ESPRESSO: A FAST END-TO-END NEURAL SPEECH RECOGNITION TOOLKIT

Yiming Wang1, Tongfei Chen1, Hainan Xu1, Shuoyang Ding1, Hang Lv1,4, Yiwen Shao1, Nanyun Peng3, Lei Xie4, Shinji Watanabe1, Sanjeev Khudanpur1,2

1 Center of Language and Speech Processing, 2 Human Language Technology Center of Excellence, Johns Hopkins University, Baltimore, MD, USA
3 Information Sciences Institute, University of Southern California, Los Angeles, CA, USA
4 ASLP@NPU, School of Computer Science, Northwestern Polytechnical University, Xi’an, China

{yiming.wang, tongfei, hxu31, dings, ysha018, shinjiw, khudanpur}@jhu.edu, npeng@isi.edu, (hanglv, lxie)@mpu-aslp.org

ABSTRACT

We present ESPRESSO, an open-source, modular, extensible end-to-end neural automatic speech recognition (ASR) toolkit based on the deep learning library PyTorch and the popular neural machine translation toolkit FAIRSEQ. ESPRESSO supports distributed training across GPUs and computing nodes, and features various decoding approaches commonly employed in ASR, including look-ahead word-based language model fusion, for which a fast, parallelized decoder is implemented. ESPRESSO achieves state-of-the-art ASR performance on the WSJ, LibriSpeech, and Switchboard data sets among other end-to-end systems without data augmentation, and is 4–11× faster for decoding than similar systems (e.g. ESPNET).

Index Terms—automatic speech recognition, end-to-end, parallel decoding, language model fusion

1. INTRODUCTION

Various open-source toolkits for building automatic speech recognition (ASR) systems have been created, with a notable example being Kaldi [1], a weighted finite state transducer based framework with extensive linear algebra support, that enables traditional hybrid ASR systems [2].

With advances in deep learning, recent work in ASR began paying attention to so-called neural end-to-end systems [3, 4, 5, inter alia], which are characterized by generally smaller code size, and greater portability and maintainability across hardware platforms and software environments. This shift is analogous to the one in the machine translation (MT) community: from feature- and syntax-based statistical machine translation (SMT) systems (e.g. Moses [6], Joshua [7]) to end-to-end neural machine translation (NMT) systems (e.g. OPENNMT [8], OpenSeq2Seq [9], FAIRSEQ [10]). As a result, there have been a few efforts in the ASR research community to create open source neural end-to-end frameworks, most notably ESPNET [11] (See also, Table 1). However, ESPNET has some important shortcomings: (i) the code is not very easily extensible and has portability issues due to its mixed dependency on two deep learning frameworks PyTorch [12] and Chainer [13]; (ii) the decoder, which uses a simple but relatively slow beam search algorithm, is not fast enough for quick turnaround of experiments.

To address these problems, we present ESPRESSO, a novel neural end-to-end system for ASR.1 ESPRESSO builds upon the popular NMT framework FAIRSEQ, and the flexible deep learning framework PyTorch. By extending FAIRSEQ, ESPRESSO inherits its excellent extensibility: new modules can easily be plugged into the system by extending standard PyTorch interfaces. Additionally, we gain ability to perform distributed training over large data sets for ASR.

We also present the first fully parallelized decoder for end-to-end ASR, with look-ahead word-based language model fusion [19], tightly integrated with the existing sets of optimized inference algorithms (e.g. beam search) inherited from FAIRSEQ and tailored to the scenario of speech recognition. Furthermore, an improved coverage mechanism is proposed to further reduce deletion and insertion errors, and is compared with related techniques such as EOS threshold [20]. ESPRESSO provides recipes for a variety of benchmark ASR data sets, including WSJ [21], LibriSpeech [22], and Switchboard [23], and achieves state-of-the-art results on these data sets.

ESPRESSO, by building upon FAIRSEQ, also has the potential to integrate seamlessly with sequence generation systems from natural language processing (NLP), such as neural machine translation and dialog systems. We envision that ESPRESSO could become the foundation for unified speech + text processing systems, and pave the way for future end-to-end speech translation (ST) and text-to-speech synthesis (TTS) systems, ultimately facilitating greater synergy between the ASR and NLP research communities.

