Accelerating Feedforward Computation via Parallel Nonlinear Equation Solving

Yang Song [1] **Chenlin Meng** [1] **Renjie Liao** [2 3] **Stefano Ermon** [1]

Abstract

Feedforward computation, such as evaluating a neural network or sampling from an autoregressive model, is ubiquitous in machine learning. The sequential nature of feedforward computation, however, requires a strict order of execution and cannot be easily accelerated with parallel computing. To enable parallelization, we frame the task of feedforward computation as solving a system of nonlinear equations. We then propose to find the solution using a Jacobi or Gauss-Seidel fixed-point iteration method, as well as hybrid methods of both. Crucially, Jacobi updates operate independently on each equation and can be executed in parallel. Our method is guaranteed to give exactly the same values as the original feedforward computation with a reduced (or equal) number of parallelizable iterations, and hence reduced time given sufficient parallel computing power. Experimentally, we demonstrate the effectiveness of our approach in accelerating (i) backpropagation of RNNs, (ii) evaluation of DenseNets, and (iii) autoregressive sampling of MADE and PixelCNN++, with speedup factors between 2.1 and 26 under various settings.

1. Introduction

With sufficient parallel computing resources, we can certainly accelerate any algorithm with a parallelizable component. However, many machine learning algorithms heavily rely on a seemlingly non-parallelizable part—feedforward computation. To evaluate the output of a neural network, layers are computed one after the other in a feedforward fashion. To sample text from an autoregressive model, words are generated in sequence one by one. Because of the inherently sequential nature, it is difficult to directly perform feedforward computation in parallel—how can one output a label before any intermediate features are extracted, or generate the last word in a sentence before having seen the initial part?

At first sight, the idea of executing in parallel the various steps that comprise a feedforward computation procedure seems hopeless. Indeed, the task is clearly impossible in general. Machine learning workloads, however, have special properties that make the idea viable in some cases. First, *computations are numerical in nature*, and can tolerate small approximation errors. For example, it is acceptable if a faster method produces image samples at the cost of small imperceptible errors. Second, *computations have been learned* from data rather than designed by hand. As a result, they might involve unnecessary steps, and have dependencies between the various (sequential) stages that are weak enough to be ignored without significantly affecting the final results. Although we might not be able to explicitly characterize this structure of redundant dependencies, as long as it is present, we can design methods to take advantage of it.

Based on these insights, we propose an approach to accelerate feedforward computation with parallelism. Despite not beneficial for certain types of feedforward computation, it works well for many cases of practical interest in machine learning. Our key idea is to interpret feedforward computation as solving a triangular system of nonlinear equations, and use efficient numerical solvers to find the solution. This is advantageous because (i) many numerical equation solvers can be easily parallelized; and (ii) iterative numerical equation solvers generate a sequence of intermediate solutions of increasing quality, so we can use early stopping to trade off approximation error with computation time. In particular, we propose to find the solution of the triangular system using nonlinear Jacobi and Gauss-Seidel (GS) methods (Ortega & Rheinboldt, 1970). Crucially, Jacobi iterations update each state independently and can be naturally executed in parallel. Moreover, we show feedforward computation corresponds to GS iterations, and can be combined with Jacobi iterations to build hybrid methods that interpolate between them.

[1]Computer Science Department, Stanford University. [2]Department of Computer Science, University of Toronto. [3]Vector Institute. Correspondence to: Yang Song <yangsong@cs.stanford.edu>, Stefano Ermon <ermon@cs.stanford.edu>.

Proceedings of the 38^{th} International Conference on Machine Learning, PMLR 139, 2021.

We empirically demonstrate the effectiveness and flexibility of our proposed numerical equation solvers by showing accelerations for three representative applications: (i) the back-propagation procedure for training RNNs; (ii) the inference of neural networks like DenseNets (Huang et al., 2017); and (iii) ancestral sampling from autoregressive models such as MADE (Germain et al., 2015) and PixelCNN++ (Salimans et al., 2017). In particular, for the RNN model considered in our experiments, our new method reduces the training time by more than a factor of two. For DenseNet, our Jacobi-type methods lead to an estimated speedup factor of 2.1. For ancestral sampling from autoregressive models, we achieve 26 and 25 times speed up for MADE sampling on MNIST (LeCun & Cortes, 2010) and CIFAR-10 (Krizhevsky et al., 2009) datasets; for PixelCNN++, we achieve 6.5 and 2.1 speedup factors respecitvely. Except for DenseNets where we simulate the performance due to computational constraints and implementation difficulties, all other results are measured with wall-clock time on a single GPU. This demonstrates that our methods not only perform well in the regime of massive parallel computing resources, but also have imminent practical values easily achievable with personal hardware.

2. Background

2.1. Feedforward Computation

Consider the problem of computing, given an input $\mathbf{u}$, a sequence of *states* $\mathbf{s}_1, \mathbf{s}_2, \cdots, \mathbf{s}_T$ defined by the following recurrence relation:

$$\mathbf{s}_t = h_t(\mathbf{u}, \mathbf{s}_{1:t-1}), \quad 1 \le t \le T, \tag{1}$$

where $\{h_t\}_{t=1}^T$ are deterministic computable functions, and $\mathbf{s}_{1:t-1}$ is an abbreviation for $\mathbf{s}_1, \mathbf{s}_2, \cdots, \mathbf{s}_{t-1}$. From now on, we use $\mathbf{s}_{a:b}$ to denote $\mathbf{s}_a, \mathbf{s}_{a+1}, \cdots, \mathbf{s}_b$ where $a \le b$ and $a, b \in \mathbb{N}^+$.

Given implementations of the functions $\{h_t\}_{t=1}^T$, traditional *feedforward computation* solves this problem by sequentially evaluating and memorizing $\mathbf{s}_t$, given $\mathbf{u}$ and the previously stored states $\mathbf{s}_{1:t-1}$. Note that it cannot be naïvely parallelized across different time steps as each state $\mathbf{s}_t$ can only be obtained after we have already computed $\mathbf{s}_1, \cdots, \mathbf{s}_{t-1}$.

Feedforward computation is ubiquitous in machine learning. The following examples will appear in our experiments: (i) evaluating the output of a neural network layer by layer (neural network inference); (ii) back-propagating gradients from the loss function to weights (neural network training), and (iii) ancestral sampling from autoregressive models. For (i), $\mathbf{u}$ corresponds to the network input, and $\mathbf{s}_t$ denotes the activations of each layer; For (ii), $\mathbf{u}$ corresponds to the input and the activations stored during the forward pass, and $\mathbf{s}_t$ represents the gradient of the loss function w.r.t. each layer; For (iii), $\mathbf{u}$ is the latent state of a pseudo-random number generator, and $\mathbf{s}_t$ is the t-th dimension of the sample to be generated. See Appendix A for more detailed descriptions.

2.2. Solving Systems of Nonlinear Equations

A system of nonlinear equations has the following form

$$f_i(x_1, x_2, \cdots, x_N) = 0, \quad i = 1, 2 \cdots, N, \tag{2}$$

where $x_1, x_2, \cdots, x_N$ are unknown variables, and $f_1, f_2, \cdots, f_N$ are nonlinear functions. There are many effective numerical methods for solving systems of nonlinear equations. In this paper we mainly focus on nonlinear Jacobi and Gauss-Seidel methods, and refer to (Ortega & Rheinboldt, 1970) for an excellent introduction to the field.

2.2.1. Nonlinear Jacobi Iteration

To solve a system of equations like Eq. (2), iterative methods start from an initial guess $\mathbf{x}^0 \triangleq (x_1^0, x_2^0, \cdots, x_N^0)$ of the solution, and gradually improve it through fixed-point iterations. We let $\mathbf{x}^k = (x_1^k, x_2^k, \cdots, x_N^k)$ denote the solution obtained at the k-th iteration. Given $\mathbf{x}^k$, the nonlinear Jacobi iteration produces $\mathbf{x}^{k+1}$ by solving each univariate equation for $i = 1, 2, \cdots, N$:

$$f_i(x_1^k, \cdots, x_{i-1}^k, x_i, x_{i+1}^k, \cdots, x_N^k) = 0 \tag{3}$$

for x_i. We then set $x_i^{k+1} = x_i$ for all i. The process stops when it reaches a fixed point, or $\mathbf{x}^{k+1}$ is sufficiently similar to $\mathbf{x}^k$ as measured by the *forward difference* $\left\|\mathbf{x}^{k+1} - \mathbf{x}^k\right\| \le \epsilon$, where $\epsilon > 0$ is a tolerance threshold. Crucially, all the N univariate equations involved can be solved *in parallel* since there is no dependency among them.

