Using MNIST and TensorFlow for image bit error tolerant testing and image bit error correction

HONGLU XU

1 INTRODUCTION

Neural network and learning algorism has been widely applied on the applications for many purposes in recent years. With the neural network learning, a lot of functions can be achieved by an artificial intelligence, such as classify an image, play chess, recognize the road symbol signs and driving a car, or even compose a piece of music or paint a picture. These all symbolled a breakthrough of artificial intelligence technology, and a lot of features are waiting to be achieved by the AI in the future.

Image correction is one of the AI features that have been discussed by the scholars in the past decades. With the development of the internet and computer vision, image correction has been a hot topic for years, and a lot of neural network learning related models and algorisms have been proposed for this feature, such as Deep NN and Convolutional NN with correction learning.

In another side, a lot of neural network learning models that implemented for other purposes already have the feature of self-correcting. This is called error tolerant. Even with some bit errors, the AI still can label an image with a high success rate, and that is truly a shining part of the neural network learning technique.

Here, for starting, we will use MNIST data [1] with some bit errors to test the error tolerant ability of three neural networks in TensorFlow [2]. After that, we will try to implement and train my own neural network for the feature error correction for the MNIST data.

2 RELATEDWORKS

Many work and discussions have been done for achieving the error correction feature by using the neural networks. Yijing and Mohammad [3] proposed a model for image data compression and error correction model. In their work, they analyzed many Artificial NN, and categorized them with the ability of self-testing and non-self-testing, and they said in their paper that hamming distance and repeat-accumulate techniques often being used for the self-testing neural networks for error correction feature. Also, still in their paper, a well implemented DNN is being proposed for correcting image bit errors. In their models, the encode and decode modules and a noisy channel will be used for image compression and error correction training, and with different compression ratio and inter neurons, the results are different. They compared all the results and conclude the answer.

In another paper from Andrew [6], he also used the MNIST for his neural network's training data. He proposed a DNN for the image bit correction technique. With the bits replaced images as the training data, he can recover the corrupted image very well. Probabilistic re-synthesis also being used in his paper as a technique for deep learning.

3 EXPERIMENTS AND RESULTS

3.1 Data Sets

In our training, data is urgently needed. Because for testing, we need to have images with bit errors as testing samples. Also, for image correction neural networks, we need these corrupted images as training data and testing data. Since for training, there need to be a great number of data pieces as training data, so that the thousands of weights and bias can be filled correctly, we need to choose the data set wisely so that a large amount of data can be obtained easily. MNIST data sets is our first choice. For training data, the data set provide 60000 samples of compressed images of unit numbers with the dimension of 28*28, and also labeled. This should be a perfect data set for many training situations about images. For testing, the data set contains 10000 samples that never appeared in the training set, with the same size, and also labeled. Thus, we've got the training data and testing data we needed, the problem now is how to make corrupted images.

By analyzing the code Alex [4] provided, and the format information that provided by the official webpage, we figured out that we could read one data in bytes of 784 (28*28), and Alex provided a very convenient way to plot the image data with NumPy. Since, there are 784 bits in one image, we used that technique to read MNIST test data and generated 5 types of corruption data that for testing, the images with 5 bits flipped, the images with 10 bits flipped, the images with 50 bits flipped, the images with 100 bits flipped and the images with 500 bits flipped. Here, for flip, since the image bits' data is ranged in 0 to 255, we implemented a method to handle the flip bits universally. For each of the bits that will be flipped, we add it with (-127) and if the result end up with less than 0, we add 255 back. Thus, all the flipped bits will differ with the original bits with 127, and all of the flipping bits will choose randomly.

$$f(x) = \begin{cases} x - 127 & (x-127>0) \\ x-127+255 & (x-127<0) \end{cases}$$

