INTERSPEECH 2016
September 8–12, 2016, San Francisco, USA

Sage: The New BBN Speech Processing Platform

Roger Hsiao, Ralf Meermeier, Tim Ng, Zhongqiang Huang, Maxwell Jordan, Enoch Kan, Tanel Alumäe, Jan Silovsky, William Hartmann, Francis Keith, Omer Lang, Manhung Siu and Owen Kimball

Raytheon BBN Technologies
10 Moulton Street, Cambridge, MA 02138, USA
{whsiao,rmeermei,msiu}@bbn.com

Abstract

To capitalize on the rapid development of Speech-to-Text (STT) technologies and the proliferation of open source machine learning toolkits, BBN has developed Sage, a new speech processing platform that integrates technologies from multiple sources, each of which has particular strengths. In this paper, we describe the design of Sage, which allows the easy interchange of STT components from different sources. We also describe our approach for fast prototyping with new machine learning toolkits, and a framework for sharing STT components across different applications. Finally, we report Sage's state-of-the-art performance on different STT tasks.
Index Terms: speech recognition toolkit

1. Introduction

BBN's Byblos STT system [1], which has always maintained state-of-the-art performance, has been our main research and application engine for more than 30 years. In addition to its use in research, multiple customized versions have also been successfully developed for commercial STT applications such as broadcast monitoring, keyword search, real-time applications such as speech-to-speech translations and even non-speech applications such as optical character recognition (OCR). Many of these customizations address task specific requirements, such as low-latency or different workflow.

The field of STT has undergone tremendous changes over the past few years. The introduction of deep learning has generated substantial and rapid performance advancements rarely seen before. Meanwhile, the increasing popularity of the open-source movement leads to the proliferation of state-of-the-art machine learning and neural network technologies that are available to researchers as free-to-download toolkits, such as Kaldi [2], Tensorflow [3], CNTK [4], Caffe [5] and others. Many of these toolkits come with frequent algorithmic, software and recipe updates contributed by the community. To stay abreast of the continuous innovation, it is advantageous to have the ability to quickly test out these updates on any problem of interest and incorporate the useful ones into our own system. The fast pace of change and proliferation of toolkits and recipe make fast prototyping and easy integration of technology paramount. However, compatibility is a major challenge. Most toolkits are not cross-compatible with each other, and some are not even "self-compatible", in that different versions or branches within a toolkit are incompatible. Furthermore, recipes and tutorials contributed by the community are often not written in ways that are easy to generalize and apply to new applications.

The fast moving technology that is constantly upgrading makes supporting multiple highly customized STT engines difficult. This motivates the design of a single framework that allows flexible application specific customization while sharing the same underlying, evolving STT components.

Sage is BBN's newly developed STT platform for model training and recognition that integrates technologies from multiple sources, each of which has its particular strength. In Sage, we combine proprietary sources, such as BBN's Byblos, with open source software (OSS), such as Kaldi, CNTK and Tensorflow. For example, Sage's deep neural networks (DNNs) [6] can be trained using Byblos, Kaldi nnet1 or nnet2, convolutional neural networks (CNNs) [7] using Kaldi or Caffe, and long short-term memory networks (LSTMs) [8] using Kaldi as well as CNTK. The integration of these technologies is achieved by creating wrapper modules around major functional blocks that can be easily connected or interchanged. In addition, Sage software has been designed to make it easy for a group of researchers to use the system, to transfer experiments from one person to another, to keep track of partial runs, etc.

Sage includes a cross-toolkit finite state transducer (FST) based recognizer that supports models built using the various component technologies. To quickly prototype with models built using newly released machine learning toolkits, Sage also includes a Python interface that can directly interact with toolkits that supply Python API, such as Tensorflow. To make use of this modularity all the way to the deployed applications our customers receive (both internal and external), we designed the "Godec" framework, which allows us to combine the components in a flexible graph, all in one single executable. This executable (or shared library, for embedding into existing projects) is intended to serve the needs of experimenters, developers and customers alike, as it can accommodate batch processing just as well as low-latency real-time applications. For example, the configuration for batch mode transcription applications without any latency requirement may include steps such as speaker independent (SI) decoding, adaptation, and then speaker adaptive (SA) decoding. On the other hand, the configuration for processing broadcast media may include components such as speech activity detection, music separation, speaker clustering, etc. as well as the recognition components. Such a framework serves both the needs of application specific customization as well as the sharing of the same underlying software.

