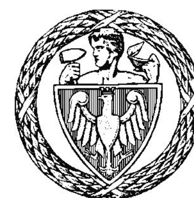
Warsaw University of Technology

FACULTY OF
MATHEMATICS AND INFORMATION SCIENCE



Bachelor's diploma thesis

in the field of study Computer Science and Information Systems

Efficient Minimum Word Error Rate Training for Attention-Based Models

Frąc Jakub

student record book number 298795

Kurdek Jeremi

student record book number 298801

thesis supervisor

dr hab. inż. Agnieszka Jastrzębska

WARSAW 2022

Abstract

Efficient Minimum Word Error Rate Training for Attention-Based Models

The task of speech transduction is converting speech from the audio domain into a text representation. With recent technological advancements, it became possible to apply highly computationally complex Deep Learning models to handle the task of speech transduction. The said solutions operate on the assumption that given enough training examples, the model is able to learn the correct mapping itself. One of the dominant Deep Learning architectures used for this purpose is the attention-based Recurrent Neural Network Transducer (RNN-T).

RNN-T model is typically trained to optimize a cross-entropy criterion corresponding to improving the log-likelihood of the data. However, the model is usually evaluated using a different metric, namely the Word Error Rate (WER). In this thesis, we present a TensorFlow implementation of an alternative training procedure allowing for direct optimization of the latter metric. Moreover, we incorporate in the solution a modified version of RNN-T Loss, called Monotonic RNN-T Loss, which is used to regularize the novel training procedure. The Monotonic RNN-T loss, as opposed to standard RNN-T, enforces strictly monotonic alignments between the input and output sequences allowing for faster decoding, crucial for MWER training.

The modified training routine is fully integrated with an open-source Automatic Speech Recognition toolkit TensorFlowASR, which allows for end-to-end MWER training. Specifically, a user is able to train a speech transduction solution on a custom dataset using a training procedure that directly optimizes the WER metric with minimal configuration.

The solution was experimentally evaluated on the public domain dataset LibriSpeech and the results, measured as relative improvement of WER, were found to be consistent with the outcomes reported by the research community.
Keywords: sequence-to-sequence models, attention models, minimum word error rate training, recurrent neural network transducer

Streszczenie

Efektywne uczenie w zakresie minimalnego współczynnika błędów słownych w modelach opartych na uwadze

Transdukcja mowy polega na przekształceniu mowy z dziedziny audio na reprezentację tekstową. Dzięki ostatniemu postępowi technologicznemu możliwe stało się zastosowanie bardzo złożonych obliczeniowo modeli głębokiego uczenia (ang. Deep Learning) do zadania przetwarzania mowy. Jedną z dominujących architektur głębokiego uczenia wykorzystywanych w tym celu jest Recurrent Neural Network Transducer (RNN-T) oparty o mechanizm uwagi.

Model RNN-T jest zwykle trenowany w celu optymalizacji kryterium entropii krzyżowej odpowiadającego poprawie log-prawdopodobieństwa danych. Jednakże model ten jest zwykle oceniany przy użyciu innej metryki, a mianowicie współczynnika błędu słów (ang. Word Error Rate, WER). W niniejszej pracy przedstawiamy implementację alternatywnej procedury treningowej w TensorFlow, pozwalającej na bezpośrednią optymalizację tej ostatniej metryki. Ponadto, w rozwiązaniu zastosowano zmodyfikowaną wersję straty RNN-T, zwaną Monotoniczną stratą RNN-T, która służy do regularyzacji nowej procedury treningowej. Monotoniczna strata RNN-T, w przeciwieństwie do standardowej RNN-T, wymusza ściśle monotoniczne dopasowanie sekwencji wejściowych i wyjściowych, co pozwala na szybsze dekodowanie, kluczowe dla treningu MWER.

Zmodyfikowana procedura treningowa jest w pełni zintegrowana z otwartym zestawem narzędzi do automatycznego rozpoznawania mowy TensorFlowASR, co pozwala przeprowadzić end-to-end trening za pomocą MWER. W szczególności, użytkownik jest w stanie trenować rozwiązanie do trandukcji mowy na dowolnym zbiorze danych za pomocą procedury treningowej, która bezpośrednio optymalizuje metrykę WER przy minimalnej konfiguracji.

Rozwiązanie to zostało eksperymentalnie ocenione na zbiorze danych LibriSpeech należącym do domeny publicznej, a wyniki, mierzone jako względna poprawa wskaźnika WER, okazały się zgodne z rezultatami podawanymi przez środowisko naukowe.

