T4_MNIST_Nihal_JG

March 31, 2020

This aim of this Problem Statement is to introduce Deep Learning, which is one of the most used techniques for Computer Vision Application. Before beginning go thorugh the following articles: 1. https://www.nature.com/articles/nature14539 : It is fairly easy to follow and provides an excellent overview of the field. (You will need to login with smail to download the pdf.)

2. http://deeplearning.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/ : A short introduction to Multi-layer perceptrons.

3. https://cs231n.github.io/convolutional-networks/ : Introduction to Convolutional Neural Networks. CNNs are generally used for computer vision problems.

This introduction should be sufficient to get you started with this problem statement. If after going through the PS, you guys are interested in further exploring the field, I would suggest the following resources: 1. http://cs231n.stanford.edu/ : Online course by Stanford. 2. http://introtodeeplearning.com/: Video Lectures from MIT 3. https://www.deeplearningbook.org/: Most popular book on Deep Learning

The aim of this get you all familiarized with Deep Learning in PyTorch, a very popular Deep Learning Library (or in general GPU computation library).

Some of the preprocessing work has been done, and you guys are expected to fill-in code where you are asked to.

In this notebook we will be doing the following tasks: 1. Define a small convolutional neural network and train in on MNIST digit dataset (as a classification task). 2. (Bonus) Take the trained network, freeze all of its weights and learn the image which gives the output of a particular digit. (This will be defined better, when we reach that task). 3. (Bonus) Try the same on a different dataset (like MNIST fashion dataset). 4. (Bonus) Try to formulate the MNIST classification problem as a regression problem instead. (i.e. Have 1 output unit which outputs a floating point value and you round it off to obtain the digit.)

Before diving into the code ensure that you copy the notebook to your drive (See the option in File Tab) and that the Runtime Type is set to GPU (Runtime tab -> Change runtime type). To see the importance of GPU in deep learning see this short article.

The following cell imports all the necessary packages. We will be using : 1. Several Modules of PyTorch. This will be used to do all the processing and importing the dataset etc. Read this and this article to get started. Here are the official docs.

2. NumPy is likely the most popular Matrix manipulation and linear algebra library available for Python. Unfortunately it does not support processing on GPU and therefore we will not be using it much here (Only as a support library for matplotlib). Here is a quick tutorial if you are interested.

3. Matplotlib is the most popular library for generating plots in Python. We will be using it for printing images.

[1]:
```
import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data
from torch.autograd import Variable


import numpy as np

from matplotlib import pyplot as plt

#This to ensure that the default device for processing PyTorch tensors is GPU␣
 ↪instead of a CPU.
device = 'cuda' #Change this to 'cpu' to run on cpu.
```

Here we will download the MNIST digit dataset and convert it into PyTorch dataset object.

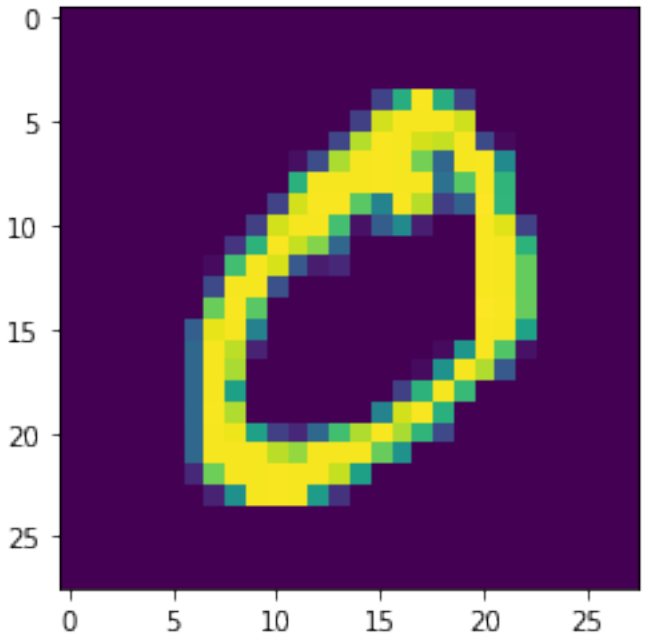
torchvision module of PyTorch provides several Computer Vision specific functionalities one of which is easy importing of several popular datasets.

[2]:
```
#We will load the MNIST dataset here.

#This creates a transform object which will be passed to the later functions.␣
 ↪It converts the training image from PIL Image objects to PyTorch image␣
 ↪objects.
transform = torchvision.transforms.Compose([torchvision.transforms.ToTensor()])

train_data = torchvision.datasets.MNIST(root = './data', train = True, download␣
 ↪= True, transform = transform )

test_data = torchvision.datasets.MNIST(root = './data',train = False ,download␣
 ↪= True,  transform = transform)



print(train_data)
print(test_data)

plt.imshow(np.asarray(train_data[1][0].reshape(28,28)))
```

```
Dataset MNIST
    Number of datapoints: 60000
    Root location: ./data
    Split: Train
```

```
    StandardTransform
Transform: Compose(
               ToTensor()
           )
Dataset MNIST
    Number of datapoints: 10000
    Root location: ./data
    Split: Test
    StandardTransform
Transform: Compose(
               ToTensor()
           )
```

[2]: `<matplotlib.image.AxesImage at 0x7f6f0d3d4d30>`



The following cell converts the DataSet objects to DataLoader objects. DataLoader objects in PyTorch provide several useful functionalities, such as automatic batching, parallel data processing on the CPU, etc. Therefore it is generally advisable to use dataloader instead of manually handling data.

Read more about PyTorch DataSet and DataLoader here

[3]:
```
#BATCHs are only used for training, while testing we conventionally use a batch␣
↪size of 1.
BATCH_SIZE = 32
train_loader = torch.utils.data.DataLoader (train_data, batch_size =␣
↪BATCH_SIZE, )
```

```
test_loader = torch.utils.data.DataLoader (test_data, batch_size = 1, )

print(train_loader)
```

```
<torch.utils.data.dataloader.DataLoader object at 0x7f6f0cce9f60>
```

TASK 1

We will define our network in the following cell. There are several ways of doing this in PyTorch, but in this example we will do it by subclassing nn.module. You should be able to do this after reading the tutorials given at the top of the page. This is extra reading to help you out. (Do remember that we are dealing with 1 channel B/W images not 3 channels colour images.)

Here are official docs of nn module to help you out.

Our network will have 4 convolutional layers followed by 2 fully-connected layers. (You free to change make slight changes to this but don't make the network too deep in the final submission).

Don't forget to have ReLU activations and maxpooling layers.

Experiment with different kernal sizes, width and sizes of hidden fully connected units.

[4]:
```
## Complete the functions __init__ and forward below

class MyConvNet(nn.Module):
  def __init__(self):
    """
      Initialize all the weights for each layer you require.

    """

    super(MyConvNet, self).__init__()

    # __begin

    self.conv1 = nn.Conv2d(1, 16, kernel_size=5)
    self.conv2 = nn.Conv2d(16, 32, kernel_size=5)
    self.conv3 = nn.Conv2d(32,64, kernel_size=3)
    self.conv4 = nn.Conv2d(64,64, kernel_size=3)

    self.fc1 = nn.Linear(8 * 8 * 4, 256)
    self.fc2 = nn.Linear(256, 10)
    #__end
  def forward (self, x):
    """
    Define a forward pass for the given network. x is a 4 dimentional input␣
↪tensor.
    Input: x.size() = [BATCH_SIZE, 1, 28, 28]
```

```
    Return a 2-dimentional, 10-unit output tensor representing the␣
↪probabiCalculated padded input size per channel: (4 x 4). Kernel size: (5 x␣
↪5). Kernel size can't be greater than actual input sizelity of each class.␣
↪Use a softmax output unit.
    Output: x.size() = [BATCH_SIZE, 10]
    """
    # Put your code here:
    # __begin

    #print(x.shape)
    x = F.relu(self.conv1(x))

    #x = F.dropout(x, p=0.5, training=self.training)
    x = F.relu(F.max_pool2d(self.conv2(x), 2))
    #print(x.shape)
    #x = F.dropout(x, p=0.5, training=self.training)

    x = F.relu(F.max_pool2d(self.conv3(x),2))
    #print(x.shape)
    #x = F.dropout(x, p=0.5, training=self.training)

    x = F.relu(F.max_pool2d(self.conv4(x),1))
    #print(x.shape)
    #x = F.dropout(x, p=0.5, training=self.training)

    x = x.view(-1,8*8*4 )
    #print(x.shape)
    x = F.relu(self.fc1(x))
    #print(x.shape)
    #x = F.dropout(x, training=self.training)

    x = self.fc2(x)
    #print(x.shape)
    #__end
    #Return statement uses softmax, so after the final fully connected layer␣
↪store the value in x (without using ReLU or any other activation).
    return F.softmax(x, dim = 1)
```

The following cell will instantiate the the network we defined above. Check the size of the output tensor (it should be [BATCH_SIZE, 10]) and see if the values in the output tensor printed below are numbers between 0 and 1. (Infact most values should be close to 0.1. Why?)

[5]:
```
#Setting few things up.

net = MyConvNet()
net.cuda()
net.to(device)
```

```
print(net)
tld = iter(train_loader)
im = next(tld)[0].to(device)
print ('Size of the output tensor:' ,net.forward(im).size())
print (net.forward(im))
```

```
MyConvNet(
  (conv1): Conv2d(1, 16, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(5, 5), stride=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
  (conv4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
  (fc1): Linear(in_features=256, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=10, bias=True)
)
Size of the output tensor: torch.Size([32, 10])
tensor([[0.0964, 0.0995, 0.1016, 0.1015, 0.0954, 0.0982, 0.0980, 0.1004, 0.1060,
         0.1029],
        [0.0967, 0.0992, 0.1017, 0.1016, 0.0954, 0.0983, 0.0975, 0.1005, 0.1061,
         0.1030],
        [0.0968, 0.0994, 0.1016, 0.1014, 0.0952, 0.0980, 0.0979, 0.1005, 0.1064,
         0.1028],
        [0.0965, 0.0993, 0.1018, 0.1016, 0.0956, 0.0976, 0.0979, 0.1006, 0.1063,
         0.1028],
        [0.0966, 0.0995, 0.1019, 0.1012, 0.0954, 0.0981, 0.0980, 0.1002, 0.1064,
         0.1027],
        [0.0967, 0.0991, 0.1016, 0.1012, 0.0957, 0.0977, 0.0979, 0.1007, 0.1065,
         0.1029],
        [0.0965, 0.0999, 0.1017, 0.1011, 0.0956, 0.0983, 0.0978, 0.1004, 0.1057,
         0.1031],
        [0.0966, 0.0994, 0.1014, 0.1018, 0.0954, 0.0975, 0.0978, 0.1007, 0.1064,
         0.1029],
        [0.0967, 0.1000, 0.1020, 0.1010, 0.0953, 0.0984, 0.0981, 0.1003, 0.1055,
         0.1029],
        [0.0968, 0.0990, 0.1016, 0.1016, 0.0956, 0.0976, 0.0980, 0.1003, 0.1064,
         0.1031],
        [0.0964, 0.0994, 0.1017, 0.1013, 0.0955, 0.0978, 0.0978, 0.1010, 0.1063,
         0.1027],
        [0.0966, 0.0995, 0.1015, 0.1015, 0.0957, 0.0979, 0.0979, 0.1005, 0.1063,
         0.1026],
        [0.0964, 0.0994, 0.1013, 0.1016, 0.0955, 0.0977, 0.0980, 0.1007, 0.1063,
         0.1030],
        [0.0965, 0.0991, 0.1017, 0.1013, 0.0958, 0.0979, 0.0978, 0.1004, 0.1067,
         0.1029],
        [0.0965, 0.1000, 0.1018, 0.1011, 0.0956, 0.0982, 0.0980, 0.1005, 0.1055,
         0.1029],
        [0.0966, 0.0996, 0.1017, 0.1010, 0.0956, 0.0981, 0.0980, 0.1005, 0.1063,
         0.1026],
```

```
[0.0968, 0.0994, 0.1016, 0.1015, 0.0957, 0.0974, 0.0977, 0.1006, 0.1067,
 0.1024],
[0.0962, 0.0996, 0.1018, 0.1010, 0.0958, 0.0979, 0.0978, 0.1009, 0.1062,
 0.1027],
[0.0970, 0.0992, 0.1017, 0.1016, 0.0953, 0.0982, 0.0978, 0.1002, 0.1061,
 0.1029],
[0.0963, 0.0997, 0.1019, 0.1014, 0.0954, 0.0979, 0.0982, 0.1006, 0.1062,
 0.1025],
[0.0973, 0.0990, 0.1017, 0.1015, 0.0954, 0.0975, 0.0979, 0.1001, 0.1068,
 0.1028],
[0.0965, 0.0990, 0.1017, 0.1014, 0.0955, 0.0982, 0.0976, 0.1006, 0.1065,
 0.1030],
[0.0965, 0.0994, 0.1020, 0.1015, 0.0956, 0.0977, 0.0979, 0.1004, 0.1061,
 0.1030],
[0.0967, 0.0993, 0.1018, 0.1016, 0.0955, 0.0976, 0.0980, 0.1006, 0.1062,
 0.1027],
[0.0968, 0.0995, 0.1018, 0.1013, 0.0954, 0.0983, 0.0979, 0.1003, 0.1064,
 0.1025],
[0.0965, 0.0993, 0.1014, 0.1016, 0.0957, 0.0973, 0.0975, 0.1004, 0.1068,
 0.1033],
[0.0962, 0.0994, 0.1020, 0.1015, 0.0952, 0.0982, 0.0981, 0.1003, 0.1065,
 0.1026],
[0.0963, 0.0992, 0.1015, 0.1014, 0.0956, 0.0974, 0.0978, 0.1010, 0.1066,
 0.1032],
[0.0962, 0.0998, 0.1016, 0.1011, 0.0959, 0.0980, 0.0976, 0.1007, 0.1062,
 0.1029],
[0.0965, 0.0994, 0.1017, 0.1014, 0.0957, 0.0980, 0.0980, 0.1007, 0.1061,
 0.1027],
[0.0960, 0.0994, 0.1016, 0.1011, 0.0957, 0.0980, 0.0983, 0.1004, 0.1067,
 0.1027],
[0.0963, 0.0995, 0.1016, 0.1014, 0.0960, 0.0975, 0.0978, 0.1007, 0.1063,
 0.1029]], device='cuda:0', grad_fn=<SoftmaxBackward>)
```

