西北工業大學

研究生专业课程考试答题册

得分：

学　　号 2021200082

姓　　名 沈　键

考试课程 机器学习与人工智能

考试日期 2022.03.07

西北工业大学研究生院

Report01 - 交通事故理赔审核预测

- 沈键
- 2021200082

1. 任务简介

在交通摩擦（事故）发生后，理赔员会前往现场勘察、采集信息，这些信息往往影响着车主是否能够得到保险公司的理赔。训练集数据包括理赔人员在现场对该事故方采集的36条信息，信息已经被编码，以及该事故方最终是否获得理赔。我们的任务是根据这36条信息预测该事故方没有被理赔的概率。

2. 数据分析

想来分析一下所给的数据特征。使用pandas读入train.csv文件并显示前几行：

In [1]:
```
import pandas as pd

train_df = pd.read_csv("./data/train.csv")
train_df.head(10)
```

Out[1]:

	CaseId	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	...	Q28	Q29	Q30	Q31	Q32	Q33	Q34	Q35	Q3(
0	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	(
1	2	0	0	0	0	0	0	0	0	0	...	0	1	1	1	1	0	0	0	(
2	3	0	0	0	0	0	0	0	1	0	...	1	2	2	2	1	0	0	0	(
3	4	0	0	0	0	0	0	0	0	0	...	1	3	2	3	1	0	0	1	
4	5	0	0	0	0	0	0	0	0	0	...	1	4	2	4	1	0	0	1	
5	6	0	0	0	0	0	0	0	0	0	...	1	2	3	5	1	0	0	0	(
6	7	0	0	0	0	0	0	0	0	0	...	0	3	1	6	1	0	0	1	
7	8	0	0	0	0	0	0	0	0	0	...	1	3	1	3	1	0	0	1	
8	9	0	0	0	0	0	0	0	2	0	...	0	2	1	2	1	0	0	0	(
9	10	0	0	0	0	0	0	0	0	0	...	0	2	1	7	1	0	0	0	(

10 rows × 38 columns

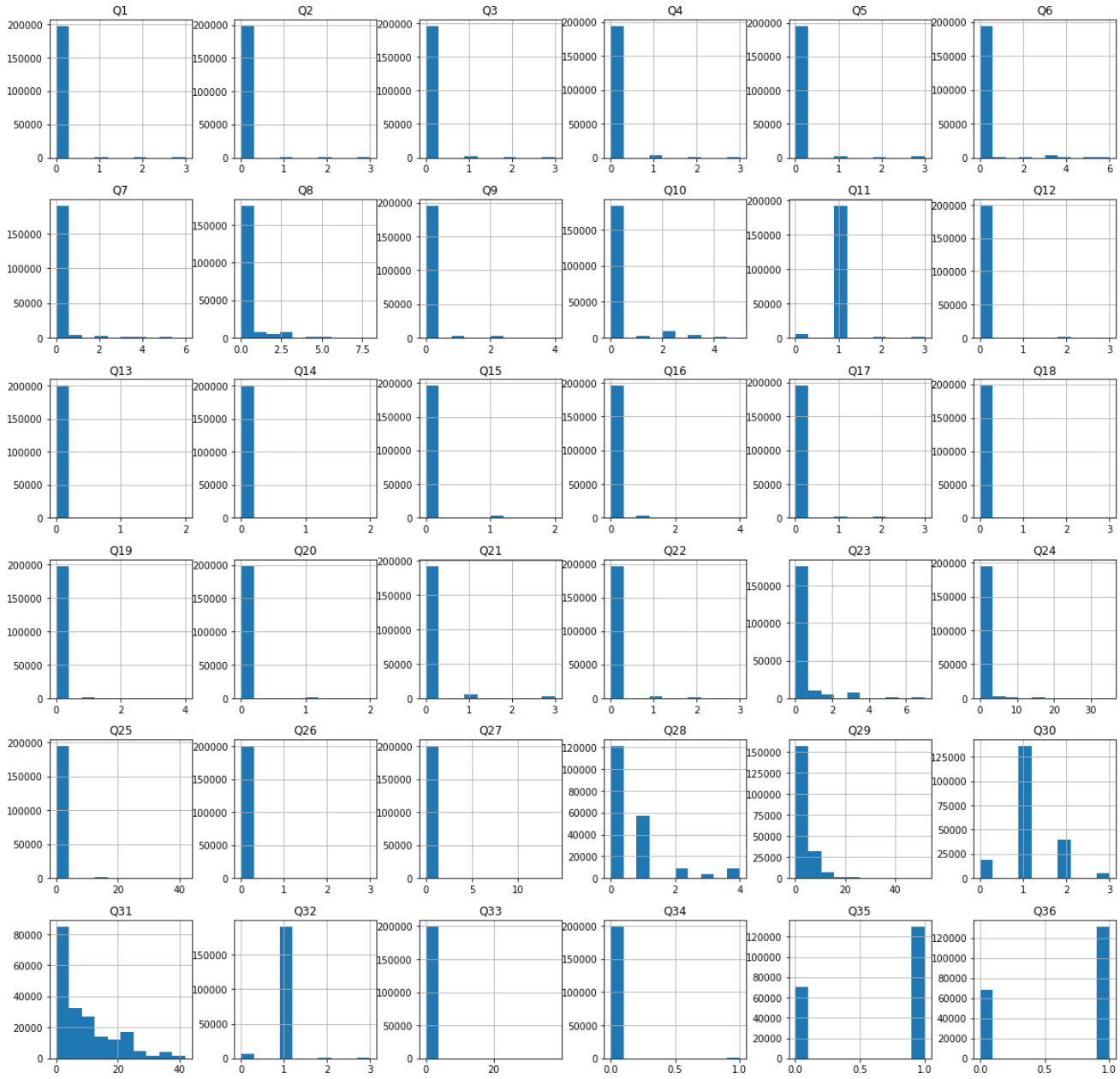
In [2]:
```
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200000 entries, 0 to 199999
Data columns (total 38 columns):
 #   Column      Non-Null Count   Dtype
---  ------      --------------   -----
 0   CaseId      200000 non-null  int64
 1   Q1          200000 non-null  int64
 2   Q2          200000 non-null  int64
 3   Q3          200000 non-null  int64
 4   Q4          200000 non-null  int64
 5   Q5          200000 non-null  int64
 6   Q6          200000 non-null  int64
 7   Q7          200000 non-null  int64
 8   Q8          200000 non-null  int64
 9   Q9          200000 non-null  int64
 10  Q10         200000 non-null  int64
 11  Q11         200000 non-null  int64
 12  Q12         200000 non-null  int64
 13  Q13         200000 non-null  int64
 14  Q14         200000 non-null  int64
 15  Q15         200000 non-null  int64
 16  Q16         200000 non-null  int64
 17  Q17         200000 non-null  int64
 18  Q18         200000 non-null  int64
 19  Q19         200000 non-null  int64
 20  Q20         200000 non-null  int64
 21  Q21         200000 non-null  int64
 22  Q22         200000 non-null  int64
 23  Q23         200000 non-null  int64
 24  Q24         200000 non-null  int64
 25  Q25         200000 non-null  int64
 26  Q26         200000 non-null  int64
 27  Q27         200000 non-null  int64
 28  Q28         200000 non-null  int64
 29  Q29         200000 non-null  int64
 30  Q30         200000 non-null  int64
 31  Q31         200000 non-null  int64
 32  Q32         200000 non-null  int64
 33  Q33         200000 non-null  int64
 34  Q34         200000 non-null  int64
 35  Q35         200000 non-null  int64
 36  Q36         200000 non-null  int64
 37  Evaluation  200000 non-null  int64
dtypes: int64(38)
memory usage: 58.0 MB
```

train.csv给了20w条数据，第一列为交通事故案例的Id，其意义不大，可以舍去，第2列至第37列为理赔人员在现场对该事故方采集的36条信息，并已经被编码，用整型表示，最后一列表示是否理赔。上图中显示了各项信息分数的分布。

In [3]:
```
train_df.iloc[:, 1:37].hist(figsize=(20, 20))
```

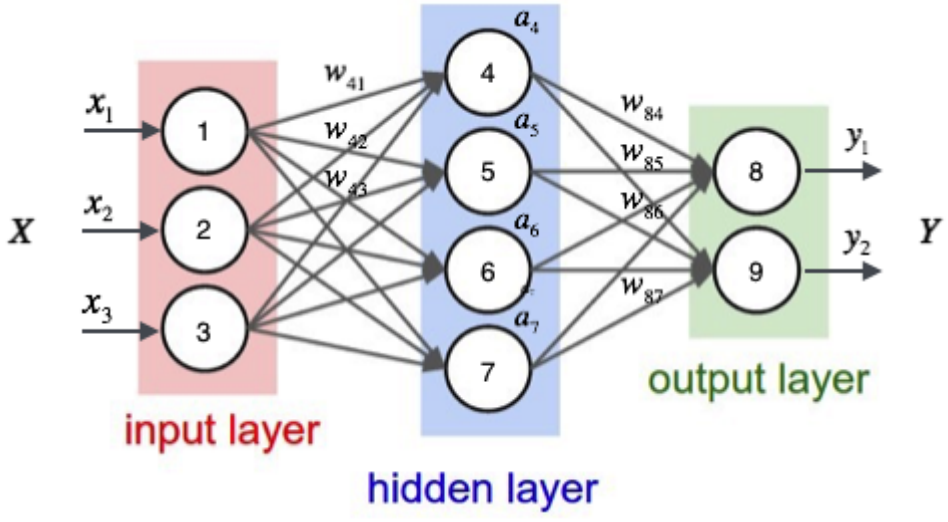
Out[3]:
```
array([[<AxesSubplot:title={'center':'Q1'}>,
        <AxesSubplot:title={'center':'Q2'}>,
        <AxesSubplot:title={'center':'Q3'}>,
        <AxesSubplot:title={'center':'Q4'}>,
        <AxesSubplot:title={'center':'Q5'}>,
        <AxesSubplot:title={'center':'Q6'}>],
       [<AxesSubplot:title={'center':'Q7'}>,
        <AxesSubplot:title={'center':'Q8'}>,
        <AxesSubplot:title={'center':'Q9'}>,
        <AxesSubplot:title={'center':'Q10'}>,
        <AxesSubplot:title={'center':'Q11'}>,
        <AxesSubplot:title={'center':'Q12'}>],
       [<AxesSubplot:title={'center':'Q13'}>,
        <AxesSubplot:title={'center':'Q14'}>,
        <AxesSubplot:title={'center':'Q15'}>,
        <AxesSubplot:title={'center':'Q16'}>,
        <AxesSubplot:title={'center':'Q17'}>,
        <AxesSubplot:title={'center':'Q18'}>],
       [<AxesSubplot:title={'center':'Q19'}>,
        <AxesSubplot:title={'center':'Q20'}>,
        <AxesSubplot:title={'center':'Q21'}>,
        <AxesSubplot:title={'center':'Q22'}>,
        <AxesSubplot:title={'center':'Q23'}>,
        <AxesSubplot:title={'center':'Q24'}>],
       [<AxesSubplot:title={'center':'Q25'}>,
        <AxesSubplot:title={'center':'Q26'}>,
        <AxesSubplot:title={'center':'Q27'}>,
        <AxesSubplot:title={'center':'Q28'}>,
        <AxesSubplot:title={'center':'Q29'}>,
        <AxesSubplot:title={'center':'Q30'}>],
       [<AxesSubplot:title={'center':'Q31'}>,
        <AxesSubplot:title={'center':'Q32'}>,
        <AxesSubplot:title={'center':'Q33'}>,
        <AxesSubplot:title={'center':'Q34'}>,
        <AxesSubplot:title={'center':'Q35'}>,
        <AxesSubplot:title={'center':'Q36'}>]], dtype=object)
```



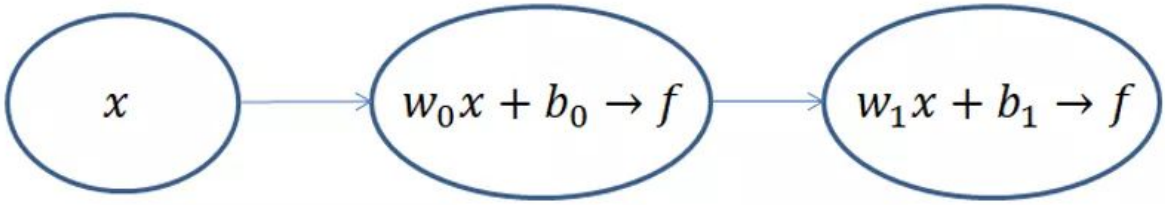
这是一个二分类的问题，并且数据集较大，当特征量并不多，考虑使用全连接神经网络解决。

3. 全连接神经网络

全连接神经网络，顾名思义，就是相邻两层之间任意两个节点之间都有连接。全连接神经网络是最为普通的一种模型（比如和CNN相比），由于是全连接，所以会有更多的权重值和连接，因此也意味着占用更多的内存和计算。其网络结构如下图所示：



其由输入层，隐藏层和输出层组成。当给定输入后，输入层通过一组线性关系公式传给隐藏层，到达隐藏层后，又通过一组线性关系输出给输出层，最后由输出层输出最终的预测值。其示意图如下：

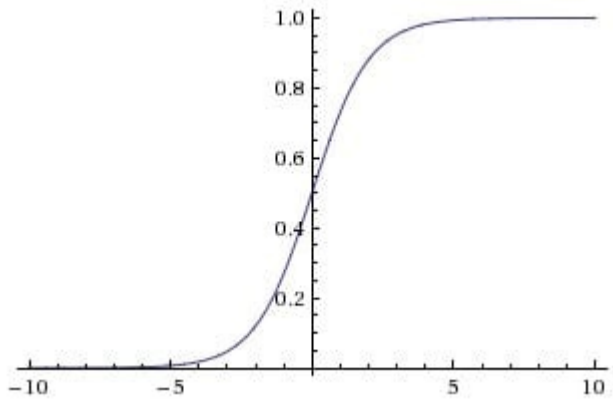


其中x表示前一层传入的输出值，w表示系数矩阵，b表示偏置项。

但光靠上面的结构，其输入值和输出值之间仍然保持线性关系，与普通的逻辑回归模型可以证明是等价的。为了增加非线性特性，数据在从隐藏层和输出层输出前，会通过一层激活函数的计算，由此引入非线性特性，增加模型的表达能力。常见的激活函数有：

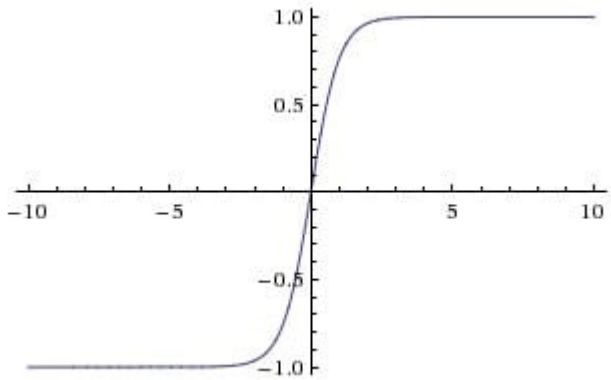
Sigmoid激活函数：

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



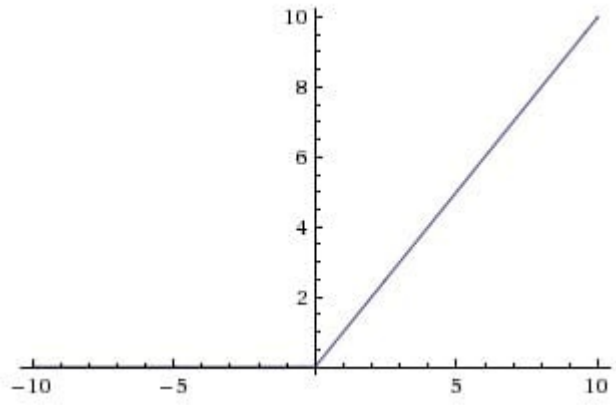
tanh激活函数：

$$\tanh(x) = 2\sigma(2x) - 1$$



ReLU激活函数:

$$\mathrm{Re}\,LU(x) = \max(0, x)$$



如果各层之间的系数矩阵和偏置项已知的话，给定输入值，就可以求出输出值来，这个过程叫做前向传播。但实际情况是，在训练前，模型的系数矩阵和偏置项的中的值是随机的，此时需要根据预测结果与真实值的误差大小修正模型的系数矩阵和偏置项，这一过程称为反向传播。首先，我们定义一个损失函数，用于评估预测结果与真实值之间的误差，本次任务中选用的损失函数为Pytorch中提供的交叉熵函数(CrossEntropyLoss)，其由log_softmax和nll_loss实现。log_softmax为对数softmax函数，其计算公式为：

$$\text{log\_softmax } = \log\left(\frac{\exp(x_i)}{\sum_j \exp(x_j)}\right)$$

nll_loss的计算方式就是将上面输出的值与对应的Label中的类别拿出来去掉负号，用计算公式表示为：

$$\text{nll\_loss } (x,\ \text{class }) = -x[\ \text{class }]$$

再计算出了预测值与输出值之间的损失函数，开始反向传播，逐层求出目标函数对各神经元权值的偏导数，构成目标函数对权值向量的梯量，作为修改权值的依据，网络的学习在权值修改过程中完成。误差达到所期望值时，网络学习结束。Pytorch中内置的数据结构Tensor支持自动微分，使得我们不需要手动给出某一项参数的具体的求导公式，给我们带来了极大的便利。

