Adversarial Examples in Neural Networks
Robert Lacok, MiA 2016

Abstract. We construct adversarial examples from the MNIST test set obtaining an error rate of 97% with a neural network classifier. By utilizing L1 and L2 regularizations we push the error rate to 81%.

Introduction and data

Neural networks are the state-of-the-art technology in computer vision. The best results are achieved by deep convolutional neural networks (CNNs), but even on relatively small datasets CNNs can take a long time to train. This can motivate using simpler models, such as fully connected NNs with just a few layers, which can still solve certain vision problems very well.

One such problem is the classification of handwritten digits using the MNIST dataset. The well-known dataset contains 60,000 training and 10,000 testing images (with corresponding labels) of digits from over 250 writers. The images are 28x28 pixels and normalized so that the centre of mass of each digit lies in the centre of the image. The originally binary (black & white) images were anti-aliased which introduced some levels of grey and pixel values in range [0, 255], but in general each pixel either captures ink or no-ink. Today it is commonly used for machine learning research and teaching.

The current best result is a test set error rate of 0.23% achieved by a committee of 35 deep CNNs. This is considered to be on a par with human performance, as most of the 23 misclassified examples are actually hard to read. To compare, a 3-layer fully-connected network can easily get to 1.30% error rate after just a few minutes of training.

Interestingly, it is possible to generate images with minimal difference to the originals which confuse these networks so much they obtain error rate as high as 99%, while keeping the confidence of their predictions around 80%. They are called adversarial examples and reveal a fundamental blind spot in the classifiers. To quote [1]: *"These algorithms have built a Potemkin village that works well on naturally occurring data, but is exposed as a fake when one visits points in space that do not have high probability in the data distribution."* In addition, as shown in [1], different networks trained on disjoint subsets of training data often agree on the incorrect classification of these examples, which makes them fascinating to study.

In this project we built a neural network using the Theano library in Python and trained it on the MNIST dataset. We implemented an algorithm to generate the 'fast gradient sign adversarial examples' as described in [1] and obtained comparable classification results. We tried several regularization techniques in order to build a classifier robust (to an extent) against adversarial examples without adversarial training.

Mathematics in Action

Neural Networks on MNIST. Neural networks are a family of machine learning models which approximate arbitrary functions. They do so by taking input in form of a vector and performing a series of matrix transformations followed by activation functions until finally reaching the output. In the case of MNIST the input vector has 784 (= 28^2) features representing the intensities of individual pixels and the final output is the predicted digit (0-9).

The base architecture of our neural network is as follows:

- Input layer ($784 \rightarrow 625$), regularized by dropping out random 20% of input pixels during training, with ReLU as activation function
- Hidden layer ($625 \rightarrow 625$), regularized by dropping out random 50% of input features during training, with ReLU as activation function
- Output layer ($625 \rightarrow 10$), regularized by dropping out random 50% of input features during training, with softmax activation function which outputs the predicted digit

ReLU is a popular activation function because it can be efficiently calculated and its gradients do not vanish or explode, which is a common problem with the sigmoid. It is defined as:

$$f(x) = \max(0, x)$$

Softmax is a squashing function used for output, which takes an arbitrary number of real numbers and returns values in the range [0, 1] which sum to 1. The values can be interpreted as probabilities and for class j it is defined as:

$$\sigma(x)_j = \frac{e^{x_j}}{\sum_{i=1}^{N} e^{x_i}} \text{ for } j = 1..N$$

Dropout is a regularization technique which addresses over-fitting. In each mini-batch it randomly switches off a subset of activations of specified layers which forces neurons to be more robust and rely on the whole population rather than specific units.

The weights are iteratively updated using mini-batch SGD, with batch size of 128. We used RMSProp for adaptive learning rate.[1]

$$MeanSquare(w, t) = 0.9MeanSquare(w, t-1) + 0.1\frac{\delta Cost}{\delta w}(t)^2$$

$$\Delta w(t) = \frac{LearningRate * \frac{\delta Cost}{\delta w}(t)}{\sqrt{MeanSquare(w, t)} + TinySmoothingValue}$$

[1]Interesting fact: The popularity of ReLU and the inventions of dropout and RMSProp can all be linked to Geoffrey Hinton, who is now based in Toronto but graduated from The University of Edinburgh.

Once a mini-batch forward propagates through the network, we define the cost as cross entropy between predictions and labels. An advantage of using a package like Theano is not having to derive expressions for back propagation for $\frac{\delta Cost}{\delta w}$ by hand as it finds them automatically.