The rest of the paper is organized as follows. In the next section, we describe the overall design of Sage. Then, we de-

This document does not contain technology or technical data controlled under either the U.S. International Traffic in Arms Regulations or the U.S. Export Administration Regulations.

http://dx.doi.org/10.21437/Interspeech.2016-1031

scribe the "Godec" framework in section 3. We report experimental results on different recognition tasks in section 4. The paper is then concluded in section 5.

2. Design

The design of Sage aims to provide an interface to incorporate technologies from different projects. Sage has to be flexible enough to allow components to be easily interchanged, in order to shorten the time for prototyping. Also, technology transfer is critical and Sage should allow researchers to transfer the latest technologies to applications easily.

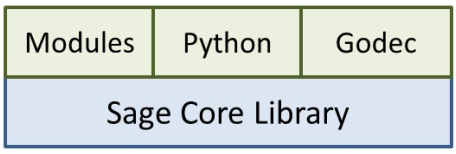

Figure 1: An overview of Sage's design

Figure 1 is an overview of Sage's design. At a high level, the Sage Core Library is an extension of BBN proprietary software and open source software including Kaldi. The design is carefully constructed so that updates from the open source projects could be easily transferred to Sage. Using the library, Sage provides the following services to meet the needs of both research and production,

- Modular recipes
- Python interface
- Godec - an all-in-one streaming framework

Both the modular design and Python interface are our attempts to enable fast prototyping, and Godec is a highly configurable and efficient message passing framework for fast technology transfer. More details of Godec are available in section 3.

2.1. Modular Recipes

Open source projects often come with recipes to build systems on standard data sets. Those recipes are often in a form of separate shell scripts that are not modularized, making it difficult to maintain the recipes or create new recipes from the existing ones. We redesign the recipes into a modular form, in which each module represents a set of interdependent procedures that can be reused. Figure 2 is an example of Sage's modular recipe that is created based on the Kaldi's standard nnet1 and nnet2 recipes.

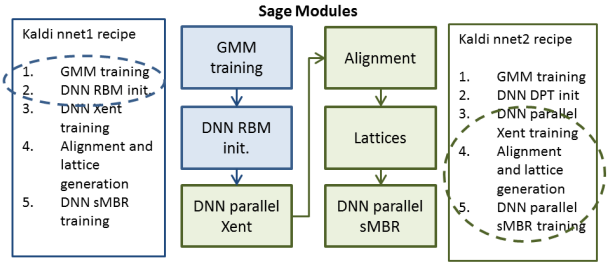

Figure 2: An example of Sage's modular recipe that is created based on Kaldi nnet1 (step 1-2) and nnet2 (step 3-5) recipes

This modular design has three advantages. First, it allows regression tests for individual modules and makes it easier for code maintenance. Second, it enables more efficient code reuse and encourages researchers to exploit different recipes to create a better recipe. Third, this framework can bring in different open source technologies outside of the Sage project. These advantages are important for rapid and effective research and software development in a collaborative environment.

2.2. Python Interface

As mentioned earlier, the rapid development of speech technologies has resulted in the proliferation of machine learning and neural network toolkits. Many of these tools are available in Python as open source packages. In order to enhance Sage with the powerful Python ecosystem, the Sage APIs are designed to be easily used to adopt new technologies. For example, through the Python Interface, Sage can integrate models trained by Tensorflow. As new technologies become available in Tensorflow, Sage would gain access to those technologies immediately without the need to reimplement them into our platform. This would greatly reduce the time and cost of exploring new techniques on the Sage platform. As a result, this design encourages and enables faster research and prototyping that would ultimately help advance the state-of-the-art in the field. Figure 3 shows a snippet of Python code where Sage decodes with a Tensorflow model.

```
import sys
from pySageDecoder import PySageDecoder, PySageDecodable
import tensorflow_wrapper as tf_wp


if __name__ == '__main__':
    if len(sys.argv) < 2:
        sys.exit("expected arguments: bbn-decoder-args tf_model_file")

    tf_model_file = sys.argv.pop()
    tf_model_op = tf_wp.load_model(tf_model_file)
    decoder = PySageDecoder()
    decoder.setup(sys.argv)

    while not decoder.done():
        features = decoder.get_features()
        log_posterior = tf_wp.get_log_posterior(features, tf_model_op)
        decodable = decoder.get_matrix_decodable(log_posterior)
        decoder.decode(decodable)
        decoder.next()

    decoder.finalize()
```

Figure 3: A snippet of Python code where Sage decodes with a Tensorflow model

3. Godec

While Sage's modularity and Python bindings are mostly geared towards enabling fast and flexible research, we also envisioned having an overarching framework that would allow for rapid transfer of advances in research, be flexible in its configuration, and yet carry little to no processing overhead. We call this framework "Godec" (**G**odec is **o**ur **dec**oder), for which we identified the following specific requirements:

- **Single executable**: Research experiments often create a multitude of small scripts and executables that are hard to transfer into a product. A single executable (or shared library to facilitate JNI and Python bindings) that is highly configurable through one central but modular configuration file speeds up transfer time.
- **Allow for submodules**: As an extension to the above, the framework should allow for clustering of components into submodules that can be reused.