4. 模型训练

使用Pytorch搭建的全连接神经网络代码如下：

In [4]:

```
import logging
import pickle
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import TensorDataset
from torch.utils.data import DataLoader
from torch.utils.data import random_split

class Insurance_Model(nn.Module):
  def __init__(self, input_dim, hidden_dim, num_classes,
               act_func=F.sigmoid):
    super().__init__()
    # hidden layer
    self.linear_layer1 = nn.Linear(input_dim, hidden_dim)
    # output layer
    self.linear_layer2 = nn.Linear(hidden_dim, num_classes)
    # activation function
    self.act_func = act_func

  def forward(self, inputs):
    outputs = self.linear_layer1(inputs)
    outputs = self.act_func(outputs)
    outputs = self.linear_layer2(outputs)
```

```
    return outputs

  @staticmethod
  def compute_accuracy(outputs, labels):
    _, preds = torch.max(outputs, dim=1)
    return torch.tensor(torch.sum(preds == labels).item() / len(preds))

  @staticmethod
  def log_epoch_loss_and_acc(prefix, epoch, epoch_loss, epoch_acc, interval=5):
    if epoch % interval == 0:
      logging.info(f'{prefix}_Epoch [{epoch}], loss: {epoch_loss:.4f},'
                   f' acc: {epoch_acc:.4f}.')

  def evaluate(self, batch, loss_func, need_acc=False, no_grad=False):
    if no_grad:
      with torch.no_grad():
        inputs, labels = batch
        outputs = self(inputs)
        loss = loss_func(outputs, labels)
    else:
      inputs, labels = batch
      outputs = self(inputs)
      loss = loss_func(outputs, labels)

    if need_acc:
      acc  = self.compute_accuracy(outputs, labels)
      return {'loss': loss, 'acc': acc}
    else:
      return {'loss': loss}

  def compute_epoch_loss_and_acc(self, dataloader, loss_func):
    results = [self.evaluate(batch, loss_func, need_acc=True, no_grad=True)
                                      for batch in dataloader]
    batch_losses = [r['loss'] for r in results]
    epoch_loss   = torch.stack(batch_losses).mean()
    batch_accs   = [r['acc'] for r in results]
    epoch_acc    = torch.stack(batch_accs).mean()
    return {'epoch_loss': epoch_loss, 'epoch_acc': epoch_acc}

  def epoch_postprocess(self, prefix, data_loader, epoch,
                        history, loss_func, log_interval):
    loss_and_acc = self.compute_epoch_loss_and_acc(data_loader, loss_func)
    epoch_loss   = loss_and_acc['epoch_loss']
    epoch_acc    = loss_and_acc['epoch_acc']
    history.append({'epoch_loss': epoch_loss,
                    'epoch_acc': epoch_acc})
    self.log_epoch_loss_and_acc(prefix, epoch,
                                      epoch_loss,
                                      epoch_acc,
                                      log_interval)

  def train(self, train_loader, val_loader, num_epochs, lr,
            loss_func=F.cross_entropy, opt_func=torch.optim.SGD,
            log_interval=5):
    optimizer = opt_func(self.parameters(), lr)
    self.history_train = []  # history of train set
    self.history_val   = []  # history of validation set

    # initial loss and accuracy of training dataset
    self.epoch_postprocess('Train', train_loader, 0,
                           self.history_train, loss_func, log_interval)

    # initial loss and accuracy of validation dataset
    self.epoch_postprocess('Val', val_loader, 0,
                           self.history_val, loss_func, log_interval)
```

```
    # iteration
    for epoch in range(num_epochs):
      for batch in train_loader:
        loss = self.evaluate(batch, loss_func, need_acc=False)['loss']
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

      # training dataset loss and accuracy
      self.epoch_postprocess('Train', train_loader, epoch+1,
                             self.history_train, loss_func, log_interval)

      # validation dataset loss and accuracy
      self.epoch_postprocess('Val', val_loader, epoch+1,
                             self.history_val, loss_func, log_interval)

  def predict(self, inputs):
    outputs = self(inputs)
    _, preds = torch.max(outputs, dim=1)
    return [preds[i].item() for i in range(len(preds))]

  def save_model(self, save_file):
    torch.save(self.state_dict(), save_file)
    pickle.dump(self.history_train, open('insurance_history_train.pkl', 'wb'))
    pickle.dump(self.history_val, open('insurance_history_val.pkl', 'wb'))


  def recover_model(self, save_file):
    self.load_state_dict(torch.load(save_file))
    self.history_train = pickle.load(open('insurance_history_train.pkl', 'rb'))
    self.history_val   = pickle.load(open('insurance_history_val.pkl', 'rb'))
```

分割训练集，按5:1的比例划分训练集和验证集。

In [5]:

```
# delete CaseId (because it has no meaning)
train_df.drop('CaseId', axis=1, inplace=True)

# convert pandas dataframe to numpy array
train_data = train_df.to_numpy()

# convert numpy array to tensor
inputs = torch.from_numpy(train_data[:, :36]).type(torch.float)
labels = torch.from_numpy(train_data[:, 36]).type(torch.long)

dataset = TensorDataset(inputs, labels)
train_ds, val_ds = random_split(dataset, [166666, 33334])
```

使用gpu加速计算，Pytorch中使用gpu计算十分简单，只需要将训练数据和模型参数转移到显存中即可(前提是配置好cuda驱动)。

In [6]:

```
def to_device(data, device):
  """Move tensor(s) to chosen device"""
  if isinstance(data, (list,tuple)):
    return [to_device(x, device) for x in data]
  return data.to(device, non_blocking=True)

class DeviceDataLoader():
  """Wrap a dataloader to move data to a device (default: cpu)"""
  def __init__(self, dl, device):
    self.dl = dl
    self.device = device

  def __iter__(self):
    """Yield a batch of data after moving it to device"""
```

```
    for b in self.dl:
      yield to_device(b, self.device)

  def __len__(self):
    """Number of batches"""
    return len(self.dl)
```

In [7]:
```
logging.basicConfig(format='%(asctime)s %(levelname)s:%(message)s', \
                    level=logging.INFO, datefmt='%m/%d/%Y %I:%M:%S %p')
```

选用Simoid作为激活函数，学习速率选为0.01，迭代步数为100时：

In [8]:
```
x_dim         = 36      # input dimension
y_dim         = 2       # label dimension
hidden_dim    = 24      # hidden layer dimension
act_func      = F.sigmoid  # activation function
batch_size    = 128
num_epochs    = 100
learning_rate = 0.01
device = torch.device('cuda')

train_loader = DataLoader(train_ds, batch_size, shuffle=True)
val_loader   = DataLoader(val_ds, batch_size)
# move dataloader to gpu
train_loader = DeviceDataLoader(train_loader, device)
val_loader   = DeviceDataLoader(val_loader, device)

# initialize linear regression model
logging.info("Initializing NN model.")
insurance_model = Insurance_Model(x_dim, hidden_dim, y_dim, act_func)
# move model parameters to gpu
to_device(insurance_model, device)
logging.info("Start training ... ")
insurance_model.train(train_loader, val_loader, num_epochs,
                      learning_rate, log_interval=10, opt_func=torch.optim.SGD)
logging.info("Training finished.")

logging.info("Save model.")
insurance_model.save_model('report01-insurance_model.pth')
```

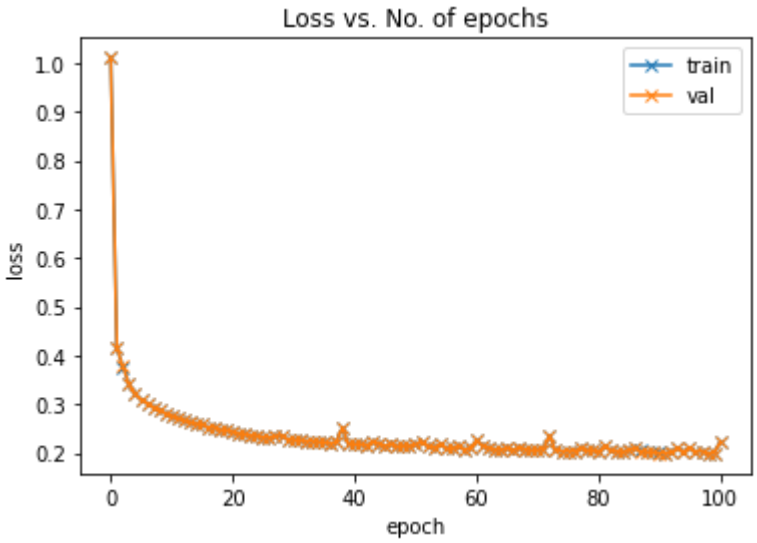
```
03/07/2022 10:07:18 AM INFO:Initializing NN model.
03/07/2022 10:07:20 AM INFO:Start training ...
c:\users\sj2050\miniconda3\lib\site-packages\torch\nn\functional.py:1806: UserWarnin
g: nn.functional.sigmoid is deprecated. Use torch.sigmoid instead.
  warnings.warn("nn.functional.sigmoid is deprecated. Use torch.sigmoid instead.")
03/07/2022 10:07:22 AM INFO:Train_Epoch [0], loss: 1.0106, acc: 0.1578.
03/07/2022 10:07:22 AM INFO:Val_Epoch [0], loss: 1.0096, acc: 0.1587.
03/07/2022 10:08:01 AM INFO:Train_Epoch [10], loss: 0.2757, acc: 0.8635.
03/07/2022 10:08:01 AM INFO:Val_Epoch [10], loss: 0.2757, acc: 0.8634.
03/07/2022 10:08:37 AM INFO:Train_Epoch [20], loss: 0.2434, acc: 0.8928.
03/07/2022 10:08:37 AM INFO:Val_Epoch [20], loss: 0.2435, acc: 0.8927.
03/07/2022 10:09:14 AM INFO:Train_Epoch [30], loss: 0.2261, acc: 0.9054.
03/07/2022 10:09:15 AM INFO:Val_Epoch [30], loss: 0.2264, acc: 0.9049.
03/07/2022 10:09:52 AM INFO:Train_Epoch [40], loss: 0.2175, acc: 0.9079.
03/07/2022 10:09:52 AM INFO:Val_Epoch [40], loss: 0.2177, acc: 0.9075.
03/07/2022 10:10:30 AM INFO:Train_Epoch [50], loss: 0.2168, acc: 0.9067.
03/07/2022 10:10:31 AM INFO:Val_Epoch [50], loss: 0.2170, acc: 0.9062.
03/07/2022 10:11:08 AM INFO:Train_Epoch [60], loss: 0.2263, acc: 0.9001.
03/07/2022 10:11:09 AM INFO:Val_Epoch [60], loss: 0.2265, acc: 0.8996.
03/07/2022 10:11:46 AM INFO:Train_Epoch [70], loss: 0.2059, acc: 0.9108.
03/07/2022 10:11:47 AM INFO:Val_Epoch [70], loss: 0.2059, acc: 0.9107.
03/07/2022 10:12:25 AM INFO:Train_Epoch [80], loss: 0.2027, acc: 0.9124.
03/07/2022 10:12:25 AM INFO:Val_Epoch [80], loss: 0.2023, acc: 0.9122.
03/07/2022 10:13:02 AM INFO:Train_Epoch [90], loss: 0.2006, acc: 0.9120.
03/07/2022 10:13:03 AM INFO:Val_Epoch [90], loss: 0.1999, acc: 0.9121.
03/07/2022 10:13:41 AM INFO:Train_Epoch [100], loss: 0.2240, acc: 0.8994.
03/07/2022 10:13:42 AM INFO:Val_Epoch [100], loss: 0.2236, acc: 0.8997.
03/07/2022 10:13:42 AM INFO:Training finished.
03/07/2022 10:13:42 AM INFO:Save model.
```

训练过程中的损失函数值和准确率变化：

In [9]:
```
history_train = pickle.load(open('insurance_history_train.pkl', 'rb'))
history_val = pickle.load(open('insurance_history_val.pkl', 'rb'))
```

In [10]:
```
import matplotlib.pyplot as plt

train_losses = [float(x['epoch_loss']) for x in history_train]
val_losses = [float(x['epoch_loss']) for x in history_val]
plt.plot(train_losses, '-x', label='train')
plt.plot(val_losses, '-x', label='val')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.title('Loss vs. No. of epochs')
```

Out[10]:
```
Text(0.5, 1.0, 'Loss vs. No. of epochs')
```

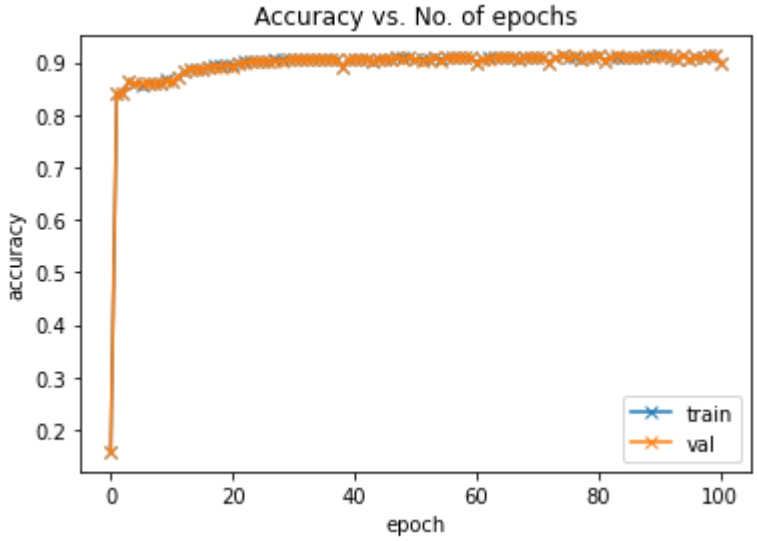


In [11]:

```
train_accs = [float(x['epoch_acc']) for x in history_train]
val_accs = [float(x['epoch_acc']) for x in history_val]
plt.plot(train_accs, '-x', label='train')
plt.plot(val_accs, '-x', label='val')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.legend()
plt.title('Accuracy vs. No. of epochs')
```

Out[11]:

```
Text(0.5, 1.0, 'Accuracy vs. No. of epochs')
```



选用ReLU作为激活函数，学习速率选为0.01，迭代步数为100时：

In [24]:

```
x_dim         = 36      # input dimension
y_dim         = 2       # label dimension
hidden_dim    = 24      # hidden layer dimension
act_func      = F.relu  # activation function
batch_size    = 64
num_epochs    = 100
learning_rate = 0.01
device = torch.device('cuda')

train_loader = DataLoader(train_ds, batch_size, shuffle=True)
val_loader   = DataLoader(val_ds, batch_size)
# move dataloader to gpu
train_loader = DeviceDataLoader(train_loader, device)
val_loader   = DeviceDataLoader(val_loader, device)

# initialize linear regression model
```

```
logging.info("Initializing NN model.")
insurance_model = Insurance_Model(x_dim, hidden_dim, y_dim, act_func)
# move model parameters to gpu
to_device(insurance_model, device)
logging.info("Start training ... ")
insurance_model.train(train_loader, val_loader, num_epochs,
                      learning_rate, log_interval=10, opt_func=torch.optim.SGD)
logging.info("Training finished.")

logging.info("Save model.")
insurance_model.save_model('report01-insurance_model.pth')
```

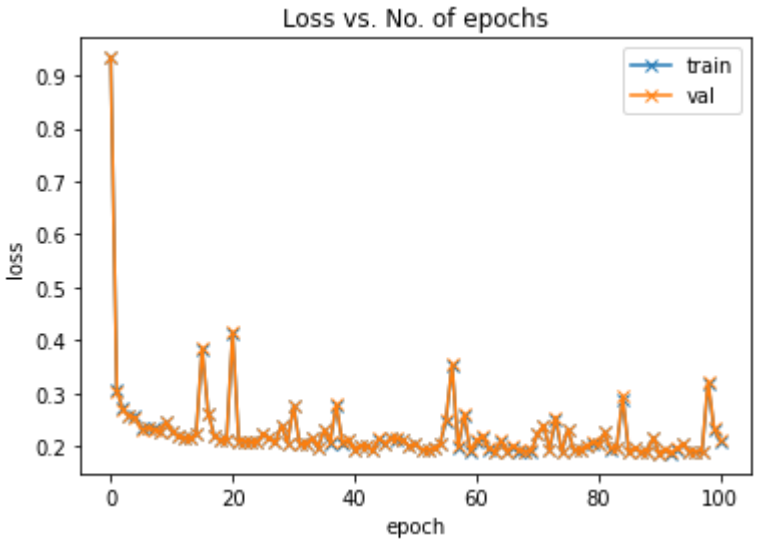
```
03/07/2022 10:43:29 AM INFO:Initializing NN model.
03/07/2022 10:43:29 AM INFO:Start training ...
03/07/2022 10:43:32 AM INFO:Train_Epoch [0], loss: 0.9338, acc: 0.1584.
03/07/2022 10:43:32 AM INFO:Val_Epoch [0], loss: 0.9337, acc: 0.1592.
03/07/2022 10:44:25 AM INFO:Train_Epoch [10], loss: 0.2281, acc: 0.9040.
03/07/2022 10:44:26 AM INFO:Val_Epoch [10], loss: 0.2285, acc: 0.9018.
03/07/2022 10:45:19 AM INFO:Train_Epoch [20], loss: 0.4112, acc: 0.8306.
03/07/2022 10:45:19 AM INFO:Val_Epoch [20], loss: 0.4156, acc: 0.8270.
03/07/2022 10:46:11 AM INFO:Train_Epoch [30], loss: 0.2760, acc: 0.8954.
03/07/2022 10:46:11 AM INFO:Val_Epoch [30], loss: 0.2749, acc: 0.8941.
03/07/2022 10:47:01 AM INFO:Train_Epoch [40], loss: 0.1933, acc: 0.9181.
03/07/2022 10:47:02 AM INFO:Val_Epoch [40], loss: 0.1944, acc: 0.9163.
03/07/2022 10:47:52 AM INFO:Train_Epoch [50], loss: 0.2034, acc: 0.9143.
03/07/2022 10:47:52 AM INFO:Val_Epoch [50], loss: 0.2034, acc: 0.9133.
03/07/2022 10:48:44 AM INFO:Train_Epoch [60], loss: 0.2081, acc: 0.9059.
03/07/2022 10:48:44 AM INFO:Val_Epoch [60], loss: 0.2101, acc: 0.9029.
03/07/2022 10:49:34 AM INFO:Train_Epoch [70], loss: 0.2219, acc: 0.9050.
03/07/2022 10:49:34 AM INFO:Val_Epoch [70], loss: 0.2222, acc: 0.9034.
03/07/2022 10:50:25 AM INFO:Train_Epoch [80], loss: 0.2025, acc: 0.9154.
03/07/2022 10:50:25 AM INFO:Val_Epoch [80], loss: 0.2023, acc: 0.9141.
03/07/2022 10:51:18 AM INFO:Train_Epoch [90], loss: 0.1857, acc: 0.9202.
03/07/2022 10:51:18 AM INFO:Val_Epoch [90], loss: 0.1867, acc: 0.9181.
03/07/2022 10:52:09 AM INFO:Train_Epoch [100], loss: 0.2087, acc: 0.9052.
03/07/2022 10:52:09 AM INFO:Val_Epoch [100], loss: 0.2113, acc: 0.9026.
03/07/2022 10:52:09 AM INFO:Training finished.
03/07/2022 10:52:09 AM INFO:Save model.
```

