```
In [1]: %matplotlib inline
```

1 Universal adversarial attack

A simplification of the universal adversarial training as proposed in Universal adversarial training by Shafahi et al., where we aim to find the optimal attack δ for a pre-trained network

Author: Martin Benning
Date: 11.04.2019
Last modified: 11.04.2019

```
In [2]: from __future__ import print_function
        import torch
        import torch.nn as nn
        import torch.nn.functional as F
        import torch.optim as optim
        from torch.autograd import Variable
        from torchvision import datasets, transforms
        import numpy as np
        import matplotlib.pyplot as plt

        pretrained_model = "../data/lenet_mnist_model.pth"
```

```
In [3]: # LeNet Model definition
        class Net(nn.Module):
            def __init__(self):
                super(Net, self).__init__()
                self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
                self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
                self.conv2_drop = nn.Dropout2d()
                self.fc1 = nn.Linear(320, 50)
                self.fc2 = nn.Linear(50, 10)

            def forward(self, x):
                x = F.relu(F.max_pool2d(self.conv1(x), 2))
                x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
                x = x.view(-1, 320)
                x = F.relu(self.fc1(x))
                x = F.dropout(x, training=self.training)
                x = self.fc2(x)
                return F.log_softmax(x, dim=1)

        # MNIST Train & test dataset and dataloader declaration

        train_loader = torch.utils.data.DataLoader(
            datasets.MNIST('../data/MNIST', train=True, download=True,
             transform=transforms.Compose([
                    transforms.ToTensor(),
                    ])),
```

```
        batch_size=6000, shuffle=True)

test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data/MNIST', train=False, download=True,
     transform=transforms.Compose([
            transforms.ToTensor(),
            ])),
        batch_size=1, shuffle=True)

# Initialize the network
model = Net()

# Load the pretrained model
model.load_state_dict(torch.load(pretrained_model, map_location='cpu'))

# Set the model in evaluation mode. In this case this is for the Dropout layers
model.eval()
```

Out[3]:
```
Net(
  (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (conv2_drop): Dropout2d(p=0.5)
  (fc1): Linear(in_features=320, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
)
```

In [5]:
```
no_of_epochs = 40
perturbation = Variable(torch.zeros(train_loader.dataset.train_data.size(1), \
                          train_loader.dataset.train_data.size(2)),
stepsize = 0.1
epsilon = 3

for epoch in range(no_of_epochs):

    for data, target in train_loader:

        output = model(data + perturbation.repeat(train_loader.batch_size, \
        1, 1, 1))

        loss = F.nll_loss(output, target)
        model.zero_grad()
        loss.backward()

        perturbation.data = perturbation.data + stepsize * perturbation.grad
        perturbation.data = perturbation.data \
        / torch.norm(perturbation.data.view(-1, 784)) * epsilon

    if epoch % 1 == 0:
```

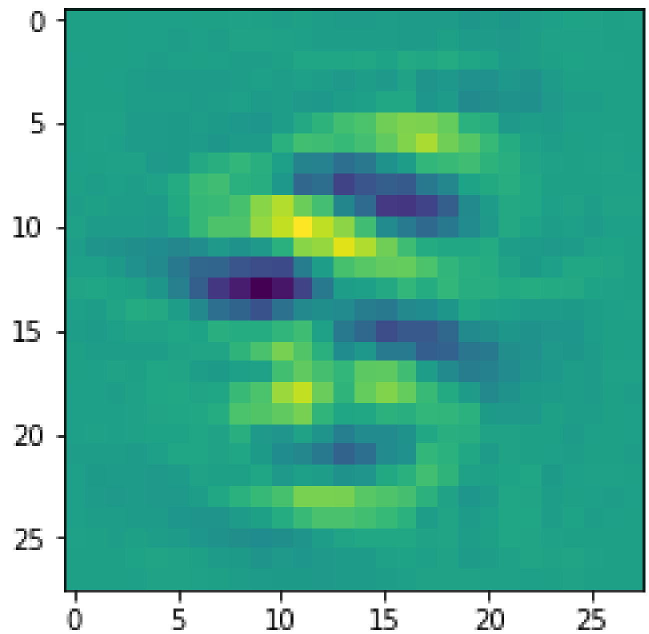
```python
                print ('Iteration [%d/%d], Loss: %.4f'
                       %(epoch + 1, no_of_epochs, loss.item()))

        print('Iteration [%d/%d] completed, Loss: %.4f'
                       %(epoch + 1, no_of_epochs, loss.item()))
```

```
Iteration [1/40], Loss: 0.6716
Iteration [2/40], Loss: 0.8238
Iteration [3/40], Loss: 0.8890
Iteration [4/40], Loss: 0.9191
Iteration [5/40], Loss: 0.9195
Iteration [6/40], Loss: 0.9615
Iteration [7/40], Loss: 0.9433
Iteration [8/40], Loss: 0.9508
Iteration [9/40], Loss: 0.9475
Iteration [10/40], Loss: 0.9696
Iteration [11/40], Loss: 0.9665
Iteration [12/40], Loss: 0.9472
Iteration [13/40], Loss: 0.9512
Iteration [14/40], Loss: 0.9720
Iteration [15/40], Loss: 0.9590
Iteration [16/40], Loss: 0.9805
Iteration [17/40], Loss: 0.9618
Iteration [18/40], Loss: 0.9893
Iteration [19/40], Loss: 0.9672
Iteration [20/40], Loss: 0.9839
Iteration [21/40], Loss: 0.9723
Iteration [22/40], Loss: 0.9580
Iteration [23/40], Loss: 0.9749
Iteration [24/40], Loss: 0.9892
Iteration [25/40], Loss: 0.9820
Iteration [26/40], Loss: 0.9826
Iteration [27/40], Loss: 0.9628
Iteration [28/40], Loss: 0.9865
Iteration [29/40], Loss: 0.9831
Iteration [30/40], Loss: 0.9876
Iteration [31/40], Loss: 0.9922
Iteration [32/40], Loss: 0.9910
Iteration [33/40], Loss: 0.9792
Iteration [34/40], Loss: 0.9843
Iteration [35/40], Loss: 0.9943
Iteration [36/40], Loss: 0.9676
Iteration [37/40], Loss: 0.9951
Iteration [38/40], Loss: 0.9843
Iteration [39/40], Loss: 0.9877
Iteration [40/40], Loss: 0.9929
Iteration [40/40] completed, Loss: 0.9929
```

