Chapter 10

Introduction to PyTorch

10.1 Learning Objectives

So far we have studied the fundamentals of machine learning and neural networks. We implemented and trained the machine learning algorithms in Python from the scratch. To make a solid understand of the fundamentals, it is essential for us to be able to implement an algorithm from the scratch (without relying on package tools). In practice, however, it is very time consuming and error-prone to build everything from the scratch when the project (e.g. deep learning) is typically sophisticated in the real world. In fact, there are some frameworks or package tools available to accelerate the development of machine learning products. 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. In addition, from now on we will use Jupyter Notebook as the design environment. Appendix A gives a tutorial of Jupyter Notebook and PyTorch installation. This chapter covers:

- Major benefits 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

10.2 Why PyTorch?

As we will see later, 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, by exposing our model to illustrative examples. In order to do so in practice, we need tools that are flexible, so 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 library 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.

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. 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 something PyTorch can actually handle: tensors. 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.

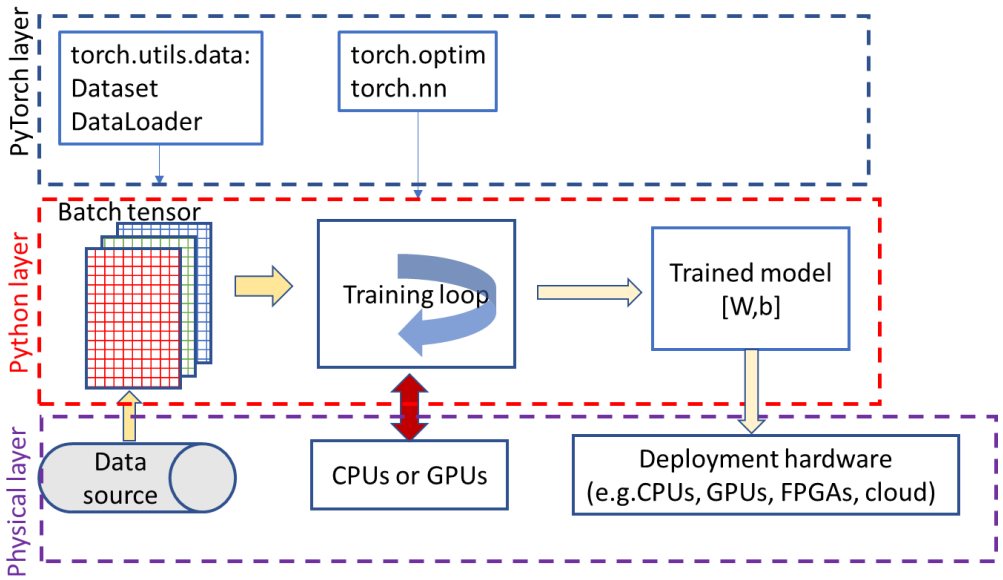


Fig. 1 PyTorch framework for deep learning

10.3 Tensors

10.3.1 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.

```
import numpy as np
import torch
points=torch.ones(2,3)   # create a tensor
points_np=points.numpy()   # convert to numpy
b=torch.tensor(points_np) #convert to tensor
```

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

```
In [35]: double_points = torch.ones(10, 2, dtype=torch.double)
         short_points = torch.tensor([[1, 2], [3, 4]], dtype=torch.short)

In [36]: double_points.dtype

Out[36]: torch.float64

In [37]: short_points.dtype

Out[37]: torch.int16
```

10.3.2 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). The vast majority of operations on and between tensors are available in the torch module and can also be called as methods of a tensor object. A summary of operations is given below.(https://jhui.github.io/2018/02/09/PyTorch-Basic-operations/)

1) Create a tensor

Creating and initializing a Tensor

```
### Create a Tensor note: case-sensitive
v = torch.tensor([2,3])         # a tensor initialized with a list, int64
v = torch.Tensor([2,3])        # a tensor initialized with a list, float32
v = torch.Tensor(2, 3)          # An un-initialized torch.FloatTensor of size 2x3
                                # v = torch.tensor(2,3) is not valid
v = torch.Tensor([[1,2],[4,5]]) # A Tensor initialized with a specific array, float32
v = torch.LongTensor([1,2,3])   # A Tensor of type Long
```

Create a random Tensor

To increase the reproducibility of result, we often set the random seed to a specific value first.

```
torch.manual_seed(1)
v = torch.rand(2, 3)     # Initialize with random number (uniform distribution)
v = torch.randn(2, 3)    # With normal distribution (SD=1, mean=0)
v = torch.randperm(4)    # Size 4. Random permutation of integers from 0 to 3
```

Tensor type

```
x = torch.randn(5, 3).type(torch.FloatTensor)
```

Identity matrices, Fill Tensor with 0, 1 or values

```
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)      # 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

# 1  1  1
# 2  2  2
# 3  3  3
v = torch.ones(3, 3)
v[1].fill_(2)
v[2].fill_(3)
```

Initialize Tensor with a range of value

```
v = torch.arange(5)             # similar to range(5) but creating a Tensor
v = torch.arange(0, 5, step=1)  # Size 5. Similar to range(0, 5, 1)

# 0 1 2
```

```
# 3 4 5
# 6 7 8
v = torch.arange(9)
v = v.view(3, 3)
```

Initialize a linear or log scale Tensor

```
v = torch.linspace(1, 10, steps=10)
# 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
```

Initialize a ByteTensor

```
c = torch.ByteTensor([0, 1, 1, 0])
```

Summary

```
Creation Ops
~~~~~~~~~~~~~~~~~~~~
.. autofunction:: eye
.. autofunction:: from_numpy
.. autofunction:: linspace
.. autofunction:: logspace
.. autofunction:: ones
.. autofunction:: ones_like
.. autofunction:: arange
.. autofunction:: range
.. autofunction:: zeros
.. autofunction:: zeros_like
```

2) Indexing, Slicing, Joining, Mutating Ops

We will prepare a Matrix that will be used in this section:

```
# 0 1 2
# 3 4 5
# 6 7 8
v = torch.arange(9)
v = v.view(3, 3)
```

Concatenate, stack

```
# Concatenation
torch.cat((x, x, x), 0)          # Concatenate in the 0 dimension

