Parallel Stochastic Gradient Descent for Stacked Denoising Autoencoders

Jason Liang
jasonzliang@utexas.edu

Keith Kelly
keith@ices.utexas.edu

***Abstract*—We implement stacked denoising autoencoders, a special neural network that is capable of learning powerful representations of high dimensional data. Learning for autoencoders is accomplished via stochastic gradient descent and we show experimental results when training autoencoders on a standard image classification dataset.**

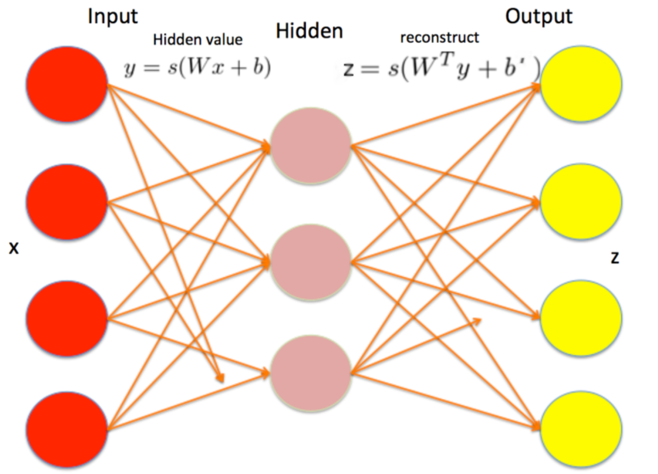

Fig. 1. Overview of an autoencoder and its encoding, decoding stages. The weight matrix of the decoding stage is the transpose of the weight matrix of the encoding stage.

I. Introduction

Autoencoders are a method for performing representation learning, an unsupervised pretraining process during which a more useful representation of the input data is automatically determined. Representation learning is important in machine learning since "the performance of machine learning methods is heavily dependent on the choice of data representation (or features) in which they are applied" [1]. For many supervised classification tasks, the high dimensionality of the input data means that the classifier requires an enormous number of training examples in order to generalize well and not overfit. One solution is to use unsupervised pretraining to learn a good representation for the input data and during actual training, transform the input examples into an easier form for the classifier to learn. Autoencoders are one such representation learning tool.

An autoencoder is a neural network with a single hidden layer and where the output layer and the input layer have the same size. Suppose that the input $x \in \mathbb{R}^m$ (and the output as well) and suppose that the hidden layer has n nodes. Then we have a weight matrix $W \in \mathbb{R}^{m \times n}$ and bias vectors b and $b^{'}$ in $\mathbb{R}^m$ and $\mathbb{R}^n$, respectively. Let $s(x) = 1/(1 + e^{-x})$ be the sigmoid (logistic) transfer function. Then we have a neural network as shown in Fig. 1. When using an autoencoder to encode data, we calculate the vector $y = s(Wx + b)$; corresponding when we use an autoencoder to decode and reconstruct back the original input, we calculate $z = s(W^T x + b^{'})$. The weight matrix of the decoding stage is the transpose of weight matrix of the encoding stage in order to reduce the number of parameters to learn. We want to optimize W, b, and $b^{'}$ so that the reconstruction is as similar to the original input as possible with respect to some loss function. In this report, the loss function used is the least squares loss: $E(t, z) = \frac{1}{2} \|t - z\|_2^2$, where t is the original input. After an autoencoder is trained, its decoding stage is discarded and the encoding stage is used to transform the training input examples as a preprocessing step. We will refer to the trained encoding stage of the autoencoder as an "autoencoder layer".

For ordinary autoencoders, we usually want that $n < m$ so that the learned representation of the input exists in a lower dimensional space than the input. This is done to ensure that the autoencoder does not learn a trivial identity transformation. However, there also exists an autoencoder variant called *denoising autoencoders* that use a different reconstruction criterion to learn overcomplete representations [2]. In other words, even if $n > m$, a denoising autoencoder can still learn a good representation of the input. This is achieved by corrupting the input image and training the autoencoder to reconstruct the original uncorrupted image. By learning how to denoise, the autoencoder is forced to understand the true structure of input data and learn a good representation of it.

