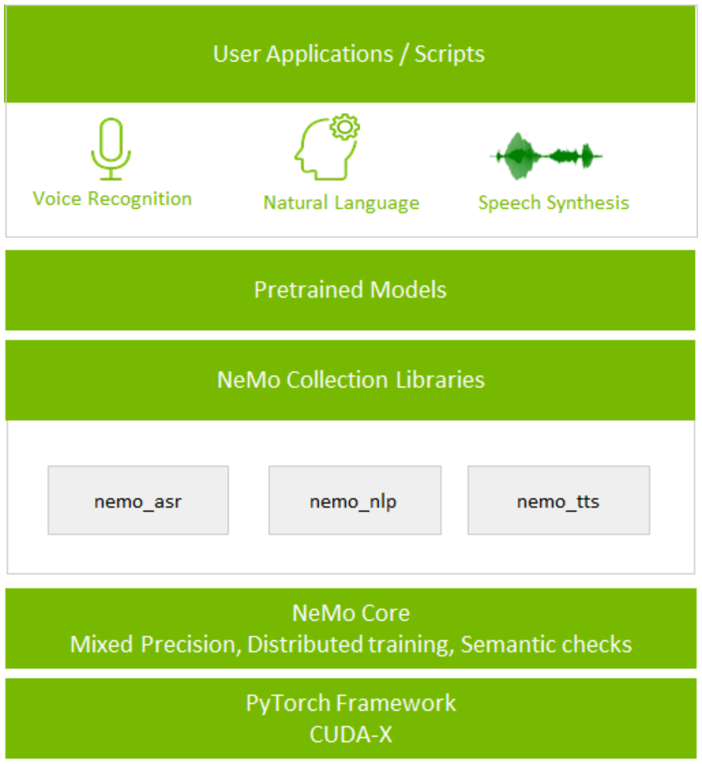
(https://www.nvidia.com/dli)

# 2.0 NLP Projects with NVIDIA NeMo (v0.11)

In this notebook, you'll learn the basics of setting up an NLP project with NVIDIA NeMo (https://developer.nvidia.com/nvidia-nemo). You'll learn what components are needed, where to find them, and how to instantiate them.



NeMo is an open source, high-performance toolkit for building conversational AI applications using a PyTorch backend. NeMo provides a level of abstraction beyond Keras or PyTorch, that makes it possible to easily compose complex neural network architectures using reusable components called *neural modules* (hence the name NeMo). NeMo includes capabilities to scale training to multi-GPU systems and multi-node clusters as well, and provides a straightforward path to model deployment for production real-time inference.

## 2.1 Programming Model

NeMo includes library collections for all aspects of the conversational AI pipeline: speech recognition ( `nemo_asr` ), natural language ( `nemo_nlp` ), and speech synthesis ( `nemo_tts` ). For our projects, we'll use the `nemo-nlp` and some core libraries.

Building an application using NeMo APIs consists of three basic steps:

1. **Create neural modules** after instantiating the NeuralModuleFactory:
   A *NeuralModule* is an abstraction between a network layer and a full neural network, for example: encoder, decoder, language model, acoustic model, etc.
2. **Create the neural graph** of tensors connecting the neural modules together:
   Define all the inputs, outputs, and connections you need to form a pipeline.
3. **Start an "action"** such as `train` for training a network or `infer` obtain a result from a network:
   We can also define *callbacks* for evaluation, logging and performance monitoring.

In [ ]:
```
# Start with the nemo and nlp imports (Ignore the torchaudio error if it app
# Import "inspect" so we can look at method signatures.
import nemo
import nemo.collections.nlp as nemo_nlp
import inspect
```

## 2.2 Create Neural Modules (Step 1)

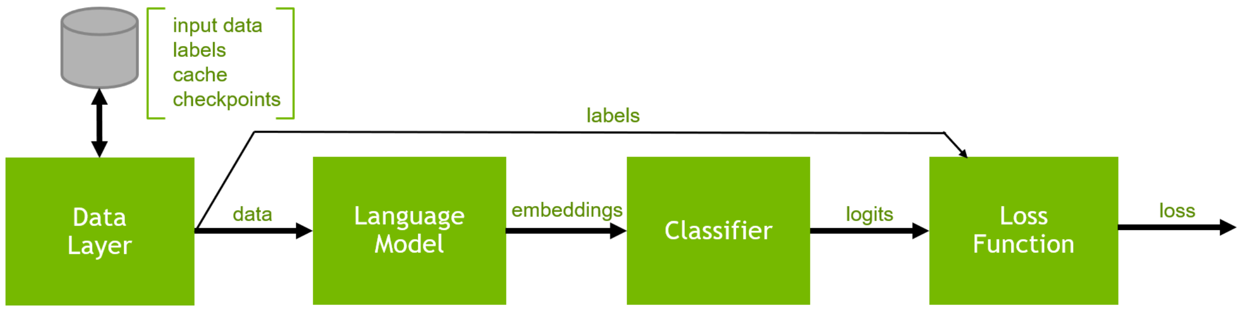
Before creating any neural modules, we must instantiate the `NeuralModuleFactory` object. There are a number of settings that we'll use in the projects to specify directories for intermediate information such as checkpoints and logs. This is how to instantiate the factory.

