Parallelizing Denoising Autoencoders

Arun Kumar, Robert Giaquinto

ABSTRACT

In recent years interest in autoencoders has emerged as a hot topic is artificial intelligence due to an explosion in data and their generative properties. Despite consistent increases in computing power, training larger neural networks has remained a challenge. One opportunity to overcoming this challenge is in using massively parallel architectures to speed up training. We find that existing serial algorithms can be adapted to parallel architectures, resulting in a performance gain exceeding 60X. Performance gains of this magnitude change what is possible in terms of capability and complexity of generative neural networks.

1. INTRODUCTION

Autoencoder neural networks are used for learning rich features from data. In recent years interest in neural networks has exploded due to the availability of larger datasets. However, even with large datasets, it's key that what models learn is generalizable – autoencoders fill this crucial gap in advanced learning from data. While supervised learning has show a lot of excitement in artificial intelligence, an ability to go beyond mapping labels to data is has significant potential[1]. Moreover, this explosion in data – while enabling the training of interesting neural networks – has made training of these networks very time consuming. In this project, we attempt to address these problems by parallelizing a denoising autoencoder.

Many approaches to reducing the training time of neural networks are algorithmic in nature. For example, stochastic gradient descent (SGD) replaces traditional gradient descent updates calculated over the entire dataset with cheap, noisy updates calculated over a minibatch of data. Our method of training an autoencoder is more than an algorithmic difference – rather, it relies on a different set of hardware.

In this paper we provide background on denoising autoencoders, describe the standard in serial training algorithms, the problems with a direct translation of the serial algorithm to a parallel architecture, a perspective of the autoencoder which lends itself to an efficient parallel implementation, and then finally we discuss the performance of our implementation.

2. BACKGROUND

2.1 Autoencoders

The autoencoder neural network is trained on a dataset where each observation consists of multiple features. Data are encoded through a hidden layer to find a dense representation of the data, and then decoded to return the original features[2]. A schema of this model is shown in Figure 1.

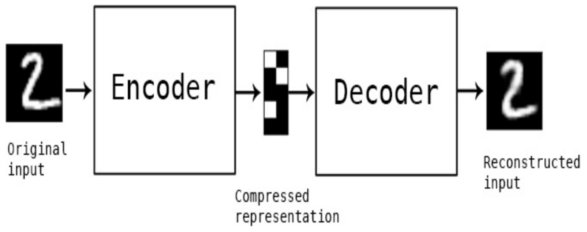


Figure 1: Autoencoder training is typically divided into two phases. In the first phase the input data is encoded, resulting in a compressed representation. In the second phase, the compressed representation is decoded resulting in a reconstruction of the original input. After sufficient learning both the learned weights and compact representation of the data employ many useful applications.

The encode step transforms the N input features of an observation $x \in \mathbb{R}^N$ using an $N \times H$ weight matrix $\mathbf{W}$, where H is the number of hidden nodes, and a bias vector $\mathbf{b}$ to from a dense representation of the data via $\mathbf{y} = s(\mathbf{W}\mathbf{x} + \mathbf{b})$, where s is the sigmoid activation function. Then, in the decode phase the compressed hidden units are transformed back into a reconstructed input $\mathbf{z} = s(\mathbf{W}^T\mathbf{y} + \mathbf{b'})$, where $\mathbf{b'}$ is a second bias vector. The goal of this process is to find optimal weights $\mathbf{W}^*$ that minimize the reconstruction error, that is the L_2 loss between the inputs and the reconstruction:

$$\mathbf{W}^* = \underset{\mathbf{W},\mathbf{b},\mathbf{b'}}{\arg\min} \frac{1}{M} \sum_{i=1}^{M} ||x_i - z_i||_2, \tag{1}$$

where the loss is computed over the entire training dataset of M observations.

In order to find weights $\mathbf{W}$ which learn useful and interesting features from the data, a common tactic is to introduce some noise to the input data points, denoted $\tilde{\mathbf{x}}$, before encoding them, and then compare the resulting reconstruction $\mathbf{z}$ to the original (un-altered) data. This approach is generally referred to as a denoising autoencoder[3]. By introducing artificial noise, the denoising autoencoder ensures that the neural network doesn't learn an identity mapping between the inputs and reconstructed outputs – but rather the network will learn to ignore noise and recognize features of the data that generalize well.

