計算機視覺作業

干皓丞，2101212850, 信息工程學院

2021 年 10 月 30 日

1　作業目標與章節摘要

根據 W6_MNIST_FC.ipynb 改良程式碼好獲得更好的效果。在此可以使用增加卷積層結構，或者嘗試增加 dropout 或者 BN 技術等，訓練出盡可能高的 MNIST 分類效果，原始程式碼在名為 kancheng/kan-cs-report-in-2021 的 Github 專案下，可以在 CV/pytorch-tensorflow-mnist/code 下找到。該報告第一節為 Pytorch MNIST Training 課堂程式碼改良，第二節為 TensorFlow MNIST Training，第三節為 CNN 方法與重要文獻，而附件 1 與附件 2 則為 Pytorch MNIST Training 中的兩次最好測試結果。

本次作业

北京大学 PEKING UNIVERSITY

- 在W6_MNIST_FC.ipynb基础上，增加卷积层结构/增加dropout或者BN技术等，训练出尽可能高的MNIST分类效果。

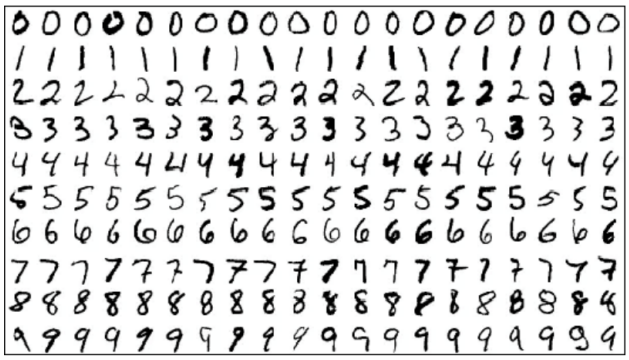

Fig. 1. 作業目標

2 Pytorch MNIST Training

Pytorch MNIST Training 原始的範例程式碼中 Test Accuracy 結果為 0.914，而自己 MacBook Pro (Retina, 15-inch, Mid 2014) 實際測試後，所跑出的 Test Accuracy 結果為 0.894 and 0.901。原始程式碼範例如下所示:

```
import torch
import torch.nn as nn
import torch.utils.data as Data
import torchvision
import torch.nn.functional as F
import numpy as np
# torch.manual_seed(1)
EPOCH = 2
LR = 0.001
DOWNLOAD_MNIST = True

train_data = torchvision.datasets.MNIST(root='./mnist/', train=True,
    transform=torchvision.transforms.ToTensor(),
                                        download=DOWNLOAD_MNIST, )
test_data = torchvision.datasets.MNIST(root='./mnist/', train=False)
train_x = torch.unsqueeze(train_data.data, dim=1).type(torch.FloatTensor)
    / 255.
train_y = train_data.targets
print(train_x.shape)
test_x = torch.unsqueeze(test_data.data, dim=1).type(torch.FloatTensor)
    [:2000] / 255.  # Tensor on GPU
test_y = test_data.targets[:2000]
test_x.shape
import matplotlib.pyplot as plt
plt.imshow(test_x[1,0,:,:].numpy(), 'gray')
test_y[:10]
class FC(nn.Module):
    def __init__(self):
        super(FC, self).__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 10)
        self.fc3 = nn.Linear(10, 10)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
```

```
34          x = F.relu(x)
35          x = self.fc2(x)
36          x = F.relu(x)
37          x = self.fc3(x)
38
39          output = x
40          return output
41
42
43  fc = FC()
44
45  optimizer = torch.optim.Adam(fc.parameters(), lr=LR)
46  # loss_func = nn.MSELoss()
47  loss_func = nn.CrossEntropyLoss()
48
49  data_size = 20000
50  batch_size = 50
51
52  for epoch in range(EPOCH):
53      random_indx = np.random.permutation(data_size)
54      for batch_i in range(data_size // batch_size):
55          indx = random_indx[batch_i * batch_size:(batch_i + 1) *
                batch_size]
56
57          b_x = train_x[indx, :]
58          b_y = train_y[indx]
59  #         print(b_x.shape)
60  #         print(b_y.shape)
61
62          output = fc(b_x)
63          loss = loss_func(output, b_y)
64
65          loss.backward()
66          optimizer.step()
67          optimizer.zero_grad()
68
69          if batch_i % 50 == 0:
70              test_output = fc(test_x)
71              pred_y = torch.max(test_output, 1)[1].data.squeeze()
72              # pred_y = torch.max(test_output, 1)[1].data.squeeze()
73              accuracy = torch.sum(pred_y == test_y).type(torch.FloatTensor
```

```
                ) / test_y.size(0)
74              print('Epoch: ', epoch, '| train loss: %.4f' % loss.data.cpu
                    ().numpy(), '| test accuracy: %.3f' % accuracy)
75
76  test_output = fc(test_x[:10])
77  pred_y = torch.max(test_output, 1)[1].data.squeeze()  # move the
        computation in GPU
78
79  print(pred_y, 'prediction number')
80  print(test_y[:10], 'real number')
81
82  test_output = fc(test_x[:1])
83  pred_y = torch.max(test_output, 1)[1].data.squeeze()  # move the
        computation in GPU
84
85  print(pred_y, 'prediction number')
86  print(test_y[:1], 'real number')
87  test_output
88  test_x[:1].shape
89  plt.imshow(test_x[:1].numpy().squeeze(), 'gray')
90
91  import torch
92  torch.eye(10)
93  0, 1, 2, ..., 9
```

