GPU-Accelerated Generation of Adversarial Examples

Joshua Smith
ECE 7720
Final Project Report

I. Introduction and Background

DEEP neural networks (DNNs) and machine learning algorithms are being used in an increasing number of safety critical applications. As this trend continues, it has become more and more important to verify properties of the networks to provide safety guarantees. As neural networks are generally characterized by extremely high dimensionality and number of parameters, verifying anything more complex than trivial properties is difficult.

Formalizing safety properties for neural networks is another challenge [1]. Given the example of an image classification network integrated with a self driving car, how does one formally describe the property that any and all the possible pictures of pedestrians must be classified as pedestrians and not anything else. In order to reduce complexity but still provide a useful and necessary formalized property, research in the field of formal verification of neural networks has centered around the idea of adversarial examples and robustness measures [2].

An adversarial example is a slightly modified input that results in dramatic or unexpected change on the output of a neural network. [2]–[4] have shown that adversarial examples exist and can be generated/crafted with relatively high precision and confidence. This fact is daunting when accounting for possible damage due to adversarial examples in safety critical systems. [2], [4] also have shown that continued training on generated adversarial examples can improve the robustness of neural networks against them. As machine learning algorithms generally require very large training sets to accurately learn complex data manifolds, the ability to rapidly generate large quantities of adversarial examples is useful to regularizing networks.

Goodfellow et. al. [2] propose an algorithm which they call the fast gradient sign method (FGSM) for generating adversarial examples. This algorithm, as shown by this work, has great potential for speedup when parallelized. The following sections document the creation and analysis of serial and parallel implementation of the FGSM algorithm. The parallel algorithm is implemented on a K80 Nvidia GPU using CUDA and the resulting speedup is presented.

II. Motivation

The following points serve as motivation for this work:

1) Szegedy et. al. [4] show that adversarial example generation is a valid method for dataset augmentation that helps to regularize networks against adversarial examples.
2) Many times, parallelized algorithms implemented on GPUs have been shown to have significant speedup over serial implementations on CPUs.
3) Machine learning algorithms normally require large amounts of data to approximate functions accurately.

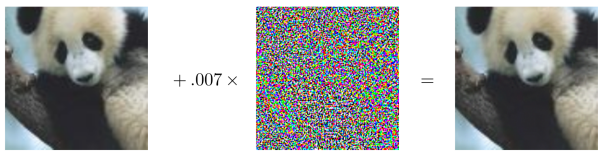
A parallelized implementation of the FGSM algorithm executed on the GPU would be a useful utility for augmenting training data to promote resistence to adversarial examples.

III. Technique

Given a network $N : x \rightarrow y$, an input x_0, a small scaling factor ϵ, trainable network weights θ, and the network cost function $J(\theta, x, y)$, the FGSM algorithm is defined as follows:

$$x_{adv} = x_0 + \epsilon sign(\nabla_{x_0} J(\theta, x, y)) \tag{1}$$

The stochastic gradient descent algorithm commonly used to train modern neural networks utilizes the gradient of the cost function with respect to the input and the weights in order to update the weights to minimize the cost function. Goodfellow et. al. [2] explain that by utilizing the gradient with repect to the tested input x_0, they are able to craft adversarial examples to which the network is particularly susceptible. The gradient is then normalized by the sign function, preserving only direction information. The sign of the gradient is then scaled by a small ϵ (normally in the range of 0.5% to 20% of the input range). As shown in Figure 1, the result of adding this small perturbation to the tested input x_0 is an attack input almost imperceptibly different. Also depicted in this figure, the crafted adversarial examples are generally misclassified with high confidence, meaning that the network is quite weak to the attack.



$\boldsymbol{x}$
"panda"
57.7% confidence

$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"nematode"
8.2% confidence

$\boldsymbol{x} + \epsilon\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"gibbon"
99.3 % confidence

Fig. 1. An adversarial example crafted using FGSM.

Each potential adversarial example produced using the FGSM algorithm has a relatively high probability of being a true adversarial example because it is crafted specifically

to trick the network. There is a possiblity though that these generated attacks do not fool the network and are correctly classified. This further motivates the rapid creation of potential adversarial examples.

