.sqrt(x)
print(y)
tensor([[   nan, 0.6178,    nan],
        [   nan,    nan,    nan]])
```

4) Reduction operations

Reduction operations usually return a scalar or a tensor with a smaller size than input tensor. The commonly used reduction operations are summarized below.

Function	Description	Example
cumprod	Returns the cumulative product of elements of input in the dimension dim.	y=torch.cumprod(v,0)
cumsum	Returns the cumulative sum of elements of input in the dimension dim.	y=torch.cumsum(v,1)
dist	torch.dist(input, other, p=2) Returns the p-norm of (input - other)	d=torch.dist(x, x2, p=2)
mean	Returns mean	d=torch.mean(v,0, True)
median	Returns median for all or a dimension	d=torch.median(v,0)
mode	Returns mode and index	d=torch.mode(v,1)
prod	Returns product of elements	d=torch.prod(v,0)
std	Standard deviation	d=torch.std(v,0)
sum	Returns sum	d=torch.sum(v, 1)
var	Variance of all elements	d=torch.var(v,0)

Examples:

```
# Reduction operations
v=v.to(torch.float32) # some operations do not support long type,
v

tensor([[0., 1., 2.],
        [3., 4., 5.],
        [6., 7., 8.]])

y=torch.cumprod(v,0) #cummulative product along dimension 0
y=torch.cumsum(v,1)  #cummulative sum along dimension 1
d=torch.dist(v, v+3, p=2)
d=torch.mean(v,0, True) # True keep dimension
d=torch.mean(v) # mean for all elements
d=torch.sum(v,1)
d=torch.median(v,0)
d=torch.mode(v,1) # return mode and index
d=torch.prod(v,0) # product of elements
d=torch.std(v,0)  # standard deviation
d=torch.var(v,0)  # variance
print(d)
tensor([9., 9., 9.])
```

5) Comparison operation

Function	Description	Example
eq	Element-wise comparison. Return a tensor with Boolean elements (True or False)	y=torch.eq(x,x2)
equal	True if two tensors are the same	d=torch.equal(x,x2)

ge, gt	True if greater or equal, if greater, element-wise	d=torch.ge(x, x2)
kthvalue	Returns a namedtuple (values, indices) where values is the k-th smallest element of each row of the input tensor in the given dimension dim	d=torch.kthvalue(v,2, 0, True)
le, lt	True if less or equal, if less than, element-wise	d=torch.le(x,x2)
max	Returns maximal elements and indices	d=torch.max(v,0)
min	Returns minimal elements and indices	d=torch.min(v,0)
ne	True if not equal, element-wise	d=torch.ne(x,x2)
sort	Returns a sorted tensor and indices	d=torch.sort(v,0)
topk	Returns top k values along the dimension	d=torch.topk(v,2)

Examples:

```
# Comparison operations

d=torch.eq(x,x2)
d=torch.max(v,0)
d=torch.equal(v,v)
d=torch.ge(x,x2)
d=torch.gt(x,x2)
d=torch.kthvalue(v, 2, 0, True)
d=torch.ne(x,x2)
d=torch.sort(v,0)
d=torch.topk(v,2)
print(d)
torch.return_types.topk(
values=tensor([[2., 1.],
        [5., 4.],
        [8., 7.]]),
indices=tensor([[2, 1],
        [2, 1],
        [2, 1]]))
```

6) Matrix, vector multiplication

Commonly used basic functions:

Function	Description	Example
dot	Computes the dot product of two 1D tensors.	d=torch.dot(torch.tensor([1,2]), torch.tensor([3,4]))
mv	Performs a matrix-vector product of the matrix input and the vector vec	d = torch.mv(input, vec)
addmv	Performs a matrix-vector product of the matrix mat and the vector vec. The vector input is added to the final result. input+mat·vec	d = torch.addmv(input, mat, vec)
mm	Performs a matrix multiplication of the matrices input and mat2.	d=torch.mm(input, mat2)
addmm	Performs a matrix multiplication of the matrices mat1 and mat2. The matrix input is added to the final result.	d=torch.addmm(input,mat1,mat2)

bmm	Batch matrix multiplication	d = torch.bmm(batch1, batch2)
addbmm	Performs a batch matrix-matrix product of matrices stored in batch1 and batch2, with a reduced add step (all matrix multiplications get accumulated along the batch dimension). input is added to the final result.	d = torch.addbmm(M, batch1, batch2)

Examples:

```
# Matrix, vector multiplication

d=torch.dot(torch.tensor([1,2]), torch.tensor([3,4]))
mat = torch.randn(2, 4)
vec = torch.tensor([1.,2.,3.,4.])
d = torch.mv(mat, vec)
# Matrix + Matrix X vector
# Size 2
M = torch.randn(2)
mat = torch.randn(2, 3)
vec = torch.randn(3)
d = torch.addmv(M, mat, vec)
# Matrix x Matrix
# Size 2x4
mat1 = torch.randn(2, 3)
mat2 = torch.randn(3, 4)
d = torch.mm(mat1, mat2)

# Matrix + Matrix X Matrix
# Size 3x4
M = torch.randn(3, 4)
mat1 = torch.randn(3, 2)
mat2 = torch.randn(2, 4)
d = torch.addmm(M, mat1, mat2)
# Batch Matrix x Matrix
# Size 10x3x5
batch1 = torch.randn(10, 3, 4)
batch2 = torch.randn(10, 4, 5)
d = torch.bmm(batch1, batch2)
# Batch Matrix + Matrix x Matrix
# Performs a batch matrix-matrix product
# 3x2 + (5x3x4 X 5x4x2 ) -> 5x3x2
M = torch.randn(3, 2)
batch1 = torch.randn(5, 3, 4)
batch2 = torch.randn(5, 4, 2)
d = torch.addbmm(M,batch1, batch2)

print(d.shape)
torch.Size([3, 2])
```

Note that our purpose in this section is to get familiar with operations on tensors, but is not to include a complete reference for all available tensor operations. There are many other operations which are not included here. Please check PyTorch website for a complete reference.

7.3 Data Representation using Tensors

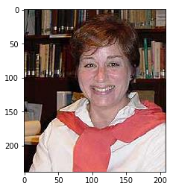
In PyTorch framework, neural networks take tensors as input and produce tensors as outputs. Furthermore, all operations within a neural network and during optimization are operations between tensors, and all parameters (for example, weights and biases) in a neural network are tensors. In this section, we will describe how to handle real-world data using tensors.

7.3.1 Images

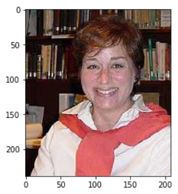
An image is represented as a collection of scalars arranged in a regular grid with a height and a width (in pixels). A grayscale image has a single scalar per pixel while a colorful image typically has three or more scalars per pixel. The three scalars for colorful images are associated with the intensity of three colors (Red, Green, Blue), and are often encoded as 8-bit integers (i.e. 0-255).

There are different formats to store images in files (e.g. *.jpg, *.gif, *.png) and different ways to load (read) image file in Python. If the image data is loaded into a NumPy array, it can be converted to PyTorch tesnor. Note that PyTorch modules dealing with image data require tensors to be laid out as C × H × W: channels, height, and width, respectively. A batch tensor of multiple images should have a dimension layout as: N × C × H × W: image, channel, height, and width. For instance, we can use the following statements to read an image file, and then load to a tensor for PyTorch modules.

