PyTorch神经网络建模：

- 数据准备
- 模型建立
- 模型训练
- 模型评估使用和保存

数据准备

数据准备是PyTorch网络中非常重要且比较困难的一个部分，实际任务中一般会面临结构化数据、图片数据、文本数据和时间序列数据，不同的数据会有不同的数据准备方法。

本文总结针对这四种数据进行准备，每种数据又有哪些不同的建模方法。

Dataset与DataLoader

PyTorch通常使用Dataset和DataLoader这两个工具类构建数据管道，建模各种数据避不开这两个工具类。

- Dataset定义数据集的内容，类似于列表的数据结构，具有确定的长度，能够用索引获取数据集中的元素。
    - 绝大多数情况下，只需实现Dataset的__len__方法和__getitem__方法，就可以轻松构建自己的数据集，并用默认数据管道进行加载
- DataLoader定义按batch加载数据集的方法，实现一个__iter__方法的可迭代对象，每次迭代输出一个batch数据。
    - DataLoader可以控制batch大小，batch中元素的采样方法，以及将batch结果整理成模型所需输入形式的方法，同时可以多进程读取数据。

概述

通过这两个工具类完成数据管道的构建，有两个问题：

- 如何获取一个batch？
- Dataset与DataLoader是如何分工合作的？

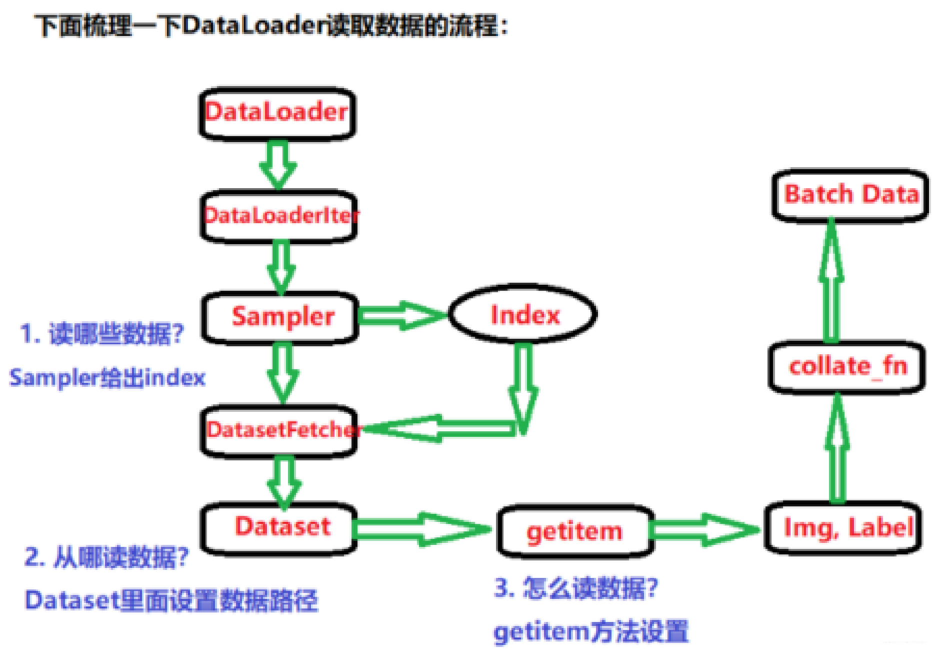
如何获取一个batch

数据集特征X，标签Y，则数据集可以表示成（X，Y），batch大小为m，则获取一个batch步骤：

- 确定数据集的长度n
    - 一共有多少个样本，这样指定每个batch时，计算机可以规划分为几个batch
- 从[0, n-1]范围内抽样m个数（batch大小）
    - 假设m=4，则拿到的结果就是一个列表，类似：indics=[1, 4, 8, 9]
- 根据下表从数据集中取m个数对应的下标元素
    - 拿到一个元组列表，类似：samples=[(X[1], Y[1]), (X[4], Y[4]), (X[8], Y[8]), (X[9], Y[9])]
- 将结果整理成两个张量作为输出：
    - 结果时两个张量，类似：batch=(features, labels)，其中：
    - features=torch.stack(X[1], X[4], X[8], X[9])
    - labels=torch.stack(Y[1], Y[4], Y[8], Y[9])

这样，就完成了一个batch数据获取操作。

Dataset与DataLoader功能分工



上述流程图，把DataLoader读取数据的流程梳理一遍。

DataLoader的作用就是构建一个数据装载器，根据提供的batch_size大小，将数据样本分成一个个batch去训练模型，这个分的过程中需要把数据取到（需要借助Dataset的getitem）：

- 第一个步骤：数据的总长度，这个需要在Dataset的__len__方法中告诉计算机
- 第二个步骤：0到n-1范围中抽样出m个数的方法，由DataLoader的sampler和batch_sampler参数指定：
  - sampler参数指定单个元素抽样方法（一般无需设置），程序默认在DataLoader的参数shuffle=True时采用随机抽样，shuffle=False时采用顺序抽样。
  - batch_sampler参数将多个抽样的元素整理成一个列表（一般无需设置），默认方法在DataLoader的参数drop_last=True时会丢弃数据集最后一个长度不能被batch大小整除的批次，在drop_last=False时保留最后一个批次。
- 第三个步骤：根据下标取数据集中的元素（需要自己写读取函数），由Dataset的__getitem__方法实现，这个函数接收的参数是一个索引。
- 第四个步骤：逻辑由DataLoader的参数collate_fn指定，一般无需设置。

使用

Dataset创建数据集

Dataset核心接口逻辑伪代码

In [1]:
```
class Dataset(object):
    def __init__(self):
        pass

    def __len__(self):
        raise NotImplementedError

    def __getitem__(self,index):
        raise NotImplementedError
```

Dataset创建数据集常用方法：

- torch.utils.data.TensorDataset：根据Tensor创建数据集（Numpy的array和Pandas的DataFrame需要先转换成Tensor）

- torchvision.datasets.ImageFolder：根据图片目录创建图片数据集
- torch.utils.data.Dataset：创建自定义数据集（需要实现len和getitem方法）
- torch.utils.data.random_split：将一个数据集分割成多份（训练集，验证集，测试机）
- 调用Dataset加法运算符（+）将多个数据集合并成一个数据集

DataLoader加载数据集

DataLoader逻辑接口代码

In [2]:
```python
class DataLoader(object):
    def __init__(self,dataset,batch_size,collate_fn,shuffle = True,drop_last = False):
        self.dataset = dataset
        self.sampler =torch.utils.data.RandomSampler if shuffle else \
           torch.utils.data.SequentialSampler
        self.batch_sampler = torch.utils.data.BatchSampler
        self.sample_iter = self.batch_sampler(
            self.sampler(range(len(dataset))),
            batch_size = batch_size,drop_last = drop_last)

    def __next__(self):
        indices = next(self.sample_iter)
        batch = self.collate_fn([self.dataset[i] for i in indices])
        return batch
```

DataLoader能够控制batch的大小，batch中元素的采样方法，以及将batch结果整理成模型所输入形式的方法，并且能够使用多进程读取数据。

DataLoader的函数签名如下：

```
DataLoader(
    dataset,
    batch_size=1,
    shuffle=False,
    sampler=None,
    batch_sampler=None,
    num_workers=0,
    collate_fn=None,
    pin_memory=False,
    drop_last=False,
    timeout=0,
    worker_init_fn=None,
    multiprocessing_context=None,
)
```

通常仅配置dataset, batch_size, shuffle, num_workers, drop_last五个参数，其他参数使用默认值即可。

DataLoader除了可以加载torch.utils.data.Dataset外，还可以加载另外一种数据集：

- torch.utils.data.IterableDataset
- Dataset数据集相当于一种列表结构，IterableDataset相当于一种迭代器结构，其更加复杂，一般较少使用。
- 参数说明：
    - dataset：数据集
    - batch_size：批次大小
    - shuffle：是否乱序
    - sampler：样本采样函数，一般无需设置
    - batch_sampler：批次采样函数，一般无需设置

- num_workers：使用多进程读取数据，设置的进程数
- collate_fn：整理一个批次数据的函数。
- pin_memory：是否设置为锁业内存。默认False，锁业内存不会使用虚拟内存(硬盘)，从锁业内存拷贝到GPU上速度会更快。
- drop_last：是否丢弃最后一个样本数量不足batch_size批次数据。
- timeout：加载一个数据批次的最长等待时间，一般无需设置。
- worker_init_fn：每个worker中dataset的初始化函数，常用于IterableDataset。

In [3]:
```python
import torch.utils.data as tud
import torch

#构建输入数据管道
ds = tud.TensorDataset(torch.arange(1,50))
dl = tud.DataLoader(ds,
                batch_size = 10,
                shuffle= True,
                num_workers=2,
                drop_last = True)
#迭代数据
for batch, in dl:
    print(batch)
```

```
tensor([49, 14, 33,  8, 22, 29, 28, 20, 27, 19])
tensor([ 7, 26, 25, 48,  6, 36,  5, 12, 45, 46])
tensor([ 9, 13, 17,  1, 24, 39, 11, 15, 16, 32])
tensor([47, 44, 21, 31, 23, 34, 35, 43,  4, 38])
```

结构化数据建模

将结构化数据封装成DataLoader形式，形成可迭代的数据管道，模型训练时，可以遍历DataLoader对象去取数据。

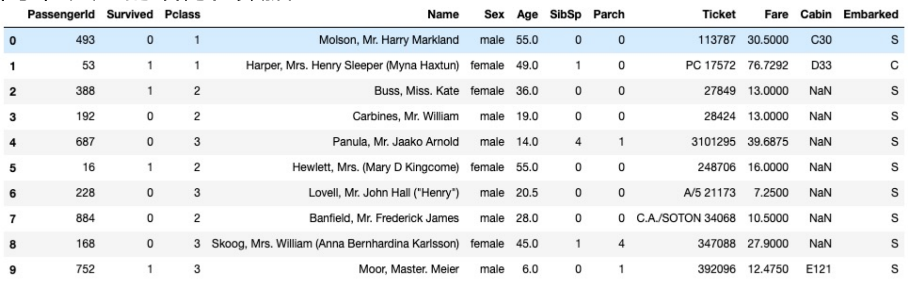
结构化数据有三种建模方式

直接用TensorDataset和DataLoader封装成可迭代的数据管道

只要把结构化的数据预处理完毕，做成训练集和测试集之后，直接拿这两个函数进行封装

In [4]:
```python
import pandas as pd

# 首先， 读入数据
dftrain_raw = pd.read_csv('data/titanic/train.csv')
dftest_raw = pd.read_csv('data/titanic/test.csv')
```

可以看到数据的样子，典型的结构化数据：



	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	493	0	1	Molson, Mr. Harry Markland	male	55.0	0	0	113787	30.5000	C30	S
1	53	1	1	Harper, Mrs. Henry Sleeper (Myna Haxtun)	female	49.0	1	0	PC 17572	76.7292	D33	C
2	388	1	2	Buss, Miss. Kate	female	36.0	0	0	27849	13.0000	NaN	S
3	192	0	2	Carbines, Mr. William	male	19.0	0	0	28424	13.0000	NaN	S
4	687	0	3	Panula, Mr. Jaako Arnold	male	14.0	4	1	3101295	39.6875	NaN	S
5	16	1	2	Hewlett, Mrs. (Mary D Kingcome)	female	55.0	0	0	248706	16.0000	NaN	S
6	228	0	3	Lovell, Mr. John Hall ("Henry")	male	20.5	0	0	A/5 21173	7.2500	NaN	S
7	884	0	2	Banfield, Mr. Frederick James	male	28.0	0	0	C.A./SOTON 34068	10.5000	NaN	S
8	168	0	3	Skoog, Mrs. William (Anna Bernhardina Karlsson)	female	45.0	1	4	347088	27.9000	NaN	S
9	752	1	3	Moor, Master. Meier	male	6.0	0	1	392096	12.4750	E121	S

