Agile MaryTTS Architecture for the Blizzard Challenge 2018

Sébastien Le Maguer[1–3], *Ingmar Steiner*[1–3], *Francesco Tombini*[1,2], *Pradipta Deb*[1,3], *Moitree Basu*[1,3], *Insa Kröger*[1–3]

[1]Multimodal Computing and Interaction, Saarland University, Germany
[2]Language Science & Technology, Saarland University, Germany
[3]Multilingual Technologies Lab, DFKI GmbH, Saarbrücken, Germany

{slemaguer,steiner,ftombini,pradipta,moitreeb,insak}@coli.uni-saarland.de

Abstract

In this paper, we present the MaryTTS entry for the Blizzard Challenge 2018. Our participation is motivated by the use of a new system architecture whose development began three years ago. To this end, we designed a fully modular pipeline which incorporates native modules and distributed processes, including a new grapheme to phoneme conversion (G2P) component. The back-end also supports this modularity, as the fundamental frequency (F_0) is predicted separately, based on a model of its dynamics. A segmental synthesizer using phonetic information and the predicted prosody is then used to produce the final signal. Even though our results are disappointing, the participation has shown that our architecture is functional and that we can now further develop interfaces to several open-source backends. This will hopefully strengthen the role of MaryTTS as a framework for research in speech synthesis.

Index Terms: Speech synthesis, modular architecture, MaryTTS

1. Introduction

Over the past three years, we have been massively restructuring the core of the open-source MaryTTS synthesis system [1]. We participated in the Blizzard Challenge during these years in order to assess this ongoing development. Our unit-selection entry in 2016 [2] aimed to adapt the stable legacy release and to assess its quality. The 2017 entry [3] used the restructured architecture as the front-end, which we coupled with the latest version of HTS [4] at the time.

For this year's Blizzard Challenge, we present the current version of the new architecture of MaryTTS. It has allowed us to develop a fully modular system to handle the front-end as well as the back-end processing. We have tested the modularity to an extreme, based on the use of internal modules, dedicated Java modules developed outside of the MaryTTS system, as well as network-based modules. In order to integrate these modules, we designed a custom data processing pipeline. The goal of this pipeline is to model the different components (e.g., the duration, fundamental frequency (F_0), and segmental parts) as independently as possible. In addition, the grapheme to phoneme conversion (G2P) paradigm was updated. All of this led to a parallel development which produced the different software, data, and model artifacts needed at the synthesis stage.

The remainder of this paper is structured as follows. First, we present the data processing scheme which produces the artifacts needed for the synthesis stage. Then, we focus on the runtime synthesis pipeline, presenting the motivation behind it and how we adapted it for this year's Blizzard Challenge. Finally, we analyze and discuss the evaluation results.

2. Data processing

The entire end-to-end data processing and synthesis pipeline is shown in Figure 1. In contrast to our previous Blizzard entries, we blurred the distinction between "voicebuilding" and subsequently running the system to synthesize the test set. Instead, we designed individual components in dedicated projects; following an input-process-output pattern, each component depends on data artifacts resolved from an internal binary package repository (Figure 1j), processes them, and publishes data (or model) artifacts for consumption by downstream components.

This encapsulation enabled fine-grained control over, and easier debugging within, each step in the pipeline. Moreover, it allowed us to reduce the technical overhead of provisioning dedicated services, to build components in a distributed fashion, and to mitigate dependency version conflicts. Stepwise refinement to individual components led to publishing updated snapshot versions of the output artifacts, triggering updates of downstream components. The entire process was managed using the Gradle[1] build automation tool.

2.1. Raw data

The raw data provided by the Blizzard Challenge organizers was processed as shown in Figure 1a. The audio, which came in a variety of codecs and containers, was first decoded, and then, using both the provided labeling and orthographic representation, aligned with the text, before being split into corresponding text and audio utterances of various length.