```
import imageio
import matplotlib.pyplot as plt

# read to numpy array
img_np = imageio.imread('../torch_tutorial/data/faces/person.jpg')
print(img_np.shape)
(239, 209, 3)

# display the image
plt.imshow(img_np)
plt.show()
```



```
img_tensor=torch.from_numpy(img_np)
print(img_tensor.shape)
torch.Size([239, 209, 3])

# display the image
plt.imshow(img_tensor)
plt.show()
```

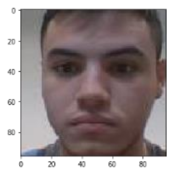


```
out=img_tensor.permute(2,0,1)   # Pytorch: Channelxheightxwidth =CxHxW
print(out.shape)
torch.Size([3, 239, 209])
```

The following codes read multiple image files in a folder, and then store them in a batch tensor with a format [batch, C, W, H].

```
batch_size = 4
batch = torch.zeros(batch_size, 3, 96, 96, dtype=torch.uint8)
```

```
import os
data_dir = '../torch_tutorial/data/student_faces/'
filenames = [name for name in os.listdir(data_dir) if os.path.splitext(name)[-1
] == '.jpg']
for i, filename in enumerate(filenames):
    img_arr = imageio.imread(os.path.join(data_dir, filename))
    img_t = torch.from_numpy(img_arr)
    img_t = img_t.permute(2, 0, 1)
    img_t = img_t[:3]
    batch[i] = img_t

print(batch.shape)
# display the image
plt.imshow(batch[1].permute(1,2,0))
plt.show()
torch.Size([4, 3, 96, 96])
```

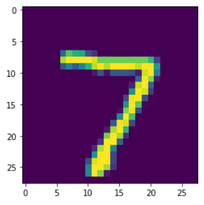


7.3.2 Excel CSV files

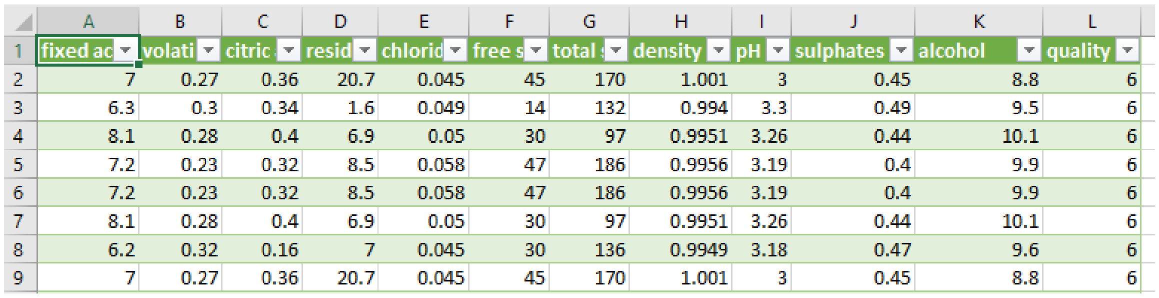
Another format for data storage is spreadsheet or CSV file. It's a table with each row corresponding to one example (or record), while each column corresponds to one feature (or attribute) or the label of the example.

We can use ***numpy.loadtxt*** to read the data from a CSV file. For example, the image examples of handwritten digits are saved as CSV files mnist_train.csv and mnist_test.csv, which are available at https://www.kaggle.com/datasets/oddrationale/mnist-in-csv. The mnist_train.csv file contains the 60,000 training examples and labels. The mnist_test.csv contains 10,000 test examples and labels. Each row consists of 785 values: the first value is the label (a number from 0 to 9) and the remaining 784 values (i.e. 28 × 28 pixels) are the pixel values (a number from 0 to 255).

```
import csv

import numpy as np

xy_path = "c:/machine_learning/mnist_test.csv"
xy_numpy = np.loadtxt(xy_path, delimiter=",")
xy_t=torch.tensor(xy_numpy)

print(xy_t.shape)
xy_t
torch.Size([10000, 785])

tensor([[7., 0., 0.,  ..., 0., 0., 0.],
        [2., 0., 0.,  ..., 0., 0., 0.],
        [1., 0., 0.,  ..., 0., 0., 0.],
        ...,
        [4., 0., 0.,  ..., 0., 0., 0.],
        [5., 0., 0.,  ..., 0., 0., 0.],
        [6., 0., 0.,  ..., 0., 0., 0.]], dtype=torch.float64)

# display the image
plt.imshow(xy_t[0][1:].view(28,28))
plt.show()
```



Some CSV files use ";" as delimiter and/or include a header line. For example, the wine quality dataset, publicly available at https://archive.ics.uci.edu/ml/datasets/wine+quality, contains a semicolon-separated collection of values organized in 12 columns preceded by a header line containing the column names. The first 11 columns contain values of chemical variables, and the last column contains the sensory quality score from 0 (very bad) to 10 (excellent), shown in Fig.7.3.



	A	B	C	D	E	F	G	H	I	J	K	L
1	fixed ac	volati	citric	resid	chlorid	free s	total	density	pH	sulphates	alcohol	quality
2	7	0.27	0.36	20.7	0.045	45	170	1.001	3	0.45	8.8	6
3	6.3	0.3	0.34	1.6	0.049	14	132	0.994	3.3	0.49	9.5	6
4	8.1	0.28	0.4	6.9	0.05	30	97	0.9951	3.26	0.44	10.1	6
5	7.2	0.23	0.32	8.5	0.058	47	186	0.9956	3.19	0.4	9.9	6
6	7.2	0.23	0.32	8.5	0.058	47	186	0.9956	3.19	0.4	9.9	6
7	8.1	0.28	0.4	6.9	0.05	30	97	0.9951	3.26	0.44	10.1	6
8	6.2	0.32	0.16	7	0.045	30	136	0.9949	3.18	0.47	9.6	6
9	7	0.27	0.36	20.7	0.045	45	170	1.001	3	0.45	8.8	6

Fig.7.3 a portion of winequality-white.csv

The following codes read the wine quality data, split the data into two parts: input features and labels, and normalize each feature. The results are stored in the corresponding tensors.

```
wine_path = "../torch_tutorial/data/winequality-white.csv"
wine_numpy = np.loadtxt(wine_path, delimiter=";", skiprows=1)
wine_tensor=torch.from_numpy(wine_numpy)
print(wine_tensor.shape)
wine_tensor
torch.Size([4898, 12])

tensor([[ 7.0000,  0.2700,  0.3600,  ...,  0.4500,  8.8000,  6.0000],
        [ 6.3000,  0.3000,  0.3400,  ...,  0.4900,  9.5000,  6.0000],
        [ 8.1000,  0.2800,  0.4000,  ...,  0.4400, 10.1000,  6.0000],
        ...,
        [ 6.5000,  0.2400,  0.1900,  ...,  0.4600,  9.4000,  6.0000],
        [ 5.5000,  0.2900,  0.3000,  ...,  0.3800, 12.8000,  7.0000],
        [ 6.0000,  0.2100,  0.3800,  ...,  0.3200, 11.8000,  6.0000]],
       dtype=torch.float64)

# split the data into input and label
input_data=wine_tensor[:,:-1]
label=wine_tensor[:, -1].long()
print(label)
tensor([6, 6, 6,  ..., 6, 7, 6])

# normalize each feature
input_data_mean=torch.mean(input_data, 0)
input_data_var=torch.var(input_data,0)
input_data_normalized = (input_data - input_data_mean) /
torch.sqrt(input_data_var)
```

We can use the following statement to read the header line to col_list that may be useful.

