Report6 - 数独

- 姓名：刘权祥
- 学号：2019300414

任务简介

数独（shù dú, Sudoku）是源自18世纪瑞士的一种数学游戏。是一种运用纸、笔进行演算的逻辑游戏。玩家需要根据9×9盘面上的已知数字，推理出所有剩余空格的数字，并满足每一行、每一列、每一个粗线宫（3*3）内的数字均含1-9，不重复。

数独盘面是个九宫，每一宫又分为九个小格。在这八十一格中给出一定的已知数字和解题条件，利用逻辑和推理，在其他的空格上填入1-9的数字。使1-9每个数字在每一行、每一列和每一宫中都只出现一次，所以又称“九宫格”。

是否可以用卷积神经网络(CNN)解决数独问题。数独有空间特征，因为它有特殊的数字排列，而CNN擅长提取空间特征，尝试使用CNN来求解数独。

整体思路

分析数独问题并确定解决方案

数独（shù dú, Sudoku）需要我们根据9×9盘面上的已知数字，推理出所有剩余空格的数字，并满足每一行、每一列、每一个粗线宫（3*3）内的数字均含1-9，不重复。

因此，数独是具有空间特征的，它有特殊的数字排列，这样的话，我们就可以尝试使用CNN来提取空间特征，进而通过CNN求解数独问题。

思考解决数独问题的其他方案

但是，在网上查找资料的时候，我了解到CNN解决数独问题的时候，是将整个数独盘作为输入，一次正向传播就直接得到完整的解决方案，这样的话就忽视了目标之间存在的归纳偏置，以及它们是以一致的方式互相作用的，所以在一些情况下CNN模型就会出现问题。

同时，有一部分文章和评论认为使用CNN或者NN解决数独是单纯的用非常多的参数去记住数独的每一个答案，虽然不知道这个观点是否正确，但我觉得就算网络的参数不多，用CNN还是可以解决数独问题的。

CNN做数独的时候是一次性得到完整的解决方案，但我们做数独的时候，实际上是一步一步进行的：我们每次填入一个数字，然后观察这个数字带来的影响。所以数独的解决可以认为是一系列互相依赖的关系推理任务。

因此对于数独问题，循环网络的效果可能比卷积网络要好。按照上面的分析，循环关系网络（RRN）会比较适合数独问题。

解决思路

根据上面的分析，我打算自己实现CNN网络解决数独问题，同时参考网上的资料尝试一下其他的方法，再把这些方法进行对比。

由于没有找到RRN处理数独问题的相关资料，所以我打算尝试一下用RNN解决数独问题，这个部分主要是参照PyTorch中的数独RNN。

数据处理

下面是数据处理的相关代码。

这里我利用pandas来读取和处理数据，利用torch.utils.data来打包数据，这样处理后面就可以用DataLoader来加载数据，比较方便。

由于这里我需要同时满足CNN与RNN网络输入输出的要求（其实可以改成同样的，但是RNN部分只是看的差不多懂了，还不太会改，改了就出问题），所以我设置了一个one_hot参数，如果是RNN就使用one hot编码，如果是CNN就直接编码成一个9*9的矩阵。

```
import torch.utils.data as data
import torch
import pandas as pd


def create_sudoku_tensors(input_data, one_hot=False, train_split=0.7):
    """
    分割训练集和测试集
    :param input_data: 待分割的数据
    :param one_hot: 是否使用one hot编码
    :param train_split: 分割的比例
    :return: 训练集，测试集
    """
    # 数据的长度
    data_size = input_data.shape[0]

    # 给每一个数独题信息进行编码
    def one_hot_encode(s):
        zeros = torch.zeros((1, 81, 9), dtype=torch.float)
        for a in range(81):
            zeros[0, a, int(s[a]) - 1] = 1 if int(s[a]) > 0 else 0
        return zeros
```

```python
    # 给每一个数独题信息进行编码
    def encode(s):
        zeros = torch.zeros((1, 9, 9), dtype=torch.float)
        for a in range(9):
            for b in range(9):
                zeros[0, a, b] = int(s[a * 9 + b]) - 1
        return zeros

    # 得到编码内容
    if one_hot:
        quizzes_t = input_data.quizzes.apply(one_hot_encode)
        solutions_t = input_data.solutions.apply(one_hot_encode)
    else:
        quizzes_t = input_data.quizzes.apply(encode)
        solutions_t = input_data.solutions.apply(encode)
    # 将编码好的内容拼接起来
    quizzes_t = torch.cat(quizzes_t.values.tolist())
    solutions_t = torch.cat(solutions_t.values.tolist())

    # 按比例进行随机分割
    randperm = torch.randperm(data_size)
    train = randperm[:int(train_split * data_size)]
    test = randperm[int(train_split * data_size):]

    # 打包训练集和标签
    return data.TensorDataset(quizzes_t[train], solutions_t[train]), \
           data.TensorDataset(quizzes_t[test], solutions_t[test])


def create_constraint_mask():
    """
    创建一个mask
    :return: mask
    """
    constraint_mask = torch.zeros((81, 3, 81), dtype=torch.float)
    # row constraints
    for a in range(81):
        r = 9 * (a // 9)
        for b in range(9):
            constraint_mask[a, 0, r + b] = 1

    # column constraints
    for a in range(81):
        c = a % 9
        for b in range(9):
            constraint_mask[a, 1, c + 9 * b] = 1

    # box constraints
    for a in range(81):
        r = a // 9
        c = a % 9
        br = 3 * 9 * (r // 3)
        bc = 3 * (c // 3)
        for b in range(9):
            r = b % 3
            c = 9 * (b // 3)
            constraint_mask[a, 2, br + bc + r + c] = 1
```

```
    return constraint_mask


def load_dataset(filepath, one_hot=False, subsample=10000):
    """
    加载数据集
    :param filepath: 数据集文件
    :param one_hot: 是否使用one hot编码
    :param subsample: 数据集总共的行数
    :return: 训练集、测试集
    """

    dataset = pd.read_csv(filepath, sep=',')
    # 返回随机 subsample 行数据
    my_sample = dataset.sample(subsample)
    # 分割出训练集和测试集
    train_set, test_set = create_sudoku_tensors(my_sample, one_hot)
    return train_set, test_set
```

通过下面的代码我们可以测试读取的数据格式。

```
# TEST
train_set, test_set = load_dataset("./src/data/sudoku_test.csv", True, 10)
for train_quiz, train_label in train_set:
    print(train_quiz.shape)
    print(train_label.shape)
    break
train_set, test_set = load_dataset("./src/data/sudoku_test.csv", False, 10)
for train_quiz, train_label in train_set:
    print(train_quiz.shape)
    print(train_label.shape)
    break
```

```
torch.Size([81, 9])
torch.Size([81, 9])
torch.Size([9, 9])
torch.Size([9, 9])
```