The following cell will define the training and the testing loop.

I have already defined the training loop to give you guys some idea. Try changing the learning rates, EPOCHS, weight_decay, and optimizers etc. to see how the learning process changes.

Your task is to define the test function which prints the test accuracy, given the model and test_loader. Do remember that while testing we use a single image (batch of size 1).

First understand the train function properly. Then attempt to write the test function.

(HINT: you don't need optimizer (torch.optim) and loss fucntion (nn.CrossEntropyLoss) for testing).

[6]:
```
def train(model, train_loader, EPOCHS = 6, lossF = None):

  if lossF == None:
```

```
   lossF = nn.CrossEntropyLoss() #Cross entropy loss is popularly used for␣
↪classification tasks.

 ## Adam is a very popular choice of optimization algorithms.
 ## It is not very sensitive to hyperparameters and therefore it becomes a␣
↪natural choice in quick experiments.
 optim = torch.optim.Adam (model.parameters(), lr = 4e-4, weight_decay=1e-3)

 model.train() #Changes the model to train mode. All the require_grad are set␣
↪to true.

 for epoch in range(EPOCHS):
   correct = 0
   for batch_idx, (X_batch, y_batch) in enumerate(train_loader):

     #Move data to device
     var_X_batch = Variable(X_batch).to(device)
     var_y_batch = Variable(y_batch).to(device)
     # print(var_X_batch.size())

     ## Forward Pass!
     output = model(var_X_batch)

     #print(output)
     #print(output.size(), var_y_batch.size())

     ## Calculate the loss incurred
     loss = lossF (output, var_y_batch)

     ## BackProp: Computes all gradients.
     loss.backward()

     ## Gradient Descent Step (Adam)
     optim.step()
     optim.zero_grad() # This is important because PyTorch keeps on adding to␣
↪the original value of gradient.

     ## Gets the predictions. From probablities (the digit with highest␣
↪probablity is the prediction)
     predicted = torch.max(output.data, axis = 1).indices

     # print(predicted)

     ## Calculates the number of correct predictions in a batch
     correct += (predicted == var_y_batch).sum()

     if (batch_idx % 200) == 0:
```

```
            print('Epoch : {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}\t Accuracy:{:.3f}%'.
↪format(
                    epoch, batch_idx*len(X_batch), len(train_loader.dataset),␣
↪100.*batch_idx / len(train_loader), loss.item(), float(correct*100) /␣
↪float(BATCH_SIZE*(batch_idx+1))))



def test(model, test_loader):
    """
    Change the value of correct to contain the number of correctly classified␣
↪test examples out of the 10000 example in test_loader.
    """
    correct = 0
    # Put your code here:
    # __begin
    for test_imgs, test_labels in test_loader:
        #print(test_imgs.shape)
        test_imgs = Variable(test_imgs).float()
        test_imgs = Variable(test_imgs).to(device)
        test_labels = Variable(test_labels).to(device)
        output = model(test_imgs)
        predicted = torch.max(output,1)[1]
        correct += (predicted == test_labels).sum()

    #__end
    print("Test accuracy:{:.3f}% ".format( float(correct * 100) /␣
↪(len(test_loader))))
```

Running the following cell will start training of the "net".

Note: Any decent bugfree conv-net should easily reach an accuracy of atleast 96%. If you are unable to do so, try improving the network architecture and/or try different training rate or longer training (higher EPOCHS)