```
col_list =next(csv.reader(open(wine_path), delimiter=';'))

col_list

['fixed acidity',
 'volatile acidity',
 'citric acid',
 'residual sugar',
 'chlorides',
 'free sulfur dioxide',
 'total sulfur dioxide',
 'density',
 'pH',
 'sulphates',
 'alcohol',
 'quality']
```

7.3.3 Converting categorical label to one-hot label

As we discussed in previous chapters, the categorical labels in a multi-classification task are usually converted to one-hot encoded labels. Consider a 10-category classification. Ten categories are first labeled as ten integers (0,1,2,3,4,5,6,7,8,9), respectively. The one-hot label of a category is a vector of 10 elements, with all elements set to 0, but 1 at the index specified by the categorical integer label. For example, category "2" will be mapped to a one-hot label [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]. The following codes show how to convert the integer labels into one-hot coded labels.

```
# generate onehot for label tensor
label_examples=torch.tensor([0,1,2,3,4,5,6,7,8,9,3,4,9,1])
label_onehot = torch.zeros(label_examples.shape[0], 10)
label_onehot.scatter_(1, label_examples.unsqueeze(1), 1.0)
label_onehot

tensor([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
        [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
        [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

7.4 Linear Regression using PyTorch

In this section, we will implement a linear regression using PyTorch. Through the example, we will show how to utilize the PyTorch resources (e.g. autograd and optim) for machine learning. These resources will significantly reduce the efforts of development.

7.4.1 Dataset

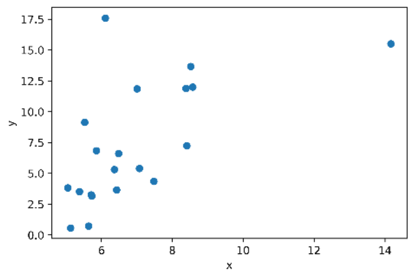
Consider a task of fitting a linear model to a dataset (X,Y). All examples are given

```
x=torch.tensor([6.1101, 5.5277, 8.5186, 7.0032, 5.8598, 8.3829, 7.4764, 8.5781, 6.4862,
5.0546, 5.7107, 14.164, 5.734, 8.4084, 5.6407, 5.3794, 6.3654, 5.1301, 6.4296, 7.0708])
y=torch.tensor([17.592,  9.1302,  13.662,  11.854,  6.8233,  11.886,  4.3483,  12,  6.5987,
3.8166,3.2522, 15.505, 3.1551, 7.2258, 0.71618, 3.5129, 5.3048, 0.56077, 3.6518, 5.3893])
```

7.4.2 Linear regression without using autograd

To get familiar with PyTorch tensors, we fit the above dataset to a linear model in the same way as we did in chapter 3, except using tensors instead of NumPy arrays. In the next section, we will utilize PyTorch autograd and optimization functions to reduce the efforts of computing the backward propagation.

```
x=torch.tensor([6.1101, 5.5277, 8.5186, 7.0032, 5.8598, 8.3829, 7.4764, 8.5781,
6.4862, 5.0546, 5.7107, 14.164, 5.734, 8.4084, 5.6407, 5.3794, 6.3654, 5.1301,
6.4296, 7.0708])

y=torch.tensor([17.592, 9.1302, 13.662, 11.854, 6.8233, 11.886, 4.3483, 12,
6.5987, 3.8166,3.2522, 15.505, 3.1551, 7.2258, 0.71618, 3.5129, 5.3048,
0.56077, 3.6518, 5.3893])

from matplotlib import pyplot as plt
fig = plt.figure(dpi=600)
plt.xlabel("x")
plt.ylabel("y")
plt.plot(x.numpy(), y.numpy(), 'o')
plt.show()
```



```
# 1) define the linear model
def model_linear(x,w,b):
    y=w*x+b
    return y

# 2) define the loss function
def loss_fn(y, label):
    se=(y-label)**2
    mse=se.mean()
    return mse

# 3) define the gradient
def grad_fn(x,y,w,b):
    y_pred=model_linear(x,w,b)
    dw=2.0*(y_pred-y)*x/y.size(0)
    db=2.0*(y_pred-y)/y.size(0)
    return torch.stack([dw.sum(),db.sum()])

# 4) define the training loop
def training_loop(n_epochs, learning_rate, params, x, y):
    loss_tensor=torch.zeros(n_epochs)
    for epoch in range(1, n_epochs+1):
        w,b=params
        y_pred=model_linear(x,w,b)
        loss=loss_fn(y_pred,y)
        grad=grad_fn(x,y,w,b)
        params=params-learning_rate*grad
        print('Epoch %d, Loss %f' % (epoch, float(loss)))
        loss_tensor[epoch-1]=loss
    return params, loss_tensor
```

```
# 5) run the training loop

params, loss_tensor= training_loop(
    n_epochs = 100,
    learning_rate = 0.002,
    params =torch.tensor([0.0,0.0]),
    x=x,
    y=y)
```

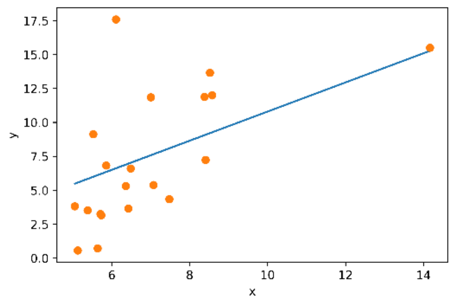
```
Epoch 1, Loss 76.284782
Epoch 2, Loss 52.976959
Epoch 3, Loss 38.543888
Epoch 4, Loss 29.606319
```

…

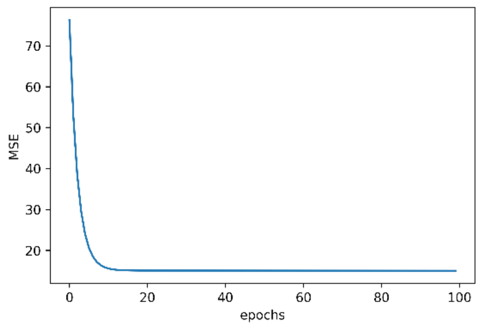
```
# 6) plot the results
params
```

```
tensor([1.0746, 0.0512])
```

```
fig = plt.figure(dpi=600)
pred=model_linear(x,params[0],params[1])
plt.xlabel("x")
plt.ylabel("y")
plt.plot(x.numpy(), pred.detach().numpy())
plt.plot(x.numpy(), y.numpy(), 'o')
```



```
fig = plt.figure(dpi=600)
plt.xlabel("epochs")
plt.ylabel("MSE")
plt.plot(loss_tensor.detach().numpy())
```



7.4.3 Linear regression using autograd

1) PyTorch autograd

In many applications, especially deep learning, it is challenging to analytically compute the derivatives of loss function with respect to parameters. PyTorch provides a component called autograd to track and compute the derivatives of a tensor with respect to its source tensors. PyTorch tensors can remember where they come from, in terms of the operations and parent tensors that originated them, and they can automatically provide the chain of derivatives of such operations with respect to their inputs. Therefore, given a forward expression, PyTorch will automatically provide the gradient of that expression with respect to its input parameters.

Applying autograd

In general, all PyTorch tensors have an attribute named *grad*. To activate the computation of gradients with respect to a tensor, say *params*, the argument, *requires_grad*, for tensor *params* has to be set **True**, so that PyTorch will track the entire family tree of tensors resulting from operations on this tensor *params*. In other words, any tensor that has *params* as an ancestor will have access to the chain of functions that were called to get from *params* to that tensor. In case these functions are differentiable (and most PyTorch tensor operations will be), the value of the derivative will be automatically populated as a grad attribute of the params tensor. Let's consider the linear regression in the previous section and compute the gradient of loss function with respect to tensor *params* ([w,b]) in one step. Fig.7.4 shows the compuation flow chart. All tensors have an attribute of *.grad, and this attribute has an initial value "None", which means that no gradient has been computed (or available).