By varying ϵ, one can generate many different attacks that can be used to augment the dataset and improve the network robustness.

A. Serial Implementation of FGSM

Figure 2 shows the proposed serial baseline implementation.

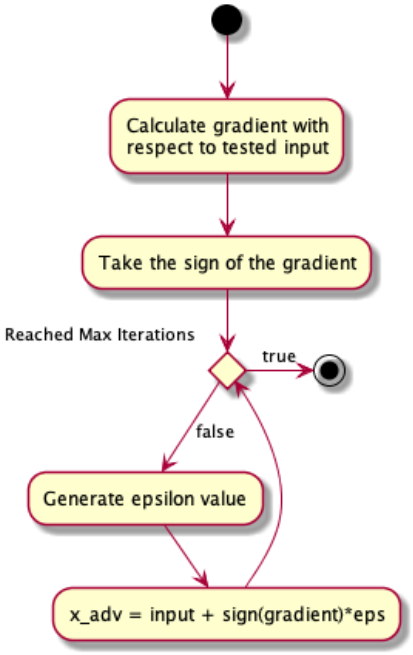


Fig. 2. Flow graph of serial FGSM.

The gradient is first calculated with respect to the tested input. The gradient does not vary until the tested input is changed and so is considered constant until the algorithm is run using a new input. The sign of the gradient is then calculated as shown in Equation 2.

$$sign(x_{ij}) = \begin{cases} -1 & x_{ij} < 0 \\ 0 & x_{ij} = 0 \\ 1 & x_{ij} > 0 \end{cases} \quad (2)$$

Then, for a user specified number of iterations, a random value is generated for ϵ, multiplied with the sign of the gradient, and added to the tested input. The set of generated attacks are collected and tested to see how many are true adversarial examples (classified differently by the network than x_0).

This algorithm was implemented using Tensorflow [5] as shown in Listing 1. As can be seen, a static graph is created that accepts tensors in the x, y, and epsilon placeholders and generates an adversarial example. The example is clipped to match the input range ([0,1] in this case) and the algorithm is repeated for *args.numgens* iterations (a user provided parameter).

Listing 1. Serial implementation of FGSM using Tensorflow

```
gradient , = tf.gradients(model['loss'], x)
epsilon = tf.placeholder(tf.float32)
optimal_perturbation = tf.multiply(tf.sign(gradient), epsilon)
adv_example_unclipped = tf.add(optimal_perturbation, x)
adv_example = tf.clip_by_value(adv_example_unclipped,0,1)

classes = tf.argmax(model['probability'],axis=1)

adv_examples = []

with tf.Session() as sess:
  for i in range(args.numgens):
      adv = adv_example.eval(
        feed_dict={
            x: x_train[idx:idx + 1],
            y: y_train[idx:idx + 1],
            epsilon: np.random.uniform(
                epsilon_range[0], epsilon_range[1],
                # size=(28, 28)
                )
        })
      class_adv = classes.eval(feed_dict={x: adv})
      if class_adv != y_train[0]:
          adv_examples += [adv]
```

This implementation is used as the baseline for evaluation of the GPU implementation discussed later. This serial implementation is based on that maintained by Ian Goodfellow (Google Brain) and Nicolas Papernot (Google Brain) in the neural network attack library CleverHans [6].

B. Parallel Implementation of FGSM

A timing analysis of the serial implemenation above revealed that almost all the time is taken up by the inner loop and the matrix operations used to calculate the adversarial example. Though there is no data dependency between generated adversarial examples or even pixels in each example, these are naively calculated in a serial fashion.

When inspected, the FGSM algorithm reveals great potential to be parallelized due to the element-wise nature of all the calculations. FGSM is essentially a random number generation, scalar-matrix multiplication, and a matrix-matrix addition, all of which can be fully parallelized.

Also, because adversarial examples have no data dependencies on eachother, all adversarial examples can be generated in parallel. Figure 3 shows the parallelized FGSM algorithm. As shown, the loop found in the serial implementation is completely eliminated.