2. SOFTWARE ARCHITECTURE AND DESIGN CHOICES

We implement ESPRESSO with the following design goals in mind:

• Pure Python / PyTorch that enables modularity and extensibility;
• Parallelization and distributed training and decoding for quick turnaround of experiments;
• Compatibility with Kaldi / ESPNET data format to enable reuse of previous / proven data preparation pipelines;
• Easy interoperability with the existing FAIRSEQ codebase to facilitate future joint research areas between speech and NLP.

We elaborate our technical rationale in the following sections.

2.1. Input format and dataset classes

Our speech data follows the format in Kaldi, where utterances are stored in the Kaldi-defined SCP format, consisting of space-
delimited lines that follows this template:

\(<\texttt{UttID}> <\texttt{FeatureFile}>:<\texttt{Offset}\>

where \(<\texttt{UttID}>\) is the utterance ID, a key that points to any utterance in the dataset, and \(<\texttt{FeatureFile}>\) is a string interpreted as an extended filename for reading from a binary file (ARK format) storing the actual acoustic feature data, following the practice\(^2\) in Kaldi.

In theory, any kind of acoustic feature vectors (e.g. MFCC) can be stored in the feature file. In ESPRESSO, we follow ESPNet and employ the commonly used 80-dimensional log Mel feature with the additional pitch features (in total, 83 dimensions for each frame).

In FAIRSEQ, there is a concept called "datasets", which contains a set of training samples and abstracts away details such as shuffling and bucketing. We follow this and create the following dataset classes in ESPRESSO:

- \texttt{data.ScpCachedDataset}: this contains the real-valued acoustic features extracted from the speech utterance. Each training batch drawn from this dataset is a real-valued tensor of shape \([\text{BatchSize} \times \text{TimeFrameLength} \times \text{FeatureDims}]\) that will be fed to the neural speech encoder (Section 2.3). Since the acoustic features are large and cannot be loaded into memory all at once, we also implement sharded loading, where given the order of the incoming examples in an epoch, a bulk of features is pre-loaded once the previous bulk is consumed for training / decoding. This helps balance the file system’s I/O load and the memory usage.
- \texttt{data.TokenTextDataset}: this contains the gold speech transcripts as text. Each training batch is a integer-valued tensor of shape \([\text{BatchSize} \times \text{SequenceLength}]\) that will be fed to the decoder (Section 2.3). The raw format just consists of space-delimited lines that follow this template:

\(<\texttt{UttID}> <\texttt{DecodedSequence}>\>

where \(<\texttt{UttID}>\) is the original utterance ID from the SCP dataset, and the \(<\texttt{DecodedSequence}>\) is the raw output of the speech recognition system.

The \textit{aligned results} provide an aligned sequence between the gold reference transcript and the predicted hypothesis. An example is shown below:

\begin{verbatim}
4k9c030b
REF: "QUOTE AN EYE FOR ANY "END-QUOTE
HYP: "QUOTE AN EYE FOR AN EYE "UNQUOTE
STP: D S S
WER: 42.86%
\end{verbatim}

Each such record consists of 5 rows: the first line is the utterance ID; REF and HYP is the reference transcript and the system output hypothesis respectively – these two are aligned using minimal edit distance. The fourth line, STP (step), contains the error the system makes at each decoding step: it could be one of S (substitution), I (insertion) and D (deletion), corresponding to the three types of errors when evaluating the word error rate (WER) commonly used to evaluate speech recognition systems. The last line is WER calculated on this utterance. Such output format facilitates easy human inspection to the different error types made by the system, rendering debugging easier for researchers.

### 2.3. Encoder-Decoder

ESPRESSO supports common sequence generation models and techniques arisen from the research in the ASR and NLP community. The \textit{de facto} standard model, the encoder-decoder with attention [28, 29] (also successfully pioneered by [4] in the speech community), is implemented as our \texttt{models.speech_lstm.SpeechLSTMModel} class. Owing to the modularity and extensibility of ESPRESSO, other models, e.g., Transformer [30], can be easily integrated from the underlying FAIRSEQ.