进行简单的数据预处理，划分出训练集和测试集

In [5]:
```python
def preprocessing(dfdata):
    dfresult= pd.DataFrame()
    #Pclass
    dfPclass = pd.get_dummies(dfdata['Pclass'])
    dfPclass.columns = ['Pclass_' +str(x) for x in dfPclass.columns ]
    dfresult = pd.concat([dfresult,dfPclass],axis = 1)

    #Embarked
    dfEmbarked = pd.get_dummies(dfdata['Embarked'],dummy_na=True)
    dfEmbarked.columns = ['Embarked_' + str(x) for x in dfEmbarked.columns]
    dfresult = pd.concat([dfresult,dfEmbarked],axis = 1)

    return(dfresult)

x_train = preprocessing(dftrain_raw).values   # x_train.shape = (712, 15)
y_train = dftrain_raw.Survived.values     # y_train.shape = (712, 1)

x_test = preprocessing(dftest_raw).values   # x_test.shape = (179, 15)
y_test = dftest_raw.Survived.values     # y_test.shape = (179, 1)
```

数据处理完毕后，可以发现训练集样本是712,特征数是15，之后就可以构建数据管道了

In [8]:
```python
dl_train = tud.DataLoader(tud.TensorDataset(torch.tensor(x_train).float(),torch.tensor(y_train).fl
                     shuffle = True, batch_size = 8)
dl_valid = tud.DataLoader(tud.TensorDataset(torch.tensor(x_test).float(),torch.tensor(y_test).floa
                     shuffle = False, batch_size = 8)

# 我们可以测试一下数据管道
for features,labels in dl_train:
    print(features,labels)
    break
```

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

自己构造数据管道迭代器，不通过Pytorch提供的接口

自己写如何划分数据，首先获取全部数据的个数，然后生成索引，根据提供的batch大小把索引划分开，返回对应的数据

In [12]:
```python
import numpy as np

# 样本数量
n = 400

# 生成测试用数据集
X = 10 * torch.rand([n, 2]) - 5.0  #torch.rand是均匀分布
w0 = torch.tensor([[2.0], [-3.0]])
b0 = torch.tensor([[10.0]])
Y = X @ w0 + b0 + torch.normal(0.0, 2.0, size = [n, 1])  # @表示矩阵乘法,增加正态扰动
```

数据集有400个样本，2个特征，构建数据管道

In [13]:
```
def data_iter(features, labels, batch_size=8):
    num_examples = len(features)
    indices = list(range(num_examples))
    np.random.shuffle(indices)  # 样本的读取顺序是随机的
    for i in range(0, num_examples, batch_size):
        indexs = torch.LongTensor(indices[i: min(i + batch_size, num_examples)])
        yield  features.index_select(0, indexs), labels.index_select(0, indexs)
```

模型训练的时候，直接遍历即可

In [14]:
```
for feature, label in data_iter(X, Y, batch_size=8):
    print(feature)
    print(label)
    break
```

```
tensor([[-1.4392, -4.0170],
        [ 2.3910,  2.2930],
        [ 3.6774,  2.5477],
        [ 1.5018,  0.4600],
        [ 2.1547,  0.6044],
        [ 4.5885, -0.9907],
        [ 4.5781,  4.5254],
        [-2.4320,  2.4013]])
tensor([[19.4501],
        [ 9.5968],
        [12.1262],
        [13.4225],
        [11.8993],
        [22.1154],
        [ 7.3442],
        [-0.8488]])
```

时序数据建模

时序数据建模需要自定义数据集，涉及自回归问题，也就是用到了历史的数据作为特征。

这种数据一般会采用滑动窗口把原数据集进行一个切割获得X和Y。

下面这个数据集是中国2020年3月之前的疫情数据，典型的时间序列数据：

	confirmed_num	cured_num	dead_num
0	457.0	4.0	16.0
1	688.0	11.0	15.0
2	769.0	2.0	24.0
3	1771.0	9.0	26.0
4	1459.0	43.0	26.0

特征是确诊人数，治愈人数和死亡人数，预测是接下来的确诊人数，治愈人数和死亡人数。（自回归）

通过继承torch.utils.data.Dataset实现自定义事件序列数据集：

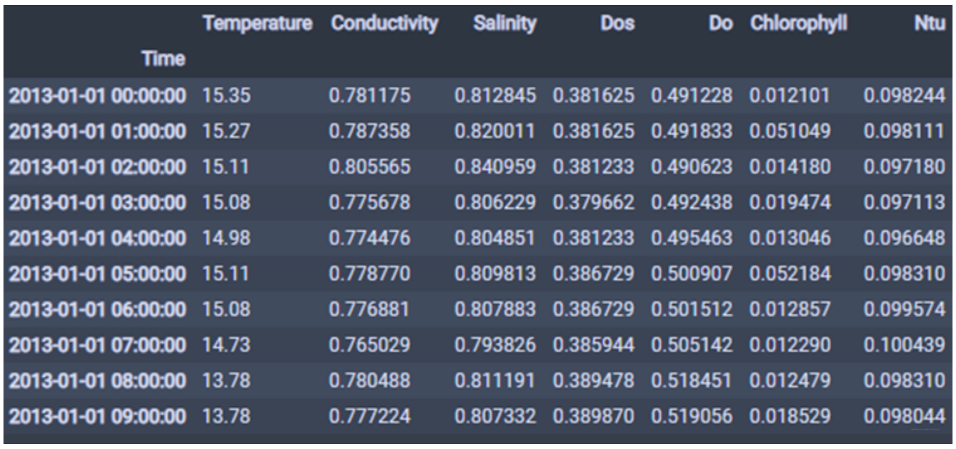
- __len__：实现len(dataset)返回整个数据集的大小
- __getitem__：获取一些索引数据，使用dataset[i]返回数据集中的第i个样本

必须要覆盖这两个方法。

In [17]:
```python
# 用某日前8天窗口数据作为输入预测该日数据
WINDOW_SIZE = 8

class Covid19Dataset(tud.Dataset):
    def __len__(self):
        return len(dfdiff) - WINDOW_SIZE

    def __getitem__(self,i):
        x = dfdiff.loc[i:i+WINDOW_SIZE-1,:]
        feature = torch.tensor(x.values)
        y = dfdiff.loc[i+WINDOW_SIZE,:]
        label = torch.tensor(y.values)
        return (feature,label)

ds_train = Covid19Dataset()

#数据较小，可以将全部训练数据放入到一个batch中，提升性能
dl_train = tud.DataLoader(ds_train, batch_size = 38)
```

这是比较常规的时序建模方式，下面是一个非线性自回归的一个自定义数据集的方式，更加复杂一些。

非线性自回归建模：既涉及到普通的结构化数据，也涉及到时序数据。



	Temperature	Conductivity	Salinity	Dos	Do	Chlorophyll	Ntu
Time							
2013-01-01 00:00:00	15.35	0.781175	0.812845	0.381625	0.491228	0.012101	0.098244
2013-01-01 01:00:00	15.27	0.787358	0.820011	0.381625	0.491833	0.051049	0.098111
2013-01-01 02:00:00	15.11	0.805565	0.840959	0.381233	0.490623	0.014180	0.097180
2013-01-01 03:00:00	15.08	0.775678	0.806229	0.379662	0.492438	0.019474	0.097113
2013-01-01 04:00:00	14.98	0.774476	0.804851	0.381233	0.495463	0.013046	0.096648
2013-01-01 05:00:00	15.11	0.778770	0.809813	0.386729	0.500907	0.052184	0.098310
2013-01-01 06:00:00	15.08	0.776881	0.807883	0.386729	0.501512	0.012857	0.099574
2013-01-01 07:00:00	14.73	0.765029	0.793826	0.385944	0.505142	0.012290	0.100439
2013-01-01 08:00:00	13.78	0.780488	0.811191	0.389478	0.518451	0.012479	0.098310
2013-01-01 09:00:00	13.78	0.777224	0.807332	0.389870	0.519056	0.018529	0.098044

要预测未来7天的温度，既涉及到温度本身的特征，也涉及到其他特征来影响温度，问题的建模思路与上面一致，基于过去一段时间的所有特征，但是写法与上面不同，不是直接一个滑动窗口去切，而是先把这些滞后特征做成一行，然后再进行shape转换。

这个比较复杂的就是滞后特征的提取，这个灵活性更高，可以在series_to_supervised函数里做各种处理工作。

```
In  [18]: class TemperatureDataSet(Dataset):
              def __init__(self, n_in, n_out):
                  """
                  函数用途： 加载数据并且初始化
                  参数说明：
                      n_in: 用多长时间进行预测
                      n_out: 预测多长时间
                  """
                  # 读取数据，去除缺失严重属性，插值
                  df = pd.read_csv("dsw.csv", index_col=0)
                  df.drop(['pH'], axis=1, inplace=True)
                  df.index = pd.to_datetime(df.index)
                  df.interpolate(method='time', inplace=True)  # 时间插值

                  # 标准化
                  self.ss = StandardScaler()
                  self.std_data = self.ss.fit_transform(df['Temperature'].values.reshape(-1, 1))

                  # 转化为监督测试集
                  data = self.series_to_supervised(self.std_data, n_in, n_out)
                  re_data = self.series_to_supervised(df['Temperature'].values.reshape(-1, 1), n_in, n_out)
                  self.x = torch.from_numpy(data.iloc[:, :n_in].values).view(-1, n_in, 1)
                  self.y = torch.from_numpy(re_data.iloc[:, n_in:].values).view(-1, n_out, 1)
                  self.len = self.x.shape[0]

              def __len__(self):
                  return self.len

              def __getitem__(self, item):
                  return self.x[item, :].type(torch.FloatTensor), self.y[item, :].type(torch.FloatTensor)

              def series_to_supervised(self, data, n_in=1, n_out=1, dropnan=True):
                  """
                  函数用途：将时间序列转化为监督学习数据集。
                  参数说明：
                      data: 观察值序列，数据类型可以是 list 或者 NumPy array。
                      n_in: 作为输入值(X)的滞后组的数量。
                      n_out: 作为输出值(y)的观察组的数量。
                      dropnan: Boolean 值，确定是否将包含 NaN 的行移除。
                  返回值:
                      经过转换的用于监督学习的 Pandas DataFrame 序列。
                  """
                  n_vars = 1 if type(data) is list else data.shape[1]
                  df = pd.DataFrame(data)
                  cols, names = list(), list()
                  # 输入序列 (t-n, ... t-1)
                  for i in range(n_in, 0, -1):
                      cols.append(df.shift(i))
                      names += [('var%d(t-%d)' % (j + 1, i)) for j in range(n_vars)]
                  # 预测序列 (t, t+1, ... t+n)
                  for i in range(0, n_out):dsw.csv
                      cols.append(df.shift(-i))
                      if i == 0:
                          names += [('var%d(t)' % (j + 1)) for j in range(n_vars)]
                      else:
                          names += [('var%d(t+%d)' % (j + 1, i)) for j in range(n_vars)]
                  # 将所有列拼合
                  agg = pd.concat(cols, axis=1)
                  agg.columns = names
                  # drop 掉包含 NaN 的行
                  if dropnan:
                      agg.dropna(inplace=True)
                  return agg
```

```
data_set = TemperatureDataSet(n_in=10, n_out=5)
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_20084/1239793094.py in <module>
     64 
     65 
---> 66 data_set = TemperatureDataSet(n_in=10, n_out=5)

~\AppData\Local\Temp/ipykernel_20084/1239793094.py in __init__(self, n_in, n_out)
      8         """
      9         # 读取数据，去除缺失严重属性，插值
---> 10         df = pd.read_csv("dsw.csv", index_col=0)
     11         df.drop(['pH'], axis=1, inplace=True)
     12         df.index = pd.to_datetime(df.index)

~\anaconda3\envs\sEMG\lib\site-packages\pandas\io\parsers.py in read_csv(filepath_or_
buffer, sep, delimiter, header, names, index_col, usecols, squeeze, prefix, mangle_
dupe_cols, dtype, engine, converters, true_values, false_values, skipinitialspace,
 skiprows, skipfooter, nrows, na_values, keep_default_na, na_filter, verbose, skip_
blank_lines, parse_dates, infer_datetime_format, keep_date_col, date_parser, dayfir
st, cache_dates, iterator, chunksize, compression, thousands, decimal, lineterminat
or, quotechar, quoting, doublequote, escapechar, comment, encoding, dialect, error_
bad_lines, warn_bad_lines, delim_whitespace, low_memory, memory_map, float_precisio
n, storage_options)
    608     kwds.update(kwds_defaults)
    609 
--> 610     return _read(filepath_or_buffer, kwds)
    611 
    612 

~\anaconda3\envs\sEMG\lib\site-packages\pandas\io\parsers.py in _read(filepath_or_bu
ffer, kwds)
    460 
    461     # Create the parser.
--> 462     parser = TextFileReader(filepath_or_buffer, **kwds)
    463 
    464     if chunksize or iterator:

~\anaconda3\envs\sEMG\lib\site-packages\pandas\io\parsers.py in __init__(self, f, eng
ine, **kwds)
    817             self.options["has_index_names"] = kwds["has_index_names"]
    818 
--> 819         self._engine = self._make_engine(self.engine)
    820 
    821     def close(self):

~\anaconda3\envs\sEMG\lib\site-packages\pandas\io\parsers.py in _make_engine(self, en
gine)
   1048             )
   1049         # error: Too many arguments for "ParserBase"
-> 1050         return mapping[engine](self.f, **self.options)  # type: ignore[call-ar
g]
   1051 
   1052     def _failover_to_python(self):

~\anaconda3\envs\sEMG\lib\site-packages\pandas\io\parsers.py in __init__(self, src, *
*kwds)
   1865 
   1866         # open handles
-> 1867         self._open_handles(src, kwds)
   1868         assert self.handles is not None
   1869         for key in ("storage_options", "encoding", "memory_map", "compression"
):
```

```
~\anaconda3\envs\sEMG\lib\site-packages\pandas\io\parsers.py in _open_handles(self, s
rc, kwds)
   1360         Let the readers open IOHanldes after they are done with their potential raises
.
   1361         """
-> 1362         self.handles = get_handle(
   1363             src,
   1364             "r",

~\anaconda3\envs\sEMG\lib\site-packages\pandas\io\common.py in get_handle(path_or_bu
f, mode, encoding, compression, memory_map, is_text, errors, storage_options)
    640                 errors = "replace"
    641             # Encoding
--> 642             handle = open(
    643                 handle,
    644                 ioargs.mode,

FileNotFoundError: [Errno 2] No such file or directory: 'dsw.csv'
```

