Lingvo: a Modular and Scalable Framework for Sequence-to-Sequence Modeling

`https://github.com/tensorflow/lingvo`

Jonathan Shen, Patrick Nguyen, Yonghui Wu, Zhifeng Chen, Mia X. Chen, Ye Jia, Anjuli Kannan, Tara Sainath, Yuan Cao, Chung-Cheng Chiu, Yanzhang He, Jan Chorowski, Smit Hinsu, Stella Laurenzo, James Qin, Orhan Firat, Wolfgang Macherey, Suyog Gupta, Ankur Bapna, Shuyuan Zhang, Ruoming Pang, Ron J. Weiss, Rohit Prabhavalkar, Qiao Liang, Benoit Jacob, Bowen Liang, HyoukJoong Lee, Ciprian Chelba, Sébastien Jean, Bo Li, Melvin Johnson, Rohan Anil, Rajat Tibrewal, Xiaobing Liu, Akiko Eriguchi, Navdeep Jaitly, Naveen Ari, Colin Cherry, Parisa Haghani, Otavio Good, Youlong Cheng, Raziel Alvarez, Isaac Caswell, Wei-Ning Hsu, Zongheng Yang, Kuan-Chieh Wang, Ekaterina Gonina, Katrin Tomanek, Ben Vanik, Zelin Wu, Llion Jones, Mike Schuster, Yanping Huang, Dehao Chen, Kazuki Irie, George Foster, John Richardson, Klaus Macherey, Antoine Bruguier, Heiga Zen, Colin Raffel, Shankar Kumar, Kanishka Rao, David Rybach, Matthew Murray, Vijayaditya Peddinti, Maxim Krikun, Michiel A. U. Bacchiani, Thomas B. Jablin, Rob Suderman, Ian Williams, Benjamin Lee, Deepti Bhatia, Justin Carlson, Semih Yavuz, Yu Zhang, Ian McGraw, Max Galkin, Qi Ge, Golan Pundak, Chad Whipkey, Todd Wang, Uri Alon, Dmitry Lepikhin, Ye Tian, Sara Sabour, William Chan, Shubham Toshniwal, Baohua Liao, Michael Nirschl, and Pat Rondon

February 2019

[†]Special thanks to Alexander Grushetsky and Adam Sadovsky for the initial design of the Params class.

Abstract

Lingvo is a Tensorflow framework offering a complete solution for collaborative deep learning research, with a particular focus towards sequence-to-sequence models. Lingvo models are composed of modular building blocks that are flexible and easily extensible, and experiment configurations are centralized and highly customizable. Distributed training and quantized inference are supported directly within the framework, and it contains existing implementations of a large number of utilities, helper functions, and the newest research ideas. Lingvo has been used in collaboration by dozens of researchers in more than 20 papers over the last two years. This document outlines the underlying design of Lingvo and serves as an introduction to the various pieces of the framework, while also offering examples of advanced features that showcase the capabilities of the framework.

Contents

1 Introduction **2**

2 Design **2**

2.1 Motivation 2

2.2 Components 4

3 Implementation **5**

3.1 Params 5

3.2 Layers 5

3.3 Variable Management 6

3.4 Input Processing 6

3.5 Model Registration 7

3.6 Overriding Params from the Command Line 7

3.7 Assertions 7

3.8 Code Layout 8

4 Life of a Training Run **9**

5 Advanced Usage **10**

5.1 Distributed Training 10

5.1.1 Common Runners 10

5.1.2 Asynchronous Training 11

5.1.3 Synchronous Training 11

5.2 Multi-task Models 11

5.3 Inference and Quantization 11

1 Introduction

This paper presents the open-source Lingvo framework developed by Google for sequence modeling with deep neural networks. To date, this framework has produced a number of state-of-the-art results in machine translation [2, 21, 23], speech recognition [3, 4, 5, 10, 11, 12, 13, 14, 16, 15, 20, 22], speech synthesis [6, 8, 19], and speech translation [7, 9]. It is currently being used by dozens of researchers in their day-to-day work.

We begin by motivating the design of the framework in Section 2, including the development environment it was built for as well as its guiding principles. That is followed by an exposition of its core components and the role possessed by each of them.

