BSP S5 - Attention-Based Models for Speech Recognition

Thursday 8th April, 2021

Le Minh Nguyen
University of Luxembourg
Email: le.nguyen.001@student.uni.lu

Vladimir Despotovic
University of Luxembourg
Email: vladimir.despotovic@uni.lu

***Abstract*—Deep Neural Networks (DNNs) improved many components of speech recognizers. They are commonly used in the DNN-Hidden Markov model (DNN-HMM) based speech recognition systems for acoustic modelling. Traditionally these components, such as acoustic, pronunciation and language models, have all been trained separately with different objectives. Recent work in this area tries to solve this disjoint training issue by creating models that are trained end-to-end, in other words, converting speech directly to transcripts. Unlike traditional DNN-HMM models, the attention-based model learns all the components of a speech recognizer jointly. This system has two components: a listener and a speller. The listener is a recurrent neural network encoder which processes filter bank spectra. The spelling component is an attention-based recurrent network decoder which outputs characters. During our experiments, our Listen, Attend and Spell (LAS) model trained on the LibriSpeech corpus, achieves a WER of 13.8%.**

1. Introduction

This paper presents the Bachelor Semester Project (BSP) made by Le Minh Nguyen together with Vladimir Despotovic as his motivated tutor. The main objective of this project is to present the architecture of an attention-based speech recognition model. An additional objective is to reuse an existing implementation of the attention-based model, adapt it to the scope of this project and compare its performance to the results of other state-of-the-art speech recognition models. Furthermore, the implemented speech recognition model should be trained on the LibriSpeech dataset which contains a large-scale corpus of read English speech. Finally, its performance will be assessed with different sets of tuned hyperparameters.

2. Project description

2.1. Domains

- Speech Recognition
- Artificial Neural Networks
- Deep Learning
- Feature extraction
- Training, validation and testing set
- Python
- Pytorch
- HPC developement environment

2.1.1. Scientific

The scientific aspects covered by this Bachelor Semester Project are the concepts of Speech Recognition and Deep Learning. The Listen, Attend and Spell (LAS) model's architecture will be investigated for the end-to-end automatic speech recognition task.

Speech Recognition. The objective of speech recognition is to map audio signals which contain a set of spoken natural language expressions to the matching sequence of words produced by the speaker. In the past, Automatic Speech Recognition (ASR) was made up of different modules such as complex feature extraction, acoustic models, language and pronunciation models. [1] Sequential models, such as Hidden Markov Models (HMM), in combination with a pre-trained language model, were used to map sequences of phones to output words. [2] A different approach is to build ASR models end-to-end. With deep learning, it replaces most of the modules with a single module. This alternative method is the main task of this report, focusing on a sequence to sequence model with attention mechanisms to create an end-to-end ASR pipeline.

Artificial Neural Networks (ANN). ANNs are computing systems inspired by the biological brain. These systems are based on a set of connected units called artificial neurons. Each connection can transmit *signals* between units. A unit can process the signal and transmit it to another unit.

Deep Learning. This field deals with learning patterns by decomposing a task's input into smaller and simpler compositions. With Deep Learning, computing systems can build complex concepts from a composition of simpler concepts.

2.1.2. Technical

The technological aspects which are covered in this project are feature extraction, reusing an existing implementation of an attention-based sequence-to-sequence

ASR model and benchmarking its performance using different configurations and hyperparameters.

Feature extraction. Data preprocessing is an important phase in machine learning. It ensures the quality of the gathered data by eliminating irrelevant and redundant information. Data preprocessing contains tasks such as cleaning, instance selection, normalization, feature extraction and feature selection. We will focus on feature extraction in this paper with the presentation of Mel filterbanks in Section 4.2.1 which are used as features extracted from a speech signal.

Training, validation and testing set. For a computing system to learn from and make predictions on data, a mathematical model is built from input data. This input data used to create the model consists of three datasets: The training, validation and testing set. The training set contains pairs of an input vector, which represent features and an output vector often called the labels. With the training set, the model learns to map the input vector to the labels. Further, the validation set serves for tuning the hyperparameters of the model, whereas the testing set evaluates how well the model generalizes the prediction over the dataset previously not seen by the model.

Python. This is a programming language which is interpreted, high-level and general-purpose. [3]

Pytorch. This library is a Python package which provides high-level features such as GPU accelerated Tensor computation and building Deep neural networks on a tape-based autograd system. It is designed for easy-to-use and efficient experimentation with Deep Learning through a user-friendly front-end. [4]

HPC environment. High-performance computing (HPC) is the capability to process and compute large amounts of data at a fast pace. Implementations of HPC are often referred to as *supercomputers* which consists of many connected computing servers called *nodes*. These nodes work in parallel to compute tasks faster. [5]

2.2. Targeted Deliverables

2.2.1. Scientific deliverables

The scientific aspect covered by this Bachelor Semester Project is the architecture of an attention-based sequence to sequence model. The focus of the scientific presentation will be on the architecture of this model and how it compares to other state-of-the-art models. Further, the model's performance will be assessed when using different hyperparameters. Another important deliverable is to present the extraction of Mel Filterbanks features from the dataset.

2.2.2. Technical deliverables

The technological aspect of this project is the reuse and adaptation of an existing implementation of an attention-based speech recognition model. Additionally, the LibriSpeech dataset will be preprocessed and used for the model's training. The goal is to deploy its training in an HPC environment for efficient computing.

2.3. Constraints

For this BSP, we set different constraints for the speech data set and the LAS model implementation.

Data set. This project does not focus on the collection of a data set. The focus lies on training our model on the LibriSpeech corpus which is an English data set containing approximately 1000 hours of read English speech. This data set is gathered from read audiobooks from the LibriVox project. [6]

LAS implementation. We do not implement the LAS architecture from the ground up since various implementation already exists. Therefore, the focus will lie on choosing a mature existing implementation, explaining its structure and how to use it efficiently for our benchmark.

3. Pre-requisites

To start on the project, certain skills in programming, mathematics and machine learning are required. In particular, the preliminary requirements of the project is as follows:

- Introduction to deep learning-based speech recognition
- Knowledge of machine learning, data preprocessing and model training
- Knowledge of Recurrent Neural Networks (RNNs) and Long short-term memory units (LSTMs).
- Understanding of vector and matrix algebra
- Introductory course in Python
- Software development

3.1. Scientific pre-requisites

Speech Recognition. A subfield combining computer science and computational linguistics which develops Automatic Speech Recognition (ASR). ASR is a methodology which enables recognizing and translating spoken language into text by machines. The objective of speech recognition is to map audio signals which contain a set of spoken natural language expressions to the matching sequence of words produced by the speaker. In the past, Automatic Speech Recognition (ASR) was made up of different modules such as complex feature extraction, acoustic models, language and pronunciation models. [1]. A different approach is to build ASR models end-to-end, by joint feature and model learning, without creating individual components as in conventional pipeline. With deep learning, most of the modules are replaced with a

single module to create a single end-to-end ASR pipeline.

Machine learning, data preprocessing and model training. Machine Learning (ML) is the study of algorithms that gives software applications the ability to perform decisions and predictions without explicit programming. ML's basis is to create algorithms that can receive input data and learn by themselves. The learning process improves its knowledge or performance through experience. ML is the idea of learning from data examples and deducing patterns. In other words, instead of having an explicit set of instructions in a program, we feed data into an algorithm and let the computer find patterns in the given data set.

Data preprocessing is the process of transforming raw data into more useful representations which make them more suitable for ML models to better deduce patterns and increase their performance.

Machine learning contains the model training process. This modelling process maximizes the model's performance while mitigating overfitting. During this process, the model will be trained with a set of hyperparameters and these are tuned accordingly over the training phase. Finally, the best performing model will be selected through performance metrics.