- **Parallel processing**: The framework needs to be highly multi-threaded, with each major component running in its own thread, to make use of a CPU's multiple cores.
- **In-memory**: Unless explicitly desired, all intermediate data should be transferred in memory, reducing unnecessary disk I/O.
- **Both batch and real-time**: The framework should allow for both streaming and batch input.
- **Low latency**: In streaming input mode, the framework should be able to produce low-latency, real-time output.
- **Stream-precise**: To allow for repeatable and deterministic offline experiments, the framework needs to be stream-precise, i.e. it needs to recreate the same conditions as during training, even when the input is streaming and he exact internal processing order might differ between otherwise identical runs.

3.1. Framework design

From the above requirements, we designed a combination of a streaming and messaging network. In this network, components connect to each other via channels, with each component consuming and emitting distinct messages that account for sections of the stream. It is worth noting that a specific message can be consumed by as many components as desired; for example, multiple ASR decoders may operate on the same feature message.

This functionality also facilitates easy debugging, since one can just add an I/O component that "tees off" a certain stream and writes it to disk for detailed inspection. Similarly, it allows for convenient isolation of components during development, since one can simulate the rest of the network through an I/O component that feeds messages into the component to be developed. In such a massively multi-threaded framework, this kind of isolation is crucial for effective development and debugging. Figure 4 uses feature extraction as an example and shows how the messages are passed from one component to another.

3.2. Stream synchronization

To achieve the above-mentioned "stream-precise" capability, for any given component, the incoming stream messages have to be perfectly synchronized. Unlike in a user interface (UI) framework where the exact time of processing a message (in relation to other ones) is not important as long as it happens fast, here we have to perfectly align the streams, otherwise experiments would not be deterministic.

This is achieved by each message carrying a "stream time" counter which says how far this message accounts for in the overall stream. Note that the stream time is completely arbitrary and not globally enforced; usually the first feeding component creates it, and all downstream components work with that definition.

One particular aspect of note is that **all** messages have to be causal, i.e. looking backwards in time. While this is an obvious requirement for feature and audio streams, it however disallows "utterance start/end" messages, since those messages would say "from here on X is true until further notice" instead of "up to this point X is true". The reason for this is that one cannot be both low-latency and stream-precise when having non-causal messages. To illustrate, imagine a feature extraction component that processes an audio stream, but also gets utterance start/end messages from a Speech Activity Detector (SAD). If it got an utterance-start message and started processing all incoming audio from there on, the scenario could arise where the SAD is slow and issues an utterance-end message for an audio stream position the feature extractor has already gone past. This can cause potentially erroneous feature vectors since utterance ends often require special feature processing. To solve the problem, the feature extraction either would have to wait indefinitely until it sees the utterance-end message (thus dropping the low-latency capability since an utterance can be arbitrarily long), or create erroneous features (which would mean not being stream-precise). The key realization is that one cannot mix causal and noncausal messages when having both these requirements. Applying the requirement of causality to utterance start/end, we instead have "conversation state" messages which essentially say "up to here we are still within an utterance". Similarly, something like an adaptation matrix message says "this adaptation matrix is valid up to here in the stream". It is a subtle design decision, but crucial for the proper function of the network.