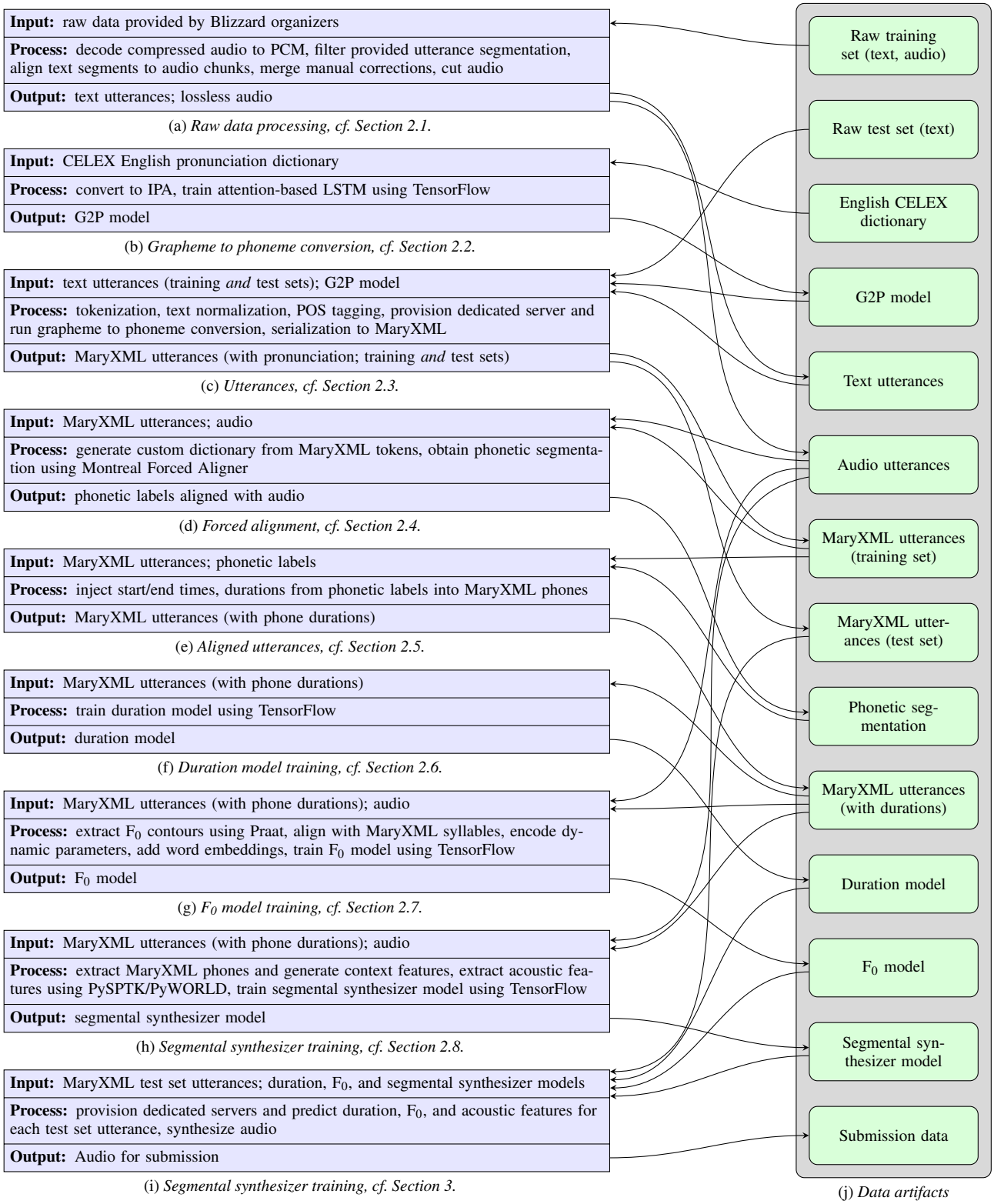
Some of the provided audiobooks were already aligned with the text. In addition to these, we manually added the text alignment for the remaining audiobooks. We extracted the text from the provided PDF files and manually checked it, making sure that it was in the correct order.