2.2.2. Nonlinear Gauss-Seidel (GS) Iteration

Nonlinear Gauss-Seidel (GS) iteration is another iterative solver for systems of nonlinear equations. Similar to Eq. (3), the k-th step of nonlinear GS is to solve

$$f_i(x_1^{k+1}, \cdots, x_{i-1}^{k+1}, x_i, x_{i+1}^k, \cdots, x_N^k) = 0 \tag{4}$$

for x_i and to set $x_i^{k+1} = x_i$ for $i = 1, 2, \cdots, N$. The process stops when it reaches a fixed point, or $\left\|\mathbf{x}^{k+1} - \mathbf{x}^k\right\| \le \epsilon$. Different from Eq. (3), GS updates leverage the new solutions as soon as they are available. This creates data dependency among adjacent univariate equations and therefore requires N sequential computations to get $\mathbf{x}^{k+1}$ from $\mathbf{x}^k$. Assuming that each univariate equation of Eq. (3) and Eq. (4) takes the same time to solve, one GS iteration costs as much time as N parallel Jacobi iterations.

Albeit one GS iteration involves sweeping over all variables and costs more compute than one Jacobi iteration, it can converge faster under certain cases, *e.g.*, solving tridiagonal linear systems (Young, 2014).

3. Feedforward Computation as Equation Solving

Our main insight is to frame a feedforward computation problem as solving a system of equations. This novel perspective enables us to use iterative solvers, such as nonlinear Jacobi and Gauss-Seidel methods, to parallelize and potentially accelerate traditional feedforward computation.

3.1. Feedforward Computation Solves a Triangular System of Equations

Given input $\mathbf{u}$, the recurrence relation among states $\mathbf{s}_1, \mathbf{s}_2, \cdots, \mathbf{s}_T$ in Eq. (1) can be explicitly expressed as the following system of nonlinear equations

$$h_t(\mathbf{u}, \mathbf{s}_{1:t-1}) - \mathbf{s}_t = 0, \quad t = 1, 2, \cdots, T \tag{5}$$

We can re-write Eq. (5) as a systems of equations in the form of Eq. (2) if we let $N = T$, $x_i \triangleq \mathbf{s}_i$, and $f_i(x_1, x_2, \cdots, x_T) \triangleq h_i(\mathbf{u}_i, \mathbf{s}_{1:i-1}) - \mathbf{s}_i$, for $i = 1, \cdots, N$. One unique property of these functions is that $f_i(\cdot)$ does not depend on $x_{i+1}, \cdots, x_N$, and therefore a recurrence relation corresponds to a *triangular system* of nonlinear equations. Standard feedforward computation, as defined in Section 2.1, can be viewed as an iterative approach to solving the above triangular system of nonlinear equations.

3.2. Jacobi Iteration for Recurrence Relations

Any numerical equation solver can be employed to solve the system of nonlinear equations in Eq. (5) and if converges, should return the same values as obtained by standard feedforward computation. As an example, we can use nonlinear Jacobi iterations to solve Eq. (5), as given in Algorithm 1. Here we use $\mathbf{s}^k_{1:T}$ to denote the collection of all states at the k-th iteration, and choose $\epsilon > 0$ as a threshold for early stopping when $\left\|\mathbf{s}^k_{1:T} - \mathbf{s}^{k-1}_{1:T}\right\| \leq \epsilon$, *i.e.*, the *forward difference* of states is small.

Although the nonlinear Jacobi iteration method is not guaranteed to converge to the correct solutions for general systems of equations (Saad, 2003), it does converge for solving triangular systems. In particular, it is easy to conclude:

Proposition 1. *Algorithm 1 converges and yields the same result as standard feedforward computation in at most T parallel iterations for any initialization of $\mathbf{s}^0_{1:T}$ if $\epsilon = 0$.*

In the same vein, we can also apply nonlinear GS iterations to Eq. (5). Interestingly, running one iteration of GS is the same as performing standard feedforward computation and hence GS for triangular systems always converges in a single step, even though there is typically no convergence guarantee for more general systems of equations.

As already discussed in Section 2, Jacobi iterations can exploit parallelism better than GS. Specifically, nonlinear Jacobi can complete T iterations in parallel during which GS is only able to finish one iteration, if we assume that (i) the recurrence relation Eq. (1) can be evaluated using the same amount of time for all $t = 1, \cdots, T$, and (ii) T Jacobi updates can be done in parallel. Thus, under these assumptions, Algorithm 1 can be much faster than the standard feedforward computation if the convergence of Jacobi iterations is fast. At least in the worst case, Algorithm 1 requires only T iterations *executed in parallel*, which takes the *same wall-clock time* as one GS iteration (*i.e.*, standard feedforward computation).

Algorithm 1 Nonlinear Jacobi Iteration

Input: $\mathbf{u}$; ϵ; T
Initialize $\mathbf{s}^0_1, \mathbf{s}^0_2, \cdots, \mathbf{s}^0_T$ and set $k \leftarrow 0$
repeat
  $k \leftarrow k + 1$
  for $t = 1$ **to** T **do in parallel**
    $\mathbf{s}^k_t \leftarrow h_t(\mathbf{u}, \mathbf{s}^{k-1}_{1:t-1})$
  end for
until $k = T$ **or** $\left\|\mathbf{s}^k_{1:T} - \mathbf{s}^{k-1}_{1:T}\right\| \leq \epsilon$
return $\mathbf{s}^k_1, \mathbf{s}^k_2, \cdots, \mathbf{s}^k_T$

3.3. Hybrid Iterative Solvers

We can combine Jacobi and GS iterations to leverage advantages from both methods. The basic idea is to group states into blocks and view Eq. (5) as a system of equations over these blocks. We can blend Jacobi and GS by first applying one of them to solve for the blocks, and then use the other to solve for individual states inside each block. Depending on which method is used first, we can define two different combinations dubbed Jacobi-GS and GS-Jacobi iterations respectively.

Algorithm 2 Nonlinear Jacobi-GS Iteration

Input: $\mathbf{u}$; $\mathcal{B}_1, \mathcal{B}_2, \cdots, \mathcal{B}_M$; ϵ; T
Initialize $\mathbf{s}^0_1, \mathbf{s}^0_2, \cdots, \mathbf{s}^0_T$ and set $k \leftarrow 0$
repeat
  $k \leftarrow k + 1$
  for $i = 1$ **to** M **do in parallel**
    $[\![a, b]\!] \leftarrow \mathcal{B}_i$
    for $j \in \mathcal{B}_i$ **do**
      $\mathbf{s}^k_j \leftarrow h_j(\mathbf{u}, \mathbf{s}^{k-1}_{1:a-1}, \mathbf{s}^k_{a:j-1})$
    end for
  end for
until $k = M$ **or** $\left\|\mathbf{s}^k_{1:T} - \mathbf{s}^{k-1}_{1:T}\right\| \leq \epsilon$
return $\mathbf{s}^k_1, \mathbf{s}^k_2, \cdots, \mathbf{s}^k_T$

Suppose we use an integer interval $\mathcal{B} = [\![a, b]\!]$ to represent a block of variables $\{\mathbf{s}_a, \mathbf{s}_{a+1}, \cdots, \mathbf{s}_b\}$, and let $\{\mathcal{B}_1, \mathcal{B}_2, \cdots, \mathcal{B}_M\}$ be a set of integer intervals that partitions $[\![1, T]\!]$. We formally define Jacobi-GS in Algorithm 2,