Adding noise to an input data point can vary depending on the application. For continuous features, random Gaussian noise $\mathcal{N}(0, \sigma)$ can be added to each feature, whereas for image data it's common to apply "salt and pepper" noise – essentially just turning off some pixels that were activated and vise-versa. An example of "salt and pepper" noise is demonstrated in figure 2.

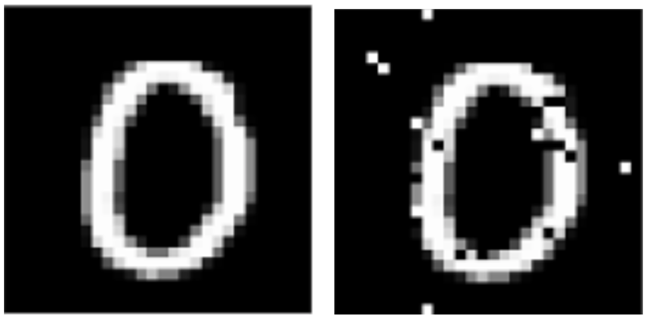

(a) Original image $\mathbf{x}$. (b) Image with salt and pepper noise $\tilde{\mathbf{x}}$.

Figure 2: Add "salt and pepper" noise to images during training of a denoising autoencoder ensures that the learned weight matrix $\mathbf{W}$ find interesting representations of the data.

2.2 Stochastic Gradient Descent Updates

Solving Equation 1 with respect to $\mathbf{W}$ yields a gradient descent algorithm. To do this, begin by combining the encode and decode phases into a single step, showing that the reconstruction can be written as a function of the inputs, that is: $f(\mathbf{x}) = s(\mathbf{b}' + g(\mathbf{x}))s(\mathbf{b}' + \mathbf{W}^T s(\mathbf{W}\tilde{\mathbf{x}} + \mathbf{b})) = \mathbf{z}$. Then, the loss function can be written as a function of a the inputs $\mathbf{x}$, that is: $\mathscr{L}(f(x)) = \frac{1}{M}\sum_{i=1}^{M}(x_i - z_i)^2$. Computing this derivative with respect to $\mathbf{W}$ requires application of the product rule. The end result is a gradient update composed of two terms, reflecting updates being back propagated through each of the two non-input layers in the network. Thus, at each iteration $t \in \{1, \ldots, T\}$ of the training algorithm, the weights are updated as follows:

$$\mathbf{W}_{i,j}^{(t)} \leftarrow \mathbf{W}_{i,j}^{(t-1)} + \rho * \Big((LH_i \cdot \tilde{\mathbf{x}}_j') + (LN_j \cdot \mathbf{y}_i)\Big)/M, \qquad (2)$$

where LH is the hidden bias and LN is the bias on the input nodes, ρ is a learning rate (typically set to 0.1), $i \in \{1, \ldots, H\}$, $j \in \{1, \ldots, N\}$, and t refers to the iteration number in the training process.

Stochastic gradient descent builds on the ideas behind gradient descent, with the goal of finding an update strategy resulting in faster convergence. The key idea behind stochastic gradient descent updates is that computing noisy updates to $\mathbf{W}$ on a mini-batch of data, can be computed very cheaply and result in significantly faster overall convergence. Thus, the gradient descent update in Equation 2 is computed on a random mini-batch of observations rather than for the entire dataset $\mathbf{x}$.

2.3 Applications

Generative models are particularly useful for generalizing the input features - for example, a photo image of a person might have an occluded view or the person might be wearing glasses. If we train our network only on images without an occlusion, it's likely to mis-classify images with occlusions. However, using generative methods like denoising autoencoders, we can create artificial images with different or noisy views of original image and use them as pre-training. Such pre-training is shown to enhance the learning capabilities and is a growing field in computer vision or natural language processing.

3. DESIGN OVERVIEW

We decompose the autoencoder training algorithm, presenting each of its components step by step. The original denoising autoencoders are composed of basic building blocks which are executed sequentially one observation at a time as described below.