在此針對 FC class 和 batch_size 進行測試，一開始的 FC 則稱之為 FC3 。FC 為控制訓練的層，而所謂的 Batch Size 為一個重要參數，該參數的意義在於做出在記憶體效率跟記憶體容量之間尋找最佳平衡，若編號為 FC3 且 Batch Size 為 50 則編號表述為 FC3-batch-50。此次實驗共有 FC3、FC4-1、FC4-2、FC7，Batch Size 為 10、30、50。

```
1   # FC3
2   class FC(nn.Module):
3       def __init__(self):
4           super(FC, self).__init__()
5           self.fc1 = nn.Linear(784, 256)
6           self.fc2 = nn.Linear(256, 10)
7           self.fc3 = nn.Linear(10, 10)
8
9       def forward(self, x):
10          x = x.view(x.size(0), -1)
11          x = self.fc1(x)
12          x = F.relu(x)
```

```
        x = self.fc2(x)
        x = F.relu(x)
        x = self.fc3(x)
        output = x
        return output
fc = FC()
```

此 FC 則稱之為 FC4-1 。

```
# FC4-1
class FC(nn.Module):
    def __init__(self):
        super(FC, self).__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)
        self.fc4 = nn.Linear(10, 10)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.relu(x)
        x = self.fc3(x)
        x = F.relu(x)
        x = self.fc4(x)
        x = F.relu(x)
        output = x
        return output
fc = FC()
```

此 FC 則稱之為 FC4-2 。

```
# FC4-2
class FC(nn.Module):
    def __init__(self):
        super(FC, self).__init__()
        self.fc1 = nn.Linear(784, 1256)
        self.fc2 = nn.Linear(1256, 128)
        self.fc3 = nn.Linear(128, 10)
        self.fc4 = nn.Linear(10, 10)

    def forward(self, x):
```

```
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.relu(x)
        x = self.fc3(x)
        x = F.relu(x)
        x = self.fc4(x)
        x = F.relu(x)
        output = x
        return output
fc = FC()
```

此 FC 則稱之為 FC7 。

```
# FC7
class FC(nn.Module):
    def __init__(self):
        super(FC, self).__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 64)
        self.fc4 = nn.Linear(64, 32)
        self.fc5 = nn.Linear(32, 16)
        self.fc6 = nn.Linear(16, 10)
        self.fc7 = nn.Linear(10, 10)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.relu(x)
        x = self.fc3(x)
        x = F.relu(x)
        x = self.fc4(x)
        x = F.relu(x)
        x = self.fc5(x)
        x = F.relu(x)
        x = self.fc6(x)
        x = F.relu(x)
        x = self.fc7(x)
```

```
28          output = x
29          return output
30  fc = FC()
```

結果由下表所示，在 FC7 在 Batch Size 為 50 的情況下，效果並不是很理想，跟 FC3 的結果相近，反而是 FC4-1 跟 FC4-2 這兩者有著不錯的表現。前者 FC4-1 在 Batch Size 為 50 時有 0.954，在 Batch Size 為 30 時，FC4-1 第一次實機測試有 0.941，而第二次實機測試則有 0.930，FC4-2 則在 Batch Size 為 30 時做到 0.961。

