Lab3

November 11, 2018

```
In [1]: from __future__ import print_function
        import numpy as np
        import torch
```

1 1

```
In [2]: x = torch.Tensor(5 , 3)
        print(x)
        print(type(x))
```

```
 2.4422e+33  3.0803e-41 -1.6391e+31
 4.5689e-41 -1.0024e+22  4.5689e-41
 4.2558e+32  3.0803e-41  4.2558e+32
 3.0803e-41  4.2558e+32  3.0803e-41
 4.2566e+32  3.0803e-41 -1.6820e+31
[torch.FloatTensor of size 5x3]

<class 'torch.FloatTensor'>
```

x was initialized by Pytorch constructor.
We just input the Tensor's size and it will create the Pytorch Tensor object with corresponded size.
The type of x is torch.Tensor

2 2

```
In [3]: y = torch.rand(5 , 3)
        print(y)
        print(type(y))
        y2 = torch.randn(5 , 3)
        print(y2)
        print(type(y2))
```

```
 0.7426  0.3589  0.0410
```

```
 0.3826  0.8828  0.7766
 0.0127  0.6751  0.9888
 0.7520  0.3937  0.1745
 0.3106  0.4066  0.9412
[torch.FloatTensor of size 5x3]

<class 'torch.FloatTensor'>

 1.1073  0.0651  0.6458
-0.6131  1.1254 -1.3151
 1.5377  1.2427 -0.9218
-0.6334  0.5704  1.7906
-0.2935 -0.2466  0.0032
[torch.FloatTensor of size 5x3]

<class 'torch.FloatTensor'>
```

Random values in y are in a uniform distribution on the interval [0,1).
The type of x is torch.Tensor.
torch.randn(5,3) will create tensor with same size but the distribution will be standard normal distribution.

3 3

```
In [4]: x=x.double()
        y=y.double()
        print(x)
        print(y)
```

```
 2.4422e+33  3.0803e-41 -1.6391e+31
 4.5689e-41 -1.0024e+22  4.5689e-41
 4.2558e+32  3.0803e-41  4.2558e+32
 3.0803e-41  4.2558e+32  3.0803e-41
 4.2566e+32  3.0803e-41 -1.6820e+31
[torch.DoubleTensor of size 5x3]


 0.7426  0.3589  0.0410
 0.3826  0.8828  0.7766
 0.0127  0.6751  0.9888
 0.7520  0.3937  0.1745
 0.3106  0.4066  0.9412
[torch.DoubleTensor of size 5x3]
```

The type displayed is torch.float64.

4 4

```
In [5]: x = torch.Tensor([[ -0.1859 , 1.3970 , 0.5236] ,
                          [ 2.3854 , 0.0707 , 2.1970] ,
                          [ -0.3587 , 1.2359 , 1.8951] ,
                          [ -0.1189 , -0.1376 , 0.4647] ,
                          [ -1.8968 , 2.0164 , 0.1092]])
        y = torch.Tensor([[ 0.4838 , 0.5822 , 0.2755] ,
                          [ 1.0982 , 0.4932 , -0.6680] ,
                          [ 0.7915 , 0.6580 , -0.5819] ,
                          [ 0.3825 , -1.1822 , 1.5217] ,
                          [ 0.6042 , -0.2280 , 1.3210]])
```

```
In [6]: print(x.shape)
        print(y.shape)
```

```
torch.Size([5, 3])
torch.Size([5, 3])
```

5 5

```
In [7]: z = torch.stack((x,y))
        z2 = torch.cat((x,y),0)
        z3 = torch.cat((x,y),1)
        print(z.shape)
        print(z2.shape)
        print(z3.shape)
```

```
torch.Size([2, 5, 3])
torch.Size([10, 3])
torch.Size([5, 6])
```

6 6

```
In [8]: print(y[4,2])
        print(z[1,4,2])
```

```
1.32099997997
1.32099997997
```

7 7

```
In [9]: print(z[:,4,2])
        print(len(z[:,4,2]))
```

```
 0.1092
 1.3210
[torch.FloatTensor of size 2]
```

```
2
```