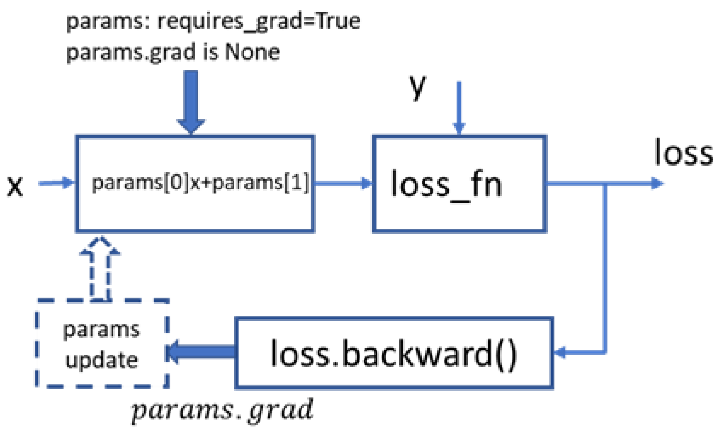


Fig.7.4 autograd computation chart

An example of using autograd is shown below.

```
params = torch.tensor([1.0, 0.0], requires_grad=True)  #initial and requires_grad to True
y_pred=model_linear(x,params[0], params[1])           #forward propagation
loss=loss_fn(y_pred,y)              #loss computation
loss.backward()            # call .backward() for autograd
params.grad              # display gradient values, dloss/dw, dloss/db
```

```
       tensor([-8.5768, -0.6954])
```

Remarks: a) the *grad* attribute of *params* contains the derivatives of the loss with respect to each element of params. Its value is "None" before the first *loss.backward()* is called. No need to compute the gradient from the analytical expression.
b) the exution of computation from *params* to *loss* is required each time before *loss.backward()* is called;
c) if we repeat the autograd from "y_pre…" statement to statement `loss.backward()` , the new gradient will accumulated (added) to the old one.

Zero gradient

As we knew previsouly, if *loss.backward* was called earlier, and the forward path and the *loss* are evaluated again, backward is called again (as in any training loop), then the gradient is accumulated (that is added) to the one computed at the previous iteration, which leads to an incorrect value for the gradient. To prevent this from happening, we need to set the gradient to zero explicitly at each iteration by inserting the following statement any location before loss.backward().

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

Disable gradient calculation

with torch.no_grad():

Context-manager that disables gradient calculation. Disabling gradient calculation is useful for inference, when you are sure that you will not call Tensor.backward(). It will reduce memory consumption for computations that would otherwise have requires_grad=True. In this mode, the result of every computation will have requires_grad=False, even when the inputs have requires_grad=True.

2) Linear regression using autograd

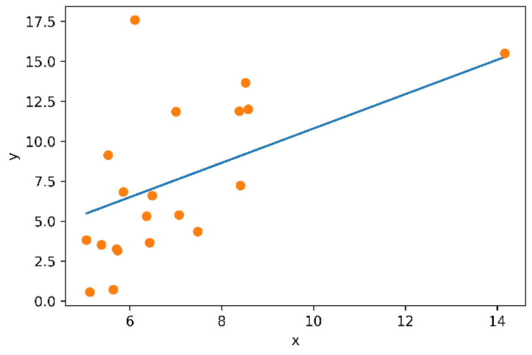
Now, we are ready to apply autograd to our previous linear regression project. Please note that: 1) we need to zero the params.grad at the beginning of iteration; 2) we disable the gradient calculation during the params updating; and 3) In-place update is used for params updating.

```
def training_loop_autograd(n_epochs, learning_rate, params, x, y):
    loss_tensor=torch.zeros(n_epochs)
    for epoch in range(1, n_epochs+1):
        if params.grad is not None:
            params.grad.zero_()
        w,b=params
        y_pred=model_linear(x,w,b)
        loss=loss_fn(y_pred,y)
        loss.backward()

        with torch.no_grad():        # no grad computation involved
            params -= learning_rate * params.grad

        #params=params-learning_rate*grad
        print('Epoch %d, Loss %f' % (epoch, float(loss)))
        loss_tensor[epoch-1]=loss
    return params, loss_tensor
```

```
# run the training loop
params, loss_tensor= training_loop_autograd(
    n_epochs = 100,
    learning_rate = 0.002,
    params =torch.tensor([0.0,0.0], requires_grad=True),
    x=x,
    y=y)

Epoch 1, Loss 76.284782
Epoch 2, Loss 52.976959
Epoch 3, Loss 38.543884
Epoch 4, Loss 29.606318
…
```

Results, shown below, are close to the results we got based on analytic gradient calculation in the previous section.

```
params

tensor([1.0746, 0.0512], requires_grad=True)

fig = plt.figure(dpi=600)
pred=model_linear(x,params[0],params[1])
plt.xlabel("x")
plt.ylabel("y")
plt.plot(x.numpy(), pred.detach().numpy())
plt.plot(x.numpy(), y.numpy(), 'o')
```



7.4.4 Linear regression using autograd and optim

To further take advantage of PyTorch, we can instantiate PyTorch optimization module for parameter updating. In chapter 6, we discussed how to update the parameters based on gradients, such as momentum and Adam algorithms, in addition to the simple constant learning rate updating. In fact, there are more optimization methods available. PyTorch provide a submodule module *torch.optim* where we can find classes implementing different optimization algorithms. It is beneficial to utilize this optimization facility in terms of code efficiency and reliability.

```
import torch.optim as optim
dir(optim)
```

```
['ASGD',
 'Adadelta',
 'Adagrad',
 'Adam',
 'AdamW',
 'Adamax',
 'LBFGS',
 'Optimizer',
 'RMSprop',
 'Rprop',
 'SGD',
 'SparseAdam',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
 '__spec__',
 'lr_scheduler']
```

To use one of the optimization algorithms in *torch.optim*, first we should construct (instantiate) an optimizer by taking a list of parameters (aka PyTorch tensors, typically with requires_grad set to True) as the first input. All parameters passed to the optimizer are retained inside the optimizer object so the optimizer can update their values and access their grad attribute. Then, we need to zero the gradient and call the optimizer. Two methods associated with the optimizer allow us to achieve this: *zero_grad* and *step*. Method *zero_grad* zeroes the grad attribute of all the parameters passed to the optimizer upon construction. Method *step* updates the value of those parameters according to the optimization strategy implemented by the specific optimizer. One iteration of parameters can be implemented as follows.

```
params = torch.tensor([0.0, 0.0], requires_grad=True)
learning_rate = 0.002
optimizer = optim.SGD([params], lr=learning_rate)   # construct an optimizer
y_pred=model_linear(x,params[0], params[1])
loss=loss_fn(y_pred,y)
loss = loss_fn(y_pred, y)                         # calculate loss
optimizer.zero_grad()      # zero grad
loss.backward()            # loss backward
optimizer.step()           # one step params update
params
```

```
tensor([0.2265, 0.0292], requires_grad=True)
```

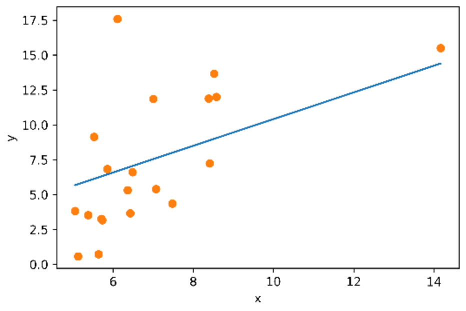
Thus, the linear regression can be implemented using autograd and optim as follows.