[7]:
```
net = net.cuda()
train (net, train_loader, EPOCHS = 20)
```

```
Epoch : 0 [0/60000 (0%)]        Loss: 2.302599   Accuracy:6.250%
Epoch : 0 [6400/60000 (11%)]    Loss: 1.784655   Accuracy:53.887%
Epoch : 0 [12800/60000 (21%)]   Loss: 1.744274   Accuracy:62.851%
Epoch : 0 [19200/60000 (32%)]   Loss: 1.681714   Accuracy:68.480%
Epoch : 0 [25600/60000 (43%)]   Loss: 1.576507   Accuracy:73.490%
Epoch : 0 [32000/60000 (53%)]   Loss: 1.494225   Accuracy:77.123%
Epoch : 0 [38400/60000 (64%)]   Loss: 1.506610   Accuracy:79.874%
Epoch : 0 [44800/60000 (75%)]   Loss: 1.487906   Accuracy:81.883%
Epoch : 0 [51200/60000 (85%)]   Loss: 1.527602   Accuracy:83.403%
Epoch : 0 [57600/60000 (96%)]   Loss: 1.514520   Accuracy:84.708%
```

```
Epoch : 1 [0/60000 (0%)]        Loss: 1.488366   Accuracy:100.000%
Epoch : 1 [6400/60000 (11%)]    Loss: 1.502373   Accuracy:96.160%
Epoch : 1 [12800/60000 (21%)]   Loss: 1.475746   Accuracy:95.776%
Epoch : 1 [19200/60000 (32%)]   Loss: 1.479310   Accuracy:95.726%
Epoch : 1 [25600/60000 (43%)]   Loss: 1.464445   Accuracy:95.876%
Epoch : 1 [32000/60000 (53%)]   Loss: 1.498445   Accuracy:95.970%
Epoch : 1 [38400/60000 (64%)]   Loss: 1.492334   Accuracy:96.074%
Epoch : 1 [44800/60000 (75%)]   Loss: 1.491384   Accuracy:96.117%
Epoch : 1 [51200/60000 (85%)]   Loss: 1.490910   Accuracy:96.110%
Epoch : 1 [57600/60000 (96%)]   Loss: 1.500375   Accuracy:96.167%
Epoch : 2 [0/60000 (0%)]        Loss: 1.471059   Accuracy:100.000%
Epoch : 2 [6400/60000 (11%)]    Loss: 1.462476   Accuracy:97.248%
Epoch : 2 [12800/60000 (21%)]   Loss: 1.476719   Accuracy:96.820%
Epoch : 2 [19200/60000 (32%)]   Loss: 1.479044   Accuracy:96.823%
Epoch : 2 [25600/60000 (43%)]   Loss: 1.482142   Accuracy:97.000%
Epoch : 2 [32000/60000 (53%)]   Loss: 1.493994   Accuracy:96.994%
Epoch : 2 [38400/60000 (64%)]   Loss: 1.521864   Accuracy:97.042%
Epoch : 2 [44800/60000 (75%)]   Loss: 1.476910   Accuracy:97.047%
Epoch : 2 [51200/60000 (85%)]   Loss: 1.492091   Accuracy:97.016%
Epoch : 2 [57600/60000 (96%)]   Loss: 1.505467   Accuracy:97.050%
Epoch : 3 [0/60000 (0%)]        Loss: 1.461615   Accuracy:100.000%
Epoch : 3 [6400/60000 (11%)]    Loss: 1.465881   Accuracy:97.404%
Epoch : 3 [12800/60000 (21%)]   Loss: 1.494033   Accuracy:97.195%
Epoch : 3 [19200/60000 (32%)]   Loss: 1.463103   Accuracy:97.270%
Epoch : 3 [25600/60000 (43%)]   Loss: 1.477686   Accuracy:97.413%
Epoch : 3 [32000/60000 (53%)]   Loss: 1.462061   Accuracy:97.418%
Epoch : 3 [38400/60000 (64%)]   Loss: 1.513970   Accuracy:97.440%
Epoch : 3 [44800/60000 (75%)]   Loss: 1.492633   Accuracy:97.455%
Epoch : 3 [51200/60000 (85%)]   Loss: 1.489715   Accuracy:97.422%
Epoch : 3 [57600/60000 (96%)]   Loss: 1.497450   Accuracy:97.399%
Epoch : 4 [0/60000 (0%)]        Loss: 1.461385   Accuracy:100.000%
Epoch : 4 [6400/60000 (11%)]    Loss: 1.490536   Accuracy:97.761%
Epoch : 4 [12800/60000 (21%)]   Loss: 1.529633   Accuracy:97.576%
Epoch : 4 [19200/60000 (32%)]   Loss: 1.462422   Accuracy:97.702%
Epoch : 4 [25600/60000 (43%)]   Loss: 1.467066   Accuracy:97.772%
Epoch : 4 [32000/60000 (53%)]   Loss: 1.472054   Accuracy:97.727%
Epoch : 4 [38400/60000 (64%)]   Loss: 1.491543   Accuracy:97.741%
Epoch : 4 [44800/60000 (75%)]   Loss: 1.493542   Accuracy:97.743%
Epoch : 4 [51200/60000 (85%)]   Loss: 1.492354   Accuracy:97.689%
Epoch : 4 [57600/60000 (96%)]   Loss: 1.492149   Accuracy:97.701%
Epoch : 5 [0/60000 (0%)]        Loss: 1.463364   Accuracy:100.000%
Epoch : 5 [6400/60000 (11%)]    Loss: 1.464313   Accuracy:98.010%
Epoch : 5 [12800/60000 (21%)]   Loss: 1.502812   Accuracy:97.857%
Epoch : 5 [19200/60000 (32%)]   Loss: 1.466466   Accuracy:97.837%
Epoch : 5 [25600/60000 (43%)]   Loss: 1.468061   Accuracy:97.885%
Epoch : 5 [32000/60000 (53%)]   Loss: 1.479162   Accuracy:97.890%
Epoch : 5 [38400/60000 (64%)]   Loss: 1.491776   Accuracy:97.926%
Epoch : 5 [44800/60000 (75%)]   Loss: 1.463630   Accuracy:97.952%
```

```
Epoch : 5 [51200/60000 (85%)]   Loss: 1.489880   Accuracy:97.908%
Epoch : 5 [57600/60000 (96%)]   Loss: 1.488033   Accuracy:97.928%
Epoch : 6 [0/60000 (0%)]        Loss: 1.462193   Accuracy:100.000%
Epoch : 6 [6400/60000 (11%)]    Loss: 1.466863   Accuracy:97.839%
Epoch : 6 [12800/60000 (21%)]   Loss: 1.483182   Accuracy:97.795%
Epoch : 6 [19200/60000 (32%)]   Loss: 1.473910   Accuracy:97.910%
Epoch : 6 [25600/60000 (43%)]   Loss: 1.464815   Accuracy:97.956%
Epoch : 6 [32000/60000 (53%)]   Loss: 1.487100   Accuracy:98.002%
Epoch : 6 [38400/60000 (64%)]   Loss: 1.491885   Accuracy:98.017%
Epoch : 6 [44800/60000 (75%)]   Loss: 1.466469   Accuracy:98.035%
Epoch : 6 [51200/60000 (85%)]   Loss: 1.491591   Accuracy:97.982%
Epoch : 6 [57600/60000 (96%)]   Loss: 1.483985   Accuracy:98.015%
Epoch : 7 [0/60000 (0%)]        Loss: 1.461212   Accuracy:100.000%
Epoch : 7 [6400/60000 (11%)]    Loss: 1.464385   Accuracy:97.963%
Epoch : 7 [12800/60000 (21%)]   Loss: 1.477983   Accuracy:98.021%
Epoch : 7 [19200/60000 (32%)]   Loss: 1.469435   Accuracy:98.102%
Epoch : 7 [25600/60000 (43%)]   Loss: 1.465861   Accuracy:98.186%
Epoch : 7 [32000/60000 (53%)]   Loss: 1.498821   Accuracy:98.186%
Epoch : 7 [38400/60000 (64%)]   Loss: 1.492798   Accuracy:98.189%
Epoch : 7 [44800/60000 (75%)]   Loss: 1.464990   Accuracy:98.178%
Epoch : 7 [51200/60000 (85%)]   Loss: 1.491070   Accuracy:98.113%
Epoch : 7 [57600/60000 (96%)]   Loss: 1.492181   Accuracy:98.140%
Epoch : 8 [0/60000 (0%)]        Loss: 1.461977   Accuracy:100.000%
Epoch : 8 [6400/60000 (11%)]    Loss: 1.462248   Accuracy:98.212%
Epoch : 8 [12800/60000 (21%)]   Loss: 1.489834   Accuracy:98.130%
Epoch : 8 [19200/60000 (32%)]   Loss: 1.465303   Accuracy:98.201%
Epoch : 8 [25600/60000 (43%)]   Loss: 1.462251   Accuracy:98.283%
Epoch : 8 [32000/60000 (53%)]   Loss: 1.490436   Accuracy:98.308%
Epoch : 8 [38400/60000 (64%)]   Loss: 1.491498   Accuracy:98.309%
Epoch : 8 [44800/60000 (75%)]   Loss: 1.461868   Accuracy:98.320%
Epoch : 8 [51200/60000 (85%)]   Loss: 1.491032   Accuracy:98.257%
Epoch : 8 [57600/60000 (96%)]   Loss: 1.473637   Accuracy:98.263%
Epoch : 9 [0/60000 (0%)]        Loss: 1.461248   Accuracy:100.000%
Epoch : 9 [6400/60000 (11%)]    Loss: 1.473389   Accuracy:98.321%
Epoch : 9 [12800/60000 (21%)]   Loss: 1.482455   Accuracy:98.262%
Epoch : 9 [19200/60000 (32%)]   Loss: 1.462209   Accuracy:98.305%
Epoch : 9 [25600/60000 (43%)]   Loss: 1.463193   Accuracy:98.389%
Epoch : 9 [32000/60000 (53%)]   Loss: 1.462231   Accuracy:98.367%
Epoch : 9 [38400/60000 (64%)]   Loss: 1.503498   Accuracy:98.361%
Epoch : 9 [44800/60000 (75%)]   Loss: 1.461265   Accuracy:98.378%
Epoch : 9 [51200/60000 (85%)]   Loss: 1.490982   Accuracy:98.317%
Epoch : 9 [57600/60000 (96%)]   Loss: 1.477924   Accuracy:98.327%
Epoch : 10 [0/60000 (0%)]       Loss: 1.461547   Accuracy:100.000%
Epoch : 10 [6400/60000 (11%)]   Loss: 1.479868   Accuracy:98.305%
Epoch : 10 [12800/60000 (21%)]  Loss: 1.498153   Accuracy:98.301%
Epoch : 10 [19200/60000 (32%)]  Loss: 1.464441   Accuracy:98.367%
Epoch : 10 [25600/60000 (43%)]  Loss: 1.467145   Accuracy:98.404%
Epoch : 10 [32000/60000 (53%)]  Loss: 1.484795   Accuracy:98.358%
```

```
Epoch : 10 [38400/60000 (64%)]  Loss: 1.490959   Accuracy:98.371%
Epoch : 10 [44800/60000 (75%)]  Loss: 1.463867   Accuracy:98.387%
Epoch : 10 [51200/60000 (85%)]  Loss: 1.492266   Accuracy:98.333%
Epoch : 10 [57600/60000 (96%)]  Loss: 1.467137   Accuracy:98.367%
Epoch : 11 [0/60000 (0%)]       Loss: 1.461564   Accuracy:100.000%
Epoch : 11 [6400/60000 (11%)]   Loss: 1.462870   Accuracy:98.274%
Epoch : 11 [12800/60000 (21%)]  Loss: 1.483328   Accuracy:98.325%
Epoch : 11 [19200/60000 (32%)]  Loss: 1.462031   Accuracy:98.425%
Epoch : 11 [25600/60000 (43%)]  Loss: 1.462825   Accuracy:98.486%
Epoch : 11 [32000/60000 (53%)]  Loss: 1.463807   Accuracy:98.483%
Epoch : 11 [38400/60000 (64%)]  Loss: 1.491063   Accuracy:98.486%
Epoch : 11 [44800/60000 (75%)]  Loss: 1.471956   Accuracy:98.499%
Epoch : 11 [51200/60000 (85%)]  Loss: 1.467975   Accuracy:98.444%
Epoch : 11 [57600/60000 (96%)]  Loss: 1.470811   Accuracy:98.461%
Epoch : 12 [0/60000 (0%)]       Loss: 1.462924   Accuracy:100.000%
Epoch : 12 [6400/60000 (11%)]   Loss: 1.462150   Accuracy:98.539%
Epoch : 12 [12800/60000 (21%)]  Loss: 1.485432   Accuracy:98.473%
Epoch : 12 [19200/60000 (32%)]  Loss: 1.466822   Accuracy:98.513%
Epoch : 12 [25600/60000 (43%)]  Loss: 1.475937   Accuracy:98.572%
Epoch : 12 [32000/60000 (53%)]  Loss: 1.463308   Accuracy:98.601%
Epoch : 12 [38400/60000 (64%)]  Loss: 1.490987   Accuracy:98.579%
Epoch : 12 [44800/60000 (75%)]  Loss: 1.483012   Accuracy:98.588%
Epoch : 12 [51200/60000 (85%)]  Loss: 1.488576   Accuracy:98.546%
Epoch : 12 [57600/60000 (96%)]  Loss: 1.498243   Accuracy:98.562%
Epoch : 13 [0/60000 (0%)]       Loss: 1.461428   Accuracy:100.000%
Epoch : 13 [6400/60000 (11%)]   Loss: 1.467439   Accuracy:98.445%
Epoch : 13 [12800/60000 (21%)]  Loss: 1.482636   Accuracy:98.340%
Epoch : 13 [19200/60000 (32%)]  Loss: 1.461794   Accuracy:98.419%
Epoch : 13 [25600/60000 (43%)]  Loss: 1.461940   Accuracy:98.537%
Epoch : 13 [32000/60000 (53%)]  Loss: 1.476435   Accuracy:98.536%
Epoch : 13 [38400/60000 (64%)]  Loss: 1.490933   Accuracy:98.564%
Epoch : 13 [44800/60000 (75%)]  Loss: 1.483143   Accuracy:98.550%
Epoch : 13 [51200/60000 (85%)]  Loss: 1.470785   Accuracy:98.546%
Epoch : 13 [57600/60000 (96%)]  Loss: 1.485798   Accuracy:98.569%
Epoch : 14 [0/60000 (0%)]       Loss: 1.461482   Accuracy:100.000%
Epoch : 14 [6400/60000 (11%)]   Loss: 1.464975   Accuracy:98.492%
Epoch : 14 [12800/60000 (21%)]  Loss: 1.473536   Accuracy:98.543%
Epoch : 14 [19200/60000 (32%)]  Loss: 1.462132   Accuracy:98.560%
Epoch : 14 [25600/60000 (43%)]  Loss: 1.464996   Accuracy:98.607%
Epoch : 14 [32000/60000 (53%)]  Loss: 1.463661   Accuracy:98.614%
Epoch : 14 [38400/60000 (64%)]  Loss: 1.491048   Accuracy:98.631%
Epoch : 14 [44800/60000 (75%)]  Loss: 1.464110   Accuracy:98.601%
Epoch : 14 [51200/60000 (85%)]  Loss: 1.486419   Accuracy:98.563%
Epoch : 14 [57600/60000 (96%)]  Loss: 1.481739   Accuracy:98.581%
Epoch : 15 [0/60000 (0%)]       Loss: 1.462240   Accuracy:100.000%
Epoch : 15 [6400/60000 (11%)]   Loss: 1.472030   Accuracy:98.756%
Epoch : 15 [12800/60000 (21%)]  Loss: 1.480627   Accuracy:98.667%
Epoch : 15 [19200/60000 (32%)]  Loss: 1.464678   Accuracy:98.705%
```

```
Epoch : 15 [25600/60000 (43%)]  Loss: 1.464290   Accuracy:98.755%
Epoch : 15 [32000/60000 (53%)]  Loss: 1.464509   Accuracy:98.748%
Epoch : 15 [38400/60000 (64%)]  Loss: 1.491058   Accuracy:98.733%
Epoch : 15 [44800/60000 (75%)]  Loss: 1.465672   Accuracy:98.711%
Epoch : 15 [51200/60000 (85%)]  Loss: 1.479563   Accuracy:98.657%
Epoch : 15 [57600/60000 (96%)]  Loss: 1.466746   Accuracy:98.659%
Epoch : 16 [0/60000 (0%)]       Loss: 1.461531   Accuracy:100.000%
Epoch : 16 [6400/60000 (11%)]   Loss: 1.468590   Accuracy:98.616%
Epoch : 16 [12800/60000 (21%)]  Loss: 1.497316   Accuracy:98.550%
Epoch : 16 [19200/60000 (32%)]  Loss: 1.466259   Accuracy:98.648%
Epoch : 16 [25600/60000 (43%)]  Loss: 1.461793   Accuracy:98.713%
Epoch : 16 [32000/60000 (53%)]  Loss: 1.466365   Accuracy:98.708%
Epoch : 16 [38400/60000 (64%)]  Loss: 1.491099   Accuracy:98.686%
Epoch : 16 [44800/60000 (75%)]  Loss: 1.471007   Accuracy:98.684%
Epoch : 16 [51200/60000 (85%)]  Loss: 1.490944   Accuracy:98.645%
Epoch : 16 [57600/60000 (96%)]  Loss: 1.488891   Accuracy:98.662%
Epoch : 17 [0/60000 (0%)]       Loss: 1.466717   Accuracy:100.000%
Epoch : 17 [6400/60000 (11%)]   Loss: 1.472402   Accuracy:98.741%
Epoch : 17 [12800/60000 (21%)]  Loss: 1.463979   Accuracy:98.488%
Epoch : 17 [19200/60000 (32%)]  Loss: 1.463227   Accuracy:98.601%
Epoch : 17 [25600/60000 (43%)]  Loss: 1.463009   Accuracy:98.674%
Epoch : 17 [32000/60000 (53%)]  Loss: 1.463393   Accuracy:98.689%
Epoch : 17 [38400/60000 (64%)]  Loss: 1.498358   Accuracy:98.678%
Epoch : 17 [44800/60000 (75%)]  Loss: 1.464586   Accuracy:98.695%
Epoch : 17 [51200/60000 (85%)]  Loss: 1.488403   Accuracy:98.657%
Epoch : 17 [57600/60000 (96%)]  Loss: 1.464547   Accuracy:98.681%
Epoch : 18 [0/60000 (0%)]       Loss: 1.461233   Accuracy:100.000%
Epoch : 18 [6400/60000 (11%)]   Loss: 1.462193   Accuracy:98.601%
Epoch : 18 [12800/60000 (21%)]  Loss: 1.472664   Accuracy:98.527%
Epoch : 18 [19200/60000 (32%)]  Loss: 1.465309   Accuracy:98.612%
Epoch : 18 [25600/60000 (43%)]  Loss: 1.462420   Accuracy:98.685%
Epoch : 18 [32000/60000 (53%)]  Loss: 1.462830   Accuracy:98.698%
Epoch : 18 [38400/60000 (64%)]  Loss: 1.490883   Accuracy:98.712%
Epoch : 18 [44800/60000 (75%)]  Loss: 1.469108   Accuracy:98.706%
Epoch : 18 [51200/60000 (85%)]  Loss: 1.472325   Accuracy:98.673%
Epoch : 18 [57600/60000 (96%)]  Loss: 1.466831   Accuracy:98.688%
Epoch : 19 [0/60000 (0%)]       Loss: 1.461330   Accuracy:100.000%
Epoch : 19 [6400/60000 (11%)]   Loss: 1.461505   Accuracy:98.865%
Epoch : 19 [12800/60000 (21%)]  Loss: 1.474587   Accuracy:98.776%
Epoch : 19 [19200/60000 (32%)]  Loss: 1.463233   Accuracy:98.788%
Epoch : 19 [25600/60000 (43%)]  Loss: 1.464973   Accuracy:98.869%
Epoch : 19 [32000/60000 (53%)]  Loss: 1.468030   Accuracy:98.842%
Epoch : 19 [38400/60000 (64%)]  Loss: 1.491192   Accuracy:98.834%
Epoch : 19 [44800/60000 (75%)]  Loss: 1.491296   Accuracy:98.827%
Epoch : 19 [51200/60000 (85%)]  Loss: 1.487845   Accuracy:98.761%
Epoch : 19 [57600/60000 (96%)]  Loss: 1.465441   Accuracy:98.782%
```