Furthermore, we had to roughly align the text to the corresponding audio. To automatize this process, we created TextGrids using a custom Praat script [5]. The TextGrids were then used to strip unwanted artifacts from the audio, such as the chime sound that indicates a page turn. Some audio passages did not fully correspond to the given text; the voice talent took some liberties with the exact wording, but generally this was rare. Nevertheless, in such instances the text had to be updated manually.

2.2. Grapheme to phoneme conversion

For grapheme to phoneme conversion (G2P), the legacy MaryTTS system used dictionaries and pronunciation rules based on finite state transducers (FSTs) and rule-based syllab-

[1]https://gradle.org/



(a) *Raw data processing, cf. Section 2.1.*

(b) *Grapheme to phoneme conversion, cf. Section 2.2.*

(c) *Utterances, cf. Section 2.3.*

(d) *Forced alignment, cf. Section 2.4.*

(e) *Aligned utterances, cf. Section 2.5.*

(f) *Duration model training, cf. Section 2.6.*

(g) *F_0 model training, cf. Section 2.7.*

(h) *Segmental synthesizer training, cf. Section 2.8.*

(i) *Segmental synthesizer training, cf. Section 3.*

(j) *Data artifacts*

Figure 1: *Data processing and synthesis pipeline. On the left, (a) to (i) show dedicated projects for each component, whose input data artifacts are resolved from, and whose output data artifacts are published to, an internal binary package repository (j).*

ification, leading to numerous limitations. To move beyond this, we developed a new experimental deep neural network (DNN) based G2P component, which predicts phonemes, syllable boundaries, and lexical stress with high precision, and uses international phonetic alphabet (IPA) symbols and corresponding phonological features.

We considered the G2P task as a probabilistic model, where given a sequence of graphemes $X = x_1, x_2, x_3, \ldots, x_p$, we compute the conditional probability $\Pr(Y|X)$ of a phoneme sequence $Y = y_1, y_2, y_3, \ldots, y_p$ given by:

$$\Pr(Y|A,X) \approx \prod_{i=1}^{p} \Pr(y_i | y_{i-j}^{i-1}, x_{i-j}^{i+j}) \quad (1)$$

where A is the alignment and j denotes the history windows which are being considered for estimation.

We used a bi-directional encoder-decoder model approach as suggested by Sutskever et al. [6]. The basic architecture of an encoder-decoder model is as follows:

- An encoder reads the grapheme sequence $X = x_1, x_2, x_3, \ldots, x_p$ as a sequence of character vectors.
- The encoder then creates an internal representation as a sequence of hidden state representations $H = h_1, h_2, h_3 \ldots, h_p$ from the embedded input characters (one-hot vector representation).
- A decoder accepts the encoder output to generate the sequence of phonemes one by one. For a particular time t, the decoder generates the phone y_t given $(y_{t-1}, y_{t-2}, y_{t-3}, \ldots, y_1)$.

Most of the time, there is a mismatch between the lengths of the grapheme and corresponding phoneme sequences and the alignment is not one-to-one. Therefore, it is important to gather contextual information from both ends of the grapheme sequence. In order to capture the context from both ends for each grapheme, we used a multi-layered stacked bidirectional LSTM (BiLSTM) model for the encoder and a multi-layered stacked unidirectional long short-term memory (LSTM) for the decoder. For our system, we not only predict the phoneme sequence for a particular grapheme sequence, but also the syllabification and lexical stress (primary, secondary, or none). Our system has the following architecture:

- 3 hidden layers each with 256 units.
- mini-batch stochastic gradient descent (SGD) with an initial learning rate of 0.001 and a batch size of 128.
- an Adam optimizer; in cases where we hit a plateau in our loss curve, we reduced the learning rate by a factor of 0.6.
- a dropout layer on top of stacked LSTMs with a keep_prob rate of 0.8.

Our G2P model was trained on the English pronunciation dictionary from CELEX [7], which we first converted to IPA notation. We ran our model for 150 epochs and saved the model state whenever we observed a drop in word error rate (WER) and phoneme error rate (PER). To evaluate the performance, we split the CELEX data into a 107 355-word training set, a 15 156-word test set, and a 3790-word validation set. We tuned all the hyper-parameters until we obtained a WER of 1.73 %, a PER of 0.29 %, a syllable error rate (SyER) of 0.67 %, and a stress error rate (StER) of 1.06 %.