---


\(^3\) https://github.com/google/sentencepiece.
**CNN-LSTM Encoder** The default encoder we used is a 4-layer stacked 2-dimensional convolution (with batch normalization between layers), with kernel size (3, 3) on both the time frame axis and the feature axis [31, 11]. 2x-downsampling is employed at layer 1 and 3, resulting in 1/4 time frames after convolution. The final output channel dimensionality is 128, with the 21 downsampled frequency features, hence a total of $128 \times 21 = 2688$ dimensional features for each time frame.

Then 3 layers of bidirectional LSTMs [32] are stacked upon the output channels yielded by the stacked convolution layers.

This architecture, with the various dimensionality, number of layers, and other hyperparameters customizable, is implemented in our models.speech_lstm.SpeechLSTMEncoder class.

**LSTM Decoder with Attention** We use a 3-layer LSTM decoder by default, with Bahdanau attention [28] on the encoded hidden states (Luong attention [29] is also implemented). We follow the architecture in the Google Neural Machine Translation (GNMT) system [33], where the context vectors generated by the attention mechanism is fed to all 3 layers of the decoding LSTM. Residual connections [34] are added between the decoder layers. These are implemented in the models.speech_lstm.SpeechLSTMDecoder class.

### 2.4. Training Strategies

**Scheduled Sampling** Scheduled sampling [35] is supported by our system, with promising results in end-to-end speech recognition [36]. With scheduled sampling, at each decoder step, the gold label is fed to the next step with $p$ probability, whereas the predicted token$^4$ is fed with $(1-p)$ probability. In our implementation such mechanism will start at an intermediate epoch $N$ in the training process: the first few epochs are always trained with gold labels. The probability $p$ can be scheduled in the training process: later epochs might use a smaller probability to encourage the model not to rely on the gold labels.

**Label Smoothing** Label smoothing [37] has been proposed to improve accuracy by computing the loss (i.e., cross entropy here) not with the “hard” (one-hot) targets from the dataset, but with a weighted mixture of these targets with some distribution [38]. We support three kinds of these distributions in ESPRESSO:

- **Uniform smoothing** [37]: The target is a mixture of $(1-p)$ probability of the one-hot target and the rest of the $p$ probability mass uniformly distributed across the vocabulary set;
- **Unigram smoothing** [39]: Mixed with a unigram language model trained on the gold transcripts;
- **Temporal smoothing** [40]: Mixed with a distribution assigning probability mass to neighboring tokens in the transcript. Intuitively, this smoothing scheme helps the model recover from beam search errors: the network is more likely to make mistakes that simply skip a subword unit of the transcript.

**Model Selection via Validation** At the end of each training epoch, we compute the WER on the validation set using greedy-search decoding without language model fusion (see Section 3). This is different from the approach in previous frameworks such as ESPNET and FAIRSEQ, where they compute the loss function on the validation set (the gold labels are fed in) to perform model selection. We argue that our approach may be more suitable since free decoding on the validation set is a closer scenario to the final metric on test sets. Owing to efficiency concerns, we do not use full-blown language model fused beam search decoding for validation (arguably this is even better).

We employ learning rate scheduling following FAIRSEQ: at the end of an epoch, if the metric on the validation set is not better than the previous epoch, the learning rate is reduced by a factor (e.g. 1/2).

Empirically we found that the reduction of the learning rate will be less frequent if using WER as the validation metric as compared to the loss value on the validation set. According to [41], a less frequent learning rate reduction generally leads to better performance.

### 3. LANGUAGE MODEL-FUSED DECODING

It is shown in recent research that a pure sequence-to-sequence transduction model for ASR without an external language model component (which is used in traditional hybrid ASR systems) is far from satisfactory [42]. This is in contrast with neural machine translation (NMT) models, where normally no external language model is needed. This performance gap is hypothesized to be caused by the fact that the ASR model is only trained on speech-transcript pairs. The gold transcript set is not large enough to produce state-of-the-art neural language models, which are typically trained on a corpus on the scale of 1 billion words.