We will consider training denoising autoencoders with stochastic gradient descent (SGD). Background literature for denoising autoencoders can be found in the related work section. The algorithm description section contains more details about how SGD is implemented for autoencoders. The experiments section describes the results of training an autoencoder on a handwritten digit image dataset. Finally in future work, we discuss our plans to improve upon existing training algorithms.

II. Related Work

There are several papers that we would like to mention developing the topics related to the present work. Yoshua Bengio, Aaron Courville, and Pascal Vincent review representation learning and why it is important, single layer and deep models, autoencoders, as well as other related architectures for deep learning [1]. For further elaboration on stacked autoencoders (and more specifically denoising autoencoders) we refer the reader to [2]. Hinton et al. describe in [3] how to successfully train deep multilayer networks, like the stacked autoencoder we use in this paper. The theory and methods behind the backpropagation algorithm, a variant of SGD designed for neural networks, that we use is classic and can be found in [4], [5]. Future plans include using a genetic algorithm (GA) to train the autoencoder instead of backpropagation. A survey of GAs and how they perform optimization can be found in [6]. Cantú-Paz discusses parallel GAs, which will be essential to efficiently training our autoencoder [7].

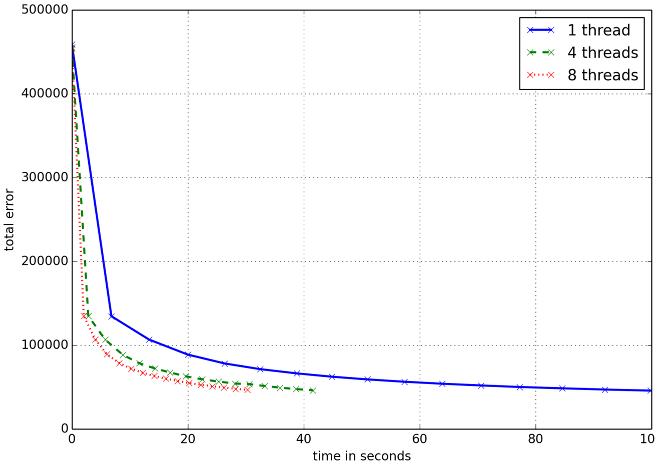

Fig. 2. Performance results on a single autoencoder layer with 500 hidden nodes and trained for 15 iterations. Plot shows time elapsed versus total training error over 5000 images for 1, 4, and 8 threads.

III. Algorithm Description

We start with a random weight matrix W and random biases b and $b^{'}$. We take a given input x, and feed it forward through the network and compute the error between the target output t and the actual output z. Often we use the squared loss error $E(t,z) = \frac{1}{2}\left\|t-z\right\|_2^2$ to determine the difference between the two. In the case of an autoencoder, the target output is the same as the input. If the error is not satisfactory, we can adjust the weight matrix and that biases in order to attempt to learn a better representation of the data. A common method of updating the weight and biases is via backpropogation [4]; when applied to training inputs one at a time, it is also known as stochastic gradient descent (SGD). We will first consider the update for the weights and biases from the last hidden layer to the output layer with a squared loss error function and derive the updates. We use as an example a simple three layer neural network (input, one hidden and output layer). Some notation is given in Table I.