We integrated the G2P model as shown in Figure 1b, using a dedicated service to predict the pronunciation on demand. Unlike for our previous Blizzard entry [3], we did not adapt or customize the pronunciation prediction for the Blizzard domain.

2.3. Utterances

As shown in Figure 1c, we processed the text utterances from both the training and test sets into a MaryXML structure, using our G2P model in a custom component. The resulting utterances were serialized to MaryXML markup for easy manipulation and consumption in downstream components.

2.4. Forced alignment

To align the utterances with the audio, we ran a forced alignment using the Montreal Forced Aligner (MFA) [8]. The predicted phonemes were first extracted into a custom pronunciation dictionary to avoid any "unknown" tokens (cf. Figure 1d).

2.5. Aligned utterances

To provide the phoneme and pause durations to downstream components (cf. Figure 1e), we injected the segment start and end times from the forced alignment into the MaryXML utterances.

2.6. Duration model

As shown in Figure 1f, we extracted features from the aligned MaryXML utterances to train a duration model, developed as a proof of concept to use DNNs directly in Java using TensorFlow [9], as well as different features at runtime.

The input is a one-hot vector composed of a quinphone context representation of the phonological features for each phoneme. The output feature is the phone duration. The trained model architecture is a feed-forward DNN (FF-DNN) of 5 layers having 512 neurons each. The model was trained using a learning rate of 0.001, a batch size of 100, an Adam optimizer, and the root mean square error (RMSE) as a loss function. Although the maximum number of epochs was set to 50, the model selected was the one obtained after 17 epochs, as no further improvement was observed.

2.7. F_0 model

To model the fundamental frequency (F_0), we adopted a purely dynamic approach [10]. Specifically, the values of encoded F_0 contours do not encode the absolute location of anchor points within the frequency domain, as it is typically done in most state-of-the-art models, but rather, the relative position of each anchor point with respect to the previous one.

Because of the importance of syllables as prosodic units, the syllable was chosen as the support level to undergo internal subdivision; we first assumed a default interval size (0.1 s), adapted to get an even number of subdivisions, which was used to determine the number and times of the anchor points for each syllable. Using the audio and aligned utterances (cf. Figure 1g), we first extracted the F_0 contour using Praat [5]. Then the following recurrent formula was used to produce a dynamic representation of the contour:

$$f_t = \begin{cases} 2^{(n_t/s)} & t = 1 \\ f_{t-1} * 2^{(n_t/s)} & t > 1 \end{cases} \quad (2)$$

where t is the time index, s is the number of pitch levels within each octave (in our case, 24), n_t is the number of pitch levels away from f_{t-1}, and f_t is the frequency of the F_0 value n_t pitch levels away from f_{t-1}. Before the first F_0 value, $n_{t=1}$ is set to 0 after the recursion is complete.

The information encoded by n_t was further decomposed into the *sign* of n_t (viz. −1, 0 and 1 for falling, level, and ris-

ing, respectively), and the *magnitude* of change. The magnitude representation was further compressed by rounding the values to their closest triangular number approximation to capture larger intervals with linearly decreasing precision (11 discrete values were sufficient to encode the F_0 for the entire corpus). Encoding the values in this way allowed us to reformulate the F_0 prediction as a classification, rather than a regression, task. In addition to the categorical sign and magnitude features, we also extracted a number of text-based features, including POS tags, word boundaries, surrounding punctuation, and word embeddings.

Next, the textual features, as well as the corresponding sign and magnitude features, were used to train a DNN model. The objective of the model is to predict the sign and magnitude features from the textual input features. The adopted DNN architecture is shown in Figure 2. In this architecture, we distinguish three stages: word embedding, linguistic interaction, and prediction.

The objective of the word embedding stage is to project the lemma information into a dedicated reduced space. This follows a standard approach in natural language processing (NLP) [11], and this layer is a typical feed-forward DNN (FF-DNN) composed of three layers of 1024, 512 and 256 neurons.

