Optimizing Recurrent Layers

User's Guide | NVIDIA Docs

DU-09796-001_v001 | February 2023

Table of Contents

List of Figures

List of Tables

Chapter 1. Quick Start Checklist

The following quick start checklist provides specific tips for recurrent layers.

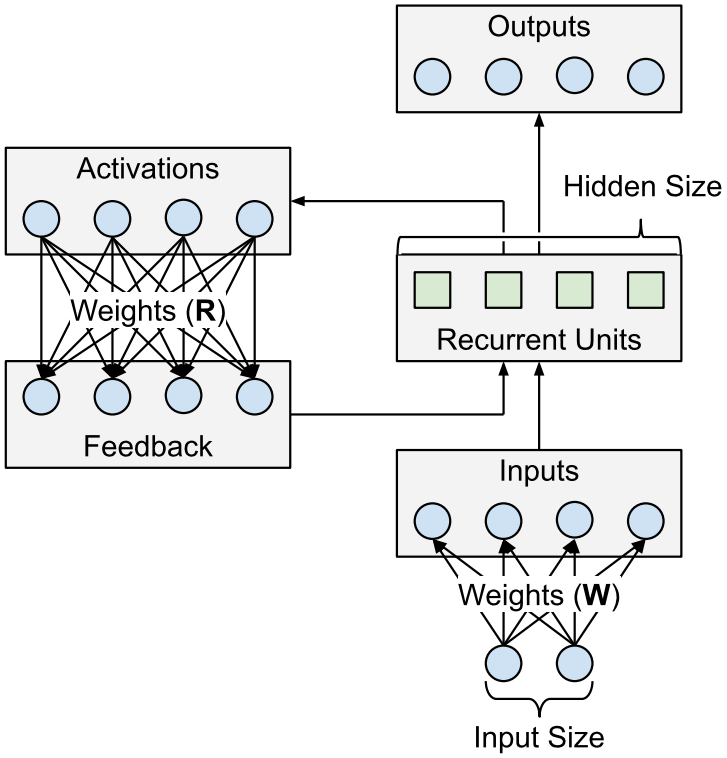
- Recurrent operations can be parallelized as described in the Recurrent Layer. We recommend using NVIDIA® cuDNN implementations, which do this automatically.
- When using the standard implementation, size-related parameters (minibatch size and hidden sizes) should be:
  - Divisible by 8 (for FP16) or 4 (for TF32) to run efficiently on Tensor Cores; refer to Tensor Core Requirements.
  - Divisible by at least 64 and ideally 256 to improve tiling efficiency; refer to Dimension Quantization Effects.
  - Greater than 128 (minibatch size) or 256 (hidden sizes) to be limited by computation rate rather than memory bandwidth (NVIDIA V100; thresholds vary by GPU); refer to Math And Memory Bounds.
- When using the persistent implementation (available for FP16 data only):
  - Hidden sizes should be divisible by 32 to run efficiently on Tensor Cores. Better tiling efficiency may be achieved by choosing hidden sizes divisible by larger multiples of 2, up to 256.
  - Minibatch size should be divisible by 8 to run efficiently on Tensor Cores. If minibatch size exceeds 96 during the forward pass or 32 during the backward pass, divisibility by a larger power of two may be required.
  - Refer to Case Study: Persistence With GNMT for an example.
- Try increasing parameters for better efficiency. Doubling minibatch size, for example, often results in each minibatch doing twice the calculation in less than twice the time. Hidden size and minibatch size are most impactful. Increasing sequence length may also improve efficiency if sequence length is very small or minibatch size does not follow the guidelines. Refer to the Input And Hidden Size and Minibatch Size And Sequence Length sections, respectively.

Chapter 2. Recurrent Layer

Recurrent neural networks (RNNs) are a type of deep neural network where both input data and prior hidden states are fed into the network's layers, giving the network a state and hence memory. RNNs are commonly used for sequence-based or time-based data.