搭建CNN模型

这里主要是根据report3使用的FashionCNN来搭建的，大体框架基本一样，不过增加了padding，去除了池化层，同时也放弃了dropout层。

这个网络接收N*1*9*9的输入，最后会输出N*81*9的预测结果。

N是数据批数，81是9*9的数独盘面数字列表，9是对于某一个盘面数字，它是每一种数字的概率。（这里数字的取值范围是0-8，但是代表的是数字1-9，这是因为如果取值为1-9，而数独不需要填0，标签中也没有0，所以标签的取值范围还是0-8，这样就等于是标签对应不上，交叉熵损失函数还是会运行失败）

```
import torch.nn as nn
```

```python
class SudokuCNN(nn.Module):
  def __init__(self):
    super(SudokuCNN, self).__init__()
    # 第一层卷积
    # 输入[1,9,9]
    self.layer1 = nn.Sequential(
      nn.Conv2d(in_channels=1, out_channels=64, kernel_size=(3, 3), padding=1),
      nn.BatchNorm2d(64),
      nn.ReLU(),
    )

    # 9-3+2*1+1
    # 第二层卷积
    # 输入[64,9,9]
    self.layer2 = nn.Sequential(
      nn.Conv2d(in_channels=64, out_channels=128, kernel_size=(3, 3), padding=1),
      nn.BatchNorm2d(128),
      nn.ReLU(),
    )

    # 第三层卷积
    # 输入[64，9，9]
    self.layer3 = nn.Sequential(
      nn.Conv2d(in_channels=128, out_channels=128, kernel_size=(3, 3), padding=1),
      nn.BatchNorm2d(128),
      nn.ReLU(),
    )

    # 全连接层
    # 输入[128, 9, 9]
    self.fc1 = nn.Linear(in_features=128 * 9 * 9, out_features=1000)
    self.fc2 = nn.Linear(in_features=1000, out_features=81 * 9)
    self.fc3 = nn.Softmax(dim=2)

  def forward(self, x):
    out = self.layer1(x)
    out = self.layer2(out)
    out = self.layer3(out)

    # 矩阵展开为向量
    out = out.view(out.size(0), -1)

    out = self.fc1(out)
    out = self.fc2(out)

    # 向量变化为矩阵（81*9）
    out = out.view(out.shape[0], 81, 9)

    out = self.fc3(out)
    return out
```