3.1 Serial Algorithm

The serial algorithm takes the feature vector for one observation x and creates a corrupted version $\tilde{x}$, with noise added to each feature. Next, the algorithm creates the hidden representation $\mathbf{y}$ by transforming the input using weights from the previous iteration. The hidden representation is transformed again in the decoding step so that we get a reconstruction of the corrupted input, denoted $\mathbf{z}$. The algorithm's goal is to learn weights $\mathbf{W}$ such that the reconstructed input $\mathbf{z}$ is fairly close to the original input while not exactly the same. The algorithm concludes by calculating the loss (i.e. error) LN and LH, and applying the gradient updates to the parameters for weights $\mathbf{W}$ and biases $\mathbf{b}, \mathbf{b}'$. The procedure, listed in Algorithm 1, is repeated over a number of training epochs, until the loss is minimized. Since, each step builds on the previous steps, the serial algorithm is expected to be run a particular order.

Algorithm 1 Serial Algorithm trains on one observation at a time

```
N,H ← nFeats, nHidden
Input Corruption:
for $i = 1 : N$ do
    $\tilde{x}[i] \leftarrow \text{corrupt}(x[i])$
Encode - Hidden Values:
for $h = 1 : H$ do
    $y_h = s(b_h' + \sum_{n=1}^{N} W_{h,n} \cdot \tilde{x_n})$
Decode - reconstruction:
for $n = 1 : N$ do
    $z_n = s(b_n + \sum_{h=1}^{H} W_{h,n} \cdot y_h)$
calculate visible bias:
for $n = 1 : N$ do
    $LN_n = x_n - z_n$
    $b_n^{(t)} = b_n^{(t-1)} + \rho \cdot LN_n/M$
calculate hidden bias:
for $h = 1 : H$ do
    $LH_h = y_h(1 - y_h)\sum_{n=1}^{N} W_{h,n} \cdot LN_n$
    $b_h^{'(t)} = b_h^{'(t-1)} + \rho \cdot LH_h/M$
Learning update to weights:
for $h = 1 : H$ do
    for $n = 1 : N$ do
        $W_{h,n}^{(t)} = W_{h,n}^{(t-1)} + \rho(LH_h \cdot \tilde{x}_n + LN_n \cdot y_j)/M$
```

3.2 Naive Parallel Algorithm

The challenge with parallelization of the serial algorithm stems from the need to update the weight matrix $\mathbf{W}$ after each training iteration. In other words, we must incrementally improve the learned weights $\mathbf{W}$, updating them globally before further training. A straight-forward approach to parallelization of the serial algorithm is to assign each block an observation from the dataset. More blocks assigned to the device corresponds to larger, less-noisy SGD mini-batch updates. Within each block, threads remove the need for loops overs the number of features by doing the work corresponding to each of the N input features. Assigning each feature node (rather than hidden node) to a thread is a sensible choice because the number of hidden nodes is traditionally around half of the number of features. Thus by assigning each block an observation and each thread a feature of each observation, we're able to remove many of the for loops in the serial code and compute larger gradient descent mini-batch updates in parallel, all which should speed up convergence.

The second challenge with a straight-forward parallelization of the serial algorithm is that loops that remain (over the H hidden nodes) require multiple updates some of the variables. For example, to compute each hidden value y_h, each thread works to compute a portion of the dot product $\sum_{n=1}^{N} W_{h,n} \cdot x_n$ – however to combine these partial dot products into a single address y_h requires atomic operations. Note this is not the same calculation as one does in a CUDA matrix multiplication – which doesn't require atomic operations, the difference here is that each block is computing this value for a single vector (from a single observation).

The problem with a direct translation of the serial code to a parallel architecture is most obvious when scaling to larger more complex datasets. As the number of hidden nodes increases, so does the number of atomic operations needed. While this is not ideal, it can be mended by the fact that $\tilde{\mathbf{x}}, \mathbf{y}, \mathbf{z}$, as well as the visible and hidden biases and errors can be stored in shared memory. However, the update to the weight matrix $\mathbf{W}$ is also nested within a double for loop, and must be updated globally so as to be available to future training iterations. Moreover, for a mid-sized neural network the $\mathbf{W}$ matrix can easily reach a size exceeding what is possible for shared memory, thus doing most computations involving $\mathbf{W}$ in shared isn't feasible. For example, in the experiments described in this paper there are 784 input features and 500 hidden units, hence $\mathbf{W}$ would require $4B \times 784 \times 500 \approx 1.5MB$ of storage. Further, since atomic updates to addresses in globally memory consume two full write-delays, the performance of such an approach is inefficient[1].