Recurrent Neural Networks and Long short-term memory units. Recurrent Neural Networks (RNNs) are a particular architecture of Artificial Neural Networks (ANNs) which can process sequential data. In traditional ANN the input and output data are assumed to be independent. However, for sequential data, this is usually not the case. For example, the prediction of the next word in a sentence is dependent on the words that appeared before. As opposed to Multilayer Perceptrons (MLPs), RNNs feedback their outputs back to their network. Thus they gain an overview of past inputs which will be beneficial for processing sequential time-series. Although, deep RNNs are challenged by learning long-term dependencies, effective sequence models such as gated RNNs are used to deal with this challenge. An example of a gated RNN is the Long short-term memory (LSTM) unit. This unit is composed of a memory cell, an input gate, an output gate and a forget gate. The cell stores information over a long period and the three gates control the flow of information through the cell. RNNs using LSTM units deal with the vanishing gradient problem because these units allow gradients to flow unchanged. [7]

Linear Algebra. A sub-field of mathematics, which works with vectors and matrices. Since the input of deep learning is data that are transformed into structures of rows and columns, linear algebra is one of the key foundations of deep learning. It is used to describe the operations of the deep learning algorithms, and implement the algorithms in code. All tasks in deep learning relate to linear algebra, from data preprocessing to the deep learning algorithms. [8]

3.2. Technical pre-requisites

Python is an interpreted, high-level and general-purpose programming language, which is conceived in the late 1980s and released in 1991 by Guido van Rossum. [9] Its design philosophy accentuates readability of the code. As many high-level programming languages, Python is dynamically typed. Further, it supports multiple programming paradigms such as procedural, object-oriented and functional programming.

Software development. Software development is the process of designing and implementation of applications and frameworks. In general, the process of software development includes writing and maintaining the source code which is often a planned and structured process. The software development contains mostly research, prototyping, modification and reuse of existing software. During the technical deliverable section, we use this structured process to analyse existing LAS implementation and adapt it to the scope of this project.

4. Scientific Deliverables

In this section, the defined scientific deliverables will be presented. The architecture of an attention-based sequence to sequence model will be investigated and compared to other state-of-the-art models. Further, the Mel Filterbanks features will be introduced and it will be shown how to extract them from the data set. Finally, the model's performance trained on the Librispeech dataset will be assessed while using different hyperparameters.

4.1. Requirements

- **FR01** Explain the Mel-Filterbanks features.
- **FR02** Show the extraction of Mel filterbanks features from an audio signal.
- **FR03** Investigate the architecture of an attention-based sequence to sequence model.
- **NFR01** Performance evaluation and comparison. During this section, we present and discuss the results obtained from training our LAS model on the LibriSpeech corpus while using different hyperparameters.

4.2. Design

4.2.1. FR01: Feature extraction of Mel-Filterbanks

Before training an end-to-end ASR model, audio data have to be preprocessed. The first task is to extract low-level features from the data which describe natural language expressions and exclude background noises and emotions. Speech processing is an important process when creating ASR systems. For a long period, Mel-Frequency Cepstral

Coefficients (MFCCs) were the most popular features. However, Mel-Filterbanks are recently getting more prominent with applications in deep learning. Computing both features contain the same approach where both of them calculate filterbanks, but MFCCs include an additional step of applying Discrete Cosine Transform (DCT) to decorrelate the features. While this is preferred for most of the traditional machine learning algorithms, in DNNs it is not necessary since they are less susceptible to highly correlated input [10]. Note that DCT is a linear transform, therefore, it discards some non-linear information in speech signals, which is not desirable in DNNs.

To extract the Mel-Filterbanks from the dataset, the following extraction steps have to be executed on a speech signal sampled at $16kHz$:

1. Frame the signal into $25ms$ frames. The frame length of a $16kHz$ signal:

$$0.025 * 16000 = 400 samples. \tag{1}$$

With a frame step of 160 $samples$ or $10ms$, the frames overlap themselves, such that the first frame containing 400 $samples$ starts at sample 0 and the second frame starts at sample 160. This pattern repeats until the end of the speech signal. If the audio file is not divisible by an even number, it is padded by zeros.

2. For each frame calculate the periodogram estimate of the power spectrum. To obtain the Discrete Fourier Transform (DFT) from the frame i, we perform:

$$S_i(k) = \sum_{N}^{n=1} s_i(n) h(n) e^{-j2\pi kn/N} \tag{2}$$

For each speech frame $s_i(n)$, the periodogram-based power spectral estimate is calculated with:

$$P_i(k) = \frac{1}{N} \mid S_i(k) \mid^2 \tag{3}$$

This is called the Periodogram estimate of the power spectrum which identifies for every frame which frequencies are present.

3. Apply the Mel-filterbank to the power spectrum and sum the energy in each filter. The Mel-filterbank is a set of 40 triangular filters. Each filter within the filterbank is triangle shaped with a value of 1 at its frequency center and decreases to 0 reaching the center of the neighboring filter. This Filterbank on a Mel-Scale is shown in Fig. 1.

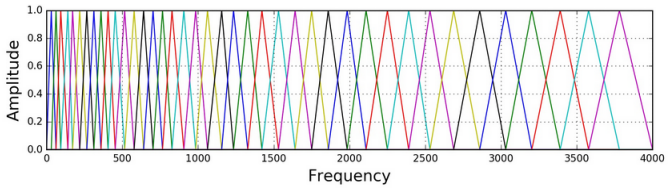

Figure 1: Filterbank on a Mel-Scale. [10]

The filterbank energies are calculated by multiplying every filterbank with the power spectrum of the signal. The resulting spectrogram can be seen in Fig. 2.

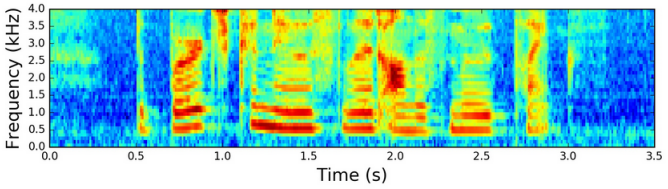

Figure 2: The spectrogram of the signal. [10]

To obtain the Mel-filterbank features, the mean of each coefficient is subtracted from all frames. The mean-normalized Mel-Filterbanks is show in Fig. 3.

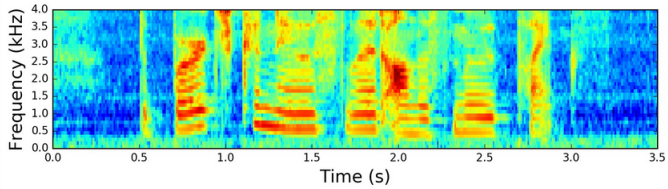

Figure 3: Normalized Mel-Filterbanks. [10]

In Section 4.3.1 we will use the *Torchaudio* module from the Pytorch Library to extract the Mel-Filterbank features from the dataset.

4.2.2. FR03: Investigation of the architecture of an attention-based sequence to sequence model.

4.2.2.1 Introduction

In this section, the *Listen, Attend and Spell* (LAS) model will be investigated and presented based on its paper [11] and some definitions found in the *Dive into Deep Learning* textbook. [12] LAS is a model which takes acoustic features as inputs and outputs characters as outputs. We define, $x = (x_1, ..., x_T)$ an input sequence containing Mel-filterbank features and $y = (\langle sos \rangle, y_1, ..., y_S, \langle eos \rangle)$ an ouput sequence of characters where $y_i \in \{a, b, ..., z, 0, ..., 9, \langle\ \rangle, \langle , \rangle, \langle . \rangle, \langle ' \rangle, \langle unk \rangle\}$. The tokens $\langle sos \rangle$ and $\langle eos \rangle$ represent the start and end of a sentence respectively. The LAS model represents a conditional distribution over previous characters y_j where $j < i$ and the

input signal x for each output character y_i. Using the chaing rule, we get:

$$P(y|x) = \prod_i P(y_i|x, y_j) \tag{4}$$

This model is composed of two modules: the listener and the speller. The listener $Listen$ represents an acoustic model encoder, while the speller $AttendAndSpell$ is an attention-based character decoder. The first module converts the input signal x into a higher-level feature representation h, whereas the speller module processes h as input and computes a probability distribution over a sequence of characters. A representation of the LAS model with these two modules is given in Fig. 4.