Słowa kluczowe: modele sekwencyjne, modele oparte na mechanizmie uwagi, minimalna stopa błędu słow

Contents

Introduction **11**

1. Theoretical background **18**

1.1. Baseline RNN-T 18

1.1.1. Training 19

1.2. Monotonic RNN-T 21

1.2.1. Loss and training 21

1.2.2. Inference 22

1.3. MWER loss function 24

1.3.1. Training 24

2. Architecture **26**

2.1. TensorFlowASR 27

2.2. Module Description 27

2.2.1. Monotonic RNN-T 27

2.2.2. BeamSearch class 28

2.2.3. MWERLoss class 29

2.2.4. MWERConformer class 31

3. Tests **33**

3.1. Testing methods introduction 33

3.2. Module test summary 34

4. Experiments **36**

4.1. Environment 36

4.1.1. Software 36

4.1.2. Hardware 36

4.2. Setup 37

4.2.1. Data 37

4.2.2. Methodology 37

4.2.3. Technical setup 37

4.3. Training and validation loss . . . 38
4.3.1. Monotonic RNN-T . . . 38
4.3.2. MWER . . . 39
4.4. Word Error Rate score . . . 41
4.5. Examples . . . 42
5. Conclusion . . . 44
A. User Manual . . .
A.1. Installation instructions . . .
A.1.1. Manual installation . . .
A.1.2. Docker installation . . .
A.1.3. Training . . .
A.1.4. Test . . .
A.1.5. Demonstration . . .

Introduction

The content of this work is grounded on the recent developments in the field of Deep Learning Automatic Speech Recognition. In order to fully describe the aim of the project, we utilize field related terminology. Below the reader may find a concise introduction to the covered topics.

Automatic Speech Recognition is a subfield of computer science related to recognizing human speech and translating it into text. In this work we are particularly interested in the speech to text translation. That is given an audio sample of human voice we desire to output a text representation of the uttered sentence.

Deep Learning

Historically, many approaches were proposed for the purpose of speech transduction, the one we are focusing on is sequence transduction using machine learning, an algorithm that acquires its own knowledge by extracting patterns from raw data. A machine learning solution tries to learn a mapping from certain data representation to the output [2]. Sometimes the task is so complex that learning directly the mapping is impossible, then we may stack multiple machine learning solutions to tackle the problem, this approach is known as deep learning.

Deep learning is a machine learning method utilizing deep stacks of artificial neural networks. Each layer, a set of artificial neurons, can be thought of as a mapping transforming an element of its input space into an element of its output space. The stack of layers, referred to as machine learning model, is then nothing more than a composition of mappings. In the ASR case, we wish to create a model capable of transforming audio representation into text and train it to obtain optimal mapping.

In order to train the model, we use supervised learning. That is during each learning step a model is presented with an input and a true output (referred to as a label). The model then maps the input to the element of the output space and compares the result with the target output.

In order to measure how accurate the mapping is, a 'loss function' is used. The loss function represents the distance between the models prediction and the label. The goal of the training is

to minimise this distance, that is minimise the loss function with respect to the model weights. To achieve this a gradient descent technique is employed, at each learning step a gradient is computed and used to update the weights.

Speech to text transduction

Audio sequence transduction is a process of converting human speech into text. Specifically the task consists of transforming recorded audio from wave form into a sequence of text tokens.

The input audio is preprocessed into a sequence of input vectors, each vector contains information about a short window of audio. The output text tokes are word pieces from the language the output sequence belongs to. The goal of the deep learning solution is to a find a mapping between the elements of the input space and word pieces from the output space.

In order to train a transducer model, we would need to know the alignment between the input sequence and the output sequence, in other words each tiny audio piece would need to have its own label attached. This would be extremely costly, as audio can be sampled with very high rate. The desired effect is to be able to train a sequence transducer in an alignment agnostic way, this way the only necessary label for an audio piece is a transcription of uttered speech. In order to allow for such training, the Recurrent Neural Network Transducer (RNN-T) architecture was proposed.

RNN-T

RNN-T allows for considering all the possible alignments of input and output sequences during training. In order to represent different alignments an additional blank token ('null' mapping) is introduced; henceforth denoted as $\varnothing$. Different alignments of the same sequence are then the same word pieces with blank symbols between them. Figure (1) demonstrates an example of how different alignments look like. Enumerating all the possible alignments would be intractable, however in order to compute the loss value we can utilize a simple efficient dynamic programming algorithm.

