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

1 Universal adversarial training & defence

A modification of the model proposed in Universal adversarial training by Shafahi et al., where we aim to simultaneously find the optimal attack δ and defence for a given network

Author(s): Martin Benning, Alex Wendland

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'))
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to
 ../data/MNIST/MNIST/raw/train-images-idx3-ubyte.gz
```

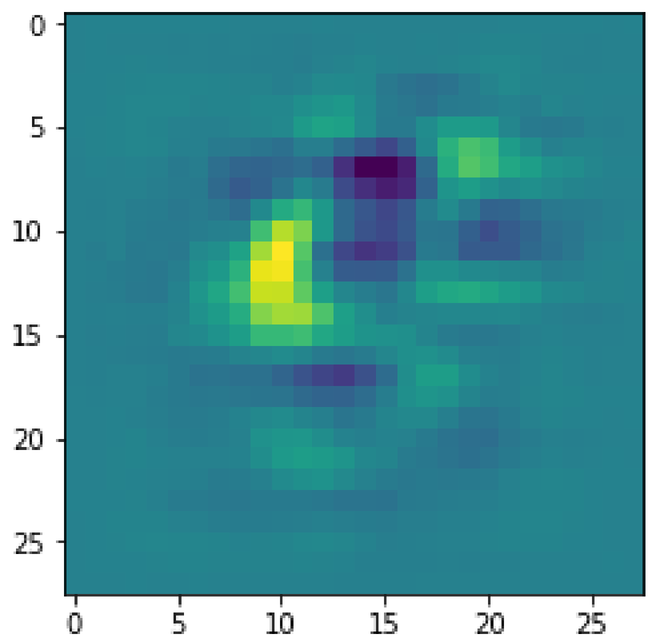
```
In [4]: perturbation = Variable(torch.zeros(train_loader.dataset.train_data.size(1), \
                                    train_loader.dataset.train_data.size(2)), \
                                    requires_grad=True)

        no_of_epochs = 150

        stepsize = 0.1
        optimiser = optim.SGD(model.parameters(), lr=stepsize, momentum=0.7)
        epsilon = 3

        for epoch in range(no_of_epochs):

            for data, target in train_loader:

                output_1 = model(data)
                output_2 = model(data + perturbation.repeat(train_loader.batch_size,\
                 1, 1, 1))

                loss_1 = F.nll_loss(output_1, target)
                loss_2 = F.nll_loss(output_2, target)
                loss = 1/2 * (loss_1 + loss_2)
                model.zero_grad()
                loss.backward()

                optimiser.step()

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

```
                                         * epsilon

            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 [150/150] completed, Loss: 0.1512
```

```
In [5]: plt.imshow(perturbation.data.numpy())
        plt.savefig('../Figures/attack_defence_perturbation.png')
        print(torch.norm(perturbation.data.view(-1, 784)))
```

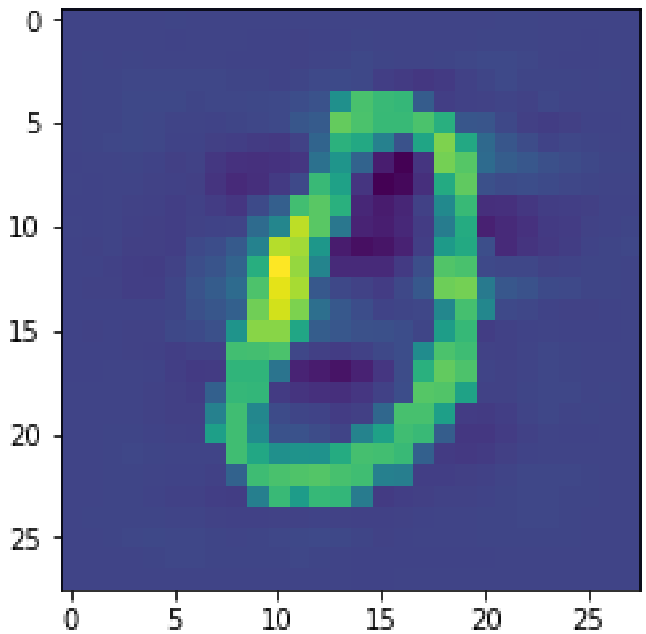
```
tensor(3.)
```




```
In [9]: # Set the model in evaluation mode. In this case this is for the Dropout layers
        model.eval()

        dataiter = iter(test_loader)
        images, labels = dataiter.next()
        plt.imshow(images[0][0].data.numpy())
        plt.savefig('../Figures/attack_defence_image.png')
        plt.imshow(images[0][0].data.numpy() + perturbation.data.numpy())
```

```
        plt.savefig('../Figures/attack_defence_perturbed_image.png')
        output1 = model(images)
        pred1 = output1.max(1, keepdim=True)[1]
        output2 = model(images + perturbation)
        pred2 = output2.max(1, keepdim=True)[1]
        print(labels, pred1, pred2)
```

```
tensor([0]) tensor([[0]]) tensor([[0]])
```



```
In [10]: 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 [11]: acc_1, ex_1 = test_accuracy(model, test_loader, torch.zeros(28, 28))
         acc_2, ex_2 = test_accuracy(model, test_loader, perturbation)
```

```
Test Accuracy = 9849 / 10000 = 0.9849
Test Accuracy = 9798 / 10000 = 0.9798
```

```
In [12]: # Initialize the network
         model_pretrained = Net()

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

         acc_3, ex_3 = test_accuracy(model_pretrained, test_loader, perturbation)
```

```
Test Accuracy = 7491 / 10000 = 0.7491
```