In ESPRESSO, we employ shallow fusion [43] as a language model integration technique, which is proven to be effective in speech recognition [42, 44]. The LSTM decoder with shallow fusion computes a weighted sum of two posterior distributions over subword units: one from the end-to-end speech recognition model, the other from the external neural language model.

We support 3 types of external neural language models:

- **Subword-unit language model**: A vanilla LSTM-based language model trained on subword units. Here subword units can either just be characters (with $<$space$>$ as an additional special token) or trained subword units (e.g. BPE [26] or SentencePiece [27]);
- **Multi-level language model** [45]: This is a combination of character-based and word-based language models. Hypotheses in the beam are first scored with the character-based language model until a word boundary ($<$space$>$) is encountered. Known words are then re-scored using the word-based language model, while the character-based language model provides for out-of-vocabulary scores;
- **Look-ahead word-based language model** [19]: This model enables outputting characters for each decoding step with a pre-trained word-based language model, by providing look-ahead word probabilities based on the word prefix (sequence of characters) decoded. This is shown in [19] to be superior to the multi-level language model.

### 3.1. Parallelization with Look-ahead Word-based LMs

The original implementation of the look-ahead word-based language model in ESPNET [11] is not operating on batches, making the decoding speed slow. In ESPRESSO, we devise a fully-parallelized version of the decoding algorithm on GPUs.

In [19], a word-based language model is converted to a character-based one via a technique using prefix trees. The prefix tree automaton $T = (Q, \Sigma, Q_0, \delta, \tau, V)$ is a finite-state automaton (see Fig. 1):

- $\Sigma$ is the character set (including $<$space$>$);
- $V \subseteq \Sigma^*$ is the word vocabulary set and also the final state set;
- $Q_0$ is the initial state of the automaton;
- $Q$ is the finite set of states;
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function; and
- $\tau : Q \rightarrow V$ is the output function.

4 This is the token with the maximum posterior probability resulting from the previous LSTM decoder step. It may not necessarily be gold.
• \( Q = \{ p \subseteq w \mid w \in V \} \subseteq \Sigma^* \) is the set of all prefixes of the words in \( V \) and also the state set;
• \( \epsilon \) is the empty string, which also serves as the initial state;
• \( \tau : Q \times \Sigma \rightarrow Q \) is the state transition function, where given a state and an input character, \( \tau(p, c) = pc \), i.e. a simple concatenation.

A look-ahead word-based LM computes the probability of the next character \( c \in \Sigma \) based on a given word history \( h \) and a word prefix \( p \in Q \) (i.e., a state in the prefix tree automaton):

\[
P(c \mid p, h) = \frac{\sum_{w \in V, p\subseteq w} P_W(pcs \mid h)}{\sum_{w \in V, p\subseteq w} P_W(ps \mid h)}.
\]

where \( P_W(w \mid h) \) is the probability of the word \( w \) predicted by the word-based LSTM language model. In Eq. (1), the numerator is the sum of the probability of all words prefixed by \( pc \), i.e. all possible words that could be generated from \( pc \) if the state is moved from \( p \) to \( pc \); the denominator is the sum of the probability of all possible words at the current state \( p \) (see Fig. 1).

\[19\] proposed an efficient way to compute the sum in Eq. (1). We denote \( p \) precedes \( q \) lexicographically as \( p < q \), and define the upper bound \( \bar{p} \) (the lexicographically greatest element prefixed by \( p \)) and lower bound \( p \) (the greatest element lexicographically less than any word prefixed by \( p \)) as:

\[
\bar{p} = \max_{w \in V, p\subseteq w} w; \quad p = \max_{w \in V, p\subseteq w, w < p} w
\]

(2)

Given that the vocabulary set is sorted lexicographically, we can efficiently compute the sum of the probability of all words preceding or equal to a given word, using efficient routines like \text{cumsum}:

\[
g(w \mid h) = \sum_{w \geq w} P_W(w' \mid h)
\]

(3)

Using these definitions, Eq. (1) can be rewritten as

\[
P(c \mid p, h) = \frac{g(pc \mid h) - g(pcs \mid h)}{g(\bar{p} \mid h) - g(p \mid h)}
\]

(4)

This method is illustrated in Fig. 1, showing that the probability of a character given a prefix can be efficiently computed via a \text{cumsum} operation and simple arithmetics.

