Building Speech AI Applications

NVIDIA

Table of Contents

Preface

Who is this e-book series for?

This e-book series is intended for business decision owners and developers within an enterprise who would like to understand the core concepts of speech AI and how to build and deploy applications for different use-cases.

The series consists of three parts:

Part 1: Introduction to Speech AI
Part 2: End-To-End Speech AI Pipelines
Part 3: Building Speech AI Applications

Over the course of this e-book series, you will learn about:

- Speech AI and its major building blocks.
- The role of speech AI in industries.
- Different components and technologies that are involved in building end-to-end speech AI pipelines
- How to get started with speech AI in your business with NVIDIA Riva.

Part 3: Building Speech AI Applications

In the previous parts of this e-book series, we have looked at the landscape of Speech AI and its main building blocks and pipelines, and learned how it can yield great value across a broad range of industries. In Part 3 of this e-book series, we will delve into how developers at enterprises can get started with developing real-time Speech AI applications and deploying models in production for high-performance inference with minimal effort.

Creating an AI/ML model from scratch to solve a business problem is a capital-intensive and time-consuming endeavor. In the current realm of deep learning based approaches, creating a model requires the collection and curation of massive datasets, developing one or more neural network architectures, continuously training and optimizing the models for accuracy, throughput, and latency. Building models from scratch requires significant expertise and investment. Enterprises frequently start from pre-trained models, so that their developers can spend more time on customizing the models for specific domain applications. In production, the models need to be updated with the latest data and tested comprehensively. Open source repositories provide unpredictable updates and leave users to the goodwill of the community in case of issues. NVIDIA provides state-of-the-art production-quality pre-trained models that deliver high throughput and accuracy.

NVIDIA Riva is a GPU-accelerated SDK with automatic speech recognition (ASR) and text-to-speech (TTS) skills for building and developing speech AI applications. Riva is fully customizable and optimized for maximum performance on NVIDIA GPUs in the cloud, data centers, at the edge, and embedded devices. NVIDIA Riva is available as a set of containers and pretrained models, free of charge, from the NVIDIA GPU Cloud™ to members of the NVIDIA Developer Program. You can also get support for large-scale Riva deployments with NVIDIA AI Enterprise software. Support includes access to NVIDIA AI experts, training, and knowledge-base resources.

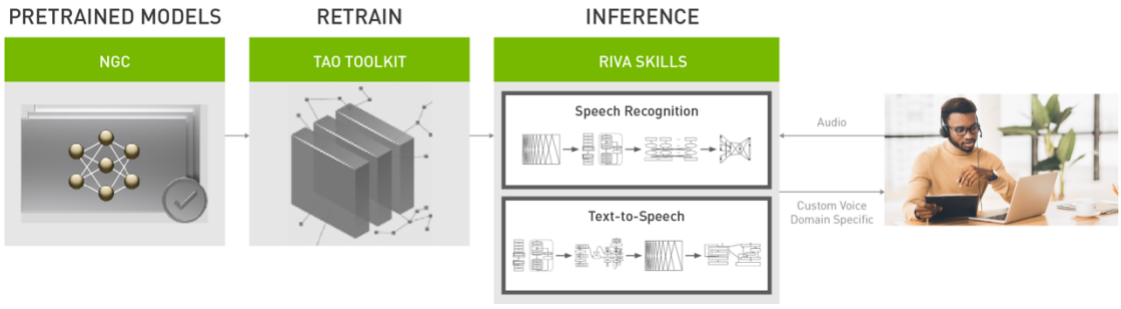

Figure 1: Building Speech AI application with NVIDIA Riva

The NVIDIA Riva platform includes world-class automatic speech recognition (ASR) that is customizable across domains, and controllable text-to-speech (TTS) for expressive synthetic voices.

With the NVIDIA Riva Speech AI platform, you can:

- Build end-to-end Speech AI applications using pre-trained NVIDIA models and skills available at NVIDIA GPU Cloud™ (NGC).
- Train and fine-tune your model on domain-specific data, with NVIDIA TAO toolkit.
- Optimize neural network performance and latency using NVIDIA® TensorRT™
- Deploy AI applications with NVIDIA Triton™ Inference Server.

All out-of-the-box Riva models are accessible via NVIDIA GPU Cloud (NGC™), a hub of GPU-optimized AI software that features:

- State-of-the-art Speech AI models, trained on thousands of hours of public and proprietary speech data, optimized for high-performance training and inference on GPUs.
- Customizability via the NVIDIA TAO Toolkit and deployability to NVIDIA Riva with a few commands.
- Jupyter notebooks and tutorials to get started.

Riva Automatic Speech Recognition

Getting Started with Out-Of-The-Box Skills

The NVIDIA Riva skill provides high-quality pre-trained ASR models across five languages:

- English
- Spanish
- German
- Russian
- Mandarin

Upgraded models and new languages are released regularly. With pretrained models, you can easily design personalized real-time call center experiences, virtual assistants, smart kiosks, and live transcription applications.

Language	Language Code	Acoustic Model	Language Model	Punctuation	text Norm	Pipeline
English	en-US	✓ (16700 hrs)	✓	✓	✓	Streaming Offline
Spanish	es-US	✓ (2800 hrs)	✓	✓	✓	Streaming Offline
German	de-DE	✓ (3500 hrs)	✓	✓	✓	Streaming Offline
Russian	ru-RU	✓ (1700 hrs)	✓	Coming Soon	Coming Soon	Streaming Offline
Mandarine*	zh-CN	✓ (2600 hrs)	✓	Coming Soon	Coming Soon	Streaming Offline

Figure 2: Pre-trained Riva ASR Models

The Riva quick start guide is the most up-to-date resource to quickly set up the Riva Speech AI suite, including ASR skills.

Riva can be deployed either on-prem, in the data center, in any cloud, embedded, or at the edge. Either using the CLI tool or Helm chart, setting up Riva services involves only several simple steps as demonstrated in Figure 3.

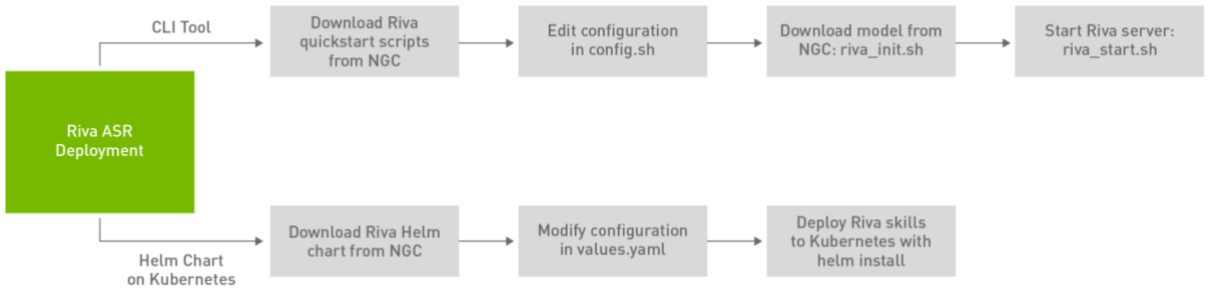

Figure 3: Riva deployment steps with CLI tool and Helm chart

Deployment Using CLI Tool

The following steps enable you to quickly deploy pre-trained models, either on a local workstation, remote server, or a virtual cloud instance, and run a sample client application. Use the CLI tool on NVIDIA GPU Cloud to download Riva quickstart resources from the command-line. Note that the VERSION_TAG should be modified to the latest Riva release.

```
export VERSION_TAG="2.1.0"
ngc registry resource download-version
nvidia/riva/riva_quickstart:${VERSION_TAG}
```

After downloading the resources to your server, a few basic steps are needed:

- Configure the deployment in `config.sh`.
- Download, optimize, and prepare the models for your platform by running `riva_init.sh`.
- Launch the Riva skills server, with `riva_start.sh`.