$$h = Listen(x) \tag{5}$$
$$P(y|x) = AttendAndSpell(h, y) \tag{6}$$

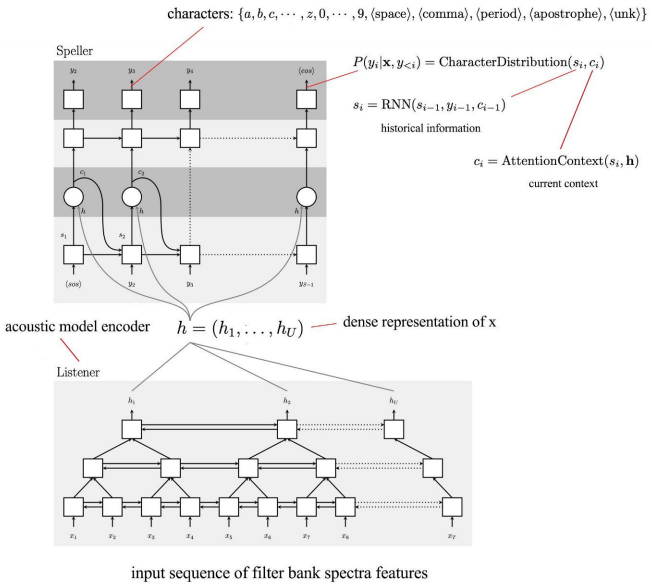

Figure 4: Representation of Listen, Attend and Spell (LAS) model. [13] The listener, a pyramidal BLSTM, encodes the input signal x to a higher level feature representation h, while the speller decodes the output y from this high level representation h.

4.2.2.2 Sequence to sequence learning

Despite deep neural networks being successfully used in various classification applications which categorise fixed input vectors to output classes, to process variable-length sequences, neural networks are combined with sequential models such as Hidden Markov Models (HMMs). These combined models result in a statistical model instead of an end-to-end neural network model. The disadvantage is that statistical models are difficult to be trained end-to-end. [11] The encoder-decoder architecture copes with this drawback. Models based on this architecture can process input and output sequences of variable lengths. This architecture design is built on two main components. The first is called the encoder which transforms a variable-length sequence into a fixed shape representation. The other component is the decoder and it maps this fixed representation to a variable-length sequence. [12] An illustration of this architecture is given in Fig. 5.

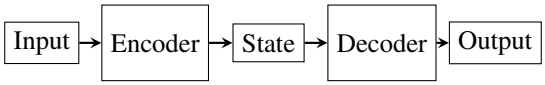

Figure 5: The encoder-decocer architecture.

Sequence to sequence learning is a process based on the encoder-decoder architecture. During training, the model provides the decoder with the labels, while during inference, the model executes a beam search to obtain a relevant output for later predictions.

These models are improved when combined with an attention mechanism. It supplies the decoder more information when producing the output. This mechanism propagates information more effectively from the encoder to the decoder at each time step.

This learning framework has been applied to many domains, such as machine translation, image captioning, etc. This learning process is not limited to a particular domain and it could be applied to build ASR systems.

4.2.2.3 Bidirectional RNN

Bidirectional RNNs stack an additional hidden layer on top of an existing RNN which runs forward from the first token. This stacked hidden layer passes information backwards from the last token to the front. This mechanism in RNNs reproduces a look-ahead ability similar to the one found in HMMs. [12] An illustration of a bidirectional RNN's architecture with a single hidden layer is given in Fig. 6.

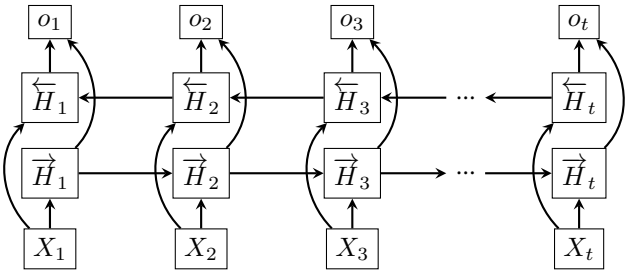

Figure 6: Architecture of a bidirectional RNN with a single hidden layer.

For each time step t, let a minibatch input $\boldsymbol{X_t} \in \mathbb{R}^{n \times d}$, where n is the number of examples and d is the number of inputs for each example. We define ϕ as activation funciton of the hidden layer. In the bidirectional design, it is assumed that at time step t the forward and backward hidden states are $\overrightarrow{\boldsymbol{H}}_t \in \mathbb{R}^{n \times h}$ and $\overleftarrow{\boldsymbol{H}}_t \in \mathbb{R}^{n \times h}$, respectively, where h

represents the number of hidden units. The update equations for the forward and backward hidden states are defined as follows:

$$\overrightarrow{\boldsymbol{H}}_t = \phi(X_t W_{xh}^{(f)} + \overrightarrow{H}_{t-1} W_{hh}^{(f)} + b_h^{(f)}) \tag{7}$$

$$\overleftarrow{\boldsymbol{H}}_t = \phi(X_t W_{xh}^{(b)} + \overleftarrow{H}_{t+1} W_{hh}^{(b)} + b_h^{(b)}) \tag{8}$$

where the weights $W_{xh}^{(f)} \in \mathbb{R}^{d \times h}$, $W_{hh}^{(f)} \in \mathbb{R}^{h \times h}$, $W_{xh}^{(b)} \in \mathbb{R}^{d \times h}$ and $W_{hh}^{(b)} \in \mathbb{R}^{h \times h}$, and biases $b_h^{(f)} \in \mathbb{R}^{1 \times h}$ and $b_h^{(b)} \in \mathbb{R}^{1 \times h}$ represent the model's parameters.

To compute the hidden state $\boldsymbol{H}_t \in \mathbb{R}^{n \times 2h}$ which will be the input of the output layer, the forward and backward hidden states $\overrightarrow{\boldsymbol{H}}_t$ and $\overleftarrow{\boldsymbol{H}}_t$ are concatenated. This hidden state $\boldsymbol{H}_t$ is passed as input to the next bidirectional layer in a deep bidirectional RNN with different hidden layers. The last layer of a bidirectional RNN computes the output $\boldsymbol{O}_t$:

$$\boldsymbol{O}_t = \boldsymbol{H}_t W_{hq} + b_q \tag{9}$$

where $W_{hq} \in \mathbb{R}^{2h \times q}$ is the weight matrix, $b_q \in \mathbb{R}^{1 \times q}$ is the bias vector and q is the number of outputs.

4.2.2.4 Listen

The $Listen$ operation is defined by a Bidirectional LSTM RNN (BLSTM) which is structured into a pyramid. This pyramidal structure reduces the length of input x from T to the length U of h. Applying the BLSTM directly to the $Listen$ operation will converge slowly and performs poorly since the $AttendAndSpell$ operation has difficulties to extract useful information from larger input sequences. [11] To mitigate this issue, a pyramid BLSTM (pBLSTM) is used. At each stacked pBLSTM layer, the time resolution of the input is decreased by a factor of 2. In normal deep BLSTM RNNs, the output at the i-th time step and j-th layer is computed as follows:

$$h_i^j = BLSTM(h_{i-1}^j, h_i^{j-1}) \tag{10}$$

Compared to the BLSTM architecture, the pBLSTM model concatenates the outputs of consecutive steps of each layer and passes it to the next layer:

$$h_i^j = pBLSTM(h_{i-1}^j, [h_{2i}^{j-1}, h_{2i+1}^{j-1}]) \tag{11}$$

The LAS model stacks 3 pBLSTMs above the BLSTM input layer. This decreases the time resolution $2^3 = 8$ times, allowing the attention model to derive important information from an input sequence with fewer time steps. An illustration of the $Listen$ operation is given in Fig. 4

4.2.2.5 Attend and Spell

The $AttendAndSpell$ operation uses an attention-based LSTM transducer to compute at every output step a probability distribution over the next character based on all the characters seen before. The context vector c_i is generated by an attentation mechanism. The decoder state s_i is computed from the previous state s_{i-1}, the previously produced character y_{i-1} and the context c_{i-1}. Further, the probability distribution y_i is a function of the decoder state s_i and the context c_i. Respectively,

$$c_i = AttentionContext(s_i, h) \tag{12}$$

$$s_i = RNN(s_{i-1}, y_{i-1}, c_{i-1}) \tag{13}$$

$$P(y_i|x, y_{<i}) = CharacterDistribution(s_i, c_i) \tag{14}$$

where $CharacterDistribution$ represents an MLP which character outputs are activated by a softmax function and RNN is equal to a 2 layer LSTM network.