The derivative of the output error E with respect to an output matrix weight W^O_{ij} is as follows.

$$\begin{aligned}\frac{\partial E}{\partial W^O_{ij}} &= \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial W^O_{ij}} \\ &= (z_j - t_j)\frac{\partial s(n_j)}{\partial x_j}\frac{\partial x_j}{\partial W^O_{ij}} \\ &= (z_j - t_j)s(n_j)(1 - s(n_j))x_i \\ &= (z_j - t_j)z_j(1 - z_j)x_i\end{aligned} \tag{1}$$

Symbol	Meaning
E	Error as computed at the output layer
x_j	Node j in the input layer
y_j	Node j in the hidden layer
z_j	Node j in the output layer
n_j	$\sum_{i=1}^{n} W_{ij}x_i + b_j$
t_j	Target output at node j
W^H_{ij}	Weight i,j from input to hidden layer
W^O_{ij}	Weight i,j from hidden to output layer
$s(x_j)$	$1/(1+e^{-x_j})$
$b^{\{H,O\}}_j$	Biases for hidden and output layer

TABLE I. TABLE GIVING NOTATION FOR THE DERIVATION OF UPDATES.

Now that we have the gradient for the error associated to a single training example, we can compute the updates.

$$\begin{aligned}\delta^O_j &= (z_j - t_j)z_j(1 - z_j) \\ W^O_{ij} &\leftarrow W^O_{ij} - \eta\delta^O_j x_i \\ b^O_j &\leftarrow b^O_j - \eta\delta^O_j\end{aligned} \tag{2}$$

The computation of the gradient for the weight matrix between hidden layers is similarly easy to compute.

$$\begin{aligned}\frac{\partial E}{\partial W^H_{ij}} &= \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial W^H_{ij}} \\ &= \left(\sum_{k=1}^{m}\frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial n_k}\frac{\partial n_k}{\partial y_j}\right)\frac{\partial y_j}{\partial n_j}\frac{\partial n_j}{\partial W^H_{ij}} \\ &= \left(\sum_{k=1}^{m}(z_k - t_k)(1 - z_k)z_k W^O_{jk}\right)y_j(1 - y_j)x_i\end{aligned} \tag{3}$$

And then using the computed gradient we can define the updates to be used for the hidden layers

$$\begin{aligned}\delta^H_j &= \left(\sum_{k=1}^{m}(z_k - t_k)(1 - z_k)z_k W^O_{jk}\right)y_j(1 - y_j) \\ W^H_{ij} &\leftarrow W^H_{ij} - \eta\delta^H_j x_i \\ b^H_j &\leftarrow b^H_j - \eta\delta^H_j\end{aligned} \tag{4}$$

In general, for a neural network we may have different output error functions and these will result in different update rules. We will also give the updates for the cross-entropy error function with softmax activation in the final layer. The cross entropy error function is given by $E(x,t) = -\sum_{i=1}^{n}(t_i \ln z_i + (1 - t_i)\ln(1 - z_i))$ and the softmax function is given by $\sigma(x_j) = e^{x_j}/(\sum_k e^{x_k})$. Following the same procedure as above for computing the gradient and the updates, we find that for hidden/output layer

$$\begin{aligned}\frac{\partial E}{\partial W^O_{ij}} &= (z_j - t_j)y_i \\ \delta^O_j &= (z_j - t_j) \\ W^O_{ij} &\leftarrow W^O_{ij} - \eta\delta^O_j x_i \\ b^O_j &\leftarrow b^O_j - \eta\delta^O_j.\end{aligned} \tag{5}$$

Note also that we find that the updates for the hidden layer is the same as in the squared error loss function with sigmoid activation. A general overview of the backpropagation algorithm is given by by Algorithm 1.

The algorithm and derivations for the autoencoder are a slight variation on the above derivations for a more general neural network. In particular, the weight matrix of the output layer (decoding stage) is the transpose of the weight matrix of the hidden layer (encoding stage). Thus $z = s(W^O(W^H x + b) + b^{'})$, $(W^H)^T = W^O$, and $W^H_{ij} = W^O_{ji}$. We note that the steps of backpropagation can only be done in sequence; thus we can only parallelize the operations done within each step. The three major operations which benefit from parallelization are computing the matrix-vector products $W^H x$ and $W^O y$, computing δ^O_j and δ^H_j, and updating the entries of the weight matrices with the gradient. For performance reasons we dont store W^O separately; instead we access W^H with transposed indexes when decoding, calculating δ^O_j, and applying the gradient update.