The second stage consists of modeling linguistic interactions. The first layer of this stage is a feed-forward (FF) layer of 256 neurons. Then, we use a bidirectional RNN (BD-RNN) of 512 gated recurrent unit (GRU) neurons each, to capture the contextual interaction.

The last stage uses the output of the second stage to predict the sign and magnitude of the F_0 encoding. This stage is composed of two layers, a simple FF layer of 512 neurons and a forward recurrent layer of 512 GRU neurons, to take the context into account. At prediction time, in order to align the sign and the magnitude, the magnitude recurrent layer depends on the prediction of the sign. This is not the case during the back propagation step, where we stop the gradients from flowing into the sign compoment. This is done to prevent the sign layers from trying to predict the magnitude.

In order to reduce overfitting, we used *dropout* [12], and to increase robustness against various utterance lengths, we augmented the data by grouping contiguous utterances into buckets of various sizes.

The trained model produces sequences of sign and associated magnitude values, from which static F_0 can be decoded by multiplying the signs and the magnitudes to recover the dynamics. We then recursively apply the dynamics to an initial seed value set to 0, to produce a sequence of static coefficients.As the frequency mapping scale does not contain any negative values, we then shift the static coefficients up, so that the lowest value is at 0. Next, we produce an initial F_0 sequence by replacing each static coefficient by its corresponding frequency in Hz and shifting it to the speaker's F_0 range, using the following translation:

$$f_t^* = f_t - \frac{1}{k}\sum_{t=1}^{k} f_t + \overline{f(s)} \tag{3}$$

where t corresponds to the time index, $\overline{f(s)}$ to the average F_0 of speaker s, f_t to the F_0 value at time t, and f_t^* to the F_0 value at time t adapted to speaker s. Finally, the sequence of F_0 values is converted into a continuous contour by means of quadratic interpolation.

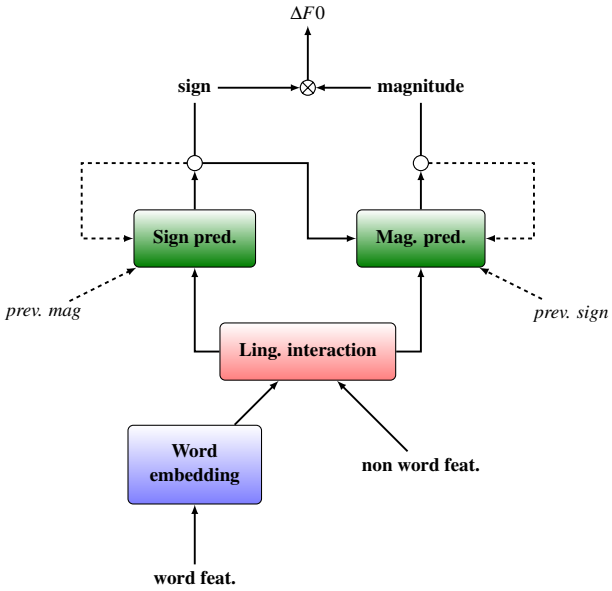


Figure 2: *DNN pipeline. The dependencies on the previous states are represented by dashed arrows, and the forward dependencies, by plain arrows. The blue box corresponds to an FF-DNN, the red box, to a network composed by a FF layer and a bidirectional recurrent layer, and the green boxes, to a network composed of an FF layer and a forward recurrent layer.*

2.8. Segmental synthesizer

With both duration and F_0 modeled using dedicated DNNs, we implemented a segmental synthesizer to predict the remaining acoustic features required by the WORLD vocoder [13] for audio rendering.

As usual, input and output frames were generated at 5 ms intervals. Each input feature vector of the segmental synthesizer consists of the quinphone label, phone duration, the percentage position of the frame within the phone, and the log F_0, extracted from the audio and aligned utterances (cf. Figure 1h). This leads to a vector of size 228, with all input vectors normalized to a standard normal distribution.