Fig. 1. Bit flip functions

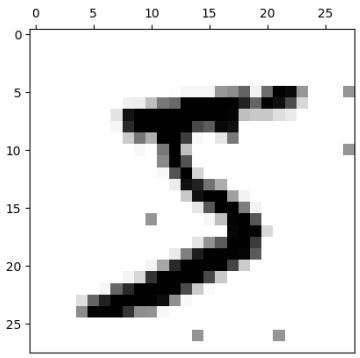

Fig. 2. A training smaple with 5-bits flipped

Figure 2 shows a result of the training set with 5-bits flipped. We can see that there are five bits randomly colored grayed where they should not. The corruption is very obvious to human, both for classification and correction, and a human being can finish the job without any difficulty, and that's why we need to see how it works when applied to an AI. For the MNIST training data, we did the same method, except with only one type, and 100 bits are flipped in all the testing data.

3.2 Error Tolerant Testing

3.2.1 Methods. For the error tolerant testing, we applied these testing data on three neural networks in TensorFlow examples [5], “mnist_softmax.py”, “fully_connected_feed.py” and “mnist_deep.py”. They all implemented by TensorFlow and all of them works well, at least not bad, in digit image classification. “mnist_softmax.py” will use the linear function and reduce mean methods for learning. “fully_connected_feed.py” will use fully connection for all the layers for learning. It will contain 2 hidden layers, with the inter-neurons of 128 and 32. All the function are also linearly, and apply softmax for the last layer. On the other hand, “mnist_deep.py” are highly implemented with deep learning and polling algorisms. Among all these neural networks, “mnist_deep.py” comes the best results in digit image classification, but others are worth testing, too. We will use two main steps for getting test results. First, we will train the neural network with the normal training data, and try to get a good score for classification, then, we will use the corrupted data for testing, see if they are different with the original results. Second, we figure out that, we trained the neural network with the corrupted data initially, we might get a better score for error toleration.

3.2.2 Results. We tested all the training data sets and testing data sets for all three neural networks, with two different training data, and here are the results.

	Original	Test 5bits	Test 10bits	Test 50bits	Test 100bits	Test 500bits	Iteration
Softmax	0.9231	0.9196	0.9205	0.9081	0.8556	0.412	6000*100
Fully Connected	0.9445	0.944	0.9453	0.9395	0.9243	0.564	10000*100
Deep	0.9911	0.9928	0.9917	0.9884	0.9819	0.7945	19900*50

Fig. 3. Error tolerant test with normal training

	Original	Test 5bits	Test 10bits	Test 50bits	Test 100bits	Test 500bits	Iteration
Softmax	0.9086	0.9182	0.9172	0.912	0.891	0.4056	6000*100
Fully Connected	0.9363	0.9408	0.9399	0.9354	0.9325	0.6645	10000*100
Deep	0.9916	0.9917	0.9893	0.9909	0.9865	0.8338	19900*50

Fig. 4. Error tolerant testing with error training

The testing results are all listed above. In Fig. 3, we used the normal training data to train these three neural networks, then, we applied 5 types of corrupted testing data to get results. The results feed what we expected. As the number of error bit increases, all of these three neural networks act worse, and the main dropping point for accuracy is at 50 bits. The simplest network, Softmax, is the worst one against the error bits. Its accuracy already dropped to 0.85 when there is nearly 1/7 of bits corrupted. On the other hand, the most complex network, Deep, seems good at error toleration. It only dropped 0.01 at point of 100 bits. However, all the neural network acts they have the ability to handle corruption situation, all of them still have a decent accuracy at 100 bits.

In the Fig 4, we have the idea that, if we trained the neural network with some error or corruption initially, it might be better at handling the error compared to the default training, and the results showing we were correct. Although most of the networks don’t act well for the accuracy with the normal testing data, but when we tried to apply some error in the testing data, the accuracy increased instead of dropping. It does make sense because we trained the network with the error image, so it

might take longer (more iteration) to get the accuracy compared to the clean training. Also, when we applied some errors in the testing data, the network seems found something it knows about, or we could say it has the skills to better recognize the data with errors, although it has never seen the image and never seen the error position. All the three networks act better on the point 100 bits, which is the number of error bits in training data, and the amount of the accuracy decrease when increasing the number of bits are less than the normal training situation. Thus, we could conclude, training by data with some errors initially could enhance the ability of error tolerant for these three neural networks.