During training, input data is fed to the network with some minibatch size (the number of data sequences in the minibatch) and sequence length (the number of time steps in each sequence). As shown in Figure 1, each layer of an RNN is composed of recurrent units, the number of which is the hidden size of the layer.

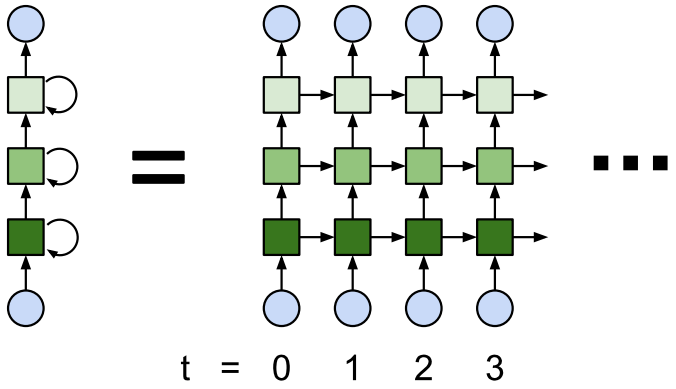
Figure 1. A single recurrent layer with two inputs and four hidden units. RNNs are usually composed of multiple such layers; this layer would receive its inputs from a layer with a hidden size of two.



Recurrent units come in a virtually unlimited number of types; many frameworks even allow you to define custom recurrent units. However, most networks use a few common types of units, including long short-term memory (LSTM) units or gated recurrent units (GRUs). These units are supported by the cuDNN API, along with the older and simpler ReLU and tanh activation units. In this guide, we focus on LSTM units, but networks using other unit types behave similarly.

Recurrent operations are not as straightforward to represent as GEMMs as some of the others we've discussed in this guide. Unlike feed-forward networks, where each input tensor is processed independently, recurrent networks have an inherent dependency of later time-steps on earlier ones.

Figure 2. A 3-layer RNN across 4 timesteps. Vertical arrows represent computations in the layer-to-layer path, which depend on activations from previous layers. Horizontal arrows represent computations in the recurrent path, which depend on the results of previous timesteps.



Chapter 3. GEMM Dimensions And Parallelization

Without considering parallelization, training for a recurrent layer may be represented as a set of individual matrix multiplies. Each phase of training (forward, activation gradient, and weight gradient) for each gate (four for LSTM units, three for GRUs, and one for ReLU and Tanh units) is equivalent to a GEMM with one dimension of one and the other two equal to hidden sizes.

We would like to parallelize these computations to maximize the work available to the GPU. When using LSTM units or GRUs, the GEMMs for different gates can be concatenated. GEMMs can also be combined within a mini-batch and, if time dependency allows, across sequence steps. Examples of the resulting dimensions are shown in the following table.

With cuDNN, including functions like cudnnRNNForwardTraining(), this is done automatically; these implementations can also be used through frameworks including PyTorch, TensorFlow, and MXNet for some types of recurrent layers. Relevant tips can be found in Parameters And Performance. If cuDNN won't work with your application, on the other hand, the types of parallelism described in this section could be implemented manually to improve performance.