Run the test function that you defined above!

Note: You should easily reach 95%.

[8]:
```
test(net, test_loader)
```

```
Test accuracy:98.570%
```

[29]:
```
# My second attempt, has only 2 conv layers and performs nearly as well

class Model2(torch.nn.Module):
    def __init__(self):
        super(Model2, self).__init__()
        self.conv_1 = torch.nn.Conv2d(1, 32, 3, stride=1, padding=1)
        self.conv_2 = torch.nn.Conv2d(32, 64, 3, stride=1, padding=1)
        self.max_pool2d = torch.nn.MaxPool2d(kernel_size=2, stride=2)
        self.linear_1 = torch.nn.Linear(7 * 7 * 64, 128)
        self.linear_2 = torch.nn.Linear(128, 10)
        self.relu = torch.nn.ReLU()

    def forward(self, x):
        x = self.conv_1(x)
        x = self.relu(x)
        x = self.max_pool2d(x)

        x = self.conv_2(x)
        x = self.relu(x)
        x = self.max_pool2d(x)

        x = x.reshape(x.size(0), -1)

        x = self.linear_1(x)
        x = self.relu(x)

        x = self.linear_2(x)

        pred = F.softmax(x, dim=1)

        return pred
```

[30]:
```
net2 = Model2()
net2.cuda()
net2.to(device)
print(net2)
tld = iter(train_loader)
im = next(tld)[0].to(device)
print ('Size of the output tensor:' ,net2.forward(im).size())
print (net2.forward(im))
```

```
Model2(
```

```
  (conv_1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv_2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (max_pool2d): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (linear_1): Linear(in_features=3136, out_features=128, bias=True)
  (linear_2): Linear(in_features=128, out_features=10, bias=True)
  (relu): ReLU()
)
Size of the output tensor: torch.Size([32, 10])
tensor([[0.1015, 0.1008, 0.0984, 0.1059, 0.0940, 0.0907, 0.1061, 0.1047, 0.0968,
         0.1009],
        [0.1011, 0.1032, 0.0976, 0.1044, 0.0927, 0.0920, 0.1068, 0.1061, 0.0957,
         0.1005],
        [0.1014, 0.1045, 0.0941, 0.1026, 0.0948, 0.0934, 0.1050, 0.1046, 0.0967,
         0.1030],
        [0.0984, 0.1037, 0.0992, 0.1032, 0.0936, 0.0926, 0.1045, 0.1049, 0.0971,
         0.1028],
        [0.0985, 0.1020, 0.0979, 0.1049, 0.0943, 0.0920, 0.1073, 0.1050, 0.0967,
         0.1014],
        [0.0988, 0.0998, 0.0983, 0.1056, 0.0948, 0.0916, 0.1064, 0.1061, 0.0975,
         0.1011],
        [0.1014, 0.0988, 0.0990, 0.1045, 0.0926, 0.0924, 0.1066, 0.1064, 0.0960,
         0.1023],
        [0.1006, 0.1002, 0.0983, 0.1053, 0.0938, 0.0916, 0.1063, 0.1072, 0.0962,
         0.1006],
        [0.1014, 0.1011, 0.0993, 0.1041, 0.0918, 0.0917, 0.1064, 0.1042, 0.0969,
         0.1031],
        [0.0993, 0.1024, 0.0984, 0.1040, 0.0943, 0.0935, 0.1052, 0.1046, 0.0965,
         0.1018],
        [0.1011, 0.0995, 0.0986, 0.1045, 0.0936, 0.0928, 0.1062, 0.1071, 0.0952,
         0.1014],
        [0.0988, 0.1035, 0.0986, 0.1035, 0.0943, 0.0929, 0.1045, 0.1057, 0.0961,
         0.1023],
        [0.1009, 0.1014, 0.0963, 0.1053, 0.0952, 0.0919, 0.1056, 0.1066, 0.0951,
         0.1017],
        [0.1000, 0.0999, 0.0982, 0.1050, 0.0934, 0.0949, 0.1060, 0.1072, 0.0940,
         0.1014],
        [0.1016, 0.1013, 0.0991, 0.1037, 0.0921, 0.0913, 0.1060, 0.1043, 0.0972,
         0.1034],
        [0.0999, 0.1021, 0.0979, 0.1041, 0.0935, 0.0923, 0.1073, 0.1049, 0.0967,
         0.1014],
        [0.1008, 0.0992, 0.0981, 0.1043, 0.0940, 0.0930, 0.1071, 0.1054, 0.0965,
         0.1016],
        [0.0999, 0.1013, 0.0996, 0.1049, 0.0939, 0.0919, 0.1071, 0.1049, 0.0958,
         0.1007],
        [0.1005, 0.0983, 0.0990, 0.1057, 0.0943, 0.0933, 0.1054, 0.1062, 0.0952,
         0.1022],
        [0.0996, 0.1013, 0.0991, 0.1053, 0.0934, 0.0922, 0.1059, 0.1051, 0.0962,
```

```
 0.1019],
[0.0971, 0.1032, 0.0963, 0.1034, 0.0956, 0.0947, 0.1052, 0.1063, 0.0954,
 0.1027],
[0.1005, 0.1049, 0.0980, 0.1042, 0.0934, 0.0916, 0.1053, 0.1051, 0.0954,
 0.1016],
[0.0987, 0.1000, 0.0988, 0.1058, 0.0945, 0.0923, 0.1063, 0.1049, 0.0967,
 0.1020],
[0.0989, 0.1033, 0.0990, 0.1032, 0.0938, 0.0926, 0.1048, 0.1051, 0.0967,
 0.1028],
[0.1002, 0.1011, 0.0987, 0.1038, 0.0935, 0.0928, 0.1052, 0.1060, 0.0962,
 0.1024],
[0.0994, 0.1012, 0.1002, 0.1045, 0.0932, 0.0913, 0.1063, 0.1066, 0.0961,
 0.1013],
[0.0994, 0.1011, 0.0971, 0.1056, 0.0945, 0.0924, 0.1063, 0.1047, 0.0965,
 0.1023],
[0.1005, 0.1018, 0.0981, 0.1047, 0.0939, 0.0907, 0.1062, 0.1074, 0.0957,
 0.1009],
[0.1004, 0.1021, 0.0980, 0.1042, 0.0936, 0.0911, 0.1076, 0.1065, 0.0953,
 0.1013],
[0.0991, 0.1024, 0.0970, 0.1043, 0.0948, 0.0925, 0.1063, 0.1050, 0.0964,
 0.1023],
[0.0995, 0.1000, 0.0968, 0.1054, 0.0932, 0.0938, 0.1058, 0.1064, 0.0965,
 0.1027],
[0.0989, 0.1014, 0.0994, 0.1039, 0.0940, 0.0922, 0.1069, 0.1051, 0.0971,
 0.1011]], device='cuda:0', grad_fn=<SoftmaxBackward>)
```