训练CNN模型

下面是训练CNN的代码，采用的是Adam优化器和CrossEntropyLoss损失函数。

测试的时候发现这里的学习率要取比较小，取大了loss可能就不下降，或者仅仅是下降了一点点，然后发生震荡。

```
import copy
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
from torch.autograd import Variable
from torch.utils.data import Dataset

# 尝试使用GPU
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# 加载数据集
batch_size = 100
train_set, test_set = load_dataset("./src/data/sudoku.csv", False, 50000)
train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size)

# 加载模型
model = SudokuCNN()
model.to(device)
# 损失函数
error = nn.CrossEntropyLoss()
# 定义学习率
learning_rate = 0.0005

# 定义优化器
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
# 循环次数
num_epochs = 50
# 迭代次数
count = 0
# 是否画图
display = True
# 最小迭代次数
min_epochs = 5
# 最小的损失
min_loss = float('inf')
# 最优模型
best_model = None

# Lists for visualization of loss_function and accuracy
loss_list = []
iteration_list = []
accuracy_list = []

for epoch in range(num_epochs):
  #    分批次加载数据集
  for train_quiz, train_label in train_loader:
    train_quiz, train_label = train_quiz.to(device), train_label.to(device)
    train_quiz = Variable(train_quiz.view(train_quiz.shape[0], 1, 9, 9))
    train_label = Variable(train_label.view(-1))

    # Forward pass
    outputs = model(train_quiz)
    outputs = outputs.view(-1, 9)
```

```
        loss = error(outputs, train_label.long())

        # Initializing a gradient as 0 so there is no mixing of gradient among the batches
        optimizer.zero_grad()

        # Propagating the error backward
        loss.backward()

        # Optimizing the parameters
        optimizer.step()

        count += 1

        # print('Epoch: {}, Batch: {}, Avg. Loss: {}'.format(epoch, count, loss_function.item()))

        # Testing the model
        if not (count % 50):  # It's same as "if count % 50 == 0"
            total = 0
            correct = 0

            for test_quiz, test_label in test_loader:
                test_quiz, test_label = test_quiz.to(device), test_label.to(device)
                test_quiz = Variable(test_quiz.view(test_quiz.shape[0], 1, 9, 9))
                test_label = Variable(test_label.view(-1))

                outputs = model(test_quiz)
                outputs = outputs.view(-1, 9)

                predictions = torch.max(outputs, 1)[1].to(device)
                correct += (predictions == test_label).sum()

                total += len(test_label)

            accuracy = correct * 100 / total
            loss_list.append(loss.data)
            iteration_list.append(count)
            accuracy_list.append(accuracy)

        if not (count % 100):
            print("Iteration: {}, Loss: {}, Accuracy: {}%".format(count, loss.data, accuracy))
```

```
Iteration: 100, Loss: 2.154716968536377, Accuracy: 17.909875869750977%
Iteration: 200, Loss: 2.1245620250701904, Accuracy: 22.032262802124023%
Iteration: 300, Loss: 2.091231107711792, Accuracy: 26.08452606201172%
Iteration: 400, Loss: 2.0490970611572266, Accuracy: 30.193002700805664%
Iteration: 500, Loss: 2.010570764541626, Accuracy: 34.447654724121094%
Iteration: 600, Loss: 1.973477840423584, Accuracy: 38.08658218383789%
Iteration: 700, Loss: 1.9365869760513306, Accuracy: 41.5409049987793%
Iteration: 800, Loss: 1.8946945667266846, Accuracy: 45.34855651855469%
Iteration: 900, Loss: 1.8635038137435913, Accuracy: 48.742549896240234%
Iteration: 1000, Loss: 1.837217092514038, Accuracy: 52.06106948852539%
Iteration: 1100, Loss: 1.8059731721878052, Accuracy: 54.52131652832031%
Iteration: 1200, Loss: 1.7925043106079102, Accuracy: 56.1606559753418%
Iteration: 1300, Loss: 1.7742193937301636, Accuracy: 57.427894592285156%
Iteration: 1400, Loss: 1.7613976001739502, Accuracy: 58.38139724731445%
Iteration: 1500, Loss: 1.7481008768081665, Accuracy: 58.94255065917969%
```

```
Iteration: 1600, Loss: 1.7377102375030518, Accuracy: 59.66576385498047%
Iteration: 1700, Loss: 1.732851266860962, Accuracy: 59.931114196777344%
Iteration: 1800, Loss: 1.728312373161316, Accuracy: 60.434730529785156%
Iteration: 1900, Loss: 1.7228854894638062, Accuracy: 60.944854736328125%
Iteration: 2000, Loss: 1.7063822746276855, Accuracy: 61.36444091796875%
Iteration: 2100, Loss: 1.7047476768493652, Accuracy: 61.6364631652832%
Iteration: 2200, Loss: 1.6966725587844849, Accuracy: 61.864280700683594%
Iteration: 2300, Loss: 1.6887925863265991, Accuracy: 62.192012786865234%
Iteration: 2400, Loss: 1.6844820976257324, Accuracy: 62.46205520629883%
Iteration: 2500, Loss: 1.681628704071045, Accuracy: 62.71118927001953%
Iteration: 2600, Loss: 1.6842025518417358, Accuracy: 62.79777145385742%
Iteration: 2700, Loss: 1.6700962781906128, Accuracy: 63.22929763793945%
Iteration: 2800, Loss: 1.6724343299865723, Accuracy: 63.38625717163086%
Iteration: 2900, Loss: 1.6581592559814453, Accuracy: 63.47193145751953%
Iteration: 3000, Loss: 1.6490434408187866, Accuracy: 63.73859786987305%
Iteration: 3100, Loss: 1.6508269309997559, Accuracy: 63.96732711791992%
Iteration: 3200, Loss: 1.648535132408142, Accuracy: 64.07316589355469%
Iteration: 3300, Loss: 1.6460976600646973, Accuracy: 64.2232894897461%
Iteration: 3400, Loss: 1.6367324590682983, Accuracy: 64.5616455078125%
Iteration: 3500, Loss: 1.6392115354537964, Accuracy: 64.76411437988281%
Iteration: 3600, Loss: 1.6202677488327026, Accuracy: 64.81967163085938%
Iteration: 3700, Loss: 1.6181883811950684, Accuracy: 64.9696273803711%
Iteration: 3800, Loss: 1.614449143409729, Accuracy: 65.27169036865234%
Iteration: 3900, Loss: 1.6134754419326782, Accuracy: 65.10295867919922%
Iteration: 4000, Loss: 1.6090635061264038, Accuracy: 65.56221771240234%
Iteration: 4100, Loss: 1.5994250774383545, Accuracy: 65.59876251220703%
Iteration: 4200, Loss: 1.6053227186203003, Accuracy: 65.92830657958984%
Iteration: 4300, Loss: 1.588384747505188, Accuracy: 65.90386962890625%
Iteration: 4400, Loss: 1.5881155729293823, Accuracy: 65.97966766357422%
Iteration: 4500, Loss: 1.590685248374939, Accuracy: 66.35835266113281%
Iteration: 4600, Loss: 1.5884830951690674, Accuracy: 66.50279235839844%
Iteration: 4700, Loss: 1.5854530334472656, Accuracy: 66.56123352050781%
Iteration: 4800, Loss: 1.5746759176254272, Accuracy: 66.59851837158203%
Iteration: 4900, Loss: 1.5789697170257568, Accuracy: 66.79217529296875%
Iteration: 5000, Loss: 1.5762251615524292, Accuracy: 66.85514068603516%
Iteration: 5100, Loss: 1.5584921836853027, Accuracy: 67.06889343261719%
Iteration: 5200, Loss: 1.5720216035842896, Accuracy: 67.26280212402344%
Iteration: 5300, Loss: 1.562175989151001, Accuracy: 67.406494140625%
Iteration: 5400, Loss: 1.5626013278961182, Accuracy: 67.07160186767578%
Iteration: 5500, Loss: 1.551091194152832, Accuracy: 67.47464752197266%
Iteration: 5600, Loss: 1.551571249961853, Accuracy: 67.6591796875%
Iteration: 5700, Loss: 1.544973373413086, Accuracy: 67.82979583740234%
Iteration: 5800, Loss: 1.5410361289978027, Accuracy: 67.7159652709961%
Iteration: 5900, Loss: 1.5440874099731445, Accuracy: 68.02312469482422%
Iteration: 6000, Loss: 1.5408742427825928, Accuracy: 68.20921325683594%
Iteration: 6100, Loss: 1.5385993719100952, Accuracy: 68.06814575195312%
Iteration: 6200, Loss: 1.531952977180481, Accuracy: 68.22312927246094%
Iteration: 6300, Loss: 1.533715844154358, Accuracy: 68.33497619628906%
Iteration: 6400, Loss: 1.5279567241668701, Accuracy: 68.45843505859375%
Iteration: 6500, Loss: 1.5233454704284668, Accuracy: 68.4155502319336%
Iteration: 6600, Loss: 1.5221917629241943, Accuracy: 68.8208999633789%
Iteration: 6700, Loss: 1.523041009902954, Accuracy: 68.8319320678711%
Iteration: 6800, Loss: 1.5214653015136719, Accuracy: 68.73653411865234%
Iteration: 6900, Loss: 1.5100504159927368, Accuracy: 69.00641632080078%