训练过程中的损失函数值和准确率变化：

In [25]:
```
history_train = pickle.load(open('insurance_history_train.pkl', 'rb'))
history_val = pickle.load(open('insurance_history_val.pkl', 'rb'))
```

In [26]:
```
import matplotlib.pyplot as plt

train_losses = [float(x['epoch_loss']) for x in history_train]
val_losses = [float(x['epoch_loss']) for x in history_val]
plt.plot(train_losses, '-x', label='train')
plt.plot(val_losses, '-x', label='val')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.title('Loss vs. No. of epochs')
```

Out[26]:
```
Text(0.5, 1.0, 'Loss vs. No. of epochs')
```

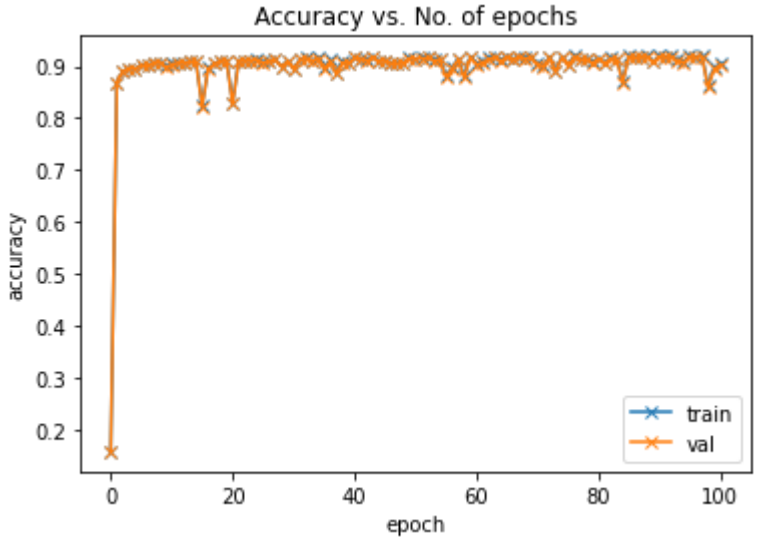


In [27]:

```
train_accs = [float(x['epoch_acc']) for x in history_train]
val_accs = [float(x['epoch_acc']) for x in history_val]
plt.plot(train_accs, '-x', label='train')
plt.plot(val_accs, '-x', label='val')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.legend()
plt.title('Accuracy vs. No. of epochs')
```

Out[27]:

```
Text(0.5, 1.0, 'Accuracy vs. No. of epochs')
```



选用tanh作为激活函数，学习速率选为0.01，迭代步数为100时：

In [16]:

```
x_dim         = 36      # input dimension
y_dim         = 2       # label dimension
hidden_dim    = 24      # hidden layer dimension
act_func      = F.tanh  # activation function
batch_size    = 128
num_epochs    = 100
learning_rate = 0.01
device = torch.device('cuda')

train_loader = DataLoader(train_ds, batch_size, shuffle=True)
val_loader   = DataLoader(val_ds, batch_size)
# move dataloader to gpu
train_loader = DeviceDataLoader(train_loader, device)
val_loader   = DeviceDataLoader(val_loader, device)

# initialize linear regression model
```

```
logging.info("Initializing NN model.")
insurance_model = Insurance_Model(x_dim, hidden_dim, y_dim, act_func)
# move model parameters to gpu
to_device(insurance_model, device)
logging.info("Start training ... ")
insurance_model.train(train_loader, val_loader, num_epochs,
                      learning_rate, log_interval=10, opt_func=torch.optim.SGD)
logging.info("Training finished.")

logging.info("Save model.")
insurance_model.save_model('report01-insurance_model.pth')
```

```
03/07/2022 10:20:06 AM INFO:Initializing NN model.
03/07/2022 10:20:06 AM INFO:Start training ...
c:\users\sj2050\miniconda3\lib\site-packages\torch\nn\functional.py:1795: UserWarnin
g: nn.functional.tanh is deprecated. Use torch.tanh instead.
  warnings.warn("nn.functional.tanh is deprecated. Use torch.tanh instead.")
03/07/2022 10:20:08 AM INFO:Train_Epoch [0], loss: 0.7299, acc: 0.3735.
03/07/2022 10:20:08 AM INFO:Val_Epoch [0], loss: 0.7296, acc: 0.3737.
03/07/2022 10:20:45 AM INFO:Train_Epoch [10], loss: 0.2216, acc: 0.9048.
03/07/2022 10:20:45 AM INFO:Val_Epoch [10], loss: 0.2220, acc: 0.9038.
03/07/2022 10:21:23 AM INFO:Train_Epoch [20], loss: 0.2235, acc: 0.9028.
03/07/2022 10:21:23 AM INFO:Val_Epoch [20], loss: 0.2231, acc: 0.9029.
03/07/2022 10:22:00 AM INFO:Train_Epoch [30], loss: 0.2048, acc: 0.9060.
03/07/2022 10:22:00 AM INFO:Val_Epoch [30], loss: 0.2037, acc: 0.9056.
03/07/2022 10:22:39 AM INFO:Train_Epoch [40], loss: 0.2216, acc: 0.9017.
03/07/2022 10:22:39 AM INFO:Val_Epoch [40], loss: 0.2223, acc: 0.8999.
03/07/2022 10:23:20 AM INFO:Train_Epoch [50], loss: 0.2089, acc: 0.9097.
03/07/2022 10:23:20 AM INFO:Val_Epoch [50], loss: 0.2084, acc: 0.9088.
03/07/2022 10:23:57 AM INFO:Train_Epoch [60], loss: 0.2126, acc: 0.9048.
03/07/2022 10:23:57 AM INFO:Val_Epoch [60], loss: 0.2136, acc: 0.9030.
03/07/2022 10:24:34 AM INFO:Train_Epoch [70], loss: 0.1915, acc: 0.9177.
03/07/2022 10:24:34 AM INFO:Val_Epoch [70], loss: 0.1924, acc: 0.9145.
03/07/2022 10:25:13 AM INFO:Train_Epoch [80], loss: 0.1916, acc: 0.9195.
03/07/2022 10:25:13 AM INFO:Val_Epoch [80], loss: 0.1922, acc: 0.9180.
03/07/2022 10:25:49 AM INFO:Train_Epoch [90], loss: 0.2093, acc: 0.9069.
03/07/2022 10:25:49 AM INFO:Val_Epoch [90], loss: 0.2096, acc: 0.9054.
03/07/2022 10:26:26 AM INFO:Train_Epoch [100], loss: 0.1877, acc: 0.9205.
03/07/2022 10:26:26 AM INFO:Val_Epoch [100], loss: 0.1887, acc: 0.9184.
03/07/2022 10:26:26 AM INFO:Training finished.
03/07/2022 10:26:26 AM INFO:Save model.
```

训练过程中的损失函数值和准确率变化：

In [17]:

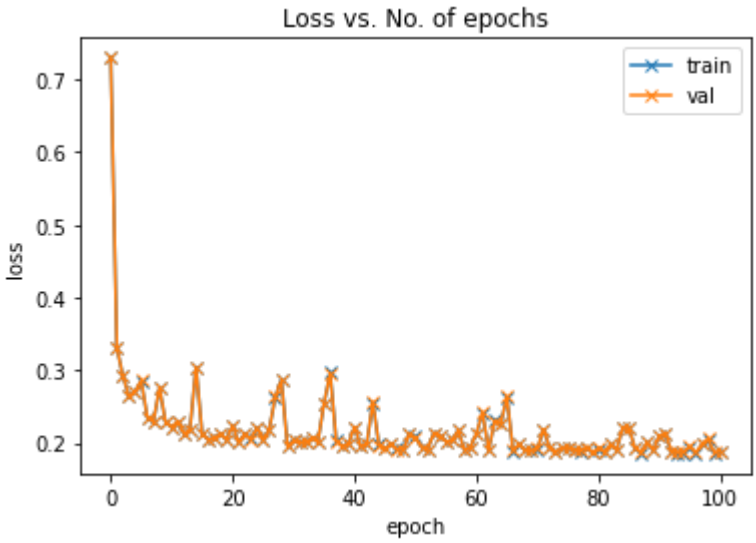
```
history_train = pickle.load(open('insurance_history_train.pkl', 'rb'))
history_val = pickle.load(open('insurance_history_val.pkl', 'rb'))
```

In [18]:

```
import matplotlib.pyplot as plt

train_losses = [float(x['epoch_loss']) for x in history_train]
val_losses = [float(x['epoch_loss']) for x in history_val]
plt.plot(train_losses, '-x', label='train')
plt.plot(val_losses, '-x', label='val')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.title('Loss vs. No. of epochs')
```

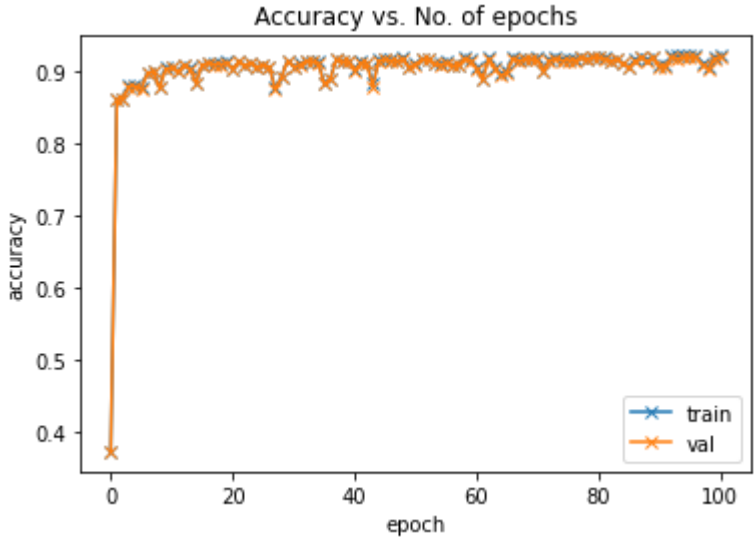
Out[18]:

```
Text(0.5, 1.0, 'Loss vs. No. of epochs')
```



In [19]:
```
train_accs = [float(x['epoch_acc']) for x in history_train]
val_accs = [float(x['epoch_acc']) for x in history_val]
plt.plot(train_accs, '-x', label='train')
plt.plot(val_accs, '-x', label='val')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.legend()
plt.title('Accuracy vs. No. of epochs')
```

Out[19]:
```
Text(0.5, 1.0, 'Accuracy vs. No. of epochs')
```



可以看到，选用tanh激活函数的模型最终训练出来的精度最佳，在验证集上可以达到91.84%的准确度。

5. 小结

在本次任务中，我们使用了全连接神经网络进行了交通事故理赔审核预测。这是我第一次使用Pytorch进行实战开发，在开发中遇到了不少问题，例如从训练样本文件中读入输入值和标签值时，需考虑读入的类型，例如标签label需要为整型，如果类型不当，Pytorch将会爆出RuntimeError: expected scalar type Long but found Float的错误。如果需要使用gpu加速计算，需要安装对应的显卡驱动，并且gpu版的Pytorch和普通版本的Pytorch并不一致，需要卸载掉原有的Pytorch再重新安装gpu版的Pytorch才能使用到gpu加速功能。同时，在开发中，也感受到了Pytorch的魅力，其所有操纵都十分简洁，但功能齐全，与平时作业手写神经网络相比，大大提升了开发效率。此外，在该任务中，我们还测试了不同的激活函数下的训练精度和效果，结果发现tanh激活函数取得效果最好。

Report02 - 泰坦尼克号的生存预测

- 沈键
- 2021200082

1. 任务简介

泰坦尼克号沉船事故是世界上最著名的沉船事故之一。1912年4月15日，在她的处女航期间，泰坦尼克号撞上冰山后沉没，造成2224名乘客和机组人员中超过1502人的死亡。这一轰动的悲剧震惊了国际社会，并因此建立了更好的船舶安全法规。 事故中导致死亡的一个原因是许多船员和乘客没有足够的救生艇。然而在被获救群体中也有一些比较幸运的因素；一些人群在事故中被救的几率高于其他人，比如妇女、儿童和上层阶级。 这个任务中，我们需要分析和判断出什么样的人更容易获救，然后要利用机器学习来预测出在这场灾难中哪些人会最终获救。

2. 分析数据

本次任务中，给了一个名为train.csv的数据集文件，里面有891名乘客的数据。首先，我们使用pandas读取该文件，查看有哪些变量。

In [1]:
```
import pandas as pd

train_df = pd.read_csv("./data/train.csv")
train_df.head(10)
```

Out[1]:

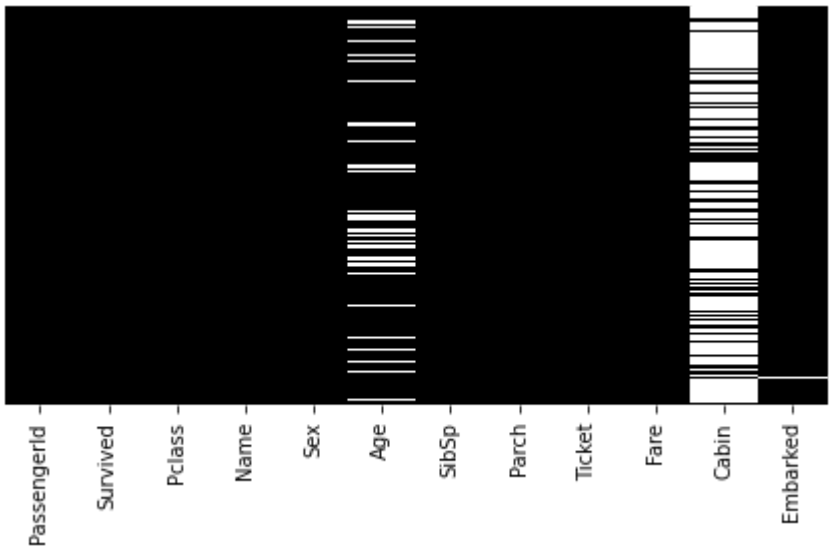
	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embar
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	
5	6	0	3	Moran, Mr. James	male	NaN	0	0	330877	8.4583	NaN	
6	7	0	1	McCarthy, Mr. Timothy J	male	54.0	0	0	17463	51.8625	E46	
7	8	0	3	Palsson, Master. Gosta Leonard	male	2.0	3	1	349909	21.0750	NaN	
8	9	1	3	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)	female	27.0	0	2	347742	11.1333	NaN	
9	10	1	2	Nasser, Mrs. Nicholas (Adele Achem)	female	14.0	1	0	237736	30.0708	NaN	

通过查看前几行，可以看到，每一行数据包括乘客的id、乘客等级、名字、性别、年龄、和该乘客一起旅行的兄弟姐妹和配偶的数量、和该乘客一起旅行的父母和孩子的数量、船票号、船票价格、船舱号、登船港口(S=英国南安普顿Southampton(起航点)/C=法国 瑟堡市Cherbourg(途经点)/Q=爱尔兰 昆士Queenstown(途经点))以及该乘客对应的最终是否存活。通过观察各列数据，还可以看出，某些列上存在缺失值，pandas带入数据集文件后，缺失值处会显示为NaN。通过热力图，可以更加直观的展示缺失值的分布情况。

In [2]:

```
import seaborn as sns
import matplotlib.pyplot as plt

%matplotlib inline
```

```
sns.heatmap(train_df.isnull(), yticklabels=False, cbar=False, cmap='CMRmap')
plt.tight_layout()
plt.show()
```



通过将缺失值处的位置通过热力图高亮显示，可以看到，年龄(Age)、船舱号(Cabin)和登船港口(Embarked)列存在缺失值，并且，年龄和船舱号的缺失值很多。

In [3]:
```
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   PassengerId  891 non-null    int64  
 1   Survived     891 non-null    int64  
 2   Pclass       891 non-null    int64  
 3   Name         891 non-null    object 
 4   Sex          891 non-null    object 
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64  
 7   Parch        891 non-null    int64  
 8   Ticket       891 non-null    object 
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object 
 11  Embarked     889 non-null    object 
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

年龄列的缺失率为(891-714)/891*100%=19.9%, 船舱号列的缺失率为(891-204)/891*100%=77.1%, 登船港口列的缺失率为(891-889)/891*100%=0.2%。可以看出船舱号列的缺失率很高，所以我们将其忽略，同时，乘客id和船票号意义不大，我们也将其忽略。接下里，我们分别研究其余因素对获救率的影响。

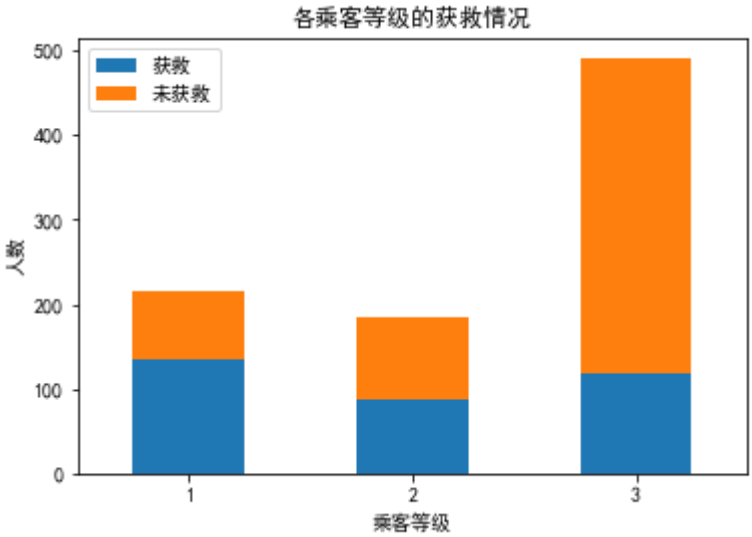
In [4]:
```
train_df.drop('PassengerId', axis=1, inplace=True)
train_df.drop('Ticket', axis=1, inplace=True)
train_df.drop('Cabin', axis=1, inplace=True)
```

2.1 乘客等级对获救率的影响

In [5]:
```
from pylab import *

mpl.rcParams['font.sans-serif'] = ['SimHei']
matplotlib.rcParams['axes.unicode_minus'] = False
```