At each timestep i of the decoder, the $AttentionContext$ function generates for each timestep u a scalar energy $e_{i,u}$ from $h_u \in h$ and s_i. The scalar energy $e_{i,u}$ is transformed into a probability distribution $\alpha_{i,u}$ over timesteps applying a softmax function. This distribution is used to create the context vector c_i when linearly blending it with the listener features h_u at different timesteps:

$$e_{i,u} = \langle \phi(s_i), \psi(h_u) \rangle \tag{15}$$

$$\alpha_{i,u} = \frac{exp(e_{i,u})}{\sum_u exp(e_{i,u})} \tag{16}$$

$$c_i = \sum_u \alpha_{i,u} h_u \tag{17}$$

where ϕ and ψ are MLP networks and c_i can be interpreted as a collection of weighted features of h. Fig. 4 illustrates the operation $AttendAndSpell$.

4.2.2.6 Learning

The $Listen$ and $AttendAndSpell$ operations are trained jointly for end-to-end speech recognition. This sequence to sequence model conditions the next step prediction on the previously emitted characters and tries to maximize the log probability:

$$\max_\theta \sum_i logP(y_i|x, y^*_{<i}; \theta) \tag{18}$$

During inference, the expected results are not available. Thus the predictions of the model can deteriorate since it was not trained to be resistant against bad prediction inputs at any time steps. To deal with this during training, we sample from the previous character distribution and use it as input for the next step prediction instead of always feeding the ground truth label:

$$\tilde{y_i} \sim CharacterDistribution(s_i, c_i) \tag{19}$$

$$\max_\theta \sum_i logP(y_i|x, \tilde{y}_{<i}; \theta) \tag{20}$$

where $\tilde{y}_{i-1}$ is the ground truth character or a character sampled from the model.

4.2.2.7 Decoding

During inference the LAS model finds the most probable character sequence given the acoustic input:

$$\hat{y} = arg\max_{y} logP(y|x) \tag{21}$$

A left-to-right beam search algorithm is used to perform decoding. A set of β incomplete hypothesis is initialized each starting with the start-of-sentence $\langle sos \rangle$ token. Each hypothesis in the beam is expanded at each time step with every possible character. Only the β most probable beams are retained. When reaching the end-of-sentence $\langle eos \rangle$ token, the candidate is removed from the beam and placed to the completed hypothesis. A word dictionary can be added to the decoding phase. However, as observed from the experiments in section 4.4.1, this is not necessary because the LAS model learns to spell real words most of the time.

4.3. Production

4.3.1. FR02: Mel Filterbank feature extraction in Python

We use the Pytorch module *Torchaudio* [14] to extract the Mel Filterbanks from the dataset. Instead of processing the whole dataset, the following steps are applied to one single audio file f:

1. Let f be an audio file sampled at $16kHz$, *Torchaudio* loads the audio file as follows:

```
import torchaudio
import torchaudio.compliance.kaldi.fbank as
    fbank

waveform, sr = torchaudio.load(filepath)
```

   with $waveform$ being the audio time series and sr the sample rate of $waveform$.

2. The Mel Filterbanks can be extracted with:

```
filterbank = fbank(waveform, num_mel_bins=40,
    channel=-1, sample_frequency=sample_rate)
```

   The function *fbank()* returns a *torch.Tensor* with the dimension:

$$shape = (padded\_window\_size//2 + 1,\ m)$$

   with $padded\_window\_size//2 + 1$ being the number of frames and m the number of Mel-filters.
3. The last step is to flatten this matrix to a vector which can be stored in a data frame for later training.

```
row = filterbank.transpose(0, 1).unsqueeze(0)
```

   After this operation, the matrix $filterbank$ is transposed and flattened. This concatenates the columns of $filterbank$ together forming a vector.

4.4. Assessment

4.4.1. NFR01: Performance evaluation and comparison

In this section, the performance of the LAS architecture is evaluated. For this study, we use 460h of the roughly 1000h LibriSpeech audio corpus which contains read English speech. It contains 132553 examples of varying lengths, sampled at $16kHz$ and saved as FLAC formatted files. This dataset represents 772 speakers reading the different sentences. The data is gathered from read audiobooks from the LibriVox project. Additionally, these readings were segmented and aligned into example sentences. [6]

First of all, we define a couple of configurations which set up the hyperparameters of the LAS model to perform different experiments:

- Non-pyramidal encoder
- SpecAgument LAS architecture

4.4.2. LAS model with non-pyramidal encoder

In this experiment, we train multiple LAS models without the encoder having a pyramidal form. The hyperparameters for this model look as follows:

```
hparas:
  valid_step: 5000
  max_step: 1000001
  optimizer: 'Adadelta'
  lr: 1.0

model:

  encoder:
    module: 'LSTM'
    bidirection: True
    dim: [512,512,512,512,512]
    dropout: [0,0,0,0,0]

  attention:
    mode: 'loc'
    dim: 300

  decoder:
    module: 'LSTM'
    dim: 512
    layer: 1
    dropout: 0
```

The model's encoder is configured with 5 BLSTM layers each composed of 512 units. No dropout is applied to the output of the encoder. The attention mechanism is based on location awareness and contains 300 units. [15] Further, the decoder includes 1 LSTM layer of 512 units. The model uses the *Adadelta* optimizer which is a stochastic gradient descent method. It is set up to train until 1000001 *steps* and validates the model at each 5000 *steps* intervals. However, the training of the model stops before reaching 1000001 *steps* since the runtime on the HPC is restricted to 2 days.

For this configuration, we investigate 4 different experiments:

- Training on 100h of the LibriSpeech corpus
  - Predicting characters
  - Predicting characters constrained with a word dictionary
- Training on 460h of the LibriSpeech corpus
  - Predicting characters
  - Predicting characters constrained with a word dictionary

4.4.2.1 100h: Predicting characters

In this experiment, we train the model on 100h of the LibriSpeech corpus to learn predicting characters from the acoustic features without a word dictionary. The training visualization is illustrated in Fig. 7. The model was able to reduce the loss during training and the loss converges to zero. The Word Error Rate (WER) metric is used as a performance measure. The WER is derived from the Levenshtein distance which computes the difference between two string sequences. During training, the model achieves a WER close to 0%, whereas, during validation the LAS model reaches a 24% WER.

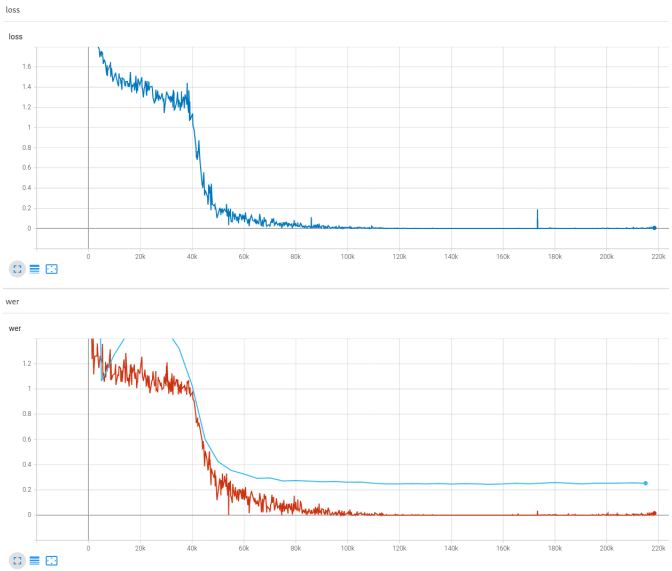

Figure 7: Training on 100h of the LibriSpeech corpus. The upper plot shows the loss of the training. The lower plot shows the training WER in red and the validation WER in blue.