# Stack
r = torch.stack((v, v))
```

Gather : reorganize data element

```
# Gather element
# torch.gather(input, dim, index, out=None)
# out[i][j][k] = input[index[i][j][k]][j][k]  # if dim == 0
# out[i][j][k] = input[i][index[i][j][k]][k]  # if dim == 1
# out[i][j][k] = input[i][j][index[i][j][k]]  # if dim == 2

# 0  1
# 4  3
# 8  7
```

```
r = torch.gather(v, 1, torch.LongTensor([[0,1],[1,0],[2,1]]))
```

Split a Tensor

```
# Split an array into 3 chunks
# (
# 0  1  2
# [torch.FloatTensor of size 1x3]
# ,
# 3  4  5
# [torch.FloatTensor of size 1x3]
# ,
# 6  7  8
# [torch.FloatTensor of size 1x3]
# )
r = torch.chunk(v, 3)

# Split an array into chunks of at most size 2
# (
# 0  1  2
# 3  4  5
# [torch.FloatTensor of size 2x3]
# ,
# 6  7  8
# [torch.FloatTensor of size 1x3]
# )
r = torch.split(v, 2)
```

Index select, mask select

```
# Index select
# 0 2
# 3 5
# 6 8
indices = torch.LongTensor([0, 2])
r = torch.index_select(v, 1, indices) # Select element 0 and 2 for each dimension 1.

# Masked select
# 0  0  0
# 1  1  1
# 1  1  1
mask = v.ge(3)

# Size 6: 3 4 5 6 7 8
r = torch.masked_select(v, mask)
```

Squeeze and unsqueeze

```
t = torch.ones(2,1,2,1) # Size 2x1x2x1
r = torch.squeeze(t)     # Size 2x2
r = torch.squeeze(t, 1)  # Squeeze dimension 1: Size 2x2x1

# Un-squeeze a dimension
x = torch.Tensor([1, 2, 3])
r = torch.unsqueeze(x, 0)       # Size: 1x3
r = torch.unsqueeze(x, 1)       # Size: 3x1
```

Non-zero elements

```
# Non-zero
```

```
# [torch.LongTensor of size 8x2]
# [i, j] index for non-zero elements
#    0     1
#    0     2
#    1     0
#    1     1
#    1     2
#    2     0
#    2     1
#    2     2
r = torch.nonzero(v)
```

take

```
# Flatten a TensorFlow and return elements with given indexes
# Size 3: 0, 4, 2
r = torch.take(v, torch.LongTensor([0, 4, 2]))
```

transpose

```
# Transpose dim 0 and 1
r = torch.transpose(v, 0, 1)
```

Summary

```
Indexing, Slicing, Joining, Mutating Ops
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
.. autofunction:: cat
.. autofunction:: chunk
.. autofunction:: gather
.. autofunction:: index_select
.. autofunction:: masked_select
.. autofunction:: nonzero
.. autofunction:: split
.. autofunction:: squeeze
.. autofunction:: stack
.. autofunction:: t           - Transpose a 2-D tensor
.. autofunction:: take
.. autofunction:: transpose
.. autofunction:: unbind      -Removes a tensor dimension
.. autofunction:: unsqueeze
.. autofunction:: where   -Select x or y Tensor elements based on condition Tensor c
```

3) Distribution

Uniform, bernoulli, multinomial, normal distribution

```
# 2x2: A uniform distributed random matrix with range [0, 1]
r = torch.Tensor(2, 2).uniform_(0, 1)

# bernoulli
r = torch.bernoulli(r)
# Size:2x2.Bernoulli with probability p stored in elements of r

# Multinomial
w = torch.Tensor([0, 4, 8, 2]) # Create a tensor of weights
r = torch.multinomial(w, 4, replacement=True) # Size 4: 3, 2, 1, 2
```

```
# Normal distribution
# From 10 means and SD
r = torch.normal(means=torch.arange(1, 11), std=torch.arange(1, 0.1, -0.1))
# Size 10
```

Summary

```
Random sampling
--------------------------------
.. autofunction:: manual_seed    - Set a manual seed
.. autofunction:: initial_seed   - Randomize a seed by the system
.. autofunction:: get_rng_state
.. autofunction:: set_rng_state
.. autodata:: default_generator
.. autofunction:: bernoulli
.. autofunction:: multinomial
.. autofunction:: normal
.. autofunction:: rand
.. autofunction:: randn
.. autofunction:: randperm

In-place random sampling
~~~~~~~~~~~~~~~~~~~~~~~~

There are a few more in-place random sampling functions defined on Tensors as well.
Click through to refer to their documentation:

- :func:`torch.Tensor.bernoulli_` - in-place version of :func:`torch.bernoulli`
- :func:`torch.Tensor.cauchy_` - numbers drawn from the Cauchy distribution
- :func:`torch.Tensor.exponential_` - numbers drawn from the exponential distribution
- :func:`torch.Tensor.geometric_` - elements drawn from the geometric distribution
- :func:`torch.Tensor.log_normal_` - samples from the log-normal distribution
- :func:`torch.Tensor.normal_` - in-place version of :func:`torch.normal`
-  :func:`torch.Tensor.random_`  -  numbers  sampled  from  the  discrete  uniform
distribution
-  :func:`torch.Tensor.uniform_`  -  numbers  sampled  from  the  continuous  uniform
distribution
```

4) Point-wise operations

```
### Math operations
f= torch.FloatTensor([-1, -2, 3])
r = torch.abs(f)      # 1 2 3

# Add x, y and scalar 10 to all elements
r = torch.add(x, 10)
r = torch.add(x, 10, y)

# Clamp the value of a Tensor
r = torch.clamp(v, min=-0.5, max=0.5)

# Element-wise divide
r = torch.div(v, v+0.03)

# Element-wise multiple
r = torch.mul(v, v)
```