3.3 Error Correction Implementation

For error correction, we will try to build neural networks and train it properly, so that it can function the error correction feature, which is when we pass some corrupted image to this neural network, it will calculate out the correct image. Here, we tried to build two models, first one is trying to use the Auto-associative Memory for correction, and the second one is try to use a basic linear fully connected neural network with back-propagation for correction.

3.3.1 Correction with the Auto-associative Memory. It is widely known that a piece of Auto-associative Memory act very well in error correction. We are aware of that and that's why we want to add this method in our correction network. However, it is true that the memory only works when it is dealing with something it has seen before, and that's also why it is called memory. It is used to recall and remember some data, not used for prediction and classification. Also, if there are too many irrelative piece of memories in one memory set, they will be canceled out, and nothing will be left in the memory set. We are fully aware of these features, however, since we have a great number of training data, we want to see if further training will help the memory sets to deal with the data it has never seen, and since the irrelative memory will be canceled out, what is we trained one memory set with the same label.

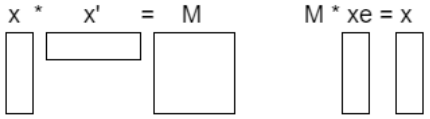

Fig. 5. Auto-associative memory algorism

Figure 5 shows the algorism of the Auto-associative Memory. In the figure, x is indicated as input data, M is memory matrix, and xe is the input data with error, so as long as we got the M matrix correct, we could use the formula M*xe to correct the corruptions. The boxes in the figure 5 indicates the size of the variables. Here, in this situation x will be the image vector which is (784,1), and x' will be (1,784). Thus, the memory matrix is (784,784). Also the xe and x will be size of (784,1). We also noticed that, it is better to keep the input data in the range of -1.0 to 1.0, so we implemented a method to scale the input vector. We apply all the integer in input vector to x = (x-127)/127. Thus, the original data (ranged from 0 to 255) will be in range -1 to 1, which is good for calculate the M matrix. Then, we will calculate the memory matrixes for each input, and sum the memory matrix only with the same label. After done a decent amount of training, we could pass the input with test data with error, and see if the memory can do the correction for the data it has never seen.

3.3.2 Memory model results. Not surprisingly, the Auto-associative Memory did very good with the data it has seen before when there are only few memories in the memory set. Also, since it acts as a memory, it will turn all the bits that is not in the memory and only stays with the bit unknown. That will result to change all the "0" to the same shape, which is not error correction. Just as figure 6 did.

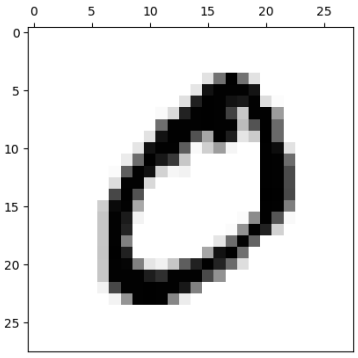

Fig. 6. Result of single piece of memory

However, as we increase the number of training, the memory set itself becomes blurring, but it appears that correction might be possible, but when we reach a large of number of trainings, the memory seems dominated everything, even if we pass an image with all black bits, as long as we said it is 0, it will generate a 0, it acts more like as a generator.