[31]:
```
train (net2, train_loader, EPOCHS = 20)
```

```
Epoch : 0 [0/60000 (0%)]        Loss: 2.302189   Accuracy:0.000%
Epoch : 0 [6400/60000 (11%)]    Loss: 1.818810   Accuracy:55.442%
Epoch : 0 [12800/60000 (21%)]   Loss: 1.626982   Accuracy:61.440%
Epoch : 0 [19200/60000 (32%)]   Loss: 1.670997   Accuracy:67.549%
Epoch : 0 [25600/60000 (43%)]   Loss: 1.559201   Accuracy:72.374%
Epoch : 0 [32000/60000 (53%)]   Loss: 1.578661   Accuracy:76.580%
Epoch : 0 [38400/60000 (64%)]   Loss: 1.516968   Accuracy:79.650%
Epoch : 0 [44800/60000 (75%)]   Loss: 1.502508   Accuracy:81.794%
Epoch : 0 [51200/60000 (85%)]   Loss: 1.515632   Accuracy:83.421%
Epoch : 0 [57600/60000 (96%)]   Loss: 1.511342   Accuracy:84.793%
Epoch : 1 [0/60000 (0%)]        Loss: 1.495278   Accuracy:96.875%
Epoch : 1 [6400/60000 (11%)]    Loss: 1.493335   Accuracy:96.580%
Epoch : 1 [12800/60000 (21%)]   Loss: 1.489852   Accuracy:96.080%
Epoch : 1 [19200/60000 (32%)]   Loss: 1.482573   Accuracy:96.152%
Epoch : 1 [25600/60000 (43%)]   Loss: 1.462774   Accuracy:96.391%
Epoch : 1 [32000/60000 (53%)]   Loss: 1.529930   Accuracy:96.441%
Epoch : 1 [38400/60000 (64%)]   Loss: 1.513393   Accuracy:96.498%
Epoch : 1 [44800/60000 (75%)]   Loss: 1.495590   Accuracy:96.525%
Epoch : 1 [51200/60000 (85%)]   Loss: 1.488971   Accuracy:96.514%
Epoch : 1 [57600/60000 (96%)]   Loss: 1.499197   Accuracy:96.590%
```

```
Epoch : 2 [0/60000 (0%)]        Loss: 1.465693   Accuracy:100.000%
Epoch : 2 [6400/60000 (11%)]    Loss: 1.490964   Accuracy:97.233%
Epoch : 2 [12800/60000 (21%)]   Loss: 1.472825   Accuracy:96.937%
Epoch : 2 [19200/60000 (32%)]   Loss: 1.471897   Accuracy:97.010%
Epoch : 2 [25600/60000 (43%)]   Loss: 1.502457   Accuracy:97.195%
Epoch : 2 [32000/60000 (53%)]   Loss: 1.523909   Accuracy:97.203%
Epoch : 2 [38400/60000 (64%)]   Loss: 1.509187   Accuracy:97.198%
Epoch : 2 [44800/60000 (75%)]   Loss: 1.489578   Accuracy:97.198%
Epoch : 2 [51200/60000 (85%)]   Loss: 1.490946   Accuracy:97.172%
Epoch : 2 [57600/60000 (96%)]   Loss: 1.490612   Accuracy:97.208%
Epoch : 3 [0/60000 (0%)]        Loss: 1.464954   Accuracy:100.000%
Epoch : 3 [6400/60000 (11%)]    Loss: 1.493724   Accuracy:97.404%
Epoch : 3 [12800/60000 (21%)]   Loss: 1.480821   Accuracy:97.296%
Epoch : 3 [19200/60000 (32%)]   Loss: 1.470373   Accuracy:97.333%
Epoch : 3 [25600/60000 (43%)]   Loss: 1.504097   Accuracy:97.538%
Epoch : 3 [32000/60000 (53%)]   Loss: 1.534427   Accuracy:97.549%
Epoch : 3 [38400/60000 (64%)]   Loss: 1.497270   Accuracy:97.552%
Epoch : 3 [44800/60000 (75%)]   Loss: 1.466729   Accuracy:97.535%
Epoch : 3 [51200/60000 (85%)]   Loss: 1.497637   Accuracy:97.470%
Epoch : 3 [57600/60000 (96%)]   Loss: 1.494090   Accuracy:97.500%
Epoch : 4 [0/60000 (0%)]        Loss: 1.463144   Accuracy:100.000%
Epoch : 4 [6400/60000 (11%)]    Loss: 1.496101   Accuracy:97.715%
Epoch : 4 [12800/60000 (21%)]   Loss: 1.488259   Accuracy:97.600%
Epoch : 4 [19200/60000 (32%)]   Loss: 1.474220   Accuracy:97.733%
Epoch : 4 [25600/60000 (43%)]   Loss: 1.487023   Accuracy:97.866%
Epoch : 4 [32000/60000 (53%)]   Loss: 1.529162   Accuracy:97.833%
Epoch : 4 [38400/60000 (64%)]   Loss: 1.495515   Accuracy:97.835%
Epoch : 4 [44800/60000 (75%)]   Loss: 1.473825   Accuracy:97.834%
Epoch : 4 [51200/60000 (85%)]   Loss: 1.498930   Accuracy:97.746%
Epoch : 4 [57600/60000 (96%)]   Loss: 1.482957   Accuracy:97.753%
Epoch : 5 [0/60000 (0%)]        Loss: 1.464787   Accuracy:100.000%
Epoch : 5 [6400/60000 (11%)]    Loss: 1.488373   Accuracy:97.823%
Epoch : 5 [12800/60000 (21%)]   Loss: 1.477630   Accuracy:97.693%
Epoch : 5 [19200/60000 (32%)]   Loss: 1.480989   Accuracy:97.821%
Epoch : 5 [25600/60000 (43%)]   Loss: 1.495892   Accuracy:97.960%
Epoch : 5 [32000/60000 (53%)]   Loss: 1.526620   Accuracy:97.958%
Epoch : 5 [38400/60000 (64%)]   Loss: 1.492473   Accuracy:97.963%
Epoch : 5 [44800/60000 (75%)]   Loss: 1.470722   Accuracy:97.957%
Epoch : 5 [51200/60000 (85%)]   Loss: 1.506606   Accuracy:97.904%
Epoch : 5 [57600/60000 (96%)]   Loss: 1.474086   Accuracy:97.892%
Epoch : 6 [0/60000 (0%)]        Loss: 1.464306   Accuracy:100.000%
Epoch : 6 [6400/60000 (11%)]    Loss: 1.490718   Accuracy:98.010%
Epoch : 6 [12800/60000 (21%)]   Loss: 1.477198   Accuracy:97.911%
Epoch : 6 [19200/60000 (32%)]   Loss: 1.480612   Accuracy:98.003%
Epoch : 6 [25600/60000 (43%)]   Loss: 1.483198   Accuracy:98.159%
Epoch : 6 [32000/60000 (53%)]   Loss: 1.506664   Accuracy:98.149%
Epoch : 6 [38400/60000 (64%)]   Loss: 1.491817   Accuracy:98.124%
Epoch : 6 [44800/60000 (75%)]   Loss: 1.469730   Accuracy:98.111%
```

```
Epoch : 6 [51200/60000 (85%)]   Loss: 1.503405   Accuracy:98.056%
Epoch : 6 [57600/60000 (96%)]   Loss: 1.481660   Accuracy:98.039%
Epoch : 7 [0/60000 (0%)]        Loss: 1.464681   Accuracy:100.000%
Epoch : 7 [6400/60000 (11%)]    Loss: 1.501098   Accuracy:98.057%
Epoch : 7 [12800/60000 (21%)]   Loss: 1.480210   Accuracy:98.013%
Epoch : 7 [19200/60000 (32%)]   Loss: 1.467210   Accuracy:98.066%
Epoch : 7 [25600/60000 (43%)]   Loss: 1.482053   Accuracy:98.194%
Epoch : 7 [32000/60000 (53%)]   Loss: 1.505728   Accuracy:98.199%
Epoch : 7 [38400/60000 (64%)]   Loss: 1.491651   Accuracy:98.207%
Epoch : 7 [44800/60000 (75%)]   Loss: 1.481989   Accuracy:98.207%
Epoch : 7 [51200/60000 (85%)]   Loss: 1.505931   Accuracy:98.173%
Epoch : 7 [57600/60000 (96%)]   Loss: 1.466335   Accuracy:98.142%
Epoch : 8 [0/60000 (0%)]        Loss: 1.464226   Accuracy:100.000%
Epoch : 8 [6400/60000 (11%)]    Loss: 1.490969   Accuracy:98.041%
Epoch : 8 [12800/60000 (21%)]   Loss: 1.486611   Accuracy:97.982%
Epoch : 8 [19200/60000 (32%)]   Loss: 1.469254   Accuracy:98.107%
Epoch : 8 [25600/60000 (43%)]   Loss: 1.485295   Accuracy:98.237%
Epoch : 8 [32000/60000 (53%)]   Loss: 1.506814   Accuracy:98.239%
Epoch : 8 [38400/60000 (64%)]   Loss: 1.491482   Accuracy:98.238%
Epoch : 8 [44800/60000 (75%)]   Loss: 1.480387   Accuracy:98.233%
Epoch : 8 [51200/60000 (85%)]   Loss: 1.510746   Accuracy:98.202%
Epoch : 8 [57600/60000 (96%)]   Loss: 1.470175   Accuracy:98.195%
Epoch : 9 [0/60000 (0%)]        Loss: 1.463546   Accuracy:100.000%
Epoch : 9 [6400/60000 (11%)]    Loss: 1.494410   Accuracy:98.181%
Epoch : 9 [12800/60000 (21%)]   Loss: 1.469811   Accuracy:98.067%
Epoch : 9 [19200/60000 (32%)]   Loss: 1.465618   Accuracy:98.206%
Epoch : 9 [25600/60000 (43%)]   Loss: 1.476494   Accuracy:98.291%
Epoch : 9 [32000/60000 (53%)]   Loss: 1.520020   Accuracy:98.305%
Epoch : 9 [38400/60000 (64%)]   Loss: 1.491564   Accuracy:98.319%
Epoch : 9 [44800/60000 (75%)]   Loss: 1.467684   Accuracy:98.305%
Epoch : 9 [51200/60000 (85%)]   Loss: 1.507861   Accuracy:98.278%
Epoch : 9 [57600/60000 (96%)]   Loss: 1.473866   Accuracy:98.274%
Epoch : 10 [0/60000 (0%)]       Loss: 1.463535   Accuracy:100.000%
Epoch : 10 [6400/60000 (11%)]   Loss: 1.506626   Accuracy:98.212%
Epoch : 10 [12800/60000 (21%)]  Loss: 1.474996   Accuracy:98.044%
Epoch : 10 [19200/60000 (32%)]  Loss: 1.468909   Accuracy:98.180%
Epoch : 10 [25600/60000 (43%)]  Loss: 1.479346   Accuracy:98.322%
Epoch : 10 [32000/60000 (53%)]  Loss: 1.515264   Accuracy:98.342%
Epoch : 10 [38400/60000 (64%)]  Loss: 1.491170   Accuracy:98.361%
Epoch : 10 [44800/60000 (75%)]  Loss: 1.466704   Accuracy:98.356%
Epoch : 10 [51200/60000 (85%)]  Loss: 1.501485   Accuracy:98.329%
Epoch : 10 [57600/60000 (96%)]  Loss: 1.480608   Accuracy:98.317%
Epoch : 11 [0/60000 (0%)]       Loss: 1.463524   Accuracy:100.000%
Epoch : 11 [6400/60000 (11%)]   Loss: 1.496235   Accuracy:98.197%
Epoch : 11 [12800/60000 (21%)]  Loss: 1.474311   Accuracy:98.044%
Epoch : 11 [19200/60000 (32%)]  Loss: 1.470642   Accuracy:98.243%
Epoch : 11 [25600/60000 (43%)]  Loss: 1.477164   Accuracy:98.346%
Epoch : 11 [32000/60000 (53%)]  Loss: 1.512678   Accuracy:98.370%
```

```
Epoch : 11 [38400/60000 (64%)]  Loss: 1.491341   Accuracy:98.410%
Epoch : 11 [44800/60000 (75%)]  Loss: 1.472550   Accuracy:98.394%
Epoch : 11 [51200/60000 (85%)]  Loss: 1.498510   Accuracy:98.380%
Epoch : 11 [57600/60000 (96%)]  Loss: 1.477948   Accuracy:98.367%
Epoch : 12 [0/60000 (0%)]       Loss: 1.464005   Accuracy:100.000%
Epoch : 12 [6400/60000 (11%)]   Loss: 1.506589   Accuracy:98.368%
Epoch : 12 [12800/60000 (21%)]  Loss: 1.478780   Accuracy:98.215%
Epoch : 12 [19200/60000 (32%)]  Loss: 1.467215   Accuracy:98.362%
Epoch : 12 [25600/60000 (43%)]  Loss: 1.476988   Accuracy:98.478%
Epoch : 12 [32000/60000 (53%)]  Loss: 1.515843   Accuracy:98.498%
Epoch : 12 [38400/60000 (64%)]  Loss: 1.491370   Accuracy:98.504%
Epoch : 12 [44800/60000 (75%)]  Loss: 1.472873   Accuracy:98.470%
Epoch : 12 [51200/60000 (85%)]  Loss: 1.500720   Accuracy:98.433%
Epoch : 12 [57600/60000 (96%)]  Loss: 1.471876   Accuracy:98.428%
Epoch : 13 [0/60000 (0%)]       Loss: 1.463105   Accuracy:100.000%
Epoch : 13 [6400/60000 (11%)]   Loss: 1.503983   Accuracy:98.336%
Epoch : 13 [12800/60000 (21%)]  Loss: 1.477224   Accuracy:98.270%
Epoch : 13 [19200/60000 (32%)]  Loss: 1.466545   Accuracy:98.352%
Epoch : 13 [25600/60000 (43%)]  Loss: 1.474158   Accuracy:98.471%
Epoch : 13 [32000/60000 (53%)]  Loss: 1.512237   Accuracy:98.498%
Epoch : 13 [38400/60000 (64%)]  Loss: 1.491049   Accuracy:98.509%
Epoch : 13 [44800/60000 (75%)]  Loss: 1.467018   Accuracy:98.479%
Epoch : 13 [51200/60000 (85%)]  Loss: 1.494388   Accuracy:98.454%
Epoch : 13 [57600/60000 (96%)]  Loss: 1.471699   Accuracy:98.445%
Epoch : 14 [0/60000 (0%)]       Loss: 1.462810   Accuracy:100.000%
Epoch : 14 [6400/60000 (11%)]   Loss: 1.499626   Accuracy:98.461%
Epoch : 14 [12800/60000 (21%)]  Loss: 1.482630   Accuracy:98.301%
Epoch : 14 [19200/60000 (32%)]  Loss: 1.466289   Accuracy:98.425%
Epoch : 14 [25600/60000 (43%)]  Loss: 1.475875   Accuracy:98.525%
Epoch : 14 [32000/60000 (53%)]  Loss: 1.508329   Accuracy:98.555%
Epoch : 14 [38400/60000 (64%)]  Loss: 1.491045   Accuracy:98.530%
Epoch : 14 [44800/60000 (75%)]  Loss: 1.469766   Accuracy:98.512%
Epoch : 14 [51200/60000 (85%)]  Loss: 1.507986   Accuracy:98.481%
Epoch : 14 [57600/60000 (96%)]  Loss: 1.474667   Accuracy:98.456%
Epoch : 15 [0/60000 (0%)]       Loss: 1.463211   Accuracy:100.000%
Epoch : 15 [6400/60000 (11%)]   Loss: 1.497269   Accuracy:98.585%
Epoch : 15 [12800/60000 (21%)]  Loss: 1.481877   Accuracy:98.457%
Epoch : 15 [19200/60000 (32%)]  Loss: 1.465198   Accuracy:98.523%
Epoch : 15 [25600/60000 (43%)]  Loss: 1.476220   Accuracy:98.607%
Epoch : 15 [32000/60000 (53%)]  Loss: 1.502374   Accuracy:98.633%
Epoch : 15 [38400/60000 (64%)]  Loss: 1.491117   Accuracy:98.621%
Epoch : 15 [44800/60000 (75%)]  Loss: 1.472667   Accuracy:98.588%
Epoch : 15 [51200/60000 (85%)]  Loss: 1.507521   Accuracy:98.556%
Epoch : 15 [57600/60000 (96%)]  Loss: 1.470615   Accuracy:98.546%
Epoch : 16 [0/60000 (0%)]       Loss: 1.462290   Accuracy:100.000%
Epoch : 16 [6400/60000 (11%)]   Loss: 1.496732   Accuracy:98.601%
Epoch : 16 [12800/60000 (21%)]  Loss: 1.487032   Accuracy:98.449%
Epoch : 16 [19200/60000 (32%)]  Loss: 1.465802   Accuracy:98.570%
```

```
Epoch : 16 [25600/60000 (43%)]  Loss: 1.476657   Accuracy:98.654%
Epoch : 16 [32000/60000 (53%)]  Loss: 1.502920   Accuracy:98.658%
Epoch : 16 [38400/60000 (64%)]  Loss: 1.491240   Accuracy:98.644%
Epoch : 16 [44800/60000 (75%)]  Loss: 1.465179   Accuracy:98.619%
Epoch : 16 [51200/60000 (85%)]  Loss: 1.483269   Accuracy:98.599%
Epoch : 16 [57600/60000 (96%)]  Loss: 1.471581   Accuracy:98.579%
Epoch : 17 [0/60000 (0%)]       Loss: 1.463269   Accuracy:100.000%
Epoch : 17 [6400/60000 (11%)]   Loss: 1.497206   Accuracy:98.523%
Epoch : 17 [12800/60000 (21%)]  Loss: 1.481294   Accuracy:98.465%
Epoch : 17 [19200/60000 (32%)]  Loss: 1.465512   Accuracy:98.554%
Epoch : 17 [25600/60000 (43%)]  Loss: 1.471454   Accuracy:98.662%
Epoch : 17 [32000/60000 (53%)]  Loss: 1.502186   Accuracy:98.654%
Epoch : 17 [38400/60000 (64%)]  Loss: 1.491244   Accuracy:98.652%
Epoch : 17 [44800/60000 (75%)]  Loss: 1.472111   Accuracy:98.626%
Epoch : 17 [51200/60000 (85%)]  Loss: 1.505261   Accuracy:98.612%
Epoch : 17 [57600/60000 (96%)]  Loss: 1.474675   Accuracy:98.600%
Epoch : 18 [0/60000 (0%)]       Loss: 1.462605   Accuracy:100.000%
Epoch : 18 [6400/60000 (11%)]   Loss: 1.494833   Accuracy:98.601%
Epoch : 18 [12800/60000 (21%)]  Loss: 1.493096   Accuracy:98.519%
Epoch : 18 [19200/60000 (32%)]  Loss: 1.466309   Accuracy:98.596%
Epoch : 18 [25600/60000 (43%)]  Loss: 1.475530   Accuracy:98.658%
Epoch : 18 [32000/60000 (53%)]  Loss: 1.504336   Accuracy:98.658%
Epoch : 18 [38400/60000 (64%)]  Loss: 1.491131   Accuracy:98.668%
Epoch : 18 [44800/60000 (75%)]  Loss: 1.469511   Accuracy:98.646%
Epoch : 18 [51200/60000 (85%)]  Loss: 1.502256   Accuracy:98.628%
Epoch : 18 [57600/60000 (96%)]  Loss: 1.470402   Accuracy:98.603%
Epoch : 19 [0/60000 (0%)]       Loss: 1.463854   Accuracy:100.000%
Epoch : 19 [6400/60000 (11%)]   Loss: 1.501996   Accuracy:98.601%
Epoch : 19 [12800/60000 (21%)]  Loss: 1.486060   Accuracy:98.504%
Epoch : 19 [19200/60000 (32%)]  Loss: 1.465211   Accuracy:98.622%
Epoch : 19 [25600/60000 (43%)]  Loss: 1.474832   Accuracy:98.701%
Epoch : 19 [32000/60000 (53%)]  Loss: 1.501074   Accuracy:98.711%
Epoch : 19 [38400/60000 (64%)]  Loss: 1.491320   Accuracy:98.696%
Epoch : 19 [44800/60000 (75%)]  Loss: 1.477404   Accuracy:98.642%
Epoch : 19 [51200/60000 (85%)]  Loss: 1.504792   Accuracy:98.608%
Epoch : 19 [57600/60000 (96%)]  Loss: 1.476464   Accuracy:98.607%
```