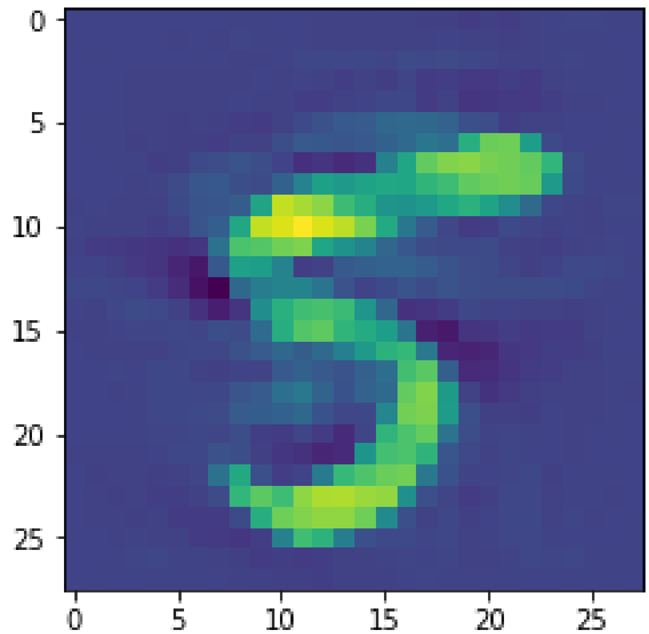
In [6]:
```
plt.imshow(perturbation.data.numpy())
plt.savefig('../Figures/attack_perturbation.png')
print(torch.norm(perturbation.data.view(-1, 784)))
```

```
tensor(3.)
```



In [10]:
```
dataiter = iter(test_loader)
images, labels = dataiter.next()
plt.imshow(images[0][0].data.numpy())
plt.savefig('../Figures/attack_image.png')
plt.imshow(images[0][0].data.numpy() + perturbation.data.numpy())
plt.savefig('../Figures/attack_perturbed_image.png')
output1 = model(images)
pred1 = output1.max(1, keepdim=True)[1]
output2 = model(images + pertubation)
pred2 = output2.max(1, keepdim=True)[1]
print(labels, pred1, pred2)
```

```
tensor([5]) tensor([[5]]) tensor([[5]])
```




```
In [11]: def test_accuracy(model, test_loader, perturbation):

             # Accuracy counter
             correct = 0
             adv_examples = []

             # Loop over all examples in test set
             for data, target in test_loader:

                 # Forward pass the data through the model
                 perturbed_data = data + perturbation
                 output = model(perturbed_data)

                 # Check for success
                 final_pred = output.max(1, keepdim=True)[1]
                 # get the index of the max log-probability
                 if final_pred.item() == target.item():
                     correct += 1
                 else:
                     # Save some adv examples for visualisation later
                     if len(adv_examples) < 5:
                         adv_ex = perturbed_data.squeeze().detach().cpu().numpy()
                         adv_examples.append( (final_pred.item(), adv_ex) )

             # Calculate final accuracy for this epsilon
             final_acc = correct/float(len(test_loader))
```

```
        print("Test Accuracy = {} / {} = {}".format(correct, \
        len(test_loader), final_acc))

        # Return the accuracy and an adversarial example
        return final_acc, adv_examples
```

```
In [12]: acc, ex = test_accuracy(model, test_loader, \ torch.zeros(perturbation.size()))
```

```
Test Accuracy = 9810 / 10000 = 0.981
```

```
In [13]: acc, ex = test_accuracy(model, test_loader, perturbation)
```

```
Test Accuracy = 6899 / 10000 = 0.6899
```