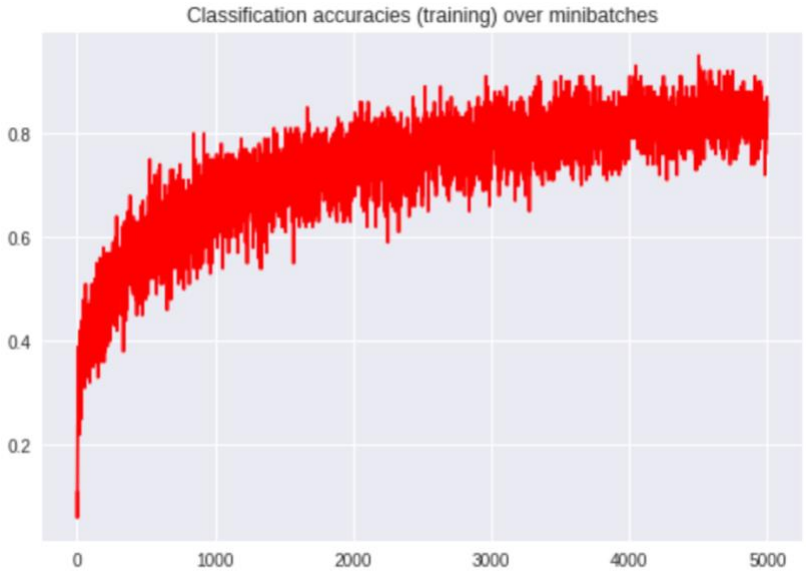
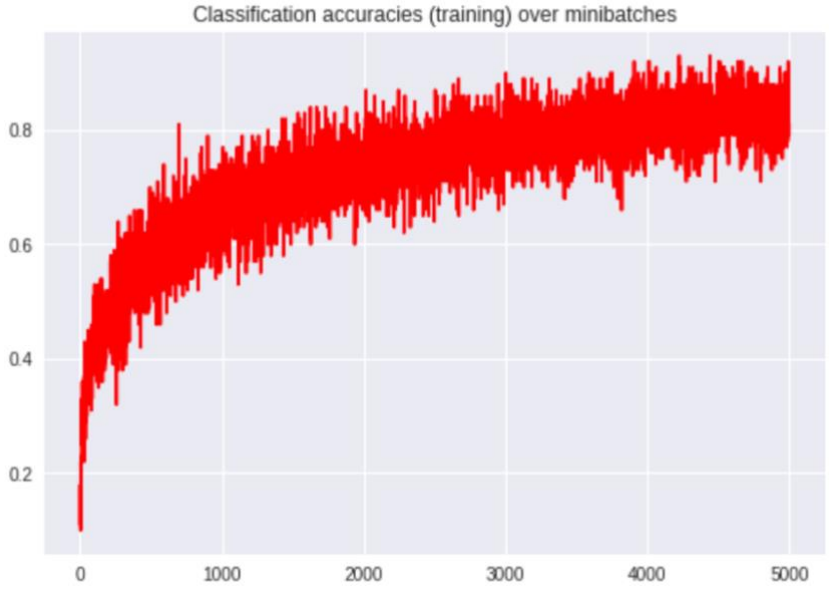
Question 1

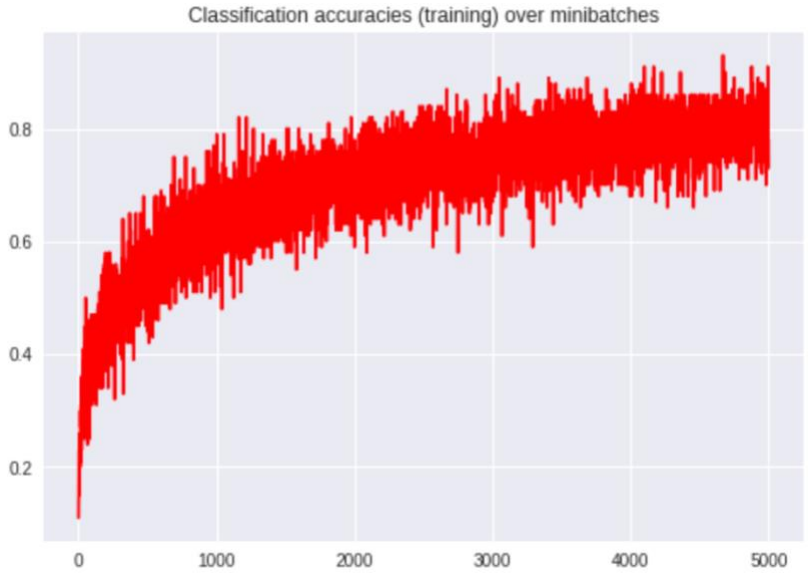
a) We achieve a final classification accuracy of 0.74 with no pre-processing.