[33]:
```
test(net2, test_loader)
```

```
Test accuracy:98.010%
```

(Bonus) **TASK 2**

Before getting started read this article on Deep Dream.

Description of the task: Now we have trained the network to classify all the digits from 0 to 10. But is there a way to visualize what the network has learnt?

Well how about we learn (by backprop + Gradient Descent) the input image after fixing the expected output and all the network parameters. In other words say we want to visualize what 7

looks like to the network. Then we will do the following: 1. Initialize image tensor as im (maybe with all zeros). 2. Pass the image through the network. 3. Compute the loss with output of the net and expected output of 7. 4. Run backpropogation upto the image tensor. 5. Update the image using the update rule. (Specified by torch.optim object)

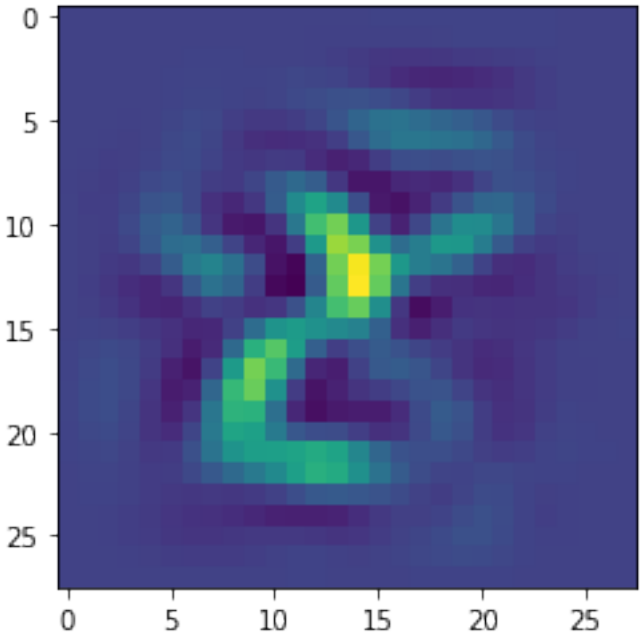
Remember that while creating the optimization object (torch.optim object) you have to pass to it a list of parameters to be optimized. Which in this case won't be model.parameters() but rather [im,].

Note: We are not using training or test data anywhere here!

[34]:
```
def train_im(model, train_loader, digit = 7 ,iters = 1000, lossF = None):
  """
  Train the input image to match the <digit>. Run <iters> iterations of␣
 ↪Gradient Descent.
  """
  im = torch.zeros_like(train_data[1][0]).view(1, 1, 28, 28).to(device)
  im = Variable(im, requires_grad = True)
  digit = Variable(torch.tensor(digit)).to(device).view(1)
  # Put your code here:
  # __begin
  optim = torch.optim.Adam ([im, ], lr = 4e-4, weight_decay=1e-3)

  if lossF == None:
    lossF = nn.CrossEntropyLoss()

  for epoch in range(iters):
    correct = 0

    output = model(im)
    loss = lossF (output, digit)

    loss.backward()

    optim.step()
    optim.zero_grad()

  #__end

  return im
```

[40]:
```
## Let's run our function for 7 and see what image we get.

im = train_im (net, train_loader, digit=8, iters = 10000)
im = np.asarray(im.view(28, 28).cpu().detach())
plt.imshow(im)
```

[40]:
```
<matplotlib.image.AxesImage at 0x7f6f012f8e48>
```

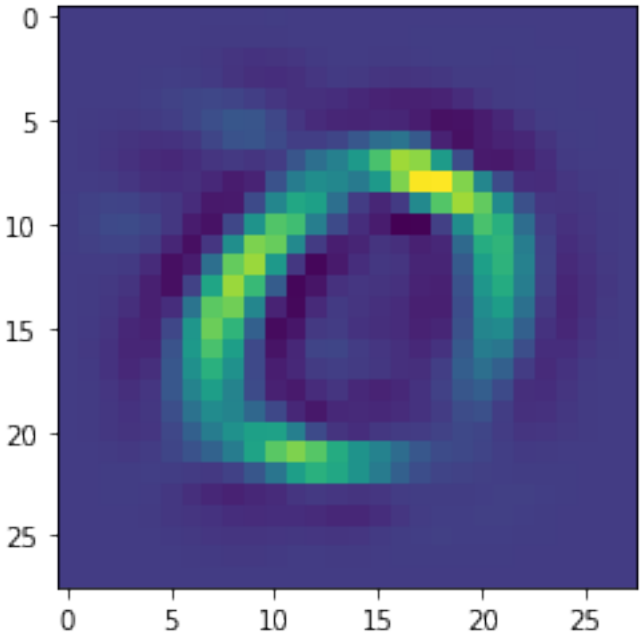


In the the following cell we will run the function for all the digits from 0 to 9 and print the outputs!

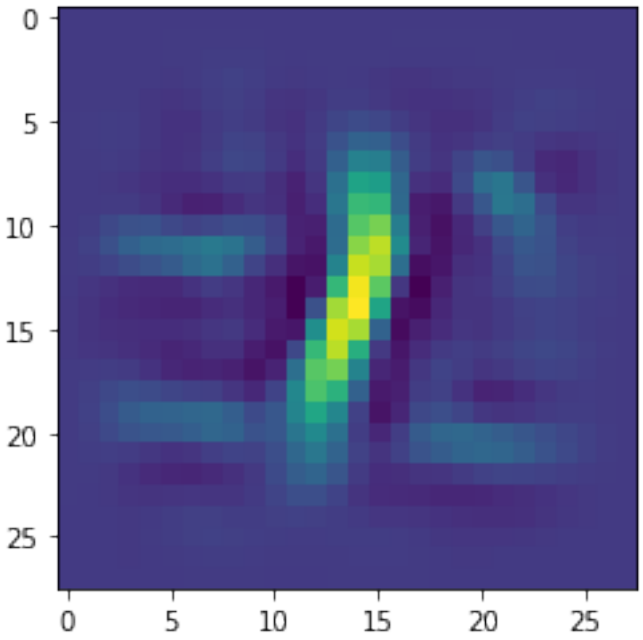
Enjoy you are done!