FC 代號	Epoch	Train Loss	Test Accuracy	Batch Size	備註
FC3	1	0.2856	0.894	50	第一次測試
FC3	1	0.3801	0.913	50	第二次測試
FC4-1	1	0.1782	0.954	50	第一次測試
FC7	1	0.2008	0.886	50	第一次測試
FC4-1	1	0.2815	0.941	30	第一次測試
FC4-1	1	0.0679	0.930	30	第二次測試
FC4-2	1	0.0153	0.961	30	第一次測試
FC7	1	0.0089	0.928	10	第一次測試

在此測試只是單純的增加層，而且該測試並無做到相當全面的程度，實際上還可在投入 dropout 或者 BN 技術等，應該能夠得到更好的效果，同時此作業嘗試在追 MNINT 的過程訓練等中，找 TensorFlow 等不同方案或者是同為 Pytorch 但是為不同思路的實現，包含當下 CNN 歷年來的發展，如 Shufflenet V2 等。而測試過程中最好的兩次結果則在附件中呈現。

3 Tensorflow MNIST Training

Tensorflow MNIST Training 的 Code 為 Google 在其 Tensorflow 平臺的官方範例嘗試，而環境部屬則與 Pytorch 相似，在此則不詳述。

```
# Training a neural network on MNIST with Keras
import tensorflow as tf
import tensorflow_datasets as tfds
(ds_train, ds_test), ds_info = tfds.load(
    'mnist',
    split=['train', 'test'],
    shuffle_files=True,
    as_supervised=True,
    with_info=True,
)

show = tfds.show_examples(ds_test, ds_info)

print("Number of original training examples:", len(ds_train))
print("Number of original test examples:", len(ds_test))

def normalize_img(image, label):
  """Normalizes images: `uint8` -> `float32`."""
  return tf.cast(image, tf.float32) / 255., label

ds_train = ds_train.map(
    normalize_img, num_parallel_calls=tf.data.AUTOTUNE)
ds_train = ds_train.cache()
ds_train = ds_train.shuffle(ds_info.splits['train'].num_examples)
ds_train = ds_train.batch(128)
ds_train = ds_train.prefetch(tf.data.AUTOTUNE)
ds_test = ds_test.map(
    normalize_img, num_parallel_calls=tf.data.AUTOTUNE)
ds_test = ds_test.batch(128)
ds_test = ds_test.cache()
ds_test = ds_test.prefetch(tf.data.AUTOTUNE)

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(10)
])
```

```
model.compile(
    optimizer=tf.keras.optimizers.Adam(0.001),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=[tf.keras.metrics.SparseCategoricalAccuracy()],
)

model.fit(
    ds_train,
    epochs=6,
    validation_data=ds_test,
)
```

```
In [25]: model = tf.keras.models.Sequential([
           tf.keras.layers.Flatten(input_shape=(28, 28)),
           tf.keras.layers.Dense(128, activation='relu'),
           tf.keras.layers.Dense(10)
         ])
         model.compile(
             optimizer=tf.keras.optimizers.Adam(0.001),
             loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
             metrics=[tf.keras.metrics.SparseCategoricalAccuracy()],
         )

         model.fit(
             ds_train,
             epochs=6,
             validation_data=ds_test,
         )

         Epoch 1/6
         469/469 [==============================] - 3s 2ms/step - loss: 0.3552 - sparse_categorical_accuracy: 0.9015 - val_loss: 0.1884 - val_spar
         se_categorical_accuracy: 0.9479
         Epoch 2/6
         469/469 [==============================] - 1s 2ms/step - loss: 0.1643 - sparse_categorical_accuracy: 0.9531 - val_loss: 0.1362 - val_spar
         se_categorical_accuracy: 0.9612
         Epoch 3/6
         469/469 [==============================] - 1s 2ms/step - loss: 0.1181 - sparse_categorical_accuracy: 0.9663 - val_loss: 0.1064 - val_spar
         se_categorical_accuracy: 0.9685
         Epoch 4/6
         469/469 [==============================] - 1s 2ms/step - loss: 0.0904 - sparse_categorical_accuracy: 0.9740 - val_loss: 0.0895 - val_spar
         se_categorical_accuracy: 0.9734
         Epoch 5/6
         469/469 [==============================] - 1s 2ms/step - loss: 0.0722 - sparse_categorical_accuracy: 0.9794 - val_loss: 0.0893 - val_spar
         se_categorical_accuracy: 0.9720
         Epoch 6/6
         469/469 [==============================] - 1s 2ms/step - loss: 0.0604 - sparse_categorical_accuracy: 0.9826 - val_loss: 0.0806 - val_spar
         se_categorical_accuracy: 0.9746
Out[25]: <keras.callbacks.History at 0x7fb69ab99c40>
```