Configuring Skills

The `config.sh` file contains a set of flags and values to specify which services and models are needed to initiate and start the configuration. Open `config.sh` and modify the following important sections:

a). **Enable/Disable Riva Services**
To enable the server for a particular capability, you should set the value of that service to true. Specifically, to enable Riva ASR skills we will set the service_enabled_ASR flag to true.

```
# Enable or Disable Riva Services
service_enabled_asr=true                          ## MAKE CHANGES HERE
```

b). **Set the Model Location**
`riva_model_loc` specifies the locations where Riva assets are stored. If an absolute path is specified, the data will be written to that location. Otherwise, a docker volume will be used (default). `riva_init.sh` will create a `rmir` and `models` directory in the volume or path specified.

```
# Models ($riva_model_loc/models)
# During the riva_init process, the RMIR files in $riva_model_loc/rmir
# are inspected and optimized for deployment. The optimized versions are
# stored in $riva_model_loc/models. The riva server exclusively uses these
# optimized versions.

riva_model_loc="riva-model-repo"                        ## MAKE CHANGES HERE
```

Downloading Required Models and Containers from NGC

The next step is to run the `riva_init.sh` script.

```
export NGC_API_KEY=<your_api_key>
bash riva_init.sh
```

It performs the following actions under the hood:

- Downloads the Riva containers.
- Downloads the Riva models in RMIR (Riva Model Intermediate Representation) format, as defined and enabled in `config.sh`, from NGC into a subdirectory at `$riva_model_loc/rmir`.
- Optimizes the models with NVIDIA TensorRT by executing `riva-deploy` for each of the RMIRs at `$riva_model_loc/rmir` to generate their corresponding Triton Inference Server model repository at `$riva_model_loc/models`.

An NGC API key needs to be provided for it to work. The key can be provided either through the environment variable NGC_API_KEY or as a configuration file, which in turn, is automatically generated by running "`ngc config set`".

Launching the Server

After downloading the required models and containers, the Riva Services server can be started by running:

```
bash riva_start.sh
```

Upon successful startup, you will see:

```
...
 > Riva waiting for Triton server to load all models...retrying in 1 second
 > Triton server is ready...
```

To inspect what happens under the hood, run the following command to inspect server logs:

```
docker logs riva-speech
```

Issuing the command details the models being loaded by the Triton server. This is especially handy for debugging issues that might occur, for example during the deployment of a custom model.

```
...
I0525 04:52:57.410936 95 grpc_server.cc:4375] Started GRPCInferenceService
at 0.0.0.0:8001
I0525 04:52:57.412354 95 http_server.cc:3075] Started HTTPService at
0.0.0.0:8000
I0525 04:52:57.454792 95 http_server.cc:178] Started Metrics Service at
0.0.0.0:8002
```

Deployment Using Kubernetes and Helm Chart

In the previous section, you deployed NVIDIA Riva using basic shell commands. As convenient as this method is during development, it becomes impractical for large-scale production deployment, that is, when managing larger numbers of servers and services. Riva is designed for Speech AI at scale, with a push-button deployment solution using Helm charts to serve models across many servers efficiently.

You can automate the steps that go from pre-trained models to optimized skills deployed in the cloud and in the data center using the Riva AI Services Helm Chart. To deploy, you can use a single command to download, set up, and run the entire Riva application or individual services through Helm charts on Kubernetes clusters.

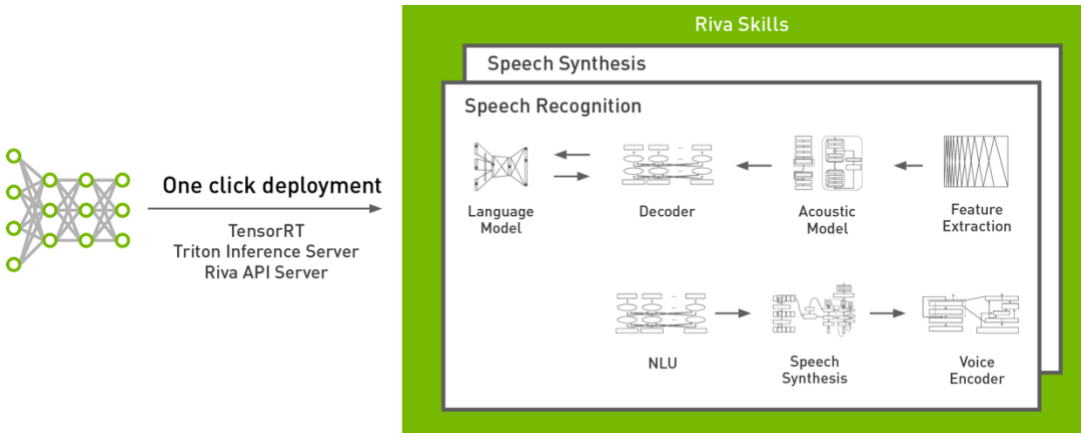

Figure 4: Helm Chart to Deploy Models to Production

Introducing Kubernetes and Helm Charts

Kubernetes, also known as K8s, is an open-source platform for automating deployment, scaling and managing containerized applications. Kubernetes includes support for GPUs, which enables enterprises to scale up training and inference deployment to multi-cloud GPU clusters seamlessly. Helm is an application package manager running on top of Kubernetes. It lets you create Helm charts where you can define, install, and upgrade Kubernetes applications.

The Riva Helm chart performs several functions including:

Pulls docker images from NGC for the Riva Services API server, Triton Inference Server, and utility containers for downloading and converting models.

- Generates the Triton Inference Server model repository.
- Starts the Triton Inference Server with the appropriate configuration.
- Exposes the Triton Inference Server and Riva servers as Kubernetes services.

Example pre-trained models are released with Riva for each of the services. The Helm chart comes pre-configured for downloading and deploying all of these models. The Helm chart configuration can be modified for custom use cases. You can change settings related to which models to deploy, where to store them, and how to expose the services.

Downloading Riva Services Helm chart

To deploy Riva, a functioning Kubernetes environment with a GPU (NVIDIA Volta or later) is required. This can be either on-prem, in-cloud e.g., AWS, GCP, EKS, or within a managed Kubernetes environment so long as the environment has GPU support enabled. Set up your environment by following the installation instructions for Kubernetes with GPU.

Next, download the Riva AI Services Helm chart from NGC with the following commands. Replace `NGC_API_KEY` with your NGC key, and `VERSION_TAG` with the latest Riva release.

```
export NGC_API_KEY=<your_api_key>
export VERSION_TAG="2.1.0"
helm fetch https://helm.ngc.nvidia.com/nvidia/riva/charts/riva-api-
${VERSION_TAG}.tgz \
        --username=\$oauthtoken --password=$NGC_API_KEY --untar
```

The preceding commands create a new directory called Riva-API in your current working directory.

Installing and Deploying Riva Skills

After downloading the Helm chart to your server, use the following two steps to install and deploy Riva skills:

Modify the values.yaml file in the `riva-api` directory to suit your case

Deploy Riva skills to Kubernetes cluster with one line of command

```
helm install riva-api riva-api
```

More information about Riva Kubernetes deployment can be found at Kubernetes — NVIDIA Riva Skills.

Generating High-Performance Inference

During inference, the trained model and associated code are deployed in the data center or public cloud, embedded, or at the edge to make predictions on new data. An inference server is used to respond to multiple users using deep learning models in the backend. This process is complex for several reasons:

- Diversity: the DL/ML landscape is diverse, with several good frameworks for training and inference. Without a unified inference environment that can cater to models from different frameworks, converting between one another is necessary, which might be challenging.
- Quality of service (QoS): different types of queries (realtime, batch) have different QoS requirements, most notably latency, and throughput. These goals are often conflicting and challenging to optimize.
- Cost: the cost of deployment becomes critical when scaling out to serving millions of customers. Disparate serving solutions can increase the cost of deployment and maintenance.

Often, organizations end up having multiple, disparate inference serving solutions, per model, per framework, or per application.

NVIDIA Triton was created to address these challenges, providing a unified serving solution that works with multiple frameworks on multiple hardware environments. Triton provides scalable and simplified inference serving, processing multiple inference requests on multiple GPUs for many neural networks or ensemble pipelines at the same time. Triton is responsible for the context switching of networks between one request and another.

Riva leverages Triton under the hood for high-performance inference. For Speech AI applications, it is important to keep the latency below a given threshold. This usually translates into execution of inference requests as soon as they arrive. However, to saturate the GPUs and increase throughput,

one must increase the batch size and delay the inference execution until more requests are received and a bigger batch can be formed. These two goals of minimizing latency and maximizing throughput are usually conflicting, hence must be balanced. One common strategy is to set a latency threshold under which the goal is to maximize the throughput. Triton's static and dynamic batching features optimize this trade-off.