图片数据建模

PyTorch中构建图片数据管道通常有两种方法：

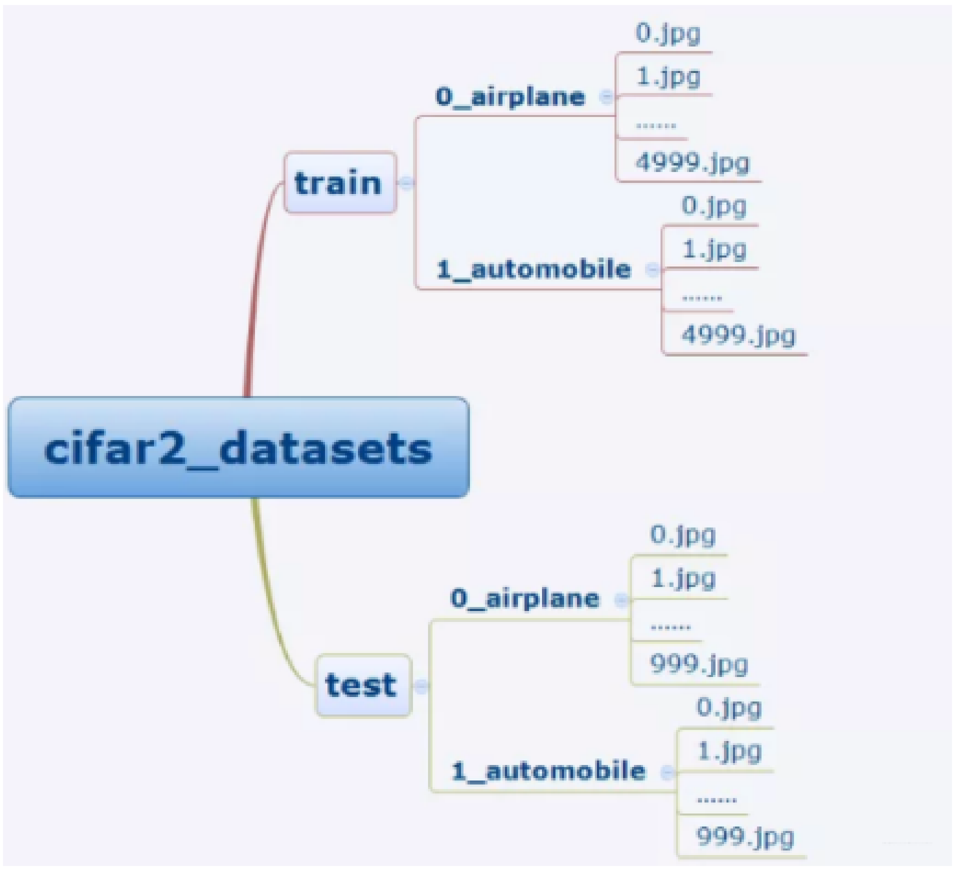
1. torchvision中的datasets.ImageFolder读取图片然后用DataLoader来并行加载。
2. 继承torch.utils.data.Dataset实现用户自定义读取逻辑然后用DataLoader并行加载

torchvision中的datasets.ImageFloder

torchvision是一个计算机视觉工具包，需要在安装PyTorch后单独安装这个包，其有三个模块：

- torchvision.transforms：常用的图像预处理方法，比如标准化、中心化、旋转、翻转等操作
- torchvision.datasets：常用的数据集dataset实现，MNIST，CIFAR-10，ImageNet等
- torchvision.models：常用模型预训练，AlexNet，VGG，ResNet，GoogLeNet

这里以CIFAR-2数据集作为示例，训练集有airplane和automobile图片各5000张，测试集照片各1000张，任务的目标是训练一个模型来对airplane和automobile进行分类。

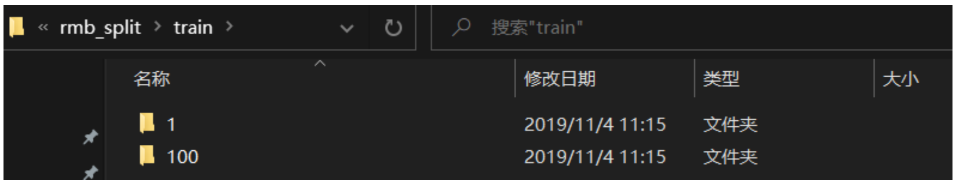


使用ImageFloder读取数据集，先定义数据的转换格式，图片数据涉及各种数据预处理，比如转成张量，裁剪，旋转，变换等操作，先定义这些操作，之后在读取数据集的时候，通过transform参数把这些处理传入，就可以直接处理。

In [ ]:
```python
# 定义预处理方式
# Compose里可以放更多的处理方式
transform_train = transforms.Compose(
    [transforms.ToTensor()])
transform_valid = transforms.Compose(
    [transforms.ToTensor()])

# 读取数据
ds_train = datasets.ImageFolder("./data/cifar2/train/",
            transform = transform_train,target_transform= lambda t:torch.tensor([t]).float())
ds_valid = datasets.ImageFolder("./data/cifar2/test/",
            transform = transform_train,target_transform= lambda t:torch.tensor([t]).float())

# 封装成数据管道
dl_train = DataLoader(ds_train,batch_size = 50,shuffle = True,num_workers=3)
dl_valid = DataLoader(ds_valid,batch_size = 50,shuffle = True,num_workers=3)
```

用户自定义读取逻辑用DataLoader加载

这里数据形式与上面一样：



名称	修改日期	类型	大小
1	2019/11/4 11:15	文件夹	
100	2019/11/4 11:15	文件夹	

类别分开，每一类里都是图片数据，首先定义一个自己写的Dataset类读取数据，生成数据集，实现逻辑依然是__getitem__里可以拿到某个索引对应的数据，注意这个函数接收的参数是一个索引，返回这个索引对应的数据。

```
In [ ]: class RMBDataset(Dataset):
            def __init__(self, data_dir, transform=None):
                """
                rmb面额分类任务的Dataset
                :param data_dir: str, 数据集所在路径
                :param transform: torch.transform，数据预处理
                """
                self.label_name = {"1": 0, "100": 1}
                self.data_info = self.get_img_info(data_dir)  # data_info存储所有图片路径和标签，在DataLoa
                self.transform = transform

            def __getitem__(self, index):
                path_img, label = self.data_info[index]
                img = Image.open(path_img).convert('RGB')     # 0~255

                if self.transform is not None:
                    img = self.transform(img)   # 在这里做transform，转为tensor等等

                return img, label

            def __len__(self):
                return len(self.data_info)

            @staticmethod
            def get_img_info(data_dir):
                data_info = list()
                for root, dirs, _ in os.walk(data_dir):
                    # 遍历类别
                    for sub_dir in dirs:
                        img_names = os.listdir(os.path.join(root, sub_dir))
                        img_names = list(filter(lambda x: x.endswith('.jpg'), img_names))

                        # 遍历图片
                        for i in range(len(img_names)):
                            img_name = img_names[i]
                            path_img = os.path.join(root, sub_dir, img_name)
                            label = rmb_label[sub_dir]
                            data_info.append((path_img, int(label)))

                return data_info
```

之后用自定义的Datasets，构建数据管道

In [ ]:
```
# transforms模块，进行数据预处理
norm_mean = [0.485, 0.456, 0.406]
norm_std = [0.229, 0.224, 0.225]

train_transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
    transforms.Normalize(norm_mean, norm_std),
])

valid_transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.ToTensor(),
    transforms.Normalize(norm_mean, norm_std),
])

## 构建MyDataset实例
train_data = RMBDataset(data_dir=train_dir, transform=train_transform)
valid_data = RMBDataset(data_dir=valid_dir, transform=valid_transform)

# 构建DataLoader
train_loader = DataLoader(dataset=train_data, batch_size=BATCH_SIZE, shuffle=True)
valid_loader = DataLoader(dataset=valid_data, batch_size=BATCH_SIZE)
```

文本数据建模

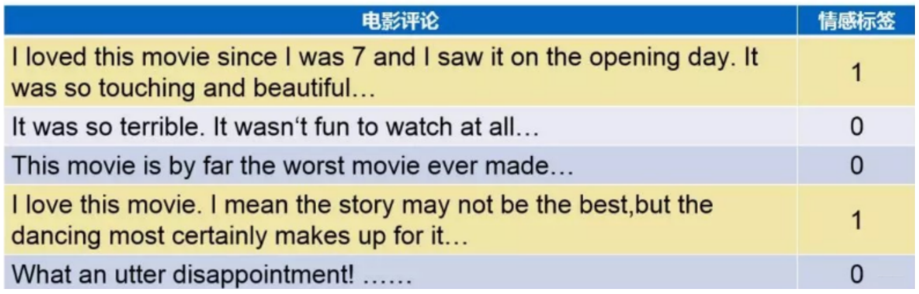
文本数据预处理较为繁琐，包括中文切词，构建词典，编码转换，序列填充，构建数据管道等等。