In our parallelized implementation, each prefix \( p \) (corresponding to a state in the automaton) is indexed as a unique integer. Hence the batch of decoding states is compactly stored as a vector of integers, each corresponding to a state on the prefix tree automaton. The automaton itself is stored as a matrix \( T \) with shape \([\text{NumberOfStates} \times \text{MaxOutDegree}]\), where the row \( T_p \) contains the index of all descendants of \( p \), logically forming an adjacency list. The index of the \( \bar{p} \) and \( p \) for each state \( p \) is also precomputed and cached. In sum, the entire prefix tree automaton is vectorized.

To compute \( P(c \mid p, h) \) for all \( c \in \Sigma \) over batches and beam hypotheses, the following steps are executed:

(i) Update \( P_W(w \mid h) \) for all \( w \in V \) from a specific decoding step of the word-based language model for those hypotheses that encounter the end-of-word (\langle space \rangle) symbol in the batch;
(ii) Update the \( g(w \mid h) \) function using \( P_W(w \mid h) \) for all \( w \in V \);
(iii) Get all possible successive states;
(iv) Get all upper and lower bounds for all successive states;

We denote \( p \) is a prefix of \( q \) as \( p \subseteq q \); e.g. "stand" \( \subseteq \) "standard".

3.2. Improved Coverage

With language model fusion, the decoder tends to make more deletion errors when the language model weight becomes larger [47]. A coverage term (a scalar value assigned to each hypothesis in the beam) is first proposed in [40] to promote longer transcripts and also to prevent attentions from looping over utterance (repeating certain \( n \)-grams when decoding) when using shallow fusion:

\[
\text{coverage} = \sum_j \mathbb{1} \left[ \sum_i a_{ij} > \tau \right]
\]

(5)

where \( a_{ij} \) is the attention weight for the decoder step \( i \) and encoder frame \( j \), \( \tau > 0 \) is a tunable hyper-parameter, and \( \mathbb{1}[] \) is the indicator function. This is the total number of encoder frames that has been sufficiently “attended” to. However, based on our experiments, applying the coverage term Eq. (5) is not sufficient to prevent words from being repeated. Instead we propose a modified version of the coverage term which penalizes looping attentions more aggressively:

\[
\text{coverage} = \sum_j \mathbb{1} \left[ \sum_i a_{ij} > \tau_1 \right] - \mathbb{1} \left[ \sum_i a_{ij} > \tau_2 \right] \left( c + \sum_i a_{ij} - \tau_2 \right)
\]

(6)

where \( \tau_2 > \tau_1 > 0, c \geq 0 \) are tunable hyper-parameters. While the first term in Eq. (6) is exactly the same as Eq. (5), the second term penalizes the hypothesis score when the accumulated attention on encoder frame \( j \) exceeds \( \tau_2 \). Specifically, if the accumulated attention weight on frame \( j \) exceeds \( \tau_1 \) but not \( \tau_2 \), only the first term is activated, increasing the coverage score to encourage more attention on
frame \( j \); if the accumulated attention weight further exceeds \( \tau_2 \), the second term (with the minus sign) is also activated leading to both less deletion errors and less repeating n-grams (shown in the WSJ part of Section 4). During beam search decoding this new coverage term as a whole is added to the hypothesis score with a weight (e.g. 0.01). In our experiments, \( \tau_1 = 0.5, \tau_2 = 1.0, \sigma = 0.7 \).

### 3.3. EOS Threshold

We implement the end-of-sentence threshold technique proposed in [20] to bias the decoder away from short transcriptions when decoding with a fused language model. End-of-sentence (\(<\text{eos}>\)) tokens can only be emitted if its probability is greater than a specific factor of the top output token candidate during beam search:

\[
\log P(<\text{eos}> \mid \mathbf{h}) > \gamma \cdot \max_{t \in \mathcal{V}} \log P(t \mid \mathbf{h})
\]

(7)

where \( \mathcal{V} \) is the vocabulary set. In our experiments, \( \gamma \) is set to 1.5.