Summary

```
    Pointwise Ops
    ~~~~~~~~~~~~~~~~~~~~~~

    .. autofunction:: abs
    .. autofunction:: acos           - arc cosine
    .. autofunction:: add
    .. autofunction:: addcdiv        - element wise: t1 + s * t2/t3
    .. autofunction:: addcmul        - element wise: t1 + s * t2 * t3
    .. autofunction:: asin           - arc sin
    .. autofunction:: atan
    .. autofunction:: atan2
    .. autofunction:: ceil           - ceiling
    .. autofunction:: clamp          - clamp elements into a range
    .. autofunction:: cos
    .. autofunction:: cosh
    .. autofunction:: div            - divide
    .. autofunction:: erf            - Gaussian error functiom
    .. autofunction:: erfinv         - Inverse
    .. autofunction:: exp
    .. autofunction:: expm1          - exponential of each element minus 1
    .. autofunction:: floor
    .. autofunction:: fmod           - element wise remainder of division
    .. autofunction:: frac           - fraction part 3.4 -> 0.4
    .. autofunction:: lerp           - linear interpolation
    .. autofunction:: log            - natural log
    .. autofunction:: log1p          - y = log(1 + x)
    .. autofunction:: mul            - multiple
    .. autofunction:: neg
    .. autofunction:: pow
    .. autofunction:: reciprocal     - 1/x
    .. autofunction:: remainder      - remainder of division
    .. autofunction:: round
    .. autofunction:: rsqrt          - the reciprocal of the square-root
    .. autofunction:: sigmoid        - sigmode(x)
    .. autofunction:: sign
    .. autofunction:: sin
    .. autofunction:: sinh
    .. autofunction:: sqrt
    .. autofunction:: tan
    .. autofunction:: tanh
    .. autofunction:: trunc          - truncated integer
```

5) Reduction operations

```
    ### Reduction operations

    # Accumulate sum
    #  0   1   2
    #  3   5   7
    #  9  12  15
    r = torch.cumsum(v, dim=0)

    # L-P norm
    r = torch.dist(v, v+3, p=2)  # L-2 norm: ((3^2)*9)^(1/2) = 9.0
```

```
# Mean
# 1 4 7
r = torch.mean(v, 1)         # Size 3: Mean in dim 1

r = torch.mean(v, 1, True)   # Size 3x1 since keep dimension = True

# Sum
# 3 12 21
r = torch.sum(v, 1)          # Sum over dim 1

# 36
r = torch.sum(v)
```

Summary

```
Reduction Ops
~~~~~~~~~~~~~~~~~~~~~
.. autofunction:: cumprod        - accumulate product of elements x1*x2*x3...
.. autofunction:: cumsum
.. autofunction:: dist           - L-p norm
.. autofunction:: mean
.. autofunction:: median
.. autofunction:: mode
.. autofunction:: norm           - L-p norm
.. autofunction:: prod           - accumulate product
.. autofunction:: std            - compute standard deviation
.. autofunction:: sum
.. autofunction:: var            - variance of all elements
```

6) Comparison operation

```
### Comparison
# Size 3x3: Element-wise comparison
r = torch.eq(v, v)

# Max element with corresponding index
r = torch.max(v, 1)
```

Sort

```
# Sort
# Second tuple store the index
# (
#  0  1  2
#  3  4  5
#  6  7  8
# [torch.FloatTensor of size 3x3]
# ,
#  0  1  2
#  0  1  2
#  0  1  2
# [torch.LongTensor of size 3x3]
r = torch.sort(v, 1)
```

k-th and top k

```
# k-th element (start from 1) ascending order with corresponding index
# (1 4 7
```

```
# [torch.FloatTensor of size 3]
# , 1 1 1
# [torch.LongTensor of size 3]
# )
r = torch.kthvalue(v, 2)

# Top k
# (
#  2  5  8
# [torch.FloatTensor of size 3x1]
# ,
#  2  2  2
# [torch.LongTensor of size 3x1]
# )
r = torch.topk(v, 1)

Comparison Ops
~~~~~~~~~~~~~~~~~~~~~~
.. autofunction:: eq             - Compare elements
.. autofunction:: equal          - True of 2 tensors are the same
.. autofunction:: ge             - Element-wise greater or equal comparison
.. autofunction:: gt
.. autofunction:: kthvalue       - k-th element
.. autofunction:: le
.. autofunction:: lt
.. autofunction:: max
.. autofunction:: min
.. autofunction:: ne
.. autofunction:: sort
.. autofunction:: topk           - top k
```

7) Matrix, vector multiplication

Dot product of Tensors

```
# Dot product of 2 tensors
r = torch.dot(torch.Tensor([4, 2]), torch.Tensor([3, 1])) # 14
```

Matrix, vector products

```
### Matrix, vector products

# Matrix X vector
# Size 2x4
mat = torch.randn(2, 4)
vec = torch.randn(4)
r = torch.mv(mat, vec)

# Matrix + Matrix X vector
# Size 2
M = torch.randn(2)
mat = torch.randn(2, 3)
vec = torch.randn(3)
r = torch.addmv(M, mat, vec)
```

Matrix, Matrix products

```
# Matrix x Matrix
# Size 2x4
```

```
mat1 = torch.randn(2, 3)
mat2 = torch.randn(3, 4)
r = 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)
r = torch.addmm(M, mat1, mat2)
```

Outer product of vectors

```
# Outer product of 2 vectors
# Size 3x2
v1 = torch.arange(1, 4)    # Size 3
v2 = torch.arange(1, 3)    # Size 2
r = torch.ger(v1, v2)

# Add M with outer product of 2 vectors
# Size 3x2
vec1 = torch.arange(1, 4)  # Size 3
vec2 = torch.arange(1, 3)  # Size 2
M = torch.zeros(3, 2)
r = torch.addr(M, vec1, vec2)
```

Batch matrix multiplication

```
# Batch Matrix x Matrix
# Size 10x3x5
batch1 = torch.randn(10, 3, 4)
batch2 = torch.randn(10, 4, 5)
r = torch.bmm(batch1, batch2)

# Batch Matrix + Matrix x Matrix
# Performs a batch matrix-matrix product
# 3x4 + (5x3x4 X 5x4x2 ) -> 5x3x2
M = torch.randn(3, 2)
batch1 = torch.randn(5, 3, 4)
batch2 = torch.randn(5, 4, 2)
r = torch.addbmm(M, batch1, batch2)
```

8) Other

Cross product

```
m1 = torch.ones(3, 5)
m2 = torch.ones(3, 5)
v1 = torch.ones(3)

# Cross product
# Size 3x5
r = torch.cross(m1, m2)
```

Diagonal matrix

```
# Diagonal matrix
# Size 3x3
r = torch.diag(v1)
```

Histogram

```
# Histogram
# [0, 2, 1, 0]
torch.histc(torch.FloatTensor([1, 2, 1]), bins=4, min=0, max=3)
```

Renormalization

```
# Renormalize
# Renormalize for L-1 at dim 0 with max of 1
# 0.0000  0.3333  0.6667
# 0.2500  0.3333  0.4167
# 0.2857  0.3333  0.3810
r = torch.renorm(v, 1, 0, 1)
```