Using Python Client

Once the Riva server is up and running with out-of-the-box models, you can send inference requests querying the server. Riva comes with a sample client docker container and a comprehensive suite of examples and tutorials in C++ and Python Jupyter notebook format. Start the client container with:

```
bash riva_start_client.sh
```

From inside the client container, you can try the different services using the provided Jupyter notebooks by running:

```
jupyter notebook --ip=0.0.0.0 --allow-root --notebook-dir=/work/notebooks
```

Up to date Riva demos and tutorials can be found on Github. The next few steps illustrate a basic ASR example using Python.

First step is to import the utility libraries.

```
import io
import grpc


import riva_api.riva_asr_pb2 as rasr
import riva_api.riva_asr_pb2_grpc as rasr_srv
import riva_api.riva_audio_pb2 as ra
```

Next, establish a GRPC connection to the server, which listens locally at port 50051 by default (if deployed on the local machine).

```
channel = grpc.insecure_channel('localhost:50051')


riva_asr = rasr_srv.RivaSpeechRecognitionStub(channel)
```

The next step is to read an audio file:

```
path = "./audio_samples/en-US_sample.wav"
with io.open(path, 'rb') as fh:
    content = fh.read()
```

Then set up an ASR request object:

```
# Set up an offline/batch recognition request
req = rasr.RecognizeRequest()
req.audio = content                                    # raw bytes
req.config.language_code = "en-US"                     # Language code of the
audio clip
req.config.max_alternatives = 1                        # How many top-N
hypotheses to return
req.config.enable_automatic_punctuation = True         # Add punctuation when
end of VAD detected
req.config.audio_channel_count = 1                     # Mono channel
```

And finally, initiate an offline transcription request and print out the results:

```
response = riva_asr.Recognize(req)
asr_best_transcript = response.results[0].alternatives[0].transcript
print("ASR Transcript:", asr_best_transcript)


print("\n\nFull Response Message:")
print(response)
```

For further examples on integrating Riva ASR skills into a complete speech AI application, see the NVIDIA Riva sample application repository.

Using Clients in a New Programming Language

All Riva Speech skills are exposed using gRPC to maximize compatibility with existing software infrastructure and ease of integration. The gRPC framework officially supports twelve languages, including C++, Java, Python, and Golang, with unofficial support for many others.

The gRPC services and messages/data structures are defined with protocol buffer definition files, which are available on GitHub. Using these files, you can generate Riva AI Services bindings to any supported programming language of your choice. The generated code can be compiled into your application, with the only additional dependency being the gRPC library.

Customizing Speech Recognition Skills

After you have tested Riva out-of-the-box ASR skills, the next step would be to customize them further to enhance accuracy for your applications to understand industry-specific jargons, different languages, accents, and dialects. There are mainly two groups of techniques:

- Deployment/inference time techniques: these techniques allow easily modifying the behavior of an existing model at deployment time or during inference while passing additional inputs.
- Train time techniques: these techniques involve training a new model or fine-tuning existing models with new data.

Figure 4 is a flow diagram showing the Riva speech recognition pipeline along with the possible customizations. The components in the Riva ASR pipeline include:

- **Feature extractor:** The audio signal first passes through a feature extractor, which segments the data into blocks then converts them from temporal domain to frequency domain (spectrogram or Mel Spectrogram).
- **Acoustic model:** Spectrogram data is then fed into an acoustic model, which outputs probabilities over characters (or more generally, text tokens) at each time step. Some acoustic models supported by Riva are: Jasper, QuartzNet and Citrinet and Conformer-CTC.
- **Decoder and language model:** A decoder converts this matrix of probabilities into a sequence of characters, which in turn make up words and sentences. A language model can give a sentence score indicating the likelihood of this sentence appearing in its training corpus. An advanced decoder can inspect multiple hypotheses while combining the acoustic model score and the language model score.
- **Punctuation and Capitalization:** The text produced by the decoder comes without punctuation and capitalization. A Punctuation and Capitalization model enhances the text with proper sentence punctuation and word capitalization for easier human consumption.
- **Inverse Text Normalization:** Finally, Inverse Text Normalization (ITN) rules are applied to transform the punctuated text into a desired written format for better readability.

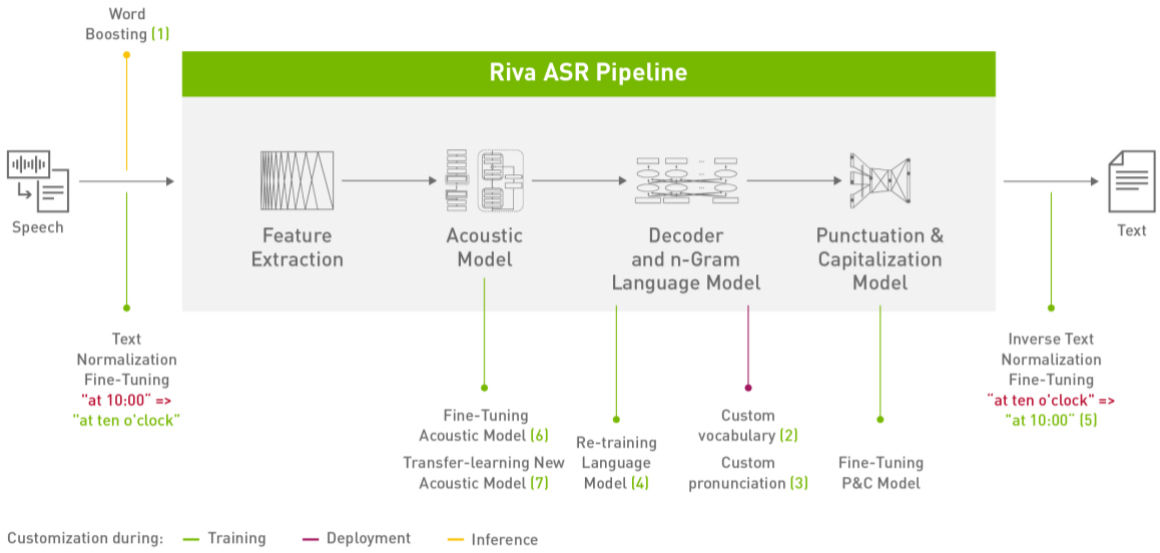

Figure 5: Riva Speech Recognition Customizations

In Table 1, the corresponding customizations are listed in increasing order of difficulty and effort:

Table 1. Riva Customization Techniques

Techniques	Difficulty	What it Does	When to Use	How to Use
1. Word boosting	Quick and easy	Extend the vocabulary while increasing the chance of recognition for a provided list of keywords.	When you know that certain words or phrases are important in a particular context.	Demo notebook
2. Custom vocabulary	Easy	Extend the default vocabulary to cover new words of interest.	When the default model vocabulary does not sufficiently cover the domain of interest.	Demo notebook
3. Custom pronunciation (Lexicon mapping)	Easy	Explicitly guide the decoder to map pronunciations (i.e., token sequences) to specific words	When you know a word can have one or several possible pronunciations.	Demo notebook
4. Retrain language model	Moderate	Train a new language model for the application domain to improve the recognition of domain-specific terms.	When domain text data is available.	Demo notebook
5. Create new inverse text normalization rules	Moderately hard	Map sequence of transcribed spoken words into a desired written format.	When a particular written format is required.	Tutorial
6. Fine-tune an existing acoustic model	Moderately hard	Fine-tune an existing acoustic model using a small amount of domain data to better suit the domain.	When transcribed domain audio data (10h-100h) is available, and other easier approaches fall short.	Citrinet, Jasper and Quartznet notebook
7. Train a new acoustic model	Hard	Train a brand new acoustic model from scratch or with cross-language transfer learning, using thousands of hours of audio data.	Only recommended when adapting Riva to a new language/dialect.	Tutorials

Deploy/Inference-Time Techniques

Model adaptation is a group of techniques that help a pre-trained model/skills adapt to new application scenarios in a quick and easy manner without fine-tuning or retraining.

Word Boosting

Of all the adaptation techniques, word boosting is the easiest and quickest to implement. Word boosting allows you to bias the ASR engine to recognize particular words of interest at request time, by giving them a higher score when decoding the output of the acoustic model.