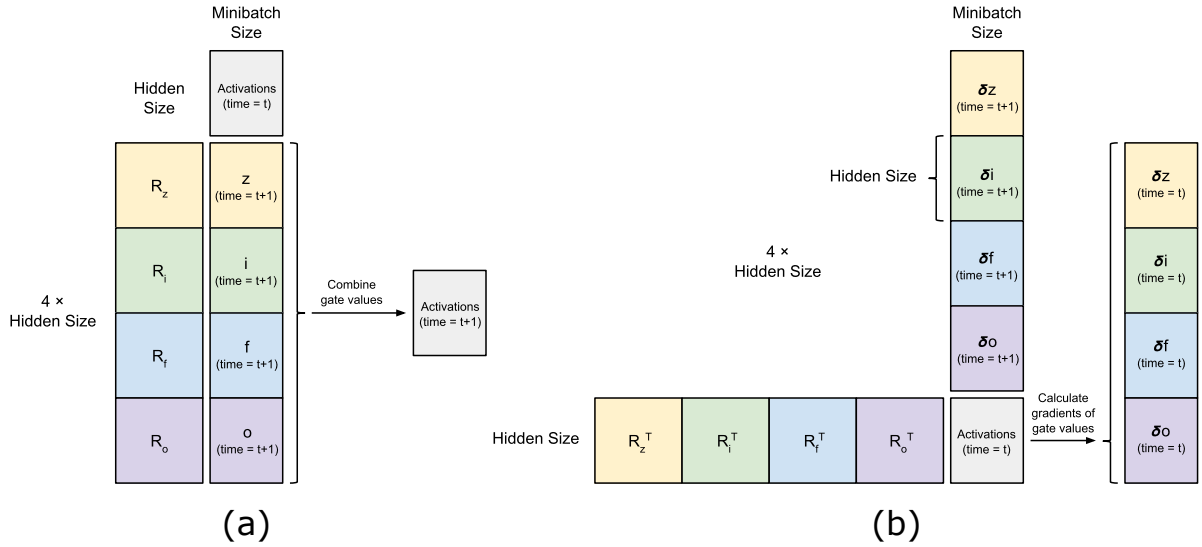
Table 1. Mapping of recurrent layer parameters to equivalent GEMM dimensions, assuming LSTM units. For GRUs, substitute a factor of 3 for the factor of 4 shown here; for ReLU or Tanh units, substitute a factor of 1. H represents the hidden size, B the minibatch size, S the sequence length, and F the unrolling factor, which is between 1 and S.

Path	Computation Phase	M	N	K	Number Of GEMMs
Recurrent	Forward Propagation	4 x H	B	H	S
	Activation Gradient	H	B	4 x H	S
	Weight Gradient	H	4 x H	B x S	1
Layer-to-Layer	Forward Propagation	4 x H (this layer)	B x F	H (the previous layer)	1

Path	Computation Phase	M	N	K	Number Of GEMMs
	Activation Gradient	H (this layer)	B x F	4 x H (next layer)	1
	Weight Gradient	H (the previous layer)	4 x H (this layer)	B x S	1

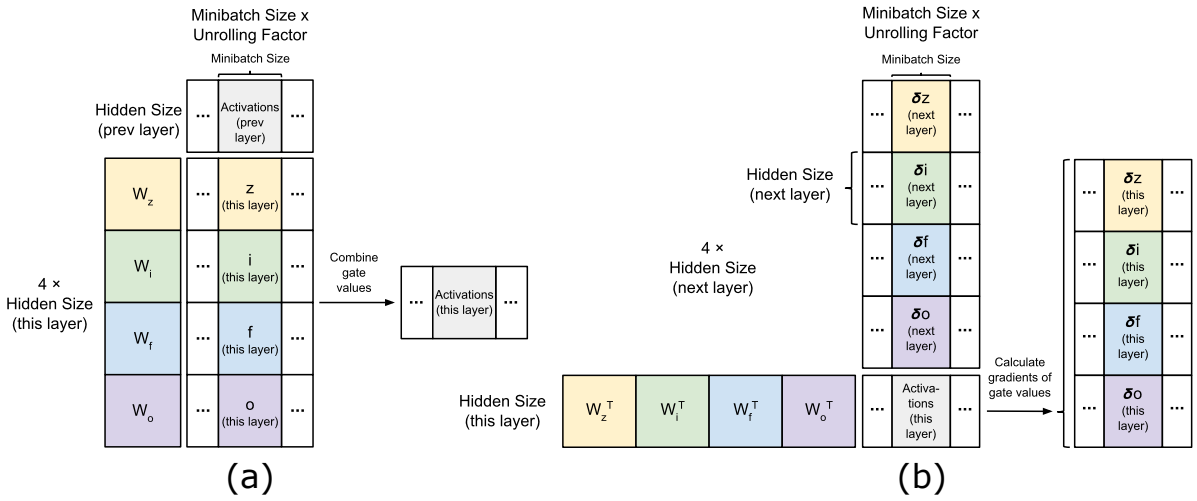
Here we show dimensions for recurrent layers composed of LSTM units (Figure 3 and Figure 4), where four GEMMs are concatenated; with ReLU/Tanh units and GRUs, there are instead one and three GEMMs, respectively. For forward and activation gradient passes in the recurrent path, each sequence step depends on the previous one. We can combine these GEMMs over the minibatch size, but not over different sequence steps.

