NeMo: a toolkit for building AI applications using Neural Modules

Oleksii Kuchaiev, Jason Li, Huyen Nguyen, Oleksii Hrinchuk, Ryan Leary, Boris Ginsburg, Samuel Kriman, Stanislav Beliaev, Vitaly Lavrukhin, Jack Cook, Patrice Castonguay, Mariya Popova, Jocelyn Huang, Jonathan M. Cohen

NVIDIA
Santa Clara, CA
{okuchaiev,jasoli,chipn,ohrinchuk,rleary,bginsburg,skriman,stanislavv,
vlavrukhin,jocook,pcastonguay,mpopova,jocelynh,jocohen}@nvidia.com

Abstract

NeMo (**Ne**ural **Mo**dules) is a Python framework-agnostic toolkit for creating AI applications through re-usability, abstraction, and composition. NeMo is built around neural modules, conceptual blocks of neural networks that take *typed* inputs and produce *typed* outputs. Such modules typically represent data layers, encoders, decoders, language models, loss functions, or methods of combining activations. NeMo makes it easy to combine and re-use these building blocks while providing a level of *semantic correctness* checking via its neural type system. The toolkit comes with extendable collections of pre-built modules for automatic speech recognition and natural language processing. Furthermore, NeMo provides built-in support for distributed training and mixed precision on latest NVIDIA GPUs. NeMo is open-source.[1]

1 Introduction

Deep Learning (DL) has made huge progress from academia to industry in the last decade. However, the process for developing, debugging, and deploying DL software is significantly more cumbersome than other complex software systems. The primary abstraction in all DL frameworks is a multidimensional tensor, typically without any dimensional semantics, e.g. whether the first dimension represents the batch size or something else. The lack of semantics and a type system complicates models' re-use and makes it difficult to build DL systems [6]. It can be challenging to reuse components of a complex DL model across different use cases or developers. The typical approach for reusing and sharing components is based on open-source pre-trained models. Combining and chaining these models together usually requires making changes to the code, which, in turn, requires debugging. This is especially tricky when the models come from different developers or use cases.

Another complicating factor is that model configurations are usually defined via a Python script instead of a data model. This leads to a blurring of lines between what would otherwise be separate concerns – computational performance, architecture definition, training procedure, visualization and analysis – all mixed together into the same Python script that is difficult to disentangle, debug, or reuse in other contexts.

All of these challenges – separation of concerns, system decomposition with well-defined verifiable interfaces, and code re-usability – are already well-explored in the world of software engineering. Many of the modern techniques any software developer takes for granted were originally invented in order to address precisely these issues.

[1] Available at: https://github.com/NVIDIA/NeMo

Preprint. Under review.

We seek to translate common software engineering practices developed to address those issues into the context of developing AI-based applications. Specifically, we focus on the problems of:

- **decomposition** of a complex system into functional blocks with well-defined interfaces;
- **static type checking** to ensure API compliance and to catch type-mismatch bugs;
- **separation of concerns** between model architecture, training procedure, DL framework, optimization algorithm;
- **high performance training** by supporting modern efficient hardware features; and
- **reusable pre-built components** that can be easily combined in novel ways.

NeMo consists of: (1) **NeMo Core**: fundamental building blocks for all neural models and type system and (2) **NeMo collections**: pre-built neural modules for particular domains such as automatic speech recognition (ASR), and natural language processing (NLP).

2 Related work

In recent years, there have been a number of high-level toolkits aimed to help users to achieve certain goals easier then by purely using DL frameworks such as TensorFlow [8] and PyTorch [17].

These toolkits could be loosely classified into two main groups: (1) higher-level neural network APIs such as Keras[11], Sonnet [18], PyTorch Ignite [3], PyTorch Lightning [4] and (2) configuration-driven tookits such as Tensor2Tensor [20], Ludwig [5], OpenSeq2Seq [13], FairSeq [16], OpenNMT [12], Seq2Seq [9] and many others.

Conceptually, NeMo Core is closer to the first group. It allows users to express models with arbitrary sets of components and hides away details of training and evaluation loops while still retaining a lot of flexibility. NeMo collections, on the other hand, are closer to the second group. They contain common modules that can be re-used in various scenarios. For example, it is straightforward in NeMo to define templates for fixed patterns, such as an encoder-decoder network.

NeMo differs from toolkits in the first group in two ways: (1) NeMo's core abstraction is a neural module rather than a layer or a tensor and (2) NeMo contains a neural type system that performs various semantic checks.