Then, in Section 3, we take a deeper dive into how fundamental concepts are implemented and what that means for users of the framework. This covers topics such as how trainable variables are managed and how hyperparameters are configured, as well as the basic APIs involved in composing layers into a model. While there will be some code snippets, those seeking complete examples with code should refer to the codelab [1].

Section 4 provides a consolidated walk-through of the flow of logic during a training run. It outlines the pieces involved from how the model is constructed to how its parameters are updated.

Finally, advanced usage such as distributed training, multi-task models, and inference are described in Section 5.

2 Design

2.1 Motivation

In research, it is critical to be able to quickly prototype and iterate on new ideas. But, when working in a collaborative environment, it is also critical to be able to easily reuse code and document past experiments.

Lingvo evolved out of the need to support a large group of applied researchers working on speech and natural language problems in a single shared codebase.

It follows these guiding principles:

- Individual pieces should be small and modular, implement the same consistent interface, and be easily extensible;
- Experiments should be shared, comparable, reproducible, understandable, and correct;
- Performance should efficiently scale to production-scale datasets and distributed training over hundreds of accelerators;
- Code should be shared as much as possible when transitioning from research to production.

Modular building blocks. Lingvo is designed for collaboration, focusing on code with a consistent interface and style that is easy to read and understand, and a flexible modular layering system that promotes code reuse. The same building blocks, such as LSTM or attention layers, can be used as-is across different models with assurance of good quality and performance. Because the blocks are general, an algorithmic improvement in one task (such as the use of multi-head attention in Machine Translation) can be immediately applied to another task (e.g. Speech Recognition). With many people using the same codebase, this makes it extremely easy to employ ideas others are trying in your own models. This also makes it simple to adapt existing models to new datasets.

The building blocks are each individual classes, making it straightforward to extend and override their implementation. Layers are composed in a hierarchical manner, separating low-level implementation details from high-level control flow.

Shared, comparable, reproducible, understandable, and correct experiments. A big problem in research is the difficulty in reproducing and comparing results, even between people working in the same team. To better document experiments and allow the same experiment to be re-run in the future, Lingvo adopts a system where all the hyperparameters of a model are configured in their own dedicated sub-directory separate from the model logic and are meant to be committed to a shared version control system. As the models are built from the same common layers, this allows our models to be compared with each other without worrying about effects from minute differences in implementation.

All models follow the same overall structure from input processing to loss computation, and all the layers have the same interface. In addition, all the hyperparameters are explicitly declared and their values are logged at runtime. Finally, there are copious amounts of assertions about tensor values and shapes as well as documentation and unit tests. This makes it very easy to read and understand new models when familiar with the framework, and to debug and ensure that the models are correct.

Performance. Lingvo is used to train on production-scale datasets. As a matter of necessity, its implementation has been optimized, from input processing to the individual layers. Support for synchronous and asynchronous distributed training is provided.

Deployment-readiness. Ideally, there should be little porting from research to product deployment. In Lingvo, inference-specific graphs are built from the same shared code used for training, and individual classes can be overwritten with device-specific implementations while the high level model architecture remains the same. In addition, quantization support is built directly into the framework.

However, these benefits come at the cost of more discipline and boilerplate, a common trade-off between scalability and fast prototyping.

2.2 Components

The following are the core components of the Lingvo framework.

Models: A *`Model`* is an abstract collection of one or more *`Tasks`*. For single-task models the *`Model`* is just a transparent wrapper around the *`Task`* and the two can be considered the same. For multi-task models, the *`Model`* controls how variables are shared between *`Tasks`* and how *`Tasks`* are sampled for training.

Tasks: A *`Task`* is a specification of a complete optimization problem, such as image classification or speech recognition. It contains an input generator, *`Layers`* representing a neural network, a loss value, and an optimizer, and is in charge of updating the model parameters on each training step.

Layers: A *`Layer`* represents an arbitrary function possibly with trainable parameters. A *`Layer`* can contain other *`Layers`* as children. `SoftMax`, `LSTM`, `Attention`, and even a *`Task`* are all examples of *`Layers`*.

Input Generators: Lingvo input generators are specialized for sequences, allowing batching input of different lengths in multiple buckets and automatically padding them to the same length. Large datasets that span multiple input files are also supported. The flexibility of the `generic_input` function enables simple and efficient implementations of custom input processors.

Params: The *`Params`* object contains hyperparameters for the model. They can be viewed as local versions of `tf.flags`. *`Layers`*, *`Tasks`*, and *`Models`* are all constructed in accordance to the specifications in their *`Params`*.