What you need to do is to pass a list of words of importance to the model along with a weight as extra context to the API call, as shown in the example below:

```
# Word Boosting
boosted_lm_words = ["BMW", "Ashgard"]
boosted_lm_score = 20.0
speech_context = rasr.SpeechContext()
speech_context.phrases.extend(boosted_lm_words)
speech_context.boost = boosted_lm_score
config.speech_contexts.append(speech_context)


# Creating StreamingRecognitionConfig instance with config
streaming_config = rasr.StreamingRecognitionConfig(config=config,
interim_results=True)
```

Word boosting provides a quick and temporary adaptation for the model to cope with new scenarios, such as recognizing proper names and products, new or domain-specific terminologies. For OOV (Out Of Vocabulary) words, the word boosting functionality will temporarily extend the vocabulary at inference time only for that request but will not affect any other request. You have to specify the list of boosted words at every request explicitly. Other adaptation methods such as custom vocabulary and lexicon mapping provide a more permanent solution, which affects every subsequent request.

Custom Vocabulary

The decoder deployed by default in Riva is a lexicon-based decoder and only emits words that are present in the provided vocabulary file. That means, domain-specific words that are not present in the vocabulary file will not have a chance of being generated.

The decoder vocabulary can be expanded at the following two occasions:

- At Riva build time: When building a custom Riva model, passing the extended vocabulary file to the `` `--decoding_vocab=<vocabulary_file>` `` parameter of the build command. Further details on Riva model build are in the subsequent sections.
- Out of the box vocabulary files for Riva languages can be found on NGC, for example, for English, the vocabulary file named `` `flashlight_decoder_vocab.txt` `` can be found at this NGC model page.
- After deployment: For a production Riva system, the lexicon file can be modified, extended and will take effect after a server restart.

It is noted that the greedy decoder (available during the riva-build process under the flag `` `--decoder_type=greedy` ``) is not vocabulary-based and hence can produce any character sequence.

Custom Pronunciation and Lexicon Mapping

When using the Flashlight decoder, the lexicon file provides a mapping between vocabulary dictionary words and its pronunciation, for example, like SentencePiece tokens for many Riva models.

Modifying the lexicon file serves the following two purposes:

- Extends the vocabulary.
- Provides one or more explicit custom pronunciations for a specific word. The example below provides several different ways to pronounce the word “manu”:

```
manu _ma n u
manu _man n n ew
manu _man n ew
```

Train-Time Techniques

Train-time techniques to customize Riva include: training new language models, fine-tuning acoustic models and creating new inverse text normalization rules. See Table 1 for the links to step-by-step guides to carry out these techniques.

With no prior expertise in AI or deep learning, developers and ML practitioners can customize highly optimized state-of-the-art pre-trained models for their use-case to achieve a high level of accuracy. This technique is known as *“Transfer Learning”*. Transfer learning is a popular technique that can be used to leverage learned features from an existing neural network model to a new one. Transfer learning is often used in instances where creating a large training dataset is not feasible.

NVIDIA Train-Adapt-Optimize (TAO) toolkit is an easy-to-use Python-based toolkit for taking purpose-built, pre-trained neural models and customizing them with your own data. The goal of TAO is to make optimized, state-of-the-art, pre-trained models easily retrainable on custom enterprise data with zero coding, thereby accelerating model development time up to 10x compared to training from scratch. It can reduce an 80-hour workload to an 8-hour one, reducing a data scientist’s workload by 90% and facilitating more train-test iterations in the same time frame.

The most important differentiator of the TAO toolkit is that it follows the zero-coding paradigm and comes with a set of ready-to-use Python scripts and configuration specifications with default parameter values that enable you to kick-start training and fine-tuning.

With a basic understanding of deep learning and minimal coding, TAO enables users to:

- Increase accuracy of Speech AI models by fine-tuning on proprietary data.
- Retrain models to adapt them for domain-specific applications by training on custom data.
- Achieve highest training performance with underlying CUDA-X AI libraries, automatic mixed precision, and Tensor Cores.
- Integrate fine-tuned models as real-time services for inference on NVIDIA Riva easily.

In the workflow diagram shown below, a user typically starts with a pre-trained model from NGC. The other input is the user’s own dataset. The dataset is fed into the data converter, which can augment the data while training to introduce variations in the dataset. This is very important in training as the data variation improves the overall quality of the model and prevents overfitting.

Figure 6 shows how to train and deploy an end-to-end ASR pipeline using pre-trained models, TAO, and Riva.

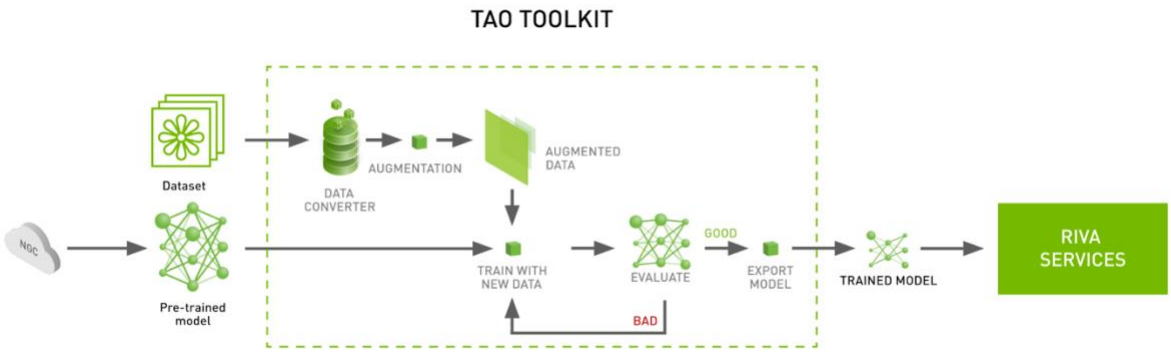

Figure 6: Training and Deployment of an End-to-end ASR Pipeline

Once the dataset is prepared and augmented, the next step in the training process is to start training. The training hyperparameters are chosen through the TAO specification file. After the first training phase, users evaluate the model against a test set to see how the model works on data it has never seen before, a.k.a., the test set. If accuracy is not as expected, then the user might have to tune some hyperparameters and fine-tune the model. Training is a very iterative process, so you might have to try a few times before converging on a satisfactory model.

After training and optimizing your model for the best performance possible, the next step is to deploy the model for production so that it can perform inference.

Acoustic Model

When the other easier customization approaches have failed to address accuracy issues in challenging situations brought about by significant acoustic factors, such as different accents, noisy environments, or poor audio quality, the next step should be fine-tuning acoustic models.

NVIDIA recommends fine-tuning ASR models with sufficient data approximately, on the order of 100 hours of speech or more. In cases of smaller datasets, such as ~10 hours, appropriate precautions should be taken to avoid overfitting to the domain, which could lead to sacrificing significant accuracy in the general domains, also known as, “catastrophic forgetting”. If fine-tuning is done on this small dataset, mix it with other larger datasets (“base”). In the case of the English language, for example, NeMo provides a list of public datasets that can be used for mixing.

Use lossless audio formats if possible. The use of lossy codecs such as MP3 can reduce quality. As a regular practice, use a minimum sampling rate of 16kHz as far as possible. Augmenting training data with noise can improve the model’s ability to cope with noisy environments. Adding background noise to audio training data can initially decrease accuracy but increase robustness in the long run.

Language Model

Custom language models (LM) can provide a permanent solution to improve the recognition of domain-specific terms and phrases. *Riva currently does not support LM fine-tuning*, however a

domain-specific custom LM can be mixed with a general domain LM using a process called interpolation.

When a substantial amount of raw text is available, a custom LM can be trained from scratch. Riva supports n-gram models trained and exported from either NVIDIA TAO Toolkit or KenLM. See Riva documentation for details.

For Riva ASR models in production, all the transcribed text in the training data is aggregated for training the language models. Limit the vocabulary size if using scraped text. Many online sources contain typos or ancillary pronouns and uncommon words. Removing these can improve the language model. When the text belongs to a narrow, niche domain, there might be an impact to the overall ASR pipeline in recognizing general domain language; this is a trade-off. Mixing domain text with general text would yield a balanced representation.

Punctuation Model

Automatic Speech Recognition (ASR) systems typically generate text with no punctuation and capitalization of the words. In Riva, the punctuation and capitalization (P&C) model is responsible for formatting the text with both punctuation and capitalization, making it easier for human readers to consume.

Your P&C model should be customized when the out-of-the-box model does not perform well in the application context, such as when applying to a new language variant.

Refer to the TAO tutorial for details on how to train, fine-tune, and deploy a custom P&C model.