When evaluating the model on the testing dataset from the LibriSpeech corpus, which contains 2,620 examples, it achieves a WER of 24% as seen in Table 1.

Error Rate (%)	Mean	σ
Character	10.2	9.3
Word	24	18.8

TABLE 1: Evaluation of the model on the testing dataset *test_clean*.

4.4.2.2 100h: Predicting characters with a word dictionary

The same configured model was trained on 100h of training data, but it uses a word dictionary to constrain the search space to valid words when decoding. The training visualization is illustrated in Fig. 8. The model achieves a validation WER of 25%. When evaluating the model on the same testing dataset, it achieves a testing WER of 25.1% as seen in Table 2. The result shows that a word dictionary is not necessary when decoding the output of the model since the model is able to learn spelling real words correctly most of the time.

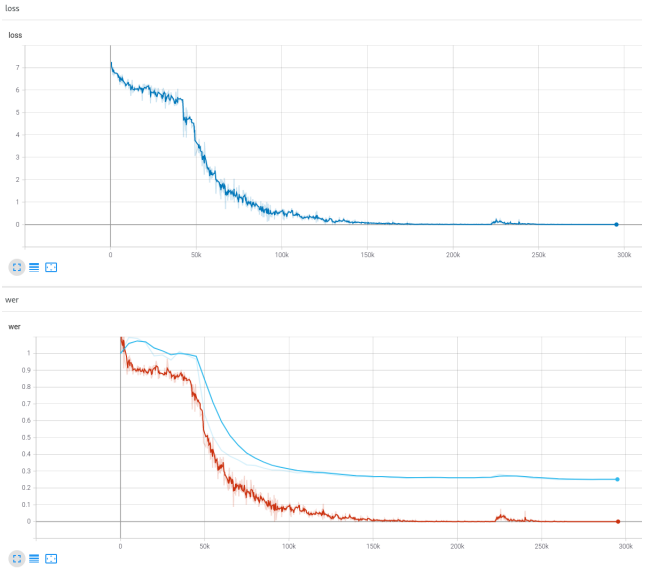

Figure 8: Training on 100h of the LibriSpeech corpus with a dictionary. The upper plot shows the loss of the training. The lower plot shows the training WER in red and the validation WER in blue.

Error Rate (%)	Mean	σ
Character	13.9	18.9
Word	25.1	23.6

TABLE 2: Evaluation of the model on the testing dataset *test_clean*.

4.4.2.3 460h: Predicting characters

This time, the model was trained on 460h of the LibriSpeech corpus. The training visualization is illustrated in Fig. 9. The model achieves a validation WER of 13%. When evaluating the model on the testing dataset, it achieves a testing WER of 14.1% as seen in Table 3.

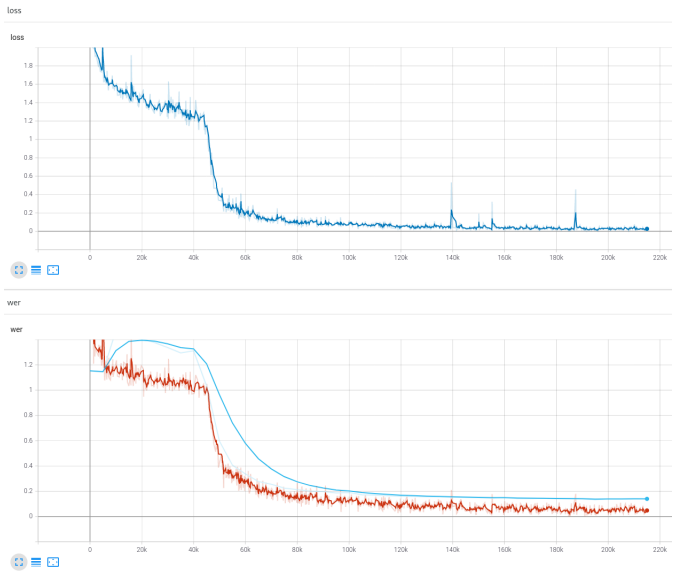

Figure 9: Training on 460h of the LibriSpeech corpus. The upper plot shows the loss of the training. The lower plot shows the training WER in red and the validation WER in blue.

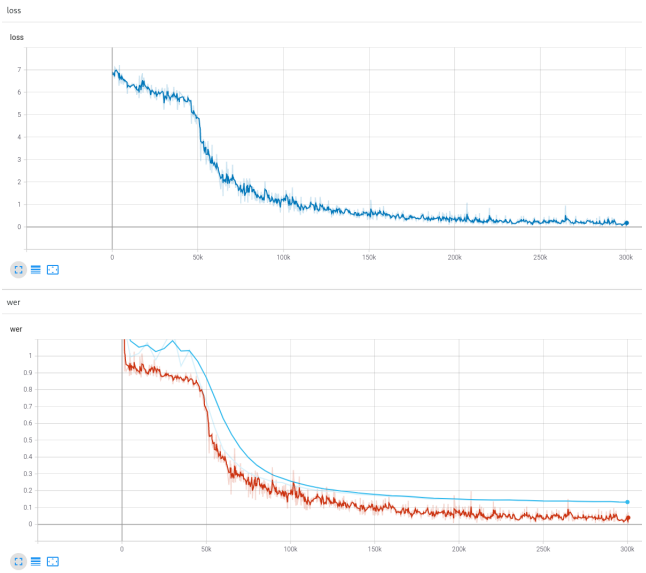

Figure 10: Training on 460h of the LibriSpeech corpus with a dictionary. The upper plot shows the loss of the training. The lower plot shows the training WER in red and the validation WER in blue.

Error Rate (%)	Mean	σ
Character	5.8	7.3
Word	14.1	15.3

TABLE 3: Evaluation of the model on the testing dataset *test_clean*.

4.4.2.4 460h: Predicting characters with a word dictionary

In this experiment, the model was trained on 460h of training data as well but constrains the decoding of the output with a word dictionary. The training visualization is illustrated in Fig. 10. The model achieves a validation WER of 13%. When evaluating the model on the testing dataset, it achieves a testing WER of 14.9% as seen in Table 4. The result shows as well that a word dictionary is not necessary when decoding the output of the model.

Error Rate (%)	Mean	σ
Character	7.7	34.3
Word	14.9	40.6

TABLE 4: Evaluation of the model on the testing dataset *test_clean*.

4.4.3. SpecAugment LAS architecture

In this experiment, we will train a LAS model with the SpecAugment architecture. [16] The hyperparameters for this model look as follows:

```
model:

  encoder:
    module: 'LSTM'
    bidirection: True
    dim: [1024,1024,1024,1024]
    dropout: [0,0,0,0]

  attention:
    mode: 'loc'
    dim: 300

  decoder:
    module: 'LSTM'
    dim: 1024
    layer: 1
    dropout: 0
```

The model has almost the same structure as in the previous experiment. However, the encoder is configured with 4 BLSTM layers instead of 5, each composed of 1024 units. In this experiment, the model was trained on 460h of training data without using a dictionary for decoding. The training visualization is illustrated in Fig. 11. The model achieves a validation WER of 15%. When evaluating the model on the testing dataset, it achieves a testing WER of 15.5% as seen in Table 5. The result shows that this architecture did not improve the performance of the model compared to the previous architecture.

Error Rate (%)	Mean	σ
Character	6.8	8.3
Word	15.5	16.7

TABLE 5: Evaluation of the model on the testing dataset *test_clean*.