Figure 3. Dimensions of equivalent GEMMs for recurrent (a) forward and (b) activation gradient passes on one layer of LSTM units.



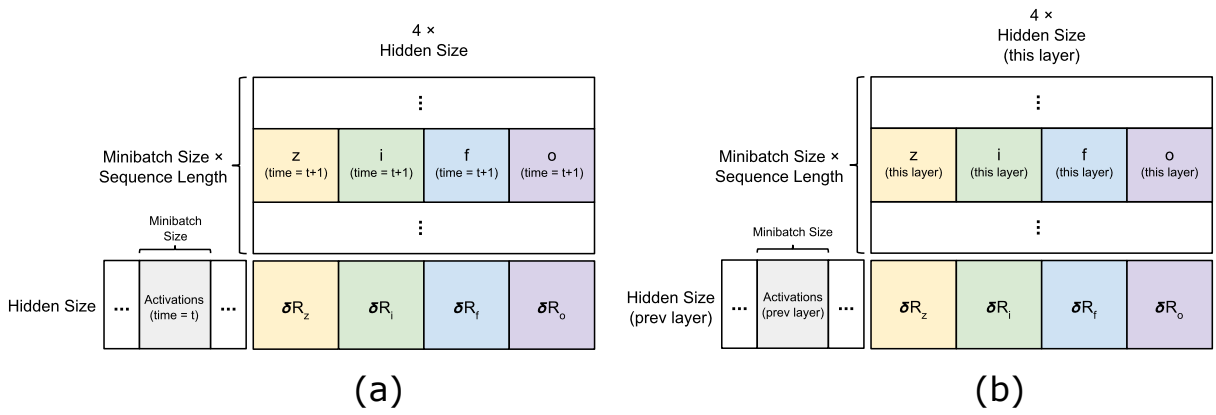
By comparison, in the layer-to-layer path, each sequence step only depends on outputs from the previous layer. Therefore, we can combine GEMMs over both the mini-batch size and multiple sequence steps. Often it isn't efficient to combine over the entire sequence length. The number of sequence steps combined for each GEMM is referred to as the unrolling factor and is chosen automatically by a heuristic.

Figure 4. Dimensions of equivalent GEMMs for layer-to-layer (a) forward and (b) activation gradient passes between a single pair of layers of LSTM units.



Likewise, for the weight gradient pass in both paths, no dependency between sequence steps exists; these GEMMs are also safe to combine over both the minibatch size and sequence steps. During this pass, GEMMs are combined over the entire sequence length.

Figure 5. Dimensions of equivalent GEMMs for (a) recurrent and (b) layer-to-layer weight gradient passes, again with LSTM units.