The initial steps are the same between the two methods. The algorithms begin to diverge in the generation of epsilon values. The serial implementation generates adversarial examples inside the inner loop whereas the parallel algorithm utilizes the specialized hardware on the GPU and the cuRAND library to rapidly and simultaneously generate all the needed epsilon values. The kernel *GenerateAdvExamples* is an entry point that then spawns other kernels and handles synchronization between steps. The adversarial examples are then generated in three steps each with an associated GPU kernel:

1) Fill the allocated result array with copies of the sign of the gradient
2) Multiply each potential adversarial example by the associated ϵ
3) Add the tested input to each potential adversarial example

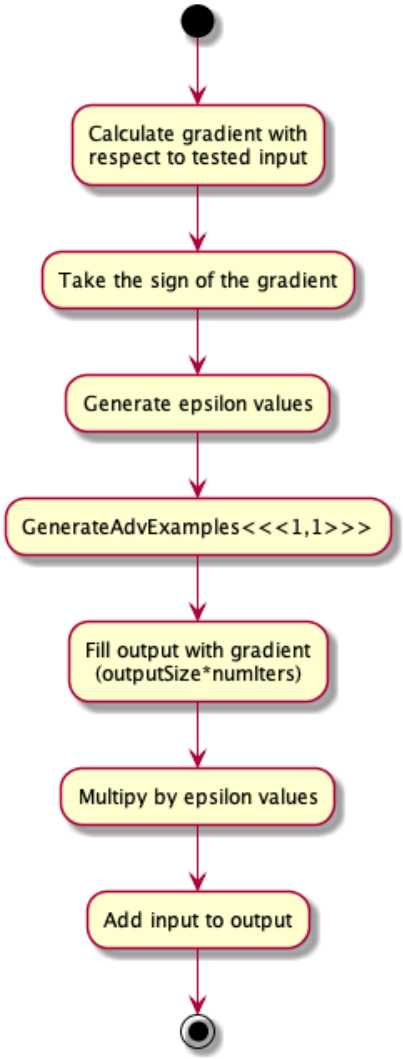


Fig. 3. Flow graph of parallel FGSM.

Every pixel in every adversarial example is calculated in parallel. Because of the massive parallelization, the expected speedup is great.

Listing 2 shows the interface between the Tensorflow model implementation and the CUDA code that generates the adversarial example. Because of the relative complexity increase, this code is a bit more involved than the serial implementation. The first noticable difference is that concrete values must be extracted from the static Tensorflow computational graph and then passed to the CUDA code rather than entirely implemented inside the graph (*grad_val*). The next difference is the code used to get and compile the CUDA source code and make it accessible to the python module. A cuRAND random number generator is then used to generate all the epsilon value on the GPU. Interestingly, due to the library pyCUDA [7], the scaling of epsilon values is also performed on the GPU. The function *gen_examples_fgsm* is then called. This is the entry point to the algorithm in charge of synchronization and dynamic parallelism.

Listing 2. Parallel implementation of FGSM using Tensorflow (kernel calling script)

```
gradient, = tf.gradients(model['loss'], x)
gradsign = tf.cast(tf.sign(gradient), tf.float32)
classes = tf.argmax(model['probability'], axis=1)
with tf.Session() as sess:
  grad_val = gradsign.eval(feed_dict={
      x: x_train[idx:idx+1],
      y: y_train[idx:idx+1]
    })
  grad_flat = np.squeeze(grad_val).flatten()
  x_flat = x_train[idx].flatten()
  with open('parallel_fgsm.cu') as f:
    src = f.read()
  src_comp = DynamicSourceModule(src)
  block = (1,1,1)
  gen_examples_fgsm = src_comp.get_function("gen_examples_fgsm")
  gen = curand.MRG32k3aRandomNumberGenerator()
  epsilon_gpu = GPUArray((args.numgens,), dtype=np.float32)
  gen.fill_uniform(epsilon_gpu)
  epsilon_gpu = epsilon_gpu * (epsmax-epsmin) + epsmin
  x_gpu = to_gpu(x_flat)
  grad_gpu = to_gpu(grad_flat)
  res_gpu = GPUArray((args.numgens*width*height,))
  gen_examples_fgsm(
      res_gpu,
      x_gpu,
      grad_gpu,
      epsilon_gpu,
      np.int32(args.numgens),
      np.int32(width*height),
      block=block
  )
  adv_examples = res_gpu.get().reshape((args.numgens,width,height))
  class_adv = classes.eval(feed_dict={x: adv_examples})
  num_adv_examples = np.sum((class_adv != y_train[idx]))
```