Inverse Text Normalization

Text Normalization converts text from written form into its verbalized form and is used as a preprocessing step before Text-to-Speech (TTS). It may also be used for preprocessing Automatic Speech Recognition (ASR) training transcripts.

On the other hand, Inverse text normalization (ITN) is a part of the Automatic Speech Recognition (ASR) post-processing pipeline. ITN is the task of converting the raw spoken output of the ASR model into its written form to improve text readability.

Riva implements the NeMo ITN, which is based on weighted finite-state transducer (WFST) grammars. The tool uses Pynini to construct WFSTs, and the created grammars can be exported and integrated into Sparrowhawk, which is an open-source version of the Kestrel TTS text normalization system, for production. To learn more on how to build grammars from the ground-up, consult the NeMo Weighted Finite State Transducers (WSFT) NeMo Weighted Finite State Transducers (WSFT) tutorial.

Deploying Your Custom ASR Model into Riva

Riva handles the deployment of full pipelines, which can be composed of one or more NVIDIA TAO models and other pre-/post-processing components. Additionally, the TAO models must be exported to an efficient inference engine and optimized for the target platform. The process of gathering all the required artifacts, such as: models, files, configurations, and user settings, and generating the inference engines, are referred to as the *Riva model repository generation.*

A TAO model is first exported to Riva format (.riva file). Riva ServiceMaker is a utility that combines all the necessary assets such as: model weights and vocabulary, model configurations, and so on, and deploys them onto the target environment. This process is illustrated in Figure 7.

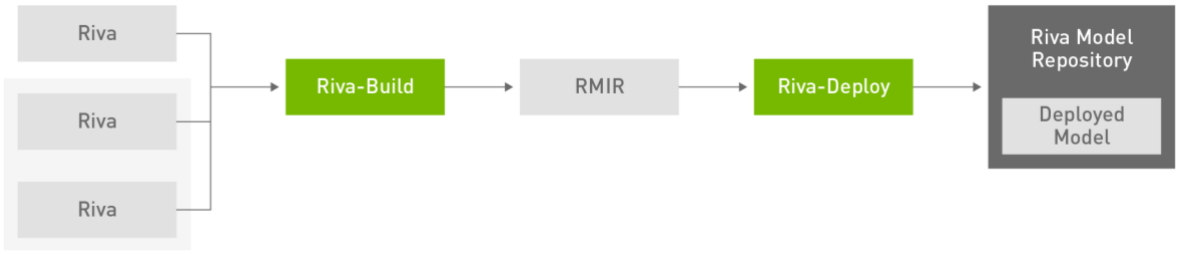

Figure 7: Deployment of TAO Models using Riva ServiceMaker

The Riva model repository generation is done in two phases using the two main components of the Riva ServiceMaker. Figure 7 shows how Riva ServiceMaker consists of two main components: `riva-build` and `riva-deploy`.

Building and Deploying Models as Skills

Before you deploy your .riva models, you need to build them. During the build phase, all the necessary artifacts (models, files, configurations, and user settings) required to deploy a Riva service are gathered together into an intermediate file called RMIR (Riva Model Intermediate Representation).

The `riva-build` tool is responsible for the combination of one or more exported models (.riva files) into a single file containing an intermediate format called Riva model intermediate representation (.rmir). This file contains a deployment-agnostic specification of the whole end-to-end pipeline, along with all the assets required for the final deployment and inference. Refer to the Riva documentation to learn more.

At this stage, the `.rmir` file can be deployed using the ServiceMaker `riva-deploy` tool. The riva-deploy tool takes as input one or more Riva Model Intermediate Representation (RMIR) files and a target model repository directory and is responsible for performing the following functions:

- Adds the Triton Inference Server custom backends for pre- and post-processing specifically for the given model.
- Generates the TensorRT engine for the input model.
- Generates the Triton Inference Server configuration files for each of the modules: preprocessing, post-processing, and the model.
- Creates an ensemble configuration specifying the pipeline for the execution and finally writes all those assets to the output model repository directory.

Setting Up Riva ASR for New Language Workflow

There are over 6500 spoken languages in use in the world today, most of which do not have commercial ASR products. Riva provides ready-to-use workflow, tools, and guidance to quickly, systematically, and bring up ASR service for a new language with ease.

Beyond the data collection phase, the Riva new language workflow is divided into three major stages:

- Data preparation
- Training and validation
- Riva deployment

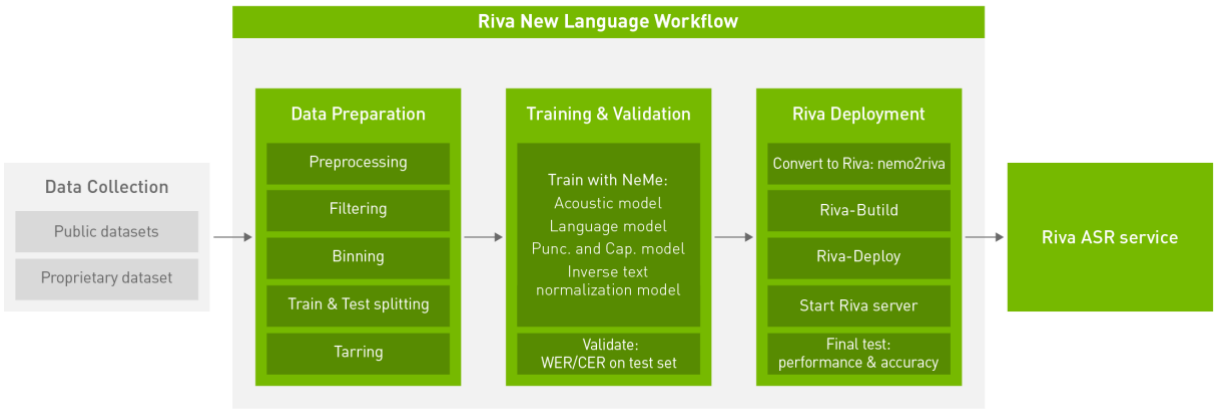

Figure 8: Riva New Language Workflow

Data Collection

When adapting Riva to a whole new language, a large amount of high-quality transcribed audio data is critical for training high-quality acoustic models.

Where applicable, there are several significant sources of public datasets that we can readily leverage. Examples include:

- Mozilla Common Voice (MCV)
- Multilingual LibriSpeech (MLS)
- Voxpopuli

In addition to training Riva world-class models, Riva has access to proprietary datasets. The amount of data for Riva models in production ranges from ~1700 hours to ~16700 hours.

Data Preparation

The data preparation phase carries out a series of preparation steps required to convert the diverse raw audio datasets into a uniform format that can be digested by the NVIDIA NeMo toolkit efficiently.

The data preparation steps include:

- Data Preprocessing
- Data Cleaning and Filtering
- Binning
- Train and Test Splitting
- Compressing Data Files

Data Preprocessing

Audio data acquired from various sources are inherently heterogeneous in terms of file format, sample rate, bit depth, number of audio channels, and so on. Therefore, as a preprocessing step, a separate data ingestion pipeline for each source must be built and then converted to a common format with the following characteristics:

- Wav format
- Bit depth: 16 bits
- Sample rate of 16 Khz
- Single audio channel

Dataset ingestion scripts are used to convert the various datasets into the standard manifest format expected by NeMo.

Text data is preprocessed using Text Normalization, which converts text from written form into its verbalized form. It is used as a preprocessing step in Automatic Speech Recognition (ASR) training transcripts.

The next step is to build a text tokenizer.

Tokenizer: There are two popular encoding choices: character encoding and sub-word encoding. Sub-word encoding models are almost nearly identical to the character encoding models. The primary difference lies in the fact that a sub-word encoding model accepts a sub-word tokenized text corpus and emits sub-word tokens in its decoding step.

Data Cleaning and Filtering

Data cleaning and filtering is carried out to filter out outlying samples in the datasets. Samples that are too long, too short or empty are filtered out.

In addition, samples that are considered 'noisy', that is, samples having very high WER (word error rate) or CER (character error rate) w.r.t. a previously trained ASR model are filtered out.

Binning

For training ASR models, audio data with different lengths may be grouped into a batch. It would make it necessary to use paddings to make them all the same length. These extra paddings are a significant waste of computational resources. Splitting the training samples into buckets with different lengths and sampling from the same bucket for each batch would increase the computational efficiency and could result in training speedup of more than 2x.