8 8

```
In [10]: print(x+y)
         print(torch.add(x,y))
         print(x.add(y))
         torch.add(x,y,out=x)
         print(x)
```

```
 0.2979  1.9792  0.7991
 3.4836  0.5639  1.5290
 0.4328  1.8939  1.3132
 0.2636 -1.3198  1.9864
-1.2926  1.7884  1.4302
[torch.FloatTensor of size 5x3]
```

```
 0.2979  1.9792  0.7991
 3.4836  0.5639  1.5290
 0.4328  1.8939  1.3132
 0.2636 -1.3198  1.9864
-1.2926  1.7884  1.4302
[torch.FloatTensor of size 5x3]
```

```
 0.2979  1.9792  0.7991
 3.4836  0.5639  1.5290
 0.4328  1.8939  1.3132
 0.2636 -1.3198  1.9864
-1.2926  1.7884  1.4302
[torch.FloatTensor of size 5x3]
```

```
 0.2979  1.9792  0.7991
 3.4836  0.5639  1.5290
 0.4328  1.8939  1.3132
 0.2636 -1.3198  1.9864
-1.2926  1.7884  1.4302
[torch.FloatTensor of size 5x3]
```

They are printing the same output.

9 9

```
In [11]: x = torch.randn(4,4)
         y = x.view(16)
         z = x.view(-1,8)
         print(x.size(),y.size(),z.size())
```

```
torch.Size([4, 4]) torch.Size([16]) torch.Size([2, 8])
```

view(16) squeezes x into one dimension.
view(-1,8) reshape x to (xxx, 8) size where xxx is determined by its original size.

10 10

```
In [12]: x=torch.randn(10,10)
         y=torch.randn(2,100)
         x2=x.view(1,-1)
         y2=y.view(-1,2)
         res=torch.mm(x2,y2)
         print(res)
         print(res.shape)
```

```
-19.8243 -12.3032
[torch.FloatTensor of size 1x2]

torch.Size([1, 2])
```

11 11

```
In [13]: a=torch.ones(5)
         print(a)
         b=a.numpy()
         print(b)

         print(type(a))
         print(type(b))
```

```
 1
```

```
 1
 1
 1
 1
[torch.FloatTensor of size 5]

[1. 1. 1. 1. 1.]
<class 'torch.FloatTensor'>
<type 'numpy.ndarray'>
```

12 12

```
In [14]: a[0]+=1
         print(a)
         print(b)
```

```
 2
 1
 1
 1
 1
[torch.FloatTensor of size 5]

[2. 1. 1. 1. 1.]
```

They match. They share their underlying memory locations.

13 13

```
In [15]: a.add_(1)
         print(a,b)
         a[:]+=1
         print(a,b)
         a=a.add(1)
         print(a,b)
```

```
 3
 2
 2
 2
 2
[torch.FloatTensor of size 5]
 [3. 2. 2. 2. 2.]
```

```
 4
 3
 3
 3
 3
[torch.FloatTensor of size 5]
 [4. 3. 3. 3. 3.]

 5
 4
 4
 4
 4
[torch.FloatTensor of size 5]
 [4. 3. 3. 3. 3.]
```

All of them will modify the value of a.
But a.add_(1) and a[:]+=1 will also modify the value of b. a.add(1) will not.

14 14

```
In [16]: a=np.ones(5)
         b=torch.from_numpy(a)
         np.add(a,1,out=a)
         print(a)
         print(b)
```

```
[2. 2. 2. 2. 2.]

 2
 2
 2
 2
 2
[torch.DoubleTensor of size 5]
```

15 15

```
In [17]: x=torch.randn(5,3)
         y=torch.randn(5,3)
         if torch.cuda.is_available():
             x=x.cuda()
             y=y.cuda()
```

```
    z=x+y
print(z)
if torch.cuda.is_available():
    print(z.cpu())
```

```
 0.6352  0.2053  2.4652
 3.0973 -1.5817 -0.6140
 0.0272 -1.4052  1.1294
-0.3573  1.6090  2.2152
 0.2849  1.0794 -3.4635
[torch.cuda.FloatTensor of size 5x3 (GPU 0)]
```

```
 0.6352  0.2053  2.4652
 3.0973 -1.5817 -0.6140
 0.0272 -1.4052  1.1294
-0.3573  1.6090  2.2152
 0.2849  1.0794 -3.4635
[torch.FloatTensor of size 5x3]
```

Create x and y and calculate z=x+y.
If gpu is available, then transfer tensor to cuda tensor firstly and then calculate it by gpu.
z.cpu() could also transfer z from cuda tensor to cpu tensor.