`Params` are hierarchical, meaning that the *`Params`* for an object can contain *`Params`* configuring child objects.

Experiment Configurations: Each experiment is defined in its own class and fully defines all aspects of the experiment from hyperparameters like learning rate and optimizer parameters to options that affect the model graph structure to input datasets and other miscellaneous options.

These standalone configuration classes make it easy to keep track of the params used for each experiment and to reproduce past experiments. It also allows configurations to inherit from other configurations.

All experiment params are registered in a central registry, and can be referenced by its name, e.g. `image.mnist.LeNet5`.

Job Runners: Lingvo's training setup is broken into separate jobs. For example, the *`Controller`* job is in charge of writing checkpoints while the *`Evaler`* job evaluates the model on the latest checkpoint. For a full description of the different job runners see Section 5.1.

NestedMap: A *NestedMap* is a generic dictionary structure for arbitrary structured data similar to `tf.contrib.framework.nest`. It is used throughout Lingvo to pass data around. Most python objects in the code are instances of either *Tensor*, a subclass of *BaseLayer*, or *NestedMap*.

Custom Ops: Lingvo supports custom op kernels written in C++ for high-performance code. For example, custom ops are used for the input pipeline, beam search, and tokenization.

3 Implementation

This section provides a more detailed look into the core Lingvo APIs. Section 3.1 introduces the *Params* class which is used to configure everything. Section 3.2 covers how *Layers* are constructed, how they work, and how they can be composed. Section 3.3 describes how variables are created and managed by each *Layer*. Section 3.4 goes over input reading and processing, and Sections 3.5, 3.6, and 3.7 briefly go over model registration, overriding params, and runtime assertions. Finally, Section 3.8 gives a simple overview of the layout of the source code.

3.1 Params

The *Params* class is a dictionary with explicitly defined keys used for configuration. Keys should be defined when the object is created, and trying to access or modify a nonexistent key will raise an exception. In practice, every *Layer* has a `Params` classmethod, which creates a new params object and defines the keys used to configure the layer with a reasonable default value. Then, in a separate experiment configuration class, these default values are overridden with experiment-specific values.

3.2 Layers

In order to construct a *Layer*, an instance of these layer's *Params* is required. The params includes details such as:

- `cls`: the layer's class,
- `name`: the layer's name, and
- `params_init`: how the variables created by this layer should be initialized.

Because the class is contained in the params, the following ways of constructing the layer are equivalent:

```
p = SomeLayerClass.Params()
layer = SomeLayerClass(p)  # Call the constructor.
layer = p.cls(p)           # Same, but call through the params.
```

All layers have a `FProp()` function, which is called during the forward step of a computation. Child layers can be created in the constructor using `self.CreateChild('child_name', child_params)`, and they can be referenced by `self.child_name`.

3.3 Variable Management

Each *`Layer`* creates and manages its own variables.

Variables are created in the layer's `__init__()` method through a call to `self.CreateVariable()`, which registers the variable in `self.vars` and the value of the variable (potentially after a transform like adding variational noise) in `self.theta`. In `FProp()`, because it may be executed on different devices in distributed training, for performance reasons it is best to access the variables through the `theta` parameter passed in to the function rather than `self.vars` or `self.theta`.

Variable placement is determined by the `cluster.GetPlacer()` function. The default policy is to place each variable on the parameter server that has the least bytes allocated. For model parallelism, an explicit policy based on e.g. variable scope can be adopted.

There are many benefits to explicitly managing variables instead of using `tf.get_variable`:

- It supports research ideas such as weight noise.
- The `variable_scope` construct can be error prone and less readable, for example accidental reuse of a variable.
- For sync replica training, sharing the weights between multiple workers on the same machine is otherwise awkward.

3.4 Input Processing

Lingvo supports inputs in either plain text or `TFRecord` format. Sequence inputs can be bucketed by length through the `bucket_upper_bound` and `bucket_batch_limit` params.

A tokenizer can be specified for text inputs. Available tokenizers include `VocabFileTokenizer` which uses a look-up table provided as a file, `BpeTokenizer` for byte pair encoding [18], and `WpmTokenizer` for word-piece models [17].