PyTorch中对文本数据建模一般采用两种方式：

- torchtext包：可以构建文本分类，序列标注，问答模型，机器翻译等NLP任务数据集
- 自定义Dataset，处理比较繁琐

IMDB数据集的目标是根据电影评论的文本内容预测评论的情感标签：

- 训练集有2w条电影评论文本
- 测试集有5k条电影评论文本
- 其中正负评论各占一半



电影评论	情感标签
I loved this movie since I was 7 and I saw it on the opening day. It was so touching and beautiful…	1
It was so terrible. It wasn‘t fun to watch at all…	0
This movie is by far the worst movie ever made…	0
I love this movie. I mean the story may not be the best,but the dancing most certainly makes up for it…	1
What an utter disappointment! ……	0

torchtext包进行文本分类数据建模

- torchtext.data.Example：表示一个样本，数据和标签
- torchtext.vocab.Vocab：词汇表，可以导入一些预训练词向量
- torchtext.data.Datasets：数据集类，__getitem__返回Example实例。
- torchtext.data.Field：定义字段的处理方法（文本字段，标签字段）创建 Example时的预处理，batch时的一些处理操作。
- torchtext.data.Iterator：迭代器，生成batch
- torchtext.datasets：包含了常见的数据集

操作逻辑，先从Field里定义一些字段预处理方式，然后用Datasets里的子类构建数据集，构建词典，把数据用Iterator封装成数据迭代器。

In [ ]:
```python
MAX_WORDS = 10000
MAX_LEN = 200
BATCH_SIZE = 20

# string.punctuation表示所有的标点字符， 下面这句话就是把每个句子里面的所有标点符号都替换成空字符，
tokenize = lambda x: re.sub('[%s]' % string.punctuation, "", x).split(" ")  # 字符串的替换

def filterLowFreqWords(arr, vocab):
    arr = [[x if x<MAX_WORDS else 0 for x in example] for example in arr]
    return arr

# 定义各个字段的预处理方法
# sequential告诉它输入是序列的形式， lower等于True， 转成小写  postprocessing=False这块不知道啥意思
TEXT = torchtext.data.Field(sequential=True, tokenize=tokenize, lower=True, fix_length=MAX_LEN,
LABEL = torchtext.data.Field(sequential=False, use_vocab=False)
#field在默认的情况下都期望一个输入是一组单词的序列，并且将单词映射成整数。这个映射被称为vocab。
#如果一个field已经被数字化了并且不需要被序列化，可以将参数设置为use_vocab=False以及sequential=Fals


# 构建表格型dataset
# torchtext.data.TabularDataset可读取csv, tsv, json等格式
ds_train, ds_test = torchtext.data.TabularDataset.splits(
    path='./data/imdb', train='train.tsv', test='test.tsv', format='tsv',
    fields=[('label', LABEL), ('text', TEXT)], skip_header=False
)

# 构建词典
TEXT.build_vocab(ds_train)

# 构建数据管道迭代器
train_iter, test_iter = torchtext.data.Iterator.splits(
    (ds_train, ds_test), sort_within_batch=True, sort_key=lambda x: len(x.text),
    batch_sizes=(BATCH_SIZE, BATCH_SIZE)
)
# 查看数据
for batch in train_iter:
    features = batch.text
    labels = batch.label
    print(features)
    print(features.shape)
    print(labels)
    break
```

自定义文本数据集

思路：先对训练文本分词构建词典，然后将训练集文本和测试集文本数据转换成token单词编码，之后转换成单词编码的训练集数据和测试集数据按样本分割成多个文件，一个文件代表一个样本。最后根据文件名列表获取对应序号的样本内容，从而构建Dataset数据集。

```
In [ ]: # 定义一些变量
MAX_WORDS = 10000  # 仅考虑最高频的10000个词
MAX_LEN = 200  # 每个样本保留200个词的长度
BATCH_SIZE = 20

train_data_path = 'data/imdb/train.tsv'
test_data_path = 'data/imdb/test.tsv'
train_token_path = 'data/imdb/train_token.tsv'
test_token_path =  'data/imdb/test_token.tsv'
train_samples_path = 'data/imdb/train_samples/'
test_samples_path =  'data/imdb/test_samples/'
```

我们构建词典，并保留最高频个词：

```
In [ ]: ##构建词典
word_count_dict = {}

#清洗文本
def clean_text(text):
    lowercase = text.lower().replace("\n"," ")   # 转成小写
    stripped_html = re.sub('<br />', ' ',lowercase)    # 正则修改
    cleaned_punctuation = re.sub('[%s]'%re.escape(string.punctuation),'',stripped_html)  # 去掉其
    return cleaned_punctuation

with open(train_data_path,"r",encoding = 'utf-8') as f:
    for line in f:
        label,text = line.split("\t")      # label和句子分开
        cleaned_text = clean_text(text)       # 清洗文本
        for word in cleaned_text.split(" "):
            word_count_dict[word] = word_count_dict.get(word,0)+1  # 统计单词频数

df_word_dict = pd.DataFrame(pd.Series(word_count_dict,name = "count"))
df_word_dict = df_word_dict.sort_values(by = "count",ascending =False)

df_word_dict = df_word_dict[0:MAX_WORDS-2] #
df_word_dict["word_id"] = range(2,MAX_WORDS) #编号0和1分别留给未知词<unkown>和填充<padding>

word_id_dict = df_word_dict["word_id"].to_dict()
```

利用构建好的词典，将文本转成token序号

```
In [ ]: #转换token
        # 填充文本
        def pad(data_list,pad_length):
            padded_list = data_list.copy()
            if len(data_list)> pad_length:    # 如果句子里面的单词个数比字典长度大， 那就取后面字典长度个
                 padded_list = data_list[-pad_length:]
            if len(data_list)< pad_length:   # 如果句子小， 前面填充1， 弄到字典长度大小
                 padded_list = [1]*(pad_length-len(data_list))+data_list
            return padded_list

        def text_to_token(text_file,token_file):
            with open(text_file,"r",encoding = 'utf-8') as fin,\
              open(token_file,"w",encoding = 'utf-8') as fout:
                for line in fin:
                    label,text = line.split("\t")
                    cleaned_text = clean_text(text)
                    word_token_list = [word_id_dict.get(word, 0) for word in cleaned_text.split(" ")]  #
                    pad_list = pad(word_token_list,MAX_LEN)
                    out_line = label+"\t"+" ".join([str(x) for x in pad_list])
                    fout.write(out_line+"\n")

        text_to_token(train_data_path,train_token_path)
        text_to_token(test_data_path,test_token_path)
```

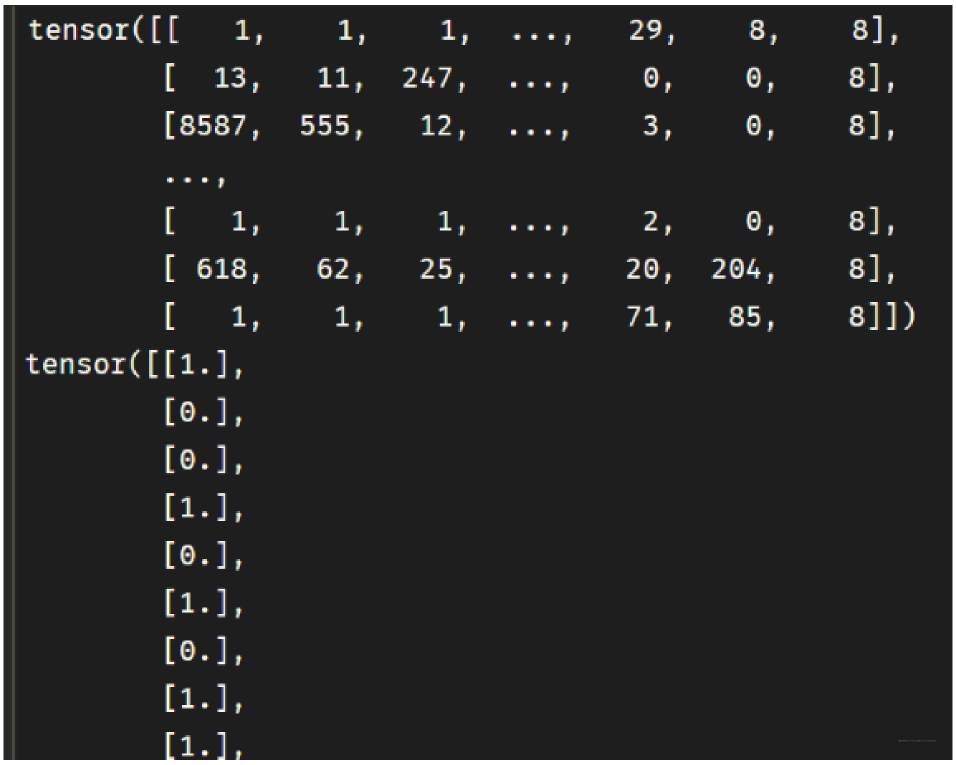
接着将token文本按照样本分割，每个文件存放一个样本的数据

```
In [ ]: # 分割样本
        import os

        if not os.path.exists(train_samples_path):
            os.mkdir(train_samples_path)

        if not os.path.exists(test_samples_path):
            os.mkdir(test_samples_path)

        def split_samples(token_path,samples_dir):
            with open(token_path,"r",encoding = 'utf-8') as fin:
                i = 0
                for line in fin:
                    with open(samples_dir+"%d.txt"%i,"w",encoding = "utf-8") as fout:
                        fout.write(line)
                    i = i+1

        split_samples(train_token_path,train_samples_path)
        split_samples(test_token_path,test_samples_path)
```

创建数据集Dataset，从文件名称列表中读取文件内容

In [ ]:
```python
class imdbDataset(Dataset):
    def __init__(self,samples_dir):
        self.samples_dir = samples_dir
        self.samples_paths = os.listdir(samples_dir)

    def __len__(self):
        return len(self.samples_paths)

    def __getitem__(self,index):
        path = self.samples_dir + self.samples_paths[index]
        with open(path,"r",encoding = "utf-8") as f:
            line = f.readline()
            label,tokens = line.split("\t")
            label = torch.tensor([float(label)],dtype = torch.float)
            feature = torch.tensor([int(x) for x in tokens.split(" ")],dtype = torch.long)
            return  (feature,label)
ds_train = imdbDataset(train_samples_path)
ds_test = imdbDataset(test_samples_path)

dl_train = DataLoader(ds_train,batch_size = BATCH_SIZE,shuffle = True,num_workers=4)
dl_test = DataLoader(ds_test,batch_size = BATCH_SIZE,num_workers=4)

for features,labels in dl_train:
    print(features)
    print(labels)
    break
```

这里数据的样子:



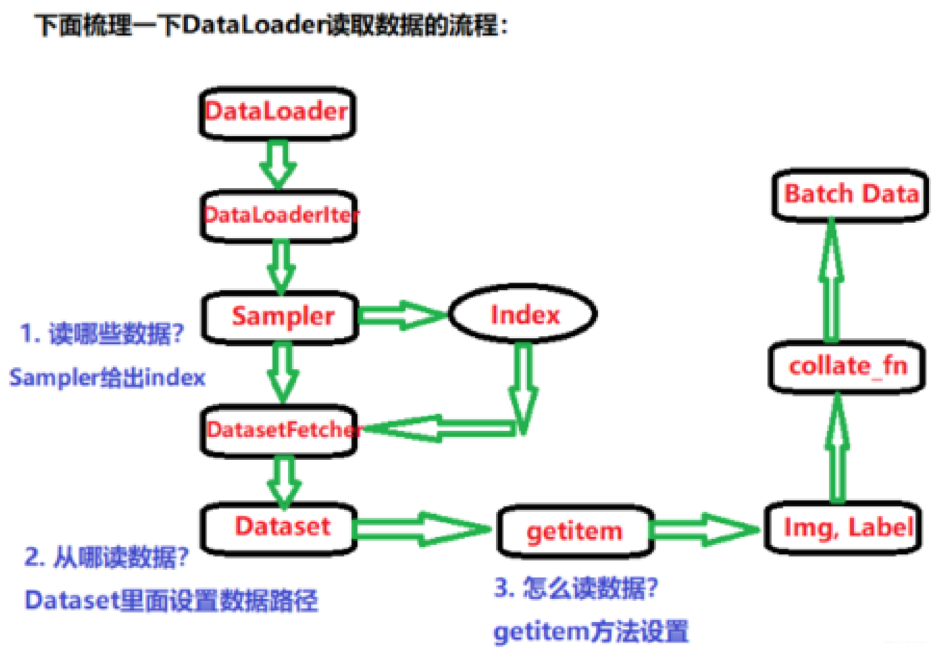
```
tensor([[   1,    1,    1,  ...,   29,    8,    8],
        [  13,   11,  247,  ...,    0,    0,    8],
        [8587,  555,   12,  ...,    3,    0,    8],
        ...,
        [   1,    1,    1,  ...,    2,    0,    8],
        [ 618,   62,   25,  ...,   20,  204,    8],
        [   1,    1,    1,  ...,   71,   85,    8]])
tensor([[1.],
        [0.],
        [0.],
        [1.],
        [0.],
        [1.],
        [0.],
        [1.],
        [1.],
```

features里每一行是一个句子，这里都转成了单词的索引表示，而label里的每一个就是句子对应的分类，好评和差评。

这种方式比较麻烦的就是前面的预处理部分，把数据进行切割，然后构建词典，存到文件等。

PyTorch神经网络建模：

- 数据准备
- 模型建立
- 模型训练
- 模型评估使用和保存



PyTorch构建模型，常用三种方式：

- 继承nn.Module基类构建自定义模型（最常见）
- 使用nn.Sequential按层顺序构建模型（最简单）
- 继承nn.Module基类构建模型，并辅助应用模型容器封装（灵活但复杂）

In [4]:
```
import torch
from torch import nn
from torchkeras import summary
```

继承nn.Module基类构建自定义模型

模型中用到的层一般在__init__函数中定义，然后在forward方法中定义模型的正向传播逻辑。

In [6]:
```python
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.dropout = nn.Dropout2d(p=0.1)
        self.adaptive_pool = nn.AdaptiveMaxPool2d((1, 1))

        self.flatten = nn.Flatten()
        self.linear1 = nn.Linear(64, 32)
        self.relu = nn.ReLU()

        self.linear2 = nn.Linear(32, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self,x):
        x = self.conv1(x)
        x = self.pool1(x)

        x = self.conv2(x)
        x = self.pool2(x)

        x = self.dropout(x)
        x = self.adaptive_pool(x)

        x = self.flatten(x)
        x = self.linear1(x)
        x = self.relu(x)

        x = self.linear2(x)
        y = self.sigmoid(x)

        return y


net = Net()
print(net)
summary(net, input_shape=(3, 32, 32))
```

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout): Dropout2d(p=0.1, inplace=False)
  (adaptive_pool): AdaptiveMaxPool2d(output_size=(1, 1))
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear1): Linear(in_features=64, out_features=32, bias=True)
  (relu): ReLU()
  (linear2): Linear(in_features=32, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 30, 30]             896
         MaxPool2d-2           [-1, 32, 15, 15]               0
            Conv2d-3           [-1, 64, 11, 11]          51,264
```

```
          MaxPool2d-4             [-1, 64, 5, 5]               0
          Dropout2d-5             [-1, 64, 5, 5]               0
  AdaptiveMaxPool2d-6             [-1, 64, 1, 1]               0
            Flatten-7                   [-1, 64]               0
             Linear-8                   [-1, 32]           2,080
               ReLU-9                   [-1, 32]               0
            Linear-10                    [-1, 1]              33
           Sigmoid-11                    [-1, 1]               0
================================================================
Total params: 54,273
Trainable params: 54,273
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.011719
Forward/backward pass size (MB): 0.359634
Params size (MB): 0.207035
Estimated Total Size (MB): 0.578388
----------------------------------------------------------------
```

使用nn.Sequential按层顺序构建模型

使用nn.Sequential按层顺序构建模型无需定义forward方法。

仅适用于简单的模型。

利用add_module方法

In [7]:
```
net = nn.Sequential()

net.add_module("conv1", nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3))
net.add_module("pool1", nn.MaxPool2d(kernel_size=2, stride=2))

net.add_module("conv2", nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5))
net.add_module("pool2", nn.MaxPool2d(kernel_size=2, stride=2))

net.add_module("dropout", nn.Dropout2d(p=0.1))
net.add_module("adaptive_pool", nn.AdaptiveMaxPool2d((1, 1)))

net.add_module("flatten", nn.Flatten())
net.add_module("linear1", nn.Linear(64, 32))
net.add_module("relu", nn.ReLU())

net.add_module("linear2", nn.Linear(32, 1))
net.add_module("sigmoid", nn.Sigmoid())

print(net)
```

```
Sequential(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout): Dropout2d(p=0.1, inplace=False)
  (adaptive_pool): AdaptiveMaxPool2d(output_size=(1, 1))
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear1): Linear(in_features=64, out_features=32, bias=True)
  (relu): ReLU()
  (linear2): Linear(in_features=32, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

利用变长参数

这种构建方式无法给每层指定名称

In [8]:
```
net = nn.Sequential(
    nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3),
    nn.MaxPool2d(kernel_size=2, stride=2),

    nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5),
    nn.MaxPool2d(kernel_size=2, stride=2),

    nn.Dropout2d(p=0.1),
    nn.AdaptiveMaxPool2d((1, 1)),
    nn.Flatten(),

    nn.Linear(64, 32),
    nn.ReLU(),

    nn.Linear(32, 1),
    nn.Sigmoid()
)

print(net)
```

```
Sequential(
  (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
  (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
  (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (4): Dropout2d(p=0.1, inplace=False)
  (5): AdaptiveMaxPool2d(output_size=(1, 1))
  (6): Flatten(start_dim=1, end_dim=-1)
  (7): Linear(in_features=64, out_features=32, bias=True)
  (8): ReLU()
  (9): Linear(in_features=32, out_features=1, bias=True)
  (10): Sigmoid()
)
```

利用OrderDict

In [9]:
```python
from collections import OrderedDict

net = nn.Sequential(OrderedDict([
            ("conv1", nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3)),
            ("pool1", nn.MaxPool2d(kernel_size=2, stride=2)),

            ("conv2", nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5)),
            ("pool2", nn.MaxPool2d(kernel_size=2, stride=2)),

            ("dropout", nn.Dropout2d(p=0.1)),
            ("adaptive_pool", nn.AdaptiveMaxPool2d((1, 1))),

            ("flatten", nn.Flatten()),
            ("linear1", nn.Linear(64, 32)),
            ("relu", nn.ReLU()),

            ("linear2", nn.Linear(32, 1)),
            ("sigmoid", nn.Sigmoid())
          ])
        )

print(net)
```

```
Sequential(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout): Dropout2d(p=0.1, inplace=False)
  (adaptive_pool): AdaptiveMaxPool2d(output_size=(1, 1))
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear1): Linear(in_features=64, out_features=32, bias=True)
  (relu): ReLU()
  (linear2): Linear(in_features=32, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

继承nn.Module基类构建并辅助应用模型容器封装

当模型比较复杂时，可以应用模型容器（nn.Sequential, nn.ModuleList, nn.ModuleDict）对模型的部分结构进行封装，这样做会让模型更有层次感，有时也能减少代码量。

下面展示的范例中，每次仅仅使用一种模型容器，但实际上这些模型容器的使用非常灵活，可以在一个模型中任意组合任意嵌套使用。

nn.Sequential

In [10]:
```python
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Dropout2d(p=0.1),
            nn.AdaptiveMaxPool2d((1, 1))
        )

        self.dense = nn.Sequential(
            nn.Flatten(),
            nn.Linear(64, 32),
            nn.ReLU(),

            nn.Linear(32, 1),
            nn.Sigmoid()
        )

    def forward(self,x):
        x = self.conv(x)
        y = self.dense(x)
        return y


net = Net()
print(net)
```

```
Net(
  (conv): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
    (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Dropout2d(p=0.1, inplace=False)
    (5): AdaptiveMaxPool2d(output_size=(1, 1))
  )
  (dense): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=64, out_features=32, bias=True)
    (2): ReLU()
    (3): Linear(in_features=32, out_features=1, bias=True)
    (4): Sigmoid()
  )
)
```

nn.ModuleList

注意：这里的ModuleList不能用Python中的列表代替

In [11]:
```python
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()

        self.layers = nn.ModuleList([
            nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Dropout2d(p=0.1),
            nn.AdaptiveMaxPool2d((1, 1)),

            nn.Flatten(),
            nn.Linear(64, 32),
            nn.ReLU(),

            nn.Linear(32, 1),
            nn.Sigmoid()]
        )

    def forward(self, x):
        for layer in self.layers:
            x = layer(x)
        return x


net = Net()
print(net)
```

```
Net(
  (layers): ModuleList(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
    (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Dropout2d(p=0.1, inplace=False)
    (5): AdaptiveMaxPool2d(output_size=(1, 1))
    (6): Flatten(start_dim=1, end_dim=-1)
    (7): Linear(in_features=64, out_features=32, bias=True)
    (8): ReLU()
    (9): Linear(in_features=32, out_features=1, bias=True)
    (10): Sigmoid()
  )
)
```

nn.ModuleDict

注意：这里的ModuleDict不能用Python中的字典代替。

In [12]:
```python
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()

        self.layers_dict = nn.ModuleDict({
                "conv1":nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3),
                "pool1": nn.MaxPool2d(kernel_size=2, stride=2),

                "conv2":nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5),
                "pool2": nn.MaxPool2d(kernel_size=2, stride=2),

                "dropout": nn.Dropout2d(p=0.1),
                "adaptive":nn.AdaptiveMaxPool2d((1, 1)),

                "flatten": nn.Flatten(),
                "linear1": nn.Linear(64, 32),
                "relu":nn.ReLU(),