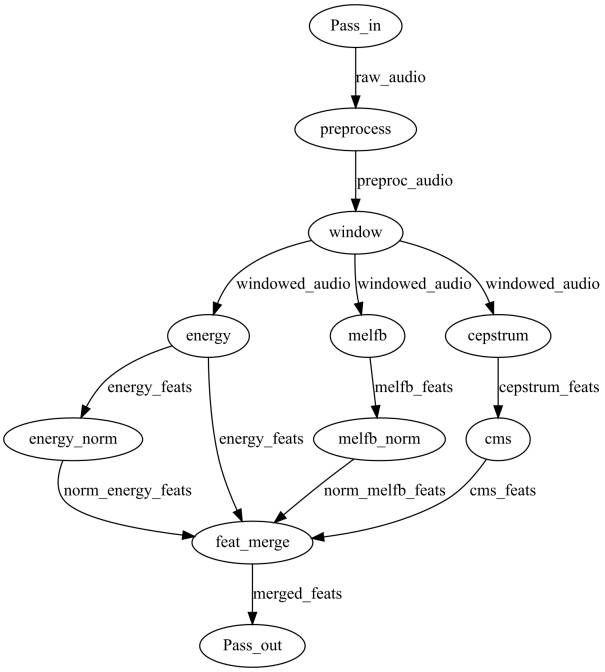

Figure 4: An example sub-network for feature extraction

3.3. Stream slicing

All components inherit from a common base class that takes care of the synchronization of the incoming messages. In particular, unlike truly asynchronous messaging frameworks, a component will not process small updates to just one of its input streams, but rather wait until it has a **contiguous** chunk in time that spans all incoming streams. There are several benefits:

- Developers of a new component do not have to worry about the asynchronous nature of the framework. The component code only sees a solid chunk in time covering all streams.
- The base class can ensure to slice the chunks so that the chunk is "atomic", e.g. so that it does not contain two separate utterances. This again reduces a major source of errors for the developer.

Figure 5 illustrates such a chunk slicing for one comonent.

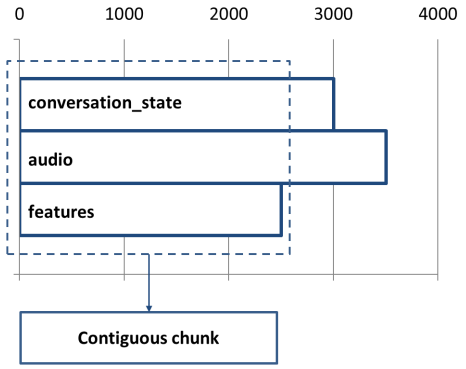

Figure 5: Slicing out a contiguous chunk from streams queued up in front of a component. Top numbers are stream time.

4. Performance and Benchmarking

We evaluated the performance of Sage using a variety of languages and training conditions. In addition to word error rate (WER), we also consider the real-time factor and memory footprint, which are important constraints for real world applications. Table 1 summaries the data we used in this evaluation. For English, we have two training procedures: the first one used the Switchboard train set that consists of around 370 hours of data, and the second one includes also the Fisher data and the total size of the train set is over 2300 hours. Both English systems use the Switchboard portion of Hub5 2000 evaluation set as the test set. The Russian data comes from a multi-lingual corpus as described in [9]. In contrast to the English data, the Russian train set is very small, which consists of only 50 hours of training data. This condition is similar to the IARPA Babel project [10] so we could test Sage's performance under low resource condition. The Mandarin data set consists of Mandarin Callhome(LDC96S34), Callfriend(LDC96S55,LDC96S56) and HKUST(LDC2005S15) corpora. All the data sets we used in our experiments are conversational telephone speech(CTS).

Table 1: The data sets used in evaluating Sage

Language	Channel	Train set	Test set
English	CTS	370-hr/2300-hr	2-hr
Russian	CTS	50-hr	4-hr
Mandarin	CTS	250-hr	3-hr

We first compared the modular recipe with an open source recipe in Kaldi. This recipe trains a time delay neural network (TDNN) using Switchboard data [11]. The TDNN consists of 4 hidden layers using P-norm as an activation function. The input to the TDNN consists of MFCC and I-vector features. Table 2 shows the results of modular and open source recipe on the Switchboard test set, and they are consistent.

With the modular design, we tried to combine different recipes. In this experiment, we used the bottleneck features (BN) proposed in [12]. The bottleneck features were prepared using a nnet1 recipe in Kaldi. Then with the features, we perform feature space maximum likelihood linear regression (FMLLR) for speaker adaptation [13], and we used the adapted features to build acoustic models like DNNs using nnet1 as described in [14], TDNNs using nnet2 [11], and also LSTMs using CNTK described in [8]. Table 3 contains the results on the English, Russian and Mandarin data, and it shows that we can improve system performance by combining different toolkits and technologies.