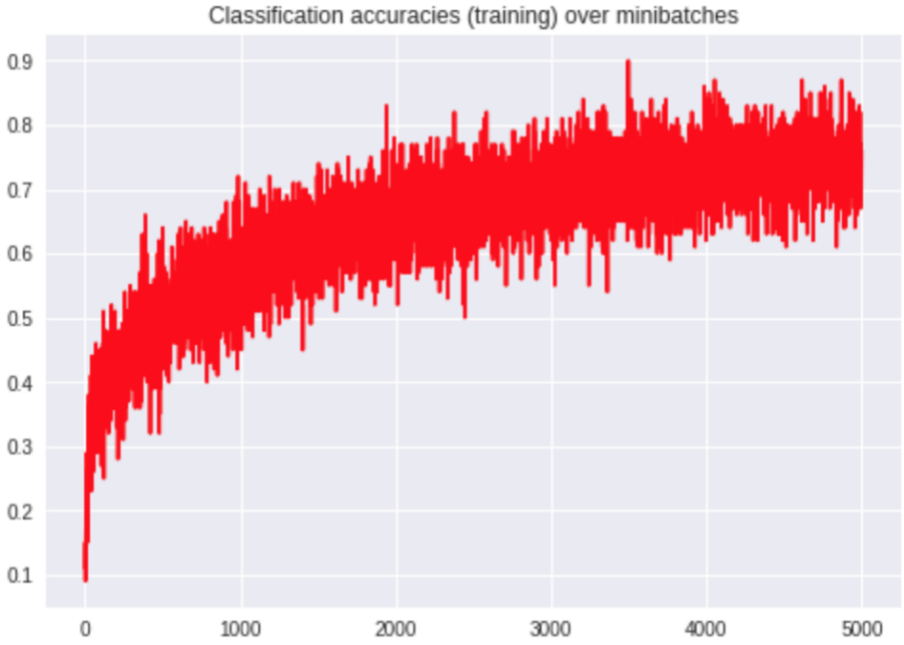
b) We achieve a final classification accuracy of 0.75 with normalization to 0 mean, 1 STDEV. The difference is not statistically significant from the previous case, which is surprising. It is possible that the overall brightness of pictures contains information as well.



c) We apply random horizontal flips with 0.5 probability on the training dataset. This augments the data by making it less likely that the model memorizes data. However, this achieves a test classification accuracy of 0.77, which is only marginally better than before.