Algorithm 3 Nonlinear GS-Jacobi Iteration

Input: $\mathbf{u}$; $\mathcal{B}_1, \mathcal{B}_2, \cdots, \mathcal{B}_M$; ϵ; T
Initialize $\mathbf{s}_1, \mathbf{s}_2, \cdots, \mathbf{s}_T$
for $i = 1$ **to** M **do**
  Initialize $\mathbf{s}_j^0$ for all $j \in \mathcal{B}_i$ and set $k \leftarrow 0$
  $[\![a, b]\!] \leftarrow \mathcal{B}_i$
  repeat
    $k \leftarrow k + 1$
    for $j \in \mathcal{B}_i$ **do in parallel**
      $\mathbf{s}_j^k \leftarrow h_j(\mathbf{u}, \mathbf{s}_{1:a-1}, \mathbf{s}_{a:j-1}^{k-1})$
    end for
  until $k = |\mathcal{B}_i|$ **or** $\left\|\mathbf{s}_{\mathcal{B}_i}^k - \mathbf{s}_{\mathcal{B}_i}^{k-1}\right\| \leq \epsilon$
  $\mathbf{s}_{\mathcal{B}_i} \leftarrow \mathbf{s}_{\mathcal{B}_i}^k$
end for
return $\mathbf{s}_1, \mathbf{s}_2, \cdots, \mathbf{s}_T$

where $\mathbf{s}_{\mathcal{B}}$ is a shorthand for $\{\mathbf{s}_i \mid i \in \mathcal{B}\}$. GS-Jacobi can be similarly defined and we provide its pseudo-code in Algorithm 3. Particularly, in Jacobi-GS (Algorithm 2), all M blocks are updated in parallel and states within each block $\mathcal{B}_i$ are updated sequentially based on the latest solutions. In GS-Jacobi (Algorithm 3), we sequentially update the M blocks based on the latest solutions of previous blocks and the states within each block $\mathcal{B}_i$ are updated in parallel.

Since Eq. (5) is a triangular system of nonlinear equations, we have the following observation:

Proposition 2. *For any initialization, Jacobi-GS (Algorithm 2) and GS-Jacobi (Algorithm 3) converge in at most M block-wise iterations and yield the same results as obtained by standard feedforward computation if $\epsilon = 0$.*

In summary, all the numerical equation solvers discussed above have guaranteed convergence in finite steps when solving our triangular systems of nonlinear equations in Eq. (5), and can thus act as valid alternatives to standard feedforward computation. Traditional asymptotic analysis of convergence rates is not applicable here, since the quotient convergence factor is undefined, and the root convergence factor is zero (per the definitions in Ortega & Rheinboldt (1970)) when methods converge in finite steps.

4. Accelerating Feedforward Computation

Below we discuss when Jacobi or hybrid methods can accelerate feedforward computation. We start with a computation model that is idealized but captures important practical aspects of Jacobi methods. The computation model assumes (i) for all $t = 1, 2, \cdots, T$, the recurrence relation Eq. (1) takes the same amount of time to compute for all values that $\mathbf{s}_{1:t-1}$ and $\mathbf{u}$ may take, and (ii) we have access to at least T processors with the same computational power. For simplicity, we only count the computational cost of evaluating the recurrence relation given in Eq. (1) and ignore other potential costs that depend more on specific hardware implementation, such as data movements and synchronization.

We now analyze the advantages of various methods when the recurrence relations have different structures under the above computation model, and when the computation model is relaxed.

4.1. When to Use the Jacobi Solver

The above computation model has already been used several times to argue that T parallel iterations of the Jacobi method costs the same wall-clock time as one sequential iteration of the GS method (*i.e.*, the standard feedforward computation). According to Proposition 1, the Jacobi algorithm converges within T parallel iterations. This implies that *running Algorithm 1 is always faster or equally fast than standard feedforward computation (or GS).*

Since Jacobi iterations use more processors for parallel execution, it is necessary to understand when the speedup of Jacobi methods is worthwhile. To get some intuition, we first consider some typical examples where Jacobi iterations may or may not lead to compelling speedups with respect to Gauss-Seidel.

Example 1: fully independent chains. The best case for Jacobi iteration is when for each $t = 1, \cdots, T$, $\mathbf{s}_t = h_t(\mathbf{u})$. For recurrent relations where different states are fully independent of each other, one parallel iteration of Jacobi suffices to yield the correct values for all states, whereas standard feedforward computation needs to compute each state sequentially. Parallelism in this case results in the maximum possible speedup factor of T.

Example 2: chains with long skip connections. Here is a slightly worse, but still advantageous case for Jacobi iterations: each state only depends on far earlier states in the sequence via long skip connections. One simple instance is when $\mathbf{s}_1 = h_1(\mathbf{u})$ and $\mathbf{s}_t = h_t(\mathbf{u}, \mathbf{s}_1)$ for $t > 1$. The Jacobi method needs only 2 parallel iterations to obtain the correct values of all intermediate states, which leads to a speedup factor of $T/2$. We note that skip connections are commonly used in machine learning models, for example in ResNets (He et al., 2016), DenseNets (Huang et al., 2017), and the computational graph of RNN backpropagation due to shared weights across time steps.

Example 3: Markov chains. The worst case for Jacobi iterations happens when the recurrence relation is strictly Markov, *i.e.*, $\mathbf{s}_1 = h_1(\mathbf{u})$ and $\mathbf{s}_t = h_t(\mathbf{s}_{t-1})$ for $t > 1$. The Markov property ensures that when $t > 1$, the only way for $\mathbf{s}_t$ to be influenced by the input $\mathbf{u}$ is through computing $\mathbf{s}_{t-1}$. Therefore, as long as $\mathbf{s}_T$ depends on $\mathbf{u}$ in a non-trivial way, it will take at least T parallel iterations for the Jacobi method to propagate information from $\mathbf{u}$ all the way to $\mathbf{s}_T$.

In this case the running time of Jacobi matches that of GS or feedforward computation under our computation model.

In general, a recurrence relation can be represented as a directed acyclic graph (DAG) with $T+1$ nodes $\{\mathbf{u}, \mathbf{s}_1, \mathbf{s}_2, \cdots, \mathbf{s}_T\}$ to indicate computational dependency between states. The number of parallel iterations needed for the Jacobi method to converge is upper bounded by the *critical path length* (Kelley Jr & Walker, 1959) (*i.e.*, the length of the longest path between all pairs of nodes), whereas the number of iterations required for standard feedforward computation is always T. Therefore, *Jacobi methods are better when the DAG has a smaller critical path length*.

In the strict sense, DAGs of many feedforward processes in machine learning may not have a small critical path length. For example, DenseNets have a critical path length of T since adjacent layers are connected, but empirically they enjoy substantial acceleration from Jacobi methods. This is because the influence of many connections is negligible (*e.g.*, weights are small) and the DAG without these weak connections can have a much smaller effective critical path length. This frequently happens because models are learned rather than manually specified, and small numerical errors do not affect results.

We stress that all examples considered above are overly simplified for illustrative purposes. Our experiments in Section 5 are on much more complicated tasks—neither RNN backpropagation, DenseNet evaluation, nor autoregressive sampling has a computational graph as simple as those examples. Empirically, we observe that Jacobi iterations have larger advantages when the computational graph of a machine learning task contains many long skip connections (*e.g.*, DenseNets), but fall short when the computational graph is closer to a Markov chain (*e.g.*, ResNets). Both are in agreement with the intuition given by our examples.

4.2. When to Use Hybrid Solvers

Our idealized computation model introduced at the beginning of this section assumes that we have T parallel processors, and updates in the recurrence relation at $t = 1, \cdots, T$ all have the same running time. When these assumptions do not hold, Jacobi-GS and GS-Jacobi are often more desirable than naïve Jacobi iterations.

First, when fewer than T parallel processors are available, we cannot directly apply the Jacobi method. In contrast, both Jacobi-GS and GS-Jacobi require a smaller number of parallel processors equal to the number of blocks and the block size respectively, and can thus be tuned at will.

Second, when the computation time is non-uniform across different t, each parallel iteration of the Jacobi method will be bottlenecked by the slowest update across all time steps. One can use Jacobi-GS and GS-Jacobi to reduce this bottleneck, since the former can group different time steps so that each block takes roughly the same time to update, balancing the work load across different parallel processors; the latter can reduce the number of steps computed in parallel, leading to a smaller bottleneck during each GS update.

Third, when serial computation has unique advantages, the Jacobi method may have degraded performance as it is purely parallel. Under certain cases, the computation for $h_t(\mathbf{u}, \mathbf{s}_{1:t-1})$ can be cached to save the time for computing $h_{t+1}(\mathbf{u}, \mathbf{s}_{1:t})$ (*cf.*, (Ramachandran et al., 2017) for autoregressive models). This makes sequential computations faster than independent executions in parallel, and therefore reduces the cost-effectiveness of Jacobi methods compared to feedforward computation. In contrast, both Jacobi-GS and GS-Jacobi are more advantageous because the sequential GS iterations within and between blocks can also benefit from the faster serial computation brought by caches.