16 16

```
In [18]: print(z.cpu().numpy())
         print(z.numpy())
```

```
[[ 0.6352043   0.20527676  2.4652302 ]
 [ 3.0973043  -1.5816753  -0.6140161 ]
 [ 0.02719772 -1.4051727   1.1294484 ]
 [-0.35725167  1.6089787   2.2151823 ]
 [ 0.28489646  1.0793769  -3.463544  ]]
```

```
        RuntimeErrorTraceback (most recent call last)

        <ipython-input-18-11d5c486e6eb> in <module>()
          1 print(z.cpu().numpy())
    ----> 2 print(z.numpy())
```

```
RuntimeError: can't convert CUDA tensor to numpy (it doesn't support GPU arrays). Use .c
```

Unfortunately, the second statement is invalid for Pytorch.

17 17

```
In [19]: import torch.autograd as ag
         x=ag.Variable(torch.ones(2,2),requires_grad=True)
         print(x)
         y=x+2
         print(y)
         print(type(y))
```

```
Variable containing:
 1  1
 1  1
[torch.FloatTensor of size 2x2]

Variable containing:
 3  3
 3  3
[torch.FloatTensor of size 2x2]

<class 'torch.autograd.variable.Variable'>
```

18 18

```
In [20]: z=y*y*3
         f=z.mean()
         print(z,f)
```

```
Variable containing:
 27  27
 27  27
[torch.FloatTensor of size 2x2]
 Variable containing:
 27
[torch.FloatTensor of size 1]
```

$$f = \frac{1}{n}\sum_{i=1}^{n} 3(x_i+2)^2$$

19 19

```
In [21]: f.backward()
```

$$\nabla_x f(x_i) = \frac{6(x_i + 2)}{n}$$

Based on the equation, the gradient should be 18/4 = 4.5

20 20

```
In [22]: x.grad

Out[22]: Variable containing:
          4.5000  4.5000
          4.5000  4.5000
         [torch.FloatTensor of size 2x2]
```

This results indicates that the gradient obtained by Pytorch is the same as our mathematical derivation.

21 21

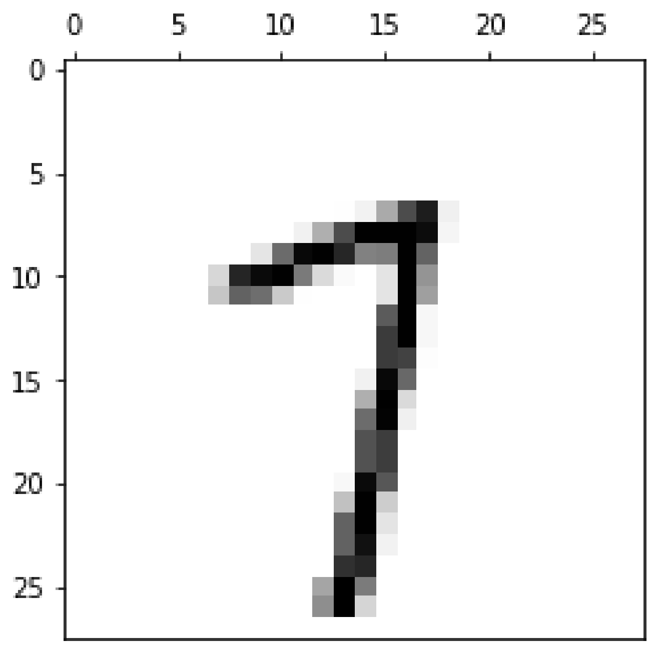
```
In [43]: import MNISTtools
         import numpy as np
         from matplotlib import pyplot

In [44]: xtrain, ltrain = MNISTtools.load(dataset = "training", path = "/datasets/MNIST")
         xtest, ltest = MNISTtools.load(dataset = "testing", path = "/datasets/MNIST")
         print(xtrain.shape)
         print(ltrain.shape)
         print(xtest.shape)
         print(ltest.shape)

(784, 60000)
(60000,)
(784, 10000)
(10000,)
```