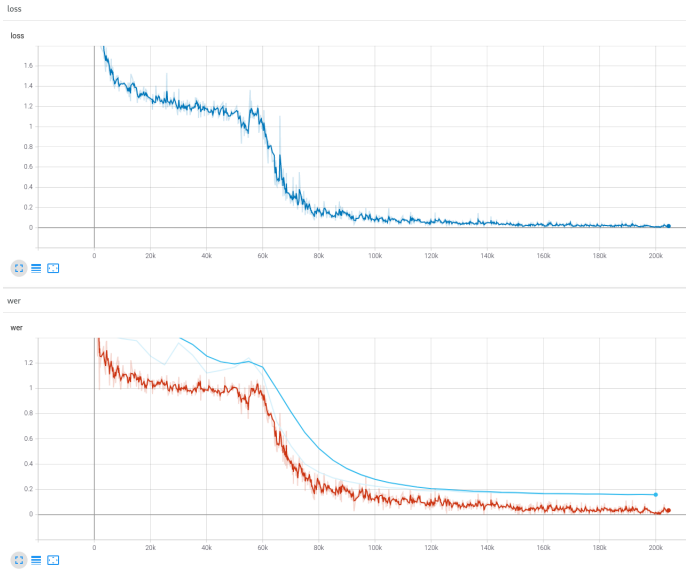

Figure 11: Training on 460h of the LibriSpeech corpus with the SpecAugment architecture. The upper plot shows the loss of the training. The lower plot shows the training WER in red and the validation WER in blue.

4.4.4. Beam Width

In this section, we will test different beam widths β during decoding of our best candidate found in section 4.4.2.3.

Beam width β	WER (%)
2	14.1
8	13.79
16	13.80

TABLE 6: Evaluation of the model trained on 460h training data with different beam width β on the testing dataset *test_clean*.

Interpreting the results in Table 6, we deduce that a higher beam width β decreases the average WER of the LAS model.

4.4.5. Attention Visualization

The attention alignment of the audio signal of a validation utterance is visualized in Fig. 12. The model was able to identify the start and the end of the utterance, although it outputs the phrase *"a man was looking in from the quarter behind at the four persons we were just discussing"* instead of *"a man was looking in from the corridor behind at the four persons we were just discussing"*. There is only one word which is spelt wrongly.

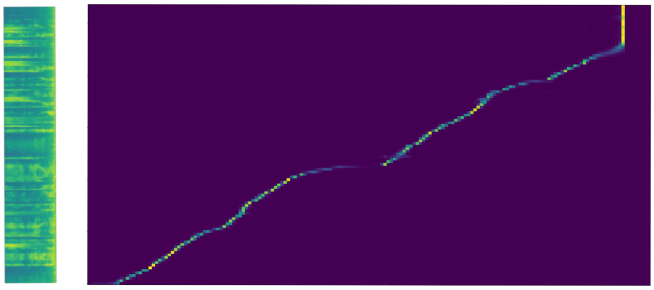

Figure 12: Audio signal and the attention alignment produced for the utterance *"a man was looking in from the corridor behind at the four persons we were just discussing"*.

4.4.6. Performance comparison

Looking at the results of other speech recognition models in Table 7, our experiments with the LAS model didn't perform as well as other reported in the literature. The reason for this might be the fact that the model was trained only on the subset of LibriSpeech dataset.

Model	WER (%)
Conformer + Wav2vec 2.0 + SpecAugment-based Noisy Student Training with Libri-Light	1.4
LAS	2.7
Our LAS experiment	13.8

TABLE 7: Comparison of Speech Recognition models on LibriSpeech *test-clean* dataset.

5. Technical Deliverables

The technological aspect of this project is the reuse and adaptation of an existing implementation of an attention-based speech recognition model. Additionally, the LibriSpeech dataset is preprocessed and used for the model's training. The goal is to deploy its training in an HPC environment for efficient computing.

5.1. Requirements

- **FR01** Presentation of an existing LAS implementation.
- **FR02** Showcase usage of LAS implementation and training in an HPC environment.
- **FR03** Introduce a feature extraction program for the LibriSpeech dataset.

5.2. Design

5.2.1. FR01: Presentation of an existing LAS implementation.

In this section, the PyTorch implementation of the LAS end-to-end [17] ASR system is presented. The main entry of this implementation is the *main.py* program:

```
import yaml
import torch
import argparse

parser = argparse.ArgumentParser(description='
    Training E2E asr.')
parser.add_argument('--config', type=str, help='
    Path to experiment config.')
paras = parser.parse_args()
config = yaml.load(open(paras.config, 'r'), Loader
    =yaml.FullLoader)

if paras.test:
    # Test ASR
    assert paras.load is None, 'Load option is
    mutually exclusive to --test'
    from bin.test_asr import Solver
    mode = 'test'
else:
    # Train ASR
    from bin.train_asr import Solver
    mode = 'train'

solver = Solver(config, paras, mode)
solver.load_data()
solver.set_model()
solver.exec()
```

The *main.py* imports the class *Solver* from *bin.train_asr* while being in the training mode. An instance of the Solver class is created with the hyperparameters and training mode as arguments (line 20). This object loads the training and validation set with the *load_data()* method (line 21). The object initialises a model with the defined hyperparameters with the *set_model()* method (line 22) and executes its training with the method *exec()* (line 23).

In the following listing, the method *load_data()* is shown:

```
def load_data(self):
    self.tr_set, self.dv_set, self.feat_dim, self.
    vocab_size, self.tokenizer, msg = \
        load_dataset(**self.config['data'])
```

The function *load_dataset()* uses PyTorch's DataLoader interface which is an utility to customize loading of datasets (line 3). This function parses the LibriSpeech dataset, extracts the Mel-filterbanks and outputs the training and the validation sets divided into batches.

The following listing showcases the *set_model()* method, which initialises the *Adadelta* optimizer, creates an *ASR()* instance for the *self.model* attribute and assigns the *CrossEntropyLoss* as the loss function.

```
def set_model(self):
    ''' Setup ASR model and optimizer '''
    #model
    init_adadelta = self.config['hparas']['
    optimizer'] == 'Adadelta'
    self.model = ASR(self.feat_dim, self.
    vocab_size, init_adadelta, **
                     self.config['model']).to(self
    .device)

    # Losses
    self.seq_loss = torch.nn.CrossEntropyLoss(
    ignore_index=0)
```

The *ASR()* class is defined as follows:

```
class ASR(nn.Module):
    ''' ASR model, including Encoder/Decoder(s)'''

    def __init__(self, input_size, vocab_size,
    init_adadelta, ctc_weight, encoder, attention,
     decoder, emb_drop=0.0):
        super(ASR, self).__init__()

        # Modules
        self.encoder = Encoder(input_size, **
    encoder)
        self.dec_dim = decoder['dim']
        self.pre_embed = nn.Embedding(vocab_size,
    self.dec_dim)
        self.embed_drop = nn.Dropout(emb_drop)
        self.decoder = Decoder(
            self.encoder.out_dim+self.dec_dim,
    vocab_size, **decoder)
        query_dim = self.dec_dim*self.decoder.
    layer
        self.attention = Attention(
            self.encoder.out_dim, query_dim, **
    attention)
```

When initializing the *ASR()* class, it creates an encoder layer, an Embedding layer for the vocabulary, a dropout layer, a decoder layer and an attention layer.

The *exec()* method is defined as follows:

```
def exec(self):
    ''' Training End-to-end ASR system '''
    n_epochs = 0

    while self.step < self.max_step:
        for data in self.tr_set:
            # Fetch data
            feat, feat_len, txt, txt_len = self.
    fetch_data(data)

            # Forward model
            ctc_output, encode_len, att_output,
    att_align, dec_state = \
                self.model(feat, feat_len, max(
    txt_len), tf_rate=tf_rate,
                           teacher=txt,
    get_dec_state=self.emb_reg)

            # Backprop
            grad_norm = self.backward(total_loss)
            self.step += 1

            # Logger
            if (self.step == 1) or (self.step %
    self.PROGRESS_STEP == 0):
                self.log()
            # Validation
            if (self.step == 1) or (self.step %
    self.valid_step == 0):
                self.validate()

            if self.step > self.max_step:
                break
        n_epochs += 1
```

This function trains the model initialized in the previous steps. The model passes the input data forward through the network with the method *self.model()*, propagates the losses back with the method *self.backward(total_loss)* and

increments the number of steps by one. The function calls the methods *self.log()* and *self.validate()* to log the current training state and validate the model's performance respectively.