Finally, Jacobi-GS often converges faster than Jacobi even without the above considerations. For example, the "block" Jacobi method in the context of solving linear triangular systems is equivalent to our Jacobi-GS when applied to linear recurrence relations, and is shown to enjoy faster convergence than naïve Jacobi iterations (Chow et al., 2018).

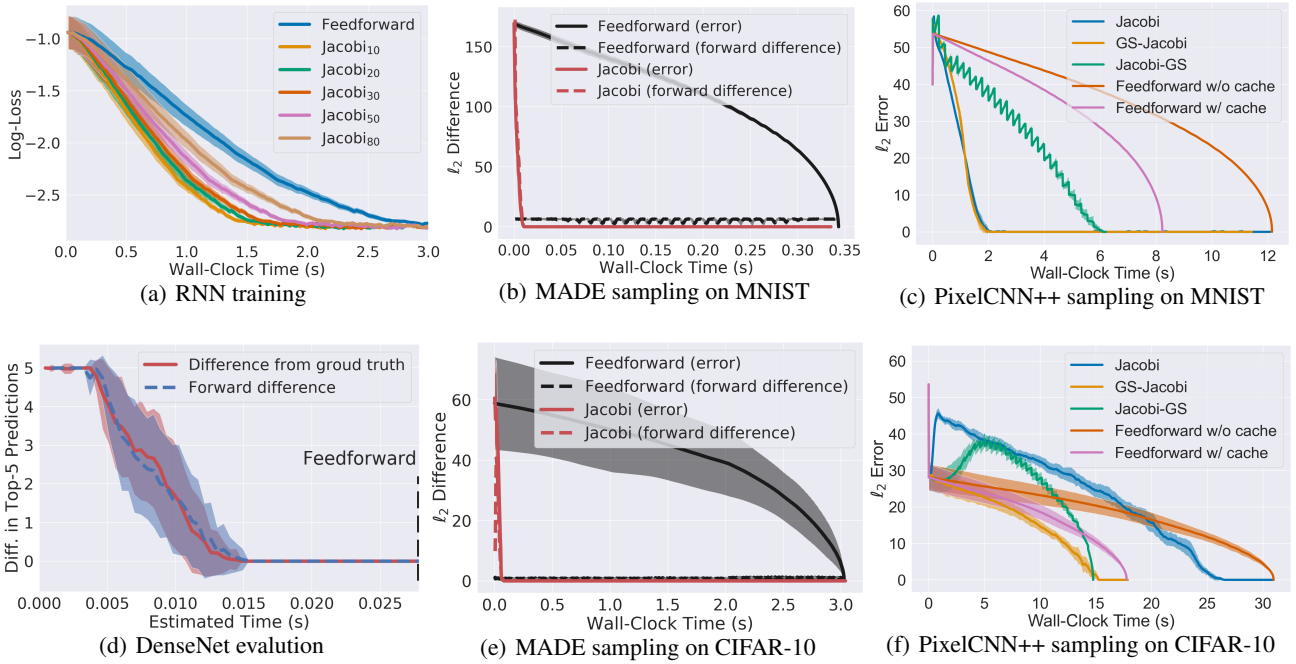
4.3. Practical Recommendations

Block size in hybrid solvers. When using hybrid methods, we should ensure that each block requires a comparable amount of computation. For Jacobi-GS, a larger block size requires fewer parallel computing units at the cost of slower running speed, while it is the opposite for GS-Jacobi. Users should balance this trade-off based on their goals and availability of computing units.

Number of iterations. Determining the number of total iterations to run in advance is hard. Instead, we recommend an adaptive approach, where users stop the iteration once the forward difference (defined in Section 2) is below a chosen tolerance value ϵ (see Algorithm 1, 2 and 3).

5. Experiments

Here we empirically verify the effectiveness of our proposed algorithms on (i) the backpropagation of RNNs, (ii) the evaluation of neural networks, and (iii) the ancestral sampling of deep autoregressive models. We report the speedups of our algorithms measured with wall-clock time on real hardware, except for the DenseNet experiment where we simulate the performance due to the difficulty of implementing our methods in current deep learning frameworks like PyTorch (Paszke et al., 2019) and TensorFlow (Abadi et al., 2015). We provide the main experimental results with key details in this section, and relegate other details/results to Appendix C/D.



(a) RNN training (b) MADE sampling on MNIST (c) PixelCNN++ sampling on MNIST

(d) DenseNet evalution (e) MADE sampling on CIFAR-10 (f) PixelCNN++ sampling on CIFAR-10

Figure 1. (a) The performance of Jacobi iterations on accelerating RNN training. Here we use "Jacobi$_n$" to denote the Jacobi method truncated at the n-th iteration, and "feedforward" for standard backpropagation. All values are averaged over 10 runs and shaded areas denote $1/10$ standard deviations. (d) The performance of Jacobi-GS on evaluating DenseNets. The y-axis represents the number of incorrect labels in top-5 predictions. The shaded areas represent standard deviations across 100 random input images. (b)(e) The performance of feedforward sampling vs. Jacobi iterations for MADE. The shaded areas represent standard deviations computed over 100 runs. (c)(f) Comparing different sampling algorithms for PixelCNN++. Results are averaged over 10 runs and shaded areas show standard deviations.

5.1. Backpropagation of RNNs

We consider accelerating the training procedure of a recurrent neural network (RNN) with Jacobi iterations. The backward pass of RNNs can benefit from Jacobi-type approaches, because the loss function is connected to all time steps in the computation graph, and therefore gradient information can quickly flow from the final loss value to all hidden states with one Jacobi update.

To demonstrate this, we train a simple RNN with one hidden layer to fit sequences. The dataset is synthesized by flattening resized MNIST digits (resolution 10×10). We report how the training loss decreases with respect to wall-clock time in Fig. 1(a). Since the length of input sequences is fixed to 100, there are a total of 100 steps in the backward pass. We use "Jacobi$_n$" to denote the Jacobi approach truncated at the n-th iteration ($n \leq 100$), and "feedforward" corresponds to the standard backpropagation algorithm. In Fig. 3 (see Appendix D), we show how Jacobi$_n$ converges to the true gradients with respect to n. We can trade-off between the accuracy and speed of gradient computation by tuning n. As demonstrated in Fig. 1(a), Jacobi methods can reduce the training time by around a factor of two with a proper n.

5.2. Evaluating DenseNets

DenseNets (Huang et al., 2017) are convolutional neural networks with a basic building block called the dense layer. Each dense layer contains two convolutions, and is connected to every other dense layer in a feedforward fashion. DenseNets are particularly suitable for Jacobi-type iterative approaches because information can quickly flow from input to output in one update via skip connections.

Setup. We use a DenseNet-201 model pre-trained on ImageNet (Russakovsky et al., 2015). *We define a state in the corresponding recurrence relation to be the feature maps of a convolutional layer*. We apply the Jacobi-GS method (Algorithm 2) to compute all states, where *each dense layer (consisting of two states) is grouped as one block*. We empirically verify that evaluating each dense layer separately takes comparable running time on GPUs. Therefore, by arranging these dense layers as blocks, Jacobi-GS can have roughly balanced workload for parallel execution.

Performance Metrics. For this task, full implementation of our algorithms will involve heterogeneous parallel execution of convolutional layers, which is not well supported by existing deep learning frameworks such as JAX (Brad-

bury et al., 2018), PyTorch (Paszke et al., 2019) or TensorFlow (Abadi et al., 2015). Therefore, we estimate the speedup for a real parallel implementation by simulating the performance of Jacobi-GS with a purely sequential implementation, assuming no overheads due to parallelism. Specifically, we run each dense layer 10 times on the GPU and take the average to measure its wall-clock time, which we denote as $t_1, t_2, \cdots, t_{98}$, since there are 98 blocks in total. We then estimate one parallel iteration of Jacobi-GS with $\max_{1\leq i\leq 98} t_i$, and the time for full feedforward computation with $\sum_{i=1}^{98} t_i$.

Results. We summarize the performance of Jacobi-GS in Fig. 1(d). We plot the curves of both error and forward difference (defined in Section 2.2.1), measured using the number of different labels in top-5 predictions. The results indicate that forward differences closely trace the ground-truth errors and therefore can be reliably used as a stopping criterion. As shown in Fig. 1(d), the estimated time for Jacobi-GS to converge is around **0.0131s**, which is **2.1** times faster than **0.0279s**, the estimated time needed for feedforward computation. Note that this is a theoretical speedup. The actual speedup might be smaller due to overheads of parallel execution.