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

```
def training_loop_autograd_optim(optimizer, n_epochs, params, x, y):
    loss_tensor=torch.zeros(n_epochs)
    for epoch in range(1, n_epochs+1):
        w,b=params                    # forward path
        y_pred=model_linear(x,w,b)
        loss=loss_fn(y_pred,y)        # loss
        optimizer.zero_grad()         # zero grad
        loss.backward()             # calculate grad by autograd
        optimizer.step()            # update params by optimizer

        print('Epoch %d, Loss %f' % (epoch, float(loss)))
        loss_tensor[epoch-1]=loss
    return params, loss_tensor
```

Before running the training loop, we need to create a tensor params, with requires_grad=True, and instantiate an optimizer. Note that the resulting params, shown below, is slightly different from *vanilla* gradient descent algorithm. The reason is that different optimizers have different converge speed, but we set the number of epochs to 100 for all. If we increase the number of epochs for the iteration loop, the results should be closer.

```
params = torch.tensor([0.0, 0.0], requires_grad=True)
learning_rate = 1e-2
optimizer = optim.RMSprop([params], lr=learning_rate)
params, loss_tensor = training_loop_autograd_optim(
n_epochs = 100,
optimizer = optimizer,
params = params,
x = x,
y = y)

Epoch 1, Loss 76.284782
Epoch 2, Loss 64.170250
Epoch 3, Loss 56.819836
Epoch 4, Loss 51.455879
…

params

tensor([0.9578, 0.8307], requires_grad=True)

fig = plt.figure(dpi=600)
pred=model_linear(x,params[0],params[1])
plt.xlabel("x")
plt.ylabel("y")
plt.plot(x.numpy(), pred.detach().numpy())
plt.plot(x.numpy(), y.numpy(), 'o')
```



One can try a different optimizer, for example, by changing "`RMSprop`" to "`Adam`". However, the value of learning rate may need to change accordingly for an acceptable result, because different optimizations may have significantly different convergence speeds. Thus, it is helpful to plot loss function versus epochs to check whether it converges.

7.5 Neural Networks using PyTorch

So far, we understand the tensors in PyTorch and the mechanism of autograd computation on tensors. PyTorch provides various optimization methods through the package *torch.optim*. In the previous section, we demonstrated how to implement a linear regression by utilizing PyTorch. In this section we will explore the power of PyTorch through a little more complicated example: neural network for image classification. For example, Torchvision provides many built-in datasets in the *torchvision.datasets* module, as well as utility classes for building your own datasets. Transformers are available for preprocessing data. PyTorch torch.nn provides classes for neural network building blocks. The class, *DataLoader*, helps us to arrange data set during the training process.

7.5.1 Download dataset and transforms

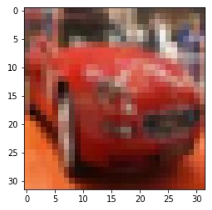
Download and load dataset

PyTorch provides a class, *torchvision.datasets*, as some popular datasets for users to download. The built-in datasets (up to date July 2020) includes (but not limited to): MNIST, Fashion-MNIST, KMNIST, EMNIST, QMNIST, FakeData, COCO, Captions, Detection, LSUN, ImageFolder, DatasetFolder, ImageNet, CIFAR, STL10, SVHN, PhotoTour, SBU, Flickr, VOC, Cityscapes, SBD, USPS, Kinetics-400, HMDB51, UCF101, and CelebA. A complete list of built-in datasets is available at https://pytorch.org/vision/stable/datasets.html. All these datasets are subclasses of torch.utils.data.Dataset, i.e, they have __getitem__ and __len__ methods implemented. Hence, they can all be passed to a torch.utils.data.DataLoader.

CIFAR-10 consists of 60,000 (32 × 32) color (RGB) images, labeled with an integer corresponding to 1 of 10 classes: airplane (0), automobile (1), bird (2), cat (3), deer (4), dog (5), frog (6), horse (7), ship (8), and truck (9). We can download the train (50,000 images) and test (10,000 images) datasets from internet to *'./data'*, load the datasets to *cifar10* and *cifar10_val*, and display one image as follows.

```
from torchvision import datasets
'''
CLASStorchvision.datasets.CIFAR10(root, train=True, transform=None, target_tran
sform=None, download=False)

--root (string) – Root directory of dataset where directory
  cifar-10-batches-py exists or will be saved to if download
  is set to True.
--train (bool, optional) – If True, creates dataset from training set,
  otherwise creates from test set.
--transform (callable, optional) – A function/transform that takes
  in an PIL image and returns a transformed version. E.g, transforms.RandomCrop
--target_transform (callable, optional) – A function/transform that
  takes in the target and transforms it.
--download (bool, optional) – If true, downloads the dataset from the internet
  and puts it in root directory.If dataset is already downloaded,
  it is not downloaded again.

Return
(image, target) where target is index of the target class.
```

```
'''

data_path = './data'
cifar10 = datasets.CIFAR10(data_path, train=True, download=True)
cifar10_val = datasets.CIFAR10(data_path, train=False, download=True)

Files already downloaded and verified
Files already downloaded and verified


print("training set:", len(cifar10))
print("validation set:", len(cifar10_val))
training set: 50000
validation set: 10000

img, label = cifar10[5]
img, label

(<PIL.Image.Image image mode=RGB size=32x32 at 0x250A5844EF0>, 1)

plt.imshow(img)
plt.show()
```



Transforms

The object, *img*, generated by the dataset class at default, has a format of PIL image, which is not ready to feed typical neural networks. Thus, we need to transform the loaded data so that they meet the requirements of the neural network input. The module, *torchvision.transforms*, defines a set of composable, function-like objects that can be passed as an argument to a torchvision dataset such as datasets.CIFAR10(…), and that perform transformations on the data after it is loaded but before it is returned by __getitem__. The list of available transforms is displayed as follows:

```
from torchvision import transforms
dir(transforms)

['CenterCrop',
 'ColorJitter',
 'Compose',
 'FiveCrop',
 'Grayscale',
 'Lambda',
 'LinearTransformation',
 'Normalize',
 'Pad',
 'RandomAffine',
 'RandomApply',
 'RandomChoice',
 'RandomCrop',
 'RandomErasing',
 'RandomGrayscale',
 'RandomHorizontalFlip',
 'RandomOrder',
 'RandomPerspective',
 'RandomResizedCrop',
 'RandomRotation',
 'RandomSizedCrop',
 'RandomVerticalFlip',
 'Resize',
 'Scale',
 'TenCrop',
 'ToPILImage',
 'ToTensor',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
 '__spec__',
 'functional',
 'transforms']
```

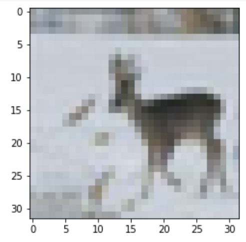
First, we transform the datasets from PIL images to tensors. Among these transforms, *ToTensor* converts NumPy arrays and PIL images to tensors. It also defines the dimensions of the output tensor as C × H × W. Whereas the values in the original PIL image ranged from 0 to 255 (8 bits per channel), the *ToTensor* transform turns the data into a 32-bit floating-point per channel, scaling the values down from 0.0 to 1.0.

```
from torchvision import transforms
to_tensor = transforms.ToTensor()
img_t = to_tensor(img)
img_t.shape

torch.Size([3, 32, 32])
```

Instead of transforming the data after loading, we can specify a transform (or a set of transforms) when we load (or download) the data.