22 22

```
In [45]: xtrain = np.transpose(xtrain.reshape(28,28,1,60000),[3,2,0,1])
         xtest = np.transpose(xtest.reshape(28,28,1,10000),[3,2,0,1])
```

23 23

```
In [46]: import matplotlib as pyplot
         MNISTtools.show(xtrain[42,0,:,:])
         print(ltrain[42])
```




```
7
```

This indicates that our preprocess is correct.

24 24

```
In [47]: import torch.autograd as ag
         xtrain=ag.Variable(torch.from_numpy(xtrain),requires_grad=True).double()
         ltrain=ag.Variable(torch.from_numpy(ltrain),requires_grad=False).double()
         xtest=ag.Variable(torch.from_numpy(xtest),requires_grad=False).double()
```

25 25

```
In [48]: import torch.nn as nn
```

25.0.1 Input = 1 x 28 x 28

25.0.2 conv1 = 6 x 24 x 24

25.0.3 pool1 = 6 x 12 x 12

25.0.4 conv2 = 16 x 8 x 8

25.0.5 pool2 = 16 x 4 x 4

26 26

In [49]:
```
import torch.nn as nn
import torch.nn.functional as F
# This is our neural networks class that inherits from nn. Module
class LeNet(nn.Module):
# Here we define our network structure
    def __init__(self):
        super(LeNet,self).__init__()
        self.conv1=nn.Conv2d(1,6,5).double()
        self.conv2=nn.Conv2d(6,16,5).double()
        self.fc1=nn.Linear(16*4*4,120).double()
        self.fc2=nn.Linear(120,84).double()
        self.fc3=nn.Linear(84,10).double()
    # Here we define one forward pass through the network
    def forward(self,x):
        x=F.max_pool2d(F.relu(self.conv1(x)),(2,2))
        x=F.max_pool2d(F.relu(self.conv2(x)),(2,2))
        x=x.view(-1,self.num_flat_features(x))
        x=F.relu(self.fc1(x))
        x=F.relu(self.fc2(x))
        x=self.fc3(x)
        return x
    # Determine the number of features in a batch of tensors
    def num_flat_features(self,x):
        size=x.size()[1:]
        return np.prod(size)
net = LeNet()
print(net)
```

```
LeNet(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=256, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

27 27

```
In [50]: params=list(net.parameters())
         for i in range(len(params)):
             print(i,params[i].size())
```

```
0 torch.Size([6, 1, 5, 5])
1 torch.Size([6])
2 torch.Size([16, 6, 5, 5])
3 torch.Size([16])
4 torch.Size([120, 256])
5 torch.Size([120])
6 torch.Size([84, 120])
7 torch.Size([84])
8 torch.Size([10, 84])
9 torch.Size([10])
```

The params for indicies 0 and 2 are [num of output channels, num of input channels, kernel width, kernel height], which corresponds to the kernels for conv layers.

The params for indicies 4,6,8 are the num of input features and the num of output features for fully connected layers.

The params for the odd indices are bias for each layer.

28 28

```
In [51]: yinit = net(xtest)
         print(100*np.mean(ltest == yinit.data.numpy().T.argmax(axis=0)))
```

```
10.32
```

29 29

```
In [52]: N = xtrain.size()[0] # Training set size
         B = 100 # Minibacth size
         NB = 600 # Number of minibatches
         T = 10 # Number of epochs
         gamma = .001 # learning rate
         rho = .9 # momentum
         criterion = nn.CrossEntropyLoss()
         optimizer = torch.optim.SGD(net.parameters(),
                                     lr = gamma,
                                     momentum = rho)
```

30 30