5.3. Autoregressive Sampling

We consider two popular autoregressive models for image generation: MADE (Germain et al., 2015) and PixelCNN++ (Salimans et al., 2017). Both generate images pixel-by-pixel in raster scan order, and thus every pixel forms a state in the corresponding recurrence relation of feedforward computation.

5.3.1. MADE

For autoregressive sampling from MADE, each iteration of feedforward computation requires a forward propagation of the whole network, which equals the cost of one parallel Jacobi iteration. This means that sampling from MADE is a perfect use case for Jacobi iterations, where no extra parallelism is needed compared to naïve feedforward computation.

Setup. We compared Jacobi iteration against feedforward sampling for models trained on MNIST (LeCun & Cortes, 2010) and CIFAR-10 (Krizhevsky et al., 2009) respectively. The experiments were repeated 100 times and we report the means and standard deviations measured in *actual wall-clock time* on a single NVIDIA Titan Xp GPU, accounting for all the overheads.

Results. For Jacobi iterations, the feedforward difference can accurately trace errors between the current and final samples, which is thus a good metric for convergence and early stopping. In contrast, feedforward differences for the standard feedforward computation are not indicative of convergence. In terms of wall-clock time, Jacobi method only requires **0.013s** to converge on MNIST, while feedforward computation needs **0.343s**. This amounts to a speedup factor around **26**. For CIFAR-10, the time difference is **0.119s** vs. **3.026s**, which implies a speedup factor around **25**. The significant speedup achieved by Jacobi methods for MADE is highly practical. It not only accelerates image generation, but can also directly improve the speed for other models where MADE sampling is a sub-process, such as computing the likelihood of Inverse Autoregressive Flows (Kingma et al., 2016), and sampling from Masked Autoregressive Flows (Papamakarios et al., 2017).

5.3.2. PixelCNN++

PixelCNN++ is a more advanced autoregressive model that typically achieves higher likelihood on image modeling tasks compared to MADE. In addition to the vanilla Jacobi method, we test the proposed hybrid methods, Jacobi-GS and GS-Jacobi. Feedforward sampling from PixelCNN++ can be accelerated by caching (Ramachandran et al., 2017), where the computation performed for one state is memorized to accelerate the computation of later states. As discussed in Section 4, parallel Jacobi updates cannot leverage these caches for faster sampling, and therefore one parallel update can be slower than one sequential update of feedforward sampling. Jacobi-GS and GS-Jacobi, in contrast, can take advantage of the caching mechanism since they incorporate sequential updates.

Setup. We use PixelCNN++ models trained on MNIST and CIFAR-10 datasets. Each experiment is performed 10 times and we show both mean and standard deviation in Fig. 1(c) and 1(f). We consider feedforward sampling with and without caches. We implement Jacobi iterations in the same way as MADE, where no cache is used. We modify the caching mechanisms from (Ramachandran et al., 2017) so that they can be applied to Jacobi-GS and GS-Jacobi approaches. For GS-Jacobi, one block contains 15 rows of pixels on MNIST and 2 rows of pixels on CIFAR-10. For Jacobi-GS, one block has one row of pixels on both datasets. All results of wall-clock time are measured on a single NVIDIA Tesla V100 GPU with 32 GB memory. The batch sizes are 16 and 4 for MNIST and CIFAR-10 respectively.

Results. We report the performance of different samplers in Tab. 1, and include a visual comparison of Jacobi iteration vs. feedforward sampling (*i.e.*, the standard ancestral sampling) in Fig. 2. Compared to the standard feedforward computation (ancestral sampling) without caching, Jacobi, Jacobi-GS and GS-Jacobi all run significantly faster. Even

Table 1. Speedups for PixelCNN++ sampling on MNIST and CIFAR-10. Algorithms are stopped when the ℓ_∞ norm between the current sample and the ground-truth image is smaller than 0.01 (when the difference in samples is imperceptible to human eyes).

Method	MNIST		CIFAR-10	
	Time (s)	Speedup	Time (s)	Speedup
Feedforward w/o cache	12.15	1.00×	30.95	1.00×
Feedforward w/ cache	8.23	1.48×	17.76	1.74×
Jacobi	1.94	6.26×	26.16	1.18×
GS-Jacobi	**1.86**	**6.53**×	14.84	2.09×
Jacobi-GS	5.95	2.04×	**14.76**	**2.10**×

Figure 2. Feedforward (1st & 3rd rows) vs. Jacobi (2nd & 4th rows) sampling for PixelCNN++ on MNIST (**top 2 rows**) and CIFAR-10 (**bottom 2 rows**). Each column corresponds to the same number of updates. We show the first few intermediate samples on the left and the final image samples on the rightmost.

against feedforward sampling + caching, our GS-Jacobi and Jacobi-GS methods still perform uniformly better. Specifically, GS-Jacobi yields **6.53** and **2.09** times speedup (on MNIST and CIFAR-10) compared to the vanilla feedforward sampling without caching, and yields **4.42** and **1.20** times speedup against feedforward sampling + caching. Similarly, Jacobi-GS leads to speedup factors of **2.04** and **2.10** compared to the vanilla feedforward sampling, and still have speedup factors of **1.38** and **1.20** against feedforward sampling + caching. Compared to GS-Jacobi, Jacobi-GS may require fewer parallel processing units. For example, Jacobi-GS only requires 28 parallel computing units on MNIST, since there are 28 blocks and each block requires only one parallel device to run. In contrast, GS-Jacobi has a block size of 15×28 and requires the same number of parallel processing units. Our Jacobi method always outperforms the vanilla feedforward sampling without caching, with a speedup factor of **6.26** and **1.18** on MNIST and CIFAR-10 respectively. However, as demonstrated by our results on CIFAR-10 (see Tab. 1), Jacobi iterations may become slower than hybrid methods since the latter can exploit caching.

6. Related Work

Accelerating feedforward computation in the context of autoregressive sampling has been studied in the literature. In particular, van den Oord et al. (2018) propose probability density distillation to distill information from a slow autoregressive model to a faster sampler. However, it may provide samples from a different distribution compared to the original (slower) autoregressive model. MintNet (Song et al., 2019) proposes a fixed-point iteration method based on Newton-Raphson to speed up the inversion of an autoregressive procedure, but it is limited to a particular model. Similar ideas have also been proposed as a theoretical possibility in (Naumov, 2017) without experimental verifications.

Concurrently, Wiggers & Hoogeboom (2020) propose to accelerate autoregressive sampling with a fixed-point iteration method and demonstrate advantages over feedforward sampling (without caching) on PixelCNN++ models. Our Jacobi approach in Algorithm 1 is equivalent to theirs, but we additionally provide hybrid methods to improve the vanilla Jacobi approach, which are able to outperform feedforward sampling with caching. Our approaches are also more general, applicable to tasks beyond autoregressive sampling such as RNN training and DenseNet inference.

Common iterative solvers for linear equations include Jacobi, Gauss-Seidel, successive over-relaxation (SOR), and more general Krylov subspace methods. Forward/back substitution, as a process of solving lower/upper triangular linear systems, can also be viewed as instances of feedforward computation. Many approaches are proposed to accelerate and parallelize this procedure. Specifically, level scheduling (Saad, 2003) performs a topological sorting to find independent groups of variables that can be solved in parallel. Block-Jacobi iteration methods (Anzt et al., 2015; 2016; Chow et al., 2018), similar to the Jacobi-GS method in our paper, are proposed to maximize the parallel efficiency on GPUs.

Jacobi-type iterations are also used in message passing algorithms for probabilistic graphical models (Elidan et al., 2012; Niu et al., 2011) and graph neural networks (GNNs, Scarselli et al. (2008)). In particular, Gaussian belief propagation (GaBP) includes the Jacobi method as a special case (Bickson, 2008) when solving Gaussian Markov random fields. The core computation of GNNs is a parameterized message passing process where methods similar to block-Jacobi scheduling are popular (Liao et al., 2018).

7. Conclusion

By interpreting the feedforward computation as solving a triangular system of nonlinear equations, we show that numerical solvers can, in some cases, provide faster evaluation at the expense of additional parallel computing power.