The input file pattern should be specified as "`type:glob_pattern`" through the `file_pattern` param. The input processor should implement the `_DataSourceFromFilePattern()` method, which returns an op that when executed reads from the file and returns some tensors. Often this op is implemented as a custom `C++` op using the `RecordProcessor` interface. The tensors returned by this op can be retrieved by calling `_BuildDataSource()`, and can be used to fill in an input batch *`NestedMap`* to be returned by the `InputBatch()` method. Finally, batch-level preprocessing can also be implemented in `PreprocessInputBatch()`.

In addition to using a custom `RecordProcessor` op, an input processor can also be defined directly in `Python` through the `generic_input` op.

3.5 Model Registration

Configuration classes lie inside `lingvo/tasks/<task>/params/<param>.py` and are annotated with `@model_registry.RegisterSingleTaskModel` for the typical case of a single-task model. This annotation adds the class to the model registry with a key of `<task>.<param>.<classname>` (e.g. `image.mnist.LeNet5`).

The class should be a subclass of `SingleTaskModelParams` and implement the `Task()` method, which returns a ***Params*** instance configuring a ***Task***. The registration code will automatically wrap the ***Task*** into a `SingleTaskModel`.

The class should also implement the `Train()`, `Test()`, and maybe `Dev()` methods. These methods return a ***Params*** instance configuring an input generator, and represent different datasets.

An example is shown in Figure 1.

```
@model_registry.RegisterSingleTaskModel
class MyTaskParams(base_model_params.SingleTaskModelParams):
  @classmethod
  def Train(cls):
    ...  # Input params.

  @classmethod
  def Task(cls):
    p = my_model.MyNetwork.Params()
    p.name = 'my_task'
    ...
    return p
```

Figure 1: Registering a single-task model.

3.6 Overriding Params from the Command Line

It is possible to override the values of any hyperparameter for a specific run using the `--model_params_override` or `--model_params_file_override` flags. This makes it simple to start similar jobs for hyperparameter tuning.

3.7 Assertions

`py_utils.py` contains functions for run-time assertions about values and shapes as well as `CheckNumerics()` for detecting NaNs. Assertions can be disabled with the command-line `--enable_asserts=false`. Similarly, `CheckNumerics` can be disabled with `--enable_check_numerics=false`.

3.8 Code Layout

- lingvo
  - trainer.py
    Entry point.
  - model_imports.py
    Imports and registers all model params in the global registry.
  - core
    - base_input_generator.py
    - base_layer.py
    - base_model.py
    - cluster.py
      Contains the policy for op placement.
    - hyperparams.py
    - attention.py, layers.py, rnn_cell.py, rnn_layers.py
      Contains implementations for many common layers.
    - optimizer.py
    - py_utils.py
      Most utility functions are here.
    - recurrent.py
      The functional RNN.
    - summary_utils.py
      Contains utilities for dealing with summaries.
    - ops
      Folder for custom C++ ops.
      - record_*.*
        The input processing pipeline.
      - py_x_ops.py
        Python bindings for the C++ ops.
      - x_ops.cc
        C++ op definitions.
  - tasks
    - <task>
      Folder for an individual task/domain/project.
      - params
        Folder for model params.
  - tools

4 Life of a Training Run

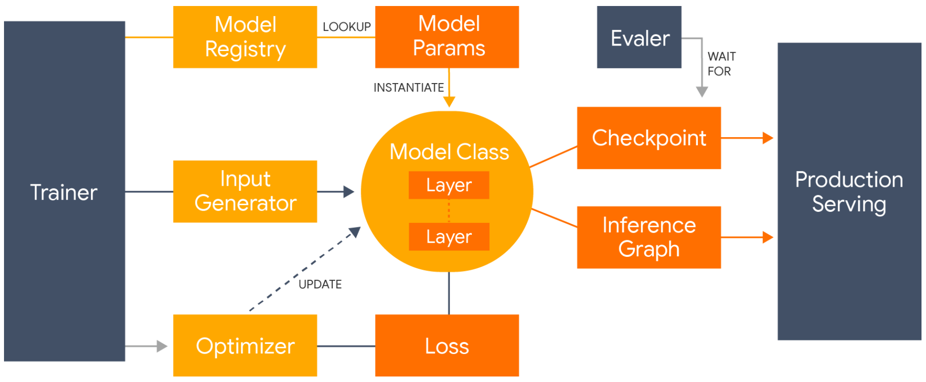

Figure 2: An overview of the Lingvo framework, outlining how models are instantiated, trained, and exported for evaluation and serving.