Train and Test Splitting

This step is a staple of any deep learning and machine learning development pipeline and is done to ensure that the model is learning to generalize without overfitting the training data. For the test set, the Riva team additionally curates data that is not from the same source as the training datasets, such as YouTube and TED talks.

Compressing Data Files

Collecting and compressing multiple audio files into a highly compressed file using the Linux “tar” function is a desirable next step. If experiments are run on a cluster with datasets stored on a distributed file system, the user will likely want to avoid constantly reading multiple small files and would prefer to “tar” their audio files.

Training and Validation

The models in a Riva ASR pipeline include:

- An acoustic model that maps raw audio input to probabilities over text tokens at each time step. This matrix of probabilities is fed into a decoder that converts probabilities into a sequence of text tokens.
- A language model that is optionally used in the decoding phase of the acoustic model output.
- A punctuation and capitalization (P&C) model, that formats the raw transcript, augmenting with punctuation and capitalization.
- An inverse text normalization (ITN) model that produces a desired written format from a spoken format.

Acoustic model

The acoustic model is by far the most important part of an ASR service. These are the most resource-intensive models, requiring a large amount of data to train on powerful GPU servers or clusters. They also have the largest impact on the overall ASR quality. Some acoustic models supported by Riva are Quartznet, Citrinet, Jasper, and Conformer.

Cross-language transfer learning is especially helpful when training new models for low-resource languages. However, even when a substantial amount of data is available, cross-language transfer learning can further boost the performance. It is based on the idea that phoneme representation can be shared across different languages.

All Riva ASR models in production, other than the English model, were trained with cross-language transfer learning from an English base model, which was trained with the most audio hours.

For cross-language transfer learning, NVIDIA recommends the NVIDIA NeMo toolkit, which is an open-source framework for developers to build and train state-of-the-art Speech AI models.

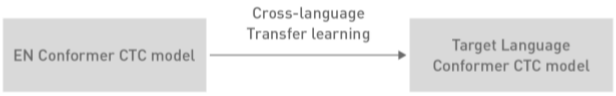

Figure 9: Cross-language Transfer Learning from English in Riva

Language model

A language model, combined with beam search in the decoding phase can further improve the quality of the ASR pipeline. NVIDIA has generally observed an additional 1-2% of WER reduction by using a simple n-gram model in the course of in-house experiments.

Training data for the language model is obtained from a training set. The training set is created by combining all the transcript text in the ASR set, normalizing, cleaning, and then tokenizing using the same tokenizer used for ASR transcript preprocessing mentioned above.

The language models supported by Riva are n-gram models, which can be trained with TAO and the KenLM toolkit.

Punctuation and Capitalization Model

The Punctuation and Capitalization (P&C) model consists of the pre-trained Bidirectional Encoder Representations from Transformers (BERT) followed by two token classification heads. One classification head is responsible for the punctuation task, the other one handles the capitalization task.

Inverse Text Normalization Model

The NeMo text inverse normalization module is leveraged for the task. NeMo ITN is based on weighted finite-state transducer (WFST) grammars. The tool uses Pynini to construct WFSTs, and the created grammars can be exported and integrated into Sparrowhawk, an open-source version of the Kestrel TTS text normalization system. for production.

Deployment

With all the models trained, the Riva service can be deployed.

Bring Your Own Models

Having obtained the final `.nemo` models that were trained using the preceding training steps, use the following steps and tools to deploy custom models to Riva:

- Refer to Riva Quick Start scripts, which provide **`nemo2riva`** conversion tool, and scripts (**`riva_init.sh`**, **`riva_start.sh`** and **`riva_start_client.sh`**) to download the **`servicemaker`**, **`riva-speech-server`** and **`riva-speech-client`** Docker images.
- Build **`.riva`** assets: using n**`emo2riva`** command in the **`servicemaker`** container.
- Build **`RMIR`** assets: use the **`riva-build tool`** in the **`servicemaker`** container.
- Deploy the model in **`.rmir`** format with **`riva-deploy`**.
- Start the server with **`riva-start.sh`**.

After the server successfully starts up, query the service to measure accuracy, latency, and throughput.

Riva Pre-trained Models on NVIDIA GPU Cloud

Alternatively, use Riva pre-trained models published on NVIDIA GPU Cloud (NGC). These models can be deployed as-is, or serve as a starting point for fine-tuning and further development.

Case Study: Speech Recognition for German Language

For the German language, there are several significant sources of public datasets that can be readily leveraged:

- Mozilla Common Voice (MCV) corpus 7.0, DE subset: 571 hours
- Multilingual LibriSpeech (MLS), DE subset: 1918 hours
- Voxpopuli, DE subset: 214 hours

In addition, NVIDIA has acquired proprietary training data amounting to a total of 3500 hours.

The training of the final model started from a NeMo DE Conformer-CTC large model, trained on MCV7.0 (567 hours), MLS (1524 hours), and VoxPopuli (214 hours). The NeMo DE conformer model itself was trained using an English Conformer model for initialization. The process is illustrated as in Figure 10.

Figure 10: German Language Acoustic Model Training

All Riva German language assets are published on NGC (including .nemo, .riva, .tlt and .rmir assets) and may be used as starting points for your development. Resources for customized development are:

- Acoustic models:
  - Citrinet ASR German:
    - Riva deployable version (.riva format)
    - Nemo version (.nemo format)
  - Conformer ASR German
    - Riva deployable version (.riva format)
    - Nemo version (.nemo format)
  - Inverse text normalization models: This model is an OpenFST finite state archive (.far) for use within the open source Sparrowhawk normalization engine and Riva.
  - Language model: A 4-gram language model trained with Kneser-Ney smoothing using KenLM is provided. This directory also contains the decoder dictionary used by the Flashlight decoder.
  - Punctuation and capitalization model: a Riva Punctuation and Capitalization model for German is provided.

Riva workflow and tools for new languages provide a systematic approach to create customized speech AI solutions, whether fine-tuning a model for an existing language or implementing a new model for a totally new dialect. Get started with the Riva new language tutorial series to learn more.

Riva Text-to-Speech

The text-to-speech (TTS) pipeline implemented for the Riva TTS service is based on a two-stage pipeline. Riva first generates a mel spectrogram using the first model, and then generates speech using the second model. This pipeline forms a text-to-speech system that enables you to synthesize natural sounding speech from raw transcripts without any additional information such as patterns or rhythms of speech.

Riva TTS currently supports two high-quality male and female voices for English, with more available in the coming releases.

Table 2: Riva TTS voices

Language	Language Code	Gender	Voice Name
English	en-US	Female	English-US-Female-1
English	en-US	Male	English-US-Male-1

Getting Started with Out-of-the-box Skills

Refer to section 1.1 on instructions of how to set up a Riva Speech AI server, including TTS skills.

To get started, enable the TTS service in the Riva `config.sh file`:

```
# Enable or Disable Riva Services
service_enabled_tts=true                          ## MAKE CHANGES HERE
```

Generating High-Performance Inference

The Riva TTS service is simple to use. Let us walk through a specific Python example.

Import the necessary libraries, including the Riva client libraries:

```
import numpy as np
import IPython.display as ipd
import grpc


import riva_api.riva_tts_pb2 as rtts
import riva_api.riva_tts_pb2_grpc as rtts_srv
import riva_api.riva_audio_pb2 as ra
```

Next, create Riva clients and connect to the Riva Speech API server. The URI described here assumes a local deployment of the Riva Speech API server on the default port. In case the server deployment is on a different host or via Helm chart on Kubernetes, the user should use the relevant URI.

```
channel = grpc.insecure_channel('localhost:50051')


riva_tts = rtts_srv.RivaSpeechSynthesisStub(channel)
```

Riva TTS supports both streaming and batch inference modes. In batch mode, audio is not returned until the full audio sequence for the requested text is generated and can achieve higher throughput. But when making a streaming request, audio chunks are returned as soon as they are generated, significantly reducing the latency (as measured by time to first audio) for large requests.

Let us take a look at an example showing batch mode TTS API usage:

```
req = rtts.SynthesizeSpeechRequest(
    text = "Is it recognize speech or wreck a nice beach?",
    language_code = "en-US",
    encoding = ra.AudioEncoding.LINEAR_PCM,     # Currently only LINEAR_PCM
is supported
    sample_rate_hz = 44100,                     # Generate 44.1KHz audio
    voice_name = "English-US-Female-1"          # The name of the voice to
generate
)


resp = riva_tts.Synthesize(req)
audio_samples = np.frombuffer(resp.audio, dtype=np.int16)
ipd.Audio(audio_samples, rate=req.sample_rate_hz)
```

Customizing Text-to-Speech Skill

Similar to Riva ASR service, Riva TTS service can be customized at inference time or training time. Inference-time customization techniques allow fine control of an existing voice whereas, training-time customization techniques allow the creation of a new voice personality.