```
In [54]: for epoch in range(T):
             running_loss = 0.0
             idxminibatches = np.random.permutation(NB) # shuffling
             for k in range(NB):
                 i = idxminibatches[k] # index of minibatch
                 # Extract i-th minibatch from xtrain and ltrain
                 idxsmp = range(i*B,i*B+B) # indices of samples for i-th minibatch
                 inputs = xtrain[idxsmp]
                 labels = ltrain[idxsmp]
                 # Initialize the gradients to zero
                 optimizer.zero_grad()
                 # Forward propagation
                 outputs = net(inputs)
                 # Error evaluation
                 loss = criterion(outputs,labels.long())
                 # Back propagation
                 loss.backward()
                 # Parameter update
                 optimizer.step()
                 # Print averaged loss per minibatch every 100 mini - batches
                 running_loss += loss[0]
                 if k % 100 == 99:
                     print('[%d, %5d] loss: %.3f'%
                           (epoch + 1,k + 1,running_loss*1.0/100))
                     running_loss = 0.0
         print ('Finished Training')
```

```
[1,   100] loss: 1.219
[1,   200] loss: 0.252
[1,   300] loss: 0.167
[1,   400] loss: 0.157
[1,   500] loss: 0.125
[1,   600] loss: 0.125
[2,   100] loss: 0.100
[2,   200] loss: 0.101
[2,   300] loss: 0.087
[2,   400] loss: 0.094
[2,   500] loss: 0.080
```

```
    KeyboardInterruptTraceback (most recent call last)

    <ipython-input-54-76ecf1db2eb3> in <module>()
```

```
     15         loss = criterion(outputs,labels.long())
     16         # Back propagation
---> 17         loss.backward()
     18         # Parameter update
     19         optimizer.step()


    /opt/conda/lib/python2.7/site-packages/torch/autograd/variable.pyc in backward(self, gra
    165                 Variable.
    166         """
--> 167         torch.autograd.backward(self, gradient, retain_graph, create_graph, retain_v
    168
    169     def register_hook(self, hook):


    /opt/conda/lib/python2.7/site-packages/torch/autograd/__init__.pyc in backward(variables
     97
     98     Variable._execution_engine.run_backward(
---> 99         variables, grad_variables, retain_graph)
    100
    101


    KeyboardInterrupt:
```

31 31

In [59]:
```python
xtrain, ltrain = MNISTtools.load(dataset = "training", path = "/datasets/MNIST")
xtest, ltest = MNISTtools.load(dataset = "testing", path = "/datasets/MNIST")

xtrain = np.transpose(xtrain.reshape(28,28,1,60000),[3,2,0,1])
xtest = np.transpose(xtest.reshape(28,28,1,10000),[3,2,0,1])

import torch.autograd as ag
if torch.cuda.is_available():
    xtrain=ag.Variable(torch.from_numpy(xtrain).cuda(),requires_grad=True).double()
    ltrain=ag.Variable(torch.from_numpy(ltrain).cuda(),requires_grad=False).double()
    xtest=ag.Variable(torch.from_numpy(xtest).cuda(),requires_grad=False).double()
else:
    xtrain=ag.Variable(torch.from_numpy(xtrain),requires_grad=True).double()
    ltrain=ag.Variable(torch.from_numpy(ltrain),requires_grad=False).double()
    xtest=ag.Variable(torch.from_numpy(xtest),requires_grad=False).double()

import torch.nn as nn
import torch.nn.functional as F
# This is our neural networks class that inherits from nn. Module
```

```
class LeNet(nn.Module):
# Here we define our network structure
    def __init__(self):
        super(LeNet,self).__init__()
        self.conv1=nn.Conv2d(1,6,5).double()
        self.conv2=nn.Conv2d(6,16,5).double()
        self.fc1=nn.Linear(16*4*4,120).double()
        self.fc2=nn.Linear(120,84).double()
        self.fc3=nn.Linear(84,10).double()

    # Here we define one forward pass through the network
    def forward(self,x):
        x=F.max_pool2d(F.relu(self.conv1(x)),(2,2))
        x=F.max_pool2d(F.relu(self.conv2(x)),(2,2))
        x=x.view(-1,self.num_flat_features(x))
        x=F.relu(self.fc1(x))
        x=F.relu(self.fc2(x))
        x=self.fc3(x)
        return x