### 4. RECIPES AND RESULTS

**ESPResso** provides running recipes for a variety of well-known data sets. We elaborate the details of our recipes on Wall Street Journal [21] (WSJ), an 80-hour English newspaper speech corpus, LibriSpeech [22], a corpus of approximately 1,000 hours of read English speech, and Switchboard [23] (SWBD), a 300-hour English telephone speech corpus.

Besides the transcripts, all of these data sets have their own extra text corpus for training language models. Input and output word embeddings are tied [48] to reduce model size. All the models are optimized using Adam [49] with an initial learning rate \( 10^{-3} \), and then halved if the metric on the validation set at the end of an epoch does not improve over the previous epoch. The training process stops if the learning rate is less than \( 10^{-5} \). Curriculum learning [50], which is quite helpful to stabilize training with long sequences (e.g. LibriSpeech) and improve performance (esp. SWBD), is employed for the first 1 (LibriSpeech) or 2 (WSJ / SWBD) epochs. All the models are trained / evaluated using NVIDIA GeForce GTX 1080 Ti GPUs. If not otherwise specified, all the models throughout this paper are trained with 2 GPUs using **fairseq** built-in distributed data parallelism. Note that no data augmentation techniques such as speed-perturbation [51] or the more recent SpecAugment [41] is applied.

The hyper-parameters for the recipes are listed in Table 2.

<table>
<thead>
<tr>
<th>Hyper-parameter</th>
<th>WSJ</th>
<th>LibriSpeech</th>
<th>SWBD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vocab. size</td>
<td>65k</td>
<td>52</td>
<td>5k</td>
</tr>
<tr>
<td>Encoder # layers</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Decoder # layers</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Emb. dim.</td>
<td>1.200</td>
<td>48</td>
<td>800</td>
</tr>
<tr>
<td>Hidden dim.</td>
<td>1.200</td>
<td>320</td>
<td>800</td>
</tr>
<tr>
<td># Params.</td>
<td>113M</td>
<td>18M</td>
<td>25M</td>
</tr>
<tr>
<td>Dropout rate</td>
<td>0.35</td>
<td>0.4</td>
<td>0.0</td>
</tr>
<tr>
<td>Avg. batch size</td>
<td>435</td>
<td>48</td>
<td>1,733</td>
</tr>
<tr>
<td>Beam size</td>
<td>50</td>
<td>60</td>
<td>35</td>
</tr>
<tr>
<td>LM fusion weight</td>
<td>0.9</td>
<td>0.47</td>
<td>0.25</td>
</tr>
</tbody>
</table>

### Table 3. WERs (%) on the WSJ dev93 and eval92 set.

<table>
<thead>
<tr>
<th></th>
<th>dev93</th>
<th>eval92</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hadian et al.</td>
<td>-</td>
<td>4.1</td>
</tr>
<tr>
<td>Baskar et al.</td>
<td>-</td>
<td>3.8</td>
</tr>
<tr>
<td>Likhomanenko et al.</td>
<td>6.4</td>
<td>3.6</td>
</tr>
<tr>
<td>Zeghidour et al.</td>
<td>6.8</td>
<td>3.5</td>
</tr>
<tr>
<td>Amodei et al.</td>
<td>4.4</td>
<td>3.1</td>
</tr>
<tr>
<td><strong>ESPResso LSTM</strong></td>
<td>14.8</td>
<td>12.1</td>
</tr>
<tr>
<td>+Look-ahead Word LM</td>
<td>7.4</td>
<td>5.1</td>
</tr>
<tr>
<td>+Improved Coverage</td>
<td>5.9</td>
<td>3.5</td>
</tr>
<tr>
<td>+EOS Threshold</td>
<td>5.9</td>
<td>3.4</td>
</tr>
</tbody>
</table>

**WSJ** We adopt the look-ahead word-based language model [19] as the external language model. We report the perplexities on the validation / evaluation set: the external word-based language model achieves 72.0 perplexity on dev93 and 59.0 on eval92.