The main difference between NeMo and toolkits in the second group is that NeMo does not impose any particular structure, e.g. many toolkits require models to follow the encoder-decoder-loss structure. NeMo also does not require configuration files to be in any particular format – users can define models directly using NeMo API. Some toolkits from the second group enforce input-output compatibility between blocks [1], but NeMo does this using a consistent, generic, and extensible type system.

Conceptually, NeMo is similar to PyTorchPipe [1], which follows the task-oriented approach, but allows for arbitrary, flexible sets of components, and also performs compatibility checks. It is essentially an application framework, while NeMo allows developer to use its underlying type and composition system, but not otherwise adopt any of the NeMo run-time functionality.

3 NeMo

The core building block in NeMo is called *Neural Module* (NM). A Neural Nodule represents a *logical* part of a neural network such as a language model, an encoder, a decoder, a data augmentation algorithm, a loss function, or other sets of layers and functions. As the primary abstraction in NeMo, NMs form the basis for describing a model and the process by which that model is trained. Formally, a Neural Module is a component that computes a set of *typed* outputs given a set of *typed* inputs. Inputs and outputs are collections of multidimensional tensors. In the same way that a programmer in an object-oriented language can choose at what level of granularity to define an object, a NeMo user can choose the level of granularity of a Neural Module. A basic rule is that inputs and outputs should "make sense" to expose via an interface. This suggests that a Neural Module is not typically a single neural network layer, but rather a collection of connected layers that "do something useful" such as an encoder, a concatenation operation, a loss function, or a data augmentation.

In our implementation, a NM is a Python class that describes its *input ports* and *output ports* using the type system described below. The current implementation relies on PyTorch, but the abstractions and code does not make any reference to the underlying framework, allowing for applications to be framework-agnostic and for the addition of new backends in the future (see Figure 1).

A NM can compute its output port values given provided input port values. For NMs that contain trainable weights, they should also be able to compute the gradient flows. Implementations of the forward and backward passes are provided by the underlying DL framework. One way to think of a NM is that it is able to "lower" itself into either another set of NMs (recursively), a set of well-defined neural network layers implemented in PyTorch, or in the case of non-trainable NMs, by directly evaluating the output values given the input values.

A NM may be parameterized. For example, an image encoder NM can be parameterized by the number of convolutional layers, filter sizes, dropout values, etc. In this way, a NM defines a parametric family of neural networks, where the variation is explicitly determined by parameter values. Parameters are passed to the NM at construction time via named parameters, and the code that defines how lowering occurs only depends on constructor-time values of these parameters[2].

Similar to functional programming languages, the evaluation of a NM's outputs or gradients cannot effect the evaluation of any other NM's inputs or gradients, except via explicitly linked inputs and outputs which follow standard neural network forward and backward propagation rules. This decoupling helps enforce clean and correct decomposition of complex models according to well-defined interfaces because evaluations cannot have side effects. Furthermore, explicitly defined parameters allow experiment tracking and integration with hyperparameter search tools to be greatly simplified.

This illustrates a major principle of NeMo: *the structure of a neural network and its forward and backward data flows should be determined by values in data structures, not by logic encoded in Python source code*.

3.1 NeMo Core

An application built with NeMo typically consists of 3 required stages and 1 optional stage:

1. Instantiate a NeuralFactory object and the necessary NMs.
2. Define the activation flow DAG by connecting NMs together.
3. (optional) Define callbacks for logging, checkpointing, visualization, and evaluation.
4. Invoke an action such as `train`, `eval`, or `infer`.

NeMo follows a lazy execution model: no computation is done until an action is called. During the definition of the activation flow directed acyclic graph (DAG), NeMo does type checking for the inputs and outputs of connected NMs. This helps catch and debug various errors prior to doing any computations. Once the DAG of modules is defined and action is called, NeMo invokes the DL framework, which we call a *backend*. NeMo is designed to be framework-agnostic, but it currently only supports PyTorch as backend.

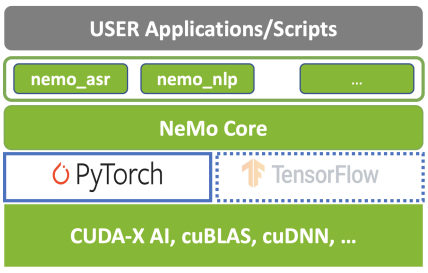

Figure 1: NeMo is a framework-agnostic toolkit which serves as abstraction level between application and DL frameworks backend.