In particular, we demonstrated that variants of Jacobi and Gauss-Seidel iterations are effective in accelerating the training of RNNs, the evaluation of DenseNets on ImageNet and the sampling from multiple deep autoregressive models, such as MADE and PixelCNN++, on several image datasets.

This observation opens up many new possible directions. We can build highly-optimized software packages to automatically parallelize some feedforward computation. More sophisticated numerical equation solving techniques, such as Krylov subspace methods and continuation methods, may provide greater acceleration than Jacobi or our hybrid methods. Our idea is particularly useful in time-critical applications, where trading parallel computing power for time is otherwise impossible.

Finally, we reiterate that our method is not beneficial for all feedforward computation. We require the process to tolerate numerical errors, have long skip connections, as well as have weak dependencies among various sequential stages that might be leveraged by numerical solvers (see the discussions in Section 4). Moreover, in some cases, it can be non-trivial for practical implementations to reap the benefits of acceleration that are possible in theory due to various overheads in software or hardware.

Acknowledgements

This research was supported by Intel Corporation, TRI, NSF (#1651565, #1522054, #1733686), ONR (N00014-19-1-2145), AFOSR (FA9550-19-1-0024). Yang Song was supported by the Apple PhD Fellowship in AI/ML.

References

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

Anzt, H., Chow, E., and Dongarra, J. Iterative sparse triangular solves for preconditioning. In *European Conference on Parallel Processing*, pp. 650–661. Springer, 2015.

Anzt, H., Chow, E., Szyld, D. B., and Dongarra, J. Domain overlap for iterative sparse triangular solves on gpus. In *Software for Exascale Computing-SPPEXA 2013-2015*, pp. 527–545. Springer, 2016.

Bickson, D. Gaussian belief propagation: Theory and aplication. *arXiv preprint arXiv:0811.2518*, 2008.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Chow, E., Anzt, H., Scott, J., and Dongarra, J. Using jacobi iterations and blocking for solving sparse triangular systems in incomplete factorization preconditioning. *Journal of Parallel and Distributed Computing*, 119:219–230, 2018.

Elidan, G., McGraw, I., and Koller, D. Residual belief propagation: Informed scheduling for asynchronous message passing. *arXiv preprint arXiv:1206.6837*, 2012.

Germain, M., Gregor, K., Murray, I., and Larochelle, H. Made: Masked autoencoder for distribution estimation. In *International Conference on Machine Learning*, pp. 881–889, 2015.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Heek, J., Levskaya, A., Oliver, A., Ritter, M., Rondepierre, B., Steiner, A., and van Zee, M. Flax: A neural network library and ecosystem for JAX, 2020. URL http://github.com/google/flax.

Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

Kelley Jr, J. E. and Walker, M. R. Critical-path planning and scheduling. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pp. 160–173, 1959.

Kingma, D. P., Salimans, T., Jozefowicz, R., Chen, X., Sutskever, I., and Welling, M. Improved variational inference with inverse autoregressive flow. In *Advances in neural information processing systems*, pp. 4743–4751, 2016.

Krizhevsky, A., Hinton, G., et al. Learning multiple layers of features from tiny images. 2009.

LeCun, Y. and Cortes, C. MNIST handwritten digit database. 2010. URL http://yann.lecun.com/exdb/mnist/.

Liao, R., Brockschmidt, M., Tarlow, D., Gaunt, A. L., Urtasun, R., and Zemel, R. Graph partition neural networks for semi-supervised classification. *arXiv preprint arXiv:1803.06272*, 2018.

Naumov, M. Parallel complexity of forward and backward propagation. *arXiv preprint arXiv:1712.06577*, 2017.

Niu, F., Ré, C., Doan, A., and Shavlik, J. Tuffy: Scaling up statistical inference in markov logic networks using an rdbms. *Proceedings of the VLDB Endowment*, 4(6), 2011.

Ortega, J. M. and Rheinboldt, W. C. *Iterative solution of nonlinear equations in several variables*, volume 30. Siam, 1970.

Papamakarios, G., Pavlakou, T., and Murray, I. Masked autoregressive flow for density estimation. In *Advances in Neural Information Processing Systems*, pp. 2338–2347, 2017.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pp. 8024–8035, 2019.

Ramachandran, P., Paine, T. L., Khorrami, P., Babaeizadeh, M., Chang, S., Zhang, Y., Hasegawa-Johnson, M. A., Campbell, R. H., and Huang, T. S. Fast generation for convolutional autoregressive models. *arXiv preprint arXiv:1704.06001*, 2017.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

Saad, Y. *Iterative methods for sparse linear systems*, volume 82. siam, 2003.

Salimans, T., Karpathy, A., Chen, X., and Kingma, D. P. Pixelcnn++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications. *arXiv preprint arXiv:1701.05517*, 2017.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

Song, Y., Meng, C., and Ermon, S. Mintnet: Building invertible neural networks with masked convolutions. In *Advances in Neural Information Processing Systems*, pp. 11002–11012, 2019.

van den Oord, A., Li, Y., Babuschkin, I., Simonyan, K., Vinyals, O., Kavukcuoglu, K., Driessche, G., Lockhart, E., Cobo, L., Stimberg, F., et al. Parallel wavenet: Fast high-fidelity speech synthesis. In *International Conference on Machine Learning*, pp. 3918–3926, 2018.

Wiggers, A. and Hoogeboom, E. Predictive sampling with forecasting autoregressive models. In *International Conference on Machine Learning*, pp. 10260–10269. PMLR, 2020.

Young, D. M. *Iterative solution of large linear systems*. Elsevier, 2014.

A. Examples of Feedforward Computation

Feedforward computation is ubiquitous in machine learning. Below we focus on three prominent examples that appear in our experiments.

A.1. Evaluating Neural Networks

Suppose we have an input $\mathbf{x}$ and a neural network of L layers defined by $f(\mathbf{x}) \triangleq a^L(\mathbf{b}^L + \mathbf{W}^L a^{L-1}(\cdots a^1(\mathbf{b}^1 + \mathbf{W}^1\mathbf{x})))$, where $a^\ell(\cdot)$, $\mathbf{b}^\ell$ and $\mathbf{W}^\ell$ denote the activation function, bias vector and weight matrix for the ℓ-th layer respectively. We typically evaluate $f(\mathbf{x})$ via feedforward computation, as can be seen by letting $T = L$, $\mathbf{u} = \mathbf{x}$, and defining $\mathbf{s}_t \triangleq a^t(\mathbf{b}^t + \mathbf{W}^t a^{t-1}(\mathbf{b}^{t-1} + \mathbf{W}^{t-1}a^{t-2}(\cdots)))$, $h_1(\mathbf{u}) \triangleq a^1(\mathbf{b}^1 + \mathbf{W}^1\mathbf{u})$ and $h_t(\mathbf{u}, \mathbf{s}_{1:t-1}) \triangleq a^t(\mathbf{b}^t + \mathbf{W}^t\mathbf{s}_{t-1})$ in Eq. (1). By changing $\mathbf{u}$, we can evaluate the neural network for different inputs.

A.2. Backpropagation

Consider the same neural network as discussed above. Let $\mathbf{r}_\ell \triangleq a^\ell(\mathbf{b}^\ell + \mathbf{W}^\ell a^{\ell-1}(\mathbf{b}^{\ell-1} + \mathbf{W}^{\ell-1}a^{\ell-2}(\cdots a^1(\mathbf{b}^1 + \mathbf{W}^1\mathbf{x}))))$, and $\mathbf{r}_0 \triangleq \mathbf{x}$. Suppose the loss function is $\mathcal{L}(\mathbf{r}_L)$. Through the chain rule, we can compute the gradient of $\mathcal{L}$ w.r.t. the ℓ-th layer by $\nabla_{\mathbf{r}_\ell}\mathcal{L} = \mathcal{L}'(\mathbf{r}_L)$ if $\ell = L$ and $\nabla_{\mathbf{r}_\ell}\mathcal{L} = (\mathbf{W}^\ell)^\mathsf{T}\nabla_{\mathbf{r}_{\ell+1}}\mathcal{L} \odot (a^{\ell+1})'(\mathbf{b}^{\ell+1} + \mathbf{W}^{\ell+1}\mathbf{r}_\ell)$ if $\ell < L$, where $\odot$ denotes the element-wise product. The backpropagation algorithm for computing $\nabla_{\mathbf{x}}\mathcal{L}$ can be viewed as feedforward computation, because we can define $\mathbf{u} = \{\mathbf{r}_0, \mathbf{r}_1, \cdots, \mathbf{r}_L\}$, and let $T = L + 1$, $\mathbf{s}_t \triangleq \nabla_{\mathbf{r}_{L-t+1}}\mathcal{L}$, $h_1(\mathbf{u}) \triangleq \mathcal{L}'(\mathbf{r}_L)$, $h_t(\mathbf{u}, \mathbf{s}_{1:t-1}) \triangleq (\mathbf{W}^{L-t+1})^\mathsf{T}\mathbf{s}_{t-1} \odot (a^{L-t+2})'(\mathbf{b}^{L-t+2} + \mathbf{W}^{L-t+2}\mathbf{r}_{L-t+1})$ in Eq. (1). Note that the gradients of $\mathcal{L}$ w.r.t. model parameters can be immediately computed after $\mathbf{s}_{1:T+1}$ has been obtained.