```
fig = plt.figure()
fig.set(alpha=0.2)

Suvived_0 = train_df.Pclass[train_df.Survived == 0].value_counts()
Suvived_1 = train_df.Pclass[train_df.Survived == 1].value_counts()
df = pd.DataFrame({u"获救": Suvived_1, u"未获救": Suvived_0})
df.plot(kind='bar', stacked=True)
plt.xticks(rotation=360)
plt.title(u'各乘客等级的获救情况')
plt.xlabel(u'乘客等级')
plt.ylabel(u'人数')
plt.show()
```

```
<Figure size 432x288 with 0 Axes>
```



可以看到，乘客等级为1的获救率最高，乘客等级为2的次之，乘客等级为3的获救率最低。显然，富人的获救率比穷人的获救率更高，且等级高的对应舱的救援设备一般都会好于等级低的船舱。

2.2 名字对获救率的影响

观察乘客名字列，可以发现，乘客名字的中间部分对应了乘客目前的社会头衔或者已婚情况。

In [6]:

```
import re

def get_title(name):
    title_search = re.search('([A-Za-z]+)\.',name)
    if title_search:
        return title_search.group(1)
    return

titles = train_df["Name"].apply(get_title)
print(pd.value_counts(titles))
```

```
Mr          517
Miss        182
Mrs         125
Master       40
Dr            7
Rev           6
Major         2
Mlle          2
Col           2
Countess      1
Lady          1
Mme           1
Jonkheer      1
Don           1
Ms            1
Capt          1
Sir           1
Name: Name, dtype: int64
```

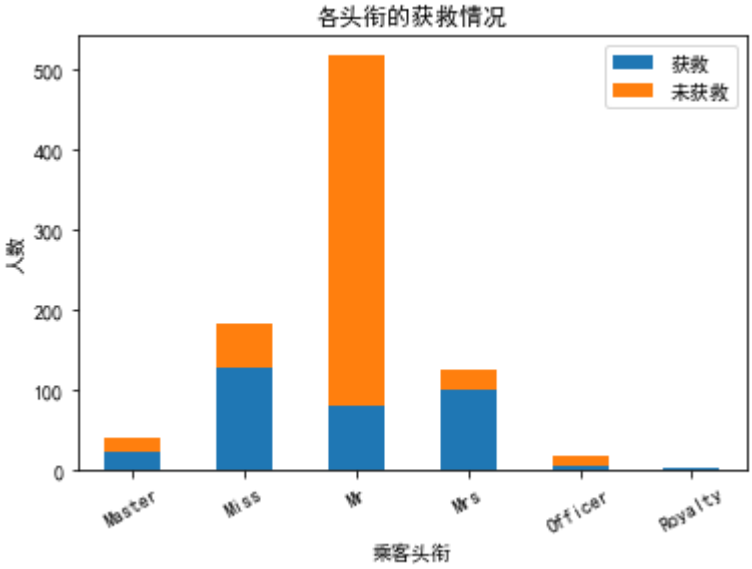
例如Miss表示未婚的女士，而Mrs为已婚的女士，Capt、Col、Major、Dr、Rev头衔的可认为是政府官员，Don、Sir、Countess、Lady头衔的可认为是皇室成员，Master和Jonkheer表示有技能的人，如果出现其他头衔，则分到Others类。

In [7]:

```
# 将Name列改名为Title
train_df.rename(columns={'Name':'Title'}, inplace=True)
train_df['Title'] = train_df['Title'].apply(get_title)
title_classification = {'Officer':['Capt', 'Col', 'Major', 'Dr', 'Rev'],
                        'Royalty':['Don', 'Sir', 'Countess', 'Lady'],
                        'Mrs':['Mme', 'Ms', 'Mrs'],
                        'Miss':['Mlle', 'Miss'],
                        'Mr':['Mr'],
                        'Master':['Master','Jonkheer']}
title_map = {}
for title in title_classification.keys():
    title_map.update(dict.fromkeys(title_classification[title], title))

train_df['Title'] = train_df['Title'].map(title_map)
```

In [8]:

```
Suvived_0 = train_df.Title[train_df.Survived == 0].value_counts()
Suvived_1 = train_df.Title[train_df.Survived == 1].value_counts()
df = pd.DataFrame({u"获救": Suvived_1, u"未获救": Suvived_0})
df.plot(kind='bar', stacked=True)
plt.xticks(rotation=30)
plt.title(u'各头衔的获救情况')
plt.xlabel(u'乘客头衔')
plt.ylabel(u'人数')
plt.show()
```



可以看到，不同头衔对应的获救情况不同，其中，已婚女士和未婚女士的获救率较高，而男士的获救率较低，这与社会上提倡女士优先有着密切的关系，同时， 皇室和有技能的人的获救率也很高，这与他们的社会地位有关，社会地位高的人往往会被优先救援。

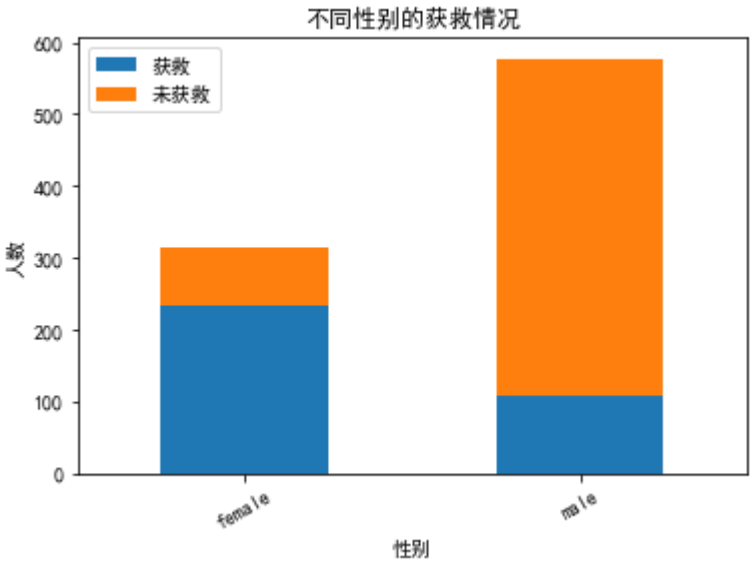
In [9]:
```
train_df.head(10)
```

Out[9]:

	Survived	Pclass	Title	Sex	Age	SibSp	Parch	Fare	Embarked
0	0	3	Mr	male	22.0	1	0	7.2500	S
1	1	1	Mrs	female	38.0	1	0	71.2833	C
2	1	3	Miss	female	26.0	0	0	7.9250	S
3	1	1	Mrs	female	35.0	1	0	53.1000	S
4	0	3	Mr	male	35.0	0	0	8.0500	S
5	0	3	Mr	male	NaN	0	0	8.4583	Q
6	0	1	Mr	male	54.0	0	0	51.8625	S
7	0	3	Master	male	2.0	3	1	21.0750	S
8	1	3	Mrs	female	27.0	0	2	11.1333	S
9	1	2	Mrs	female	14.0	1	0	30.0708	C

2.3 性别对获救率的影响

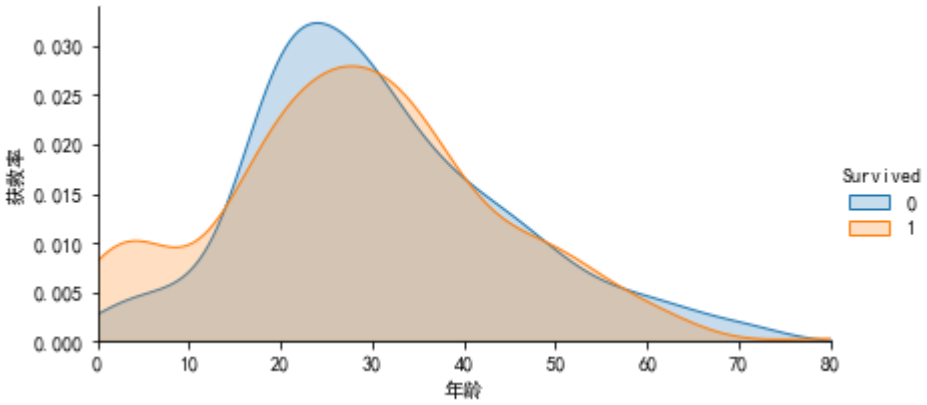
In [10]:
```
Suvived_0 = train_df.Sex[train_df.Survived == 0].value_counts()
Suvived_1 = train_df.Sex[train_df.Survived == 1].value_counts()
df = pd.DataFrame({u"获救": Suvived_1, u"未获救": Suvived_0})
df.plot(kind='bar', stacked=True)
plt.xticks(rotation=30)
plt.title(u'不同性别的获救情况')
plt.xlabel(u'性别')
plt.ylabel(u'人数')
plt.show()
```



如同在头衔分析中提到的一样，女士的获救率高于男士，与社会上提倡女士优先有着密切的关系。

2.4 年龄对获救率的影响

In [11]:
```
facet = sns.FacetGrid(train_df, hue="Survived",aspect=2)
facet.map(sns.kdeplot,'Age',shade= True)
facet.set(xlim=(0, train_df['Age'].max()))
facet.add_legend()
plt.xlabel(u'年龄')
plt.ylabel(u'获救率')
plt.show()
```

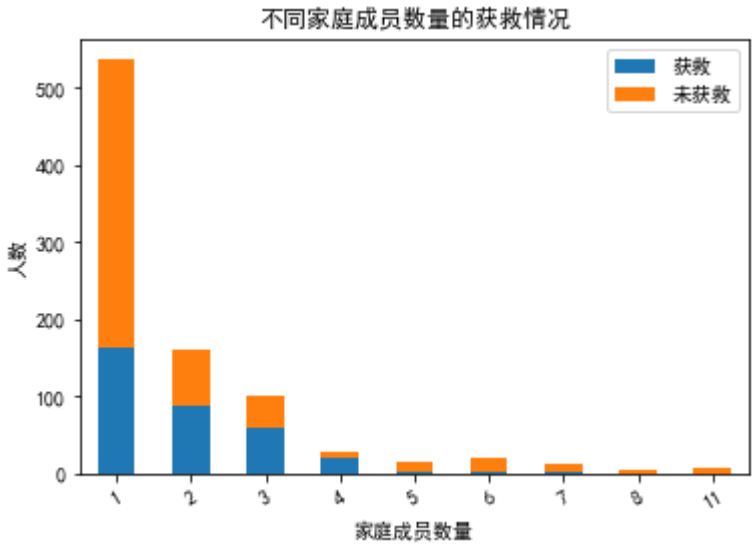


从上图中可以看出，年龄(<=15)小存活率高，年龄大(>=60)的存活率低，这与社会对小孩的保护以及老年人体力不行有关。

2.5 家庭成员数量对获救率的影响

家庭成员包括一起旅行的兄弟姐妹和配偶的数量以及一起旅行的父母和孩子的数量。

In [12]:
```
train_df['FamilySize'] = train_df['SibSp'] + train_df['Parch'] + 1
Suvived_0 = train_df.FamilySize[train_df.Survived == 0].value_counts()
Suvived_1 = train_df.FamilySize[train_df.Survived == 1].value_counts()
df = pd.DataFrame({u"获救": Suvived_1, u"未获救": Suvived_0})
df.plot(kind='bar', stacked=True)
plt.xticks(rotation=30)
plt.title(u'不同家庭成员数量的获救情况')
plt.xlabel(u'家庭成员数量')
plt.ylabel(u'人数')
plt.show()

plt.show()
```

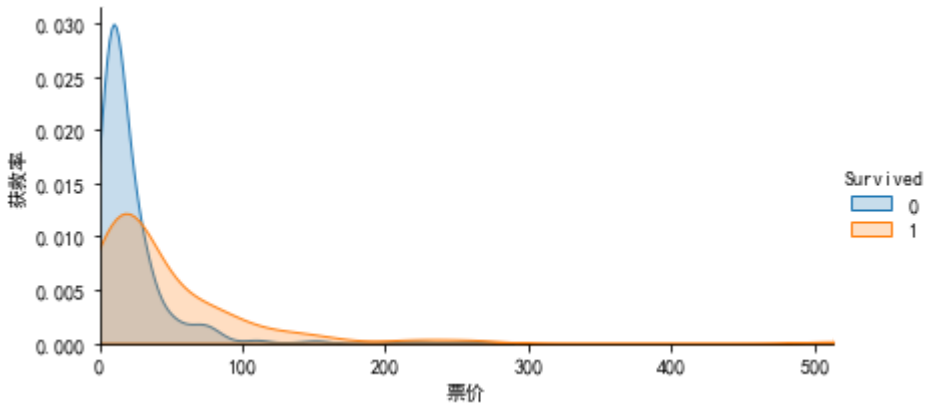


In [13]:
```
train_df.drop('SibSp', axis=1, inplace=True)
train_df.drop('Parch', axis=1, inplace=True)
```

船上家庭成员在3~4个左右的获救率最高，家庭成员太少或太多存活率都不高。一般而言，一个家庭里的成员会优先救自己本家庭中的人，但如果家庭人员太多，也会出现耽误了太多时间救援家庭成员而导致自己未及时离开的问题。

2.6 票价对获救率的影响

In [14]:
```
facet = sns.FacetGrid(train_df, hue="Survived",aspect=2)
facet.map(sns.kdeplot,'Fare',shade= True)
facet.set(xlim=(0, train_df['Fare'].max()))
facet.add_legend()
plt.xlabel(u'票价')
plt.ylabel(u'获救率')
plt.show()
```

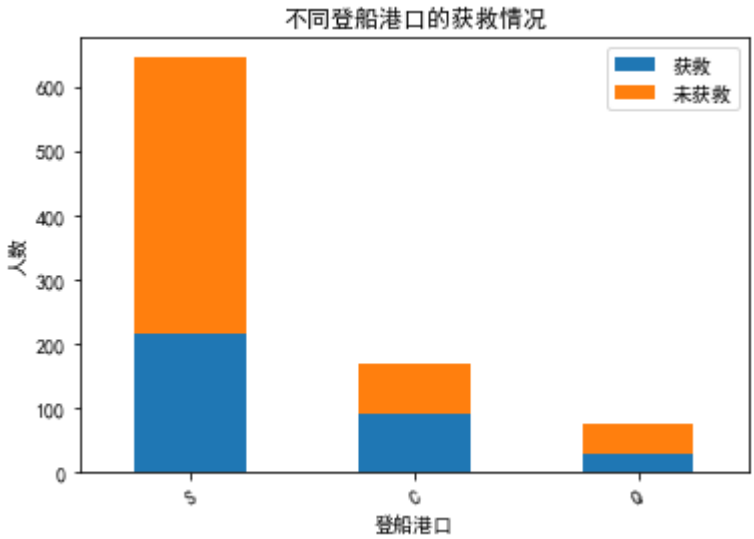


一般而言，票价高的往往位置比较好，救援设施也更好，获救率也越高。

2.7 登船港口对获救率的影响

In [15]:
```
Suvived_0 = train_df.Embarked[train_df.Survived == 0].value_counts()
Suvived_1 = train_df.Embarked[train_df.Survived == 1].value_counts()
df = pd.DataFrame({u"获救": Suvived_1, u"未获救": Suvived_0})
df.plot(kind='bar', stacked=True)
plt.xticks(rotation=30)
plt.title(u'不同登船港口的获救情况')
plt.xlabel(u'登船港口')
```