Summary

```
Other Operations
~~~~~~~~~~~~~~~~~~~~~~~
.. autofunction:: cross           - cross product
.. autofunction:: diag            - convert vector to diagonal matrix
.. autofunction:: histc           - histogram
.. autofunction:: renorm          - renormalize a tensor
.. autofunction:: trace           - tr(M)
.. autofunction:: tril            - lower triangle of 2-D matrix
.. autofunction:: triu            - uppser triangle
```

9) A summary of available operations:

```
Tensors
---------------------------------
.. autofunction:: is_tensor
.. autofunction:: is_storage
.. autofunction:: set_default_tensor_type
.. autofunction:: numel
.. autofunction:: set_printoptions

Serialization
---------------------------------
.. autofunction:: save          - Saves an object to a disk file
.. autofunction:: load         - Loads an object saved with torch.save() from a file

Parallelism
---------------------------------
..  autofunction:: get_num_threads  -  Gets  the  number  of  OpenMP  threads  used  for
parallelizing CPU operations
.. autofunction:: set_num_threads

Spectral Ops
~~~~~~~~~~~~~~~~~~~~~~~
.. autofunction:: stft          - Short-time Fourier transform
.. autofunction:: hann_window   - Hann window function
.. autofunction:: hamming_window  - Hamming window function
.. autofunction:: bartlett_window - Bartlett window function

BLAS and LAPACK Operations
~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
.. autofunction:: addbmm           - Batch add and mulitply matrices nxp + b×n×m X
b×m×p -> bxnxp
.. autofunction:: addmm           - Add and mulitply matrices nxp + n×m X m×p -> nxp
.. autofunction:: addmv           - Add and matrix, vector multipy n + nxm X m -> n
.. autofunction:: addr            - Outer product of vectors
.. autofunction:: baddbmm         - Batch add and mulitply matrices
.. autofunction:: bmm             - Batch mulitply matrices b×n×m X b×m×p -> b×n×p
.. autofunction:: btrifact        - LU factorization
.. autofunction:: btrifact_with_info
.. autofunction:: btrisolve
.. autofunction:: btriunpack
.. autofunction:: dot             - Dot product of 2 tensors
.. autofunction:: eig             - Eigenvalues and eigenvectors ofsquare matrix
.. autofunction:: gels            - Solution for least square or p-norm(AX - B)
.. autofunction:: geqrf
.. autofunction:: ger             - Outer product of 2 vectors
.. autofunction:: gesv            - Solve linear equations
.. autofunction:: inverse         - Inverse of square matrix
.. autofunction:: det             - Determinant of a 2D square Variable
.. autofunction:: matmul          - Matrix product of tensors
.. autofunction:: mm                            - Matrix multiplication
.. autofunction:: mv              - Matrix vector product
.. autofunction:: orgqr           - Orthogal matrix Q
.. autofunction:: ormqr           - Multiplies matrix by the orthogonal Q matrix
.. autofunction:: potrf           - Cholesky decomposition
.. autofunction:: potri            - Inverse of a positive semidefinite matrix with
Cholesky
.. autofunction:: potrs           - Solve linear equation with positive semidefinite
.. autofunction:: pstrf          - Cholesky decomposition of a positive semidefinite
matrix
.. autofunction:: qr              - QR decomposition
.. autofunction:: svd             - SVD decomposition
.. autofunction:: symeig          - Eigenvalues and eigenvectors
.. autofunction:: trtrs            - Solves a system of equations with a triangular
coefficient
```

10.4 Data Representation using Tensors

In this section, we will describe how to represent real-world data using tensors.

10.4.1 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.

There are different formats to store images in files and different ways to load (read) image file in Python. As long as the 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 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
img_arr = imageio.imread('../torch_tutorial/data/faces/person.jpg')
img_tensor=torch.from_numpy(img_arr)          #torch.Size([239, 209, 3])
out=img_tensor.permute(2,0,1)                 #torch.Size([3, 239, 209])
```

10.4.2 Excel CSV files

Another format for data storage is spreadsheet or CSV file. We can use *numpy.loadtxt* to read the data in a CSV file. For example, the image samples of handwritten digits are saved as CSV files mnist_train.csv and mnist_test.csv. In these MNIST CSV files, each row represents one image. The first element in a row represents the label for corresponding row. The following statements read

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

Some CSV files use ";" as delimiter and/or include a header line. For example, the wine quality dataset, publicly available, 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.2.

	A	B	C	D	E	F	G	H	I	J	K	L
1	fixed ac	volati	citric	resid	chlorid	free s	total s	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.2 a portion of winequality-white.csv

```
wine_path = "../torch_tutorial/data/winequality-white.csv"
wine_numpy = np.loadtxt(wine_path, delimiter=";", skiprows=1)
wine_tensor=torch.from_numpy(wine_numpy)
```

```
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)
```

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']
```

10.5 Linear Regression using PyTorch

In this section, we will implement a linear regression example using PyTorch. Through the example, we will show how to use the PyTorch resources for machine learning. These resources will significantly reduce the efforts of development.

10.5.1 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])
```

The dataset is visualized by the following statements, as shown in Fig.3.

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

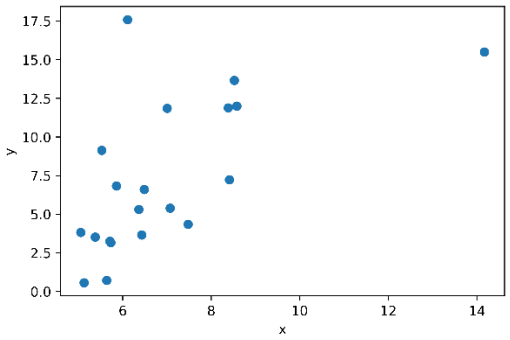


Fig.3 Training set

10.5.2 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.

1) Define linear model

```
def model_linear(x,w,b):
    y=w*x+b
    return y
```

2) Calculate loss function.