This section gives an overview of what happens behind the scenes from the start of a training run to the end of the first step for a single-task model.

Training is started by launching the `trainer` with the name of a model and the path to a log directory. The model name is resolved using the model registry to obtain the *`Params`* for the model, and the various job runners for training are created.

The *`Params`* at this point is just the params for the top level *`Model`* with the overrides in the experiment configuration corresponding to the model name specified. No *`Layers`* have been instantiated yet.

Each runner then independently instantiates the model and builds the tensorflow graphs and ops that they will execute based on their job. For example, the *`Trainer`* will build a `train_op` spanning both the `FProp()` and `BProp()` graphs and involves updating model parameters, while the *`Evaler`* and *`Decoder`* will build a `eval_metrics` op involving only the `FProp()` graph with `p.is_eval` set. There can be multiple *`Evalers`* and *`Decoders`*, one for each evaluation dataset.

Instantiating the model calls its `__init__()` method, which constructs the *`Params`* for child layers and instantiates them recursively through calls to `self.CreateChild()`. These child layer params could be exposed as part of the top level model params, perhaps as a "params template", allowing them to be configured in the params files, or they could be constructed completely from scratch in the `__init__()` method based on the information available at that time. The `__init__()` method is also in charge of creating the variables managed by the *`Layer`* through `self.CreateVariable()`.

Once the graphs are built, the *`Trainer`* runner will wait for the *`Controller`* runner to initialize the variables or restore them from a checkpoint, while the evaluation runners will wait for a new checkpoint to be written to the log directory.

After the variables are initialized, the *`Trainer`* will run training, i.e. calling `session.run` with the model's `train_op`, in a loop, and the *`Controller`* will produce training summaries for each step. When enough steps have passed, the *`Controller`* writes a new checkpoint to the log directory. The *`Evaler`* and *`Decoder`* detect this new checkpoint and evaluates the model at that checkpoint to generate summaries. This process then continues until the user terminates all jobs or `p.train.max_steps` is reached.

For more details about the various runners such as the difference between Evalers and Decoders, as well as information about which devices individual ops in the graph will be placed on during distributed training, see Section 5.1.

5 Advanced Usage

This section provides some examples of advanced features. This is by no means an exhaustive list of all the existing features, and many new features are continually being added.

Section 5.1 describes the distributed training setup. Section 5.2 details how multi-task models are configured and registered, and Section 5.3 gives a brief look into inference and productionization support.

5.1 Distributed Training

Both synchronous as well as asynchronous distributed training are supported. In asynchronous mode, each individual worker job executes its own training loop and is completely independent from the other workers. In synchronous mode, there is a single training loop driven by a trainer client that distributes work onto the various worker jobs.

Here we summarize the different types of job runners under each configuration.

A shared directory on a distributed file system where checkpoints can be written and loaded is assumed to exist.

5.1.1 Common Runners

Controller: This job handles initializing variables and saving/loading checkpoints as well as writing training summaries.

Evaler: This job loads the latest checkpoint and runs and exports evaluation summaries. Multiple evalers can be started for different datasets.

Decoder: This job loads the latest checkpoint and runs and exports decoding summaries. Multiple decoders can be started for different datasets. Decoders are different from evalers in that the ground-truth is used during evaluation but not during decoding. A concrete example is that Evalers can use teacher-forcing while Decoders may need to rely on beam search.

5.1.2 Asynchronous Training

Trainer: This is the worker job which runs the training op and sends variable updates.

Parameter Server: Variable values are stored here. Trainer jobs send updates and receive global values periodically.

Data Processor: This is an optional job for loading data before dispatching them to trainers, to offload the cost associated with loading and preprocessing data from the trainer to a separate machine.

5.1.3 Synchronous Training

Worker: The worker job in sync training runs the training op like the trainer job in async training but they do not perform variable updates.

Trainer client: The trainer client drives the training loop and aggregates their results before updating the variables. There are no parameter servers in sync training. Instead, the worker jobs act as parameter servers, and the trainer client sends the relevant variable updates to each worker.

5.2 Multi-task Models

A multi-task model is composed of individual *`Tasks`* sharing variables. Existing options for variable sharing range from sharing just the encoder (`multitask_model.SharedEncoderModel`) to fine-grained control with `multitask_model.RegExSharedVariableModel`.