A.3. Sampling from Autoregressive Models

Autoregressive models define a high-dimensional probability distribution $p(\mathbf{x})$ via the chain rule $p(\mathbf{x}) = \prod_{i=1}^N p(x_i|x_{1:i-1})$. We can draw samples from this distribution using a sequential process called ancestral sampling. Concretely, we first draw $\tilde{x}_1 \sim p(x_1)$, and then $\tilde{x}_t \sim p(x_t|\tilde{x}_{1:t-1})$ for $t = 2, 3, \cdots, N$ successively. Let $\mathbf{u} = (u_1, u_2, \cdots, u_N)$ denote the states of the pseudo-random number generator that correspond to samples $\tilde{x}_1, \tilde{x}_2, \cdots, \tilde{x}_N$. For example, $u_1, u_2, \cdots, u_N$ may be uniform random noise used in inverse CDF sampling. The ancestral sampling process is an instance of feedforward computation, as in Eq. (1) we can set $T = N$, $\mathbf{s}_t = \tilde{x}_t$, and let $h_t(\mathbf{u}, \mathbf{s}_{1:t-1})$ be the pseudo-random number generator that produces $\tilde{x}_t$ from $p(x_t \mid \tilde{x}_{1:t-1})$ given $\mathbf{u}$. We can randomly sample the input $\mathbf{u}$ to generate different samples from the autoregressive model.

B. Proofs

Here we provide the convergence proofs for Jacobi, Jacobi-GS and GS-Jacobi algorithms.

Proposition 1. *Algorithm 1 converges and yields the same result as standard feedforward computation in at most T parallel iterations for any initialization of $\mathbf{s}^0_{1:T}$ if $\epsilon = 0$.*

Proof. We prove the conclusion by induction, and without loss of generality we assume the algorithm terminates at the T-th iteration. Suppose the true solutions for Eq. (5) are $\mathbf{s}^*_1, \mathbf{s}^*_2, \cdots, \mathbf{s}^*_T$. For the first parallel iteration, we have $\mathbf{s}^1_1 \leftarrow h_1(\mathbf{u}) = \mathbf{s}^*_1$. Now we hypothesize that for the k-th ($k \geq 1$) parallel iteration, $\forall j \leq k : \mathbf{s}^k_j = \mathbf{s}^*_j$. Suppose the hypothesis for k is true. Considering the $(k+1)$-th iteration, we have $\mathbf{s}^{k+1}_{k+1} \leftarrow h_{k+1}(\mathbf{u}, \mathbf{s}^k_{1:k}) = h_{k+1}(\mathbf{u}, \mathbf{s}^*_{1:k}) = \mathbf{s}^*_{k+1}$. In addition, for $i < k+1$, we have $\mathbf{s}^{k+1}_i \leftarrow h_i(\mathbf{u}, \mathbf{s}^k_{1:i-1}) = h_i(\mathbf{u}, \mathbf{s}^*_{1:i-1}) = \mathbf{s}^*_i$. Therefore, we have proved that the hypothesis holds true for $k+1$. Since we have shown that the hypothesis is true for $k = 1$, by induction it is true for all $k \geq T$, which implies $\mathbf{s}^T_{1:T} = \mathbf{s}^*_{1:T}$. In other words, the algorithm gives the true solutions to Eq. (5) in at most T parallel iterations. □

Proposition 2. *For any initialization, Jacobi-GS (Algorithm 2) and GS-Jacobi (Algorithm 3) converge in at most M block-wise iterations and yield the same results as obtained by standard feedforward computation if $\epsilon = 0$.*

Proof. We first prove the convergence of Jacobi-GS. Suppose the true solutions are $\mathbf{s}^*_1, \mathbf{s}^*_2, \cdots, \mathbf{s}^*_T$, and without loss of generality the algorithm terminates at $k = M$. For the first parallel iteration, we consider block $\mathcal{B}_1 = [\![a_1, b_1]\!]$. After completing all the GS steps for the first parallel iteration, it is easy to see that $\forall i \in [\![a_1, b_1]\!] : \mathbf{s}^1_i = \mathbf{s}^*_i$. Now we hypothesize that after the k-th ($k \geq 1$) parallel iteration, $\forall t \leq k, \forall i \in \mathcal{B}_t : \mathbf{s}^k_i = \mathbf{s}^*_i$. Consider the $(k+1)$-th iteration. Note that for all

$i \leq k+1$, we have $\forall j \in \mathcal{B}_i = [\![a_i, b_i]\!] : \mathbf{s}_j^{k+1} \leftarrow h_j(\mathbf{u}, \mathbf{s}_{1:a_i-1}^k, \mathbf{s}_{a_i:j-1}^{k+1}) = h_j(\mathbf{u}, \mathbf{s}_{1:a_i-1}^*, \mathbf{s}_{a_i:j-1}^{k+1})$, and GS iterations make sure that $\forall j \in \mathcal{B}_i : \mathbf{s}_j^{k+1} = \mathbf{s}_j^*$. This proves that the hypothesis is true for $k+1$. Since we have shown the correctness of the hypothesis for $k=1$, by induction we know the hypothesis holds true for all $1 \leq k \leq M$. This implies that $\mathbf{s}_{1:T}^M = \mathbf{s}_{1:T}^*$.

Next, we prove the convergence of GS-Jacobi. For the first GS iteration, we know $\forall j \in \mathcal{B}_1 : \mathbf{s}_j^{|\mathcal{B}_1|} = \mathbf{s}_j^*$ from Proposition 1, and therefore $\mathbf{s}_{\mathcal{B}_1} = \mathbf{s}_{\mathcal{B}_1}^{|\mathcal{B}_1|} = \mathbf{s}_{\mathcal{B}_1}^*$. We can simply continue this reasoning to conclude that $\forall 1 \leq i \leq M : \mathbf{s}_{\mathcal{B}_i} = \mathbf{s}_{\mathcal{B}_i}^{|\mathcal{B}_i|} = \mathbf{s}_{\mathcal{B}_i}^*$.

□

C. Extra Experimental Details

C.1. RNN

We train a standard one-layer RNN on resized MNIST images. MNIST is dataset of hand-written digits with 50000 training data and 10000 test data. The original resolution of these images is 28×28, and we resize it to 10×10. The RNN has 128 hidden units, and uses the `SoftPlus` activation function. All weights are initialized with Gaussian noise of mean zero and standard deviation 0.1. The bias parameters are initialized to zero. We train the RNN with stochastic gradient descent, where the learning rate is 0.0001, batch size is 1, and momentum is 0. All experiments are implemented with PyTorch and run on an Nvidia Titan Xp GPU. All GPU timing is done after properly calling `torch.cuda.synchronize()`.

C.2. DenseNets

We use the DenseNet-201 model provided by the PyTorch (Paszke et al., 2019) model zoo, which has been pre-trained on the ImageNet (Russakovsky et al., 2015) dataset with a top-5 error of 6.43%. We use an Nvidia Titan Xp GPU in our experiments. All GPU timing is done after properly calling `torch.cuda.synchronize()`.