```
def loss_fn(y, label):
    se=(y-label)**2
    mse=se.mean()
    return mse
```

3) Calculate 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)*1.0
    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)
```

6) Plot results.

```
params
```

```
tensor([1.0324, 0.3026])
```

```
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())
```

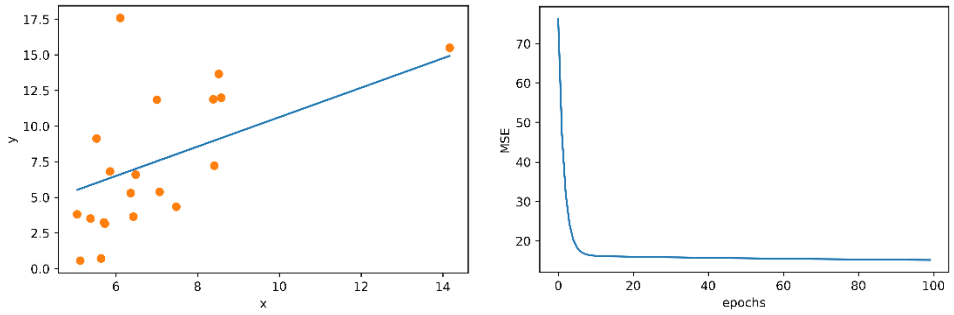


Fig.4 Result of linear regression: a) fitting curve, b) Mean square errors versus epochs

10.5.3 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 use the dataset in Fig.3 and compute the gradient of loss function with respect to tensor *params* ([w,b]) in one step. Fig 5 shows the compuation flow chart.

```
params = torch.tensor([1.0, 0.0], requires_grad=True)  #initial and requirea_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
```

```
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. 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, the new gradient will accumulated (added) to the old one.

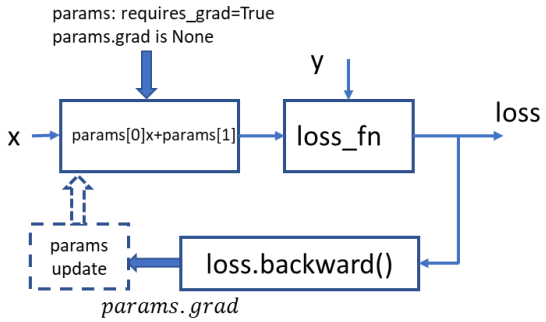


Fig. 5 autograd computation chart

Zero gradient

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

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

2) Linear regression using autograd

Now, we are ready to apply autograd to our previous linear regression project.

```
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

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)
```

The results are shown below, which are very close to Fig.4 (but not exactly the same because different computation approaches).

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

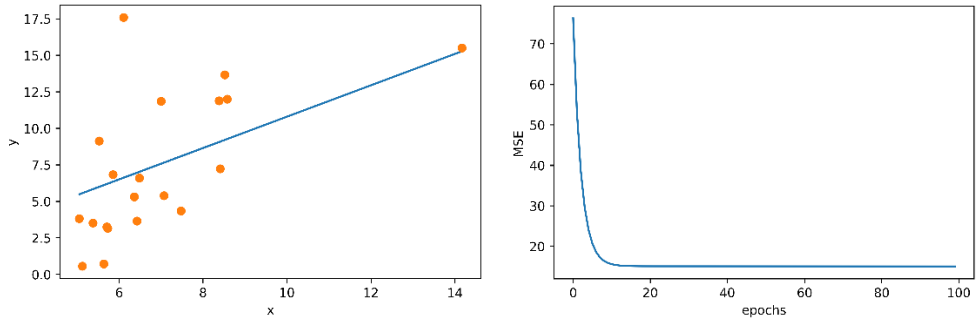


Fig. 6 Results of linear regression using PyTorch autograd

10.5.4 Linear regression using autograd and optim

To further take advantage of PyTorch, we can instantiate PyTorch optimization module for parameter updating. In chapter 8 and chapter 9, 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 algorithm 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.

```
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
```

```
params = torch.tensor([0.0, 0.0], requires_grad=True)
learning_rate = 0.002
optimizer = optim.SGD([params], lr=learning_rate)             #define a optimizer
params,  loss_tensor=  training_loop_autograd_optim(n_epochs  =  100,optimizer  =
optimizer,params = params,x = x,y = y)

params
```

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

One can try a different optimizer, for example, by changing "SGD" 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.

10.6 Neural Networks using PyTorch

The advantages of PyTorch over NumPy are not apparent in the linear regression example presented in the previous section. In this section we will taste a flavor of power of PyTorch through a little more complicated example: neural network for image classification. The implementation of deep learning neural networks in later chapters will heavily rely on using PyTorch.

10.6.1 Download dataset and transforms

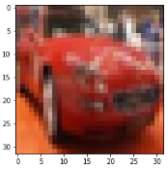
Download and load dataset

PyTorch provides a class, *torch.utils.data.Dataset*, as some popular sources for users to download. The available datasets (up to date July 2020) are: 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. 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, load the datasets and display one image as follows.

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

img, label = cifar10[5]
plt.imshow(img)
plt.show()
```



The description of the dataset class at https://pytorch.org is pasted here.

CLASS `torchvision.datasets.CIFAR10(root, train=True, transform=None, target_transform=None, download=False)`

CIFAR10 Dataset.

Parameters

- **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.

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 subsequent neural network. 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[5]
img_t.shape, label
```

```
(torch.Size([3, 32, 32]), 1)
```

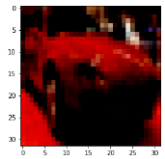
```
plt.imshow(img_t.permute(1, 2, 0))    # change dimension order to H,W,C
plt.show()
```

Second, we need to 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,C
plt.show()
```



Combination of download and transforms

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

```
from torchvision import datasets
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)
```

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

10.6.2 Create customized datasets

Since we will design a neural network for a binary classification task: airplane or bird, the dataset to the neural network should only include the images of either airplane (originally labeled as 0) or bird (label as 2). Now let's create our own datasets for airplanes and birds only. In the new datasets, 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. There are 10,000 images in *norm_cifar2* and 2,000 images in *norm_cifar2_test* with 50% of each class.

10.6.3 Build model

PyTorch torch.nn provides classes for almost all neural network building blocks. The following codes define a neural network with one 512-unit hidden layer and 2-unit output layer.