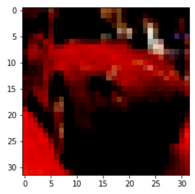
```
tensor_cifar10 = datasets.CIFAR10(data_path, train=True, download=False,
transform=transforms.ToTensor())
img_t, label = tensor_cifar10[20]
img_t.shape, label

(torch.Size([3, 32, 32]), 4)
```

```
plt.imshow(img_t.permute(1, 2, 0))       # change dimension order to H,W,Channel
plt.show()
```



Second, to improve the learning efficiency, we usually normalize the data from the range of [0,1] to the range of [-1,1]. To achieve this, we can chain two transforms: *transforms.ToTensor()*, *transforms.Normalize* using *transforms.Compose*.

```
transform = transforms.Compose([transforms.ToTensor(),
                      transforms.Normalize((0.5,0.5,0.5), (0.5, 0.5, 0.5))])
norm_cifar10 = datasets.CIFAR10(data_path, train=True, download=False,
transform=transform)
norm_img, label=norm_cifar10[5]
plt.imshow(norm_img.permute(1, 2, 0))   # change dimension order to H,W,Channel
plt.show()
```



Combination of download and transforms

Therefore, we can use the following statements for downloading and transforming (toTensor and Normalize):

```
data_path = './data'
transform = transforms.Compose([transforms.ToTensor(),
                transforms.Normalize((0.5,0.5,0.5), (0.5, 0.5, 0.5))])

norm_cifar10 = datasets.CIFAR10(data_path, train=True, download=True,
transform=transform)
norm_cifar10_test = datasets.CIFAR10(data_path, train=False, download=True,
transform=transform)

Files already downloaded and verified
Files already downloaded and verified
```

Now norm_cifar10 and norm_cifar10_test are lists [(image, label)] for normalized train set and test set, where image is tensor [3, 32, 32], and label is integer.

7.5.2 Create customized datasets from CIFAR-10

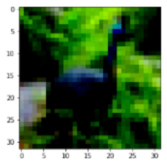
If we want to design a neural network for a binary classification task: airplane or bird, the dataset to the neural network should only include the images of airplane (originally labeled as 0) or bird (label as 2). Now let’s create this sub dataset for airplanes and birds only. In the new dataset, the labels for airplane and birds will be changed to “0” and “1”.

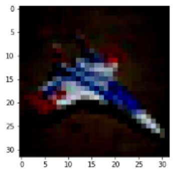
```
label_map = {0: 0, 2: 1}
class_names = ['airplane', 'bird']
norm_cifar2 = [(img, label_map[label]) for img, label in norm_cifar10 if label
in [0, 2]]
norm_cifar2_test = [(img, label_map[label]) for img, label in norm_cifar10_test
if label in [0, 2]]
```

The resulting datasets, stored in lists *norm_cifar2* and *norm_cifar2_test*, are normalized and labeled as 0 for airplane and 1 for bird. We can verify that there are 10,000 images in *norm_cifar2* and 2,000 images in *norm_cifar2_test*.

```
img, label=norm_cifar2[0]
```

```
plt.imshow(img.permute(1, 2, 0))      # change dimension order to H,W,Channel
plt.show()
```


norm_cifar2[0]   (bird)


norm_cifar2[5] (airplane)

7.5.3 Build model using nn.Sequential()

PyTorch *torch.nn* provides classes for almost all neural network building blocks. As an example, the following codes define a neural network with one 512-unit hidden layer and 2-unit output layer.

```
import torch.nn as nn
n_out = 2
model = nn.Sequential(
    nn.Linear(3072,512,),
    nn.Tanh(),
    nn.Linear(512,n_out,),
    nn.LogSoftmax(dim=1)
    )
```

nn.Linear is used to define a linear operation before activation in each layer. An instance, nn.Linear(3072, 512), accepts input tensor with shape (*, 3072) where * means any number of dimensions including none, and delivers output tensor with shape (*, 512) where all but the last dimension are the same shape as the input. 3072 and 512 are the size of each sample for input and output respectively. The input data of neural networks is typically applied to the linear module in the first hidden layer as a batch at a time for one iteration. Typically, for batch processing, the input data for the model has a shape of (batch_size, 3072).

nn.Tanh() defines the activation for the hidden layer. loss_fn = nn.NLLLoss(logsoftmax_outs, labels) takes the output of nn.LogSoftmax for a batch as the first argument and a tensor of class indices (zeros and ones, in our case) as the second argument. Please note that all the building blocks are included in order in nn.Sequential().

Now, let’s run one forward propagation on one image.

```
# get the image and label, torch.Size([3, 32, 32])
img, label = norm_cifar2[10]

# convert to batch, torch.Size([1, 3072])
img_batch = img.view(-1).unsqueeze(0)

out=model(img_batch) # compute the softmax output
out
# out torch.Size([1, 2])
# tensor([[-0.8021, -0.5949]], grad_fn=<LogSoftmaxBackward>)

loss = nn.NLLLoss()              # define loss function
loss(out, torch.tensor([label]))
# tensor(0.5949, grad_fn=<NllLossBackward>)
```

The following codes run the stochastic gradient descent (i.e. batch size =1) for training.

```
learning_rate = 1e-2
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
loss_fn = nn.NLLLoss()
n_epochs = 10
for epoch in range(n_epochs):
    for img, label in norm_cifar2:
        out = model(img.view(-1).unsqueeze(0))
        # img: torch.Size([3,32,32])
        # img.view(-1).unsqueeze(0): torch.Size([1,3072])
        # out: torch.Size([1,2])
        loss = loss_fn(out, torch.tensor([label]))
        # torch.tensor([label]) is the target index
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    print("Epoch: %d, Loss: %f" % (epoch, float(loss)))
```

```
Epoch: 0, Loss: 3.246083
Epoch: 1, Loss: 2.461254
Epoch: 2, Loss: 3.279747
Epoch: 3, Loss: 3.953358
Epoch: 4, Loss: 5.009982
Epoch: 5, Loss: 6.931544
Epoch: 6, Loss: 3.779669
Epoch: 7, Loss: 9.554771
Epoch: 8, Loss: 11.038675
Epoch: 9, Loss: 13.619962
```

Note that the loss is increasing as the training proceeds. This implies that the training process is not converging. A reader is encouraged to figure out the reason and fix it.

7.5.4 Train the model using DataLoader

In this section, to utilize PyTorch more efficiently, we modify the model in the previous section and its training approach in the following aspects:

1) Model architecture
   The model will have two hidden layers: the first with 25 units and the second with 12 units. The hidden layers use ReLU for activation.

2) Loss function
   In practice, it is convenient to compute the loss using nn.CrossEntropyLoss, which is equivalently the combination of nn.LogSoftmax and nn.NLLLoss. Furthermore, LogSoftmax is a monotonical function whose output is interpreted as the probabilities of classes. The prediction will pick a class associated with the maximum element in the output of LogSoftmax. Thus, the output of the nn.Linear in output layer (i.e. the input of LogSoftmax) can be directly used for prediction. Therefore, nn.LogSoftmax can be removed in the feed-forward model, and nn.CrossEntropyLoss can be used to compute the loss by taking linear output at the output layer and the labels (ground truth).