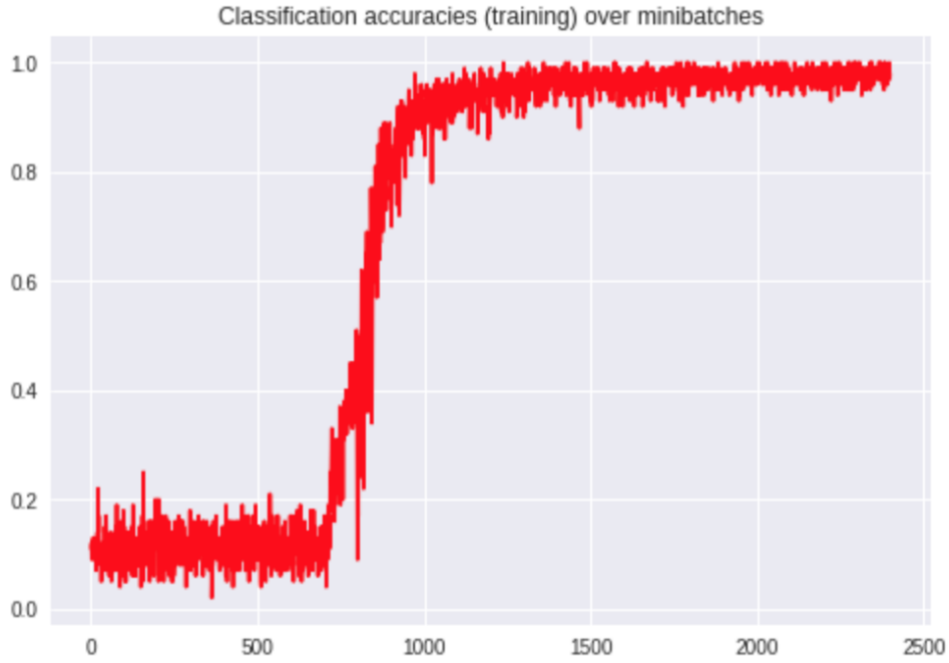


d) We pad the training image with 4 0-pixels on each side and crop a random 32 x 32 pixel square from the padded images. It performs comparably to the previous transforms, with 0.75 test classification accuracy. I hoped that this would increase accuracy, since it should teach the model something invariant to the pictures' positions.



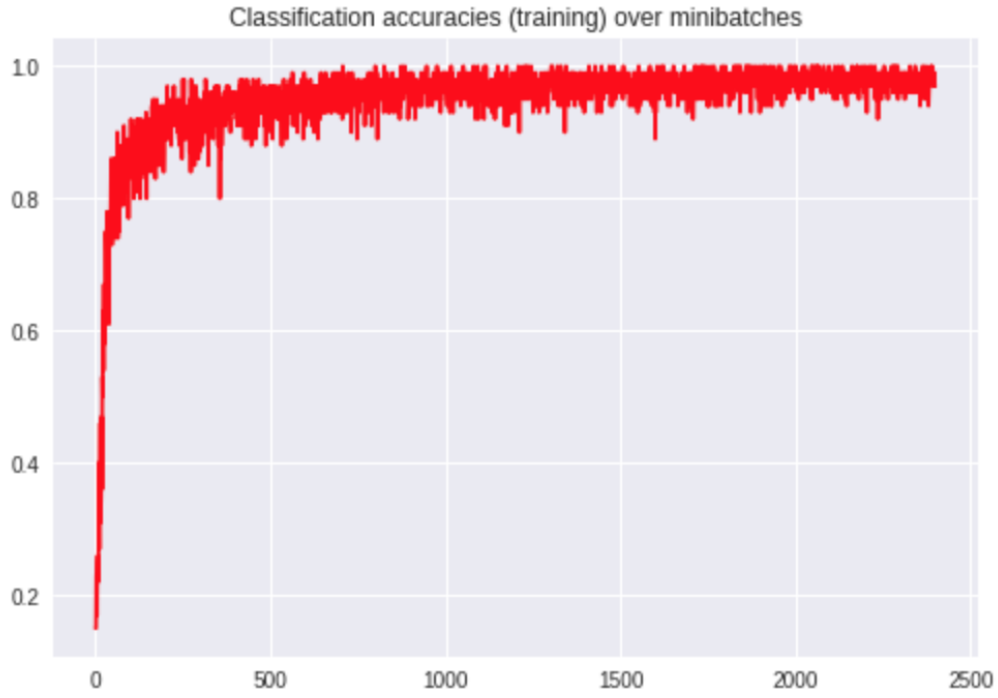
Question 2

a) We train a regular CNN on the MNIST dataset. We normalize both training and testing datasets to [-1, 1]. We achieve 98.2% testing accuracy after 4 iterations over training data.