Direct translation of the serial algorithm to a parallel architecture by removing the largest for loops proves to be inefficient. In the next section we introduce an approach that exploits elements of the underlying mathematics of the training algorithm to achieve break-through performance.

3.3 Proposed method

Following our analysis, we propose the method of parallelization illustrated in Figure 3. In autoencoders, training of each observation needs to leverage the learned weights and biases from previous observations before optimizing the weights one step further. Therefore, our proposal is modular in nature; we parallelize each component of the training algorithm and propagate the updated values forward. Each of the modules has its own parallel implementation that runs much faster while maintaining a similar accuracy. At its core, our method reorganizes the underlying mathematics of the serial algorithm to maximize the use of speedy tiled matrix multiplications[4].

SGD methods have been shown to generalize better on unseen data and converge faster on large scale learning[5], thereby, learning the underlying representations better. Thus, our method implements an SGD approach using mini-batches. Note, a batch-size of 1 is equivalent to training one observation at a time (similar to serial algorithm). For mini-batch processing, one key consideration involves learning updates to weights. There are various proposed methods like averaging gradients across a mini-batch before updating the weights. However, by maintaining intermediate matrices we're able to update the weights without an explicit need of statistical means, thereby reducing the processing load. To do this, the intermediate error calculation steps use a weight matrix that's available to the whole mini-batch. Using this minibatch method, we demonstrate

[1]Experiments showed that such an approach runs modestly better than the serial algorithm on a CPU for small neural networks, however for networks large enough to accuracy model the MNIST digits dataset the naive parallel implementation runs at approximately 10x slower than the serial version.

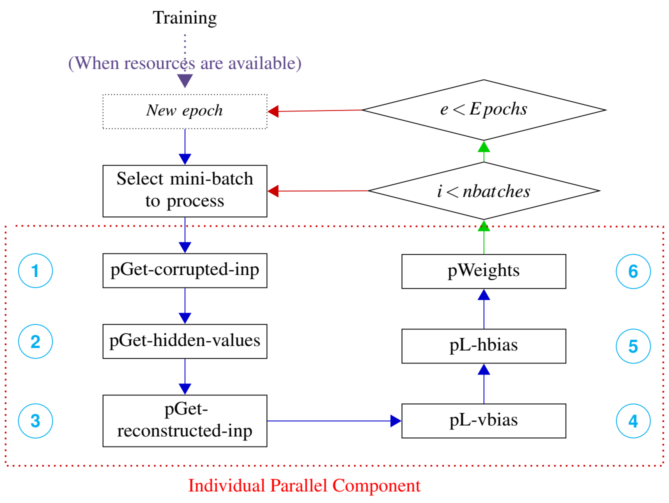


Figure 3: High level overview of the parallel training algorithm. For each epoch and for each mini-batch six separate kernels must be launched.

that our parallel algorithm is capable of learning faster and better.

4. IMPLEMENTATION

Our implementation primarily revolves around two key artifacts: First, we observe that encoding, decoding, learning weights and biases in autoencoders has an underlying computation structure that can be unrolled to make way for matrix computations. Second, we introduce SGD updates. To update the weight matrix, error calculations are made for every observation of a mini-batch – but each observation uses the same initial weight matrix. At the end, weights are updated using gradients computed over the whole mini-batch. It is important to note that, in single observation learning, weights get updated after every record but in SGD methods, updates are made at the end of every mini-batch.

Figure 3 highlights that each of the involved modules is individually parallelized, the steps of each module are illustrated in Algorithm 2.

Learning models run over a number of epochs until the loss function converges to a minimum, or for a fixed number of epochs. Before any epochs start, we allocate resources on the device for $\mathbf{x}, \tilde{\mathbf{x}}, \mathbf{y}, \mathbf{z}, LN, LH, \mathbf{b}, \mathbf{b}'$ and $\mathbf{W}$ that are required during training of each epoch. Then we transfer each of these training inputs to the device. To ensure similar results as CPU, the random initialization of $\mathbf{W}, \mathbf{b}$, and $\mathbf{b}'$ used by the gold-standard serial algorithm are copied into the devices versions. At the beginning of an epoch, the number of batches required to divide up the dataset are calculated, and the program runs for each of these mini-batches. Within a mini-batch, we invoke kernels to perform parallel operations and we discuss each of these modules in more detail here.