Iteration: 7000, Loss: 1.5138262510299683, Accuracy: 69.01736450195312%
Iteration: 7100, Loss: 1.5127310752868652, Accuracy: 69.09728240966797%
Iteration: 7200, Loss: 1.5038254261016846, Accuracy: 69.1618881225586%
Iteration: 7300, Loss: 1.5068650245666504, Accuracy: 69.27423858642578%
```

```
Iteration: 7400, Loss: 1.5071908235549927, Accuracy: 69.39086151123047%
Iteration: 7500, Loss: 1.5061938762664795, Accuracy: 69.27044677734375%
Iteration: 7600, Loss: 1.4978629350662231, Accuracy: 69.46814727783203%
Iteration: 7700, Loss: 1.5001975297927856, Accuracy: 69.54411315917969%
Iteration: 7800, Loss: 1.5003002882003784, Accuracy: 69.54493713378906%
Iteration: 7900, Loss: 1.4912667274475098, Accuracy: 69.66880798339844%
Iteration: 8000, Loss: 1.4940776824951172, Accuracy: 69.84320831298828%
Iteration: 8100, Loss: 1.4959856271743774, Accuracy: 69.9002456665039%
Iteration: 8200, Loss: 1.4937599897384644, Accuracy: 69.8642807006836%
Iteration: 8300, Loss: 1.4872251749038696, Accuracy: 69.90435791015625%
Iteration: 8400, Loss: 1.4873061180114746, Accuracy: 70.05119323730469%
Iteration: 8500, Loss: 1.4868336915969849, Accuracy: 70.16468811035156%
Iteration: 8600, Loss: 1.479051113128662, Accuracy: 70.15498352050781%
Iteration: 8700, Loss: 1.4797264337539673, Accuracy: 70.33111572265625%
Iteration: 8800, Loss: 1.4820475578308105, Accuracy: 70.33678436279297%
Iteration: 8900, Loss: 1.4802125692367554, Accuracy: 70.48592376708984%
Iteration: 9000, Loss: 1.4759306907653809, Accuracy: 70.38946533203125%
Iteration: 9100, Loss: 1.4782756567001343, Accuracy: 70.3354721069336%
Iteration: 9200, Loss: 1.477366328239441, Accuracy: 70.62855529785156%
Iteration: 9300, Loss: 1.4702521562576294, Accuracy: 70.62370300292969%
Iteration: 9400, Loss: 1.4713573455810547, Accuracy: 70.69925689697266%
Iteration: 9500, Loss: 1.4719921350479126, Accuracy: 70.89967346191406%
Iteration: 9600, Loss: 1.475656270980835, Accuracy: 70.6971206665039%
Iteration: 9700, Loss: 1.4676440954208374, Accuracy: 70.90287780761719%
Iteration: 9800, Loss: 1.4667026996612549, Accuracy: 70.93720245361328%
Iteration: 9900, Loss: 1.4661680459976196, Accuracy: 70.91629028320312%
Iteration: 10000, Loss: 1.460091233253479, Accuracy: 70.88888549804688%
Iteration: 10100, Loss: 1.4653587341308594, Accuracy: 71.04485321044922%
Iteration: 10200, Loss: 1.467116117477417, Accuracy: 71.12896728515625%
Iteration: 10300, Loss: 1.464478611946106, Accuracy: 70.97234344482422%
Iteration: 10400, Loss: 1.4584628343582153, Accuracy: 71.2210693359375%
Iteration: 10500, Loss: 1.4616717100143433, Accuracy: 71.31967163085938%
Iteration: 10600, Loss: 1.4598495960235596, Accuracy: 71.20197296142578%
Iteration: 10700, Loss: 1.4545570611953735, Accuracy: 71.21719360351562%
Iteration: 10800, Loss: 1.4592394828796387, Accuracy: 71.20929718017578%
Iteration: 10900, Loss: 1.4576523303985596, Accuracy: 71.28288269042969%
Iteration: 11000, Loss: 1.4578795433044434, Accuracy: 71.33753204345703%
Iteration: 11100, Loss: 1.4532661437988281, Accuracy: 71.40394592285156%
Iteration: 11200, Loss: 1.4541207551956177, Accuracy: 71.44477081298828%
Iteration: 11300, Loss: 1.4538921117782593, Accuracy: 71.46114349365234%
Iteration: 11400, Loss: 1.4527443647384644, Accuracy: 71.38937377929688%
Iteration: 11500, Loss: 1.4516217708587646, Accuracy: 71.36658477783203%
Iteration: 11600, Loss: 1.4533382654190063, Accuracy: 71.38288116455078%
Iteration: 11700, Loss: 1.4505600929260254, Accuracy: 71.39069366455078%
Iteration: 11800, Loss: 1.4487203359603882, Accuracy: 71.51250457763672%
Iteration: 11900, Loss: 1.4499599933624268, Accuracy: 71.6051025390625%
Iteration: 12000, Loss: 1.4480421543121338, Accuracy: 71.55358123779297%
Iteration: 12100, Loss: 1.4436876773834229, Accuracy: 71.3968734741211%
Iteration: 12200, Loss: 1.4521938562393188, Accuracy: 71.55810546875%
Iteration: 12300, Loss: 1.4474445581436157, Accuracy: 71.50106811523438%
Iteration: 12400, Loss: 1.4446181058883667, Accuracy: 71.63514709472656%
Iteration: 12500, Loss: 1.441675066947937, Accuracy: 71.57744598388672%
Iteration: 12600, Loss: 1.4454002380371094, Accuracy: 71.66764831542969%
Iteration: 12700, Loss: 1.4437016248703003, Accuracy: 71.55349731445312%
Iteration: 12800, Loss: 1.4371684789657593, Accuracy: 71.57999420166016%
Iteration: 12900, Loss: 1.442228078842163, Accuracy: 71.67620849609375%
Iteration: 13000, Loss: 1.44290030002594, Accuracy: 71.61579895019531%
Iteration: 13100, Loss: 1.4370155334472656, Accuracy: 71.60065460205078%
```

```
Iteration: 13200, Loss: 1.4392213821411133, Accuracy: 71.5341567993164%
Iteration: 13300, Loss: 1.438966155052185, Accuracy: 71.8024673461914%
Iteration: 13400, Loss: 1.4369463920593262, Accuracy: 71.59809875488281%
Iteration: 13500, Loss: 1.4334628582000732, Accuracy: 71.64510345458984%
Iteration: 13600, Loss: 1.4384645223617554, Accuracy: 71.66716003417969%
Iteration: 13700, Loss: 1.4363285303115845, Accuracy: 71.64378356933594%
Iteration: 13800, Loss: 1.4370197057724, Accuracy: 71.71299743652344%
Iteration: 13900, Loss: 1.4335864782333374, Accuracy: 71.7084732055664%
Iteration: 14000, Loss: 1.4337224960327148, Accuracy: 71.8115234375%
Iteration: 14100, Loss: 1.432400107383728, Accuracy: 71.59169006347656%
Iteration: 14200, Loss: 1.4304015636444092, Accuracy: 71.76954650878906%
Iteration: 14300, Loss: 1.436324954032898, Accuracy: 71.62395477294922%
Iteration: 14400, Loss: 1.4352554082870483, Accuracy: 71.74542999267578%
Iteration: 14500, Loss: 1.4327868223190308, Accuracy: 71.76172637939453%
Iteration: 14600, Loss: 1.4302220344543457, Accuracy: 71.83456420898438%
Iteration: 14700, Loss: 1.432002067565918, Accuracy: 71.80493927001953%
Iteration: 14800, Loss: 1.429073691368103, Accuracy: 71.71778106689453%
Iteration: 14900, Loss: 1.4286552667617798, Accuracy: 71.8115234375%
Iteration: 15000, Loss: 1.4349045753479004, Accuracy: 71.7448501586914%
Iteration: 15100, Loss: 1.431746244430542, Accuracy: 71.76329040527344%
Iteration: 15200, Loss: 1.4296308755874634, Accuracy: 71.74642181396484%
Iteration: 15300, Loss: 1.4244180917739868, Accuracy: 71.8749771118164%
Iteration: 15400, Loss: 1.4285101890563965, Accuracy: 71.89193725585938%
Iteration: 15500, Loss: 1.426265001296997, Accuracy: 71.75711059570312%
Iteration: 15600, Loss: 1.4267221689224243, Accuracy: 71.7991714477539%
Iteration: 15700, Loss: 1.4316762685775757, Accuracy: 71.79200744628906%
Iteration: 15800, Loss: 1.428530216217041, Accuracy: 71.85003662109375%
Iteration: 15900, Loss: 1.42473304271698, Accuracy: 71.76024627685547%
Iteration: 16000, Loss: 1.4257224798202515, Accuracy: 71.85118865966797%
Iteration: 16100, Loss: 1.425919532775879, Accuracy: 71.84485626220703%
Iteration: 16200, Loss: 1.4272578954696655, Accuracy: 71.77826690673828%
Iteration: 16300, Loss: 1.4233217239379883, Accuracy: 71.826904296875%
Iteration: 16400, Loss: 1.4271397590637207, Accuracy: 71.75325012207031%
Iteration: 16500, Loss: 1.4266648292541504, Accuracy: 71.77587890625%
Iteration: 16600, Loss: 1.422743320465088, Accuracy: 71.81974792480469%
Iteration: 16700, Loss: 1.422473430633545, Accuracy: 71.9375228881836%
Iteration: 16800, Loss: 1.424087405204773, Accuracy: 71.86995697021484%
Iteration: 16900, Loss: 1.4232264757156372, Accuracy: 71.7813949584961%
Iteration: 17000, Loss: 1.4207887649536133, Accuracy: 71.8785171508789%
Iteration: 17100, Loss: 1.4262917041778564, Accuracy: 71.87982940673828%
Iteration: 17200, Loss: 1.4250632524490356, Accuracy: 71.79407501220703%
Iteration: 17300, Loss: 1.423291802406311, Accuracy: 71.8446044921875%
Iteration: 17400, Loss: 1.419376254081726, Accuracy: 71.94452667236328%
Iteration: 17500, Loss: 1.4205855131149292, Accuracy: 71.94963073730469%
```