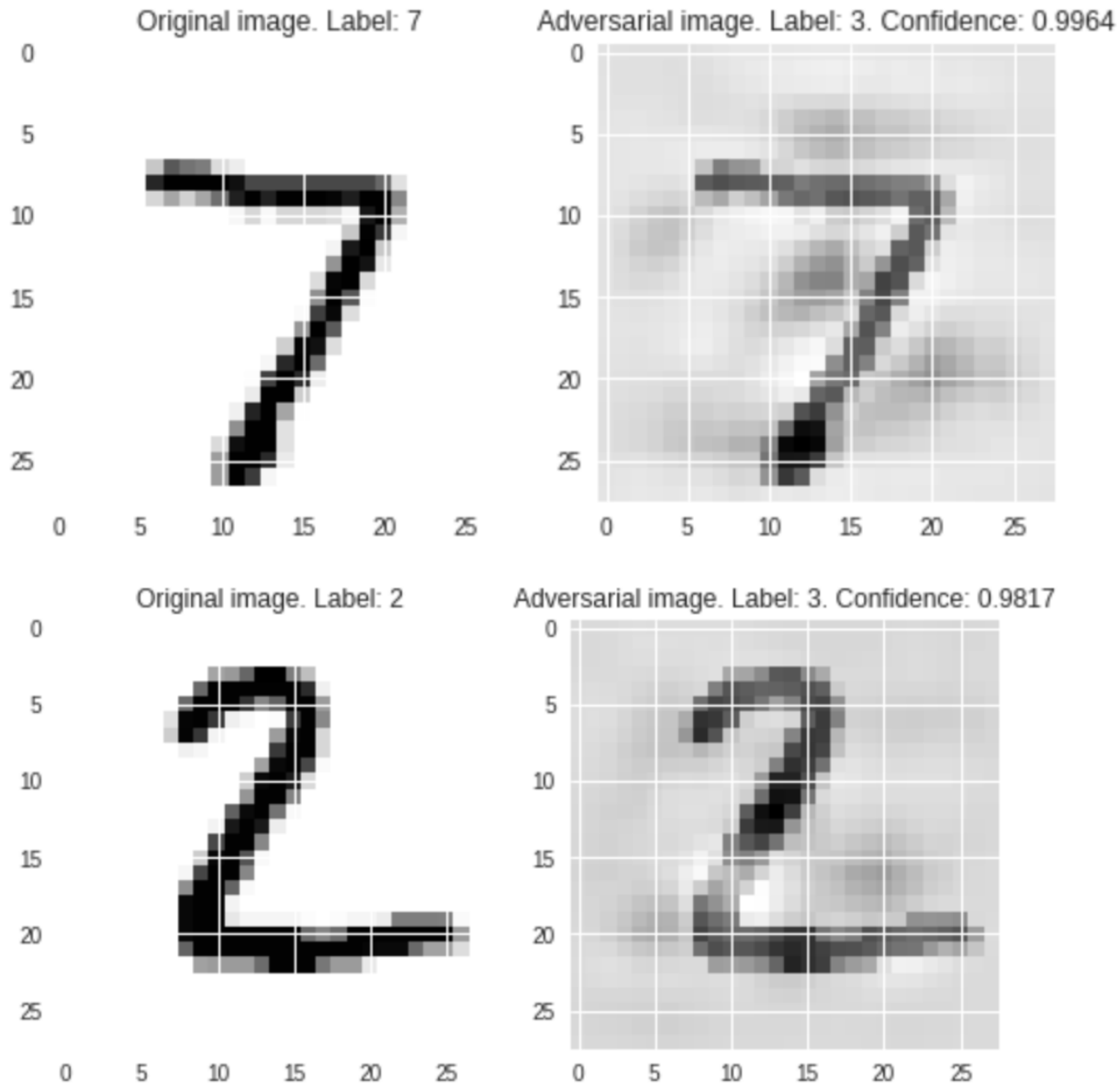
b) We implement the ‘sign’ and straight-through gradient. This code can be found in the code submission under the file *MySign.py*.

c) We replace all usages of ReLU with the modified Sign function, except in the output layer. We run the BNN with the same hyper-parameters as in part (a) and find that the network converges much faster. It seems to hit 98% training accuracy in just 1 pass over the data, whereas the original network required slightly more than 2 passes to converge. It also performs comparably, with 99.0% accuracy. The straight-through approximate gradient is good for this network since the true gradient function is 0 almost-everywhere. Instead, we can imagine this gradient as the probability that the given neuron might fire under some perturbation, and effectively 'dead' or 'fixed' if it is far away enough from 0.