1. *pGet-corrupted-input:* Each thread computes a corrupted input $\tilde{x}$ for each input feature in a batch record. This operation is directly parallelizable and each thread reads an input feature $x[i], \forall i \in [1, N]$, only once and outputs the corrupted value. We launch as many threads as are the total features in input.

2. *pGet-hidden-values:* Hidden values are obtained by applying the weights from previous iteration to corrupted inputs $\tilde{x}$. $\mathbf{W}$ has dimensions $H \times N$ while $\tilde{x}$ is $batchsize \times N$. Therefore, we reformulate the problem and apply weights to input: $y =$

Algorithm 2 Parallel Denoising Autoencoder

```
Parameters: miniBatch Size, Tile Width, Epochs, Learning Rate $\rho$, Corruption Rate
Make Initial Allocations
Transfer data to device
Training
for $e = 1 : Epochs$ do {for epoch}
    for $m = 1 : miniBatches$ do {for each miniBatch}
        1. pGet-corrupted-input
        Read miniBatch $\mathbf{x}$
        $\tilde{\mathbf{x}} \leftarrow$ corrupt($\mathbf{x}$) with one thread per feature
        2. pGet-hidden-values
        Using tiled matrix multiplication parallel threads compute $\mathbf{y} = s(\mathbf{b} + \mathbf{W}\tilde{\mathbf{x}}^\top)$
        3. pGet-reconstructed-input
        Using tiled matrix multiplication parallel threads compute $\mathbf{z} = s(\mathbf{b}' + \mathbf{W}^\top \mathbf{y})$
        4. pL-vbias
        Each thread computes loss for a feature
        $LN = \mathbf{x} - \mathbf{z}^\top$
        Update feature bias $\mathbf{b}$ using $LN$ and $\rho$
        5. pL-hbias
        Tiled matrix multiplication $LH = \mathbf{W} \cdot LN^\top$
        Use $LH$, $\rho$, and $\mathbf{y}$ to update hidden bias $\mathbf{b}'$
        6. pWeights
        a thread makes $\mathbf{W}$ update for a feature and hidden unit
Evaluation: Evaluate on held out dataset
```

$Wx^\top$ facilitating parallelism for y's computation. We use two dimensional grids, blocks and set tile-width for tiled matrix multiplication. The hidden representation y has dimensions $H \times batchsize$, and we launch enough kernels such that each thread computes an element of the output using tiled matrix multiplication.

3. *pGet-reconstructed-input:* Using the hidden values y, we re-formulate the computation of reconstructed inputs as $z = W^\top y$ and add the reconstruction loss LN for each feature bias. Applying the sigmoid activation function to these values gives us the final reconstructed input z. This matrix form allows us to again use tiled matrix multiplication with two dimensional grids and blocks to compute output for an entire mini-batch in parallel.

4. *pL-vbias:* After computing the reconstructed inputs using weights and biases from the previous iteration, our goal is to use the reconstruction z to compute a measure of error. This error is propagated through the network by making a learning update to the weights and biases. Computing loss from x and z is straightforward: $LN = x - z^\top$ and done in parallel by assigning threads for each feature. After calculating the loss value, each thread makes an update to $\mathbf{b}$, weighted by the size of each gradient step ρ.

5. *pL-hbias:* Using the loss on each feature LN, and update to b' is made. To do this, first compute the hidden loss $LH = W \cdot LN^\top$ using tiled matrix multiplication. Each thread computes a LH and multiplies it by respective $y(1-y)$, as derived in gradient equations. Finally, using this intermediate value LH a learning update is made to $\mathbf{b}'$.

6. *pWeights:* Every element of $\mathbf{W}$ is computed as $W[i][j] \mathrel{+}= \rho \cdot (LH[i] \cdot \tilde{x}[j] + LN[j] \cdot y[i])/M$. While this is not a straight-forward matrix multiplication, but every hidden row and visible column element of $\mathbf{W}$ is calculated by a pass through this multiplication step. We parallelize this step by processing one hidden row per block, and one feature per thread. This approach effectively simplifies and distributes the computation workload to maximize throughput.