```
plt.ylabel(u'人数')
plt.show()
```



船港口不同，生存率不同。在C港口上船的获救率最高，而在S港口上船的获救率最低。

3. 逻辑回归模型

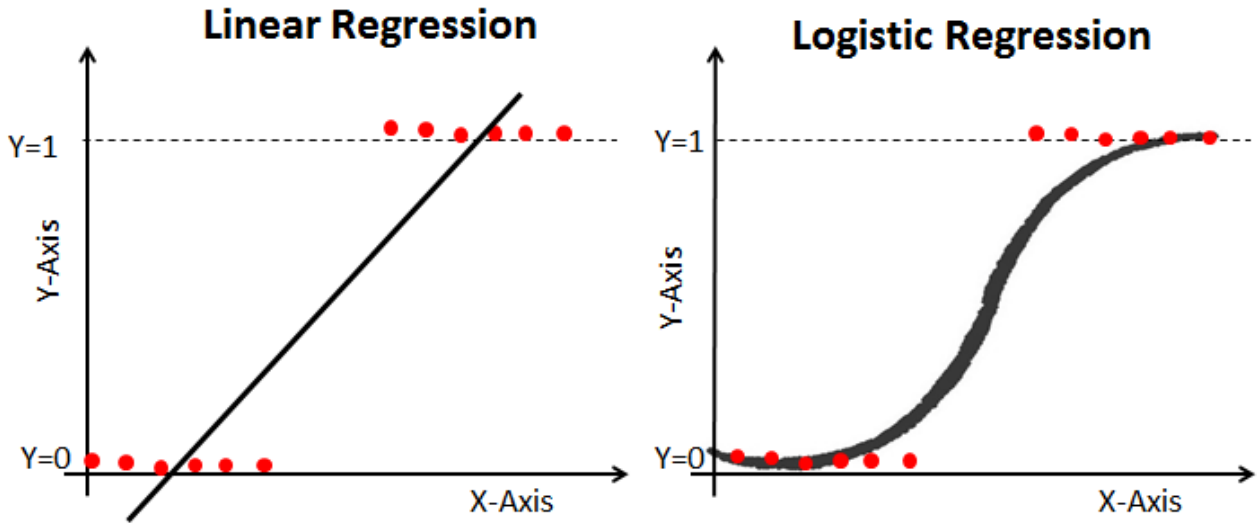
这是一个二分类问题，可以使用逻辑回归模型进行学习和预测。线性模型假设输出值与输入值之间满足：

$$Y = X \times W^T + b$$

其中，Y为输出值，X为输入特征量，W为输入特征量的权重系数矩阵，b为偏置项。

当系数矩阵W和偏置项b确定后，输入一个特征向量，即可计算出一个值。但这样获得值是连续的，可能很大，也可能会很小，而分类范围，需要在[0,1]，逻辑回归就是一种减小预测范围，将预测值限定为[0,1]间的一种回归模型，其回归方程与回归曲线如下图所示。逻辑曲线在z=0时，十分敏感，在z>>0或z<<0处，都不敏感，将预测值限定为(0,1)。常用的逻辑回归函数为Sigmoid函数，其表达式为：

$$g(z) = \frac{1}{1 + e^{-z}}$$



接着，再通过定义一个损失函数来描述预测值与真实值之间的误差，当预测值与真实值之间的误差越大时，损失函数也越大，通过梯度下降法可以得到系数矩阵和偏置项的优化趋势，再设立合适的学习率迭代求解，可得到对应较小损失函数值下的系数矩阵和偏置项的值，模型训练结束。

4. 模型训练

4.1 处理缺失值

目前的数据中，剔除船舱号后，年龄列和和登船港口列存在缺失值，需要进行补充。对于年龄，可以采用使用平均年龄进行填充，而对于登船港口，可以使用人数最多的登船港口(S)进行填充。此外，由于年龄是一个一个的数字，在数据量不够大的情况，这样一个一个的数字没太大意义，我们要按照年龄段进行划分，票价也是如此。

In [16]:
```
def handle_age(age):
    if age ≤ 15:
        return 0
    elif age ≤ 60:
        return 1
    else:
        return 2

def handle_fare(fare):
    if fare ≤ 32:
        return 0
    elif fare ≤ 100:
        return 1
    elif fare ≤ 200:
        return 2
    else:
        return 3

train_df['Age'] = train_df['Age'].fillna(train_df['Age'].mean()).map(handle_age)
train_df['Embarked'] = train_df['Embarked'].fillna('S')
train_df['Fare'] = train_df['Fare'].map(handle_fare)
```

In [17]:
```
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 8 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Survived    891 non-null    int64
 1   Pclass      891 non-null    int64
 2   Title       891 non-null    object
 3   Sex         891 non-null    object
 4   Age         891 non-null    int64
 5   Fare        891 non-null    int64
 6   Embarked    891 non-null    object
 7   FamilySize  891 non-null    int64
dtypes: int64(5), object(3)
memory usage: 55.8+ KB
```

In [18]:
```
train_df.head()
```

Out[18]:

	Survived	Pclass	Title	Sex	Age	Fare	Embarked	FamilySize
0	0	3	Mr	male	1	0	S	2
1	1	1	Mrs	female	1	1	C	2
2	1	3	Miss	female	1	0	S	1
3	1	1	Mrs	female	1	1	S	2
4	0	3	Mr	male	1	0	S	1

4.2 数字化数据

当前的数据集还还不能直接进行训练，因为数据集中部分列是由字符串表示的，无法参与到数值运算过程中，因此，我们需要将字符串处理成数字的形式。

In [19]:
```
train_df.head()
```

Out[19]:

	Survived	Pclass	Title	Sex	Age	Fare	Embarked	FamilySize
0	0	3	Mr	male	1	0	S	2
1	1	1	Mrs	female	1	1	C	2
2	1	3	Miss	female	1	0	S	1
3	1	1	Mrs	female	1	1	S	2
4	0	3	Mr	male	1	0	S	1

在头衔对获救率影响的分析中，我们将头衔分类成了'Officer', 'Royalty', 'Mrs', 'Miss', 'Mr', 'Master'，使用数字1-6分别表示上述类别。

In [20]:
```
title_map2num = {'Officer': 1, 'Royalty': 2, 'Mrs': 3, 'Miss': 4, 'Mr':5, 'Master': 6
train_df['Title'] = train_df['Title'].map(title_map2num)
```

In [21]:
```
train_df.head()
```

Out[21]:

	Survived	Pclass	Title	Sex	Age	Fare	Embarked	FamilySize
0	0	3	5	male	1	0	S	2
1	1	1	3	female	1	1	C	2
2	1	3	4	female	1	0	S	1
3	1	1	3	female	1	1	S	2
4	0	3	5	male	1	0	S	1

对于性别这列，将female映射为0, male映射为1。

In [22]:
```
sex_map2num = {'female': 0, 'male': 1}
train_df['Sex'] = train_df['Sex'].map(sex_map2num)
```

In [23]:
```
train_df.head()
```

Out[23]:

	Survived	Pclass	Title	Sex	Age	Fare	Embarked	FamilySize
0	0	3	5	1	1	0	S	2
1	1	1	3	0	1	1	C	2
2	1	3	4	0	1	0	S	1
3	1	1	3	0	1	1	S	2
4	0	3	5	1	1	0	S	1

将三个登船港口分别映射为0, 1, 2。

In [24]:

```
embarked_map2num = {'S': 0, 'C': 1, 'Q': 2}
train_df['Embarked'] = train_df['Embarked'].map(embarked_map2num)
```

In [25]:

```
train_df.head()
```

Out[25]:

	Survived	Pclass	Title	Sex	Age	Fare	Embarked	FamilySize
0	0	3	5	1	1	0	0	2
1	1	1	3	0	1	1	1	2
2	1	3	4	0	1	0	0	1
3	1	1	3	0	1	1	0	2
4	0	3	5	1	1	0	0	1

4.3 使用Pytorch搭建逻辑回归模型

In [26]:

```
import logging
import pickle
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import TensorDataset
from torch.utils.data import DataLoader
from torch.utils.data import random_split

class Titanic_Model(nn.Module):
  def __init__(self, input_dim, num_classes):
    super().__init__()
    self.input_dim   = input_dim
    self.num_classes = num_classes

    self.linear_layer = nn.Linear(input_dim, num_classes)

  def forward(self, inputs):
    outputs = self.linear_layer(inputs)
    return outputs

  @staticmethod
  def compute_accuracy(outputs, labels):
    _, preds = torch.max(outputs, dim=1)
    return torch.tensor(torch.sum(preds == labels).item() / len(preds))

  @staticmethod
  def log_epoch_loss_and_acc(prefix, epoch, epoch_loss, epoch_acc, interval=5):
    if epoch % interval == 0:
      logging.info(f'{prefix}_Epoch [{epoch}], loss: {epoch_loss:.4f},'
```

```
                f' acc: {epoch_acc:.4f}.')

  def evaluate(self, batch, loss_func, need_acc=False, no_grad=False):
    if no_grad:
      with torch.no_grad():
        inputs, labels = batch
        outputs = self(inputs)
        loss = loss_func(outputs, labels)
    else:
      inputs, labels = batch
      outputs = self(inputs)
      loss = loss_func(outputs, labels)

    if need_acc:
      acc  = self.compute_accuracy(outputs, labels)
      return {'loss': loss, 'acc': acc}
    else:
      return {'loss': loss}

  def compute_epoch_loss_and_acc(self, dataloader, loss_func):
    results = [self.evaluate(batch, loss_func, need_acc=True, no_grad=True)
                                    for batch in dataloader]
    batch_losses = [r['loss'] for r in results]
    epoch_loss   = torch.stack(batch_losses).mean()
    batch_accs   = [r['acc'] for r in results]
    epoch_acc    = torch.stack(batch_accs).mean()
    return {'epoch_loss': epoch_loss, 'epoch_acc': epoch_acc}

  def epoch_postprocess(self, prefix, data_loader, epoch,
                        history, loss_func, log_interval):
    loss_and_acc = self.compute_epoch_loss_and_acc(data_loader, loss_func)
    epoch_loss   = loss_and_acc['epoch_loss']
    epoch_acc    = loss_and_acc['epoch_acc']
    history.append({'epoch_loss': epoch_loss,
                    'epoch_acc': epoch_acc})
    self.log_epoch_loss_and_acc(prefix, epoch,
                                    epoch_loss,
                                    epoch_acc,
                                    log_interval)

  def train(self, train_loader, val_loader, num_epochs, lr,
            loss_func=F.cross_entropy, opt_func=torch.optim.SGD,
            log_interval=5):
    optimizer = opt_func(self.parameters(), lr)
    self.history_train = []  # history of train set
    self.history_val   = []  # history of validation set

    # initial loss and accuracy of training dataset
    self.epoch_postprocess('Train', train_loader, 0,
                           self.history_train, loss_func, log_interval)

    # initial loss and accuracy of validation dataset
    self.epoch_postprocess('Val', val_loader, 0,
                           self.history_val, loss_func, log_interval)

    # iteration
    for epoch in range(num_epochs):
      for batch in train_loader:
        loss = self.evaluate(batch, loss_func, need_acc=False)['loss']
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

      # training dataset loss and accuracy
      self.epoch_postprocess('Train', train_loader, epoch+1,
                             self.history_train, loss_func, log_interval)
```

```
      # validation dataset loss and accuracy
      self.epoch_postprocess('Val', val_loader, epoch+1,
                             self.history_val, loss_func, log_interval)

  def predict(self, inputs):
    outputs = self(inputs)
    _, preds = torch.max(outputs, dim=1)
    return [preds[i].item() for i in range(len(preds))]

  def save_model(self, save_file):
    torch.save(self.state_dict(), save_file)
    pickle.dump(self.history_train, open('titanic_history_train.pkl', 'wb'))
    pickle.dump(self.history_val, open('titanic_history_val.pkl', 'wb'))


  def recover_model(self, save_file):
    self.load_state_dict(torch.load(save_file))
    self.history_train = pickle.load(open('titanic_history_train.pkl', 'rb'))
    self.history_val   = pickle.load(open('titanic_history_val.pkl', 'rb'))
```

在训练前，还需要将测试集划分为训练集和验证集，当前采用5:1的形式进行划分。

In [27]:

```
# convert pandas dataframe to numpy array
train_data = train_df.to_numpy()
# convert numpy array to tensor
inputs = torch.from_numpy(train_data[:, 1:]).type(torch.float)
labels = torch.from_numpy(train_data[:, 0]).type(torch.long)
dataset = TensorDataset(inputs, labels)
train_ds, val_ds = random_split(dataset, [742, 149])
```

使用gpu加速计算，Pytorch中使用gpu计算十分简单，只需要将训练数据和模型参数转移到显存中即可(前提是配置好cuda驱动)。

In [28]:

```
def to_device(data, device):
  """Move tensor(s) to chosen device"""
  if isinstance(data, (list,tuple)):
    return [to_device(x, device) for x in data]
  return data.to(device, non_blocking=True)

class DeviceDataLoader():
  """Wrap a dataloader to move data to a device (default: cpu)"""
  def __init__(self, dl, device):
    self.dl = dl
    self.device = device

  def __iter__(self):
    """Yield a batch of data after moving it to device"""
    for b in self.dl:
      yield to_device(b, self.device)

  def __len__(self):
    """Number of batches"""
    return len(self.dl)
```

In [29]:

```
logging.basicConfig(format='%(asctime)s %(levelname)s:%(message)s', \
                    level=logging.INFO, datefmt='%m/%d/%Y %I:%M:%S %p')
x_dim         = 7   # input dimension
y_dim         = 2   # label dimension
train_sz      = 742
val_sz        = 149
batch_size    = 16
```

```
num_epochs    = 100
learning_rate = 0.005
device = torch.device('cuda')

train_loader = DataLoader(train_ds, batch_size, shuffle=True)
val_loader   = DataLoader(val_ds, batch_size)
# move dataloader to gpu
train_loader = DeviceDataLoader(train_loader, device)
val_loader   = DeviceDataLoader(val_loader, device)

# initialize linear regression model
logging.info("Initializing linear regression model.")
titanic_model = Titanic_Model(x_dim, y_dim)
# move model parameters to gpu
to_device(titanic_model, device)
logging.info("Start training ... ")
titanic_model.train(train_loader, val_loader, num_epochs,
                    learning_rate, log_interval=10, opt_func=torch.optim.SGD
)
logging.info("Training finished.")