                "linear2": nn.Linear(32, 1),
                "sigmoid": nn.Sigmoid()
            })

    def forward(self,x):
        layers = ["conv1","pool1","conv2","pool2","dropout","adaptive",
                  "flatten","linear1","relu","linear2","sigmoid"]
        for layer in layers:
            x = self.layers_dict[layer](x)
        return x


net = Net()
print(net)
```

```
Net(
  (layers_dict): ModuleDict(
    (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
    (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
    (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (dropout): Dropout2d(p=0.1, inplace=False)
    (adaptive): AdaptiveMaxPool2d(output_size=(1, 1))
    (flatten): Flatten(start_dim=1, end_dim=-1)
    (linear1): Linear(in_features=64, out_features=32, bias=True)
    (relu): ReLU()
    (linear2): Linear(in_features=32, out_features=1, bias=True)
    (sigmoid): Sigmoid()
  )
)
```

PyTorch神经网络建模：

- 数据准备
- 模型建立
- 模型训练
- 模型评估使用和保存

三种常见的训练模型代码编写风格：

- 脚本形式训练循环
- 函数形式训练循环
- 类形式训练循环

In [1]:
```
import torch
from torch import nn
from torchkeras import summary, Model

import torchvision
from torchvision import transforms
```

数据准备

In [2]:
```
transform = transforms.Compose([transforms.ToTensor()])

ds_train = torchvision.datasets.MNIST(root="data/minist/", train=True, download=True, transform=
ds_valid = torchvision.datasets.MNIST(root="data/minist/", train=False, download=True, transform

dl_train =  torch.utils.data.DataLoader(ds_train, batch_size=128, shuffle=True, num_workers=4)
dl_valid =  torch.utils.data.DataLoader(ds_valid, batch_size=128, shuffle=False, num_workers=4)

print(len(ds_train))  # 60000
print(len(ds_valid))  # 10000
```

```
60000
10000
```

构建模型

In [3]:
```
net = nn.Sequential()

net.add_module("conv1", nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3))
net.add_module("pool1", nn.MaxPool2d(kernel_size=2, stride=2))

net.add_module("conv2", nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5))
net.add_module("pool2", nn.MaxPool2d(kernel_size=2, stride=2))

net.add_module("dropout", nn.Dropout2d(p=0.1))
net.add_module("adaptive_pool", nn.AdaptiveMaxPool2d((1, 1)))

net.add_module("flatten", nn.Flatten())
net.add_module("linear1", nn.Linear(64, 32))
net.add_module("relu", nn.ReLU())

net.add_module("linear2", nn.Linear(32, 10))
net.add_module("sigmoid", nn.Sigmoid())

print(net)
summary(net, input_shape=(1, 32, 32))
```

```
Sequential(
  (conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout): Dropout2d(p=0.1, inplace=False)
  (adaptive_pool): AdaptiveMaxPool2d(output_size=(1, 1))
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear1): Linear(in_features=64, out_features=32, bias=True)
  (relu): ReLU()
  (linear2): Linear(in_features=32, out_features=10, bias=True)
  (sigmoid): Sigmoid()
)
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 30, 30]             320
         MaxPool2d-2           [-1, 32, 15, 15]               0
            Conv2d-3           [-1, 64, 11, 11]          51,264
         MaxPool2d-4             [-1, 64, 5, 5]               0
         Dropout2d-5             [-1, 64, 5, 5]               0
 AdaptiveMaxPool2d-6             [-1, 64, 1, 1]               0
           Flatten-7                   [-1, 64]               0
            Linear-8                   [-1, 32]           2,080
              ReLU-9                   [-1, 32]               0
           Linear-10                   [-1, 10]             330
          Sigmoid-11                   [-1, 10]               0
================================================================
Total params: 53,994
Trainable params: 53,994
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.003906
Forward/backward pass size (MB): 0.359772
Params size (MB): 0.205971
Estimated Total Size (MB): 0.569649
----------------------------------------------------------------
```

模型训练设置

定义模型的评估指标，损失函数，优化方法

In [4]:

```
import datetime
import numpy as np
import pandas as pd
from sklearn.metrics import accuracy_score

def accuracy(y_pred,y_true):
    y_pred_cls = torch.argmax(nn.Softmax(dim=1)(y_pred), dim=1).data
    return accuracy_score(y_true, y_pred_cls)

loss_func = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(params=net.parameters(), lr=0.01)
metric_func = accuracy
metric_name = "accuracy"
```

三种风格训练代码

脚本式训练风格

In [5]:
```
epochs = 3
log_step_freq = 100

dfhistory = pd.DataFrame(columns = ["epoch", "loss", metric_name, "val_loss", "val_" + metric_name

print("Start Training...")
nowtime = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
print("%s" % nowtime)

for epoch in range(1, epochs + 1):

    # 训练循环
    net.train()
    loss_sum = 0.0
    metric_sum = 0.0
    step = 1

    for step, (features, labels) in enumerate(dl_train, 1):

        # 梯度清零
        optimizer.zero_grad()

        # 正向传播求损失
        predictions = net(features)
        loss = loss_func(predictions, labels)
        metric = metric_func(predictions, labels)

        # 反向传播求梯度
        loss.backward()
        optimizer.step()

        # 打印batch级别日志
        loss_sum += loss.item()
        metric_sum += metric.item()
        if step % log_step_freq == 0:
            print(("[step = %d] loss: %.3f, " + metric_name + ": %.3f") %
                  (step, loss_sum / step, metric_sum / step))

    # 验证循环
    net.eval()
    val_loss_sum = 0.0
    val_metric_sum = 0.0
    val_step = 1

    for val_step, (features, labels) in enumerate(dl_valid, 1):
        with torch.no_grad():
            predictions = net(features)
            val_loss = loss_func(predictions, labels)
            val_metric = metric_func(predictions, labels)

        val_loss_sum += val_loss.item()
        val_metric_sum += val_metric.item()

    # 记录日志
    info = (epoch, loss_sum / step, metric_sum / step,
            val_loss_sum / val_step, val_metric_sum / val_step)
    dfhistory.loc[epoch - 1] = info

    # 打印epoch级别日志
    print(("\nEPOCH = %d, loss = %.3f," + metric_name + \
           "  = %.3f, val_loss = %.3f, " + "val_" + metric_name + " = %.3f")
          % info)
    nowtime = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    print("\n" + "%s" % nowtime)
print('Finished Training...')
```

```
Start Training...
2022-02-09 09:37:38
[step = 100] loss: 1.773, accuracy: 0.629
[step = 200] loss: 1.659, accuracy: 0.768
[step = 300] loss: 1.613, accuracy: 0.824
[step = 400] loss: 1.588, accuracy: 0.853

EPOCH = 1, loss = 1.577,accuracy  = 0.866, val_loss = 1.517, val_accuracy = 0.939

2022-02-09 09:37:51
[step = 100] loss: 1.514, accuracy: 0.939
[step = 200] loss: 1.510, accuracy: 0.942
[step = 300] loss: 1.509, accuracy: 0.944
[step = 400] loss: 1.509, accuracy: 0.942

EPOCH = 2, loss = 1.510,accuracy  = 0.941, val_loss = 1.509, val_accuracy = 0.938

2022-02-09 09:38:03
[step = 100] loss: 1.520, accuracy: 0.924
[step = 200] loss: 1.520, accuracy: 0.922
[step = 300] loss: 1.524, accuracy: 0.910
[step = 400] loss: 1.534, accuracy: 0.893

EPOCH = 3, loss = 1.534,accuracy  = 0.889, val_loss = 1.549, val_accuracy = 0.821

2022-02-09 09:38:16
Finished Training...
```

函数风格

In [6]:
```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.dropout = nn.Dropout2d(p=0.1)
        self.adaptive_pool = nn.AdaptiveMaxPool2d((1, 1))

        self.flatten = nn.Flatten()
        self.linear1 = nn.Linear(64, 32)
        self.relu = nn.ReLU()

        self.linear2 = nn.Linear(32, 10)

        self.loss_func = nn.CrossEntropyLoss()
        self.optimizer = torch.optim.Adam(params=self.parameters(), lr=0.01)
        self.metric_func = accuracy
        self.metric_name = "accuracy"

    def forward(self,x):
        x = self.conv1(x)
        x = self.pool1(x)

        x = self.conv2(x)
        x = self.pool2(x)

        x = self.dropout(x)
        x = self.adaptive_pool(x)

        x = self.flatten(x)
        x = self.linear1(x)
        x = self.relu(x)

        y = self.linear2(x)

        return y
```

In [7]:
```
def train_step(model, features, labels):

    # 训练模式，dropout层发生作用
    model.train()

    # 梯度清零
    model.optimizer.zero_grad()

    # 正向传播求损失
    predictions = model(features)
    loss = model.loss_func(predictions, labels)
    metric = model.metric_func(predictions, labels)

    # 反向传播求梯度
    loss.backward()
    model.optimizer.step()

    return loss.item(), metric.item()


@torch.no_grad()
def valid_step(model, features, labels):

    # 预测模式，dropout层不发生作用
    model.eval()

    predictions = model(features)
    loss = model.loss_func(predictions, labels)
    metric = model.metric_func(predictions, labels)

    return loss.item(), metric.item()


model = Net()
# 测试train_step效果
features, labels = next(iter(dl_train))
train_step(model, features, labels)
```

Out[7]:
```
(2.3156378269195557, 0.09375)
```

In [8]:
```
def train_model(model, epochs, dl_train, dl_valid, log_step_freq):

    metric_name = model.metric_name
    dfhistory = pd.DataFrame(columns = ["epoch", "loss", metric_name, "val_loss", "val_" + metric_
    print("Start Training...")
    nowtime = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    print("%s" % nowtime)

    for epoch in range(1, epochs + 1):

        # 训练循环
        loss_sum = 0.0
        metric_sum = 0.0
        step = 1

        for step, (features, labels) in enumerate(dl_train, 1):

            loss, metric = train_step(model, features, labels)

            # 打印batch级别日志
            loss_sum += loss
            metric_sum += metric
            if step % log_step_freq == 0:
                print(("[step = %d] loss: %.3f, " + metric_name + ": %.3f") %
                      (step, loss_sum / step, metric_sum / step))

        # 验证循环
        val_loss_sum = 0.0
        val_metric_sum = 0.0
        val_step = 1

        for val_step, (features, labels) in enumerate(dl_valid, 1):

            val_loss, val_metric = valid_step(model, features, labels)

            val_loss_sum += val_loss
            val_metric_sum += val_metric

        # 记录日志
        info = (epoch, loss_sum / step, metric_sum / step,
                val_loss_sum / val_step, val_metric_sum / val_step)
        dfhistory.loc[epoch - 1] = info

        # 打印epoch级别日志
        print(("\nEPOCH = %d, loss = %.3f," + metric_name + \
               "  = %.3f, val_loss = %.3f, " + "val_" + metric_name + " = %.3f")
               % info)
        nowtime = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        print("\n" + "%s" % nowtime)

    print('Finished Training...')
    return dfhistory

epochs = 3
dfhistory = train_model(model, epochs, dl_train, dl_valid, log_step_freq = 100)
```

```
Start Training...
2022-02-09 09:38:18
[step = 100] loss: 0.881, accuracy: 0.696
[step = 200] loss: 0.563, accuracy: 0.811
[step = 300] loss: 0.439, accuracy: 0.854
[step = 400] loss: 0.367, accuracy: 0.879

EPOCH = 1, loss = 0.337,accuracy  = 0.889, val_loss = 0.096, val_accuracy = 0.969
```

```
2022-02-09 09:38:30
[step = 100] loss: 0.122, accuracy: 0.961
[step = 200] loss: 0.125, accuracy: 0.961
[step = 300] loss: 0.118, accuracy: 0.964
[step = 400] loss: 0.117, accuracy: 0.964

EPOCH = 2, loss = 0.119,accuracy  = 0.964, val_loss = 0.073, val_accuracy = 0.979

2022-02-09 09:38:42
[step = 100] loss: 0.099, accuracy: 0.970
[step = 200] loss: 0.103, accuracy: 0.968
[step = 300] loss: 0.104, accuracy: 0.969
[step = 400] loss: 0.099, accuracy: 0.970

EPOCH = 3, loss = 0.100,accuracy  = 0.970, val_loss = 0.066, val_accuracy = 0.980