5.3. Production

5.3.1. FR02: Showcase usage of LAS implementation and training in an HPC environment.

The following script *job_launcher_script.sh*, requests the needed resources and executes the application when scheduled for a job on the HPC:

```
1  #!/bin/bash -l
2  #SBATCH -J las460fc
3  #SBATCH --mail-type=begin,end,fail
4  #SBATCH --mail-user=user_id@student.uni.lu
5  #SBATCH -N 1
6  #SBATCH -n 8
7  #SBATCH -G 1
8  #SBATCH --time=2-00:00:00
9  #SBATCH -p gpu
10 #SBATCH --qos=normal
11 #SBATCH --output='output/las-fixed-100-character%A
       .out'
12
13 echo "== Starting run at $(date)"
14 echo "== Job ID: ${SLURM_JOBID}"
15 echo "== Node list: ${SLURM_NODELIST}"
16 echo "== Submit dir. : ${SLURM_SUBMIT_DIR}"
17
18 source .venv/bin/activate
19 python3 main.py --config config/libri/
       las_460_fixed_character.yaml --logdir
       las_460_fixed_character_logdir --name
       las_460_fixed_character
```

This job launcher requests one computing node with the argument $-N$ 1. On this computing node, 8 CPU cores and 1 GPU are requested with the arguments $-n$ 8 and $-G$ 1 respectively. To use GPUs in the computing node, the launcher has to enable $-p$ *gpu*. On line 18, it activates the virtual environment so that the Python application can be executed.

To run the LAS model in training mode, the launcher script executes the following command:

```
1 $ python3 main.py --config $config_file --logdir
      $log_dir --name $model_name
```

where *$config_file* is the file defining the model's hyperparameters, *$log_dir* is the directory name for logging the tensorboard data and *$model_name* is the name under which a checkpoint of the model is made.

To submit our application to the queue for execution, we run the following command in the HPC user shell which schedules the SLURM job launcher:

```
1 $ sbatch job_launcher_script.sh
```

5.3.2. FR03: Introduce a feature extraction program for the LibriSpeech dataset.

In this section, we present program which extracts the Mel-filterbank features from the LibriSpeech dataset. The program executes the following steps:

1) process_flac2wav()

   This function parses the LibriSpeech file directories and converts all audio files, formatted as FLAC files, to WAV files.

2) process_wav2speech_feature()

   This function parses the LibriSpeech file directories and extracts the Mel-filterbanks from the WAV files.

3) process_speech_feature2dataset(train_file_list, train_path, 'train.CSV')

   This function creates a CSV file for the training dataset which contains examples with the file path to the extracted features and their label.

4) process_speech_feature2dataset(dev_file_list, dev_path, 'dev.CSV')

   This function creates a CSV file for the validation dataset which contains examples with the file path to the extracted features and their label.

5) process_speech_feature2dataset(test_file_list, test_path, 'test.CSV')

   This function creates a CSV file for the testing dataset which contains examples with the file path to the extracted features and their label.

6) save_vocab()

   This function saves the character vocabulary list to a CSV file.

A listing of this program is provided in the Appendix.

5.4. Assessment

All the technical requirements were accomplished. The implementation of the *Listen, Attend and Spell - PyTorch* and its usage were presented. Additionally, we explained our program for extracting the Mel-filterbanks from the LibriSpeech dataset.

6. Conclusion

In this Bachelor Semester Project report, the Listen, Attend and Spell architecture was presented. An existing implementation of the LAS model was reused to train it on

the LibriSpeech dataset. A few experiments with different sets of hyperparameters were carried out. Furthermore, the performances were assessed and compared to other state-of-the-art speech recognition models, as shown in Table 7. Finally, we presented the usage and training of the LAS implementation in an HPC environment as well as the program to extract the Mel-filterbanks from the speech dataset.

Acknowledgment

I would like to thank my tutor Vladimir Despotovic for his constructive feedback and mentorship. Additionally, I thank him for supervising my report and proofreading it. I would recommend fellow BiCS Students interested in Speech recognition to work with Vladimir Despotovic.

References

[1] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos, E. Elsen, J. H. Engel, L. Fan, C. Fougner, T. Han, A. Y. Hannun, B. Jun, P. LeGresley, L. Lin, S. Narang, A. Y. Ng, S. Ozair, R. Prenger, J. Raiman, S. Satheesh, D. Seetapun, S. Sengupta, Y. Wang, Z. Wang, C. Wang, B. Xiao, D. Yogatama, J. Zhan, and Z. Zhu, "Deep speech 2: End-to-end speech recognition in english and mandarin," *CoRR*, vol. abs/1512.02595, 2015. [Online]. Available: http://arxiv.org/abs/1512.02595

[2] W. S. J. Cai, "End-to-end deep neural network for automaticspeech recognition," 2015. [Online]. Available: https://cs224d.stanford.edu/reports/SongWilliam.pdf

[3] "Python software foundation, python language reference, version 3.7. available at," http://www.python.org/.

[4] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[5] "What is high-performance computing?" https://www.netapp.com/data-storage/high-performance-computing/what-is-hpc/, accessed: 12/12/20.

[6] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur, "Librispeech: an asr corpus based on public domain audio books," in *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*. IEEE, 2015, pp. 5206–5210.

[7] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[9] "General python faq," https://docs.python.org/3/faq/general.html#what-is-python, accessed 19/05/19.

[10] H. M. Fayek, "Speech processing for machine learning: Filter banks, mel-frequency cepstral coefficients (mfccs) and what's in-between," 2016, https://haythamfayek.com/2016/04/21/speech-processing-for-machine-learning.html.

[11] W. Chan, N. Jaitly, Q. V. Le, and O. Vinyals, "Listen, attend and spell," 2015.

[12] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, "Dive into deep learning," 2020, https://d2l.ai.

[13] "Speech recognition — deep speech, ctc, listen, attend, and spell," https://jonathan-hui.medium.com/speech-recognition-deep-speech-ctc-listen-attend-and-spell-d05e940e9ed1, accessed: 08/01/21.

[14] "torchaudio: an audio library for pytorch," https://github.com/pytorch/audio, accessed: 06/01/21.

[15] J. Chorowski, D. Bahdanau, D. Serdyuk, K. Cho, and Y. Bengio, "Attention-based models for speech recognition," 2015.

[16] D. S. Park, W. Chan, Y. Zhang, C.-C. Chiu, B. Zoph, E. D. Cubuk, and Q. V. Le, "Specaugment: A simple data augmentation method for automatic speech recognition," *Interspeech 2019*, Sep 2019. [Online]. Available: http://dx.doi.org/10.21437/Interspeech.2019-2680

[17] A. H. Liu, T.-W. Sung, S.-P. Chuang, H. yi Lee, and L. shan Lee, "Sequence-to-sequence automatic speech recognition with word embedding regularization and fused decoding," 2019.