Adversarial examples. These examples are specifically constructed to confuse the NNs. Let θ denote the parameters of the model (the weights of all layers), $\mathbf{x}$ be the input vector with associated label y. $CE(\theta, \mathbf{x}, y)$ is then the cross entropy cost function for an example $\mathbf{x}$. We are only interested in signs of the gradients of CE w.r.t. x, as we will set the magnitude to ϵ, for example = 0.10 or 0.25, in order to add maximum perturbation in the 'confusing' direction. An adversarial example by this method is then generated as:

$$\mathbf{x}_A = \mathbf{x} + \epsilon \text{sign}\left(\nabla_{\mathbf{x}} CE\left(\theta, \mathbf{x}, y\right)\right)$$

Results

Let us analyse the performance of the network on the original test set and 3 augmented ones. We construct two sets of adversarial examples from the test set by adding the signs of the gradients of the cost function with two values of ϵ. To show that it is not the magnitude of the noise causing the extremely high error rates we also include a test set with added random noise with a high value of ϵ. Examples of digits from all four datasets can be seen in Figure 1.

Test set	Avg. error rate	Avg. confidence
Original	0.0137	0.9931
Adversarial, ϵ = 0.10	0.5977	0.7344
Adversarial, ϵ = 0.25	0.9709	0.6283
Random noise, ϵ = 0.25	0.0241	0.9896

Table 1: Results of the network on 4 test sets.

The results displayed in Table 1 correlate with the ones obtained by the authors in [1]. On the original test set the error rate is very small and the confidence is high. Adding noise with magnitude ϵ = 0.25 in random directions causes a negligible decrease in performance. Adding adversarial perturbations with ϵ = 0.10 significantly hinders the accuracy of the classifier, and with ϵ = 0.25 the error rate is over 97%. As the digits have equal distributions in test set, this is worse than just randomly picking digits.

While Goodfellow et al. [1] reported only marginal decrease in confidence we observe quite a large drop. This is a positive sign as it enables us to threshold the predictions by confidence - it is better to have images which we know we don't know how to classify rather than confidently going for incorrect labels.

Regularization. We performed L_1 and L_2 regularizations to improve the predictions accuracy on the adversarial sets. These penalize large values of weights and thus promote "smooth", more linear

solutions. They do so by adding penalty terms (representing the L_1 and L_2 norms of the weights w of the model) to the cross entropy cost:

$$RegularizedCE = CE + \lambda_1 \sum(|w|) + \lambda_2 \sum(w.^2)$$

By choosing different values for $\lambda_{1,2}$ we can vary the sizes of the penalty terms. We performed 9 runs and the best results were obtained with values, $\lambda_1 = \lambda_2 = 0.0001$, in Table 2. Considered values were possible combinations from $\{0.001, 0.0001, 0.00001\}$. Larger penalties tended to push the error rate on the original test set to around 10%. Smaller penalties seemed to have no effect on the accuracies on the adversarial sets.

Test set	Avg. error rate	Avg. confidence
Original	0.0306	0.9644
Adversarial, $\epsilon = 0.10$	0.1986	0.8507
Adversarial, $\epsilon = 0.25$	0.8142	0.6735
Random noise, $\epsilon = 0.25$	0.0424	0.9565

Table 2: Results of the network on 4 test sets.

We observed that the error rates on the adversarial sets lowered. On the set with smaller adversarial perturbations the network correctly classified 40% more examples, on the set with larger perturbations 16% more examples.

Conclusion

We successfully constructed adversarial examples with the method outlined in [1]. We performed experiments with different regularization coefficients and noticed improvements on both adversarial test sets. While the improvement is meaningful, the error rate on both is still very high and the problem is far from solved. Further work could include experiments with different network architectures.

References

[1] Goodfellow, Ian J., Jonathon Shlens, and Christian Szegedy. "Explaining and harnessing adversarial examples." arXiv preprint arXiv:1412.6572 (2014).

[2] Radford, Alec. "Theano Tutorials." GitHub (2014). Online: *https://github.com/Newmu/Theano-Tutorials* (Accessed 7/4/2016).

