Parallel Stochastic Gradient Descent for Stacked Denoising Autoencoders

Jason Liang
jasonzliang@utexas.edu

Keith Kelly
keith@ices.utexas.edu

Abstract—The abstract goes here.

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 in which the output layer and the input layer have the same size, so that the original input can be used to determine how well or how poorly the autoencoder encodes the data. Consider a neural network with one hidden layer and one output layer which has the same size as the input layer. 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: $L(z) = \frac{1}{2} \|t - z\|_2^2$, 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. In this report, we will refer to the trained encoding stage of the autoencoder as an " autoencoder layer".

Usually for ordinary autoencoders, we 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.
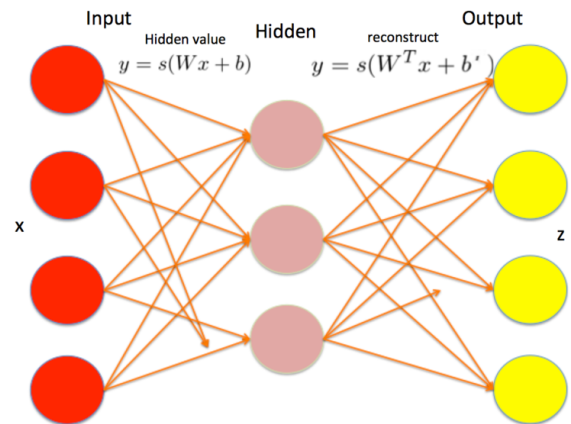

Fig. 1. Overview of an autoencoder and its encoding, decoding stages. The weight matrix of the decoding stage is the transpose of the weight matrix of the encoding stage.

II. RELATED WORK

related work here, cite like this [2]

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 $E(t, z) = \frac{1}{2} \|t - z\|_2^2$ 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 backpropagation [3]; 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 (one input, one hidden and one output node). Some notation:

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_{ij}^H	Weight i, j from input to hidden layer
W_{ij}^O	Weight i, j from hidden to output layer
$s(x_j)$	$1/(1 + e^{-x_j})$

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

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

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

$$
\begin{aligned}
\delta_j^O &= (z_j - t_j)z_j(1 - z_j) \\
W_{ij}^O &\leftarrow W_{ij} - \eta\delta_j^O x_i \\
b_j &\leftarrow b_j - \eta\delta_j
\end{aligned}
\tag{2}
$$

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

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

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

$$
\begin{aligned}
\delta_j^H &= \left(\sum_{k=1}^{m}(z_k - t_k)(1 - z_k)z_k W_{jk}^O\right)y_j(1 - y_j) \\
W_{ij}^H &\leftarrow W_{ij}^H - \eta\delta_j^H x_i \\
b_j &\leftarrow b_j - \eta\delta_j^H
\end{aligned}
\tag{4}
$$

In general, for a neural network we may have different output error function 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

$$
\begin{aligned}
\frac{\partial E}{\partial W_{ij}^O} &= (z_j - t_j)y_i \\
\delta_j^O &= (z_j - t_j) \\
W_{ij}^O &\leftarrow W_{ij} - \eta\delta_j^O x_i \\
b_j &\leftarrow b_j - \eta\delta_j.
\end{aligned}
\tag{5}
$$

Note also that we find that the updates for the input/hidden layer is the same as in the squared error loss function with sigmoid activation.

The algorithm and derivations for the autoencoder (a special type of neural network) are a slight variation on the above derivations for a more general neural network.

We give the whole algorithm in algorithm 1.

Algorithm 1 Backpropagation

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 δ_j^O
    $W_{ij} \leftarrow W_{ij} - \eta\delta_j^O x_i$
    $b_j \leftarrow b_j - \eta\delta_j^O$
    for all Layers in reverse order **do**
      Compute hidden layer delta δ_k^H
      $W_{ij}^H \leftarrow W_{ij}^H - \eta\delta_j^H x_i$
      $b_j \leftarrow b_j - \eta\delta_j^H$
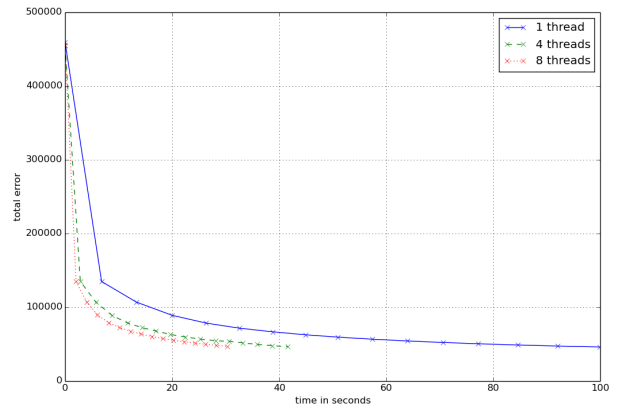   **end for**
  end for
end for

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.