The output vector comprises the mel-generalized cepstrum (MGC) and band aperiodicity parameter (BAP) coefficients and the voicing property, extracted using Python wrappers for WORLD and SPTK[2]. This yields a vector of size 68, which is also normalized to a standard normal distribution.

Finally, the synthesizer relies on a DNN model. The neural network performing the synthesis comprises a sequence of 10 self-normalizing FF layers [14], followed by one final RNN layer just before the output. Each layer is composed of 512 nodes and the recurrent one uses GRU neurons. The static coefficients represent the final output of the segmental synthesizer. This model was trained using the same training corpus as the F_0 model. The mel-cepstral distortion (MCD) of analysis/resynthesis using this model is around 7 dB.

3. Synthesis

The core idea of the new MaryTTS architecture is to have a fully modular runtime system. This implies that each module can be replaced by another, as available, without restarting the

[2]http://sp-tk.sourceforge.net/

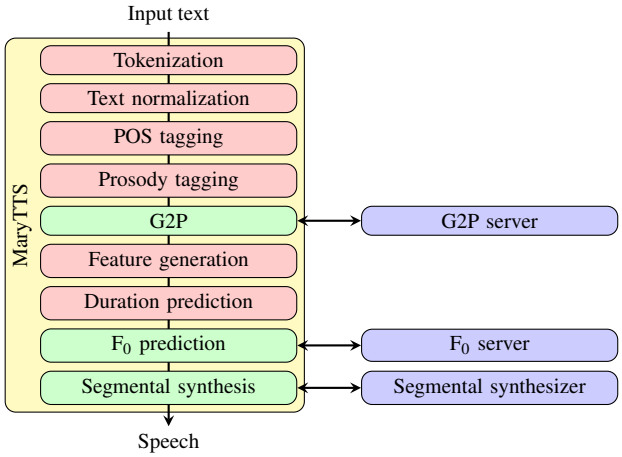


Figure 3: *MaryTTS runtime synthesis pipeline. The boxes indicate native Java modules (red), and client modules (green) communicating with dedicated servers (blue). The communication is implemented by serializing the data to be sent and loading the results in the internal MaryTTS utterance representation. The corresponding configuration is presented in Listing 1.*

```
{
  "marytts.runutils.Request": {
    "logger_level": "INFO",
    "input_serializer": "marytts.io.serializer.TextSerializer",
    "output_serializer": "marytts.io.serializer.TextGridAudioSerializer",
    "module_sequence": [
      "marytts.language.en.JTokenizer",
      "marytts.language.en.Preprocess",
      "marytts.language.en.OpenNLPPosTagger",
      "marytts.modules.nlp.ProsodyGeneric",
      "marytts.g2pdnn.G2PDNNModule",
      "marytts.modules.acoustic.TargetFeatureLister",
      "marytts.modules.BlizzardDNNDurationPrediction",
      "marytts.modules.IntonationPrediction",
      "marytts.modules.SegmentalSynthesizer"
    ]
  },
  "marytts.modules.BlizzardDNNDurationPrediction": {
    "predictor_model": "duration/",
    "normaliser": "QuinphoneNormaliser"
  },
  "marytts.modules.IntonationPrediction": {
    "hostname": "localhost",
    "port": 8850
  },
  "marytts.modules.SegmentalSynthesizer": {
    "hostname": "localhost",
    "port": 8895
  }
}
```

Listing 1: *MaryTTS configuration used for runtime synthesis.*

system. It also implies that we have full control over the module by defining its parameters and which models are loaded. To present these details, we first describe a standard runtime pipeline, and its constraints and advantages. Then, we focus on the application of this pipeline to render the Blizzard samples.

3.1. Runtime pipeline

MaryTTS relies on a client/server architecture. The client sends a process request, and the server processes it and returns the result to the client. Therefore, the server is assumed to be running with all available modules already configured. This is achieved when the server starts, as it inspects the classpath to discover which modules are available.

At the runtime stage, a request sent by a client should contain two pieces of information: the data to process and a configuration. The configuration describes how the data should be loaded, how the results should be rendered, the sequence of processing modules, and the parameters for each module. As the configuration process is based on Java reflection, we are able to fully control the synthesis process and easily extend it by adding the proper setters. The entire pipeline was designed to have a flexible framework which allows prototyping by extending and accessing the process as easily as possible.