In [ ]:
```
# Instantiate the NeuralModuleFactory before creating neural modules.
nf = nemo.core.NeuralModuleFactory()
```

Neural Modules can be conceptually classified into four potentially overlapping categories:

- Data Layers - modules that perform extract, transform, load, and feed of the data
- Trainable Modules - modules that contain trainable weights
- Non Trainable Modules - non-trainable module, for example, table lookup, data augmentation, greedy decoder, etc.
- Loss Modules - modules that compute loss functions

For text classification and NER, we will need a data layer, a trainable language model, a trainable classifier, and a loss module (shown as boxes in the diagram below).



It is possible to build your own neural modules, but there are plenty available for us to use already.

### 2.2.1 Data Layer and Tokenizer

Run the following cell to see a list of available NLP data layer neural modules.

In [ ]:
```
# List the callable data layer objects.
[method_name for method_name in dir(nemo_nlp.nm.data_layers)
                  if callable(getattr(nemo_nlp.nm.data_layers, method_name))
                     and method_name[0] not in ['_']
]
```

For text classification project, we'll use the `BertTextClassificationDataLayer` neural module, while for NER we'll use `BertTokenClassificationDataLayer`. Take a look at the class signatures below.

In [ ]:
```
inspect.signature(nemo_nlp.nm.data_layers.BertTextClassificationDataLayer)
```

In [ ]:
```
inspect.signature(nemo_nlp.nm.data_layers.BertTokenClassificationDataLayer)
```

Note that it has a few differences, such as the imported `text_file` and `label_file` parameters needed for NER, rather than the single `input_file` used for the text classification task.

In both cases, we need to also provide the `tokenizer` and `max_seq_length` parameters. For the maximum sequence length, we have a tradeoff here between providing enough information for the system to employ useful attention, while maintaining efficiency of memory and processing time. The maximum size for BERT is 512. Depending on the particular task and constraints, we might typically choose 32, 64, 128, 256, or 512.

## Tokenizer

A *tokenizer* segments text and documents into a given type of token, which may be sentences, sentence pieces, words, word pieces, or even characters. Take a look at the choices currently available within the NeMo library:

In [ ]:
```
# List the callable tokenizer objects.
[method_name for method_name in dir(nemo_nlp.data.tokenizers)
                  if callable(getattr(nemo_nlp.data.tokenizers, method_name))
                     and method_name[0] not in ['_']
]
```

Since we'll be starting with pretrained BERT models and fine-tuning from there, our tokenizer must use the one that the BERT model expects, which happens to be the `NemoBertTokenizer`. In addition to breaking apart the input text, the tokenizer converts each token to an *id* associated with a vocabulary file. This vocabulary can either be provided by the user, or must be calculated when the tokenizer is specified, which may take a bit of time.

The details of the tokenizer instantiation can be abstracted somewhat by using the `get_tokenizer` helper. This is a little easier in practice since it abstracts some details related to exactly which BERT or Megatron pretrained model we choose. For our example, we'll use `bert-base-uncased`.

```
In [ ]: tokenizer = nemo.collections.nlp.data.tokenizers.get_tokenizer(tokenizer_nam
                                                                    pretrained_mo
```

The BERT tokenizer uses *wordpiece* tokenization, and since we have specified an uncased model, we would expect the tokenizer to turn all the words to lower case as well. Run the following cell to see how the tokenizer breaks apart a sentence with a wordpiece algorithm, and encodes it.

```
In [ ]: # Show tokens from an input
        test_sentence = 'How about this stupendous sentence?'
        tokenized = tokenizer.text_to_tokens(test_sentence)
        encoded = tokenizer.text_to_ids(test_sentence)
        print('{}\n{}\n{}'.format(test_sentence, tokenized, encoded))
```

We can now instantiate a nominal data layer neural module.

```
In [ ]: dl = nemo_nlp.nm.data_layers.BertTextClassificationDataLayer(input_file='dat
                                                                  tokenizer=token
                                                                  max_seq_length=
```

## 2.2.2 Language Model and Classifier

Run the following cell to see a list of available NLP trainable neural modules.

```
In [ ]: # List the callable trainable neural module objects
        [method_name for method_name in dir(nemo_nlp.nm.trainables)
                           if callable(getattr(nemo_nlp.nm.trainables, method_name))
                              and method_name[0] not in ['_']
        ]
```

### Language Model

We'll use the `get_pretrained_lm_model` method to specify a language model. We can see a list of available models for us to try.

```
In [ ]: nemo_nlp.nm.trainables.get_pretrained_lm_models_list()
```

It's easy to instantiate this type of neural module. Recall that we already chose 'bert-base-uncased' for the tokenizer. The language model chosen here should match.

```
In [ ]: # Instantiate the language model.
        lm = nemo_nlp.nm.trainables.get_pretrained_lm_model('bert-base-uncased')
```

### Classification Model

For text classification we'll need the `SequenceClassifier` trainable neural module, while the NER project will require the similar `TokenClassifier` neural module. Take a look at the signatures below:

In [ ]:
```
inspect.signature(nemo_nlp.nm.trainables.SequenceClassifier)
```

In [ ]:
```
inspect.signature(nemo_nlp.nm.trainables.TokenClassifier)
```

In both cases, we must provide the `hidden_size`, which we can pull from the language model, and the number of label categories in `num_classes`. We may wish to alter the other default values as well.