Table 2: Comparing an open source Kaldi recipe with Sage's modular recipe

Language	Model	Recipe	WER(%)
English	TDNN	Open source	14.0
English	TDNN	Modular	14.0

Table 3: Results of combining different tools and recipes using Sage modules

Language	Feature(tool)	Model(tool)	WER(%)
English	BN(nnet1)	DNN(nnet1)	11.5
English	BN(nnet1)	LSTM(CNTK)	10.9
Russian	BN(nnet1)	DNN(nnet1)	38.6
Russian	BN(nnet1)	LSTM(CNTK)	38.2
Mandarin	BN(nnet1)	TDNN(nnet2)	21.2

We evaluated the speed and memory footprint of Sage's decoding pipeline. To evaluate the performance, we used the 2300 hours English system. This system adopts two-pass decoding where it uses a Gaussian mixture model for SI decoding and a DNN for SA decoding. The DNN consists of 6 layers and each layer has 2048 sigmoid units. The input to this DNN is the spliced bottleneck features trained on the same data with FMLLR for speaker adaptation. Table 4 shows the real-time factor and also the memory consumption of two operating points. This result shows that we could build a high performance STT system running at real-time with only 3GB of memory, which has only little degradation compared to the research system.

Table 4: Performance of the English 2300-hr system at different real-time factor and memory consumption

Language	10x RT	1x RT
English(2300-hr)	9.5% (36GB)	10.8% (3GB)

5. Conclusions

In this paper, we introduce Sage - the latest BBN speech processing platform. The design of Sage aims to satisfy the needs of both research and production. Through its modular design and Python interface, Sage can take advantage of many open source toolkits, each of which has its own strengths. This design allows researchers to quickly experiment with new technologies, and ultimately reduces the cost of prototyping. With Sage, we also introduce Godec, which is an all-in-one streaming framework aimed to support various applications. In this paper, we also show how Sage can combine different open source recipes to create a better system. To demonstrate Sage's capability as a production system, we built a high performance English system and showed that it can run in real-time with a modest amount of memory consumption.

6. References

[1] Y. Chow, M. Dunham, O. Kimball, M. Krasner, G. Kubala, J. Makjoul, P. Price, S. Roucos, and R. Schwartz, "BYBLOS: The BBN Continuous Speech Recognition System," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, 1987.

[2] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlíček, Y. Qian, P. Schwarz, J. Silovský, G. Stemmer, and K. Veselý, "The Kaldi Speech Recognition Toolkit," in *Proceedings of the IEEE Workshop on Automatic Speech Recognition and Understanding*, 2011.

[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[4] A. Agarwal, E. Akchurin, C. Basoglu, G. Chen, S. Cyphers, J. Droppo, A. Eversole, B. Guenter, M. Hillebrand, R. Hoens, X. Huang, Z. Huang, V. Ivanov, A. Kamenev, P. Kranen, O. Kuchaiev, W. Manousek, A. May, B. Mitra, O. Nano, G. Navarro, A. Orlov, M. Padmilac, H. Parthasarathi, B. Peng, A. Reznichenko, F. Seide, M. L. Seltzer, M. Slaney, A. Stolcke, Y. Wang, H. Wang, K. Yao, D. Yu, Y. Zhang, , and G. Zweig, "An Introduction to Computational Networks and the Computational Network Toolkit," Microsoft, Tech. Rep. MSR-TR-2014-112, 2014.

[5] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[6] G. Hinton, L. Deng, D. Yu, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, G. Dahl, and B. Kingsbury, "Deep Neural Networks for Acoustic Modeling in Speech Recognition," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[7] T. N. Sainath, A. rahman Mohamed, B. Kingsbury, and B. Ramabhadran, "Deep convolutional neural networks for lvcsr," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2013.

[8] H. Sak, A. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," in *Proceedings of the INTERSPEECH*, 2014.

[9] C. Cieri, J. P. Campbell, H. Nakasone, D. Miller, and K. Walker, "The Mixer Corpus of Multilingual, Multichannel Speaker Recognition Data," in *LREC*, 2004.

[10] T. N. Sainath, B. Kingsbury, F. Metze, N. Morgan, and S. Tsakalidis, "An Overview of the Base Period of the Babel Program," *IEEE SLTC Newsletter*, November 2013.

[11] V. Peddinti, D. Povey, and S. Khudanpur, "A time delay neural network architecture for efficient modeling of long temporal contexts," in *Proceedings of the INTERSPEECH*, 2015.

[12] F. Grézl and P. Fousek, "Optimizing Bottle-neck Features for LVCSR," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2008, pp. 4729–4732.

[13] M. J. F. Gales, "Maximum Likelihood Linear Transformations for HMM-based Speech Recognition," *Computer Speech and Language*, vol. 12, pp. 75–98, 1998.

[14] K. Veselý, A. Ghoshal, L. Burget, and D. Povey, "Sequence-discriminative Training of Deep Neural Networks," in *Proceedings of the INTERSPEECH*, 2013.