The Riva TTS pipeline and possible optimization techniques is illustrated in Figure 11.

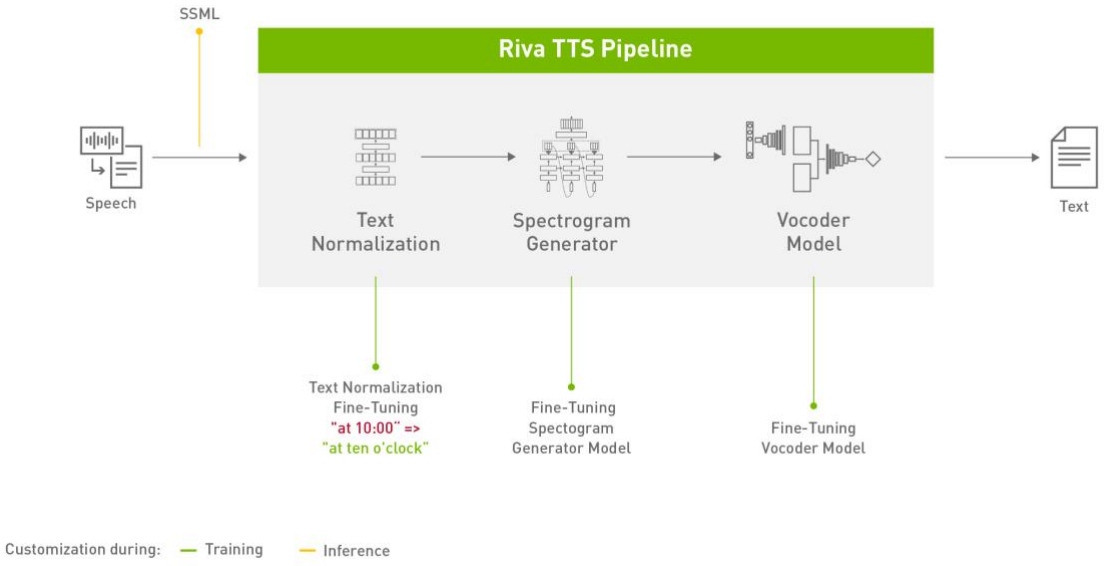

Figure 11: Riva TTS Pipeline and Possible Optimization Techniques

Inference-Time Techniques

Riva supports parts of the Speech Synthesis Markup Language (SSML) specification. SSML is a markup for directing the performance of the virtual speaker. Using SSML, you can adjust pitch, rate through the <prosody> tag, and pronunciation through the <phoneme> tag.

Prosody

The prosody tag allows control of two aspects of the synthesized speech: pitch and rate.

Pitch

Pitch is the relative high sound or low sound of a tone as perceived by the ear. Riva supports an additive relative change to the pitch by specifying a pitch attribute in the range of [-3, 3].

This value returns a pitch shift of the attribute value multiplied with the speaker's pitch standard deviation, as determined by how the FastPitch model is trained.

For example, for the pretrained checkpoint that is trained on the LJSpeech dataset, the standard deviation is 52.185. Hence, a pitch shift of 1.25 would result in a change of 1.25*52.185=~65.23Hz pitch shift up.

Values outside the range of [-3, 3] result in an error being logged, and no audio returned. Beside numerical values, pitch attribute values of "x-low", "low", "medium", "high", "x-high", and "default" are also accepted.

The pitch attribute is expressed in the following formats:

- pitch="1"
- pitch="+1.8"
- pitch="-0.65"

- pitch="high"
- pitch="default"

The following example shows speech at two different pitches.

```
<speak>
    <prosody pitch="1.0">Now I'm speaking a</prosody>
    <prosody pitch="high">bit</prosody>
    higher.
</speak>
```

Rate

Riva supports a percentage relative change to the rate with an attribute range of [25%, 250%]. Values outside this range result in an error being logged and no audio returned. Rate attributes values of “x-low”, “low”, “medium”, “high”, “x-high”, and “default” are also accepted.

The rate attribute is expressed in the following formats:

- rate="35%"
- rate="+200%"
- rate="low"
- rate="default"

```
<speak>
     <prosody rate="200%">This is a fast sentence.</prosody>
     <prosody rate="x-low">This is a extra slow sentence.</prosody>
</speak>
```

Phoneme

The phoneme tag can be used to override the pronunciation of words from the predicted pronunciation. For a given word or sequence of words, provide an explicit pronunciation by setting the “ph” attribute and the phone set used with the alphabet attribute. Currently, only x-arpabet is supported for pronunciation dictionaries based on CMUdict, with IPA support in the coming Riva release. The full list of phonemes in the CMUdict can be found here.

The following example shows two ways of pronouncing “tomato”:

```
<speak>
    Same thing!
    <phoneme alphabet="x-arpabet"
ph="{@T}{@AH0}{@M}{@EY1}{@T}{@OW2}">tomato</phoneme>,
    <phoneme alphabet="x-arpabet"
ph="{@T}{@AH0}{@M}{@AA1}{@T}{@OW2}">tomato</phoneme>.
</speak>
```

Training-Time Techniques

Similar to the Riva ASR service, fine-tuning Riva TTS models can be carried out using the Train-Adapt-Optimize (TAO) toolkit, which supports fine-tuning of the two-stage pipeline. In particular, TAO supports TTS pipelines made up of two models: FastPitch for spectrogram generation and HiFiGAN as vocoder.

FastPitch is a mel-spectrogram generator, designed to be used as the first part of a neural text-to-speech system in conjunction with a neural vocoder. FastPitch is a fully parallel text-to-speech model based on FastSpeech, conditioned on fundamental frequency contours. The model predicts pitch contours during inference. By altering these predictions, the generated speech can be more expressive, better match the semantic of the utterance, and in the end more engaging to the listener. FastPitch is based on a fully parallel Transformer architecture, with much higher real-time factor than Tacotron2 for mel-spectrogram synthesis of a typical utterance.

HifiGAN is a neural vocoder model for text-to-speech applications. It is the second part of a two-stage speech synthesis pipeline, with a mel-spectrogram generator such as FastPitch as the first stage. HifiGAN is a neural vocoder based on a generative adversarial network framework. During training, the model uses a powerful discriminator consisting of small sub-discriminators, each one focusing on specific periodic parts of a raw waveform. The generator is fast and has a small footprint, while producing high quality speech.

The TAO speech synthesis collection on NGC provides pretrained FastPitch and HiFiGAN models that were trained on LJSpeech sampled at 22kHz.

NVIDIA recommends using these FastPitch and HiFiGAN checkpoints on NGC. In order to get a fine-tuned TTS pipeline, you need to fine-tune FastPitch. For best results, you need to fine-tune HiFiGAN as well.

For a good quality model, at least 30 minutes of audio is recommended. NVIDIA recommends the NVIDIA Custom Voice Recorder tool to generate a good dataset for fine-tuning.

Text Normalization

Text Normalization is the task of converting text from written form into its verbalized form, for example "123" generates "one hundred twenty three", “$10” generates “ten dollars”, and so on. Riva leverages NeMo for the text normalization task. Under the hood, Nemo uses weighted finite-state transducer (WFST) grammars to map strings in written form to strings in spoken form. NeMo provides a comprehensive tutorial on customizing text normalization rules with WFST.

NeMo currently provides out of the box support for three languages (English, Spanish and German). With a functional NeMo installation, the German ITN grammars can be exported with the pynini_export.py tool as follows:

```
python3 pynini_export.py --output_dir . --grammars itn_grammars --input_case
cased --language de
```

This exports the `tokenizer_and_classify` and `verbalize` Fsts as OpenFst finite state archive (FAR) files, ready to be deployed with Riva.

```
[NeMo I 2022-04-12 14:43:17 tokenize_and_classify:80] Creating ClassifyFst
grammars.
Created ./de/classify/tokenize_and_classify.far
Created ./de/verbalize/verbalize.far
```

To deploy these text normalization rules with Riva, pass the FAR files to the `riva-build` command under these options:

```
riva-build speech_synthesis \
    --wfst_tokenizer_model=tokenize_and_classify.far \
    --wfst_verbalizer_model=verbalize.far \

    <other parameters>
```

Fine-tuning Spectrogram and Vocoder Models

Refer to the Jupyter notebook for a step-by-step demonstration of the process.