Fig. 2. Tensorflow MNIST Training

4 Convolutional Neural Networks

在此將 CNN 歷年來重要的研究文獻整理如下所示:

方法名	年代	研究者	備註
LeNet	1998	Y. Lecun et al.	
AlexNet	2012	Alex Krizhevsky et al.	
VGG	2014.09	Karen Simonyan et al.	
Inception Net	2014.09	Christian Szegedy et al.	Google Inc.
Inception Net V2	2015.02	Sergey Ioffe et al.	Google Inc.
Inception Net V3	2015.12	Christian Szegedy et al.	Google Inc.
Inception Net V4	2016.02	Christian Szegedy et al.	Google Inc.
Xception	2016.10	François Chollet	Google Inc.
ResNet	2015.12	Kaiming He et al.	Microsoft Research
DenseNet	2016.08	Gao Huang et al.	
MobileNet V1	2017.04	Andrew G. Howard et al.	Google Inc.
MobileNet V2	2018.01	Mark Sandler et al.	Google Inc.
MobileNet V3	2019.05	Andrew Howard et al.	Google AI, Google Brain
ShuffleNet V1	2017.07	Xiangyu Zhang et al.	
ShuffleNet V2	2018.07	Xiangyu Zhang et al.	

1. Y. Lecun et al., Gradient-based learning applied to document recognition, LeNet, 1998.
2. Alex Krizhevsky et al., ImageNet Classification with Deep Convolutional Neural Networks, AlexNet, 2012.
3. Karen Simonyan et al., Very Deep Convolutional Networks for Large-Scale Image Recognition, VGG, 2014.
4. Christian Szegedy et al., Going Deeper with Convolutions, Inception Net, 2014, Google Inc.
5. Sergey Ioffe et al., Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, Inception Net V2 , 2015, Google Inc.
6. Christian Szegedy et al., Rethinking the Inception Architecture for Computer Vision, Inception Net V3, 2015, Google Inc.
7. Christian Szegedy et al., Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning, Inception Net V4, 2016, Google Inc.
8. François Chollet, Xception: Deep Learning with Depthwise Separable Convolutions, Xception, 2016, Google Inc.
9. Kaiming He et al., Deep Residual Learning for Image Recognition, ResNet, 2015, Microsoft Research
10. Gao Huang et al., Densely Connected Convolutional Networks, DenseNet, 2016

11. Andrew G. Howard et al., MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications, MobileNet V1, 2017, Google Inc.