实现过程中出现的问题

前面处理数据和定义模型都挺顺利的，但是训练CNN模型这里出现的问题特别特别多，整个实现过程好像也就这部分出了问题，耽误了比较长的时间。

这里主要是因为自己对交叉熵损失函数的具体实现过程不太了解，同时对pytorch处理tensor的一些常用手法不太熟练。

一开始我是用one hot编码在这里训练和测试，但是交叉熵损失函数并不支持one hot编码，并且它还会报错，报错内容大概是“multi target”相关的，这导致我一开始没有精准的定位问题所在，然后我尝试了其他的损失函数，有的可以用但可能因为我**当时的数据格式处理本身有问题**，最后的结果不收敛。所以后面我专门去学习了pytorch中的损失函数，并且也知道了**交叉熵损失函数是不支持one hot编码的。**

之后我就修改了处理数据的代码，让CNN得到9*9的数据，而RNN继续使用one hot编码。但是因为我的网络计算出来的是batch*81*9的，并且标签也是batch*81的，这样在CrossEntropyLoss函数里面还是会报错，报错内容大概是“期望 batch*9 但是得到了 batch*81”。（batch是数据批数）

这里的报错其实也很奇怪，上网查了比较久也没解决这个问题。后面我好好想了一下，**这个数独问题可以理解为一个特殊的分类问题，我们对每一个格子进行分类，判断它是什么数字类型。**这样的话，就很自然的想到要把预测结果和标签都展开，预测结果展开为[(batch*81), 9]的二维矩阵，而标签就展开为[batch*81]的向量。

通过这样的处理方式，我终于能够开始训练这个CNN网络了。（上面找问题找了四五天，不过解决其实就花了一个下午，感觉定位问题的位置很关键！）

一个样例

这里是一个样例代码，我在这个过程中犯的错误用比较简单的形式给再现了出来。

首先定义好各种变量。

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

# 假设这是一个3分类问题，一共有4组样本

# 下面是这个模型的输出
pred_y = np.array([[0.30722019, -0.8358033, -1.24752918],
              [0.72186664, 0.58657704, -0.25026393],
              [0.16449865, -0.44255082, 0.68046693],
              [-0.52082402, 1.71407838, -1.36618063]])
pred_y = torch.from_numpy(pred_y)

# 真实的标签如下所示，很明显这里就是one hot编码
true_y_one_hot = np.array([[1, 0, 0],
              [0, 1, 0],
              [0, 1, 0],
              [0, 0, 1]])
true_y_one_hot = torch.from_numpy(true_y_one_hot)

# 这是采用普通编码的标签
# true_y_1是正确编码的，对于数独问题，我们认为这里的标签实际上代表1，2，2，3
true_y_1 = np.array([0, 1, 1, 2])
true_y_1 = torch.from_numpy(true_y_1)
# true_y_2是错误编码的，这最后会导致输出类型和标签对应不上
true_y_2 = np.array([1, 2, 2, 3])
true_y_2 = torch.from_numpy(true_y_2)
```

如果使用one hot编码，这样的运行结果如下所示：

```
loss = F.cross_entropy(pred_y, true_y_one_hot)
print(loss)
```

```
---------------------------------------------------------------------------

RuntimeError                              Traceback (most recent call last)

<ipython-input-32-64c355d4eafe> in <module>
----> 1 loss = F.cross_entropy(pred_y, true_y_one_hot)
      2 print(loss)
```

```
~/.local/lib/python3.6/site-packages/torch/nn/functional.py in cross_entropy(input, target, weight, size_average, ignore_index, reduce, reduction)
   2822     if size_average is not None or reduce is not None:
   2823         reduction = _Reduction.legacy_get_string(size_average, reduce)
-> 2824     return torch._C._nn.cross_entropy_loss(input, target, weight, _Reduction.get_enum(reduction), ignore_index)
   2825
   2826
```

```
RuntimeError: 1D target tensor expected, multi-target not supported
```

如果编码不正确，输出类型和标签不对应，就会发生下面这样的情况（在数独问题里面其实还要复杂一点）：

```
loss = F.cross_entropy(pred_y, true_y_2)
print(loss)
```

```
---------------------------------------------------------------------------

IndexError                                Traceback (most recent call last)

<ipython-input-33-a6c07789433b> in <module>
----> 1 loss = F.cross_entropy(pred_y, true_y_2)
      2 print(loss)
```

```
~/.local/lib/python3.6/site-packages/torch/nn/functional.py in cross_entropy(input, target, weight, size_average, ignore_index, reduce, reduction)
   2822     if size_average is not None or reduce is not None:
   2823         reduction = _Reduction.legacy_get_string(size_average, reduce)
-> 2824     return torch._C._nn.cross_entropy_loss(input, target, weight, _Reduction.get_enum(reduction), ignore_index)
   2825
   2826
```