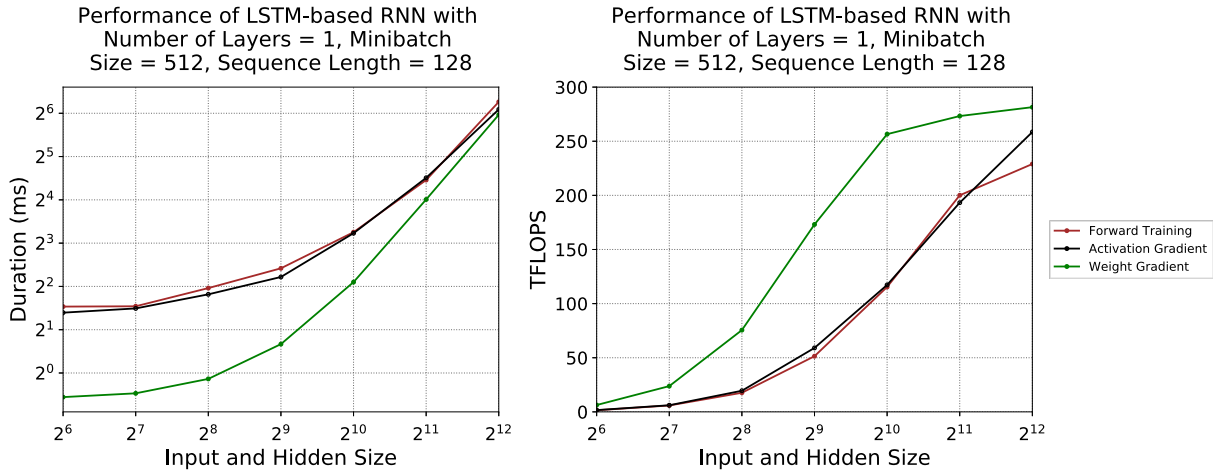
In both recurrent and layer-to-layer calculations, the same weights are used to compute activations across batches and sequences. An implementation with persistent weights is available when the hidden size is small enough that weight matrices can be cached locally instead of being loaded repeatedly. This persistent implementation tends to outperform the default when minibatch size is small; depending on your framework, it can be selected automatically.

Chapter 4. Parameters And Performance

4.1. Input And Hidden Size

The hidden size of a layer, and for layer-to-layer operations, the hidden sizes of the neighboring layers as well, contribute(s) to two of the three equivalent GEMM dimensions for each RNN operation. For forward and activation gradient passes, these dimensions are M and K; for the weight gradient pass, M and N. The remaining dimension is controlled by the minibatch size and/or sequence length. Hidden size is thus one of the most important parameters governing performance.

Figure 6. Calculations are more efficient for larger numbers of inputs and hidden nodes. The weight gradient pass performs better than forward or activation gradient passes, as the corresponding GEMM tends to have M and N that are more similar and a sufficiently large K for both recurrent and layer-to-layer paths. NVIDIA A100-SXM4-80GB, CUDA 11.2, cuDNN 8.1.