5. EXPERIMENT / VERIFICATION

5.1 Data

To evaluate the accuracy and efficiency of our algorithms, we experiment on the well-known MNIST hand-written digits dataset. The MNIST dataset consists of 60,000 black and white images, each has dimensions 28×28. Each image is flattened into a vector of length $28 \times 28 = 784$, thus there are 784 features for each observation. Pixel intensities for each image are scaled from $[0, 255]$ to a real valued float in $[0, 1]$. The MNIST dataset also contains 10,000 images reserved for a test set. Since our dataset is of images, we add "salt and pepper" noise as corruption to our images on approximately 30% of pixels.

5.2 Accuracy

To evaluate the accuracy of our models, we compute the L_2 loss over the entire set of observations, which corresponds to a root-mean-squared error (RMSE) calculation. In other words, $RMSE = \frac{1}{M}\sum_{i=1}^{M}(x_i - z_i)^2$ where each of the $x_i, z_i \in \mathbb{R}^{784}$. RMSE serves as a standard evaluation of continuous data, such as images, and reflects how closely the model is able to reconstruct the original image while also learning a compressed representation in the hidden nodes. Our goal is to achieve RMSEs approximately equal to those achieved by the gold-standard serial algorithm run on the CPU. However, it should be noted that due to the random noise added to corrupt each image, we do not expect the accuracy of the parallel implementation to exactly equal the accuracy of the gold-standard.

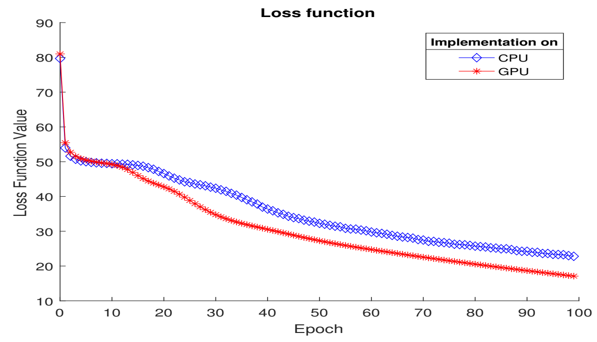
5.3 Performance

Our primary objective in developing a massively parallel version of the autoencoder training algorithm is to achieve significant improvements in the time it takes for the algorithm to converge to a stable, optimal solution. The challenge in presenting these results fairly is that the serial algorithm trains on a single observation at a time (a mini-batch of 1), whereas the parallel version of the algorithm can train on a larger mini-batch to achieve more stable gradient updates in each iteration. Thus, we present the performance of our algorithm in two ways. First, compare the relative speed-up of the parallel algorithm versus the serial version for varying sizes of the input dataset, as well as varying parameters of the model (such as tile size and mini-batch size). Secondly, we compare the accuracy in terms of total loss across all observations as a function of time. In these comparisons we seek to show that the parallel version of the model converges to an optimal solution in a shorter period of time.

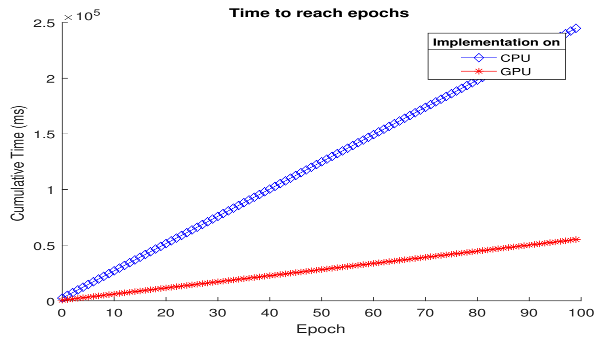
6. RESULTS AND DISCUSSION

We demonstrate results of our experiment on the MNIST hand-written digits dataset. There are two main categories of evaluation in our experiments. First, how well the parallel implementation perform with respect to a serial version in terms of speed up. And, secondly, how well the algorithm minimizes the loss between the original data and the reconstructed inputs.

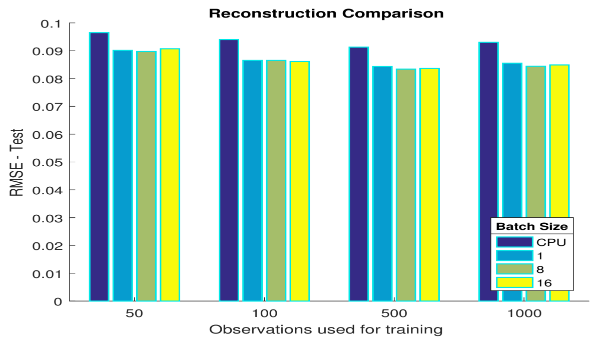
A parallel version is able to learn and minimize the loss function at a much faster speed than a corresponding serial version as demonstrated in the top left of Figure 4. Both the algorithms are