Users can create their own NMs by combining existing NMs or providing an implementation in a particular framework. In practice, any PyTorch `nn.Module` can be easily converted into a NeMo's NM by adding input and output port definitions - i.e. what is expected by and returned from the `forward` function.

Similar to scikit-learn and Keras, NeMo allows users to create callbacks for routines performed during training such as evaluation, logging, and performance monitoring.

[2]This is enforced via convention for now.

3.2 Neural types system

NeMo Core defines the interface and functionality of the `NeuralModule` base class. Each input and output of a NM has a *Neural Type*. Neural Types describe the semantics, axis order, and dimensions of a tensor. The purpose of this type system is to catch semantic and dimensionality errors during model creation and facilitate module re-use. If the output type of one NM output port matches the input type of another NM input port, it is legal to connect these two NMs together, regardless of where they came from or how they are implemented. Because we use static type checking, NeMo can catch semantic and dimensionality errors during the DAG creation stage.

A Neural Type is a mapping from each tensor axis ID to an *Axis Type*. An Axis Type contains semantic and dimensional information of a tensor's axis. Semantic information is represented with the help of "semantic tags" - Python classes related by "is-a" kind of inheritance. For example, if module A's output is of the semantic type `WordEmbedding` while module's B expected input is `Embedding`, and `WordEmbedding` is inherited from `Embedding`, then module B can accept A's output as input but not vice-versa. See Table 3.2 for examples.

A `NeuralType` is constructed from a dictionary, `axis2type`, which maps an axis index to its `AxisType`. For example, the input and output ports of a typical ResNet encoder can be described as follows:

```
input_ports = {"x": NeuralType({0: AxisType(BatchTag),
                                1: AxisType(ChannelTag),
                                2: AxisType(HeightTag, 224),
                                3: AxisType(WidthTag, 224)})}
output_ports = {"output": NeuralType({0: AxisType(BatchTag),
                                      1: AxisType(ImageEmbeddingTag)})}
```

NeMo defines binary comparison operation for any pair of `NeuralType` objects with various comparison results, such as `SAME`, `LESS`, `GREATER`, `DIM_INCOMPATIBLE`, `TRANSPOSE_SAME`, and `INCOMPATIBLE`. This type system also allows for non-tensor objects (such as scalars) and the `root` type which is somewhat analogous to `void*` in C++: a port of `root` type can accept any tensor[3].

Examples of errors that NeMo's type system can catch at model definition time include: "*Ranks match but semantics don't*", "*Concatenating along the semantically wrong dimensions*", and "*Dimensions mismatch*". For example, consider an encoder-decoder model where the decoder expects input in the form of `[batch_dim, time_dim, channel_dim]` and the encoder, written by another developer, outputs a `[time_dim, batch_dim, channel_dim]` tensor. Time and batch dimensions are often dynamic, and if `channel_dim` remains constant, standard frameworks will run smoothly but the model will fail to converge, forcing the developer to find and fix this silent error. However, NeMo will throw a semantic type error at the moment these modules are connected. In this case, the result of type comparison operation will be `TRANSPOSE_SAME` instead of `SAME`.[4].

Output port type	Input port type	Comparison Result
{0: Batch, 1: Channel}	{0: Batch, 1: Spectrogram}	GREATER (INCOMPATIBLE)
{0: Batch, 1: Spectrogram}	{0: Batch, 1: Channel}	LESS (COMPATIBLE)
{0: Batch, 1: Spectrogram}	{0: Batch, 1: Encoded}	INCOMPATIBLE
{0: Batch, 1: Spectrogram}	{0: Spectrogram, 1: Batch}	TRANSPOSE_SAME
{0: Batch, 1: Spectrogram:64}	{0: Batch, 1: Channel:40}	DIM_INCOMPATIBLE
{0: Batch, 1: Spectrogram:64}	{}	SAME

Table 1: Examples of NeMo Neural Types. {} denotes "root" neural type. These examples assume the following: (1) Spectrogram and Encoded types are inherited from the Channel type (2) Module's A output port is connected to module's B input port.

[3] In the future, we plan to add a template type system modeled on a simplified version of C++ templates to support type-safe generics for operations such as concatenation.

[4] It is possible to add *implicit casts* where the system automatically inserts simple operations such as transposition to "fix" simple type mismatches, similar to how C++ can automatically promote `int` to `float`.

3.3 High performance training

NeMo is built to take full advantage of the latest DL hardware such as NVIDIA's Volta and Turing GPUs. It automatically supports *mixed precision* training, using *float16* for computationally intensive operations such as matrix multiplies and convolutions while keeping some things in *float32*, and employing dynamic loss scaling[15].