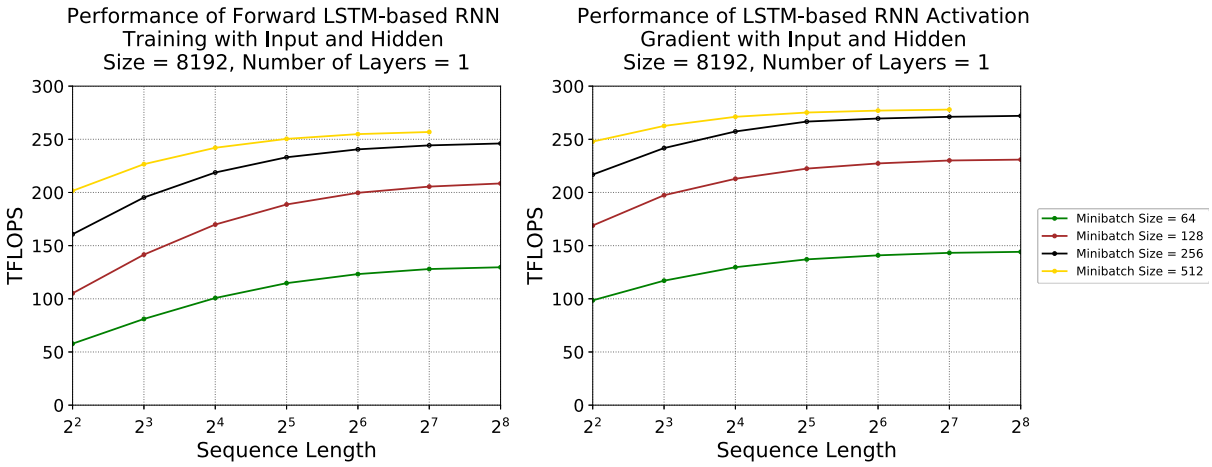


GEMMs tend to not perform as well when one or more dimensions are small. With hidden sizes up to roughly 256 units, GEMMs are likely to be memory-bound. Tiling and GPU occupancy issues can also occur. At these sizes, hidden size can often be increased with little impact on training duration. If a small hidden size is non-negotiable, it may be worthwhile to implement parallelism between multiple layers of the network. This can be done in the cuDNN API by defining a stack of layers with cudnnSetRNNDescriptor(); layers that cannot efficiently utilize the GPU on their own are automatically computed in parallel.

4.2. Minibatch Size And Sequence Length

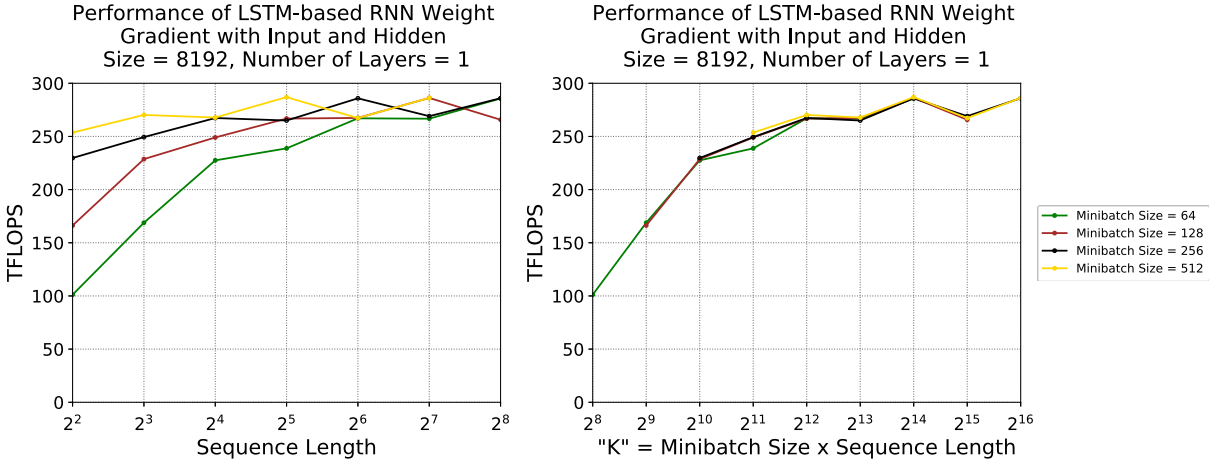
Mini-batch size always contributes to at least one GEMM dimension, so larger mini-batches tend to be more efficient. During the forward and activation gradient passes, the "N" dimension is equal to the mini-batch size (for the recurrent path) or the product of mini-batch size and sequence length (for the layer-to-layer path). This leads to some interesting effects: mini-batch size tends to impact performance more strongly than sequence length. A small mini-batch size can result in poor performance even for a large sequence length.

Figure 7. For forward and activation gradient passes, the "N" dimension depends upon minibatch and, in the layer-to-layer calculations, sequence length. NVIDIA A100-SXM4-80GB, CUDA 11.2, cuDNN 8.1.



For the weight gradient pass, the "K" dimension is equal to the product of mini-batch size and sequence length for both recurrent and layer-to-layer paths. Performance is thus similar for equal values of this dimension regardless of the individual values of mini-batch size and sequence length.

Figure 8. During the weight gradient pass, the “K” dimension is the same for both paths. NVIDIA A100-SXM4-80GB, CUDA 11.2, cuDNN 8.1.