2022-02-09 09:38:55
Finished Training...
```

类风格

使用torchkeras定义的模型接口构建模型，并调用compile方法和fit方法训练模型。

推荐使用该方式

In [9]:
```python
import torchkeras

class CnnModel(nn.Module):
    def __init__(self):
        super().__init__()

        self.layers = nn.ModuleList([
            nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Dropout2d(p=0.1),
            nn.AdaptiveMaxPool2d((1, 1)),

            nn.Flatten(),
            nn.Linear(64, 32),
            nn.ReLU(),

            nn.Linear(32, 10)]
        )

    def forward(self,x):
        for layer in self.layers:
            x = layer(x)
        return x


model = torchkeras.Model(CnnModel())    # 封装成了keras里面模型的格式
print(model)

model.summary(input_shape=(1, 32, 32))
```

```
Model(
  (net): CnnModel(
    (layers): ModuleList(
      (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1))
      (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
      (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (4): Dropout2d(p=0.1, inplace=False)
      (5): AdaptiveMaxPool2d(output_size=(1, 1))
      (6): Flatten(start_dim=1, end_dim=-1)
      (7): Linear(in_features=64, out_features=32, bias=True)
      (8): ReLU()
      (9): Linear(in_features=32, out_features=10, bias=True)
    )
  )
)
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 30, 30]             320
         MaxPool2d-2           [-1, 32, 15, 15]               0
            Conv2d-3           [-1, 64, 11, 11]          51,264
         MaxPool2d-4             [-1, 64, 5, 5]               0
         Dropout2d-5             [-1, 64, 5, 5]               0
 AdaptiveMaxPool2d-6             [-1, 64, 1, 1]               0
           Flatten-7                   [-1, 64]               0
            Linear-8                   [-1, 32]           2,080
              ReLU-9                   [-1, 32]               0
           Linear-10                   [-1, 10]             330
================================================================
Total params: 53,994
```

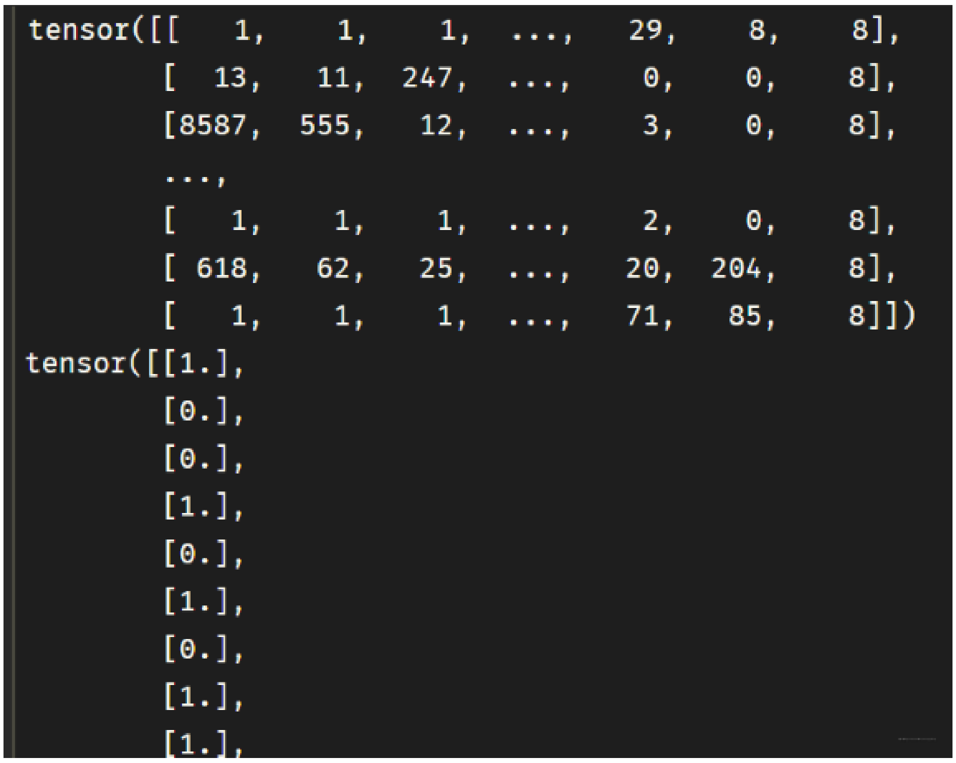
```
Trainable params: 53,994
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.003906
Forward/backward pass size (MB): 0.359695
Params size (MB): 0.205971
Estimated Total Size (MB): 0.569572
----------------------------------------------------------------
```

In [10]:
```
# keras风格
model.compile(loss_func = nn.CrossEntropyLoss(),
             optimizer= torch.optim.Adam(model.parameters(), lr=0.02),
             metrics_dict={"accuracy": accuracy})

# 训练只需要一句话
dfhistory = model.fit(3, dl_train = dl_train, dl_val=dl_valid, log_step_freq=100)
```

```
Start Training ...

================================================================================2022-02-09 09:3
8:55
{'step': 100, 'loss': 0.963, 'accuracy': 0.689}
{'step': 200, 'loss': 0.599, 'accuracy': 0.81}
{'step': 300, 'loss': 0.462, 'accuracy': 0.854}
{'step': 400, 'loss': 0.397, 'accuracy': 0.875}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|   1   | 0.368 |  0.885   |  0.118   |    0.965     |
+-------+-------+----------+----------+--------------+

================================================================================2022-02-09 09:3
9:07
{'step': 100, 'loss': 0.126, 'accuracy': 0.963}
{'step': 200, 'loss': 0.132, 'accuracy': 0.962}
{'step': 300, 'loss': 0.134, 'accuracy': 0.961}
{'step': 400, 'loss': 0.145, 'accuracy': 0.959}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|   2   | 0.146 |  0.959   |  0.118   |    0.964     |
+-------+-------+----------+----------+--------------+

================================================================================2022-02-09 09:3
9:20
{'step': 100, 'loss': 0.119, 'accuracy': 0.966}
{'step': 200, 'loss': 0.126, 'accuracy': 0.965}
{'step': 300, 'loss': 0.141, 'accuracy': 0.961}
{'step': 400, 'loss': 0.142, 'accuracy': 0.961}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|   3   | 0.141 |  0.961   |  0.088   |    0.975     |
+-------+-------+----------+----------+--------------+

================================================================================2022-02-09 09:3
9:32
Finished Training...
```

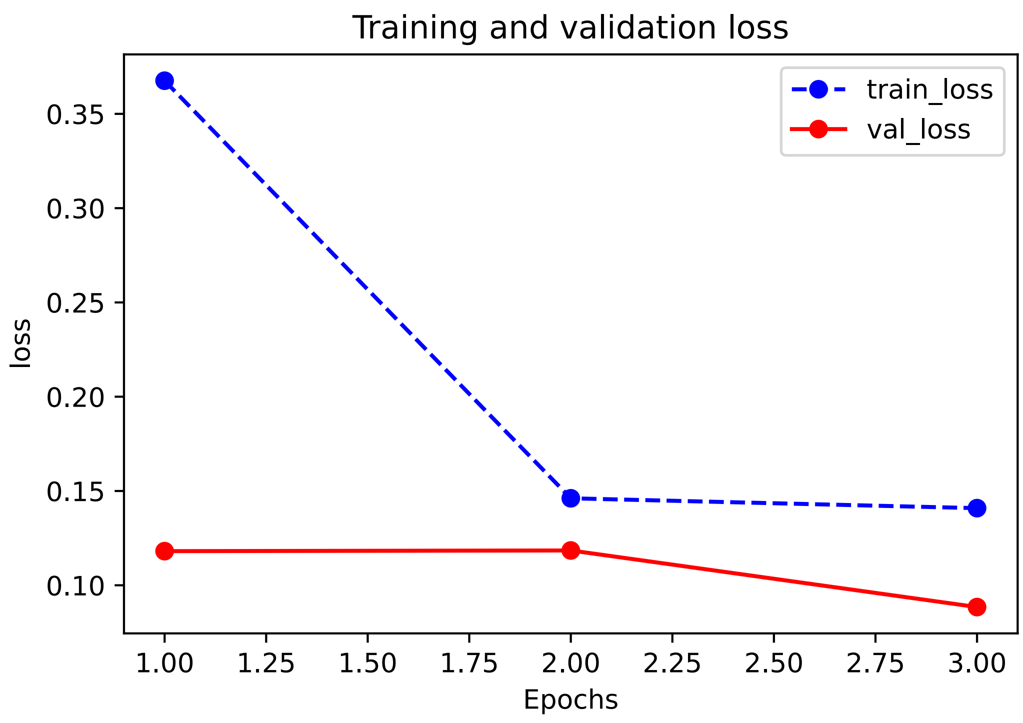
模型评估

评估在训练集和验证集上的效果，模型训练过程中，都是保留着一个dfhistory的，这个是DataFrame结构，里面是训练过程中训练集和验证集上模型的正确率和损失的变化，可以通过可视化这个来看模型的训练情况。



```
tensor([[   1,    1,    1,  ...,   29,    8,    8],
        [  13,   11,  247,  ...,    0,    0,    8],
        [8587,  555,   12,  ...,    3,    0,    8],
        ...,
        [   1,    1,    1,  ...,    2,    0,    8],
        [ 618,   62,   25,  ...,   20,  204,    8],
        [   1,    1,    1,  ...,   71,   85,    8]])
tensor([[1.],
        [0.],
        [0.],
        [1.],
        [0.],
        [1.],
        [0.],
        [1.],
        [1.],
```

In [11]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'svg'

import matplotlib.pyplot as plt

def plot_metric(dfhistory, metric):
    train_metrics = dfhistory[metric]
    val_metrics = dfhistory['val_' + metric]
    epochs = range(1, len(train_metrics) + 1)
    plt.plot(epochs, train_metrics, 'bo--')
    plt.plot(epochs, val_metrics, 'ro-')
    plt.title('Training and validation ' + metric)
    plt.xlabel("Epochs")
    plt.ylabel(metric)
    plt.legend(["train_" + metric, 'val_' + metric])
    plt.show()


# 观察损失和准确率的变化
plot_metric(dfhistory, "loss")
plot_metric(dfhistory, "accuracy")
```



Training and validation accuracy

模型使用

In [12]:
```
def predict(model, dl):
    model.eval()

    with torch.no_grad():
        result = torch.cat([model.forward(t[0]) for t in dl])

    return (result.data)


#预测概率
y_pred_probs = predict(model, dl_valid)

#预测类别
y_pred = torch.where(y_pred_probs > 0.5, torch.ones_like(y_pred_probs),
                     torch.zeros_like(y_pred_probs))

print(y_pred_probs, y_pred)
```

```
tensor([[  6.3831,   9.4558,   6.4552,  ...,  30.3724,  -6.4151,  16.4176],
        [ 10.3321,  13.2796,  20.0930,  ...,   5.1266,   1.9966,  -1.5246],
        [ -2.5078,  17.1817, -10.2687,  ..., -12.3159, -14.6532,  -3.4462],
        ...,
        [  2.5689,   6.8503,   9.1359,  ...,   8.1353,   5.6055,   9.4993],
        [  3.7595,  -1.8394,   2.8264,  ...,  -1.2887,   2.9036,  -1.8120],
        [ 13.8020,   8.5427,   4.9006,  ...,  -6.0255,   2.5966,  -2.5880]]) tensor([[1., 1.,
1.,  ..., 1., 0., 1.],
        [1., 1., 1.,  ..., 1., 1., 0.],
        [0., 1., 0.,  ..., 0., 0., 0.],
        ...,
        [1., 1., 1.,  ..., 1., 1., 1.],
        [1., 0., 1.,  ..., 0., 1., 0.],
        [1., 1., 1.,  ..., 0., 1., 0.]])
```

模型保存

PyTorch两种保存方式，调用pickle序列化方法实现：

- 只保存模型参数
- 保存整个模型

保存模型参数（推荐使用）

In [16]:
```
print(model.state_dict().keys())

# 保存模型参数

torch.save(model.state_dict(), "net_parameter.pkl")

# # 使用
# net_clone = CnnModel()
# net_clone.load_state_dict(torch.load("net_parameter.pkl"))

# 如果报错，模型参数前面有模型名称，则需要做如下处理
checkpoint = torch.load("net_parameter.pkl", map_location=torch.device('cpu'))
from collections import OrderedDict

new_state_dict = OrderedDict()
for k, v in checkpoint.items():
    name = k.replace("net.", "")  # 移除net
    new_state_dict[name] = v

net_clone.load_state_dict(new_state_dict, strict=False)
```

```
odict_keys(['net.layers.0.weight', 'net.layers.0.bias', 'net.layers.2.weight', 'net.layers.2.bi
as', 'net.layers.7.weight', 'net.layers.7.bias', 'net.layers.9.weight', 'net.layers.9.bias'])
```