Listing 3 shows how each kernel is called, the launch parameters, and the synchronization that happens between each step in the algorithm. *fill_with* is responsible for filling the output with copies of the sign of the gradient, *gradsign*. *mult_vec_seg* multiplies each scalar epsilon value with the associated output. *add_vec_seg_clip* adds the tested input to each output and clips the results to the range [0,1].

Listing 3. Entry kernel responsible for synchronization and dynamic parallelism

```
__global__ void gen_examples_fgsm(
  float *res,
  float *x,
  float *gradsign,
  float *epsilon,
  int num_examples,
  int len_example
)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx == 0)
  {
    int res_len = len_example*num_examples;
    fill_with<<<res_len,1>>>(
      res,
      gradsign,
      len_example,
      num_examples);
    cudaDeviceSynchronize();
    mult_vec_seg<<<res_len,1>>>(
      res,
      epsilon,
      len_example,
      num_examples);
    cudaDeviceSynchronize();
    add_vec_seg_clip<<<res_len,1>>>(
      res,
      x,
      len_example,
      num_examples,
      0.0,
      1.0);
  }
}
```

Each of the listings below (4, 5, and 6) show the individual implemenation of each of the kernels called by

gen_examples_fgsm. All follow a similar template. Each is an operation between the output array and either *gradsign*, *epsilon*, or *x*. The basic operation is:

1) Find the index of the associated output element.
2) Find the index of the associated input element.
3) Execute the operation between the elements pointed two by these two elements and save the result in the output.

Listing 4. Kernel responsible for filling output array with copies of the sign of the gradient

```
__global__ void fill_with(
  float *res,
  float *gradsign,
  int len,
  int num_fill
)
{
  int resIdx = blockDim.x * blockIdx.x + threadIdx.x;
  int gradIdx = resIdx % len;
  if (resIdx < num_fill * len)
  {
    res[resIdx] = gradsign[gradIdx];
  }
}
```

Listing 5. Kernel responsible for the scalar-matrix multiplication between each sign gradient and the associated epsilon

```
__global__ void mult_vec_seg(
  float *res,
  float *epsilon,
  int len,
  int num_fill
)
{
  int resIdx = blockDim.x * blockIdx.x + threadIdx.x;
  int epsIdx = resIdx / len;
  if (resIdx < num_fill * len)
  {
    res[resIdx] *= epsilon[epsIdx];
  }
}
```

Listing 6. Kernel responsible for adding the tested input to each potential adversarial example and perform clipping

```
__global__ void add_vec_seg_clip(
  float *res,
  float *x,
  int len,
  int num_fill,
  float clip_min,
  float clip_max
)
{
  int resIdx = blockDim.x * blockIdx.x + threadIdx.x;
  int xIdx = resIdx % len;
  if (resIdx < num_fill * len)
  {
    res[resIdx] += x[xIdx];
    if(res[resIdx] < clip_min) res[resIdx] = clip_min;
    if(res[resIdx] > clip_max) res[resIdx] = clip_max;
  }
}
```

All code and neural network models for this project can be found here [1].

IV. Methodology

Generation of adversarial examples with FGSM requires a trained network model. The architecture of the model used in this project is shown in Figure 4. It is a relatively common architecture type in the image classification application. The trained model was evaluated on a test subset of the MNIST data that had not been presented during the training process and resulted in an accuracy of 99.31%.