3) Training using DataLoader
   The *torch.utils.data* module has a class that helps with shuffling and organizing the data in minibatches: *DataLoader*. The *DataLoader* constructor takes a Dataset object as input, along

with batch_size and a shuffle Boolean that indicates whether the data needs to be shuffled at the beginning of each epoch. A *DataLoader* can be iterated over, so we can use it directly in the inner loop.

Now let’s put all things together and train the neural network.

```
import torch.optim as optim
import torch
import torch.nn as nn
train_loader = torch.utils.data.DataLoader(norm_cifar2, batch_size=64,
shuffle=True)
# norm_cifar2: list[[tensor[3,32,32], label], [tensor[3,32,32], label],…]
model = nn.Sequential(
    nn.Linear(3072,25),
    nn.ReLU(),
    nn.Linear(25,12),
    nn.ReLU(),
    nn.Linear(12,2)
 )
learning_rate = 1e-2
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

loss_fn = nn.CrossEntropyLoss()
n_epochs = 100
for epoch in range(n_epochs):
    for imgs, labels in train_loader:
        batch_size = imgs.shape[0]
        #64 for all, except 16 for the last batch, 10000/64=145.25
        #print(labels.shape)
        outputs = model(imgs.view(batch_size, -1))
        # imgs.view(batch_size,-1): [batch_size, 3072]
        # outputs: [batch_size,2]
        loss = loss_fn(outputs, labels)
        # labels: [64]
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print("Epoch: %d, Loss: %f" % (epoch, float(loss)))
```

```
Epoch: 0, Loss: 0.491145
Epoch: 1, Loss: 0.562495
Epoch: 2, Loss: 0.326086
Epoch: 3, Loss: 0.628043
Epoch: 4, Loss: 0.337991
…
Epoch: 94, Loss: 0.019337
Epoch: 95, Loss: 0.022442
Epoch: 96, Loss: 0.034940
Epoch: 97, Loss: 0.022771
Epoch: 98, Loss: 0.161970
Epoch: 99, Loss: 0.052434
```

To test the accuracy of the trained model on the test dataset, we can similarly feed the data in batches, and compare the predictions with the labels. Note that the data in the batch is not shuffled, and that the gradient calculation is disabled.

```
val_loader = torch.utils.data.DataLoader(norm_cifar2_test, batch_size=64,
shuffle=False)
correct = 0
total = 0
with torch.no_grad():
    for imgs, labels in val_loader:
        batch_size = imgs.shape[0]
        outputs = model(imgs.view(batch_size, -1))

        values, predicted = torch.max(outputs, dim=1) # return (values,indices)
        #print(values, predicted.shape)
        total += labels.shape[0]
        correct += int((predicted == labels).sum())
print("Accuracy: %f" % (correct / total))
Accuracy: 0.837500
```

7.5.5 Access parameters of the trained model

The overall architecture of model can be printed by

```
model.parameters

<bound method Module.parameters of Sequential(
  (0): Linear(in_features=3072, out_features=25, bias=True)
  (1): ReLU()
  (2): Linear(in_features=25, out_features=12, bias=True)
  (3): ReLU()
  (4): Linear(in_features=12, out_features=2, bias=True)
)>
```

The following statement prints all the named parameters, their shapes and values.

```
for name, param in model.named_parameters():
    if param.requires_grad:
        print (name, param.data.shape, param.data)
```

PyTorch offers a quick way to determine how many parameters a model has through the parameters() method of nn.Model (the same method we use to provide the parameters to the optimizer). To find out how many elements are in each tensor instance, we can call the numel method. Summing those gives us our total count. Counting parameters might require us to check whether a parameter has requires_grad set to True, as well.

```
numel_list = [p.numel() for p in model.parameters()]
sum(numel_list), numel_list

(77163, [76800, 25, 300, 12, 24, 2])
```

7.6 An Example by PyTorch: Finger Signs

In this section, we will show how easily one can implement a neural network for finger sign recognition by PyTorch. The neural network architecture is shown in Fig.7.5. The dataset files are train_signs.h5 (1080 images for training) and test_signs.h5 (120 images for testing). Each image (64x64x3) is labeled by one of digits from 0 to 5 corresponding to a certain combination of finger positions.

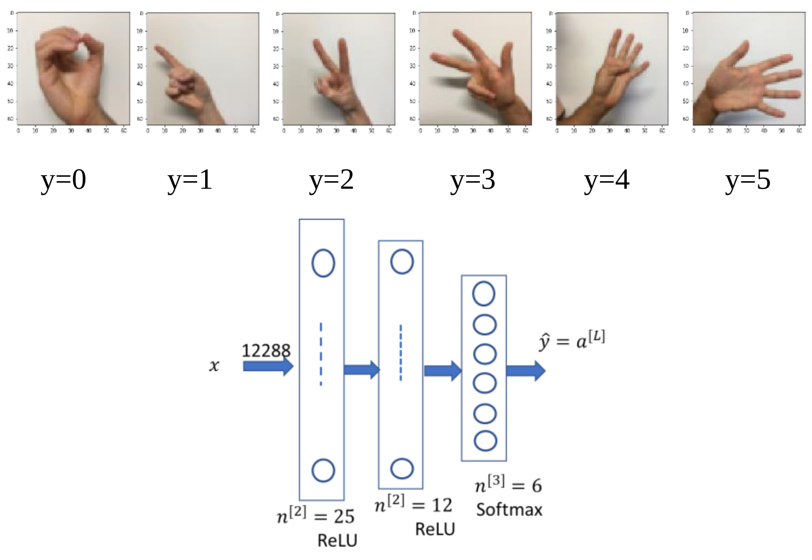


Fig.7.5 neural network for finger sign recognition

Through this example, we will learn to:

1) Create a class *Dataset* for DataLoader.
2) Apply regularization: weight penalty or dropout in neural network.
3) Use model.train() for training and model.eval() for testing.

Import packages.

```
import torch
from torchvision import transforms, datasets
import torch.nn.functional as F
import torch.nn as nn
import torch.optim as optim
import h5py
import numpy as np
from torch.utils.data import Dataset, DataLoader

%matplotlib inline
from matplotlib import pyplot as plt

torch.manual_seed(0)
```

7.6.1 Create Dataset for DataLoader

If the datasets are not available in the built-in “torchvision.datasets”, we need to generate datasets which have the same format as torchvision.datasets so that we can use torch.utils.data.DataLoader in training loop or testing process. In our case, since the datasets are stored in *.h5 files, an effort is needed to generate the corresponding dataset class.

```
class signs_train_Dataset(Dataset):

    def __init__(self,root_dir,filename, transforms):
        self.transform = transforms
        self.root_dir = root_dir
```

```python
        # read the train data file
        self.train_dataset = h5py.File(self.root_dir+filename, "r")
        # your train set features
        self.train_set_x_orig = np.array(self.train_dataset["train_set_x"][:])
        # your train set labels
        self.train_set_y_orig = np.array(self.train_dataset["train_set_y"][:])

    def __len__(self):
        return (self.train_set_x_orig.shape[0])

    def __getitem__(self, index):
        image = self.train_set_x_orig[index]
        if self.transform:
            image = self.transform(image)
        y = self.train_set_y_orig[index]
        return [image, y]


class signs_test_Dataset(Dataset):

    def __init__(self,root_dir,filename, transforms):
        self.transform = transforms
        self.root_dir = root_dir
        # read the test data file
        self.test_dataset = h5py.File(self.root_dir+filename, "r")
        # your test set features
        self.test_set_x_orig = np.array(self.test_dataset["test_set_x"][:])
        # your test set labels
        self.test_set_y_orig = np.array(self.test_dataset["test_set_y"][:])

    def __len__(self):
        return (self.test_set_x_orig.shape[0])

    def __getitem__(self, index):
        image = self.test_set_x_orig[index]
        y = self.test_set_y_orig[index]
        if self.transform:
            image = self.transform(image)
        return [image, y]
```

Based on the definitions of the Datasets above, we can generate the datasets by specifying the file path, file name, and transforms, and then create DataLoaders by specifying batch size and shuffle.

```python
train_dataset = signs_train_Dataset("C:/Users/weido/ch11/",'train_signs.h5', tr
ansforms=transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,0
.5,0.5), (0.5, 0.5, 0.5))]))

test_dataset = signs_test_Dataset("C:/Users/weido/ch11/",'test_signs.h5', trans
forms=transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,0.5,
0.5), (0.5, 0.5, 0.5))]))
```

```python
train_dataloader = DataLoader(train_dataset, batch_size=32, shuffle=True)

test_dataloader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

Now let’s explore and verify *train_dataloader* and *test_dataloader*.