7. Appendix

```
from tqdm import tqdm
from pydub import AudioSegment
from joblib import Parallel, delayed
from python_speech_features import logfbank, fbank, mfcc

import os
import argparse
import numpy as np
import pandas as pd
import scipy.io.wavfile as wav


def init_parser():
    parser = argparse.ArgumentParser(description='Librispeech preprocess.')

    parser.add_argument('root', metavar='root', type=str,
                        help='Absolute file path to LibriSpeech. (e.g. /usr/downloads/LibriSpeech/)')

    parser.add_argument('train_set', metavar='train_set', type=str,
                        help='Training datasets to process in LibriSpeech. (e.g. train-clean-100/)')

    parser.add_argument('--dev_set', metavar='dev_set', type=str,
                        help='Validation datasets to process in LibriSpeech. (e.g. dev-clean/)')

    parser.add_argument('--test_set', metavar='test_set', type=str,
                        help='Testing datasets to process in LibriSpeech. (e.g. test-clean/)')

    parser.add_argument('--n_jobs', dest='n_jobs', action='store', default=-2 ,
                        help='number of cpu availible for preprocessing.\n -1: use all cpu, -2: use all
    cpu but one')

    parser.add_argument('--n_filters', dest='n_filters', action='store', default=40,
                        help='Number of filters for fbank. (Default : 40)')

    parser.add_argument('--win_size', dest='win_size', action='store', default=0.025,
                        help='Window size during feature extraction (Default : 0.025 [25ms])')

    parser.add_argument('--norm_x', dest='norm_x', action='store', default=False,
                        help='Normalize features s.t. mean = 0 std = 1')

    parser.add_argument('--speech_feature', dest='speech_feature', action='store', default='fbank',
                        help='Speech feature for feature extration (Default : fbank)')
    return parser

def parse_labels(root, dataset_dir):
    labels = []
    dataset_path = os.path.join(root, dataset_dir)
    for speaker in sorted(os.listdir(dataset_path)):
        for chapter in sorted(os.listdir(os.path.join(dataset_path, speaker))):
            text_filepath = os.path.join(dataset_path, speaker, chapter)
            text_filename = '{speaker}-{chapter}.trans.txt'.format(speaker=speaker, chapter=chapter)
            with open(os.path.join(text_filepath, text_filename), 'r') as f:
                lines = f.read().splitlines()
                labels.extend([' '.join(line.split(' ')[1:]).lower() for line in lines])
    return labels

def parse_audio(root, dataset_dir, search_filetype='.flac'):
    audio_files = []
    dataset_path = os.path.join(root, dataset_dir)
    for speaker in sorted(os.listdir(dataset_path)):
        for chapter in sorted(os.listdir(os.path.join(dataset_path, speaker))):
            for audio_file in sorted(os.listdir(os.path.join(dataset_path, speaker, chapter))):
                if audio_file.endswith(search_filetype):
                    audio_files.append(os.path.join(dataset_path, speaker, chapter, audio_file))
    return audio_files

def flac2wav(f_path):
    flac_audio = AudioSegment.from_file(f_path, "flac")
    flac_audio.export(f_path[:-4] + 'wav', format="wav")
```

```
69
70 def wav2fbank(f_path):
71     (rate, sig) = wav.read(f_path)
72     fbank_feat = fbank(sig, rate, winlen=win_size, nfilt=n_filters)
73     filename = f_path[:-4] + '-fbank' + str(n_filters)
74     np.save(filename, fbank_feat)
75     return filename + '.npy'
76
77 def wav2mfcc(f_path):
78     (rate, sig) = wav.read(f_path)
79     fbank_feat = mfcc(sig, rate, winlen=win_size, nfilt=n_filters)
80     filename = f_path[:-4] + '-mfcc' + str(n_filters)
81     np.save(filename, fbank_feat)
82     return filename + '.npy'
83
84 def norm(f_path, mean, std):
85     np.save(f_path, (np.load(f_path)-mean)/std)
86
87 def process_flac2wav():
88     global train_file_list, dev_file_list, test_file_list
89
90     print('Processing flac2wav')
91
92     print('Training')
93     train_file_list = parse_audio(root, train_path)
94     _ = Parallel(n_jobs=n_jobs,backend="threading")(delayed(flac2wav)(f) for f in tqdm(train_file_list))
95
96     print('Validation')
97     dev_file_list = parse_audio(root, dev_path)
98     _ = Parallel(n_jobs=n_jobs,backend="threading")(delayed(flac2wav)(f) for f in tqdm(dev_file_list))
99
100    print('Testing')
101    test_file_list = parse_audio(root, test_path)
102    _ = Parallel(n_jobs=n_jobs,backend="threading")(delayed(flac2wav)(f) for f in tqdm(test_file_list))
103
104 def process_wav2speech_feature(speech_feature='fbank'):
105     global train_file_list, dev_file_list, test_file_list
106
107     speech_features = {
108         'fbank': wav2fbank,
109         'mfcc': wav2mfcc,
110     }
111
112     wav2feature = speech_features.get(speech_feature)
113     print('Processing wav2{speech_feature}'.format(speech_feature=speech_feature))
114     print('Training')
115     train_file_list = Parallel(n_jobs=n_jobs,backend="threading")(delayed(wav2feature)(f[:-4] + 'wav')
        for f in tqdm(train_file_list))
116     print('Validation')
117     dev_file_list = Parallel(n_jobs=n_jobs,backend="threading")(delayed(wav2feature)(f[:-4] + 'wav') for
         f in tqdm(dev_file_list))
118     print('Testing')
119     test_file_list = Parallel(n_jobs=n_jobs,backend="threading")(delayed(wav2feature)(f[:-4] + 'wav')
        for f in tqdm(test_file_list))
120
121 def process_speech_feature2dataset(dataset_file_list, dataset_path, output_filename):
122     global mean_x, std_x
123     # dataset_file_list = parse_audio(root, dataset_path, search_filetype='.npy')
124     dataset_text = parse_labels(root, dataset_path)
125
126     X = []
127     for f in dataset_file_list:
128         X.append(np.load(f))
129
130     # Normalize X
131     if norm_x:
132         if mean_x == None and std_x == None:
133             mean_x = np.mean(np.concatenate(X, axis=0), axis=0)
134             std_x = np.std(np.concatenate(X, axis=0), axis=0)
135         _ = Parallel(n_jobs=n_jobs,backend="threading")(delayed(norm)(f, mean_x, std_x) for f in tqdm(
    data_file_list))
```

```
    # Sort data by signal length (long to short)
    audio_len = [len(x) for x in X]
    dataset_file_list = [dataset_file_list[idx] for idx in reversed(np.argsort(audio_len))]
    dataset_text = [dataset_text[idx] for idx in reversed(np.argsort(audio_len))]

    # text to index sequence
    tmp_list = []
    for text in dataset_text:
        build_vocab(text)
        tmp = [str(VOCAB[char]) for char in text]
        tmp_list.append(tmp)
    dataset_text = tmp_list
    del tmp_list

    output_filepath = os.path.join(root, output_filename)

    print('Writing dataset to {output_filepath}'.format(output_filepath=output_filepath))

    dataset_text = [' '.join(sentence) for sentence in dataset_text]

    dataset = {
        'input': dataset_file_list,
        'label': dataset_text
    }

    df = pd.DataFrame(dataset)
    df.to_csv(output_filepath, index=False)

def build_vocab(text):
    global VOCAB, IVOCAB
    for token in text:
        for char in token:
            if char not in VOCAB:
                next_index = len(VOCAB)
                VOCAB[char] = next_index
                IVOCAB[next_index] = char

def save_vocab():
    global IVOCAB
    IVOCAB = pd.DataFrame(list(IVOCAB.items()), columns=['key', 'value'])
    IVOCAB.to_csv(os.path.join(root, 'VOCAB.csv'), index=False)

if __name__ == '__main__':

    paras = init_parser().parse_args()

    root = paras.root
    train_path = paras.train_set
    dev_path = paras.dev_set
    test_path = paras.test_set
    n_jobs = paras.n_jobs
    n_filters = paras.n_filters
    win_size = paras.win_size
    norm_x = paras.norm_x
    speech_feature = paras.speech_feature

    VOCAB = {'<PAD>': 0, '<SOS>': 1, '<EOS>': 2}
    IVOCAB = {0: '<PAD>', 1: '<SOS>', 2: '<EOS>'}
    mean_x = None
    std_x = None

    print('----------Processing Datasets----------')

    print('Training sets :', train_path)
    print('Validation sets :', dev_path)
    print('Testing sets :', test_path)

    print('---------------------------------------')

    process_flac2wav()
```

```
207
208     print('---------------------------------------')
209
210     process_wav2speech_feature()
211
212     print('---------------------------------------')
213
214     print('Preparing Training Dataset')
215
216     process_speech_feature2dataset(train_file_list, train_path, 'train.csv')
217
218     print('Preparing Validation Dataset')
219
220     process_speech_feature2dataset(dev_file_list, dev_path, 'dev.csv')
221
222     print('Preparing Testing Dataset')
223
224     process_speech_feature2dataset(test_file_list, test_path, 'test.csv')
225
226     print('Saving VOCAB')
227
228     save_vocab()
```