```
IndexError: Target 3 is out of bounds.
```

正常运行的话，是下面这样的情况：

```
loss = F.cross_entropy(pred_y, true_y_1)
print(loss)
```

```
tensor(1.5929, dtype=torch.float64)
```

测试CNN模型

通过下面的代码可以简单测试训练好的CNN模型。

```
for test_quiz, test_label in test_loader:
    test_quiz, test_label = test_quiz.to(device), test_label.to(device)
    test_quiz = Variable(test_quiz.view(test_quiz.shape[0], 1, 9, 9))
    test_label = Variable(test_label.view(-1))

    outputs = model(test_quiz)
    outputs = outputs.view(-1, 9)

    predictions = torch.max(outputs, 1)[1].to(device)
    correct += (predictions == test_label).sum()

    total += len(test_label)

accuracy = correct * 100 / total

print("Test Accuracy: {}%".format(accuracy))
```

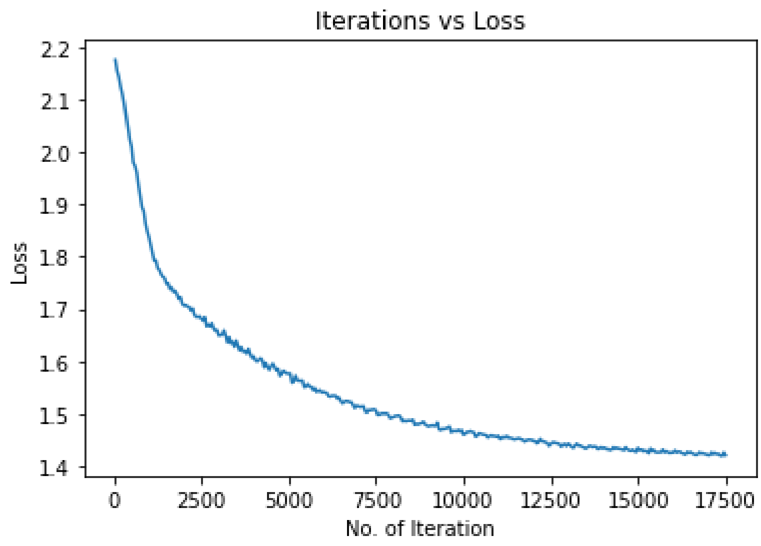
```
Test Accuracy: 71.94963073730469%
```

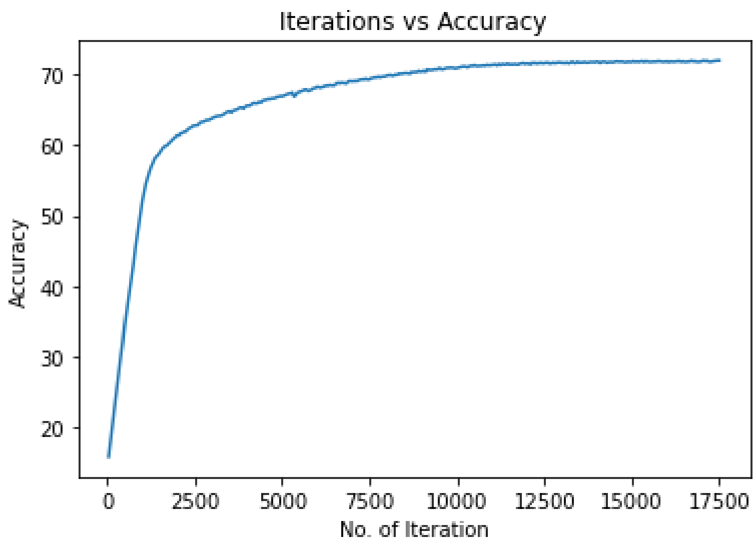
结果可视化

我们可以把训练和测试过程中的损失与准确率画出来。

```
# 画出迭代中的损失
plt.figure(1)
plt.plot(iteration_list, loss_list)
plt.xlabel("No. of Iteration")
plt.ylabel("Loss")
plt.title("Iterations vs Loss")
# 画出迭代中的准确度
plt.figure(2)
plt.plot(iteration_list, accuracy_list)
plt.xlabel("No. of Iteration")
plt.ylabel("Accuracy")
plt.title("Iterations vs Accuracy")

plt.show()
```





尝试使用RNN模型

在一开始做这个项目的时候，我查了很多关于CNN解决数独问题的资料，期间偶然了解到数独用循环网络做的效果会比较好，同时又找到一篇PyTorch中的数独RNN的文章，所以我打算尝试一下RNN模型做数独问题。

在这个环节中，RNN模型是从这篇文章里面搬运过来的，而训练过程有参考这篇文章，但为了和CNN进行对比也做了不少修改。

搭建RNN模型

下面是搭建的RNN模型。

```
import torch
import torch.nn as nn


class SudokuRNN(nn.Module):
    def __init__(self, constraint_mask, n=9, hidden1=100):
        super(SudokuRNN, self).__init__()
```

```
        self.constraint_mask = constraint_mask.view(1, n * n, 3, n * n, 1)
        self.n = n
        self.hidden1 = hidden1

        # Feature vector is the 3 constraints
        self.input_size = 3 * n

        self.l1 = nn.Linear(self.input_size,
                            self.hidden1, bias=False)
        self.a1 = nn.ReLU()
        self.l2 = nn.Linear(self.hidden1,
                            n, bias=False)
        self.softmax = nn.Softmax(dim=1)

    # x is a (batch, n^2, n) tensor
    def forward(self, x):
        n = self.n
        bts = x.shape[0]
        c = self.constraint_mask
        min_empty = (x.sum(dim=2) == 0).sum(dim=1).max()
        x_pred = x.clone()
        for a in range(min_empty):
            # score empty numbers
            constraints = (x.view(bts, 1, 1, n * n, n) * c).sum(dim=3)
            # empty cells
            empty_mask = (x.sum(dim=2) == 0)

            f = constraints.reshape(bts, n * n, 3 * n)
            y_ = self.l2(self.a1(self.l1(f[empty_mask])))

            s_ = self.softmax(y_)

            # Score the rows
            x_pred[empty_mask] = s_

            s = torch.zeros_like(x_pred)
            s[empty_mask] = s_
            # find most probable guess
            score, score_pos = s.max(dim=2)
            mmax = score.max(dim=1)[1]
            # fill it in
            nz = empty_mask.sum(dim=1).nonzero().view(-1)
            mmax_ = mmax[nz]
            ones = torch.ones(nz.shape[0])
            x.index_put_((nz, mmax_, score_pos[nz, mmax_]), ones)
        return x_pred, x
```

训练并测试

```
import copy
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
from torch.autograd import Variable
from torch.utils.data import Dataset