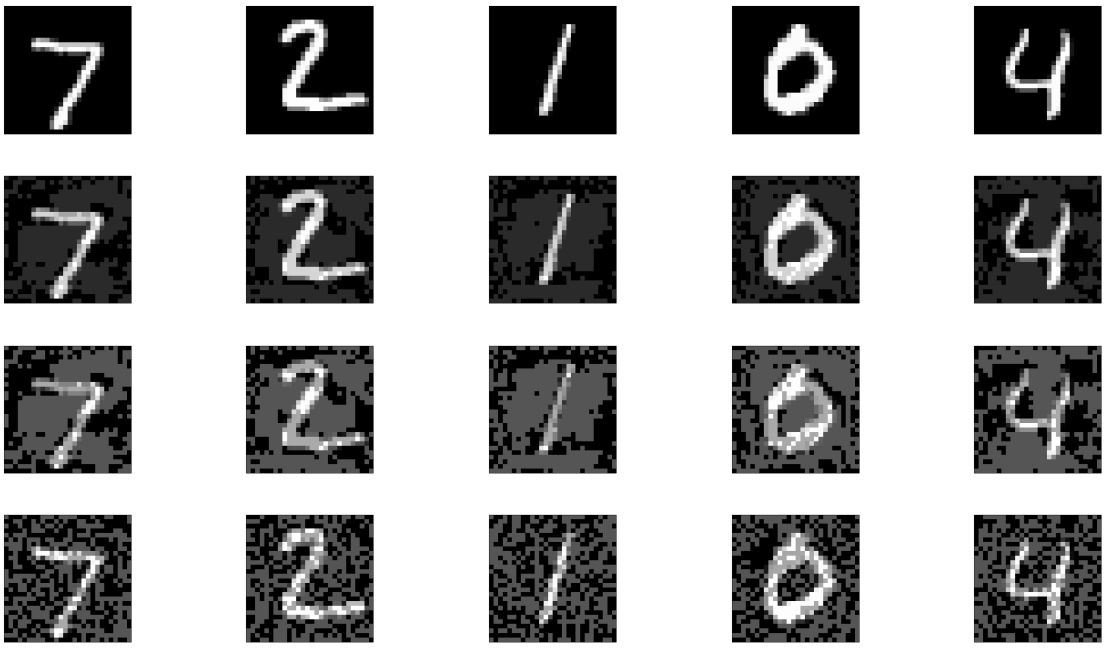

Figure 1: Examples of images from the test sets. Line 1: original test set; line 2: adversarial, $\epsilon = 0.10$; line 3: adversarial, $\epsilon = 0.25$; line 4: random noise, $\epsilon = 0.25$.