```python
for imgs, labels in train_dataloader:
   print(imgs.shape, labels.shape)

for imgs, labels in test_dataloader:
   print(imgs.shape, labels.shape)
```

The results will show how many batches and the shape of each batch in each dataloader.

7.6.2 Neural network model

The neural network is defined as follows. We add dropout to the first hidden layer.

```
model = nn.Sequential(
    nn.Linear(12288,25),
    nn.ReLU(),
    # dropout
    nn.Dropout(p=0.4),
    nn.Linear(25,12),
    nn.ReLU(),
    nn.Linear(12,6)
 )
```

7.6.3 Train the model

The training process covers two for-loops. The outer for-loop is designed for epoch iterations while the inner for-loop iterates on batches. We can add weight regularization by modifying the loss, see the part commented out. Typically, to implement regularization, we either use dropout or weight penalty, but not both. Thus, we suggest that you remove the dropout in the model definition if you want to do weight regularization here.

```
learning_rate = 1e-3
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

loss_fn = nn.CrossEntropyLoss()
n_epochs = 500
for epoch in range(n_epochs):
    for imgs, labels in train_dataloader:
        batch_size = imgs.shape[0]
        #print(labels.shape)
        outputs = model.train()(imgs.view(batch_size, -1))
        # input to the model: [batch_size, 12288]
        # output: [batch_size, 6]
        loss = loss_fn(outputs, labels) # loss: scalar
        #####--- regularization ----####
        #l2_lambda = 0.005
        #l2_norm = sum(p.pow(2.0).sum() for p in model.parameters())
        #loss = loss + l2_lambda * l2_norm
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    if epoch % 10==0:
        print("Epoch: %d, Loss: %f" % (epoch, float(loss)))
```

```
Epoch: 0, Loss: 1.832473
Epoch: 10, Loss: 1.676556
Epoch: 20, Loss: 1.554291
…
Epoch: 470, Loss: 0.425223
Epoch: 480, Loss: 0.339189
Epoch: 490, Loss: 0.272545
```

7.6.4 Test the model

Now, we test the trained model using test_dataloader. To perform the inference, be sure to disable the gradient calculation using *with torch.no_grad()*, and *model.eval()*.

```
correct = 0
total = 0
with torch.no_grad():
    for imgs, labels in test_dataloader:
        batch_size = imgs.shape[0]
        outputs = model.eval()(imgs.view(batch_size, -1))
        values, predicted = torch.max(outputs, dim=1)
        total += labels.shape[0]
        correct += int((predicted == labels).sum())
print("Accuracy: %f" % (correct / total))
```

```
Accuracy: 0.891667
```

The above test, if applied on the train_dataloader, shows the training accuracy of 0.998, which indicates a slight overfitting because the accuracy on test_dataloader is just 0.891667.

Summary

This chapter presents the basics of PyTorch, and demonstrate the major steps of developing and training a neural network using PyTorch. There are a few important things of PyTorch make our job easy:

1) PyTorch classes (Dataset, Compose, DataLoader) help us prepare data.
2) torch.nn provides modules, such as Linear, sigmoid, Tanh, ReLU, LogSoftmax, NLLLoss, CrossEntropyLoss, so that we can build a neural network efficiently by instantiating relevant modules. We don’t need to develop the code at a matrix-level.
3) Most importantly, during the training process, PyTorch can automatically track and calculate the gradient of loss with respect to parameters.
4) Various optimizers are available in PyTorch library.
5) Implementations of regularizations (weight penalty or dropout).
6) It is easy to access the information about the parameters of a trained model.

PyTorch is a powerful tool for us to implement classical deep learning architectures (e.g. convolution neural networks) in the subsequent chapters.

Files:
C:/Users/weido/torch_tutorial/basics_pytorch.ipynb
 ‘../torch_tutorial/data/faces/person.jpg’

‘../torch_tutorial/data/student_faces/andrew0.jpg, Antonio5.jpg, heidi0.jpg, Jonathon5.jpg
C:/machine_learning/mnist_test.csv, mnist_train.csv
‘../torch_tutorial/data/winequality-white.csv’
C: /Users/weido/ch11/finger_signs.ipynb
C:/Users/weido/ch11/test_signs.h5
C:/Users/weido/ch11/train_signs.h5

References

[1] Chapters 1 to 7, “Deep Learning with PyTorch”, Eli Stevens, Luca Antiga, Thomas Viehmann. 2020 by Manning Publications Co.

[2] https://pytorch.org/

Exercises

1. Build and train a neural network to classify images: bird/cat using PyTorch:

    1) Create your own training set and test set from CIFAR-10. The images in your datasets are either bird or cat, and normalized.

    2) The neural network has two hidden layers with ReLU activations. The first hidden layer has 20 units and the second layer has 10 units. The output layer is Linear with two units. Thus, nn.CrossEntropyLoss is used for loss computation.

    3) Train your network on your own training set, created in 1).

    4) Test your network on your own test set, created in 1). Print the accuracy.

    5) How many parameters in your neural network? Print the values of all parameters.

2. Utilizing PyTorch resources, build and train a neural network to recognize handwritten digits (MNIST dataset). You are free to select a reasonable architecture for the neural network. At the end, test your trained model, and print the accuracy on the test set.

3. In Section 7.5.3, the training process does not converge because the loss keeps increasing. Try the following modifications (separately) to see whether they work:

    1) Reduce the value of *learning_rate*, and increase the value of *n_epochs*.

    2) Implement mini-batch gradient descent optimization, with an appropriate value of batch size.

4. To address the overfitting in the example in Section 7.6, try the following experiments:

    1) Change the p value in the nn.Dropout, and train the neural network.

    2) Remove the dropout and add weight regularization. You choose an appropriate value for regularization factor λ, and train the neural network.

3) Reduce the complexity of the neural network, and train the neural network.

5. Suppose we want to create a DataLoader for training a neural network for a 10-class classification task. For training, we have 100 images for each class. In the folder "/*images*", 10 subfolders are created to store the images for 10 classes respectively. For example, "*/images/C0*" stores 100 images for class 0, and "*/images/C9*" stores 100 images for class 9, and so on. In each subfolder Cj (where j=0,1,…,9), the image files are named as Cj*_img0.jpg, Cj_img1.jpg, …, Cj_img99.jpg*.

   Create the following extended class *my_Dataset*

   ```
   class my_Dataset(Dataset):
   ```

   so that we can generate a train_dataloader

   ```
   train_dataset = my_Dataset("./images", transforms=transforms)
   train_dataloader = DataLoader(train_dataset, batch_size=4, shuffle=True)
   ```