12. Mark Sandler et al., MobileNetV2: Inverted Residuals and Linear Bottlenecks, MobileNet V2, 2018, Google Inc.

13. Andrew Howard et al., Searching for MobileNetV3, MobileNet V3, 2019, Google AI and Google Brain.

14. Xiangyu Zhang et al., ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices, ShuffleNet V1, 2017.

15. Xiangyu Zhang et al., ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design, ShuffleNet V2, 2018.

5 附件 1

下為 FC4-2 則在 Batch Size 為 30 時做到 0.961 的實際測試狀況。

```
Epoch:  0 | train loss: 2.3179 | test accuracy: 0.108
Epoch:  0 | train loss: 1.7215 | test accuracy: 0.438
Epoch:  0 | train loss: 0.9065 | test accuracy: 0.595
Epoch:  0 | train loss: 0.7863 | test accuracy: 0.695
Epoch:  0 | train loss: 0.7010 | test accuracy: 0.753
Epoch:  0 | train loss: 0.2025 | test accuracy: 0.772
Epoch:  0 | train loss: 0.5316 | test accuracy: 0.784
Epoch:  0 | train loss: 0.4025 | test accuracy: 0.808
Epoch:  0 | train loss: 0.4179 | test accuracy: 0.840
Epoch:  0 | train loss: 0.2629 | test accuracy: 0.851
Epoch:  0 | train loss: 0.5766 | test accuracy: 0.885
Epoch:  0 | train loss: 0.6210 | test accuracy: 0.878
Epoch:  0 | train loss: 0.3336 | test accuracy: 0.901
Epoch:  0 | train loss: 0.1728 | test accuracy: 0.917
Epoch:  0 | train loss: 0.2423 | test accuracy: 0.919
Epoch:  0 | train loss: 0.0907 | test accuracy: 0.932
Epoch:  0 | train loss: 0.1517 | test accuracy: 0.934
Epoch:  0 | train loss: 0.5776 | test accuracy: 0.928
Epoch:  0 | train loss: 0.2446 | test accuracy: 0.924
Epoch:  0 | train loss: 0.1774 | test accuracy: 0.938
Epoch:  0 | train loss: 0.1122 | test accuracy: 0.939
Epoch:  0 | train loss: 0.5981 | test accuracy: 0.932
Epoch:  0 | train loss: 0.0891 | test accuracy: 0.939
Epoch:  0 | train loss: 0.2379 | test accuracy: 0.941
Epoch:  0 | train loss: 0.1671 | test accuracy: 0.933
Epoch:  0 | train loss: 0.1797 | test accuracy: 0.945
Epoch:  0 | train loss: 0.2858 | test accuracy: 0.941
Epoch:  0 | train loss: 0.0684 | test accuracy: 0.942
Epoch:  0 | train loss: 0.1976 | test accuracy: 0.945
Epoch:  0 | train loss: 0.2697 | test accuracy: 0.941
Epoch:  0 | train loss: 0.1319 | test accuracy: 0.938
Epoch:  0 | train loss: 0.2735 | test accuracy: 0.947
Epoch:  0 | train loss: 0.2526 | test accuracy: 0.928
Epoch:  0 | train loss: 0.0490 | test accuracy: 0.934
Epoch:  1 | train loss: 0.0266 | test accuracy: 0.945
Epoch:  1 | train loss: 0.3994 | test accuracy: 0.949
Epoch:  1 | train loss: 0.4008 | test accuracy: 0.948
Epoch:  1 | train loss: 0.1298 | test accuracy: 0.954
```

```
Epoch:  1 | train loss: 0.1138 | test accuracy: 0.946
Epoch:  1 | train loss: 0.1101 | test accuracy: 0.953
Epoch:  1 | train loss: 0.0453 | test accuracy: 0.949
Epoch:  1 | train loss: 0.0528 | test accuracy: 0.955
Epoch:  1 | train loss: 0.0294 | test accuracy: 0.950
Epoch:  1 | train loss: 0.0197 | test accuracy: 0.949
Epoch:  1 | train loss: 0.1641 | test accuracy: 0.946
Epoch:  1 | train loss: 0.1442 | test accuracy: 0.937
Epoch:  1 | train loss: 0.2131 | test accuracy: 0.951
Epoch:  1 | train loss: 0.0344 | test accuracy: 0.950
Epoch:  1 | train loss: 0.1194 | test accuracy: 0.955
Epoch:  1 | train loss: 0.0861 | test accuracy: 0.952
Epoch:  1 | train loss: 0.6072 | test accuracy: 0.957
Epoch:  1 | train loss: 0.1805 | test accuracy: 0.952
Epoch:  1 | train loss: 0.0142 | test accuracy: 0.956
Epoch:  1 | train loss: 0.0102 | test accuracy: 0.953
Epoch:  1 | train loss: 0.3421 | test accuracy: 0.955
Epoch:  1 | train loss: 0.0862 | test accuracy: 0.952
Epoch:  1 | train loss: 0.3688 | test accuracy: 0.956
Epoch:  1 | train loss: 0.0111 | test accuracy: 0.954
Epoch:  1 | train loss: 0.3614 | test accuracy: 0.955
Epoch:  1 | train loss: 0.0869 | test accuracy: 0.952
Epoch:  1 | train loss: 0.0213 | test accuracy: 0.956
Epoch:  1 | train loss: 0.2109 | test accuracy: 0.962
Epoch:  1 | train loss: 0.2889 | test accuracy: 0.951
Epoch:  1 | train loss: 0.0637 | test accuracy: 0.962
Epoch:  1 | train loss: 0.0596 | test accuracy: 0.959
Epoch:  1 | train loss: 0.0034 | test accuracy: 0.965
Epoch:  1 | train loss: 0.0190 | test accuracy: 0.963
Epoch:  1 | train loss: 0.0153 | test accuracy: 0.961
tensor([7, 2, 1, 0, 4, 1, 4, 9, 6, 9]) prediction number
tensor([7, 2, 1, 0, 4, 1, 4, 9, 5, 9]) real number
```