For the encoder-decoder model, the vocabulary size of subword units (characters) for WSJ is 52, the same as in ESPNET [6] Temporal label smoothing with \( p = 0.05 \) and scheduled sampling with \( p = 0.5 \) starting at epoch 6 are adopted.

Baseline end-to-end systems are compared: Hadian et al. [15], an end-to-end [7] model with the lattice-free MMI objective [52]; Baskar et al. [36], an encoder-decoder model with discriminative training in ESPNET; Zeghidour et al. [53], a pure convolutional network with ASG loss [54]; Likhomanenko et al. [55], a lexicon-free decoding method with the acoustic model proposed in [53]; and the last one, Deep Speech 2 [56].

We show the beam search decoding results of various configurations of ESPRESSO in Table 3 with beam size 50. The breakdown of the three kinds of errors is shown in Table 4. The first row gives WERs where no language model fusion is applied. The second row is after integrating the look-ahead word-based language model, with its optimal LM fusion weight 0.5. Although it has already significantly reduced the overall WER, deletion errors increase. Further applying the improved coverage yields better performance by suppressing deletion errors. If we only use the first term in Eq. (6) which is equivalent to the coverage term in [40], the insertions errors will increase from 0.8 to 1.3 on dev93, and from 0.6 to 0.9 on eval92. A manual inspection of the decoded results reveals that these additional insertions are mostly repeated words. This observation validates the effectiveness of our improved coverage mechanism. Alternatively, if we apply the EOS threshold, we achieve state-of-the-art performance on WSJ among end-to-end models.

**LibriSpeech** SentencePiece is used as the subword units in our LibriSpeech setup for both external language modeling and encoder-decoder model. We combine dev-clean and dev-other sets together as a single validation set for both language model and encoder-decoder model training.

---

1. There are 8 additional symbols constituting the training transcriptions, plus 3 atomic symbols: \(<\text{IN}>, <\text{MR}>, <\text{NOISE}>\), plus 4 special symbols: \(<\text{pad}>, <\text{eos}>, <\text{unk}>, <\text{space}>\).
2. For the encoder-decoder model, the vocabulary size of subword units (characters) for WSJ is 52, the same as in ESPNET [6] Temporal label smoothing with \( p = 0.05 \) and scheduled sampling with \( p = 0.5 \) starting at epoch 6 are adopted.
3. This is a hybrid system. “End-to-end” here is in the sense that it does not need HMM-GMM training or tree-building steps.
4. The result is with “full bichar” (using all possible 2-gram characters as context-dependent modeling units), data speed-perturbation, and a 3-gram word language model.
5. It uses 12,000 hours transcribed data for acoustic model training and large crawled text for language model training, making it not comparable.
### Table 4. Breakdown of the WERs (%) on WSJ.

<table>
<thead>
<tr>
<th></th>
<th>dev93</th>
<th>eval92</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sub</td>
<td>Ins</td>
</tr>
<tr>
<td>ESPRESSO LSTM</td>
<td>12.0</td>
<td>1.4</td>
</tr>
<tr>
<td>+Look-ahead Word LM</td>
<td>4.6</td>
<td>0.8</td>
</tr>
<tr>
<td>+Improved Coverage</td>
<td>4.3</td>
<td>0.8</td>
</tr>
<tr>
<td>+EOS Threshold</td>
<td>4.1</td>
<td>0.9</td>
</tr>
</tbody>
</table>

### Table 5. WERs (%) on the LibriSpeech dev and test sets.

<table>
<thead>
<tr>
<th></th>
<th>dev</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>clean</td>
<td>other</td>
</tr>
<tr>
<td>Zeghidour et al. [53]</td>
<td>3.1</td>
<td>10.0</td>
</tr>
<tr>
<td>Hannun at al. [20]</td>
<td>3.0</td>
<td>8.9</td>
</tr>
<tr>
<td>Park et al. [41] (w/o SpecAugment)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Lüscher et al. [57]</td>
<td><strong>2.6</strong></td>
<td><strong>8.4</strong></td>
</tr>
<tr>
<td>ESPRESSO LSTM</td>
<td>3.8</td>
<td>11.5</td>
</tr>
<tr>
<td>+Subword LM</td>
<td>3.3</td>
<td>8.9</td>
</tr>
<tr>
<td>+Improved Coverage</td>
<td>2.9</td>
<td>8.8</td>
</tr>
<tr>
<td>+EOS Threshold</td>
<td>2.8</td>
<td><strong>8.4</strong></td>
</tr>
</tbody>
</table>

Again, we report the perplexities on the validation / evaluation sets. ESPRESSO obtains 35.4 and 37.3 on the dev-clean and dev-other sets, and 37.2 and 37.2 on test-clean and test-other.