NeMo also supports gradient accumulation, a technique that accumulates gradients on workers and updates weights only after a certain number of batches have been processed. This allows very large batch simulations on cards with limited RAM and facilitates distributed multi-GPU runs by reducing the amount of inter-worker communication. NeMo also supports multi-GPU and multi-node training using NVIDIA's APEX library[2].

4 NeMo collections

A NeMo collection is the DL equivalent of a software collection of related functions. Common NMs for particular domains are pre-built and packaged into "collections". Currently, NeMo provides collections for automatic speech recognition (ASR) and natural language processing (NLP), but users can easily add new collections. Some of those NMs may come with pre-trained weights. A NeMo collection is the DL equivalent of a software library containing a collection of related functions. In practice, a collection is simply a Python module that defines NM classes, neural types, and associated helper routines.

4.1 Automatic Speech Recognition

`nemo_asr` is a collection of neural modules and helper functions that can be used to train and evaluate Automatic Speech Recognition (ASR) models. It currently supports two model types: CTC-based and sequence-to-sequence attention-based.

4.1.1 CTC-based speech recognition

As an example, we describe the steps necessary to train a Jasper-like ASR model[14] using `nemo_asr`.

First, we create a Neural Module Factory object which manages training and instantiation of neural modules. Jasper uses a `JasperEncoder`, a `JasperDecoderForCTC`, and a `CTCLossNM`. Each of these NMs is passed parameters at construction time - we omit them here to make the code simpler to read.

```
nf = nemo.core.NeuralModuleFactory(...)
data_layer = nemo_asr.AudioToTextDataLayer(...)
jasper_encoder = nemo_asr.JasperEncoder(...)
jasper_decoder = nemo_asr.JasperDecoderForCTC(...)
ctc_loss = nemo_asr.CTCLossNM(...)
```

Next, we define the Directed Acyclic Graph (DAG) of how activations flow from output ports to input ports.

```
spec, spec_len, transcript, transcript_len = data_layer()
encoded, encoded_len = jasper_encoder(audio_signal=spec, length=spec_len)
log_probs = jasper_decoder(encoder_output=encoded)
loss = ctc_loss(log_probs=log_probs, targets=transcript,
                input_length=encoded_len, target_length=transcript_len)
```

In the first line, the `data_layer` produces 4 output ports, which are returned as a tuple. The `jasper_encoder` has two named input ports, which are connected to two of the `data_layer` output ports.

During the definition of this DAG neural type system checks are performed to ensure the correct usage of various modules together. Finally, once the DAG is described, we should call a neural factory's action, such as `train` to trigger the training procedure and data flow.

```
nf.train(tensors_to_optimize=[loss],
         callbacks=[train_cb, saver_cb], ...)
```

4.1.2 Neural modules re-use: attention-based speech recognition

As an illustration of model reuse, we show how to build a different ASR model using attention-based sequence learning, but reusing the same data layer and the Jasper encoder. The model we'll build is conceptually similar to LAS [10]

```
... // instantiate most modules as before
connector = nemo_asr.JasperRNNConnector(...)
decoder = nemo.common.DecoderRNN(...)
seq_loss = nemo.common.SequenceLoss(...)

spec, spec_len, transcript, transcript_len = data_layer()
encoded, encoded_len = jasper_encoder(audio_signal=spec, length=spec_len)
encoded = connector(tensor=encoded)
log_probs, _ = decoder(targets=transcripts, encoder_outputs=encoded)
train_loss = seq_loss(log_probs=log_probs, targets=transcripts)
```

In this example, we switch out `JasperDecoderForCTC` for a `DecoderRNN`, and `CTCLoss` for a `SequenceLoss` NM. Note, that we use a `JasperRNNConnector` NM to correct for the dimensionality mismatch between `JasperEncoder` and `DecoderRNN`. Notice that `DecoderRNN` comes from a different collection, and could be first pre-trained as a stand-alone language model.

4.2 Natural Language Processing

`nemo_nlp` is a collection of neural modules and callback functions which can be used for various NLP-related tasks such as neural machine translation (NMT), language modeling, sentence classification, asr correction, joint intent classification and slot filling. It also supports BERT pre-training and fine-tuning for each task. For BERT, we rely on the implementation from `pytorch-transformers`[7]. We plan to extend this collection in future.

4.2.1 Neural Machine Translation

Here we show how to build `Tranformer-BIG` [19] using `nemo_nlp`.