[1]https://github.com/joshua-smith4/ECE7720Final.git

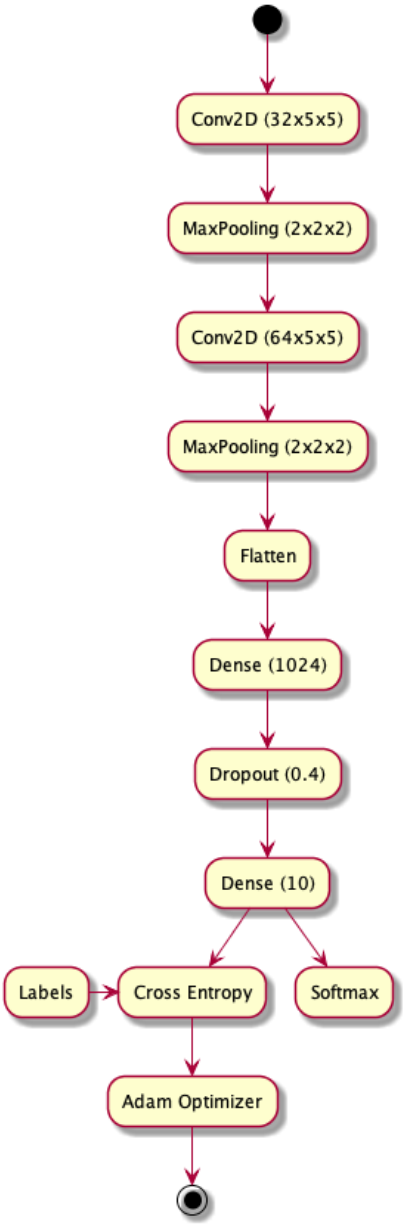


Fig. 4. Architecture description of the convolutional neural network used. The network was trained on the MNIST dataset [8]

The implementations shown and detailed above were run on hardware with the following specifications:

- **CPU**: AMD Ryzen Threadripper 1920X 12-Core Processor
- **GPU**: GeForce GTX 1060 - 6 GB
- 132 GB memory
- Nvidia driver version 410.48
- CUDA 10.0

A. Instructions to Run

The code found here is pretty simple. To reproduce the results in this paper run the following commands after all necessary dependencies have been provided:

```
/path/to/python3.6 gen_adv_ex_fgsm_serial.py --numgens=[iterations]
/path/to/python3.6 gen_adv_ex_fgsm_parallel.py --numgens=[iterations]
```

Attacks Generated	# Successful Attacks	Duration (s)
1	1	0.026
10	7	0.081
100	59	0.640
1000	632	5.726
5000	3211	29.292
10000	6410	58.990
50000	32223	293.392

TABLE I
RESULTS FOR SERIAL IMPLEMENTATION OF FGSM ALGORITHM.

Attacks Generated	# Successful Attacks	Duration (s)
1	1	0.349
10	4	0.366
100	67	0.375
1000	665	0.557
5000	3215	1.172
10000	6459	1.629
50000	32129	5.496

TABLE II
RESULTS FOR PARALLEL IMPLEMENTATION OF FGSM ALGORITHM.

V. RESULTS

The results gathered support the assumptions and motivations discussed previously and are very promising. Table V shows how the execution time of the serial implementation scales with the number of generated attacks. It can be seen that the serial implementation does not scale well to large numbers of generated attacks. Also, as a side note, the percentage of successfully generated adversarial examples can be seen from tables V and V.

Table V shows how the GPU implementation scales to larger numbers of generated attacks. As can be seen and as expected, the GPU implementation handles the high input sizes much better than the serial implementation and scales extremely well.

Figure 5 shows the results from the two tables above side by side. The GPU version scales much better to high input sizes. It should be noted that the serial implementation performs significantly better with small numbers of generated attacks (1 and 10). It is believed that this is due to the memory copy and GPU synchronization overhead of the GPU implementation.

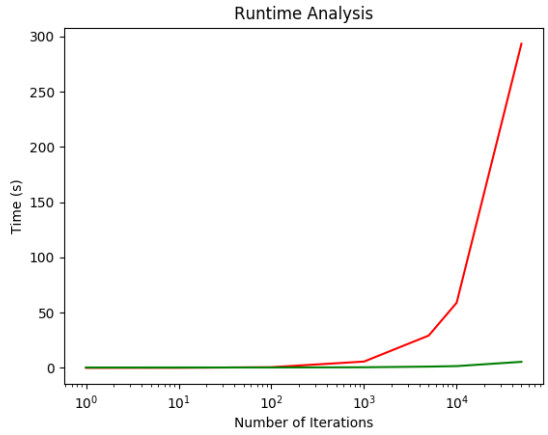


Fig. 5. Analysis of runtime of the two implementations: Serial (red), Parallel (green).