In [ ]:
```
# Instantiate the classification neural module
cl = nemo_nlp.nm.trainables.SequenceClassifier(hidden_size=lm.hidden_size, n
```

### 2.2.3 Loss Function

Finally, we define the loss function neural module. For both of our projects, we will use `CrossEntropyLossNM`. Here's a list of what's available:

In [ ]:
```
# List the callable loss common neural module objects
[method_name for method_name in dir(nemo.backends.pytorch.common.losses)
                    if callable(getattr(nemo.backends.pytorch.common.losses, m
                       and method_name[0] not in ['_']
]
```

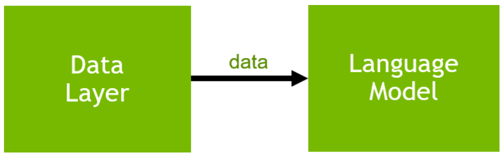
In [ ]:
```
inspect.signature(nemo.backends.pytorch.common.losses.CrossEntropyLossNM)
```

In [ ]:
```
# Instantiate the loss function neural module
lf = nemo.backends.pytorch.common.losses.CrossEntropyLossNM()
```

## 2.3 Create Neural Graph

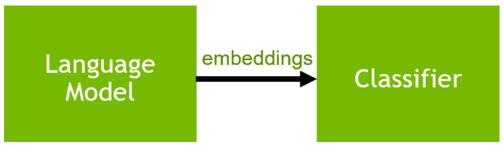
Now that there are four modules in place, defining the graph is just a matter of connecting each of the modules. We'll define the output of each as the input of the next in the pipeline.

To set up the data link, first get the outputs from the data layer.



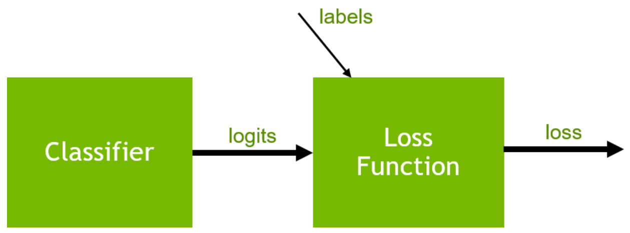
In [ ]:
```
input_data, token_types, attn_mask, labels = dl()
```

Next, determine the embeddings, or outputs from the language model, based on the data inputs provided by the data layer.



In [ ]:
```
embeddings = lm(input_ids=input_data, token_type_ids=token_types, attention_
```

In a similar fashion, define the logits (the raw, non-normalized prediction tensors) generated by the classifier. The loss function outputs are dependent upon both the logits and the labels available from the data layer.



In [ ]:
```
logits = cl(hidden_states=embeddings)
loss=lf(logits=logits, labels=labels)
```

Great! Now that a graph is defined and ready for action.

## 2.4 Start an Action

The actions of training and inference are run from the `NeuralModuleFactory` that you instantiated at the start of the exercise. Though we don't need to actually execute the actions in this notebook, take a look at the method signatures to see what kind of information we'll need to provide.

In [ ]:
```
inspect.signature(nemo.core.NeuralModuleFactory.train)
```

In [ ]:
```
inspect.signature(nemo.core.NeuralModuleFactory.infer)
```

To complete the setup for training in the project notebooks, you'll need to specify some functions for optimization, learning rate policy, and callbacks. The callbacks are invoked during the processing for purposes or recording intermediate parameters. This is useful for saving checkpoints and monitoring results. The signatures can be viewed below, and actual use examples are provided in the project notebooks.

In [ ]:
```
inspect.signature(nemo.core.SimpleLogger)
```

In [ ]:
```
inspect.signature(nemo.core.TensorboardLogger)
```

In [ ]:
```
inspect.signature(nemo.core.EvaluatorCallback)
```

In [ ]:
```
inspect.signature(nemo.core.CheckpointCallback)
```

## Congratulations!

You've explored the NeMo API and learned:

- The three parts of a project in NeMo are: neural modules, neural graphs, and an action
- Neural modules we need for these projects are data layers, language models, classifiers, and loss functions
- Neural graphs are created by defining the inputs and outputs between the neural modules
- Actions are `train` and `infer`

You are ready to build the text classification project.

Move on to [3.0 Build a 3-class Text Classifier (030_TextClassification.ipynb)](030_TextClassification.ipynb).

[(https://www.nvidia.com/dli)](https://www.nvidia.com/dli)