[36]:
```
for i in range (10):
  im = train_im (net, train_loader, digit=i, iters=10000)
  im = np.asarray(im.view(28, 28).cpu().detach())
  print (i)
  plt.imshow(im)
  plt.show()
```

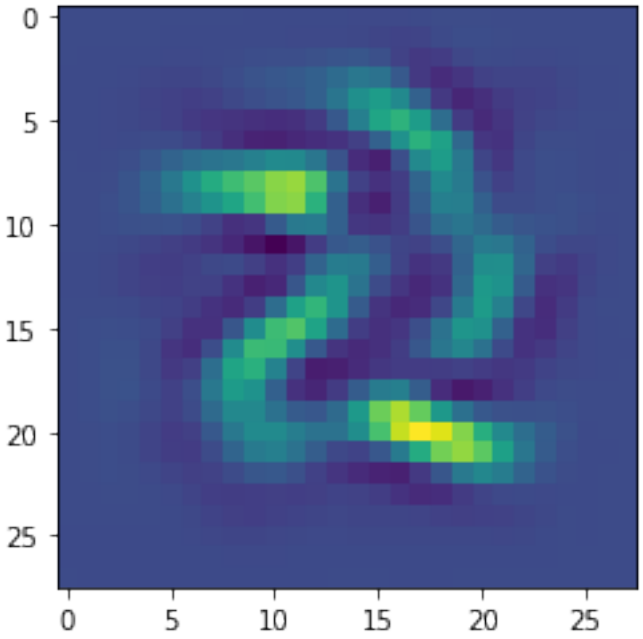
0



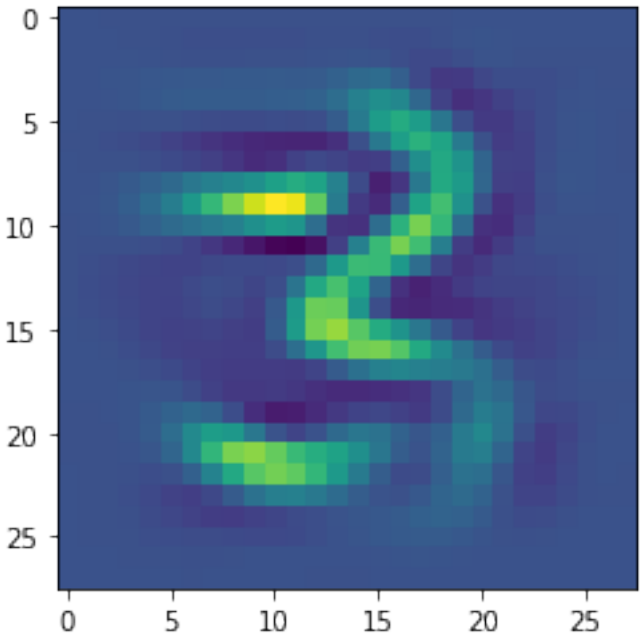
1



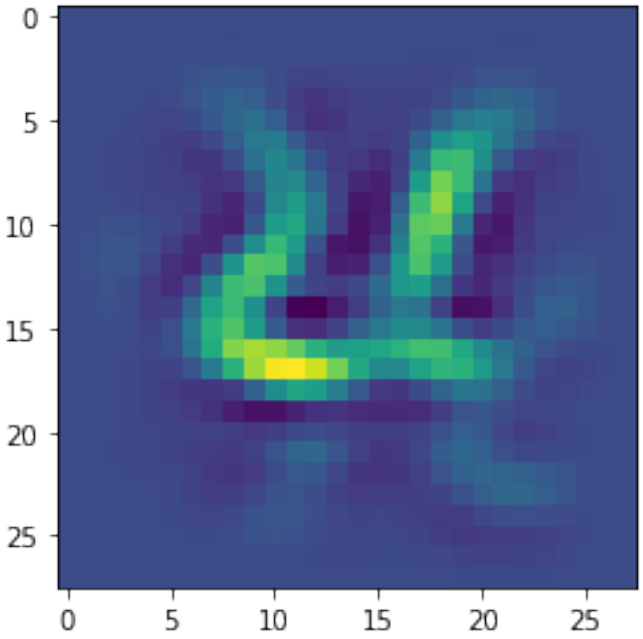
2



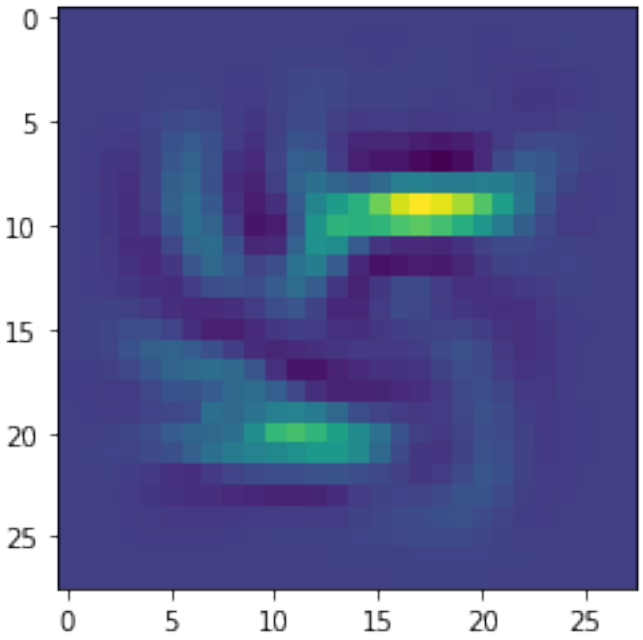
3



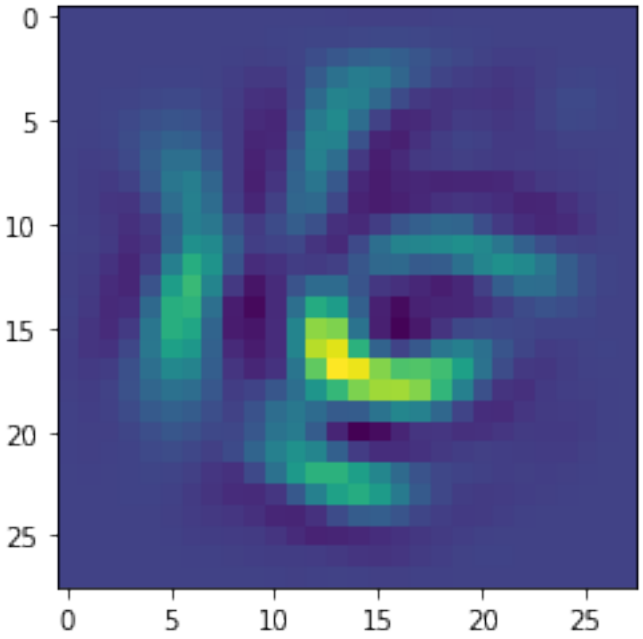
4



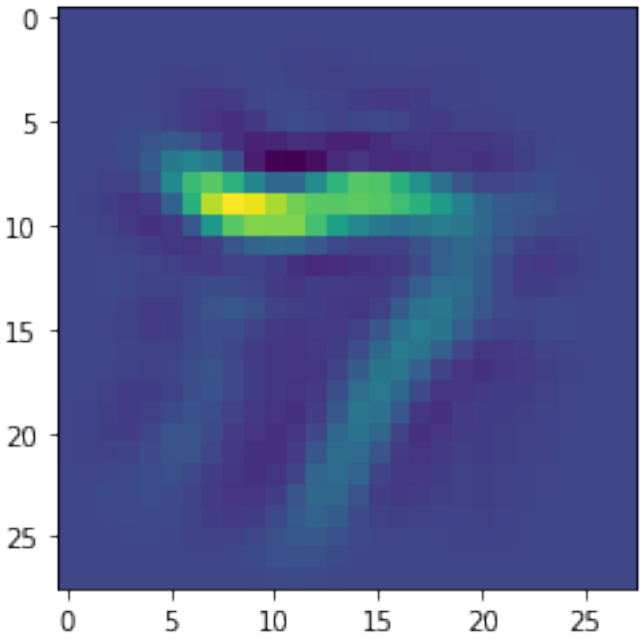
5



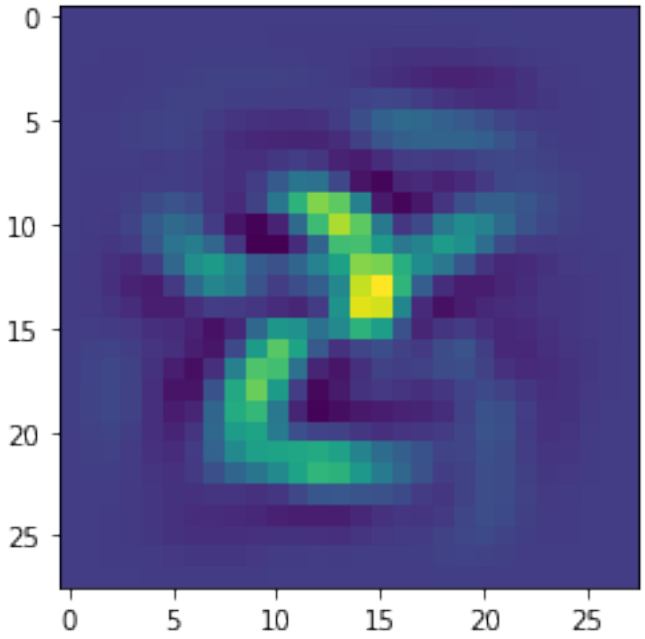
6



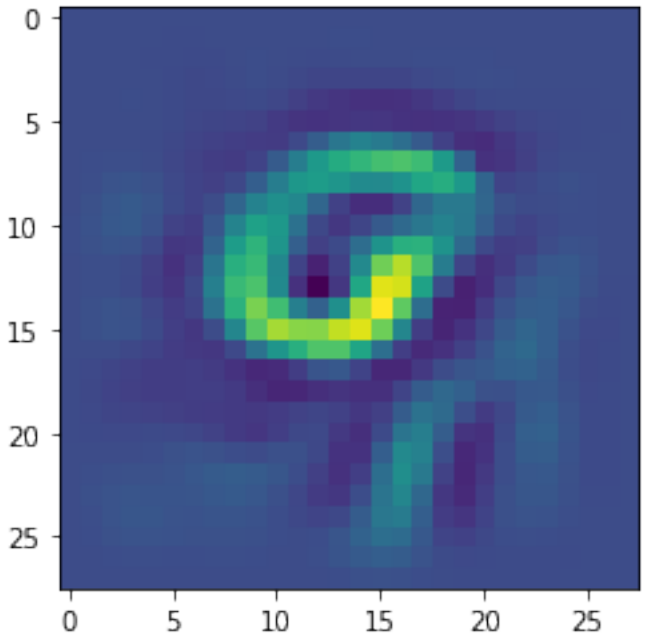
7



8



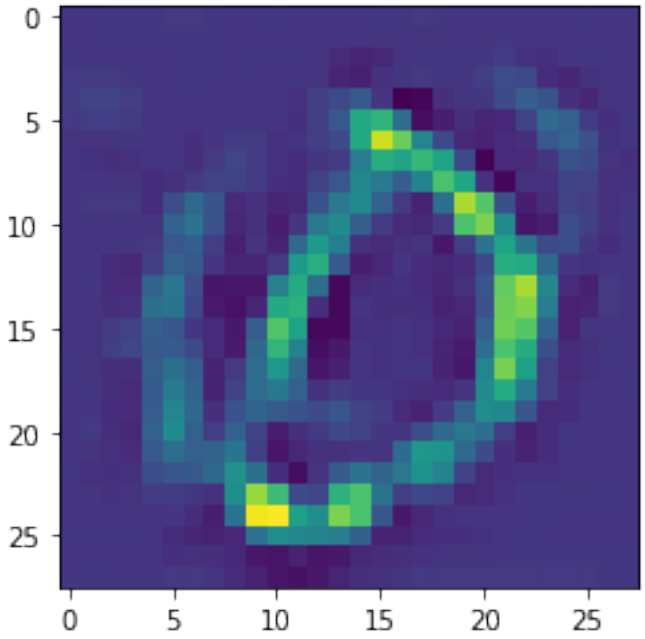
9



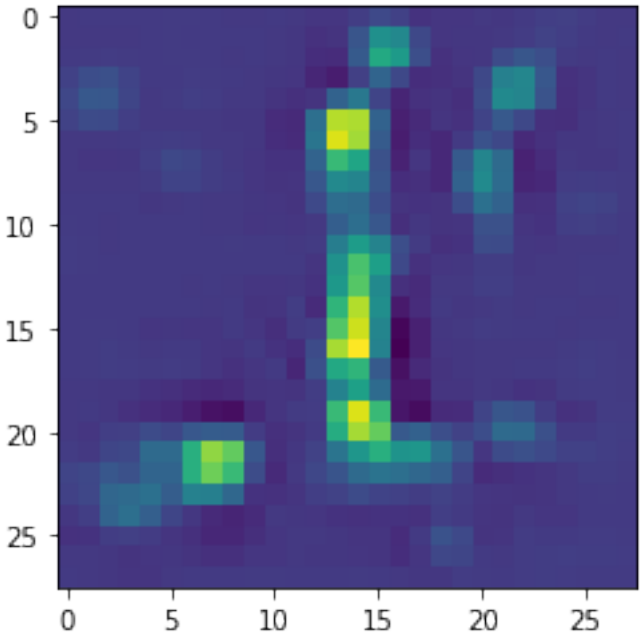
[37]:
```
# Same Procedure as above, but with the better performing model. The features␣
↪learned are not as smooth for the better model imo
```

```
for i in range (10):
  im = train_im (net2, train_loader, digit=i, iters=10000)
  im = np.asarray(im.view(28, 28).cpu().detach())
  print (i)
  plt.imshow(im)
  plt.show()
```