IV. EXPERIMENTAL RESULTS

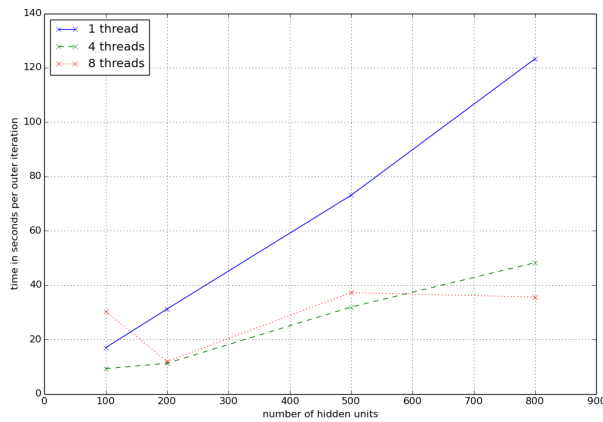For our experiments, we train our autoencoder over the MNIST handwritten digit dataset. The MNIST dataset is

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

composed of 60000 training images and 10000 testing images; each image is in greyscale, 28 by 28 pixels in size, and has a corresponding label range 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 4 and 8 threads when compared to just using one. Nonetheless, the speedup is not linear and is due to 2 reasons: 1) Possible cache conflicts as each thread read and writes to different locations in the weight matrix. 2) Certain steps of backpropagation/SGD must be done sequentially, for example, calculating the update for W^T must be done before doing the same for W.

Fig. 3, shows the time per iteration of SGD over all 60000 training images versus the number of threads and hidden nodes in the autoencoder layer. For 1 thread, the speedup is essentially linear with the number of hidden nodes. For 4 threads and 8 threads, the speedup becomes sublinear due to the overhead resulting from parallelization. This is especially noticeable when using 8 threads and having lower number of hidden nodes; the time per iteration for 8 threads with 100 hidden nodes is even slower than just using 1 thread. Using 8 threads only becomes faster than using 4 when the number of hidden nodes reach 800.

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
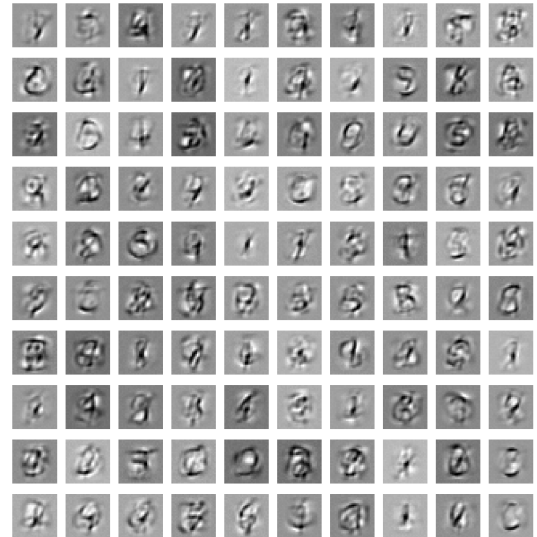

Fig. 4. Visualization of the filters of the first 100 hidden nodes in an autoencoder trained over all 60000 images.

can visualize it as a 28 by 28 pixel image. The filters will not identical to the input images, do show some similarity to them. In Fig. 4, we visualize the reconstructed output when given test images of digits as input. As we can see reconstructed output for most of the input images are easily recognizable as digits.
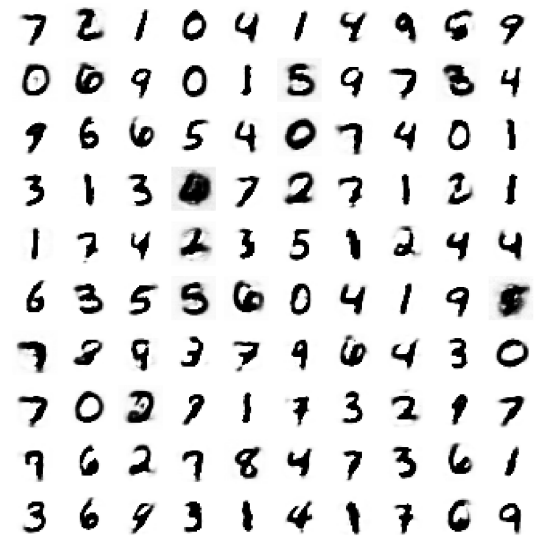

Fig. 5. Visualization of the reconstructed output when given test images of digits.

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 [4]. 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 simpler to parallelize at two levels [5]: 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.

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.

Algorithm 2 Genetic Algorithm

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

VI. CONCLUSION

We have implemented a stacked

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

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

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