Out[16]:
```
<All keys matched successfully>
```

保存整个模型

In [18]:
```
torch.save(net, 'net_model.pkl')
net_loaded = torch.load('net_model.pkl')
```

PyTorch神经网络建模：

- 数据准备
- 模型建立
- 模型训练
- 模型评估使用和保存

训练耗时两个原因：

- 数据准备：使用更多的进程来准备数据
- 参数迭代：GPU进行加速训练

PyTorch使用GPU加速

In [4]:
```
import torch
import torch.nn as nn

features = torch.tensor(range(10))
labels = torch.tensor(range(10))

model = nn.Linear(3, 3)
```

In [5]:
```
# 定义模型
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model.to(device) # 移动模型到cuda

# 训练模型
features = features.to(device) # 移动数据到cuda
labels = labels.to(device)  # 或者  labels = labels.cuda() if torch.cuda.is_available() else label
```

如果要使用多个GPU训练模型，可以将模型设置为数据并行风格。

则模型移动到GPU上之后，会在每个GPU上拷贝一个副本，并把数据平分到各个GPU上进行训练

In [6]:
```
# 定义模型
if torch.cuda.device_count() > 1:
    model = nn.DataParallel(model) # 包装为并行风格模型

# 训练模型
features = features.to(device) # 移动数据到cuda
labels = labels.to(device) # 或者 labels = labels.cuda() if torch.cuda.is_available() else labels
```

GPU相关操作

In [7]:
```
# 查看gpu信息
if_cuda = torch.cuda.is_available()
print("if_cuda=", if_cuda)
```

```
if_cuda= True
```

In [8]:
```
# GPU的数量
gpu_count = torch.cuda.device_count()
print("gpu_count=", gpu_count)
```
```
gpu_count= 1
```

In [9]:
```
# 将张量在gpu和cpu间移动
tensor = torch.rand((100,100))
tensor_gpu = tensor.to("cuda:0") # 或者 tensor_gpu = tensor.cuda()
print(tensor_gpu.device)
print(tensor_gpu.is_cuda)

tensor_cpu = tensor_gpu.to("cpu") # 或者 tensor_cpu = tensor_gpu.cpu()
print(tensor_cpu.device)
```
```
cuda:0
True
cpu
```

In [13]:
```
# 查看数据与模型的device
tensor = torch.rand((100,100))
print(tensor.device)

print(next(model.parameters()).device)
```
```
cpu
cuda:0
```

In [14]:
```
# 将模型中的全部张量移动到gpu上
net = nn.Linear(2, 1)
print(next(net.parameters()).is_cuda)
net.to("cuda:0") # 将模型中的全部参数张量依次到GPU上，无需重新赋值net = net.to("cuda:0")
print(next(net.parameters()).is_cuda)
print(next(net.parameters()).device)
```
```
False
True
cuda:0
```

In [15]:
```
# 创建支持多个gpu数据并行的模型
linear = nn.Linear(2,1)
print(next(linear.parameters()).device)

model = nn.DataParallel(linear)
print(model.device_ids)
print(next(model.module.parameters()).device)
```
```
cpu
[0]
cuda:0
```

In [16]:
```
# 注意保存参数时要指定保存model.module的参数
torch.save(model.module.state_dict(), "model_parameter.pkl")

linear = nn.Linear(2,1)
linear.load_state_dict(torch.load("model_parameter.pkl"))
```
Out[16]:
```
<All keys matched successfully>
```

In [17]:
```python
# 清空cuda缓存，该方在cuda超内存时十分有用
torch.cuda.empty_cache()
```

线性回归范例

In [21]:
```python
import time

# 准备数据
n = 1000000

X = 10 * torch.rand([n, 2]) - 5.0  #torch.rand是均匀分布
w0 = torch.tensor([[2.0, -3.0]])
b0 = torch.tensor([[10.0]])
# @表示矩阵乘法, 增加正态扰动
Y = X @ w0.t() + b0 + torch.normal(0.0, 2.0, size = [n, 1])

# 移动到GPU上
print("torch.cuda.is_available() = ", torch.cuda.is_available())
X = X.cuda()
Y = Y.cuda()

# 定义模型
class LinearRegression(nn.Module):
    def __init__(self):
        super().__init__()

        self.w = nn.Parameter(torch.randn_like(w0))
        self.b = nn.Parameter(torch.zeros_like(b0))

    #正向传播
    def forward(self,x):
        return x @ self.w.t() + self.b


linear = LinearRegression()

# 移动模型到GPU上
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
linear.to(device)

# 训练模型
optimizer = torch.optim.Adam(linear.parameters(), lr=0.1)
loss_func = nn.MSELoss()


def train(epoches):
    tic = time.time()
    for epoch in range(epoches):
        optimizer.zero_grad()

        Y_pred = linear(X)
        loss = loss_func(Y_pred, Y)

        loss.backward()
        optimizer.step()

        if epoch % 50==0:
            print({"epoch":epoch, "loss":loss.item()})

    toc = time.time()
    print("time used:",toc-tic)


train(500)
```

```
torch.cuda.is_available() =  True
{'epoch': 0, 'loss': 214.75949096679688}
{'epoch': 50, 'loss': 33.45890808105469}
{'epoch': 100, 'loss': 9.039307594299316}
```

```
{'epoch': 150, 'loss': 4.49655818939209}
{'epoch': 200, 'loss': 4.030478000640869}
{'epoch': 250, 'loss': 4.006821632385254}
{'epoch': 300, 'loss': 4.006302356719971}
{'epoch': 350, 'loss': 4.006300449371338}
{'epoch': 400, 'loss': 4.0062994956970215}
{'epoch': 450, 'loss': 4.006317615509033}
time used: 0.6919867992401123
```

torchkeras

torchkeras单个GPU

In [29]:
```python
import torchkeras

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

model = torchkeras.Model(LinearRegression())

# 注意此处compile时指定了device
model.compile(loss_func = nn.MSELoss(),
             optimizer= torch.optim.Adam(model.parameters(), lr=0.1),
             device = device)
```

torchkeras多GPU

In [30]:
```python
class CnnModule(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.ModuleList([
            nn.Conv2d(in_channels=1,out_channels=32,kernel_size = 3),
            nn.MaxPool2d(kernel_size = 2,stride = 2),
            nn.Conv2d(in_channels=32,out_channels=64,kernel_size = 5),
            nn.MaxPool2d(kernel_size = 2,stride = 2),
            nn.Dropout2d(p = 0.1),
            nn.AdaptiveMaxPool2d((1,1)),
            nn.Flatten(),
            nn.Linear(64,32),
            nn.ReLU(),
            nn.Linear(32,10)]
        )
    def forward(self,x):
        for layer in self.layers:
            x = layer(x)
        return x

net = nn.DataParallel(CnnModule())  #Attention this line!!!   这一行， 要封装成并行的方式
model = torchkeras.Model(net)

model.summary(input_shape=(1,32,32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 30, 30]             320
         MaxPool2d-2           [-1, 32, 15, 15]               0
            Conv2d-3           [-1, 64, 11, 11]          51,264
         MaxPool2d-4             [-1, 64, 5, 5]               0
         Dropout2d-5             [-1, 64, 5, 5]               0
 AdaptiveMaxPool2d-6             [-1, 64, 1, 1]               0
           Flatten-7                   [-1, 64]               0
            Linear-8                   [-1, 32]           2,080
              ReLU-9                   [-1, 32]               0
           Linear-10                   [-1, 10]             330
================================================================
Total params: 53,994
Trainable params: 53,994
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.003906
Forward/backward pass size (MB): 0.359695
Params size (MB): 0.205971
Estimated Total Size (MB): 0.569572
----------------------------------------------------------------
```

In [33]:
```
import torch
from torch import nn
from torchkeras import summary, Model

import torchvision
from torchvision import transforms

transform = transforms.Compose([transforms.ToTensor()])

ds_train = torchvision.datasets.MNIST(root="data/minist/", train=True, download=True, transform=
ds_valid = torchvision.datasets.MNIST(root="data/minist/", train=False, download=True, transform

dl_train =  torch.utils.data.DataLoader(ds_train, batch_size=128, shuffle=True, num_workers=4)
dl_valid =  torch.utils.data.DataLoader(ds_valid, batch_size=128, shuffle=False, num_workers=4)

print(len(ds_train))  # 60000
print(len(ds_valid))  # 10000
```

```
60000
10000
```

```
In  [34]: from sklearn.metrics import accuracy_score

          def accuracy(y_pred,y_true):
              y_pred_cls = torch.argmax(nn.Softmax(dim=1)(y_pred),dim=1).data
              return accuracy_score(y_true.cpu().numpy(),y_pred_cls.cpu().numpy())
              # 注意此处要将数据先移动到cpu上，然后才能转换成numpy数组

          device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
          model.compile(loss_func = nn.CrossEntropyLoss(),
                       optimizer= torch.optim.Adam(model.parameters(),lr = 0.02),
                       metrics_dict={"accuracy":accuracy},device = device) # 注意此处compile时指定了device

          dfhistory = model.fit(3,dl_train = dl_train, dl_val=dl_valid, log_step_freq=100)
```

```
Start Training ...

================================================================================2022-02-09 10:5
4:30
{'step': 100, 'loss': 0.899, 'accuracy': 0.676}
{'step': 200, 'loss': 0.573, 'accuracy': 0.802}
{'step': 300, 'loss': 0.459, 'accuracy': 0.846}
{'step': 400, 'loss': 0.398, 'accuracy': 0.87}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|   1   | 0.362 |  0.882   |  0.078   |    0.976     |
+-------+-------+----------+----------+--------------+

================================================================================2022-02-09 10:5
4:35
{'step': 100, 'loss': 0.129, 'accuracy': 0.962}
{'step': 200, 'loss': 0.135, 'accuracy': 0.96}
{'step': 300, 'loss': 0.157, 'accuracy': 0.955}
{'step': 400, 'loss': 0.151, 'accuracy': 0.957}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|   2   | 0.148 |  0.958   |  0.131   |     0.97     |
+-------+-------+----------+----------+--------------+

================================================================================2022-02-09 10:5
4:39
{'step': 100, 'loss': 0.168, 'accuracy': 0.954}
{'step': 200, 'loss': 0.163, 'accuracy': 0.956}
{'step': 300, 'loss': 0.195, 'accuracy': 0.95}
{'step': 400, 'loss': 0.212, 'accuracy': 0.947}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|   3   | 0.209 |  0.948   |   0.11   |    0.973     |
+-------+-------+----------+----------+--------------+

================================================================================2022-02-09 10:5
4:43
Finished Training...
```

保存模型

In [36]:
```
# save the model parameters
torch.save(model.net.module.state_dict(), "model_parameter.pkl")  # 这里的model.net.module

net_clone = CnnModule()
net_clone.load_state_dict(torch.load("model_parameter.pkl"))

model_clone = torchkeras.Model(net_clone)
model_clone.compile(loss_func = nn.CrossEntropyLoss(),
             optimizer= torch.optim.Adam(model.parameters(),lr = 0.02),
             metrics_dict={"accuracy":accuracy},device = device)
model_clone.evaluate(dl_valid)
```

Out[36]:
```
{'val_loss': 0.10967098912990451, 'val_accuracy': 0.9728045886075949}
```