Figure 6 shows the speedup factor $\frac{T_{cpu}}{T_{gpu}}$. Speedup becomes extremely apparent and significant when iterations exceed 100.

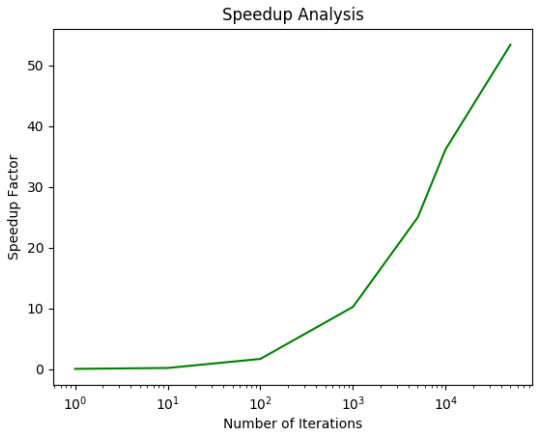
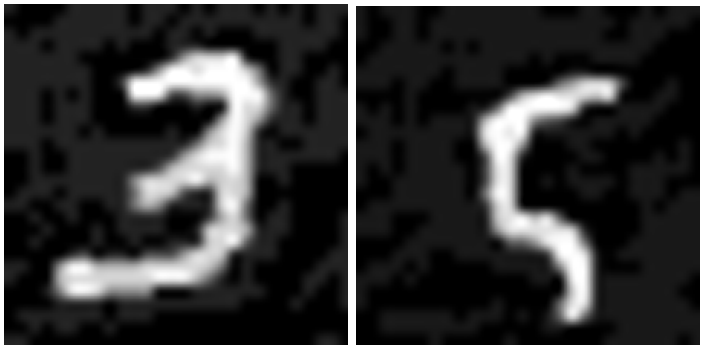


Fig. 6. Speedup analysis: $\frac{T_{cpu}}{T_{gpu}}$

Figure 7 shows several adversarial examples generated by the FGSM algorithms implemented in this paper.

(a) Misclassified as a 4 (b) Misclassified as a 7



(c) Misclassified as a 7 (d) Misclassified as an 8

Fig. 7. Several adversarial examples generated using the implemented FGSM algorithm.

VI. FUTURE WORK

This work is part of a larger formal verification framework for neural networks shown in Figure 8. The proposed framework is an abstraction refinement approach to the formal verification of adversarial robustness. The second step in the

framework generates a set of adversarial examples and then uses them to eliminate portions of the input space to which they pertain as unsafe.