Multi-task model params should be a subclass of `MultiTaskModelParams` and implement the `Model()` method, which returns a *`Params`* instance configuring a `MultiTaskModel`. The `task_params` and `task_probs` attributes define respectively the params and relative weight of each *`Task`*.

An example of registering a multi-task model is shown in Figure 3.

Knowledge distillation is also supported via `base_model.DistillationTask`. For knowledge distillation, the teacher parameters must be loaded from a checkpoint file by specifying `params.train.init_from_checkpoint_rules` in the `Task()` definition.

5.3 Inference and Quantization

Once models have been trained, they must be deployed on a server or on an embedded device. Typically, during inference, models will be executed on a device with fixed-point arithmetic. To achieve the best quality, the dynamic range must be kept in check during training. We offer quantized layers that wrap the training and inference computation functions for convenience.

```
@model_registry.RegisterMultiTaskModel
class MyMultiTaskParams(base_model_params.MultiTaskModelParams):
  @classmethod
  def Train(cls):
    p = super(MyMultiTaskParams, cls).Train()
    task1_input_params = ...
    p.Define('task1', task1_input_params, '')
    # Or, refer to existing single task model params.
    p.Define('task2', MyTaskParams.Train(), '')
    return p

  @classmethod
  def Model(cls):
    p1 = my_model.MyNetwork.Params()
    p1.name = 'task1'
    ...
    # Or, refer to existing single task model.
    p2 = MyTaskParams.Task()
    p2.name = 'task2'

    p = base_model.MultiTaskModel.Params()
    p.name = 'my_multitask_model'
    p.task_params = hyperparams.Params()
    p.task_params.Define('task1', p1, '')
    p.task_params.Define('task2', p2, '')
    p.task_probs = hyperparams.Params()
    p.task_probs.Define('task1', 0.5, '')
    p.task_probs.Define('task2', 0.5, '')
    return p
```

Figure 3: Registering a multi-task model.

Inference has different computational characteristics than training. For latency reasons, the batch size is smaller, sometimes even equal to just 1. For sequence models, often a beam search is performed. It may even be preferable to drive inference one timestep at a time. Several constraints dominate the choice of how to run the computation: 1) available operations, 2) desired latency, 3) parallelizability, and 4) memory and power consumption. To enable the greatest amount of flexibility given these constraints, we leave it to the designer of the model to express inference in the optimal way by explicitly exporting inference graphs rather than leaving it to a graph converter. A basic inference graph can be written in a few lines of code, reusing the same functions used for building the training graph, while for more complicated inference graphs it is possible to even completely swap out the implementation of a low level layer.

References

[1] Introduction to Lingvo. `https://colab.research.google.com/github/tensorflow/lingvo/blob/master/codelabs/introduction.ipynb`.

[2] M. X. Chen, O. Firat, A.Bapna, M. Johnson, W. Macherey, G. Fosterand L. Jones, M. Schuster, N. Shazeer, N. Parmar, A. Vaswani, J. Uszkoreit, L. Kaiser, Z. Chen, Y. Wu, and M. Hughes. The Best of Both Worlds: Combining Recent Advances in Neural Machine Translation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, pages 76–86. Association for Computational Linguistics, 2018.

[3] C. C. Chiu and C. Raffel. Monotonic Chunkwise Attention. *Proc. International Conference on Learning Representations (ICLR)*, 2018.

[4] C. C. Chiu, T. N. Sainath, Y. Wu, R. Prabhavalkar, P. Nguyen, Z. Chen, A. Kannan, R. J. Weiss, K. Rao, E. Gonina, N. Jaitly, B. Li, J. Chorowski, and M. Bacchiani. State-of-the-art Speech Recognition With Sequence-to-Sequence Models. *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2018.

[5] C. C. Chiu, A. Tripathi, K. Chou, C. Co, N. Jaitly, D. Jaunzeikare, A. Kannan, P. Nguyen, H. Sak, A. Sankar, J. Tansuwan, N. Wan, Y. Wu, and X. Zhang. Speech recognition for medical conversations. *Proc. Interspeech*, 2018.

[6] W. N. Hsu, Y. Zhang, R. J. Weiss, H. Zen, Y. Wu, Y. Wang, Y. Cao, Y. Jia, Z. Chen, J. Shen, et al. Hierarchical generative modeling for controllable speech synthesis. *arXiv preprint arXiv:1810.07217*, 2018.