6 附件 2

下為 FC4-1 在 Batch Size 為 50 時有 0.954 的實際測試狀況。

```
Epoch:  0 | train loss: 2.3079 | test accuracy: 0.105
Epoch:  0 | train loss: 1.2954 | test accuracy: 0.518
Epoch:  0 | train loss: 0.8506 | test accuracy: 0.718
Epoch:  0 | train loss: 0.4927 | test accuracy: 0.789
Epoch:  0 | train loss: 0.5421 | test accuracy: 0.832
Epoch:  0 | train loss: 0.3407 | test accuracy: 0.846
Epoch:  0 | train loss: 0.5657 | test accuracy: 0.848
Epoch:  0 | train loss: 0.3150 | test accuracy: 0.881
Epoch:  0 | train loss: 0.3149 | test accuracy: 0.882
Epoch:  0 | train loss: 0.7052 | test accuracy: 0.892
Epoch:  0 | train loss: 0.2937 | test accuracy: 0.905
Epoch:  0 | train loss: 0.3780 | test accuracy: 0.909
Epoch:  0 | train loss: 0.1319 | test accuracy: 0.919
Epoch:  0 | train loss: 0.2125 | test accuracy: 0.919
Epoch:  0 | train loss: 0.4903 | test accuracy: 0.915
Epoch:  0 | train loss: 0.0368 | test accuracy: 0.925
Epoch:  0 | train loss: 0.2886 | test accuracy: 0.925
Epoch:  0 | train loss: 0.3133 | test accuracy: 0.930
Epoch:  0 | train loss: 0.1374 | test accuracy: 0.927
Epoch:  0 | train loss: 0.2062 | test accuracy: 0.939
Epoch:  1 | train loss: 0.1921 | test accuracy: 0.937
Epoch:  1 | train loss: 0.2336 | test accuracy: 0.937
Epoch:  1 | train loss: 0.2420 | test accuracy: 0.937
Epoch:  1 | train loss: 0.0554 | test accuracy: 0.942
Epoch:  1 | train loss: 0.1244 | test accuracy: 0.938
Epoch:  1 | train loss: 0.0200 | test accuracy: 0.937
Epoch:  1 | train loss: 0.1651 | test accuracy: 0.947
Epoch:  1 | train loss: 0.1534 | test accuracy: 0.942
Epoch:  1 | train loss: 0.1084 | test accuracy: 0.947
Epoch:  1 | train loss: 0.1853 | test accuracy: 0.943
Epoch:  1 | train loss: 0.0954 | test accuracy: 0.951
Epoch:  1 | train loss: 0.2550 | test accuracy: 0.951
Epoch:  1 | train loss: 0.0765 | test accuracy: 0.946
Epoch:  1 | train loss: 0.3135 | test accuracy: 0.948
Epoch:  1 | train loss: 0.0769 | test accuracy: 0.948
Epoch:  1 | train loss: 0.2409 | test accuracy: 0.948
Epoch:  1 | train loss: 0.0431 | test accuracy: 0.952
Epoch:  1 | train loss: 0.0727 | test accuracy: 0.956
```

```
Epoch:  1 | train loss: 0.0572 | test accuracy: 0.948
Epoch:  1 | train loss: 0.1782 | test accuracy: 0.954
tensor([7, 2, 1, 0, 4, 1, 4, 9, 5, 9]) prediction number
tensor([7, 2, 1, 0, 4, 1, 4, 9, 5, 9]) real number
```