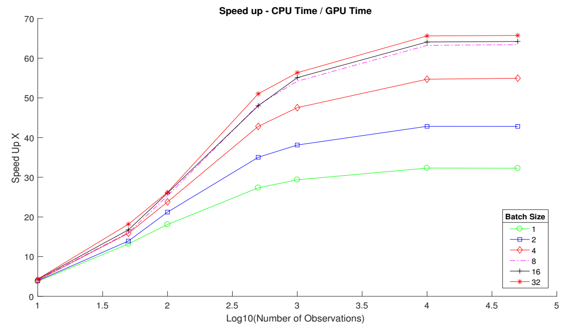
1.Loss Function



2.Parallel implementation reaches an epoch faster.



3.Reconstruction



4.Speed Up for different batch sizes.

Figure 4: Our parallel CUDA implementation demonstrates that autoencoders can be parallelized and we achieve speed ups of 60X on MNIST dataset. *Top left*. Parallel version learns underlying structure better than serial version as loss function for the two models demonstrates. *Top right*. For every epoch, the parallel version processes the dataset faster, augmenting its learning capability. *Bottom left*. Parallel implementation reconstructs inputs at a similar level of accuracy as serial version, implying that our implementation is correct for all SGD batch-sizes tested. *Bottom right*. In general the speed-up of parallel version relative to the serial version is more pronounced for larger datasets, with peak improvement exceeding 60X. Batch-size chosen for SGD can have a significant impact on performance of parallel algorithm.

run for the same number of epochs, however, the parallel version consistently produces more accurate reconstructions as measure in loss. In learning systems, this kind of performance is significant to make more viable applications and speed of processing matters. It establishes that our implementation is accurate and provides an improved learning capability.

Training capabilities in learning algorithms heavily rely on the number of epochs. Every epoch goes through a full cycle of observations to make an update to the learning parameters like weights and biases. Thus, the speed at which a learning algorithm covers epochs shows its learning potential. Figure 4 (top right) demonstrates that our parallel version is able to reach epochs faster which reinforces its value.

Next, to evaluate the accuracy of our models, we evaluate how closely our model reconstructs an image with respect to original image values and compare it to the serial version. As shown in the bottom left of Figure 4, we achieve a similar reconstruction as achieved by the gold-standard serial algorithm run on the CPU. Due to random denoising before reconstruction, no two implementations will have the exact same reconstruction error, however, it is evident that our implementation produces better reconstruction quality than the serial version. We show the reconstruction for different batch-sizes, confirming our hypothesis that mini-batches for training do not degrade reconstruction quality.

Finally, having established that our approach maintains reconstruction quality, we demonstrate speed ups for different batch sizes in the bottom right of Figure 4. The number of input training sizes is on a log_{10} scale and speed up is time taken by GPU divided by time taken by CPU. As the number of observations increases, the parallel version continue to improve its performance compared to the serial version for all batch-sizes tested. We vary batch-sizes from 1-32 and observe that for large datasets and a batch-size of 8, a speed up of >60X is achievable. It means that if a reconstruction autoencoder takes 60 hrs to learn feature representation with acceptable quality, our parallel version can learn it in approx 1 hr and generate quality images. With recent advancements in deep learning methods, such improvements could prove beneficial to develop real time intelligent agents.

7. CONCLUSION

Autoencoders are a useful approach for learning compact representations of data. However the complexity of the neural networks required for large datasets makes training difficult. We present a parallel implementation of the autoencoder training algorithm that is fundamentally different from the standard serial approaches. Our adaption of the algorithm to a parallel architecture is able to achieve significant performance gains over the serial version while simultaneously converging to more optimal solutions.

For future work we are interested in whether using float data types (instead of double) can further increase the performance gains of our algorithm without compromising accuracy. How much does the relative performance improve when we extend it to more complex models like stacked denoising autoencoders? And finally, for very large datasets, do CUDA streams allow for a more efficient data access pipeline?

8. REFERENCES

[1] 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. 11:3371–3408, December 2010.

[2] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[3] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008.

[4] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[5] Olivier Bousquet and Leon Bottou. The tradeoffs of large scale learning, 2008.