logging.info("Save model.")
titanic_model.save_model('report02-titanic_model.pth')
```

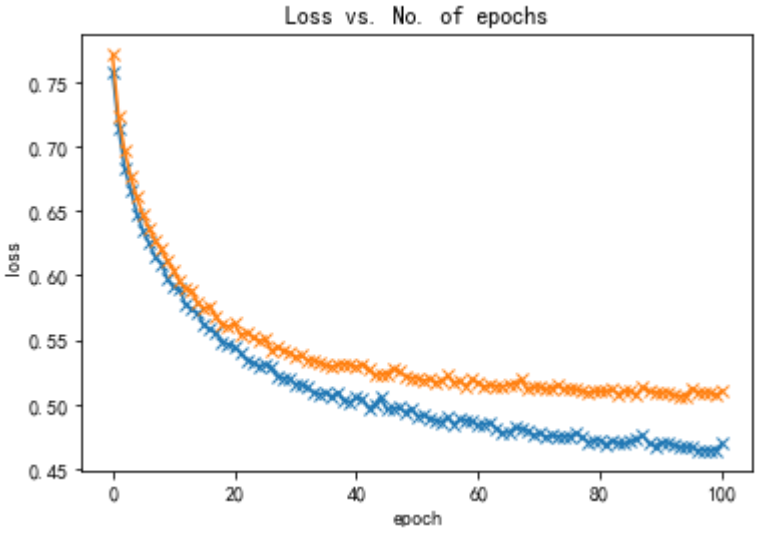
```
03/06/2022 10:43:48 PM INFO:Initializing linear regression model.
03/06/2022 10:43:50 PM INFO:Start training ...
03/06/2022 10:43:51 PM INFO:Train_Epoch [0], loss: 0.7585, acc: 0.6144.
03/06/2022 10:43:51 PM INFO:Val_Epoch [0], loss: 0.7717, acc: 0.5962.
03/06/2022 10:43:51 PM INFO:Train_Epoch [10], loss: 0.5908, acc: 0.6755.
03/06/2022 10:43:51 PM INFO:Val_Epoch [10], loss: 0.6045, acc: 0.6525.
03/06/2022 10:43:52 PM INFO:Train_Epoch [20], loss: 0.5441, acc: 0.7176.
03/06/2022 10:43:52 PM INFO:Val_Epoch [20], loss: 0.5625, acc: 0.7412.
03/06/2022 10:43:52 PM INFO:Train_Epoch [30], loss: 0.5153, acc: 0.7699.
03/06/2022 10:43:52 PM INFO:Val_Epoch [30], loss: 0.5365, acc: 0.7412.
03/06/2022 10:43:53 PM INFO:Train_Epoch [40], loss: 0.5048, acc: 0.7699.
03/06/2022 10:43:53 PM INFO:Val_Epoch [40], loss: 0.5287, acc: 0.7475.
03/06/2022 10:43:53 PM INFO:Train_Epoch [50], loss: 0.4907, acc: 0.7996.
03/06/2022 10:43:53 PM INFO:Val_Epoch [50], loss: 0.5201, acc: 0.7600.
03/06/2022 10:43:54 PM INFO:Train_Epoch [60], loss: 0.4833, acc: 0.8001.
03/06/2022 10:43:54 PM INFO:Val_Epoch [60], loss: 0.5168, acc: 0.7862.
03/06/2022 10:43:54 PM INFO:Train_Epoch [70], loss: 0.4777, acc: 0.8036.
03/06/2022 10:43:54 PM INFO:Val_Epoch [70], loss: 0.5135, acc: 0.7738.
03/06/2022 10:43:55 PM INFO:Train_Epoch [80], loss: 0.4720, acc: 0.7983.
03/06/2022 10:43:55 PM INFO:Val_Epoch [80], loss: 0.5098, acc: 0.7800.
03/06/2022 10:43:55 PM INFO:Train_Epoch [90], loss: 0.4705, acc: 0.7948.
03/06/2022 10:43:55 PM INFO:Val_Epoch [90], loss: 0.5083, acc: 0.7862.
03/06/2022 10:43:56 PM INFO:Train_Epoch [100], loss: 0.4694, acc: 0.8001.
03/06/2022 10:43:56 PM INFO:Val_Epoch [100], loss: 0.5098, acc: 0.7800.
03/06/2022 10:43:56 PM INFO:Training finished.
03/06/2022 10:43:56 PM INFO:Save model.
```

从运行结果中可以看到，训练得到的模型在测试集上的识别准确率为80.01%，在验证集上的识别准确率为78.00%。画出迭代过程中的损失函数值与准确率的变化趋势图：

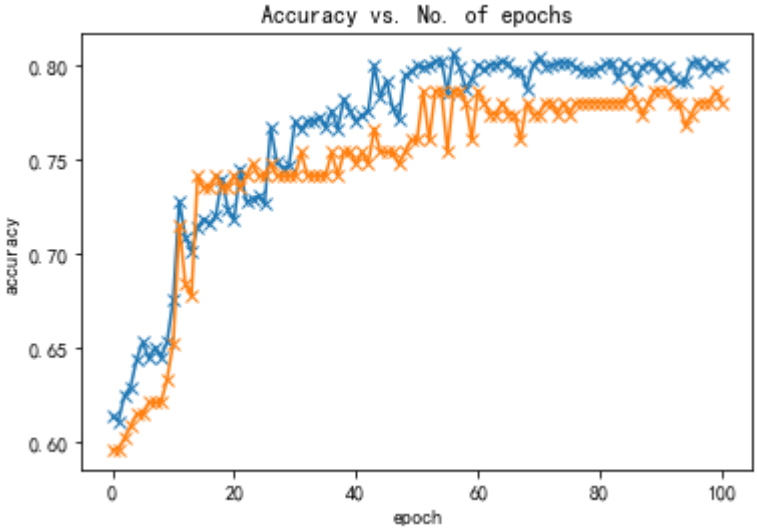
In [30]:
```
history_train = pickle.load(open('titanic_history_train.pkl', 'rb'))
history_val = pickle.load(open('titanic_history_val.pkl', 'rb'))
```

In [31]:
```
train_losses = [float(x['epoch_loss']) for x in history_train]
val_losses = [float(x['epoch_loss']) for x in history_val]
plt.plot(train_losses, '-x', val_losses, '-x')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.title('Loss vs. No. of epochs')
```

Out[31]:
```
Text(0.5, 1.0, 'Loss vs. No. of epochs')
```



In [32]:
```
train_accs = [float(x['epoch_acc']) for x in history_train]
val_accs = [float(x['epoch_acc']) for x in history_val]
plt.plot(train_accs, '-x', val_accs, '-x')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.title('Accuracy vs. No. of epochs')
```



可以看到，模型在学习速率为0.005的情况下，训练到80步的时候，准确率在79%左右不再上升。

5. 小结

在这份报告中，我们使用Pytorch搭建逻辑回归模型进行了泰坦尼克号生存率预测。首先，我们分析并提炼了数据集的特征数据，在这个过程中，熟悉了pandas中的dataframe数据结构的基本操作，发现其在批量处理数据时十分方便，并将一些连续值的变量变成分段变量以提高数据的凝练度。此外，该任务中，由于样本中存在大量缺失值，所以还对如何处理缺失值进行了学习。

Report03 - 服装分类

- 沈键
- 2021200082

1. 任务简介

FashionMNIST 是一个替代 MNIST 手写数字集的图像数据集。 它是由 Zalando（一家德国的时尚科技公司）旗下的研究部门提供。其涵盖了来自 10 种类别的共 7 万个不同商品的正面图片。FashionMNIST 的大小、格式和训练集/测试集划分与原始的 MNIST 完全一致。60000/10000 的训练测试数据划分，28x28 的灰度图片。可以直接用它来测试机器学习和深度学习算法性能，且不需要改动任何的代码。

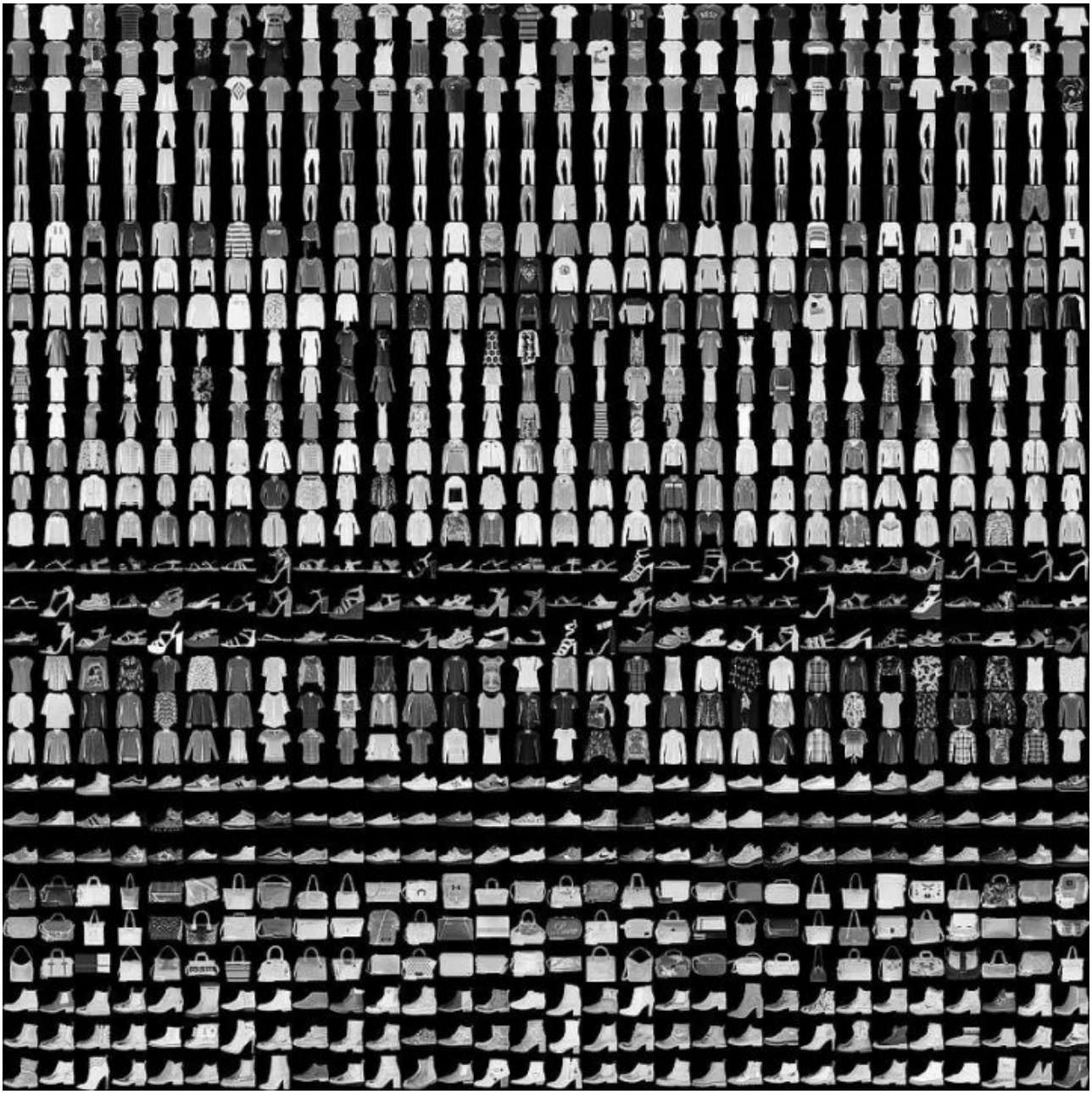
2. 数据分析

先来下载FashionMNIST数据集：

In [1]:
```
import torch
import torchvision
import torchvision.transforms as transforms
```

In [2]:
```
train_ds = torchvision.datasets.FashionMNIST(root='data/', train=True, download=True,
                                              transform=transforms.ToTensor())
val_ds   = torchvision.datasets.FashionMNIST(root='data/', train=False, download=True
                                              transform=transforms.ToTensor())
```

FashionMNIST数据集大致如下（每个类别占三行）：



可以看到，FashionMNIST总共有10个类别，不同标签与类别的映射关系如下：

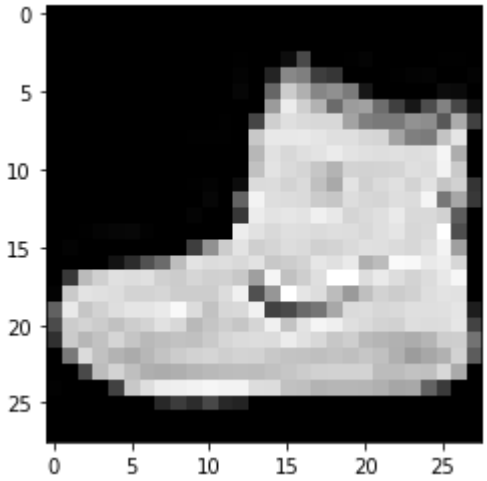
Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

取出一张图来：

In [3]:
```
import matplotlib.pyplot as plt
```

```
image, label = train_ds[0]
print('image.shape:', image.shape)
plt.imshow(image.permute(1, 2, 0), cmap='gray')
print('Label:', label)
```

```
image.shape: torch.Size([1, 28, 28])
Label: 9
```



In [4]:

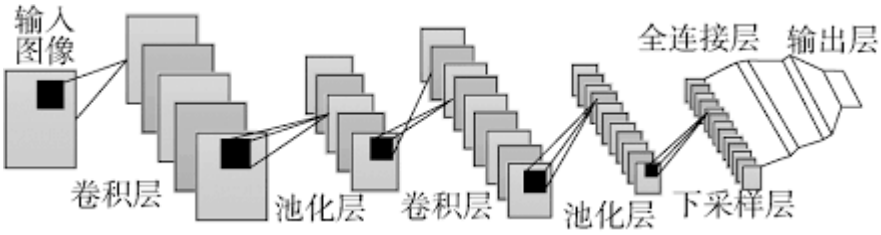
```
print(image[0, 10:15 , 10:15])
print(torch.max(image), torch.min(image))
```

```
tensor([[0.0000, 0.0000, 0.0000, 0.7569, 0.8941],
        [0.0118, 0.0000, 0.0471, 0.8588, 0.8627],
        [0.0235, 0.0000, 0.3882, 0.9569, 0.8706],
        [0.0000, 0.0000, 0.2157, 0.9255, 0.8941],
        [0.0000, 0.0000, 0.9294, 0.8863, 0.8510]])
tensor(1.) tensor(0.)
```

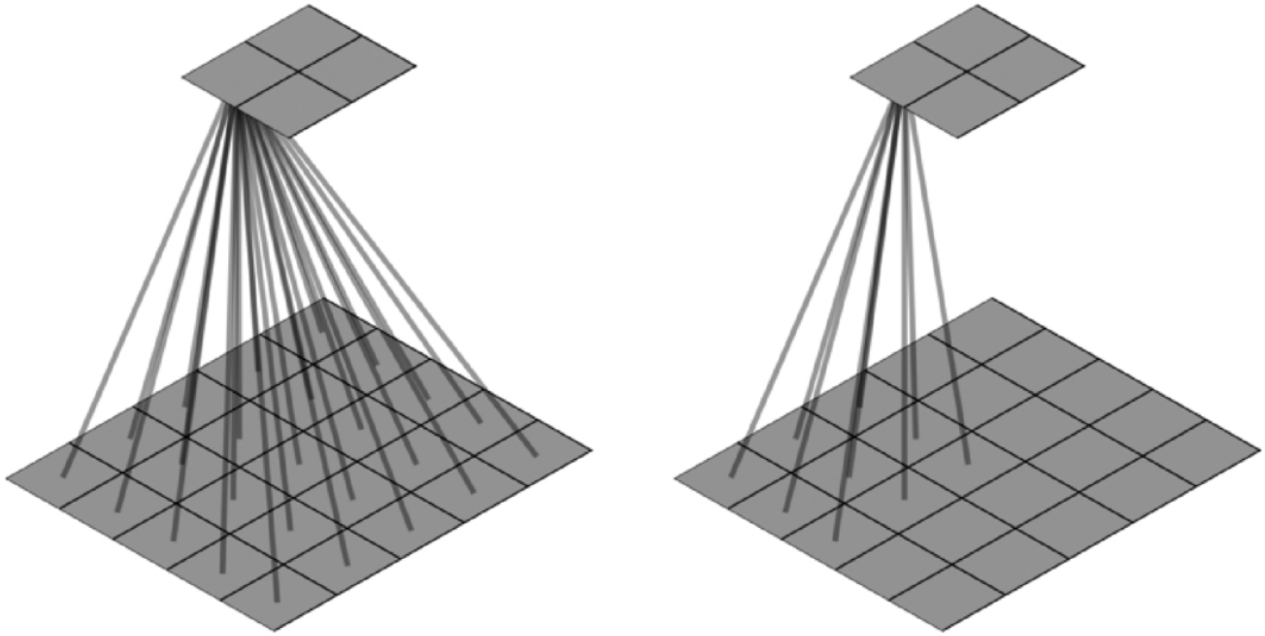
数据集中的每一张图为28x28大小的灰度图片，并且，灰度值已归一化，取值范围落在[0, 1]。这是一个多分类的图片识别问题，考虑使用卷积神经网络进行训练和识别。

3. 卷积神经网络

卷积神经网络与全连接神经网络很相似，依旧是层级网络，只是层的功能和形式做了变化，并多了许多传统神经网络没有的层次，例如卷积层、池化层。其结构示意图如下：



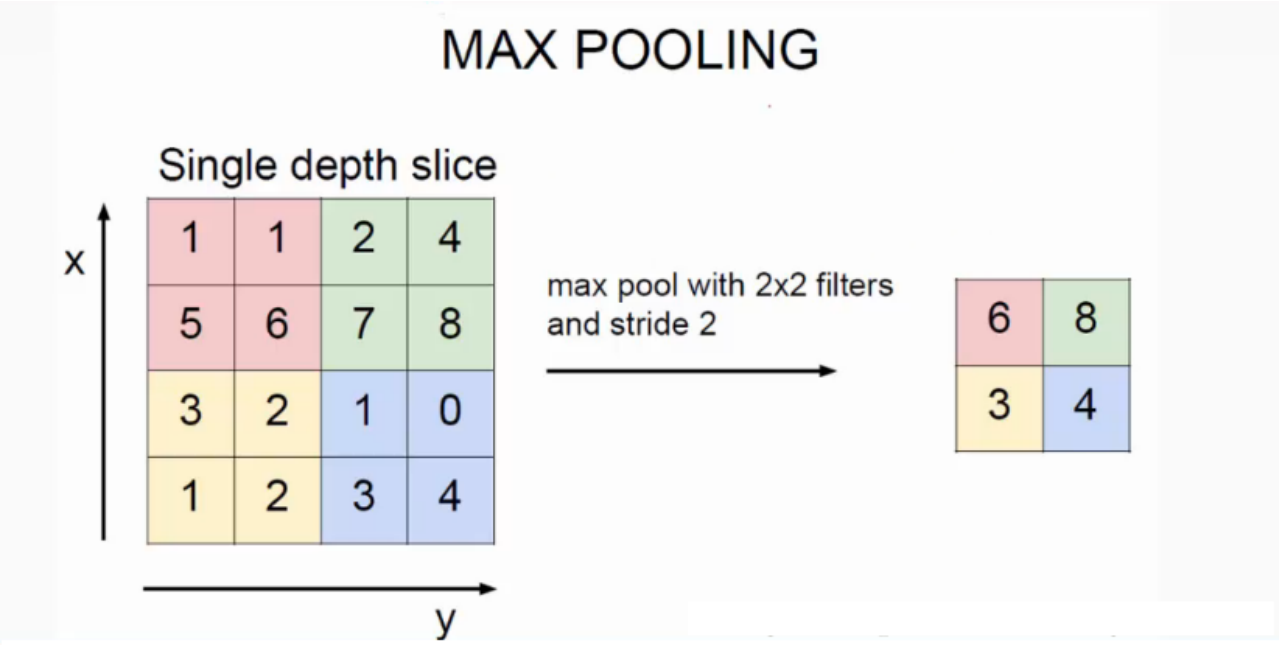
卷积网络也是收到了生物学的启发，一般人们认为图片中距离相近的部分相关性较大，而距离比较远的部分相关性较小，所以卷积网络使用了局部感受野，卷积层中的神经元连接不是全连接的，而是后一层的每个神经元连接前一层的一部分神经元，从而大大减少了参数量。



（a）BP神经网络的全连接结构　　（b）卷积神经网络的局部连接结构

卷积核的另一个特点和优势时权值共享。卷积核的权重是经过学习得到的，并且在卷积过程中卷积核的权重是不会改变的，这就是参数共享的思想。权值共享降低了网络的复杂性，特别是多维输入向量的图像可以直接输入网络这一特点避免了特征提取和分类过程中数据重建的复杂度。

一个经典的卷积层包含3个部分——卷积计算、非线性激活函数和池化，池化也有一个滑动窗口，用于在图像中进行滑动计算，这一点跟卷积核有点类似，但池化层中没有需要训练的参数。卷积网络中的权值参数还是很多，池化层可对提取到的特征信息进行降维，一方面使特征图变小，简化网络计算复杂度并在一定程度上避免过拟合的出现；一方面进行特征压缩，提取主要特征。池化操作使得网络的输入具有平移不变特征。常用的池化函数有max-pooling和mean-pooling，max-pooling为取池化窗口中的最大值，mean-pooling取的是池化窗口中的平均值。



再经过几层卷积层后，特征图的尺寸越来越小，最后通过将特征图展平，后面部分进行全连接，对提取出来的图像特征进行分类。

在卷积网络的设计中，除了需要设计合适的深度外，还需要给定卷积核和池化窗口的尺寸大小和移动步长，并且，为了避免丢失图像的边缘特征，往往还需要对特征图进行Padding操作，即在图像的外缘填充数据，这样，可以使得卷积后得到的特征图的大小跟原始图像的大小相同。

4. 模型训练

在此次任务中，我们选取的卷积神经网络由2层卷积层和2层全连接层构成，池化函数选用的是最大池化，激活函数选用的为ReLUh函数。其代码如下：

In [5]:

```
import logging
import pickle
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader

class Fashion_Model(nn.Module):
  def __init__(self, input_dim, num_classes):
    super().__init__()
    self.input_dim   = input_dim
    self.num_classes = num_classes

    self.network = nn.Sequential(
      nn.Conv2d(input_dim[0], 32, kernel_size=5, stride=1, padding='same'),
      nn.ReLU(),
      nn.MaxPool2d(2, 2),

      nn.Conv2d(32, 64, kernel_size=5, stride=1, padding='same'),
      nn.ReLU(),
      nn.MaxPool2d(2, 2),

      nn.Flatten(),
      nn.Linear(64*(input_dim[1]//4*input_dim[2]//4), 1024),
      nn.ReLU(),

      nn.Linear(1024, 10))

  def forward(self, inputs):
    outputs = self.network(inputs)
    return outputs

  @staticmethod
  def compute_accuracy(outputs, labels):
    _, preds = torch.max(outputs, dim=1)
    return torch.tensor(torch.sum(preds == labels).item() / len(preds))

  @staticmethod
  def log_epoch_loss_and_acc(prefix, epoch, epoch_loss, epoch_acc, interval=5):
    if epoch % interval == 0:
      logging.info(f'{prefix}_Epoch [{epoch}], loss: {epoch_loss:.4f},'
                   f' acc: {epoch_acc:.4f}.')

  def evaluate(self, batch, loss_func, need_acc=False, no_grad=False):
    if no_grad:
      with torch.no_grad():
        inputs, labels = batch
        outputs = self(inputs)
        loss = loss_func(outputs, labels)
    else:
      inputs, labels = batch
      outputs = self(inputs)
      loss = loss_func(outputs, labels)

    if need_acc:
      acc  = self.compute_accuracy(outputs, labels)
      return {'loss': loss, 'acc': acc}
    else:
      return {'loss': loss}
```