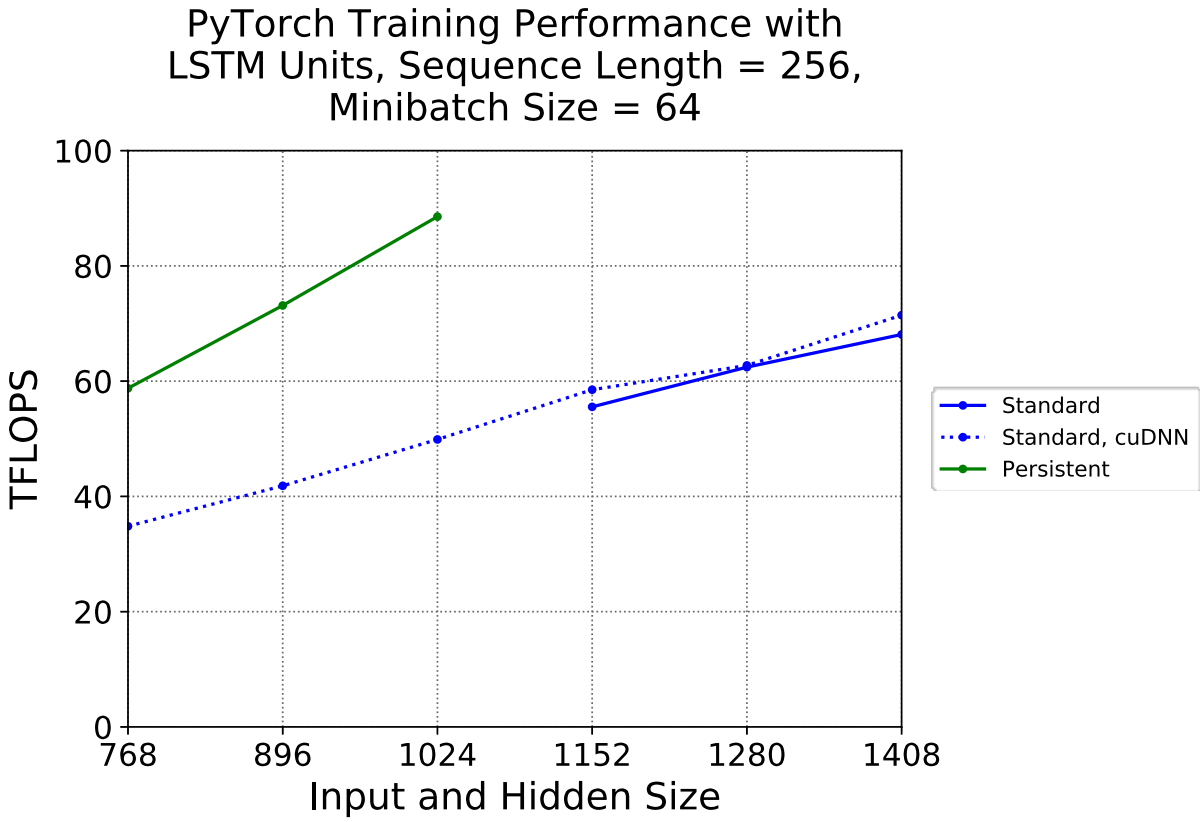
Chapter 5. Case Study: Persistence With GNMT

The Google Neural Machine Translation system (GNMT) and many other language processing networks make use of stacks of recurrent layers. GNMT's architecture is composed of encoder, decoder, and attention modules, with both the encoder and decoder containing stacked LSTM layers.

Earlier in this section, we mentioned that when minibatch size is small, the calculation speed of recurrent layers like these can be limited by data movement with the standard implementation. One alternative is a persistent implementation, in which weight matrices are cached locally to avoid repeated loading from memory. In cuDNN, this implementation can be selected with cudnnRNNAlgo_t. Some frameworks have also made it available, including PyTorch, which we use for this example. The persistent implementation cannot be manually selected with PyTorch; instead, it is automatically used when it is expected to improve performance.