3.2. Application for the Blizzard Challenge

As stated above, the goal of our participation is to assess that the synthesis process is valid and can be as modular as desired. To this end, in addition to the synthesis of the submission data shown in Figure 1i, we also applied the runtime synthesis pipeline shown in Figure 3.

We distinguish two kinds of modules, native Java modules and client modules. The client modules are used for G2P and F_0 prediction, as well as segmental synthesis. Each of these requires a dedicated server; in our case, we provisioned Docker containers[3] for G2P and F_0 prediction, in order to satisfy specific CUDA and TensorFlow version requirements and host system configuration compatibility. The reason for this technical overhead is the fact that, at this experimental stage, these modules were developed separately in the context of different student projects. The data used for these modules comes, respectively, from the results of Section 2.2 for the G2P module, the results of Section 2.7 for the F_0 prediction modules, the results of Section 2.8 for the segmental synthesizer.

The remainder of the MaryTTS modules are Java modules. This includes the duration prediction module whose models, described in Section 2.6, are directly loaded for inference in Java. This heterogeneity among the module types demonstrates that MaryTTS can be used for prototyping, which is the main goal of this new framework architecture.

The configuration used for the synthesis is shown in Listing 1. As mentioned in Section 2, in order to have more precise control over the output, and to avoid rerunning the entire pipeline if a problem was detected, we split the process in three main places: after the grapheme to phoneme conversion, after the duration prediction, and after the F_0 prediction. We tested the validity of this multi-step approach by manually comparing the output of the entire process achieved in the step-wise paradigm to that of the "one shot" paradigm, and verified that the results were equivalent. This possibility demonstrates the flexibility and modularity of the system, as well as its debugging capabilities.

4. Results

The results achieved by our system in the Blizzard Challenge evaluation are presented in Figure 4 for naturalness, speaker similarity, and intelligibility. The natural reference "system" is identified by the letter **A**. The benchmark systems are identified by **B** for unit-selection, **C** for hidden Markov model (HMM), **D** for the DNN, and **E** for the DNN with trajectory training. Our system is identified by the letter **H**.

Considering our approach, our hope (aside from testing the new architecture paradigm) was to achieve intelligibility and similarity at the level of the benchmark DNN and improve the naturalness compared to this system. However, the results show that we fell short of this objective, which could be explained by multiple factors. First, the duration model is trained without any knowledge of prosodic information (pause, stress, etc.).

[3] https://www.docker.com/

Figure 4: *MOS for naturalness and speaker similarity and WER for intelligibility for all systems across all listeners.*

In addition, it doesn't predict any sub-phone duration, and is therefore too naïve to be realistic.

The second part concerns the segmental synthesizer. The model directly outputs the acoustic coefficients, without any dynamic information. We assumed that this dynamic could have been captured by the LSTM layer preceding the output layer of the segmental synthesizer network, but we will need to fine-tune the network architecture and hyper-parameters to improve the quality of the output audio.

Finally, notwithstanding the "agile" nature of our synthesis pipeline, the biggest issue turned out to be training time. Unlike during the preparation of last year's Blizzard participation, this year we had access to a dedicated graphics processing unit (GPU), but training the various models using TensorFlow still took several days. Shortly before the submission deadline, we caught a bug in our data cleaning process (cf. Section 2.1) that resulted in text and audio misalignments, which propagated to the forced alignment, and from there further into the downstream components. Although the bug was then fixed, we did not have enough time to rerun the full pipeline before the deadline. We suspect that this may have further negatively affected the quality of our synthesis output.

5. Conclusion

As we have seen in this paper, the MaryTTS system has reached a stage where it can be used to achieve speech synthesis using a variety of state-of-the-art techniques. However, even though the modularity is working as expected, the output quality achieved is not yet optimal. The next stage is to complete development of back-end support by adding compatibility with current state-of-the-art, open-source systems such as HTS [4] and Merlin [15]. To do so, we will extend the data processing pipeline to be compatible with these systems and also to distribute, in a compatible way, models produced by these systems.