Once we have trained a single autoencoder layer, we can stack another autoencoder layer on top of the first one for further training. This second autoencoder takes the output of the hidden layer (encoding stage) of the first autoencoder as input and attempts the minimize the input's reconstruction cost.

Algorithm 1 Backpropogation

```
Initialize the weights and biases randomly
for iter = $1, 2, 3...$ do
    for all Examples $x$ in training set (randomize) do
        $z \leftarrow$ Feedforward $x$
        Compute output layer $\delta^O_j$
        $W_{ij} \leftarrow W_{ij} - \eta\delta^O_j x_i$
        $b_j \leftarrow b_j - \eta\delta^O_j$
        for all Layers in reverse order do
            Compute hidden layer delta $\delta^H_k$
            $W^H_{ij} \leftarrow W^H_{ij} - \eta\delta^H_j x_i$
            $b_j \leftarrow b_j - \eta\delta^H_j$
        end for
    end for
end for
```

IV. Experimental Results

A. Performance of Stochastic Gradient Descent Training

For the following experiments, we train our autoencoder over the MNIST handwritten digit dataset. The MNIST dataset is composed of 60000 training images and 10000 testing images Each image is in greyscale, is 28 by 28 pixels in size, and has a corresponding label ranging from 0 to 9. Thus, the input vector for our autoencoder has 784 dimensions. We also make use of the denoising criterion mentioned in [2], and for each training image, randomly corrupt it by setting each pixel to zero with probability 0.25.

We first analyze how the number of threads affects the rate at which training error decreases. We train a single autoencoder layer with 500 hidden nodes for 15 iterations over 5000 training images. An iteration involves going through all training images and for each image, use SGD to update the weight matrix. Fig. 2 shows the relationship between training error, total time elapsed, and the number of threads used. Regardless of the number of threads, the training error decreases sharply in the first few iterations before flattening out to around the same value after 15 iterations. The rate at which error decreases is significantly faster for four and eight threads when compared to just using one. Nonetheless, the speedup is not linear and is due to two reasons: 1) Possible cache conflicts as each thread read and writes to different locations in the weight matrix. 2) All the steps for backpropagation/SGD, described in Algorithm 1, must be done sequentially. Parallelization can only be done within each step and incurs an overhead cost.

Fig. 3 shows the time per iteration of SGD (over all 60000 training images) versus the number of threads and hidden

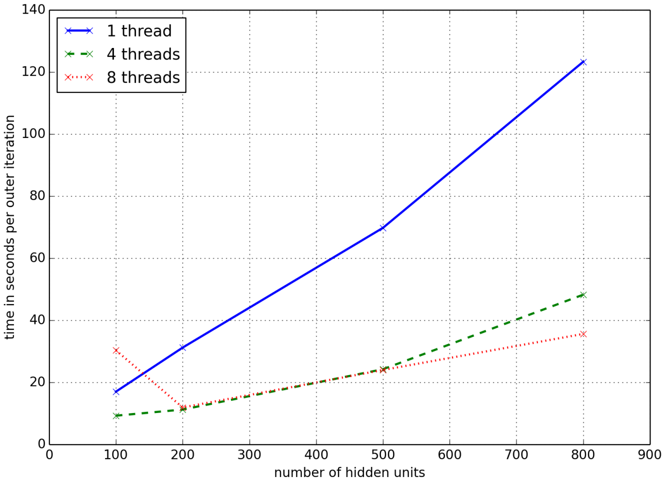

Fig. 3. Time per iteration versus number of threads and hidden nodes. All 60000 training images are used.