# FIXME 这里RNN不能使用GPU，尝试着用就会报错。感觉是定义模型的时候出的问题。。。
```

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

# 加载数据集
batch_size = 100
train_set, test_set =load_dataset("./src/data/sudoku.csv", True, 10000)
# train_set, test_set = load_dataset("./data/sudoku_test.csv", True, 1000)
train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size)

# 加载模型
model = SudokuRNN(create_constraint_mask())
model.to(device)
# 损失函数
loss_fun = nn.MSELoss()
# 定义学习率
learning_rate = 0.01
# 定义优化器
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay=0.000)
# 循环次数（用cpu训练，不敢弄太大了，训练时间太长）
num_epochs = 1
# 迭代次数
count = 0
# 是否画图
display = True
# 最小迭代次数
min_epochs = 0
# 最小的损失
min_loss = float('inf')
# 最优模型
best_model = None

# Lists for visualization of loss_function and accuracy
loss_list = []
iteration_list = []
accuracy_list = []
accuracy = 0

for epoch in range(num_epochs):
  #   分批次加载数据集
  for train_quiz, train_label in train_loader:
    train_quiz, train_label = train_quiz.to(device), train_label.to(device)
    train_quiz = Variable(train_quiz)
    train_label = Variable(train_label)

    model.train()
    optimizer.zero_grad()

    pred, mat = model(train_quiz)

    loss = loss_fun(pred, train_label)

    loss.backward()

    # Optimizing the parameters
    optimizer.step()

    count += 1
```

```
    model.eval()
    # Testing the model
    if not (count % 2):  # It's same as "if count % 50 == 0"
      total = 0
      correct = 0

      for test_quiz, test_label in test_loader:
        test_quiz, test_label = test_quiz.to(device), test_label.to(device)
        test_quiz = Variable(test_quiz)
        test_label = Variable(test_label)

        test_pred, test_fill = model(test_quiz)

        correct += (test_fill.max(dim=2)[1] == test_label.max(dim=2)[1]).sum().item()

        total += len(test_label) * 81

      accuracy = correct / total
      loss_list.append(loss.data)
      iteration_list.append(count)
      accuracy_list.append(accuracy)

      print("ACC: ", accuracy)
      print("error in cells: ", total - correct)

    if not (count % 5):
      print("Iteration: {}, Loss: {}, Accuracy: {}".format(count, loss.item(), accuracy))
```

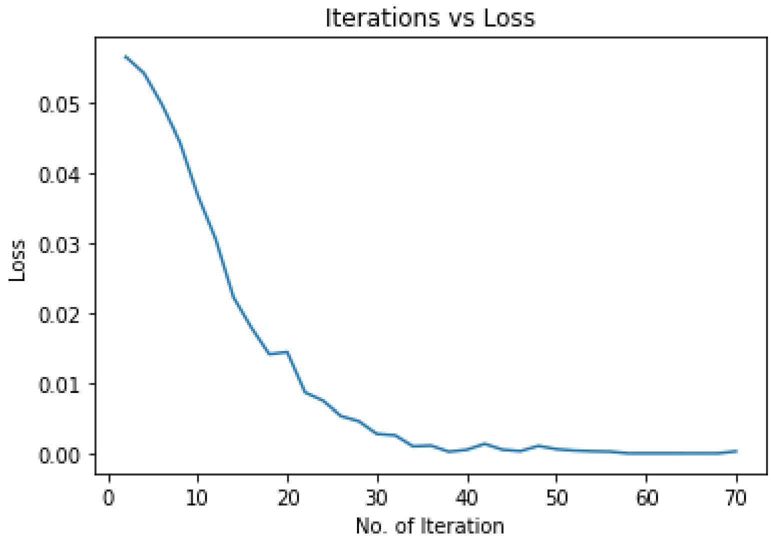
```
ACC:  0.6399711934156379
error in cells:  87487
ACC:  0.7732098765432098
error in cells:  55110
Iteration: 5, Loss: 0.05192449316382408, Accuracy: 0.7732098765432098
ACC:  0.820440329218107
error in cells:  43633
ACC:  0.837559670781893
error in cells:  39473
ACC:  0.8610576131687243
error in cells:  33763
Iteration: 10, Loss: 0.03683121129870415, Accuracy: 0.8610576131687243
ACC:  0.8852592592592593
error in cells:  27882
ACC:  0.9017078189300411
error in cells:  23885
Iteration: 15, Loss: 0.020408209413290024, Accuracy: 0.9017078189300411
ACC:  0.914522633744856
error in cells:  20771
ACC:  0.925395061728395
error in cells:  18129
ACC:  0.9415925925925926
error in cells:  14193
Iteration: 20, Loss: 0.014409015886485577, Accuracy: 0.9415925925925926
ACC:  0.9642263374485597
error in cells:  8693
ACC:  0.9754609053497942
error in cells:  5963
```

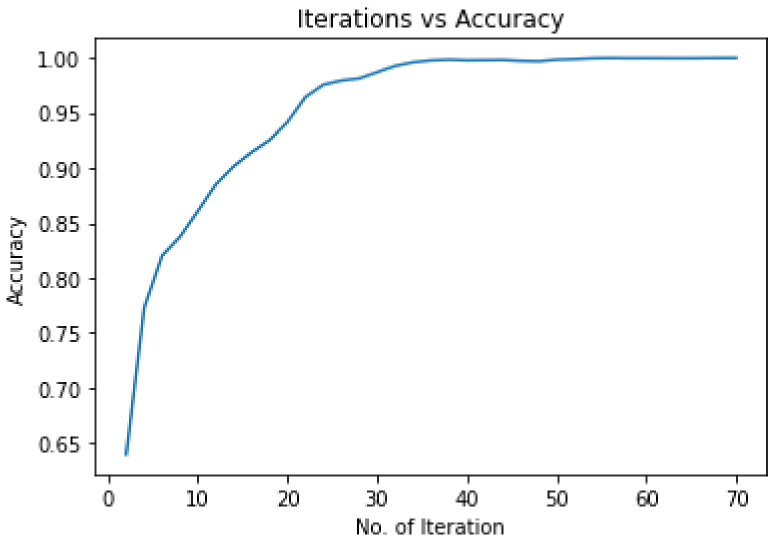
```
Iteration: 25, Loss: 0.004415363539010286, Accuracy: 0.9754609053497942
ACC:  0.9792551440329218
error in cells:  5041
ACC:  0.9811810699588477
error in cells:  4573
ACC:  0.9867818930041152
error in cells:  3212
Iteration: 30, Loss: 0.002790345111861825, Accuracy: 0.9867818930041152
ACC:  0.9926543209876543
error in cells:  1785
ACC:  0.9959300411522634
error in cells:  989
Iteration: 35, Loss: 0.0008899690001271665, Accuracy: 0.9959300411522634
ACC:  0.997641975308642
error in cells:  573
ACC:  0.9982757201646091
error in cells:  419
ACC:  0.9977160493827161
error in cells:  555
Iteration: 40, Loss: 0.0005271595437079668, Accuracy: 0.9977160493827161
ACC:  0.9978888888888889
error in cells:  513
ACC:  0.9980781893004115
error in cells:  467
Iteration: 45, Loss: 7.928410923341289e-05, Accuracy: 0.9980781893004115
ACC:  0.9971934156378601
error in cells:  682
ACC:  0.9968230452674897
error in cells:  772
ACC:  0.9982839506172839
error in cells:  417
Iteration: 50, Loss: 0.0006127970409579575, Accuracy: 0.9982839506172839
ACC:  0.9987572016460905
error in cells:  302
ACC:  0.9995267489711934
error in cells:  115
Iteration: 55, Loss: 0.0003331980260554701, Accuracy: 0.9995267489711934
ACC:  0.9997901234567901
error in cells:  51
ACC:  0.9995925925925926
error in cells:  99
ACC:  0.999641975308642
error in cells:  87
Iteration: 60, Loss: 1.6483129002153873e-05, Accuracy: 0.999641975308642
ACC:  0.9996090534979424
error in cells:  95
ACC:  0.9995720164609053
error in cells:  104
Iteration: 65, Loss: 0.0002745767415035516, Accuracy: 0.9995720164609053
ACC:  0.9996378600823045
error in cells:  88
ACC:  0.9998024691358025
error in cells:  48
ACC:  0.9998106995884773
error in cells:  46
Iteration: 70, Loss: 0.00030164315830916166, Accuracy: 0.9998106995884773
```