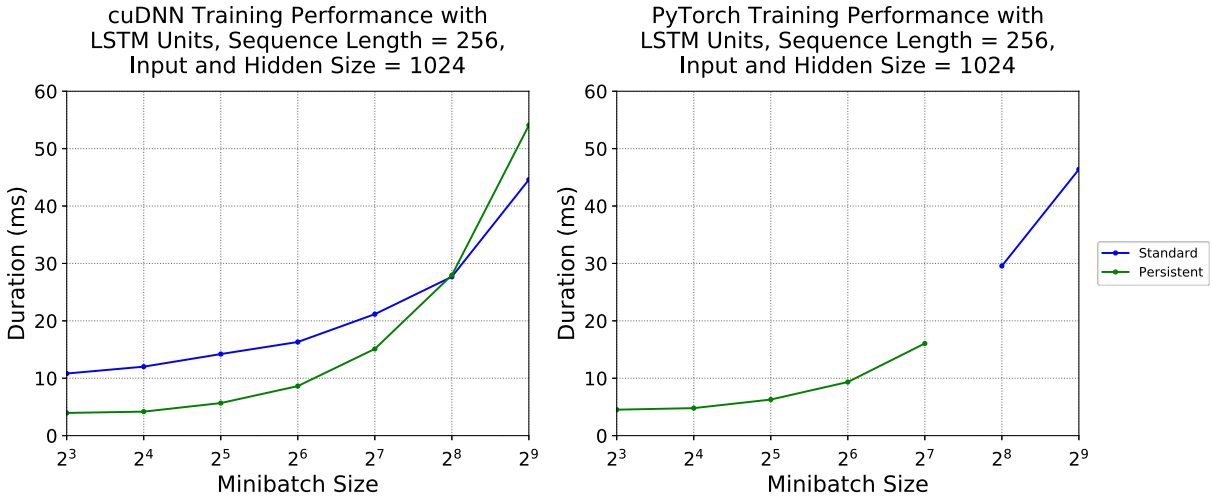
There are three major restrictions to keep in mind with persistence. First, the persistent implementation is only available when using FP16. Second, the weight matrices must be small enough to be cached locally across iterations. This means that hidden size must be below a threshold (with the specific value depending on the unit type). When training GNMT with a sequence length of 256 and a minibatch size of 64, persistence can be used with LSTM layers that have hidden sizes up to 1024 units. Ensuring that a layer is small enough to use persistence can significantly improve performance (Figure 9).

Figure 9. The persistent implementation may be selected for hidden sizes 1024 and lower with this type of recurrent layer. Layers with too many units do not benefit from persistence. cuDNN execution times with standard mode always used are shown for comparison. NVIDIA A100-SXM4-80GB, CUDA 11.2, cuDNN 8.1, PyTorch 1.8.



Third, the persistent implementation most often outperforms the standard one when minibatch size is small. Figure 10a shows the training performance of LSTM layers with 1024 units using a sequence length of 256 in both standard and persistent mode across mini-batch sizes; for minibatches with fewer than 256 elements, the persistent calculation is significantly faster. In this range, we recommend choosing hidden size to enable persistent mode (and all parameters to enable Tensor Cores, as described in the Quick Start Checklist) whenever possible. For larger mini-batches, where the standard implementation performs similarly to or better than the persistent one, this is unnecessary. PyTorch switches between implementations for best performance as mini-batch increases (Figure 10b).

Figure 10. Training performance with (a) cuDNN, for both standard and persistent implementations, and (b) PyTorch, switching automatically between standard and persistent as minibatch size changes. The persistent implementation is particularly efficient with small minibatches. For larger mini-batches, the standard implementation is faster and selected by PyTorch instead. NVIDIA A100-SXM4-80GB, CUDA 11.2, cuDNN 8.1, PyTorch 1.8.



Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

Google

Android, Android TV, Google Play and the Google Play logo are trademarks of Google, Inc.

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
www.nvidia.com

Trademarks

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
www.nvidia.com