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

loss_fn = nn.NLLLoss()
```

nn.Linear is used to define a linear operation before activation in each layer. An instance, nn.Linear(3072, 512), accepts input tensor with size (N, *, 3072), and delivers output tensor with size (N,*, 512). (Note * indicates possible additional dimension). 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 size of (batch_size, 3072).

nn.Tanh() defines the activation for the hidden layer. nn.LogSoftmax(dim=1)

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.

10.6.4 Training loop

Now let's put all things together and train the neural network. First we need to prepare training data in minibatches. 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

```
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, 512),
    nn.Tanh(),
    nn.Linear(512, 2),
    nn.LogSoftmax(dim=1))

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 imgs, labels in train_loader:
        batch_size = imgs.shape[0]     #64, except the last one
        outputs = model(imgs.view(batch_size, -1))
        print(imgs.shape)                        # shape: [64,3,32,32]
        print(imgs.view(batch_size, -1).shape)  #shape:[64,3072]
        print(outputs.shape)                  # shape: [64,2]    except the last one
```

```
        print(labels.shape)     #[64] except the last, integer 0 or 1.
        loss = loss_fn(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

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

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.

```
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)) # shape:[batch_size,2]
        _, predicted = torch.max(outputs, dim=1)   # predicted shape [64]
        total += labels.shape[0]
        correct += int((predicted == labels).sum())
print("Accuracy: %f" % (correct / total))
```

```
Accuracy: 0.805000
```

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). The following codes show the neural network using nn.CrossEntropyLoss. The model can be tested by the same testing code above (note that the numerical result may be minorly different, for example, in this case, we got accuracy of 0.809500)

```
import torch
import torch.nn as nn
train_loader = torch.utils.data.DataLoader(norm_cifar2, batch_size=64,
shuffle=True)
model = nn.Sequential(
    nn.Linear(3072, 512),
    nn.Tanh(),
    nn.Linear(512, 2)
   # nn.LogSoftmax(dim=1)
 )
learning_rate = 1e-2
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
#loss_fn = nn.NLLLoss()
loss_fn = nn.CrossEntropyLoss()
n_epochs = 10
for epoch in range(n_epochs):
```

```
    for imgs, labels in train_loader:
        batch_size = imgs.shape[0]
        outputs = model(imgs.view(batch_size, -1))
        loss = loss_fn(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

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

It is very convenient to implement a neural network with a different size and architecture by modifying the model configuration, without changing anything else. For example, the following model implements a network with three layers and ReLU activation for hidden layers. The accuracy of the trained model on the test set is 0.833500.

```
model = nn.Sequential(
    nn.Linear(3072,25),
    nn.ReLU(),
    nn.Linear(25,12),
    nn.ReLU(),
    nn.Linear(12,2)
 )
```

10.6.5 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)
)>
```

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() if p.requires_grad == True]
sum(numel_list), numel_list
```

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

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)
```

(output)

```
0.weight torch.Size([25, 3072])
tensor([[-0.0144, -0.0058,  0.0187,  ..., -0.0130,  0.0038, -0.0028],
        [-0.0063,  0.0008,  0.0160,  ..., -0.0055,  0.0015, -0.0197],
        [-0.0096,  0.0060,  0.0196,  ..., -0.0143,  0.0119,  0.0094],
        ...,
        [-0.0050, -0.0146, -0.0180,  ..., -0.0051, -0.0141, -0.0145],
        [ 0.0072, -0.0140,  0.0082,  ...,  0.0100, -0.0035, -0.0147],
        [ 0.0139, -0.0179,  0.0013,  ..., -0.0052, -0.0066,  0.0149]])
0.bias torch.Size([25])
tensor([-0.0125,  0.0230,  0.0304,  0.0456,  0.0533, -0.0206,  0.0061, -0.0027,
         0.0185, -0.0043,  0.0203,  0.0412, -0.0060,  0.0118, -0.0064, -0.0212,
         0.0275,  0.0165, -0.0081,  0.0326, -0.0047,  0.0393,  0.0363,  0.0180,
         0.0436])
2.weight torch.Size([12, 25])
tensor([[ 0.1484, -0.1733,  0.0122,  0.0654, -0.0764,  0.1092, -0.1770, -0.1628,
         -0.1718, -0.0217, -0.1092, -0.0356, -0.0770, -0.0565, -0.0049, -0.0336,
          0.2019,  0.2010,  0.0689,  0.2255,  0.1301,  0.2127, -0.1775,  0.2244,
         -0.1374],
        [ 0.0442, -0.0958,  0.0450, -0.0476, -0.0904,  0.2145,  0.1287, -0.0100,
          0.2784,  0.1010, -0.0165, -0.2137, -0.1817,  0.1668, -0.1813, -0.1567,
          0.1639, -0.0049, -0.0680,  0.2274,  0.1573,  0.1645,  0.1153, -0.1248,
          0.0104],
        [ 0.0399, -0.1887,  0.1152, -0.0822,  0.0974, -0.0757,  0.1783, -0.1870,
         -0.1392,  0.0318, -0.0292, -0.1867, -0.1854,  0.0526, -0.0044,  0.0693,
         -0.1783, -0.0296,  0.1449, -0.0554,  0.1251, -0.1393,  0.0849, -0.1485,
          0.0348],
        [ 0.0291,  0.0328,  0.0498, -0.2116, -0.0592, -0.0847,  0.1746,  0.1958,
          0.2107,  0.0582,  0.2015,  0.0184, -0.0383, -0.0500,  0.1960,  0.1780,
         -0.0938,  0.2136, -0.1026,  0.1720, -0.0147,  0.2472,  0.2523,  0.2792,
         -0.1473],
        [-0.1168,  0.0735,  0.2039,  0.2862,  0.2498, -0.1381, -0.1758,  0.0138,
         -0.0581, -0.0543,  0.0493,  0.2211,  0.0789,  0.1152, -0.1884, -0.1932,
          0.1926,  0.1413, -0.0772,  0.1773,  0.2434, -0.2375,  0.0176, -0.2105,
          0.1594],
        [ 0.1687, -0.0887,  0.0003,  0.0650, -0.0623,  0.0599, -0.0952, -0.1325,
         -0.1550,  0.1603,  0.1117, -0.1380, -0.1544,  0.1746, -0.1949, -0.1178,
          0.0801, -0.1383, -0.0836,  0.0650,  0.1162, -0.0166,  0.0030,  0.2013,
          0.1224],
        [-0.1160, -0.0988, -0.1081, -0.1787, -0.0358, -0.1449,  0.1422, -0.1903,
         -0.0531, -0.0052, -0.1967,  0.0043,  0.1849,  0.1545,  0.0592, -0.1401,
         -0.0536, -0.0940, -0.1819,  0.0622, -0.0846, -0.0992, -0.1136, -0.1946,
         -0.1574],
        [ 0.0174,  0.1869, -0.0388,  0.0869,  0.1488, -0.1485, -0.1963, -0.1648,
         -0.0080, -0.1716, -0.0353,  0.2972,  0.0854, -0.0762, -0.1463, -0.0580,
         -0.0792, -0.0407, -0.1087, -0.1454, -0.1177, -0.0706,  0.1998,  0.1154,
          0.2943],
        [-0.1891,  0.1618,  0.0288, -0.1135,  0.1622, -0.1020, -0.1337, -0.0465,
         -0.0158,  0.0248, -0.0635, -0.1705, -0.0573,  0.1764,  0.0265,  0.1409,
         -0.0451, -0.0838, -0.0678,  0.1163, -0.1627,  0.0888,  0.0102, -0.1063,
          0.1827],
        [ 0.1185,  0.0703,  0.1008,  0.1690,  0.1030, -0.0219, -0.1716, -0.1627,
         -0.1250, -0.0190, -0.1324, -0.0858,  0.1226,  0.1560,  0.1017,  0.1579,
         -0.0305, -0.1351,  0.0894,  0.0231,  0.1892,  0.1760,  0.0460, -0.0711,
          0.0994],
        [ 0.0488,  0.0425, -0.1309, -0.1435, -0.1560, -0.0118,  0.0051, -0.0033,
         -0.0988,  0.1181, -0.1691,  0.0294, -0.1606,  0.0701, -0.1887, -0.0840,
```

```
          0.1404, -0.0526,  0.1377, -0.0242,  0.0582, -0.0195,  0.1974,  0.1286,
         -0.0524],
        [ 0.1954,  0.0509, -0.0274, -0.1040,  0.0500,  0.1710, -0.1665, -0.1139,
          0.1118,  0.1905,  0.0564, -0.0564, -0.1100, -0.0449,  0.1502,  0.1713,
          0.0035,  0.1009, -0.0756, -0.0780, -0.0535,  0.0189,  0.0350, -0.1388,
         -0.1024]])
2.bias torch.Size([12])
tensor([-0.0260, -0.0121,  0.1049,  0.0013,  0.2123, -0.0391,  0.1478,  0.1923,
        -0.0756,  0.0682, -0.0605, -0.0390])
4.weight torch.Size([2, 12])
tensor([[ 0.1281,  0.3899,  0.0056,  0.4693, -0.4253, -0.0672,  0.2201, -0.4079,
         -0.2819, -0.1939, -0.2666, -0.1323],
        [-0.1715, -0.2017,  0.2118, -0.4644,  0.5111, -0.1403,  0.0249,  0.2868,
         -0.2329,  0.0316, -0.2240, -0.1265]])
4.bias torch.Size([2]) tensor([-0.1318, -0.0004])
```