Finally, we will also extend the process to integrate baseline objective evaluations which should achieved at the training stage.

6. Acknowledgments

This work was funded by the German Research Foundation (DFG) under grants EXC 284 and SFB 1102. We used a Quadro P5000 GPU donated by the NVIDIA Corporation.

7. References

[1] I. Steiner and S. Le Maguer, "Creating new language and voice components for the updated MaryTTS text-to-speech synthesis platform," in *11th Language Resources and Evaluation Conference (LREC)*, Miyazaki, Japan, 2018, pp. 3171–3175. URL: http://www.lrec-conf.org/proceedings/lrec2018/pdf/1045.pdf

[2] S. Le Maguer and I. Steiner, "The MaryTTS entry for the Blizzard Challenge 2016," in *Blizzard Challenge*, Cupertino, CA, USA, 2016. URL: http://festvox.org/blizzard/bc2016/MARYTTS_Blizzard2016.pdf

[3] ——, "The "uprooted" MaryTTS entry for the Blizzard Challenge 2017," in *Blizzard Challenge*, Stockholm, Sweden, 2017. URL: http://festvox.org/blizzard/bc2017/MaryTTS_Blizzard2017.pdf

[4] H. Zen, T. Nose, J. Yamagishi, S. Sako, T. Masuko, A. W. Black, and K. Tokuda, "The HMM-based speech synthesis system (HTS) version 2.0," in *6th ISCA Speech Synthesis Workshop (SSW)*, Bonn, Germany, 2007, pp. 294–299. URL: https://www.isca-speech.org/archive_open/ssw6/ssw6_294.html

[5] P. Boersma, "Praat, a system for doing phonetics by computer," *Glot International*, vol. 5, no. 9/10, pp. 341–345, 2001.

[6] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in Neural Information Processing Systems 27 (NIPS)*, 2014, pp. 3104–3112. URL: http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf

[7] R. H. Baayen, R. Piepenbrock, and L. Gulikers, "The CELEX lexical database (CD-ROM)," 1995, version 2.5.

[8] M. McAuliffe, M. Socolof, S. Mihuc, M. Wagner, and M. Sonderegger, "Montreal Forced Aligner: trainable text-speech alignment using Kaldi," in *Interspeech*, Stockholm, Sweden, 2017, pp. 498–502, doi:10.21437/Interspeech.2017-1386.

[9] M. Abadi *et al.*, "TensorFlow: A system for large-scalemachine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, USA, 2016, pp. 265–283. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[10] F. Tombini, "A dynamic deep learning approach for intonation modeling," Master's thesis, Saarland University, Saarbrücken, Germany, 2018, doi:10.22028/D291-27237.

[11] O. Levy and Y. Goldberg, "Neural word embedding as implicit matrix factorization," in *Advances in Neural Information Processing Systems 27 (NIPS)*, Montreal, QC, Canada, 2014, pp. 2177–2185. URL: https://papers.nips.cc/paper/5477-neural-word-embedding-as-implicit-matrix-factorization

[12] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014. URL: http://jmlr.org/papers/v15/srivastava14a.html

[13] M. Morise, F. Yokomori, and K. Ozawa, "WORLD: a vocoder-based high-quality speech synthesis system for real-time applications," *IEICE Transactions on Information and Systems*, vol. 99-D, no. 7, pp. 1877–1884, 2016, doi:10.1587/transinf.2015EDP7457.

[14] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, "Self-normalizing neural networks," in *Advances in Neural Information Processing Systems 30 (NIPS)*, Long Beach, CA, USA, 2017, pp. 972–981. URL: https://papers.nips.cc/paper/6698-self-normalizing-neural-networks

[15] Z. Wu, O. Watts, and S. King, "Merlin: An open source neural network speech synthesis system," in *9th ISCA Speech Synthesis Workshop (SSW)*, Sunnyvale, CA, USA, 2016, pp. 202–207, doi:10.21437/SSW.2016-33.