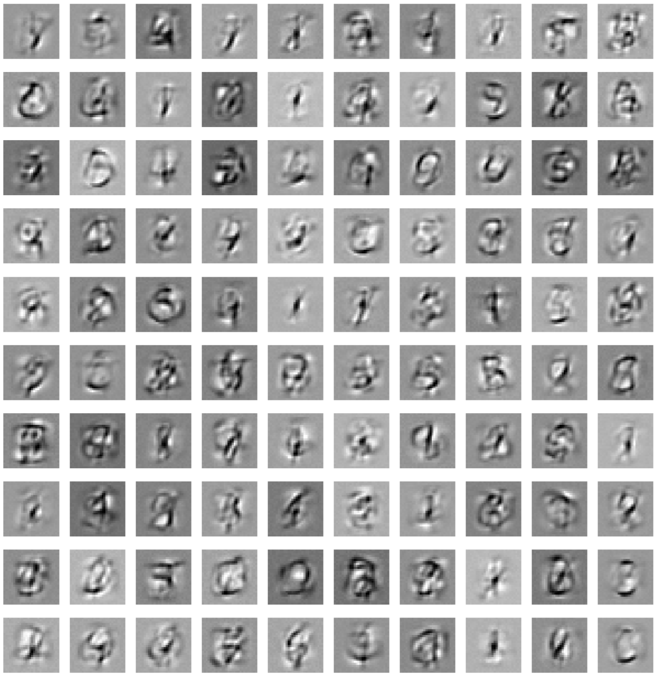

Fig. 4. Visualization of the filters of the first 100 hidden nodes in an denoising autoencoder trained over all 60000 images.

nodes in the autoencoder layer. For one thread, the speedup is essentially linear with the number of hidden nodes. For four threads and eight threads, the speedup becomes sublinear due to the overhead resulting from parallelization. This is especially noticeable when using eight threads and having lower number of hidden nodes; the time per iteration for eight threads with 100 hidden nodes is even slower than just using one thread. Using 8 threads only becomes faster than using four when the number of hidden nodes reach 800.

B. Visualization of Autoencoder

Next, in Fig. 4, we visualize the filters that are learned by training an autoencoder layer with 500 hidden nodes over all 60000 training images. The filter for each hidden node is a row vector of the weight matrix and indicates which aspects of the input the hidden unit is sensitive to. Since each row in the weight matrix is the same dimensionality as the input, we can visualize it as a 28 by 28 pixel image. The filters are not identical to the input images, but do show some similarity to them. In Fig. 4, we visualize the reconstructed digits when given noisy test digits as input. The reconstructed outputs for most of the input images are easily recognizable as digits, which indicates that the autoencoder is indeed denoising and learning a good representation of the images.

Finally we evaluate the classification accuracy of a deep neural network that has multiple stacked denoising autoencoders. We train 3 stacked autoencoder layers, each with 1000 hidden units, and using noise levels 0.1, 0.2, and 0.3 respectively. Each layer is trained for 15 iterations with a learning rate of 0.001. After the unsupervised pretraining, a conventional feedforward network with 1000 input units, 500 hidden units and 10 outputs is connected to the hidden units of the last autoencoder layer. This conventional network is then trained for 30 iterations (learning rate 0.1) in a supervised manner, where the target t is the indicator vector representation of the training label. Our final classification accuracy is 98.04%. In comparison, the accuracy achieved with a SVM with RBF kernel is 98.60% [2].

C. Representation Learning for Supervised Classification

Recall that one of the main reasons for using an autoencoder is to determine a more useful representation of the data for other tasks, for example in a classification task. To this end, we constructed and trained (15 iterations) an autoencoder with just a single layer and 1000 hidden units and used it to create a more useful representation of the digits in the MNIST dataset. After this more useful representation is constructed, we can then use the output from the autoencoder as input to another type of classification algorithm. Since the autoencoder produces a better representation of the data, we expect that given the encoded data, the other classification algorithms should perform better. The results of these experiments is given in Table. IV-C.

To test this,we used liblinear to attempt to train a model and then predict on a test set for both the enocded and unencoded datasets. With the original data liblinear gives an accuracy of 91.68% on the test set when using the default parameters. However, when the encoded data from the trained autoencoder gives an accuracy of 97.07%. This is a nontrivial improvement in the classification accuracy. Thus, the autoencoder has created a better representation of the data which maed it easier for liblinear to classify. This verifies that the autoencoder is doing what it is expected to do.