10.7 An Example by PyTorch: Finger Signs

In this section, we will show how easily one can implement the neural network (in chapter 9) for finger sign recognition by PyTorch. 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.

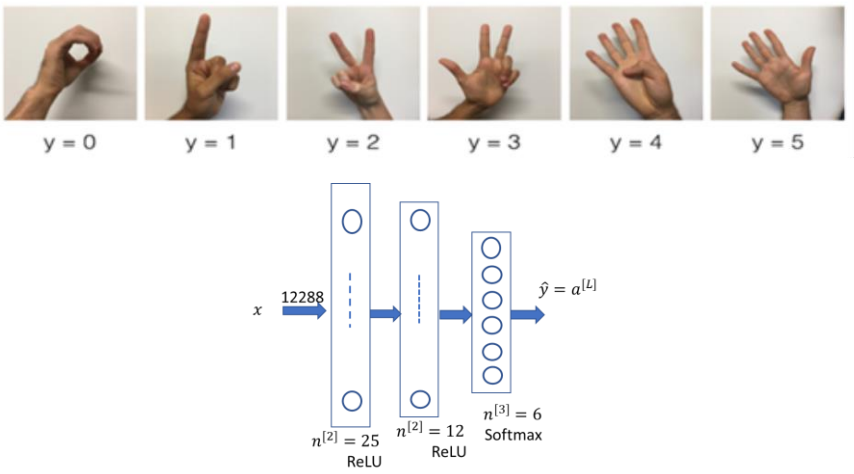


Fig.7 neural network for finger sign recognition

If the datasets are not originated using "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 this case, since the datasets are stored in *.h5 files, an effort is needed to generate the corresponding dataset class.

10.7.2 Implementation without regularization

```
#---------- import packages

import torch
from torch.utils.data import Dataset, DataLoader
import torch.nn as nn
```

```
import torch.nn.functional as F
import torchvision
from torchvision import transforms, datasets
from torch import optim
from torch.autograd import Variable
from torch.utils import data

import h5py
import numpy as np
import matplotlib.pyplot as plt

torch.manual_seed(0)



#--------------define Dataset for train set
class signs_train_Dataset(Dataset):

def __init__(self,root_dir,filename, transforms):


    self.transform = transforms
    self.root_dir = root_dir

    #self.train_dataset = h5py.File(self.root_dir+'/train_signs.h5', "r")
    self.train_dataset = h5py.File(self.root_dir+filename, "r")
    self.train_set_x_orig = np.array(self.train_dataset["train_set_x"][:]) # train set features
    self.train_set_y_orig = np.array(self.train_dataset["train_set_y"][:]) # train set labels
    self.train_set_y_orig = self.train_set_y_orig.reshape((1, self.train_set_y_orig.shape[0]))

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

def __getitem__(self, index):
    image = self.train_set_x_orig[index]
    self.train_set_y_orig = self.train_set_y_orig.reshape(self.train_set_x_orig.shape[0])
    y = self.train_set_y_orig[index]

    if self.transform:
        image = self.transform(image)


    y = torch.tensor(y)
    return [image, y]

#--------------define Dataset for test set in the same way
class signs_test_Dataset(Dataset):

def __init__(self,root_dir,filename, transforms):


    self.transform = transforms
    self.root_dir = root_dir

    self.test_dataset = h5py.File(self.root_dir+filename, "r")
    self.test_set_x_orig = np.array(self.test_dataset["test_set_x"][:]) # your test set features
    self.test_set_y_orig = np.array(self.test_dataset["test_set_y"][:]) # your test set labels
    self.test_set_y_orig = self.test_set_y_orig.reshape((1, self.test_set_y_orig.shape[0]))

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

def __getitem__(self, index):
    image = self.test_set_x_orig[index]
    self.test_set_y_orig = self.test_set_y_orig.reshape(self.test_set_x_orig.shape[0])
    y = self.test_set_y_orig[index]

    if self.transform:
        image = self.transform(image)
```

```
    y = torch.tensor(y)
    return [image, y]

#----- generate train_dataset and test_dataset, and their dataloaders

train_dataset = signs_train_Dataset("C:/Users/weido/ch11/",'train_signs.h5', transforms=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', transforms=transforms.Co
    mpose([transforms.ToTensor(),transforms.Normalize((0.5,0.5,0.5), (0.5, 0.5, 0.5))]))

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

test_dataloader = DataLoader(test_dataset, batch_size=32,
                        shuffle=True)

#------ define nn.sequential model

model = nn.Sequential(
    nn.Linear(12288,25),
    nn.ReLU(),
    nn.Linear(25,12),
    nn.ReLU(),
    nn.Linear(12,6)
 )

#---- training loop

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

loss_fn = nn.CrossEntropyLoss()
n_epochs = 1000
for epoch in range(n_epochs):
    for imgs, labels in train_dataloader:   # imgs:[batch_size, 3, 64,64], labels:[batch_size]
        batch_size = imgs.shape[0]
        #print(labels.shape)
        outputs = model(imgs.view(batch_size, -1)) # model requires input:[batch_size, 12288]
        loss = loss_fn(outputs, labels) # outputs:[batch_size,6], labels:[batch_size], loss:scalar
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    if epoch % 10==0:
        print("Epoch: %d, Loss: %f" % (epoch, float(loss)))
```