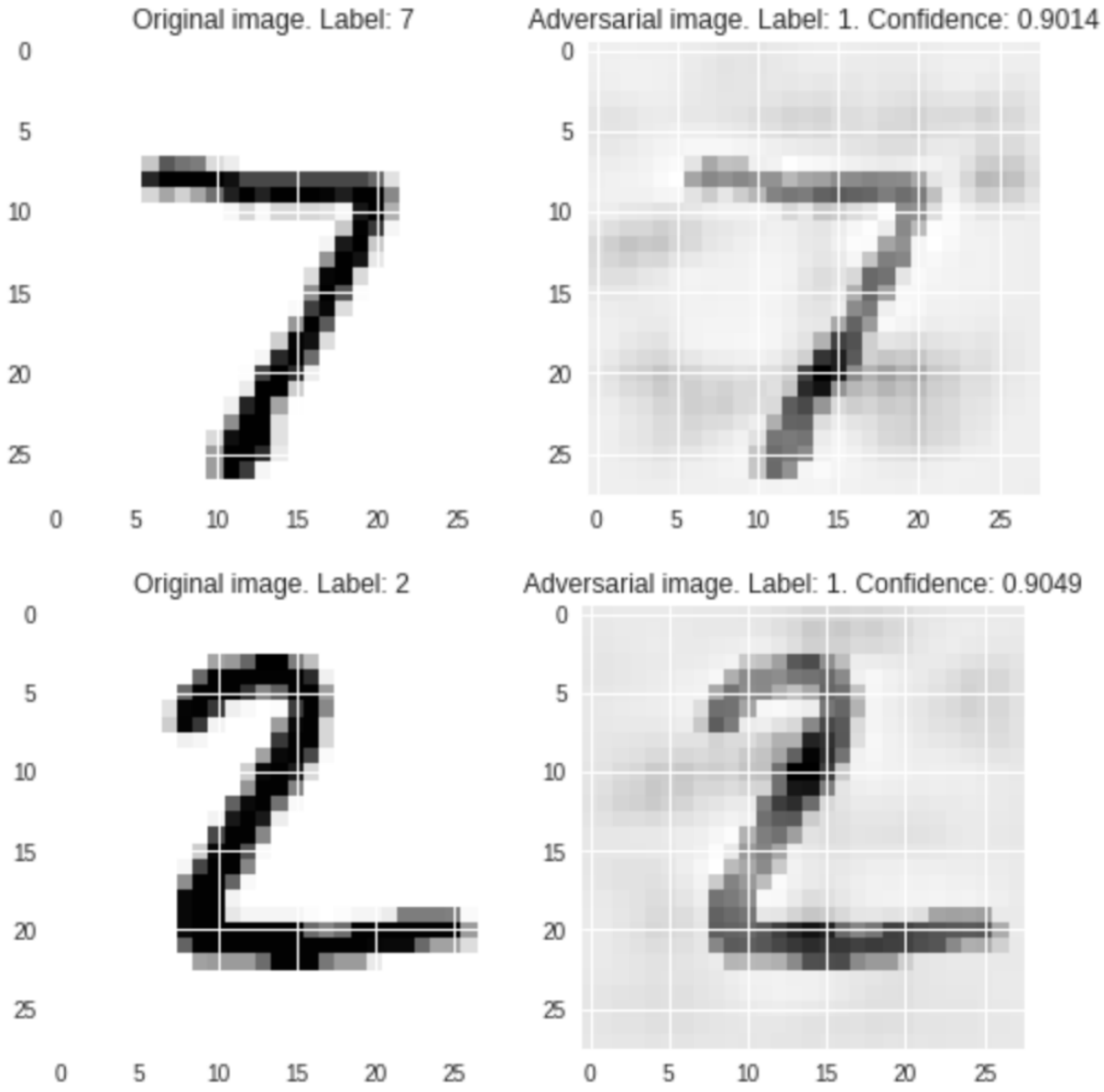
Question 3

a) We generate perturbed images to increase loss from the true label. The iteration stops when we attain > 0.9 confidence in a wrong label. The following images obtain:



Unlike the original images, these perturbed images are very 'noisy'. The original images are sparse (white in most places), whereas these images have a grey value most places. They do, however, resemble the original digits very strongly.

b) We generate perturbed images to minimize loss from a false label. The iteration stops when we attain > 0.9 confidence in the specified wrong label. The following images obtain:

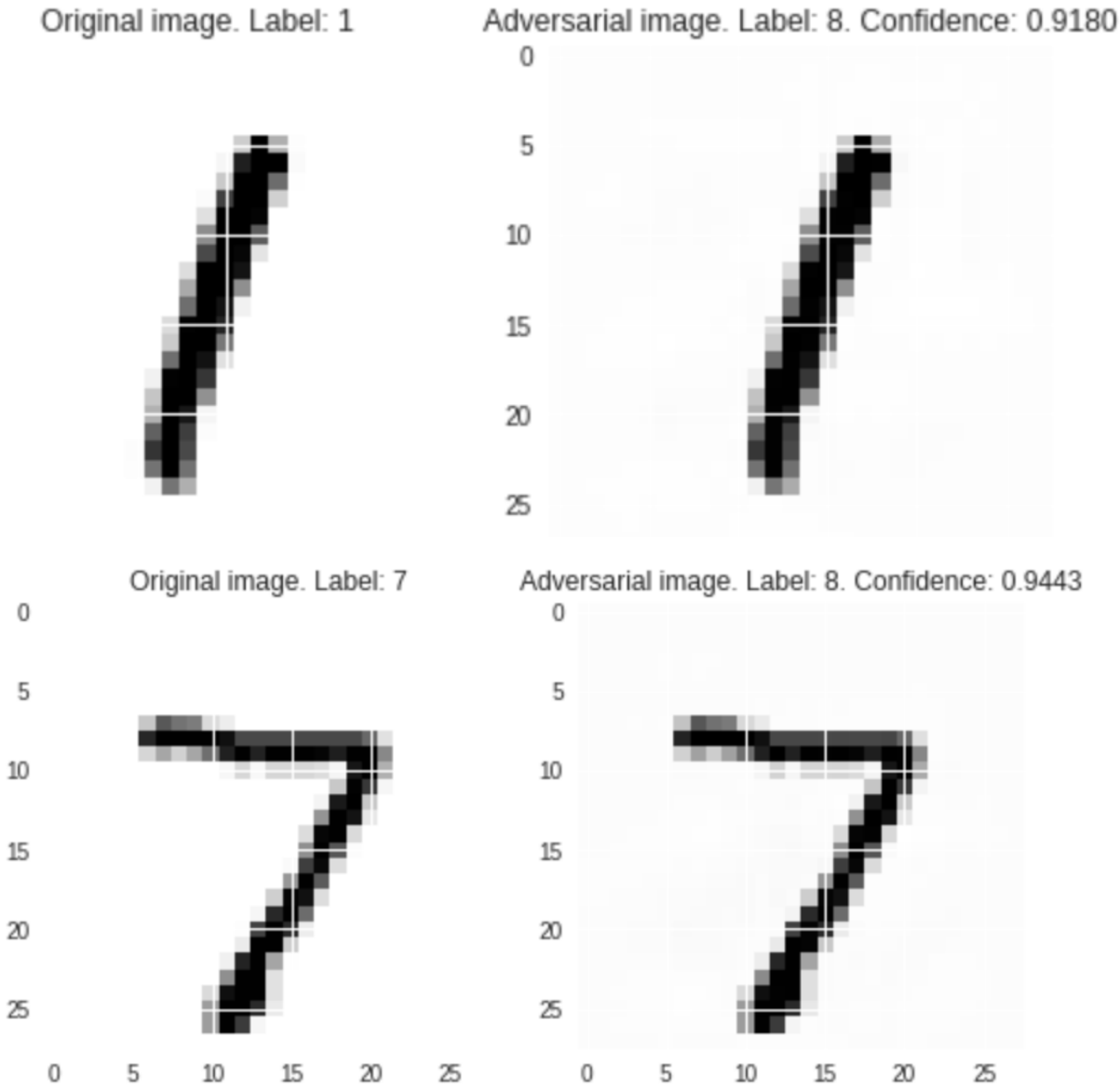


Again, they share the same characteristics as the images generated in part (a), but now we have shown that we can choose the label

c) We use the BNN and define the perturbed image as

(sign(adversarial_noise) * epsilon) + original_image

This works amazingly well!



The images are impossible to differentiate by eye, and are similarly sparse (mostly white). Yet the net has high confidence in a wrong label.

d) As mentioned in previous parts, these images fool the network generated in part 2. This is unsurprising, since we are using the network in 2 (in eval mode) to learn these adversarial images. These images have been picked for their ability to fool the network. I create 10 such adversarial images and train the network on them, but it was not good enough to beat a second round of adversarial images. It is still highly sensitive to small changes in the visual space.