Uniform smoothing with $p = 0.1$ is applied throughout the entire training. No scheduled sampling is used. The vanilla shallow fusion is used without the “look-ahead” technique. The results, along with several baseline systems, are demonstrated and compared in Table 5.

We can see that both the improved coverage or EOS threshold help in this setup as well, where actually deletion error reductions contribute mostly. In addition, we achieve state-of-the-art results on end-to-end models for LibriSpeech without any data augmentation.

### Switchboard

The vocabulary consists of 1,000 subword units segmented by SentencePiece trained on both Switchboard and Fisher transcripts. As there is no official validation set, we hold out the same 4,000-example subset of the training data as in Kaldi for validation. We apply scheduled sampling starting at epoch 6 with probability from 0.9 to 0.6. Uniform smoothing is used with $p = 0.1$.

The language model achieves a validation perplexity of 17.3. No coverage or EOS threshold does not help on this dataset, and we suspect it is because its optimal LM fusion weight is not as large as those for the other two datasets, resulting in less deletion errors.

### 5. TRAINING AND DECODING SPEED

In this section we compare ESPRESSO and ESPNET on the training and decoding time with single GPU, using the WSJ dataset.

For fair comparison, we create architectures in ESPRESSO (different from those in Section 4) that mimics the WSJ recipe in ESPNET as closely as possible. Data preparation and vocabulary building are identical. The neural architecture is mostly the same (e.g. number and dimension of LSTM layers), with a few minor exceptions: e.g. ESPNET’s use of location-based attention (which ESPRESSO does not employ), pooling CNN layers (ESPRESSO uses strided CNN without pooling), and joint cross-entropy+CTC loss (ESPRESSO uses only cross-entropy loss).

Table 7 gives training wall time comparisons on both the external word-based language model and the encoder-decoder model, which are averaged over 20 epochs and 15 epochs respectively. ESPRESSO is 17.9% faster than ESPNET on the language model training, and 16.0% faster on the encoder-decoder model training. We conjecture that the main reason of such speed gain for language model training is that in FAIRSEQ (and hence in ESPRESSO) training examples are sorted based on input sequence lengths before batching (i.e., bucketing; ESPNET does not use it for language modeling), so that the average sequence length in batches is smaller.

A notable advantage of ESPRESSO compared to ESPNET is the decoding speed. In order to have a more fair comparison, we enable GPU batch decoding in ESPNET [60], and make batch and beam sizes of the two systems the same. We measure the decoding time on the WSJ eval92 data set, which consists of 333 utterances. Table 7 shows that, without language model fusion, ESPRESSO is 3.7× faster than ESPNET. With the look-ahead language model fusion, the speedup is even more prominent—more than 10× faster—mostly due to our parallelized implementation of the look-ahead language model fusion (cf. Section 3.1), as opposed to ESPNET, where LM scores are computed iteratively over examples within a minibatch.

### 6. CONCLUSION

In this paper we present ESPRESSO, an open-source end-to-end ASR toolkit built on top of FAIRSEQ, an extensible neural machine translation toolkit. In addition to advantages inherited from FAIRSEQ, ESPRESSO supports various other features for ASR that are seamlessly integrated with FAIRSEQ, including reading data in Kaldi format, and efficient parallelized language model-fused decoding. We also provide ASR recipes for WSJ, LibriSpeech, and Switchboard data sets, and achieve state-of-the-art performance among end-to-end systems. By sharing the underlying infrastructure with FAIRSEQ, we hope ESPRESSO will facilitate future joint research in speech and natural language processing, especially in sequence transduction tasks such as speech translation and speech synthesis.


7. REFERENCES