```
  def compute_epoch_loss_and_acc(self, dataloader, loss_func):
    results = [self.evaluate(batch, loss_func, need_acc=True, no_grad=True)
                                    for batch in dataloader]
    batch_losses = [r['loss'] for r in results]
    epoch_loss   = torch.stack(batch_losses).mean()
    batch_accs   = [r['acc'] for r in results]
    epoch_acc    = torch.stack(batch_accs).mean()
    return {'epoch_loss': epoch_loss, 'epoch_acc': epoch_acc}

  def epoch_postprocess(self, prefix, data_loader, epoch,
                        history, loss_func, log_interval):
    loss_and_acc = self.compute_epoch_loss_and_acc(data_loader, loss_func)
    epoch_loss   = loss_and_acc['epoch_loss']
    epoch_acc    = loss_and_acc['epoch_acc']
    history.append({'epoch_loss': epoch_loss,
                    'epoch_acc': epoch_acc})
    self.log_epoch_loss_and_acc(prefix, epoch,
                                    epoch_loss,
                                    epoch_acc,
                                    log_interval)

  def train(self, train_loader, val_loader, num_epochs, lr,
            loss_func=F.cross_entropy, opt_func=torch.optim.Adam,
            log_interval=5):
    optimizer = opt_func(self.parameters(), lr)
    self.history_train = []  # history of train set
    self.history_val   = []  # history of validation set

    # initial loss and accuracy of training dataset
    self.epoch_postprocess('Train', train_loader, 0,
                           self.history_train, loss_func, log_interval)

    # initial loss and accuracy of validation dataset
    self.epoch_postprocess('Val', val_loader, 0,
                           self.history_val, loss_func, log_interval)

    # iteration
    for epoch in range(num_epochs):
      for batch in train_loader:
        loss = self.evaluate(batch, loss_func, need_acc=False)['loss']
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

      # training dataset loss and accuracy
      self.epoch_postprocess('Train', train_loader, epoch+1,
                             self.history_train, loss_func, log_interval)

      # validation dataset loss and accuracy
      self.epoch_postprocess('Val', val_loader, epoch+1,
                             self.history_val, loss_func, log_interval)

  def predict(self, inputs):
    outputs = self(inputs)
    _, preds = torch.max(outputs, dim=1)
    return [preds[i].item() for i in range(len(preds))]

  def save_model(self, save_file):
    torch.save(self.state_dict(), save_file)
    pickle.dump(self.history_train, open('fashion_history_train.pkl', 'wb'))
    pickle.dump(self.history_val, open('fashion_history_val.pkl', 'wb'))


  def recover_model(self, save_file):
    self.load_state_dict(torch.load(save_file))
```

```
    self.history_train = pickle.load(open('fashion_history_train.pkl', 'rb'))
    self.history_val   = pickle.load(open('fashion_history_val.pkl', 'rb'))
```

使用gpu加速计算，Pytorch中使用gpu计算十分简单，只需要将训练数据和模型参数转移到显存中即可(前提是配置好cuda驱动)。

In [6]:
```
def to_device(data, device):
  """Move tensor(s) to chosen device"""
  if isinstance(data, (list,tuple)):
    return [to_device(x, device) for x in data]
  return data.to(device, non_blocking=True)

class DeviceDataLoader():
  """Wrap a dataloader to move data to a device (default: cpu)"""
  def __init__(self, dl, device):
    self.dl = dl
    self.device = device

  def __iter__(self):
    """Yield a batch of data after moving it to device"""
    for b in self.dl:
      yield to_device(b, self.device)

  def __len__(self):
    """Number of batches"""
    return len(self.dl)
```

In [7]:
```
logging.basicConfig(format='%(asctime)s %(levelname)s:%(message)s', \
                    level=logging.INFO, datefmt='%m/%d/%Y %I:%M:%S %p')
```

学习速率选为0.001，每次训练的批数据集大小为128。

In [8]:
```
inp_dim       = (1, 28, 28)
out_dim       = 10
batch_size    = 128
num_epochs    = 50
learning_rate = 1e-4
device = torch.device('cuda')
```

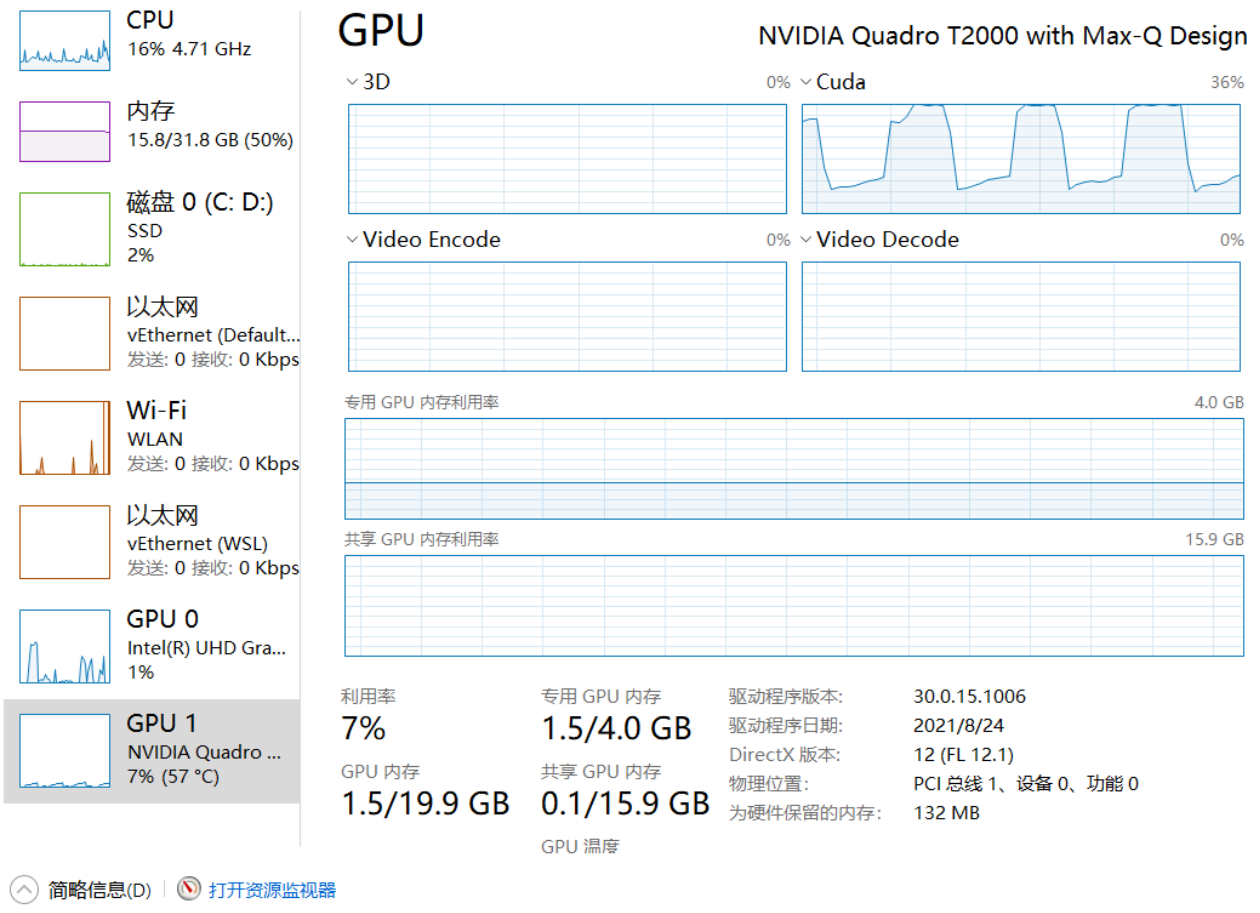
In [9]:
```
train_loader = DataLoader(train_ds, batch_size, shuffle=True)
val_loader   = DataLoader(val_ds, batch_size)
```

In [10]:
```
# move dataloader to gpu
train_loader = DeviceDataLoader(train_loader, device)
val_loader   = DeviceDataLoader(val_loader, device)
```

In [11]:
```
# initialize model
logging.info("Initializing ...")
fashion_model = Fashion_Model(inp_dim, out_dim)
# move model parameters to gpu
to_device(fashion_model, device)
logging.info("Start training ...")
fashion_model.train(train_loader, val_loader, num_epochs,
              learning_rate, log_interval=5)
logging.info("Training finished.")

fashion_model.save_model('MNIST.pth')
logging.info("Save model.")
```

```
03/07/2022 01:30:07 PM INFO:Initializing  ...
03/07/2022 01:30:10 PM INFO:Start training ...
03/07/2022 01:30:19 PM INFO:Train_Epoch [0], loss: 2.2974, acc: 0.0947.
03/07/2022 01:30:21 PM INFO:Val_Epoch [0], loss: 2.2974, acc: 0.0949.
03/07/2022 01:31:37 PM INFO:Train_Epoch [5], loss: 0.3220, acc: 0.8856.
03/07/2022 01:31:39 PM INFO:Val_Epoch [5], loss: 0.3506, acc: 0.8736.
03/07/2022 01:33:02 PM INFO:Train_Epoch [10], loss: 0.2631, acc: 0.9040.
03/07/2022 01:33:03 PM INFO:Val_Epoch [10], loss: 0.3018, acc: 0.8888.
03/07/2022 01:34:27 PM INFO:Train_Epoch [15], loss: 0.1988, acc: 0.9285.
03/07/2022 01:34:28 PM INFO:Val_Epoch [15], loss: 0.2526, acc: 0.9078.
03/07/2022 01:35:50 PM INFO:Train_Epoch [20], loss: 0.1711, acc: 0.9382.
03/07/2022 01:35:51 PM INFO:Val_Epoch [20], loss: 0.2413, acc: 0.9110.
03/07/2022 01:37:11 PM INFO:Train_Epoch [25], loss: 0.1511, acc: 0.9451.
03/07/2022 01:37:13 PM INFO:Val_Epoch [25], loss: 0.2400, acc: 0.9116.
03/07/2022 01:38:37 PM INFO:Train_Epoch [30], loss: 0.1289, acc: 0.9529.
03/07/2022 01:38:38 PM INFO:Val_Epoch [30], loss: 0.2459, acc: 0.9132.
03/07/2022 01:40:01 PM INFO:Train_Epoch [35], loss: 0.0975, acc: 0.9664.
03/07/2022 01:40:02 PM INFO:Val_Epoch [35], loss: 0.2351, acc: 0.9219.
03/07/2022 01:41:27 PM INFO:Train_Epoch [40], loss: 0.0825, acc: 0.9721.
03/07/2022 01:41:28 PM INFO:Val_Epoch [40], loss: 0.2509, acc: 0.9164.
03/07/2022 01:42:51 PM INFO:Train_Epoch [45], loss: 0.0619, acc: 0.9798.
03/07/2022 01:42:52 PM INFO:Val_Epoch [45], loss: 0.2606, acc: 0.9181.
03/07/2022 01:44:15 PM INFO:Train_Epoch [50], loss: 0.0433, acc: 0.9884.
03/07/2022 01:44:16 PM INFO:Val_Epoch [50], loss: 0.2662, acc: 0.9212.
03/07/2022 01:44:16 PM INFO:Training finished.
03/07/2022 01:44:16 PM INFO:Save model.
```

通过管理器查看gpu的使用情况，可以看到我们搭建的模型确实使用到了gpu进行运算，在训练过程中，Cuda的占用率很高。



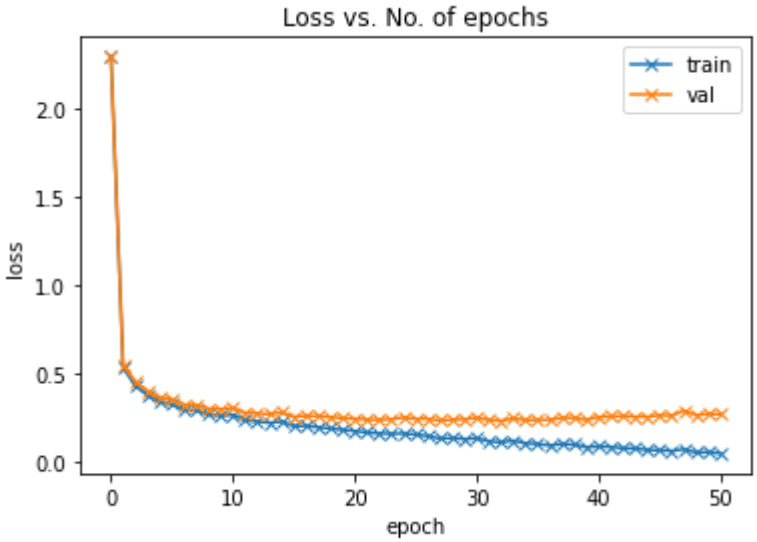
训练过程中，训练集和验证集上的损失函数值与识别准确度的变化图如下：

In [13]:

```
history_train = pickle.load(open('fashion_history_train.pkl', 'rb'))
history_val = pickle.load(open('fashion_history_val.pkl', 'rb'))
```

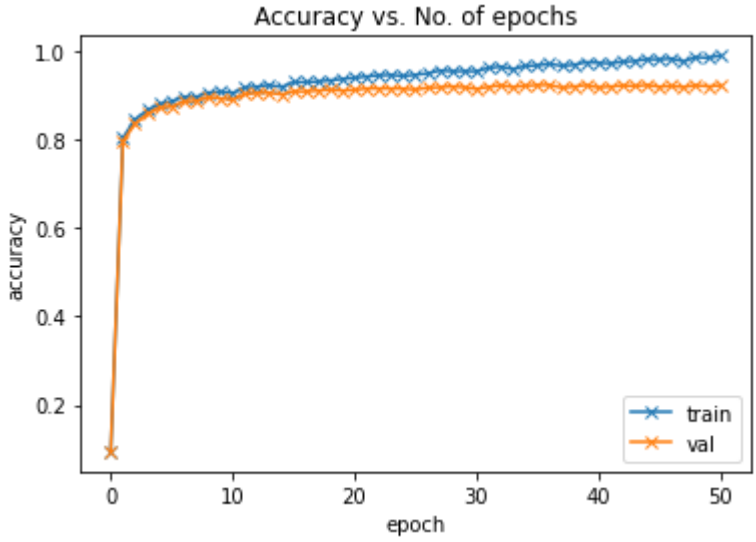
In [14]:
```
import matplotlib.pyplot as plt

train_losses = [float(x['epoch_loss']) for x in history_train]
val_losses = [float(x['epoch_loss']) for x in history_val]
plt.plot(train_losses, '-x', label='train')
plt.plot(val_losses, '-x', label='val')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.title('Loss vs. No. of epochs')
```

Out[14]:
```
Text(0.5, 1.0, 'Loss vs. No. of epochs')
```



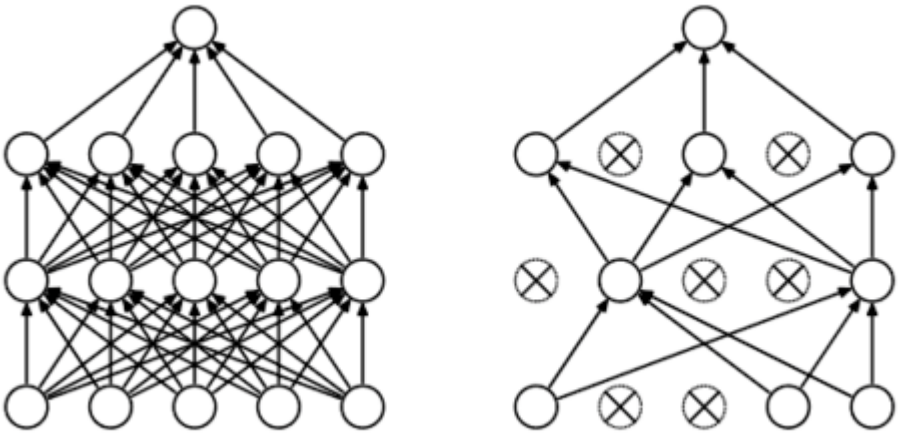
In [15]:
```
train_accs = [float(x['epoch_acc']) for x in history_train]
val_accs = [float(x['epoch_acc']) for x in history_val]
plt.plot(train_accs, '-x', label='train')
plt.plot(val_accs, '-x', label='val')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.legend()
plt.title('Accuracy vs. No. of epochs')
```

Out[15]:
```
Text(0.5, 1.0, 'Accuracy vs. No. of epochs')
```



训练出来的CNN模型在验证集上的识别准确率为92.12%。从迭代过程中，训练集和验证集上的损失函数与准确度的变化中可以看出，模型在迭代到30步左右时，模型在验证集上的准确率不再上升，并且损失函数值逐渐增大，出现了过拟合的现象。常见的抵抗过拟合的方法由增加数据量、提前停止、Dropout、正则化和标签光滑等。这里，我们使用Dropout技术来抵抗过拟合的问题。Dropout通常是在神经网络隐

藏层的部分使用，使用的时候会临时关闭掉一部分的神经元，可通过一个参数来控制神经元关闭的概率。



(a) Standard Neural Net　　(b) After applying dropout.

In [16]:

```
class Fashion_Model2(nn.Module):
  def __init__(self, input_dim, num_classes):
    super().__init__()
    self.input_dim   = input_dim
    self.num_classes = num_classes

    self.network = nn.Sequential(
      nn.Conv2d(input_dim[0], 32, kernel_size=5, stride=1, padding='same'),
      nn.ReLU(),
      nn.MaxPool2d(2, 2),

      nn.Conv2d(32, 64, kernel_size=5, stride=1, padding='same'),
      nn.ReLU(),
      nn.MaxPool2d(2, 2),

      nn.Flatten(),
      nn.Linear(64*(input_dim[1]//4*input_dim[2]//4), 1024),
      nn.ReLU(),

      # Dropout
      nn.Dropout(0.5),

      nn.Linear(1024, 10))

  def forward(self, inputs):
    outputs = self.network(inputs)
    return outputs

  @staticmethod
  def compute_accuracy(outputs, labels):
    _, preds = torch.max(outputs, dim=1)
    return torch.tensor(torch.sum(preds == labels).item() / len(preds))

  @staticmethod
  def log_epoch_loss_and_acc(prefix, epoch, epoch_loss, epoch_acc, interval=5):
    if epoch % interval == 0:
      logging.info(f'{prefix}_Epoch [{epoch}], loss: {epoch_loss:.4f},'
                   f' acc: {epoch_acc:.4f}.')

  def evaluate(self, batch, loss_func, need_acc=False, no_grad=False):
    if no_grad:
      with torch.no_grad():
        inputs, labels = batch
        outputs = self(inputs)
```