Similarly, we performed the same experiment as above, except in this case we used libsvm with an RBF kernel and all the default parameters. Without encoding the data first, we get an accuracy of 94.46%, but using the encoded data gives a prediction accuracy of 95.48%. As above, the encoded data allows libsvm better classify the data.

Using logistic regression to perform the classification, we experienced similar results. Again we use liblinear with all default options except selecting logistic regression. Using the original MNIST data, this algorithm achieved an accuracy of 91.82% while with the encoded data we achieved an accuracy of 96.86%.

	Linear SVM	Kernel SVM (RBF)	Logistic Regression
Original	91.68%	94.46%	91.82%
Encoded	97.07%	95.48%	96.86%

TABLE II. SUMMARY OF THE RESULTS OF RUNNING DIFFERENT CLASSIFCATION ALGORITHMS ON THE RAW MNIST DATA AND ON THE OUTPUT FROM A TRAINED AUTOENCODER. WE SEE IN ALL CASES THAT USING THE ENCODED DATA PRODUCES A BETTER RESULT.

D. Training the Autencoder with a Genetic Algorithm

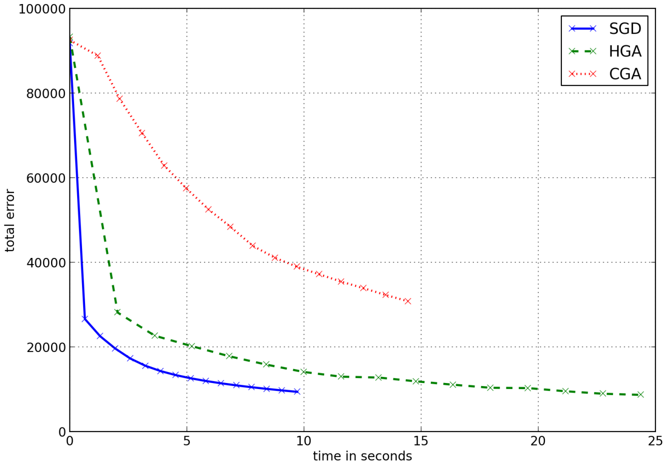

Fig. 5. Comparison of the performance of SGD, HGA, and CGA. SGD is fastest, while HGA achieves the lowest reconstruction error.

In Fig. 5, we compare the performance of stochastic gradient descent (SGD), our previously mentioned hybrid GA (HGA), which uses backpropagation to update the best individuals in the population, and a conventional GA (CGA), which does not use any gradient information. All three algorithms are used to train a single autoencoder layer with 1000 hidden units and cycle through 1000 training digits for 15 iterations. For SGD and HGA, we fix the learning rate to 0.0002. SGD is the fastest by roughly a factor of two when compared to HGA, while CGA is somewhere in between. However, HGA is able to acheive the lowest reconstruction error (8709 vs SGD's 9427). CGA performed the worst out of all 3 algorithms, having a reconstruction error that is three times larger than that of the other algorithms. The hyperparameters for HGA and CGA are hand tuned and not necessary optimal. However, we believe that wil properly tuned hyperparameters, HGA might be competitive with SGD in terms of performance and time required to go below a certain error threshold.

V. Future Work

For future work, we plan on using a genetic algorithm (GA) to train the autoencoder's weight matrix and biases. A GA is a black-box optimization algorithm that iteratively improves upon a population of candidate solution vectors until the global optima of the objective function is reached [6]. It uses operators such as mutation and crossover, which are inspired by biological evolution. GAs are superior when compared to SGD in training autoencoders in two aspects: 1) Since the loss function is highly nonconvex, SGD will always converge to a local minima, while GA are capable of eventually reaching the global optima. 2) SGD is not trivial to parallelize; as seen by our experimental results, SGD does not achieve linear speedup. On the other hand, GAs are much simplier to parallelize in one of two following ways [7]: a) Each individual in a population can be evaluated in parallel. b) The mutation and crossover operators operate on each element of a solution vector independently and thus are embarrassing parallel. An overview of how genetic algorithms work is given by Algorithm 2. For autoencoders, each individual in the population is some particular weight matrix and its two associated bias vectors. The objective function would be the loss function described earlier and the fitness is how small the error outputted by the loss function is.