Briefly, the steps are:

- Install and configure TAO
- Download and preprocess the transcribed speech datasets, either from public open-source (for a public voice) or from the NVIDIA Custom Voice Recorder tool for your own recorded custom voice data
- Perform Text normalization
- Fine-tune the spectrogram generator model
- Fine-tune the vocoder model

Fine-tune Spectrogram Generator

The spectrogram generator model can be fine-tuned using TAO using just 1 line of command. Below is an example:

```
!tao spectro_gen finetune \
     -e $SPECS_DIR/spectro_gen/finetune.yaml \
     -g $NUM_GPUS \
     -k tlt_encode \
     -r $RESULTS_DIR/spectro_gen/finetune \
     -m $pretrained_fastpitch_model \
     train_dataset=$DATA_DIR/$finetune_data_name/merged_train.json \
     validation_dataset=$DATA_DIR/$finetune_data_name/manifest_val.json \
     prior_folder=$RESULTS_DIR/spectro_gen/finetune/prior_folder \
     trainer.max_epochs=200 \
     n_speakers=2 \
     pitch_fmin=$pitch_fmin \
     pitch_fmax=$pitch_fmax \
     pitch_avg=$pitch_mean \
     pitch_std=$pitch_std \
     trainer.precision=16
```

The resulting model can be exported to Riva for deployment with a single command:

```
!tao spectro_gen export \
     -e $SPECS_DIR/spectro_gen/export.yaml \
     -g 1 \
     -k $KEY \
     -m $RESULTS_DIR/spectro_gen/finetune/checkpoints/finetuned-model.tlt \
     -r $RESULTS_DIR/spectro_gen/export \
     export_format=RIVA \
     export_to=spectro_gen.riva
```

Fine-tune Vocoder Model

The vocoder model can be fine-tuned using TAO with the following command:

```
!tao vocoder finetune \
     -e $SPECS_DIR/vocoder/finetune.yaml \
     -g $NUM_GPUS \
     -k $KEY \
     -r $RESULTS_DIR/vocoder/finetune \
     -m $pretrained_hifigan_model \
     train_dataset=$DATA_DIR/$finetune_data_name/hifigan_train_ft.json \
     validation_dataset=$DATA_DIR/$finetune_data_name/hifigan_dev_ft.json \
     trainer.max_epochs=200
```

The resulting model can be exported to Riva for deployment with a single command:

```
!tao vocoder export \
     -e $SPECS_DIR/vocoder/export.yaml \
     -g 1 \
     -k $KEY \
     -m $RESULTS_DIR/vocoder/finetune/checkpoints/finetuned-model.tlt \
     -r $RESULTS_DIR/vocoder/export \
     export_format=RIVA \
     export_to=vocoder.riva
```

Deploying Your Custom TTS Model to Riva

Once the models have been fine-tuned, the next step is to deploy the custom models to Riva for high-performance serving. Refer to the demo notebook for a step-by-step instructions. Briefly, the steps are:

- Use Riva ServiceMaker to take models in .riva format and convert to .rmir
- Deploy the models on to the Riva Server
- Send inference requests from a demo client using Riva API bindings

Building and Deploying Models as Skills

The Riva ServiceMaker is the set of tools that aggregates all the necessary artifacts such as models, files, configurations, and user settings, for Riva deployment to a target environment. It has two main components:

- riva-build
- riva-deploy

Riva-build

This step helps build a Riva-ready version of the model. Its only output is an intermediate format (called a RMIR) of an end-to-end pipeline for the supported services within Riva. Two models that together comprise the two-stage TTS pipeline must be deployed:

- FastPitch (spectrogram generator)
- HiFi-GAN (vocoder).

The riva-build tool is responsible for the combination of one or more exported models (.riva files) into a single file containing an intermediate format called Riva Model Intermediate Representation (.rmir). This file contains a deployment-agnostic specification of the whole end-to-end pipeline along with all the assets required for the final deployment and inference. Refer to the NVIDIA Rival skills documentation to learn more.

To specify the speaker voice ID, use `` `--voice_name=new_speaker` `` with the riva-build command.

Riva-deploy

The Riva-deploy tool takes as input one or more Riva Model Intermediate Representation (RMIR) files and a target model repository directory. It creates an ensemble configuration specifying the pipeline for the execution and finally writes all those assets to the output model repository directory.

After the Riva-deploy step, stop the server using the `riva_stop.sh` script and restart the server using the `riva_start.sh` script.

Finally, start a Python client querying the new model, as shown in the example below.

First, import the necessary Riva helper libraries:

```
import os
import soundfile
import grpc
import riva_api.riva_tts_pb2 as rtts
import riva_api.riva_tts_pb2_grpc as rtts_srv
import riva_api.riva_audio_pb2 as ra
import IPython.display as ipd
import numpy as np
```

Next, establish a gRPC connection with the Riva server, which listens on port 50051 for a local deployment:

```
server = "localhost:50051"


channel = grpc.insecure_channel(server)
client = rtts_srv.RivaSpeechSynthesisStub(channel)
```

A TTS request object is formed, specifying various parameters, amongst which, the desired speaker voice:

```
req = rtts.SynthesizeSpeechRequest()
req.text = "Is it recognize speech or wreck a nice beach?"
req.language_code = "en-US"                      # currently required to be
"en-US"
req.encoding = ra.AudioEncoding.LINEAR_PCM     # Supports LINEAR_PCM, FLAC,
MULAW and ALAW audio encodings
req.sample_rate_hz = 22050                       # ignored, audio returned
will be 22.05KHz
req.voice_name = "new_speaker"       # subvoice to generate the audio
output.
```

Finally, send the request to the server, receive the speech data and playback on a speaker:

```
data_type = np.int16                             # For RIVA version < 1.10.0
please set this to np.float32


resp = client.Synthesize(req)
audio_samples = np.frombuffer(resp.audio, dtype=data_type)
ipd.Audio(audio_samples, rate=22050)
```

Conclusion

Speech AI has become an integral and essential part of consumers' everyday's lives. Enterprises are discovering new ways of bringing great value to their products by incorporating Speech AI services.

NVIDIA is helping enterprises adopt speech AI with an end-to-end customizable workflow, SDKs, and partners. NVIDIA's offering includes Riva, a powerful deep-learning based SDK for Speech AI, offering high accuracy, customizability, and performance. With state-of-the-art world-class pretrained models free for use, an enterprise can quickly set up automatic speech recognition and speech synthesis services and incorporate them into their applications. Riva is optimized to run on the NVIDIA Triton inference server and NVIDIA GPUs, thus offering a comprehensive inference solution that best utilizes existing infrastructure. Riva can run anywhere, from the data center, in the cloud, embedded, and to the edge.

NVIDIA Riva together with the NVIDIA TAO toolkit and NVIDIA NeMo form a comprehensive ecosystem for R&D needs in Speech AI, allowing organizations to systematically and efficiently set up new ASR and TTS pipelines to suit custom applications for specialized domains, catering to new and multiple languages with customizable voice profiles. You can learn more about Riva by visiting the getting started and documentation pages.

Testimonials

NVIDIA Riva is adopted by multiple industries and verticals, beginning with T-Mobile in their customer experience centers to Tarteel for their AI-powered Quran companion. Following are a few customers and their positive experiences with Riva

T-Mobile

"With NVIDIA Riva services, fine-tuned using T-Mobile data, we're building products to help us resolve customer issues in real time. After evaluating several automatic speech recognition solutions, T-Mobile has found Riva to deliver a quality model at extremely low latency, enabling experiences our customers love." Matthew Davis, Vice President of Product and Technology, T-Mobile

RingCentral

"Using NVIDIA Riva speech-to-text, we're able to transcribe meeting audio in real-time with high accuracy while concurrently running thousands of streams, which translates to more engaging meeting experiences for millions of RingCentral users." Prashant Kukde, Associate Vice President, RingCentral

Tarteel.AI

"By fine-tuning NVIDIA Riva automatic speech recognition for the Arabic language, we've achieved state-of-the-art WER of 4% for Arabic transcription and are able to provide live feedback to recitations, enabling thousands of users to easily learn the Quran." — Anas, CEO of Tarteel.ai