```
      loss = loss_func(outputs, labels)
    else:
      inputs, labels = batch
      outputs = self(inputs)
      loss = loss_func(outputs, labels)

    if need_acc:
      acc  = self.compute_accuracy(outputs, labels)
      return {'loss': loss, 'acc': acc}
    else:
      return {'loss': loss}

  def compute_epoch_loss_and_acc(self, dataloader, loss_func):
    results = [self.evaluate(batch, loss_func, need_acc=True, no_grad=True)
                                      for batch in dataloader]
    batch_losses = [r['loss'] for r in results]
    epoch_loss   = torch.stack(batch_losses).mean()
    batch_accs   = [r['acc'] for r in results]
    epoch_acc    = torch.stack(batch_accs).mean()
    return {'epoch_loss': epoch_loss, 'epoch_acc': epoch_acc}

  def epoch_postprocess(self, prefix, data_loader, epoch,
                        history, loss_func, log_interval):
    loss_and_acc = self.compute_epoch_loss_and_acc(data_loader, loss_func)
    epoch_loss   = loss_and_acc['epoch_loss']
    epoch_acc    = loss_and_acc['epoch_acc']
    history.append({'epoch_loss': epoch_loss,
                    'epoch_acc': epoch_acc})
    self.log_epoch_loss_and_acc(prefix, epoch,
                                    epoch_loss,
                                    epoch_acc,
                                    log_interval)

  def train(self, train_loader, val_loader, num_epochs, lr,
            loss_func=F.cross_entropy, opt_func=torch.optim.Adam,
            log_interval=5):
    optimizer = opt_func(self.parameters(), lr)
    self.history_train = []  # history of train set
    self.history_val   = []  # history of validation set

    # initial loss and accuracy of training dataset
    self.epoch_postprocess('Train', train_loader, 0,
                           self.history_train, loss_func, log_interval)

    # initial loss and accuracy of validation dataset
    self.epoch_postprocess('Val', val_loader, 0,
                           self.history_val, loss_func, log_interval)

    # iteration
    for epoch in range(num_epochs):
      for batch in train_loader:
        loss = self.evaluate(batch, loss_func, need_acc=False)['loss']
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

      # training dataset loss and accuracy
      self.epoch_postprocess('Train', train_loader, epoch+1,
                             self.history_train, loss_func, log_interval)

      # validation dataset loss and accuracy
      self.epoch_postprocess('Val', val_loader, epoch+1,
                             self.history_val, loss_func, log_interval)

  def predict(self, inputs):
    outputs = self(inputs)
```

```
    _, preds = torch.max(outputs, dim=1)
    return [preds[i].item() for i in range(len(preds))]

  def save_model(self, save_file):
    torch.save(self.state_dict(), save_file)
    pickle.dump(self.history_train, open('fashion_history_train2.pkl', 'wb'))
    pickle.dump(self.history_val, open('fashion_history_val2.pkl', 'wb'))


  def recover_model(self, save_file):
    self.load_state_dict(torch.load(save_file))
    self.history_train = pickle.load(open('fashion_history_train2.pkl', 'rb'))
    self.history_val   = pickle.load(open('fashion_history_val2.pkl', 'rb'))
```

In [17]:

```
# initialize model
logging.info("Initializing ...")
fashion_model2 = Fashion_Model2(inp_dim, out_dim)
# move model parameters to gpu
to_device(fashion_model2, device)
logging.info("Start training ...")
fashion_model2.train(train_loader, val_loader, num_epochs,
                     learning_rate, log_interval=5)
logging.info("Training finished.")

fashion_model2.save_model('MNIST2.pth')
logging.info("Save model.")
```

```
03/07/2022 01:46:02 PM INFO:Initializing ...
03/07/2022 01:46:02 PM INFO:Start training ...
03/07/2022 01:46:11 PM INFO:Train_Epoch [0], loss: 2.3066, acc: 0.0643.
03/07/2022 01:46:12 PM INFO:Val_Epoch [0], loss: 2.3063, acc: 0.0666.
03/07/2022 01:47:31 PM INFO:Train_Epoch [5], loss: 0.3234, acc: 0.8845.
03/07/2022 01:47:33 PM INFO:Val_Epoch [5], loss: 0.3505, acc: 0.8744.
03/07/2022 01:49:01 PM INFO:Train_Epoch [10], loss: 0.2574, acc: 0.9060.
03/07/2022 01:49:03 PM INFO:Val_Epoch [10], loss: 0.2970, acc: 0.8931.
03/07/2022 01:50:31 PM INFO:Train_Epoch [15], loss: 0.2252, acc: 0.9172.
03/07/2022 01:50:32 PM INFO:Val_Epoch [15], loss: 0.2757, acc: 0.9009.
03/07/2022 01:52:00 PM INFO:Train_Epoch [20], loss: 0.1864, acc: 0.9331.
03/07/2022 01:52:02 PM INFO:Val_Epoch [20], loss: 0.2512, acc: 0.9090.
03/07/2022 01:53:24 PM INFO:Train_Epoch [25], loss: 0.1640, acc: 0.9399.
03/07/2022 01:53:25 PM INFO:Val_Epoch [25], loss: 0.2453, acc: 0.9150.
03/07/2022 01:54:46 PM INFO:Train_Epoch [30], loss: 0.1398, acc: 0.9488.
03/07/2022 01:54:47 PM INFO:Val_Epoch [30], loss: 0.2376, acc: 0.9173.
03/07/2022 01:56:11 PM INFO:Train_Epoch [35], loss: 0.1270, acc: 0.9547.
03/07/2022 01:56:13 PM INFO:Val_Epoch [35], loss: 0.2400, acc: 0.9167.
03/07/2022 01:57:38 PM INFO:Train_Epoch [40], loss: 0.1085, acc: 0.9598.
03/07/2022 01:57:39 PM INFO:Val_Epoch [40], loss: 0.2392, acc: 0.9187.
03/07/2022 01:59:02 PM INFO:Train_Epoch [45], loss: 0.0890, acc: 0.9686.
03/07/2022 01:59:03 PM INFO:Val_Epoch [45], loss: 0.2389, acc: 0.9203.
03/07/2022 02:00:25 PM INFO:Train_Epoch [50], loss: 0.0743, acc: 0.9737.
03/07/2022 02:00:26 PM INFO:Val_Epoch [50], loss: 0.2519, acc: 0.9241.
03/07/2022 02:00:26 PM INFO:Training finished.
03/07/2022 02:00:26 PM INFO:Save model.
```

In [18]:

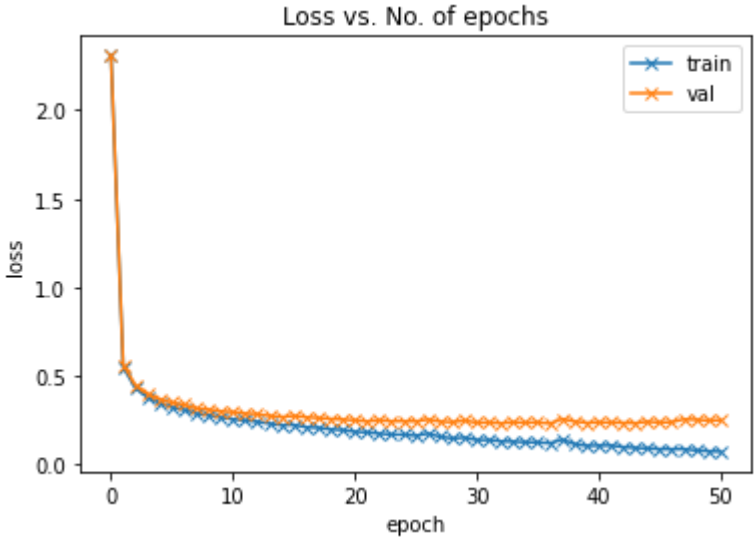
```
history_train2 = pickle.load(open('fashion_history_train2.pkl', 'rb'))
history_val2 = pickle.load(open('fashion_history_val2.pkl', 'rb'))
```

In [19]:

```
train_losses = [float(x['epoch_loss']) for x in history_train2]
val_losses = [float(x['epoch_loss']) for x in history_val2]
plt.plot(train_losses, '-x', label='train')
plt.plot(val_losses, '-x', label='val')
plt.xlabel('epoch')
plt.ylabel('loss')
```

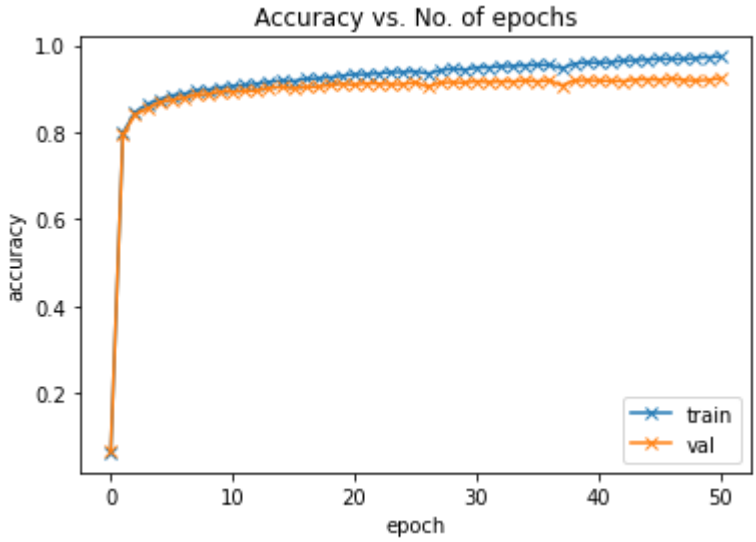
```
plt.legend()
plt.title('Loss vs. No. of epochs')
```

Out[19]:
```
Text(0.5, 1.0, 'Loss vs. No. of epochs')
```



In [20]:
```
train_accs = [float(x['epoch_acc']) for x in history_train2]
val_accs = [float(x['epoch_acc']) for x in history_val2]
plt.plot(train_accs, '-x', label='train')
plt.plot(val_accs, '-x', label='val')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.legend()
plt.title('Accuracy vs. No. of epochs')
```

Out[20]:
```
Text(0.5, 1.0, 'Accuracy vs. No. of epochs')
```



从迭代过程中输出的损失函数值与识别准确度上可以发现，采用了dropout技术后，过拟合现象略有缓解，在30步后，验证集上的识别准确率还能继续增加，最终验证集上的识别准确率达到了92.41%。

5. 测试真实场景下的表现

为了验证上述训练模型在真实场景下的表现，我们使用爬虫爬取了京东商城里的相关，并用上述训练好的模型进行识别分类。

使用了Scrapy框架的爬取京东商城图片的爬虫代码如下：

In [23]:
```
import scrapy
```

```
class JDSpider(scrapy.Spider):
  name           = 'JDspider'
  class_name     = '包'
  download_urls  = set()
  max_num        = 100
  start_page_num = 1
  BASE_URL       = f'https://search.jd.com/Search?keyword={class_name}'
  start_urls     = [f'{BASE_URL}&page={start_page_num}']

  custom_settings = {
    'DOWNLOAD_DELAY': 6
  }

  def parse(self, response):
    img_url_list = []
    for product in response.css('li.gl-item img'):
      if len(self.download_urls) < self.max_num and \
                    'data-lazy-img' in product.attrib:
        image_url = 'https:'+product.attrib['data-lazy-img']
        if image_url not in self.download_urls:
          img_url_list.append(image_url)
          self.download_urls.add(image_url)

    yield {'image_urls': img_url_list}

    if len(self.download_urls) < self.max_num:
      self.start_page_num = self.start_page_num + 1
      next_page = f'{self.BASE_URL}&page={self.start_page_num}'
      yield response.follow(next_page, callback=self.parse)
```

先爬取了100张T-shirt类的商品图片，大致如下所示：


0dd3b9838498b92...

0e9f31a6d51472d6...

1aca5054f04a855b...

1aedb84375347cc...

1ca0b4f4189c894b...

1e9d7c9cbbde729d...

1f7f8766064257bae...

2a85ea8ebee74f5f...

2ebc69714840d72...

2fc34453780f973b...

3ef24962c13c1123...

4feae3a3fb75c6ad...

05ffb42ae3acc88f6...

5ae5afce59da11ed...

5c7dabe24dbee8bf...

可以看到，京东商城上爬下来的图片与FashionMNIST给的数据集相差还是蛮大的，主要在于京东商城上的图片会出现模特以及其他介绍字样，这都会对最终识别结果造成影响。此外，爬取下来的图片是彩色的，我们需要先将其转换成灰度图片，并且图片的尺寸也要进行一定的缩放。以下面这张图为例：

In [47]:
```
from PIL import Image
import numpy as np
img = Image.open('./codes/JDscraper/JDscraper/T-shirt/full/3ef24962c13c1123b5b338b3aa
plt.imshow(img)
plt.axis('off')
plt.show()
```

转化后：

In [48]:
```
image = np.array(img.resize((28, 28)).convert('L'))
image = (255-image)/255.0
plt.imshow(image, cmap='gray')
plt.axis('off')
plt.show()
```

In [49]:
```
def predict(model, input):
    with torch.no_grad():
        output = model(input)
        poss, pred = torch.max(F.softmax(output, dim=1), dim=1)
    return (pred[0].item(), poss[0].item())
```

In [51]:
```
img_tensor = torch.from_numpy(np.array([np.array([image])], dtype="float32"))
device = torch.device('cuda')
img_tensor = img_tensor.cuda()
label, poss = predict(fashion_model2, img_tensor)
print('pred:', label, ', poss:', poss)
```

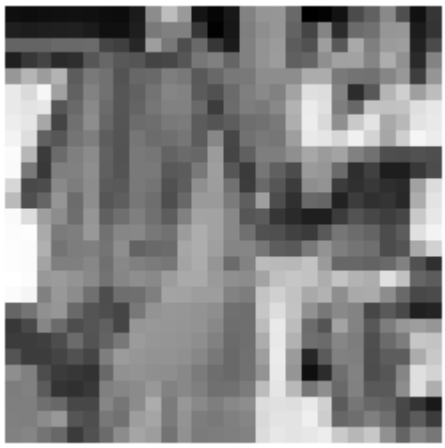
```
pred: 0 , poss: 0.8067902326583862
```

模型预测值为0，而0对应的正是T-shirt类，该次识别的置信度为80.68%。如果识别得到的置信度小于某个阈值时，我们将其认为不支持的类别。

接着，我们使用有模特和干扰字样的图片进行识别时：

In [52]:
```
img = Image.open('./codes/JDscraper/JDscraper/T-shirt/full/7cf33d97e051e6100bd37accb2
plt.imshow(img)
plt.axis('off')
plt.show()
```

In [53]:
```
image = np.array(img.resize((28, 28)).convert('L'))
image = (255-image)/255.0
plt.imshow(image, cmap='gray')
plt.axis('off')
plt.show()
```



In [54]:
```
img_tensor = torch.from_numpy(np.array([np.array([image])], dtype="float32"))
device = torch.device('cuda')
img_tensor = img_tensor.cuda()
label, poss = predict(fashion_model2, img_tensor)
print('pred:', label, ', poss:', poss)
```

```
pred: 8 , poss: 0.9999761581420898
```

此时，可以看到，模型将上述图片归类成了bag类，与真实值不符，对100张爬取下来的T-shirt图进行识别时，最终的正确率为

In [59]:
```
import os
files = os.listdir('./codes/JDscraper/JDscraper/T-shirt/full')
preds = []
for f in files:
    img = Image.open('./codes/JDscraper/JDscraper/T-shirt/full/'+f)
    image = np.array(img.resize((28, 28)).convert('L'))
    image = (255-image)/255.0
    img_tensor = torch.from_numpy(np.array([np.array([image])], dtype="float32"))
```

```
    img_tensor = img_tensor.cuda()
    label, poss = predict(fashion_model2, img_tensor)
    preds.append(label)
```

In [63]:
```
preds.count(0)/len(preds)
```

Out[63]:
```
0.13
```

最终，在爬取下来的T-shirt图片上的识别准确率只有13%。通过浏览京东商城上的商品图，发现bag类的图片相对而言比较纯净，又爬取了100张bag类别的图进行识别。


b58b67b965bdcad...


b95b0f81d8ccdb1f...


ba78190331564a08...

bb29a57d7f21ae96f...


c3b3f300385bfa50...


c711df729a7dba83...


c2032c71202f22a5...

d1dc7a2605d67aa...


d4e22854acca2f09...


d11fc6d6c9837dcc...


d55edc111a5a3a9a...


dc70a7c1d24b1db...


de08cdeb50224e3f...


dfed8021b03bca12...


e2a06ff2925f638fdd...

In [64]:
```
import os
files = os.listdir('./codes/JDscraper/JDscraper/bag/full')
preds = []
for f in files:
    img = Image.open('./codes/JDscraper/JDscraper/bag/full/'+f)
    image = np.array(img.resize((28, 28)).convert('L'))
    image = (255-image)/255.0
    img_tensor = torch.from_numpy(np.array([np.array([image])], dtype="float32"))
    img_tensor = img_tensor.cuda()
    label, poss = predict(fashion_model2, img_tensor)
    preds.append(label)
```

In [65]:
```
preds.count(8)/len(preds)
```

Out[65]:
```
0.81
```

对爬取下来的bag类别的图的识别准确率能达到81%。可以看出，使用FashionMNIST训练出来的模型的抗干扰能力比较弱，当预测的输入值中存在其他干扰时，对识别准确率影响较大。

6. 小结

在这次任务中，我学习了卷积神经网络的结构，并通过Pytorch实现，可以看到，使用Pytorch仅需要将全连接神经网络的代码的网络部分稍加修改即可转换成卷积神经网络，开发速度很快。与前2次任务相比，FashionMNIST数据集的输入量较多(28x28)，使用gpu计算加速效果明显。此外，针对训练过程中出现的过拟合现象，使用了Dropout技术进行缓解，进一步提高了模型识别的准确率。在使用训练的模型对真实场景中的图片进行识别的过程中发现，模型的抗干扰能力较弱，只有真实场景中的图片与FashionMNIST数据集中的图片相似时，才能得到一个比较准确的识别结果。