Appendix

Python code, based on Theano tutorials [2].

```
import theano
from theano import tensor as T
from theano.sandbox.rng_mrg import MRG_RandomStreams as RandomStreams
import numpy as np
from load import mnist
import cPickle
import matplotlib
matplotlib.use('Agg')
from matplotlib import pyplot as plt

srng = RandomStreams()

TRAINING = False

# Convert into correct type for theano
def floatX(X):
    return np.asarray(X, dtype=theano.config.floatX)
```

```
# Weights are shared theano variables
def init_weights(shape):
    return theano.shared(floatX(np.random.randn(*shape) * 0.01))

# RMSProp to update weights
def RMSprop(cost, params, lr=0.001, rho=0.9, epsilon=1e-6):
    grads = T.grad(cost=cost, wrt=params)
    updates = []
    for p, g in zip(params, grads):
        acc = theano.shared(p.get_value() * 0.)
        acc_new = rho * acc + (1 - rho) * g ** 2
        gradient_scaling = T.sqrt(acc_new + epsilon)
        g = g / gradient_scaling
        updates.append((acc, acc_new))
        updates.append((p, p - lr * g))
    return updates

# Dropout regularization
def dropout(X, p=0.):
    if p > 0:
        retain_prob = 1 - p
        X *= srng.binomial(X.shape, p=retain_prob, dtype=theano.config.floatX)
        X /= retain_prob
    return X

# Neural network model, 3 fully connected layers
def model(X, w_h, w_h2, w_o, p_drop_input, p_drop_hidden):
    # Input layer: dropout + relu
    X = dropout(X, p_drop_input)
    h = T.nnet.relu(T.dot(X, w_h))

    # Hidden layer: dropout + relu
    h = dropout(h, p_drop_hidden)
    h2 = T.nnet.relu(T.dot(h, w_h2))

    # Output layer: dropout + softmax
    h2 = dropout(h2, p_drop_hidden)
    py_x = T.nnet.softmax(T.dot(h2, w_o))
    return h, h2, py_x

print 'Loading MNIST data...'
trX, teX, trY, teY = mnist(onehot=True)

# Initialize theano variables for X, Y, and shared variables for weights
```

```
X = T.fmatrix()
Y = T.fmatrix()

if TRAINING:
    # For training of the net, we initialize weights to random values
    w_h = init_weights((784, 625))
    w_h2 = init_weights((625, 625))
    w_o = init_weights((625, 10))
    params = [w_h, w_h2, w_o]
else:
    # To run experiments, just read weights we learned before
    print 'Loading model...'
    with open('LearnedParams.model','rb') as fp:
        params = cPickle.load(fp)
    w_h, w_h2, w_o = params

# Dropout model for training
noise_h, noise_h2, noise_py_x = model(X, w_h, w_h2, w_o, 0.2, 0.5)
# Use all-weights model for prediction
h, h2, py_x = model(X, w_h, w_h2, w_o, 0., 0.)
y_x = T.argmax(py_x, axis=1)
# To get confidence, we use this y_x expression instead
#y_x = T.max(py_x, axis=1)

# Define cost and update theano expressions
# Add l1 and l2 regularizations for later experiments
l1 = abs(w_h).sum() + abs(w_h2).sum() + abs(w_o).sum()
l2 = (w_h**2).sum() + (w_h2**2).sum() + (w_o**2).sum()
l1coef = 0
l2coef = 0
cost = T.mean(T.nnet.categorical_crossentropy(noise_py_x, Y)) + l1coef * l1 + l2coef * l2
updates = RMSprop(cost, params, lr=0.001)

# Define train and predict theano functions
train = theano.function(inputs=[X, Y], outputs=cost,
                        updates=updates, allow_input_downcast=True)
predict = theano.function(inputs=[X], outputs=y_x, allow_input_downcast=True)


if TRAINING:
    # Train in 50 epochs
    for i in range(50):
        # Select minibatch and train
        for start, end in zip(range(0, len(trX), 128), range(128, len(trX), 128)):
            cost = train(trX[start:end], trY[start:end])
```

```
        # Show test set accuracy. Its cost is not used for optimization,
        # it is just to show progress.
        print i, ':  ', np.mean(np.argmax(teY, axis=1) == predict(teX))

        # In each step save the learned weights
        with open('LearnedParams.model','wb') as fp:
            cPickle.dump(params,fp)


#------------------------------------------------------------------------------
#
# Now we have a trained model, either loaded or trained
# Time to create adversarial examples and test them

# Theano function which calculates gradient of the cost function w.r.t. input image
cost_ad = T.mean(T.nnet.categorical_crossentropy(py_x, Y))
get_grad = theano.function(inputs=[X, Y], outputs=T.grad(cost_ad, X),
                           allow_input_downcast=True)

try:
    with open('MNIST_test_adversarial_010.bin', 'rb') as fp:
        adX = cPickle.load(fp)
except IOError:
    print 'Adversarial examples not found, generating them now...'
    # Eps is a parameter for strength of the added noise.
    # Since MNIST dataset is almost binary,
    # 0.25 epsilon should be 'bellow' the resolution, but we can
    # also try other similar values, like .10
    eps = 0.10
    adX = []
    for i in range(len(teX)):
        if (i % 1000 == 0):
            print i
        # Calculate gradients
        gs = get_grad(teX[i:i+1], teY[i:i+1]).T[:,0]
        # Add signs of gradients multiplied by epsilon to an image
        img_ad = teX[i] + eps * np.sign(gs)
        adX.append(img_ad)
    with open('MNIST_test_adversarial_010.bin', 'wb') as fp:
       cPickle.dump(adX, fp)

# Find accuracy of the classifier on the test set and adversarial set
print 'Test set:                ', np.mean(np.argmax(teY, axis=1) == predict(teX))
print 'Adversarial set, e=0.10: ', np.mean(np.argmax(teY, axis=1) == predict(adX))
```

```
# When finding confidence of predictions, we change the y_x expression above and
# display just mean confidence of all examples like this:
# print 'Test set:                ', np.mean(predict(teX))


try:
    with open('MNIST_test_adversarial_025.bin', 'rb') as fp:
        adX = cPickle.load(fp)
except IOError:
    print 'The other file not found'
print 'Adversarial set, e=0.25: ', np.mean(np.argmax(teY, axis=1) == predict(adX))

# Let us display some images from both

# Function for plotting a single MNIST image
def plot_mnist_digit(image, name):
    image = np.reshape(image, [28, 28])
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    ax.matshow(image, cmap=plt.cm.gray)
    plt.xticks(np.array([]))
    plt.yticks(np.array([]))
    plt.savefig(name)

for i in range(10):
    plot_mnist_digit(teX[i], 'test{0}.jpg'.format(i))
    plot_mnist_digit(adX[i], 'less_ad{0}.jpg'.format(i))


# Generate random noise examples
eps = 0.25
rdX = []
for i in range(len(teX)):
    # Create random noise, zeros and ones, evenly distributed
    noise = np.random.binomial(1, 0.5, size=(784))
    # Turn into -1s and 1s
    noise = 2*noise - 1
    img_rd = teX[i] + eps * noise
    rdX.append(img_rd)

for i in range(10):
    plot_mnist_digit(rdX[i], 'rd{0}.jpg'.format(i))

print 'Random noise set, e=0.25: ', 1-np.mean(np.argmax(teY, axis=1) == predict(rdX))
```