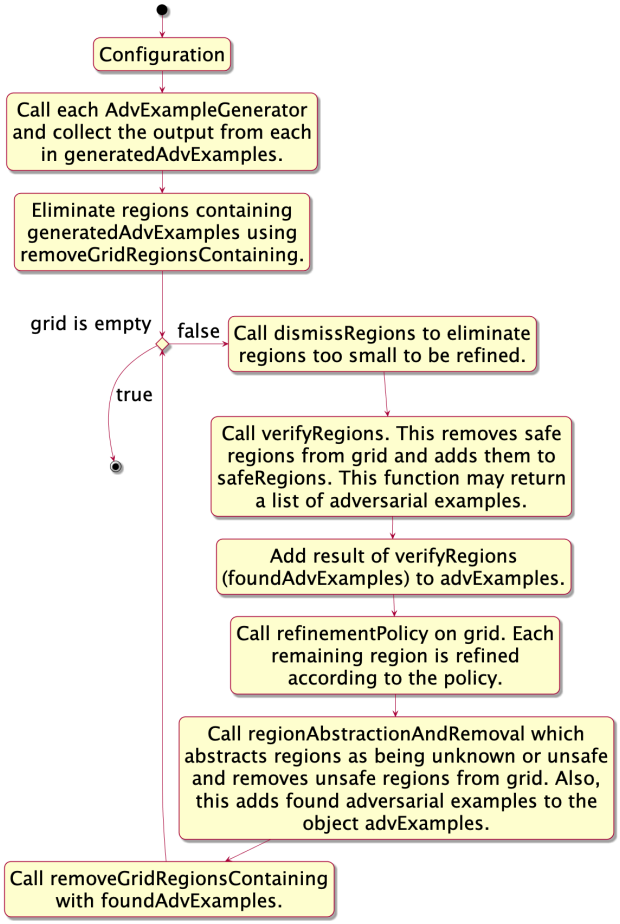


Fig. 8. Abstraction refinement formal verification framework for neural networks.

Many researchers have worked on techniques to generate adversarial examples. Many of these techniques have high levels of exploitable parallelism but have not yet been implemented on parallel hardware. One such algorithm is called the Jacobian-based saliency map algorithm (JSMA). Some results of generating adversarial examples using this method are shown in Figure 9. This method differs from FGSM in that attacks can be crafted to target specific classes (e.g. attack specifically changing a 0 to a 1). Future work includes a parallel implemenation of this algorithm to have a greater variety of produced adversarial examples in the proposed framework in Figure 8.

VII. Conclusion

The large amount of exploitable parallelism in the FGSM algorithm makes it a great candidate for parallel implementation on the GPU. As shown in the results, incredible speedup can be acheived with the GPU implementation, enabling rapid dataset augmentation using adversarial examples that improves adversarial robustness. Increased adversarial robustness improves the safety of machine learning algorithms and helps make them more suitable for safety-critical systems.

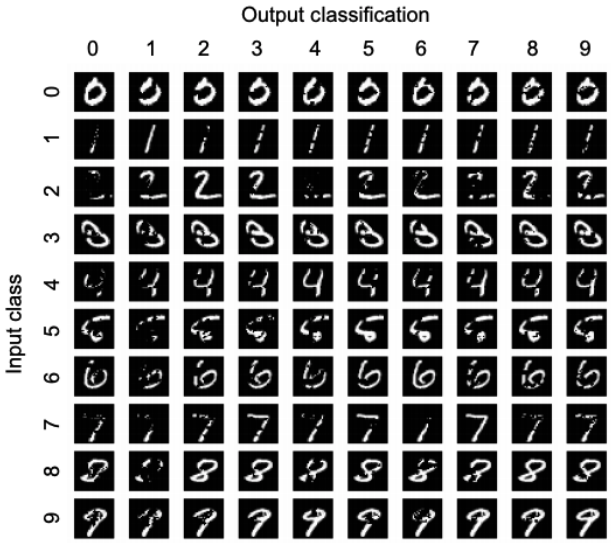


Fig. 9. MNIST results of adversarial examples generated using JSMA.

References

[1] S. A. Seshia and D. Sadigh, "Towards verified artificial intelligence," *CoRR*, vol. abs/1606.08514, 2016. [Online]. Available: http://arxiv.org/abs/1606.08514

[2] I. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *International Conference on Learning Representations*, 2015. [Online]. Available: http://arxiv.org/abs/1412.6572

[3] A. Kurakin, I. J. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," *CoRR*, vol. abs/1607.02533, 2016. [Online]. Available: http://arxiv.org/abs/1607.02533

[4] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *International Conference on Learning Representations*, 2014. [Online]. Available: http://arxiv.org/abs/1312.6199

[5] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[6] N. Papernot, F. Faghri, N. Carlini, I. Goodfellow, R. Feinman, A. Kurakin, C. Xie, Y. Sharma, T. Brown, A. Roy, A. Matyasko, V. Behzadan, K. Hambardzumyan, Z. Zhang, Y.-L. Juang, Z. Li, R. Sheatsley, A. Garg, J. Uesato, W. Gierke, Y. Dong, D. Berthelot, P. Hendricks, J. Rauber, and R. Long, "Technical report on the cleverhans v2.1.0 adversarial examples library," *arXiv preprint arXiv:1610.00768*, 2018.

[7] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation," *Parallel Comput.*, vol. 38, no. 3, pp. 157–174, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2011.09.001

[8] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/