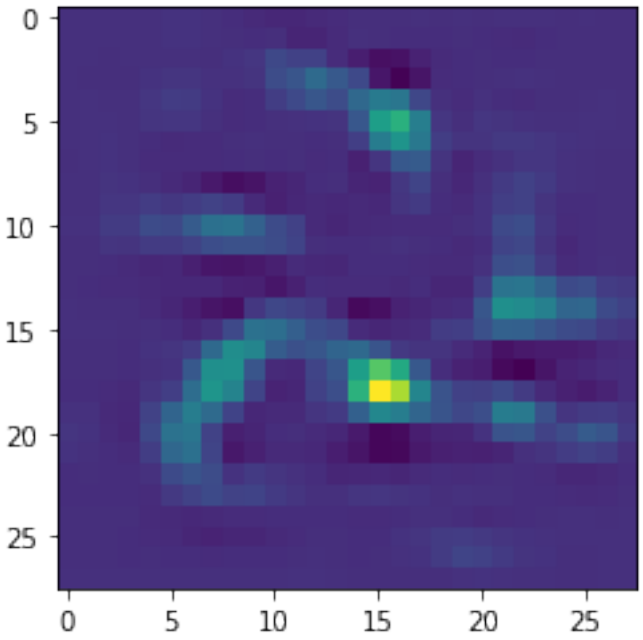
0



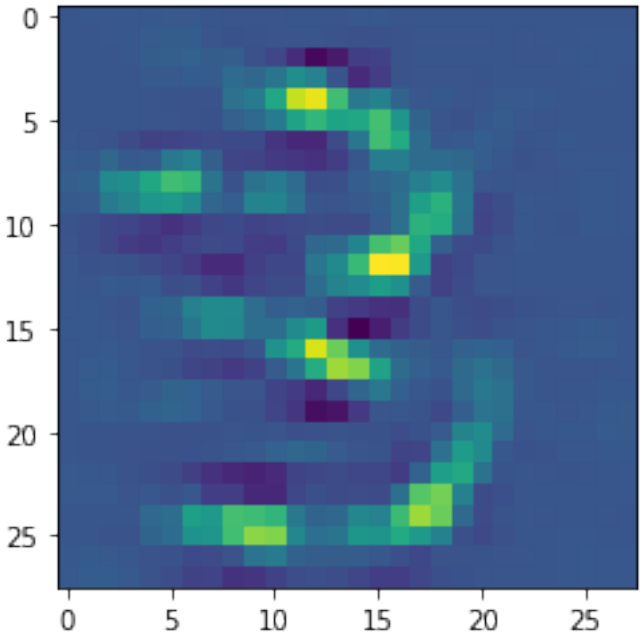
1



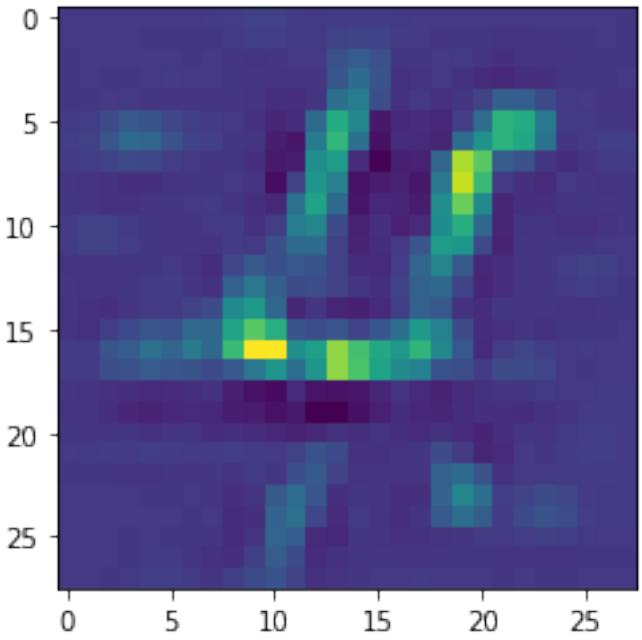
2



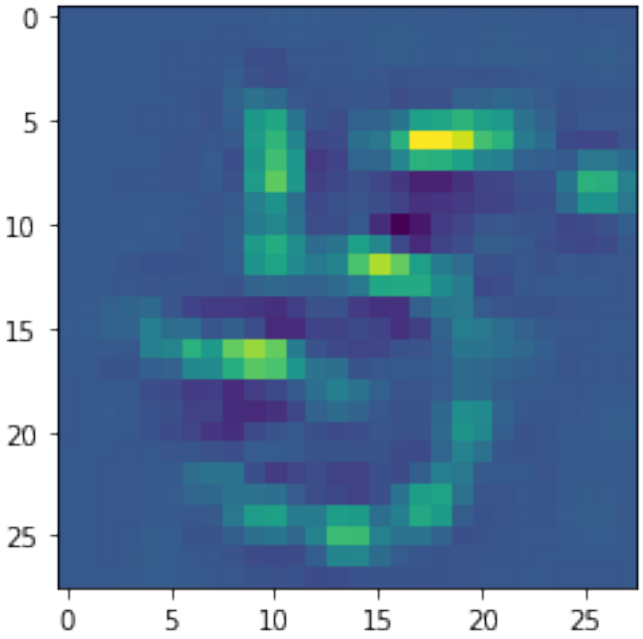
3



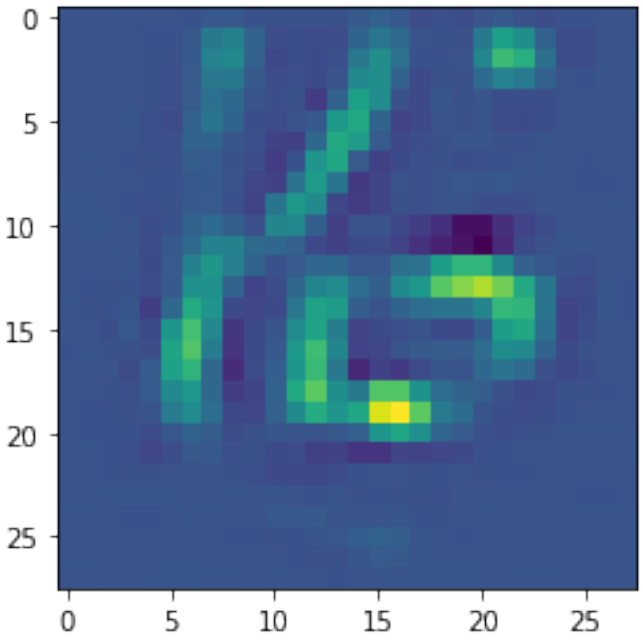
4



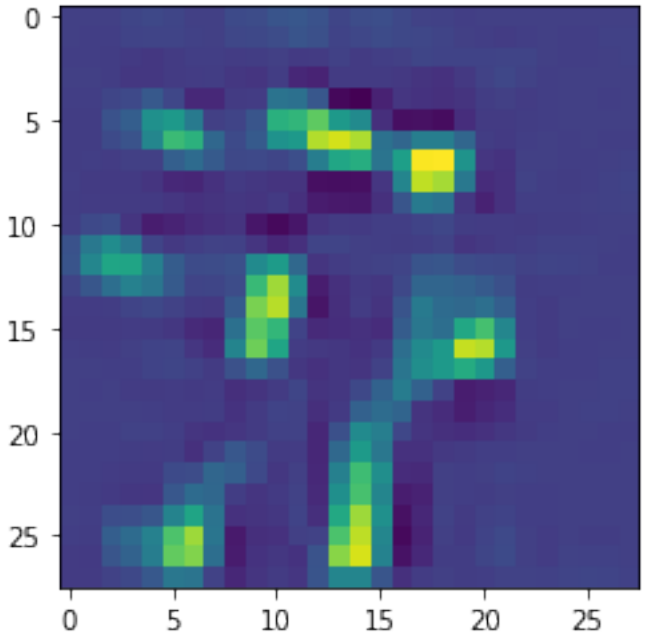
5



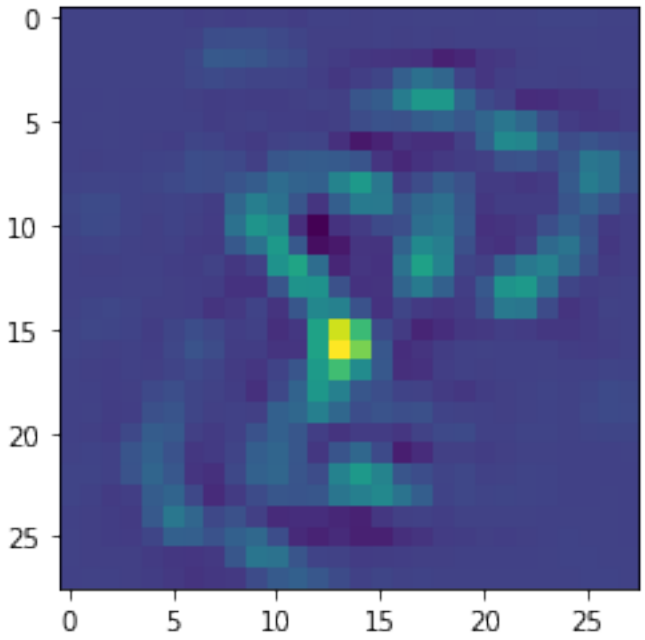
6



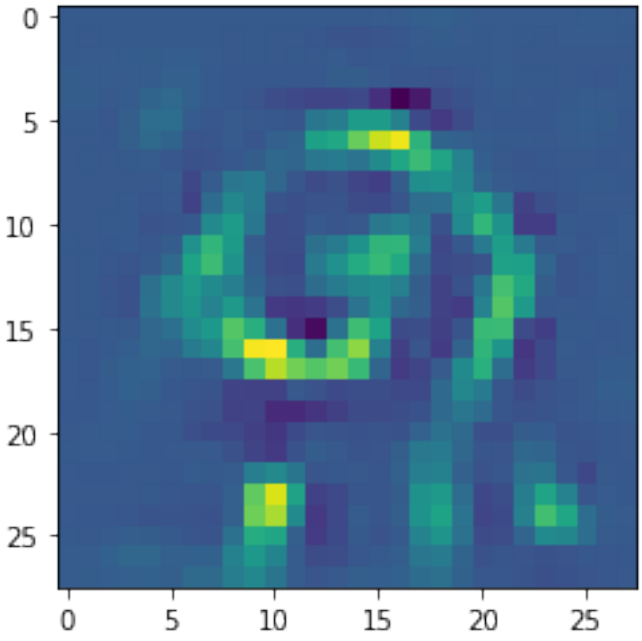
7



8



9


0
5
10
15
20
25
0
5
10
15
20
25