    # Determine the number of features in a batch of tensors
    def num_flat_features(self,x):
        size=x.size()[1:]
        return np.prod(size)

net = LeNet()
if torch.cuda.is_available():
    net = net.cuda()

print(net)
```

```
LeNet(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=256, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

```
In [60]: N = xtrain.size()[0] # Training set size
         B = 100 # Minibacth size
         NB = 600 # Number of minibatches
         T = 10 # Number of epochs
         gamma = .001 # learning rate
         rho = .9 # momentum
         criterion = nn.CrossEntropyLoss()
         optimizer = torch.optim.SGD(net.parameters(),
```

```
                                 lr = gamma,
                                 momentum = rho)

    for epoch in range(T):
        running_loss = 0.0
        idxminibatches = np.random.permutation(NB) # shuffling
        for k in range(NB):
            i = idxminibatches[k] # index of minibatch
            # Extract i-th minibatch from xtrain and ltrain
            idxsmp = range(i*B,i*B+B) # indices of samples for i-th minibatch
            inputs = xtrain[idxsmp]
            labels = ltrain[idxsmp]

            if torch.cuda.is_available():
                inputs, labels = inputs.cuda(), labels.cuda()

            # Initialize the gradients to zero
            optimizer.zero_grad()
            # Forward propagation
            outputs = net(inputs)
            # Error evaluation
            loss = criterion(outputs,labels.long())
            # Back propagation
            loss.backward()
            # Parameter update
            optimizer.step()
            # Print averaged loss per minibatch every 100 mini - batches
            running_loss += loss[0]
            if k % 100 == 99:
                print('[%d, %5d] loss: %.3f'%
                        (epoch + 1,k + 1,running_loss*1.0/100))
                running_loss = 0.0
    print ('Finished Training')
```

```
[1,   100] loss: 0.840
[1,   200] loss: 0.241
[1,   300] loss: 0.159
[1,   400] loss: 0.149
[1,   500] loss: 0.148
[1,   600] loss: 0.121
[2,   100] loss: 0.096
[2,   200] loss: 0.090
[2,   300] loss: 0.089
[2,   400] loss: 0.082
[2,   500] loss: 0.082
[2,   600] loss: 0.082
[3,   100] loss: 0.066
[3,   200] loss: 0.067
```

```
[3,   300] loss: 0.069
[3,   400] loss: 0.054
[3,   500] loss: 0.060
[3,   600] loss: 0.053
[4,   100] loss: 0.045
[4,   200] loss: 0.051
[4,   300] loss: 0.053
[4,   400] loss: 0.040
[4,   500] loss: 0.049
[4,   600] loss: 0.045
[5,   100] loss: 0.049
[5,   200] loss: 0.034
[5,   300] loss: 0.039
[5,   400] loss: 0.040
[5,   500] loss: 0.043
[5,   600] loss: 0.037
[6,   100] loss: 0.030
[6,   200] loss: 0.032
[6,   300] loss: 0.036
[6,   400] loss: 0.033
[6,   500] loss: 0.037
[6,   600] loss: 0.033
[7,   100] loss: 0.028
[7,   200] loss: 0.028
[7,   300] loss: 0.027
[7,   400] loss: 0.029
[7,   500] loss: 0.030
[7,   600] loss: 0.032
[8,   100] loss: 0.021
[8,   200] loss: 0.034
[8,   300] loss: 0.026
[8,   400] loss: 0.026
[8,   500] loss: 0.024
[8,   600] loss: 0.027
[9,   100] loss: 0.016
[9,   200] loss: 0.022
[9,   300] loss: 0.025
[9,   400] loss: 0.026
[9,   500] loss: 0.024
[9,   600] loss: 0.021
[10,   100] loss: 0.015
[10,   200] loss: 0.016
[10,   300] loss: 0.021
[10,   400] loss: 0.019
[10,   500] loss: 0.021
[10,   600] loss: 0.020
Finished Training
```

32   32

In [61]:
```
if torch.cuda.is_available():
    xtest = xtest.cuda()

yinit = net(xtest)

if torch.cuda.is_available():
    print(100*np.mean(ltest == yinit.cpu().data.numpy().T.argmax(axis=0)))
else:
    print(100*np.mean(ltest == yinit.data.numpy().T.argmax(axis=0)))
```

```
98.86
```

Improved from 10.32% accuracy to 98.86% accuracy.