[7] Ron J. Weiss, Jan Chorowski, Navdeep Jaitly, Yonghui Wu, and Zhifeng Chen. Sequence-to-sequence models can directly translate foreign speech. *Proc. Interspeech*, pages 2625–2629, 08 2017.

[8] Y. Jia, Y. Zhang, R. J. Weiss, Q. Wang, J. Shen, F. Ren, Z. Chen, P. Nguyen, R. Pang, I. Lopez-Moreno, and Y. Wu. Transfer Learning from Speaker Verification to Multispeaker Text-To-Speech Synthesis. *Advances in Neural Information Processing Systems*, 2018.

[9] Ye Jia, Melvin Johnson, Wolfgang Macherey, Ron J Weiss, Yuan Cao, Chung-Cheng Chiu, Naveen Ari, Stella Laurenzo, and Yonghui Wu. Leveraging weakly supervised data to improve end-to-end speech-to-text translation. *arXiv preprint arXiv:1811.02050*, 2018.

[10] A. Kannan, Y. Wu, P. Nguyen, T. N. Sainath, Z. Chen, and R. Prabhavalkar. An analysis of incorporating an external language model into a sequence-to-sequence model. In *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2018.

[11] D. Lawson, C. C. Chiu, G. Tucker, C. Raffel, K. Swersky, and N. Jaitly. Learning hard alignments with variational inference. *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2018.

[12] B. Li, T. N. Sainath, K. Sim, M. Bacchiani, E. Weinstein, P. Nguyen, Z. Chen, Y. Wu, and K. Rao. Multi-Dialect Speech Recognition With a Single Sequence-to-Sequence Model. *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2018.

[13] R. Pang, T. N. Sainath, R. Prabhavalkar, S. Gupta, Y. Wu, S. Zhang, and C. C. Chiu. Compression of End-to-End Models. In *Proc. Interspeech*, 2018.

[14] R. Prabhavalkar, T. N. Sainath, Y. Wu, P. Nguyen, Z. Chen, C. C. Chiu, and A. Kannan. Minimum Word Error Rate Training for Attention-based Sequence-to-sequence Models. In *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2018.

[15] T. N. Sainath, C. C. Chiu, R. Prabhavalkar, A. Kannan, Y. Wu, P. Nguyen, and Z. Chen Z. Improving the Performance of Online Neural Transducer Models. *Proc. Interspeech*, 2018.

[16] T. N. Sainath, P. Prabhavalkar, S. Kumar, S. Lee, A. Kannan, D. Rybach, V. Schogol, P. Nguyen, B. Li, Y. Wu, Z. Chen, and C. C. Chiu. No Need for a Lexicon? Evaluating the Value of the Pronunciation Lexica in End-to-End Models. *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2018.

[17] M. Schuster and K. Nakajima. Japanese and Korean Voice Search. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 5149–5152. IEEE, 2012.

[18] R. Sennrich, B. Haddow, and A. Birch. Neural Machine Translation of Rare Words with Subword Units, 2015.

[19] J. Shen, R. Pang, R. J. Weiss, M. Schuster, N. Jaitly, Z. Yang, Z. Chen, Y. Zhang, Y. Wang, R. J. Skerry-Ryan, R. A. Saurous, Y. Agiomyrgiannakis, and Y. Wu. Natural TTS Synthesis By Conditioning WaveNet on Mel Spectrogram Predictions. *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2018.

[20] S. Toshniwal, T. N. Sainath, R. J. Weiss, B. Li, P. Moreno, E. Weinstein, and K. Rao. End-to-End Multilingual Speech Recognition using Encoder-Decoder Models. *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2018.

[21] R. J. Weiss, J. Chorowski J, N. Jaitly, Y. Wu, and Z. Chen. Sequence-to-Sequence Models Can Directly Translate Foreign Speech. *Proc. Interspeech*, 2017.

[22] I. Williams, A. Kannan, P. Aleksic, D. Rybach, and T. N. Sainath TN. Contextual Speech Recognition in End-to-End Neural Network Systems using Beam Search. *Proc. Interspeech*, 2018.

[23] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. Google's Neural Machine Translation system: Bridging the gap between human and machine translation. *arXiv preprint*, 1609.08144, 2016.