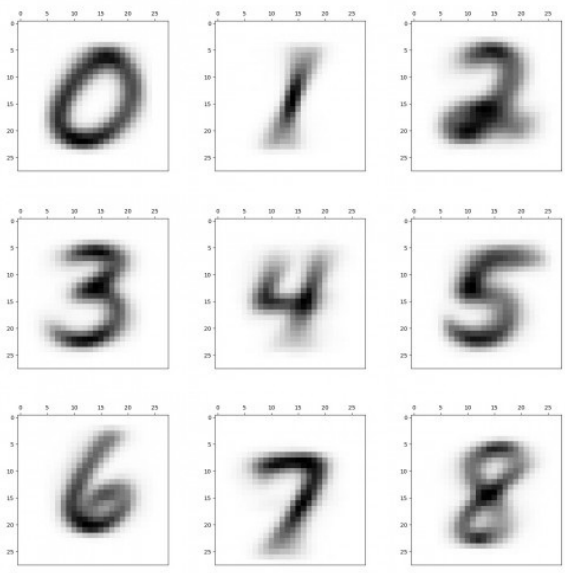

Fig. 7. Generated for 300 iteration

3.3.3 Fully connected network. We also implemented a fully connected network for learning. The fully connected network with back-propagation is always the simplest methods to implement some AI features. Though, it is simple, many basic functions still workable for this network. The idea we were thinking about is, since the theory neural network is to let the AI learn something from the

network, could we teach the AI to let it know what is an error bit, what is not. Thus, our goal is to let AI learn what kind of thing is an error bit. We will have one hidden layer with the inter-neurons of two times of the image size, which is 2*784 = 1568. The input will be training data with some bit errors, and the target is the correct image for that training data. Thus, by comparing the true image and the error image, we hope the AI could learn something to correct the error. Since it is a simple test, we used the learning function for all the layers, and a scaler at the final layer to make sure the output y is in the range of 0 to 255. After training it for a decent number of iterations, we plan to use these the saved weights and biases to generate a new image from a sample image with error. If everything works right, we hope it can correct the errors.

3.3.4 Fully connected network results. We didn't conclude results for this one. For some reason, this simple network runs really slow, and it keeps increasing its operation time for each iteration. Soon, it just will get to 10 second for each iteration and still keeps increasing. Since we have so many variables, an enormous of data set such as 50000 iterations will be needed to fully fill the variables. Thus, we didn't finish this. We got the image generated for 100 iterations, but no matter what image we passed in, it will generate the same image. See below as figure 8.

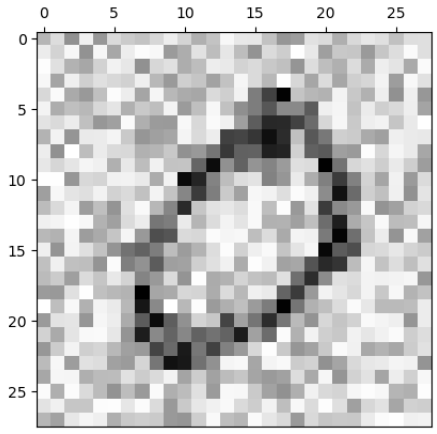

Fig. 8. Fully connected network for 100 iterations

4 Future Works

We still have many models that might work with the situation of error bit correction. Such as, if we could label the positions of all the error bit occurred, we could use these positions as targets, so that the AI could learn which bits is like an error bit, which is not. Also we could use some of the networks implemented by TensorFlow. As they already successfully did the classification, there might be some weights and layers that be useful to the error correction network, we might could combine these networks together.

REFERENCES

[1] MNIST dataset http://yann.lecun.com/exdb/mnist/

[2] TensorFlow https://www.tensorflow.org/
[3] Yijing Z. Watkins, Mohammad R. Sayeh, Image Data Compression and Noisy Channel Error Correction Using Deep Neural Network, In Procedia Computer Science, Volume 95, 2016, Pages 145-152, ISSN 1877-0509, https://doi.org/10.1016/j.procs.2016.09.305. (http://www.sciencedirect.com/science/article/pii/S1877050916324784)
Keywords: Deep neural network; image compression; artificial neural network; channel error-correction; Levenberg-Marguardt algorithm;
[4] Alex Kesling's work for reading data, https://gist.github.com/akesling/5358964
[5] TensorFlow examples source code https://github.com/tensorflow/tensorflow
[6] Simpson, Andrew J. R.. "Deep Transform: Error Correction via Probabilistic Re-Synthesis." CoRR abs/1502.04617 (2015): n. pag.