结果可视化

```
# 画出迭代中的损失
plt.figure(1)
plt.plot(iteration_list, loss_list)
plt.xlabel("No. of Iteration")
plt.ylabel("Loss")
plt.title("Iterations vs Loss")
# 画出迭代中的准确度
plt.figure(2)
plt.plot(iteration_list, accuracy_list)
plt.xlabel("No. of Iteration")
plt.ylabel("Accuracy")
plt.title("Iterations vs Accuracy")

plt.show()
```





对比CNN与RNN

对CNN模型进行测试

```
import torch
import time
from torch.autograd import Variable
from torch.utils.data import Dataset

print("==================对CNN模型进行测试==================")

model_path="./src/SudokuCNN_model_1.40394.pth"
test_size=10000

# 为公平起见，CNN也是用cpu
device = torch.device("cpu")

# 加载数据集
batch_size = 100
# 这里因为是测试，所以就取分割数据中的训练集部分做测试了(不是训练模型的训练集！)
test_set, _ = load_dataset("./src/data/sudoku.csv", False, test_size)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size)

# 加载模型
model = SudokuCNN()
model.to(device)
model.load_state_dict(torch.load(model_path))

# Testing the model
total = 0
correct = 0

# 开始计时
start = time.clock()

for test_quiz, test_label in test_loader:
    test_quiz, test_label = test_quiz.to(device), test_label.to(device)
    test_quiz = Variable(test_quiz.view(test_quiz.shape[0], 1, 9, 9))
    test_label = Variable(test_label.view(-1))

    outputs = model(test_quiz)
    outputs = outputs.view(-1, 9)

    predictions = torch.max(outputs, 1)[1].to(device)
    correct += (predictions == test_label).sum()

    total += len(test_label)

# 结束计时
end = time.clock()

accuracy = correct * 100 / total

print("SudokuCNN accuracy: {}%".format(accuracy))
print("SudokuCNN cells in error: {}".format(total - correct))
print("SudokuCNN time = %s" % str(end - start))

print("==================================================")
```

```
===================对CNN模型进行测试===================
SudokuCNN accuracy: 75.1141128540039%
SudokuCNN cells in error: 141103
SudokuCNN time = 8.729239000000003
======================================================
```

对RNN模型进行测试

```
print("===================对RNN模型进行测试===================")

model_path="./src/SudokuRNN_model_0.00001.pth"
test_size=10000

# 尝试使用GPU
# device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
device = torch.device("cpu")

# 加载数据集
batch_size = 100
# 这里因为是测试，所以就取分割数据中的训练集部分做测试了(不是训练模型的训练集！)
test_set, _ = load_dataset("./src/data/sudoku.csv", True, test_size)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size)

# 加载模型
model = SudokuRNN(create_constraint_mask())
model.to(device)
model.load_state_dict(torch.load(model_path))

# Testing the model
total = 0
correct = 0

# 开始计时
start = time.clock()

for test_quiz, test_label in test_loader:
    test_quiz, test_label = test_quiz.to(device), test_label.to(device)
    test_quiz = Variable(test_quiz)
    test_label = Variable(test_label)

    test_pred, test_fill = model(test_quiz)

    correct += (test_fill.max(dim=2)[1] == test_label.max(dim=2)[1]).sum().item()

    total += len(test_label) * 81

# 结束计时
end = time.clock()

accuracy = correct * 100 / total

print("SudokuRNN accuracy: {}%".format(accuracy))
print("SudokuRNN cells in error: {}".format(total - correct))
print("SudokuRNN time = %s" % str(end - start))

print("======================================================")
```

```
==================对RNN模型进行测试==================
SudokuRNN accuracy: 99.96490299823633%
SudokuRNN cells in error: 199
SudokuRNN time = 337.7354459999997
======================================================
```

比较分析

从上面的结果我们发现:

- RNN的准确率远远超过CNN，RNN达到了99.6%的准确率，而CNN只有75%的准确率。
- RNN的错误单元格数目也远比CNN少，后者的错误数是前者的100倍。
- 但是RNN解决数独问题花费的时间比CNN多得多。

总结

这是我写的第二个大作业，我觉得数独问题和之前的服装分类问题是相通的：

- 服装分类问题是一个非常经典的多分类问题，我们需要对10种服装进行分类；
- 而数独问题可以认为是一个特殊的多分类问题，我们可以说是对每一个单元格进行分类，最后通过这些单元格的分类结果，得到数独的解决方案。

之前在做服装分类问题的时候，遇到的问题主要就是pytorch不熟练和图片预处理的效果不好；而这次写数独问题的时候，pytorch的基本用法已经比较熟练了，但是因为一开始对数独问题的理解不够透彻，并且对pytorch损失函数的细节不太清楚，产生了不少问题。

通过这个项目，我觉得我主要是学习到了pytorch损失函数的原理和期望输入的数据格式，以及如何灵活的运用卷积神经网络CNN来解决分类以外的问题。

关于卷积神经网络解决数独问题是不是通过大量的参数记忆答案:

我在网上查阅资料的时候，有一部分文章和评论认为使用CNN或者NN解决数独是单纯的用非常多的参数去记住数独的每一个答案，这个观点我不知道是否正确，但是我对这个看法产生了比较大的兴趣，所以我后面有尝试修改我设计的卷积网络的参数，比如把卷积层节点调整为256或者512，再把全连接层的节点增加1000到2000个，再对网络进行训练和测试。

不过在我的测试中，我发现无论我怎么改参数（改太大会提示我GPU显存不够），这个网络的准确率最后都是75%左右，并且后面梯度下降的特别慢。所以我觉得可能卷积神经网络是通过空间特征提取，学习到了解决数独问题的函数，而不是仅仅通过大量的参数来记忆数独的答案。