Epoch: 0, Loss: 1.816068
Epoch: 10, Loss: 1.729996
Epoch: 20, Loss: 1.603678
Epoch: 30, Loss: 1.436398
Epoch: 40, Loss: 1.276127
....
Epoch: 980, Loss: 0.009608
Epoch: 990, Loss: 0.005490

```
#-------test loop

correct = 0
total = 0
with torch.no_grad():
    for imgs, labels in test_dataloader:
        batch_size = imgs.shape[0]
        outputs = model(imgs.view(batch_size, -1))
        values, predicted = torch.max(outputs, dim=1)
        print(imgs.shape, labels.shape, predicted.shape)
        #print(labels, predicted)
```

```
        total += labels.shape[0]
        correct += int((predicted == labels.sum())
print("Accuracy: %f" % (correct / total))
```

```
torch.Size([32, 3, 64, 64]) torch.Size([32]) torch.Size([32])
torch.Size([32, 3, 64, 64]) torch.Size([32]) torch.Size([32])
torch.Size([32, 3, 64, 64]) torch.Size([32]) torch.Size([32])
torch.Size([24, 3, 64, 64]) torch.Size([24]) torch.Size([24])
Accuracy: 0.883333
```

10.7.2 Implementation with regularization

If train_dataloader is used for testing, the accuracy is 1.0. This implies an overfitting on the da tasets. Regularization can be applied to improve the generalization.

1) Weight regularization

```
for epoch in range(n_epochs):
    for imgs, labels in train_dataloader:   # imgs:[batch_size, 3, 64,64], labels:[batch_size]
        batch_size = imgs.shape[0]
        #print(labels.shape)
        outputs = model(imgs.view(batch_size, -1)) # model requires input:[batch_size, 12288]
        loss = loss_fn(outputs, labels) # outputs:[batch_size,6], labels:[batch_size], loss:scalar
        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)))
```

The accuracy of testing set is 0.9000, and the accuracy of training set is 1.0. Note that regularization can be increased by increasing the value of l2_lambda.

2) Dropout regularization

First, modify the model as follows, and then whenever model is instantiated, either model.train() is used for training loop or model.eval() is used for evaluation (prediction).

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

for epoch in range(n_epochs):
    for imgs, labels in train_dataloader:   # imgs:[batch_size, 3, 64,64], labels:[batch_size]
        batch_size = imgs.shape[0]
        #print(labels.shape)
        outputs = model.train()(imgs.view(batch_size, -1))

        # model requires input:[batch_size, 12288]
        loss = loss_fn(outputs, labels) # outputs:[batch_size,6], labels:[batch_size], loss:scalar
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    if epoch % 10==0:
        print("Epoch: %d, Loss: %f" % (epoch, float(loss)))

…
```

```
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)
        print(imgs.shape, labels.shape, predicted.shape)
        #print(labels, predicted)
        total += labels.shape[0]
        correct += int((predicted == labels.sum())
print("Accuracy: %f" % (correct / total))
```

The accuracy of testing set is 0.900, and the accuracy of training set is 1.0. Note that regularization can be increased by increasing the value of p in Droput().

Summary

This chapter presents the basics of PyTorch, and demonstrate the major steps of developing a machine learning algorithm using PyTorch, with an emphasis on neural networks. 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.

Specifically, it is essential to understand and use the following statements:

```
torch.utils.data.Dataset:  #download dataset with optional transforms, path, train/test set

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

for img,label in norm_cifar10                          # iteration on dataset

torch.utils.data.DataLoader
# load data in batches with arguments: dataset, batch size, shuffle option.
# return a batched dataset

for imgs, labels in train_loader:
# iterate on dataset in batches
# imgs shape [batch_size, C, H,W]
# labels shape [batch_size]

imgs.view(batch_size, -1)          # reshape to [batch_size, CxHxW] for model
```

It is easy to define a model using nn.Sequential() and its loss function using nn.CrossEntropyLoss().

```
model = nn.Sequential(
    nn.Linear(3072, 512),
    nn.Tanh(),
    nn.Linear(512, 2),
 )
loss_fn = nn.CrossEntropyLoss()
```

During the training process, we don't need to compute the gradients and update parameters explicitly. The combination of the following statements is typically used in the training loop.

```
loss = loss_fn(outputs, labels)
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

PyTorch is a powerful tool for us to implement classical deep learning algorithms in the subsequent chapters.

References

[1] Chapters 1 to 7, "Deep Learning with PyTorch", Eli Stevens, Luca Antiga, Thomas Viehmann, Foreword by Soumith Chintala. 2020 by Manning Publications Co.

[2] https://pytorch.org/

Exercises

1. Do linear regression on the dataset: ex1data1.csv (chapter 3) using both PyTorch autograd and optimizer.

2. 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.

3. 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.