First, we instantiate the NMs representing logical parts of the model: `TranslationDataLayer`, `TransformerEncoderNM`, `TransformerDecoderNM`, `TransformerLogSoftmaxNM`, and `PaddedSmoothedCrossEntropyLossNM`.

Then we construct the DAG of activation flow that looks like this:

```
src, src_mask, tgt, tgt_mask, labels, sent_ids = train_data_layer()
src_hiddens = encoder(input_ids=src, input_mask_src=src_mask)
tgt_hiddens = decoder(input_ids_tgt=tgt, hidden_states_src=src_hiddens,
                      input_mask_src=src_mask, input_mask_tgt=tgt_mask)
log_softmax = log_softmax(hidden_states=tgt_hiddens)
train_loss = loss(log_probs=log_softmax, target_ids=labels)
```

The code for training and callbacks is similar to the previous examples.

NeMo retains underlying framework's efficiency. In our experiment, this model achieves 29.2 BLEU / 28.5 SacreBLEU on newstest2014 after training for about 15 hours on WMT16 English-German using single machine with 8 GPUs.

5 Conclusions and future work

NeMo addresses many of the issues often encountered in developing DL applications by transferring best practices from software engineering. It operates with a higher level abstraction, the neural module, and introduces a neural type system capable of semantic checks. It also comes with collections of pre-built modules for conversational AI - `nemo_asr` and `nemo_nlp` to make building and re-using deep neural networks easier.

We are working on expanding existing NeMo collections and adding new ones. Also, exploring the right design for a neural type system and the most useful levels of abstractions for modules is an ongoing research direction.

References

[1] Pytorchpipe. https://github.com/ibm/pytorchpipe. Accessed: 2019-08-19.

[2] A pytorch extension: Tools for easy mixed precision and distributed training in pytorch. https://github.com/NVIDIA/apex. Accessed: 2019-08-19.

[3] Pytorch ignite. https://pytorch.org/ignite/. Accessed: 2019-08-19.

[4] Pytorch lightning. https://github.com/williamFalcon/pytorch-lightning. Accessed: 2019-08-19.

[5] Ludwig. http://ludwig.ai. Accessed: 2019-08-19.

[6] Tensor considered harmful. http://nlp.seas.harvard.edu/NamedTensor. Accessed: 2019-08-19.

[7] pytorch-transformers from huggingface. https://github.com/huggingface/pytorch-transformers. Accessed: 2019-08-19.

[8] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

[9] Denny Britz, Anna Goldie, Thang Luong, and Quoc Le. Massive Exploration of Neural Machine Translation Architectures. ArXiv e-prints, March 2017.

[10] William Chan, Navdeep Jaitly, Quoc Le, and Oriol Vinyals. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pages 4960–4964. IEEE, 2016.

[11] François Chollet et al. Keras. https://keras.io, 2015.

[12] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander Rush. OpenNMT: Open-source toolkit for neural machine translation. In Proceedings of ACL 2017, System Demonstrations, pages 67–72, Vancouver, Canada, July 2017. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/P17-4012.

[13] Oleksii Kuchaiev, Boris Ginsburg, Igor Gitman, Vitaly Lavrukhin, Carl Case, and Paulius Micikevicius. Openseq2seq: extensible toolkit for distributed and mixed precision training of sequence-to-sequence models. arXiv e-prints arXiv:1805.10387, 2018.

[14] J. Li, V. Lavrukhin, B. Ginsburg, R. Leary, O. Kuchaiev, J. M. Cohen, H. Nguyen, and R. T. Gaddei. Jasper: An end-to-end convolutional neural acoustic model. arXiv e-prints arXiv:1904.03288, 2019.

[15] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. ICLR, 2017.

[16] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. In Proceedings of NAACL-HLT 2019: Demonstrations, 2019.

[17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[18] Malcolm Reynolds, Gabriel Barth-Maron, Frederic Besse, Diego de Las Casas, Andreas Fidjeland, Tim Green, Adrià Puigdomènech, Sébastien Racanière, Jack Rae, and Fabio Viola. Open sourcing Sonnet - a new library for constructing neural networks. `https://deepmind.com/blog/open-sourcing-sonnet/`, 2017.

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. arXiv preprint arXiv: 1706.03762, 2017.

[20] Ashish Vaswani, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N. Gomez, Stephan Gouws, Llion Jones, Łukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer, and Jakob Uszkoreit. Tensor2tensor for neural machine translation. CoRR, abs/1803.07416, 2018. URL `http://arxiv.org/abs/1803.07416`.