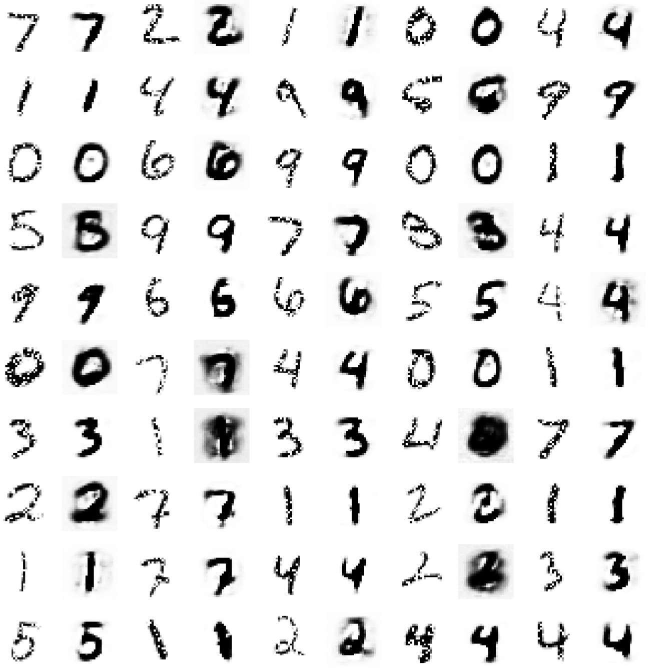

Fig. 6. Visualization of the reconstruction capabilities of an denoising autoencoder with 500 hidden units. Odd columns show noisy digit input images, even columns show reconstructed outputs.

We also plan on evaluating the performance of our autoencoder on additional harder image datasets mentioned in [2], such as *bg-rand, bg-img-rot*, which contain images with noise and rotation.

VI. Conclusion

We have implemented stacked denoising autoencoders and shown that it achieves accuracy comparable to state of the art classifiers like a SVM with RBF kernel. We also shown that our autoencoder layers are learning good representations and are capable of denoising and reconstructing the input with little error. For future work, we plan on exploring other optimization algorithms such as genetic algorithms for training autoencoders.

Algorithm 2 Genetic Algorithm

```
Initialize $N$ individuals randomly
for iter = $1, 2, 3...$ do
    Evaluate each individual with objective function and assign fitness.
    Create $\alpha N$ ($0 < \alpha < 1$) new individuals by selecting good individuals from population and applying mutation and crossover operators to them.
    Replace worst $\alpha N$ individuals in population with newly created individuals.
end for
```

References

[1] Yoshua Bengio, Aaron C. Courville, and Pascal Vincent. Unsupervised feature learning and deep learning: A review and new perspectives. *CoRR*, abs/1206.5538, 2012.

[2] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*, 11:3371–3408, 2010.

[3] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554, July 2006.

[4] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural Networks, 1989. IJCNN., International Joint Conference on*, pages 593–605. IEEE, 1989.

[5] Léon Bottou. Stochastic gradient learning in neural networks. In *Proceedings of Neuro-Nîmes 91*, Nimes, France, 1991. EC2.

[6] Mandavilli Srinivas and Lalit M Patnaik. Genetic algorithms: A survey. *Computer*, 27(6):17–26, 1994.

[7] Erick Cantú-Paz. A survey of parallel genetic algorithms. *Calculateurs paralleles, reseaux et systems repartis*, 10(2):141–171, 1998.