C.3. MADE

Our MADE network has two layers, each with 512 neurons. For training MADE on both MNIST and CIFAR-10, we use a batch size of 128, a learning rate of 0.001 for the Adam optimizer, and a step-wise learning rate decay of 0.999995. The models were trained for 1000 epochs. During sampling, we produce 100 images in parallel from our MADE model. We use a logistic distribution to model each conditional probability. For MNIST images, the resolution is $28 \times 28 \times 1$ and thus $T = 784$. For CIFAR-10 images, the resolution is $32 \times 32 \times 3$ and therefore $T = 3072$. We use a single Nvidia Titan Xp GPU for our experiments and measure wall-clock time after properly calling `torch.cuda.synchronize()`.

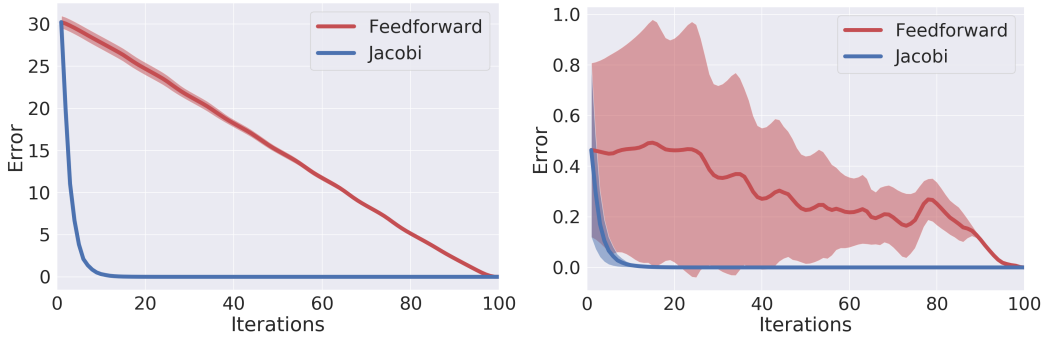
C.4. PixelCNN++

For CIFAR-10, we use the same architecture and checkpoint provided by the original PixelCNN++ paper (Salimans et al., 2017). For MNIST, the architecture is the same as that for CIFAR-10, except that we shrink the number of filters to $1/4$. We train the models on MNIST using a batch size of 32, a learning rate of 0.0002 for the Adam optimizer, and a step-wise learning rate decay of 0.999995. The model was trained for 590 epochs. All models are implemented in JAX (Bradbury et al., 2018) and FLAX (Heek et al., 2020).

For MNIST, the image resolution is $28 \times 28 \times 1$ and therefore $T = 784$. For CIFAR-10 images, the resolution is $32 \times 32 \times 3$, but different from MADE, PixelCNN++ views all channels at one location as one state, which means $T = 32 \times 32 = 1024$. In our experiments, we run everything on an Nvidia Tesla V100 GPU (32 GB). All GPU timing is done by calling `.block_until_ready()` properly.

D. Extra Experimental Results

We provide additional results on how fast the Jacobi algorithm converges for RNN backpropagation. Note that the performance of Jacobi methods will change gradually as the RNN model parameters evolve during training. We report the convergence results before training starts in Fig. 3(a), and after training finishes in Fig. 3(b), where Jacobi methods have a clear advantage in both cases. We also provide a demonstration on the standard feedforward sampling procedure vs. our Jacobi sampling method for MADE in Fig. 4. In Fig. 5, we show how various methods convergence with respect to the number of (parallel) iterations in lieu of the wall-clock time (*cf*., Fig. 1).



(a) Before training (b) After training

Figure 3. Convergence of gradient errors when using Jacobi iterations to accelerate the backpropagation of RNNs. Gradient errors are measured with ℓ_2 norm, and averaged over 10 runs. The shaded area denotes $1/10$ of standard deviations.

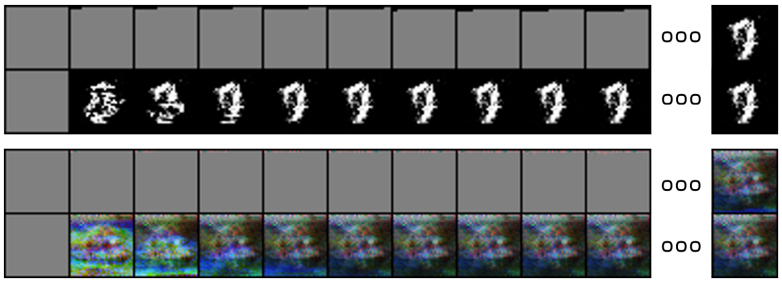
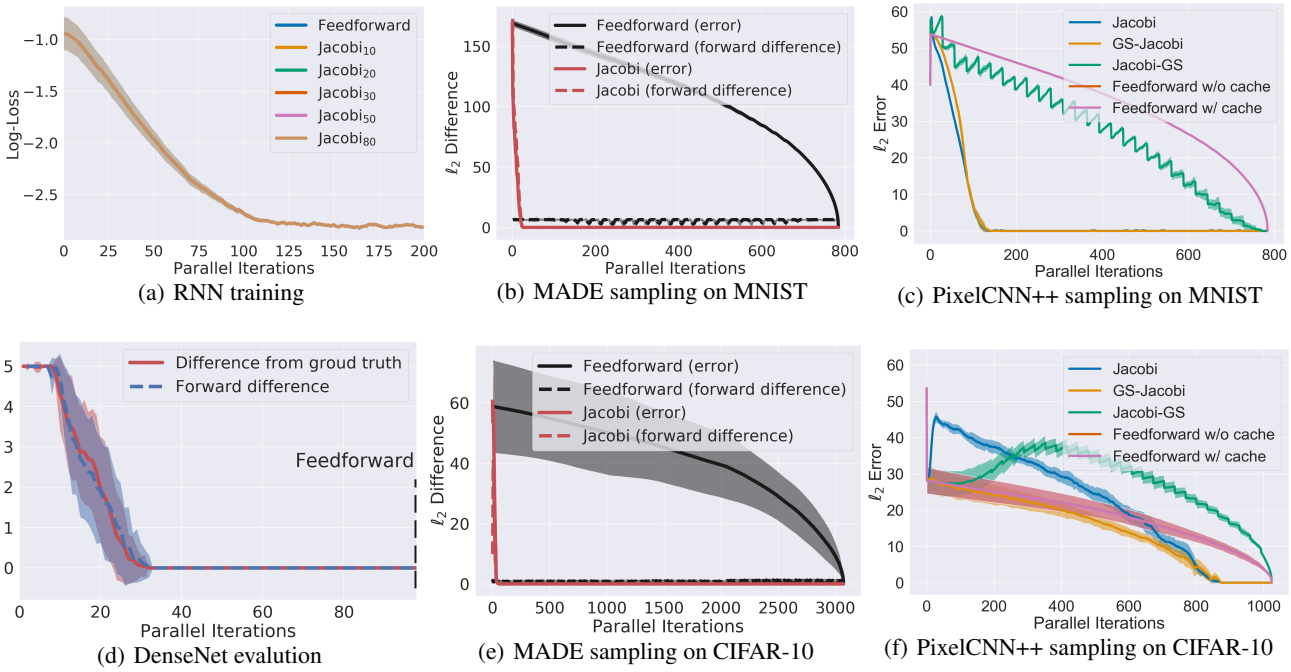


Figure 4. Demonstration of the Jacobi sampling process for MADE on MNIST (**top two rows**) and CIFAR-10 (**bottom two rows**). The odd rows correspond to standard feedforward sampling, and the even rows are from the Jacobi sampling process. We show the intermediate samples every five (parallel) iterations on the left side of the ellipses, and the final image samples on the right.



(a) RNN training (b) MADE sampling on MNIST (c) PixelCNN++ sampling on MNIST

(d) DenseNet evalution (e) MADE sampling on CIFAR-10 (f) PixelCNN++ sampling on CIFAR-10

Figure 5. (a) The performance of Jacobi iterations on accelerating RNN training. Here we use "Jacobi$_n$" to denote the Jacobi method truncated at the n-th iteration, and "feedforward" for standard backpropagation. All values are averaged over 10 runs and shaded areas represent $1/10$ of standard deviations. All curves coincide with each other. (d) The performance of Jacobi-GS on evaluating DenseNets. The y-axis represents the number of incorrect labels in top-5 predictions. The shaded areas represent standard deviations across 100 random input images. (b)(e) The performance of feedforward sampling vs. Jacobi iterations for MADE. The shaded areas represent standard deviations computed over 100 runs. (c)(f) Comparing different sampling algorithms for PixelCNN++. Results are averaged over 10 runs and shaded areas show standard deviations.