A detailed presentation of RNN-T architecture is given in the next chapter, for now it is sufficient to notice that minimising the RNN-T loss is equivalent to minimising the conditional probability of outputting correct sentence transcription, given a set of audio features. However, the performance of these models is measured in completely different manner, namely by Word

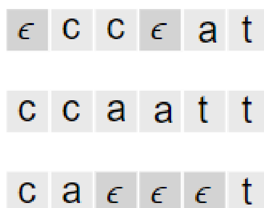


Figure 1: *Different alignments of the word 'cat'*

Error Rate (WER), which is – to put it simply – an edit distance for sentences. The WER between output of model and the sample label is a score for a single sample.

MWER

Recent papers by Prabhavalkar et. al. [10] and Guo et. al.[7] showed that it is possible to replace the cross entropy loss with a loss function based on WER. That is instead of minimising the conditional probability of outputting correct sentence transcription given set of audio features, we can directly minimise the expected value of word errors in the conditional probability distribution generated by the model. This procedure is known as Minimum Word Error Rate Training (MWER). The new training regime was found to improve the performance of the model, measured as improvement of the WER metric.

Monotonic RNN-T Loss

RNN-T model is capable of streaming recognition. Empirical studies found an undesired phenomenon in streaming recognition where the model does not output anything for a little while, and suddenly outputs multiple labels at the same time, instead of smoothly streaming the recognition results over time. In order to solve this issue a modified version of RNN-T Loss was introduced, called Monotonic RNN-T Loss [12].

From this project standpoint, we are particularly interested in one of the Monotonic RNN-T properties. That is, the modified implementation offers slightly worse WER, but enables Monotonic inference offering great speed up in comparison to regular RNN-T inference. In order to perform MWER training inference is called multiple times, therefore introducing the Monotonic RNN-T Loss reduces the training time significantly.

Aim of the thesis

The aim of the work was to create an open source implementation of the MWER training method allowing for a simple utilization of the technique by the research community to train end-to-end speech recognition model.

Implementation of end-to-end pipeline is beyond the scope of the project, therefore we decided to modify an existing open-source solution, Automatic Speech Recognition toolkit TensorFlowASR [9]. The scope of the work includes implementation of a modified version of the regular RNN-T Loss, called Monotonic RNN-T Loss [12] and wrapping it around with MWER Loss module capable of performing MWER training. The said module is then integrated with the TensorFlowASR Library allowing the user to use its convenient out-of-the box interface to train a speech recognition solution.

Other solutions

At the point of writing the thesis we could not identify any open-source solutions implementing the MWER Training based on Monotonic RNN-T Loss. To this date, we did find only one open source toolkit utilising MWER Training, that is the Tencent's 'Pika' project [11] that is a result of Tencent's research team paper [13]. However, the aforementioned solution does not allow for training a model using MWER loss from scratch, but rather improving an already trained RNN-T model using this technique. This, combined with the fact that the solution utilizes a different technology stack makes the solution unsuitable for comparison with our project.

We have not identified any open-source implementations of the Monotonic RNN-T Loss yet.

There are numerous solutions implementing RNN-T Loss available one of them being the TensorFlowASR [9] library, which offers a TensorFlow implementation of RNN-T Loss. We treat this implementation as a baseline for comparisons with the Monotonic RNN-T Loss.

Quality measurement

The goal of this thesis is to provide a working, open source implementation of MWER Training based on Monotonic RNN-T Loss. We will assume the solution is functional, and therefore we met the goal of the project, if the following conditions are met:

1. Solution passes the tests verifying the correct computation of losses and gradients.

2. MWER aftertraining procedure provides model with better accuracy in terms of WER metric compared to Monotonic RNN-T training.

Requirements

In order to formalize the desired output of the project we have define a set of functional and non-functional requirements.

Functional Requirements

The end user will be able to access a TensorFlow speech processing library using MWER loss to provide speech recognition capacities.

Table 1: *Functional Requirements*

Name	Description	System Response
MWER Loss value calculation.	Calculation of the value of loss function for given output of neural network.	A single value of loss function.
MWER Gradient calculation of loss with respect to logits.	Given logits as neural network output, the system efficiently calculates a gradient of loss with respect to the logits.	A tensor of values in the shape of logit tensor.
RNN-T Monnotonic Loss value calculation.	Calculation of the value of loss function for given output of neural network.	A single value of loss function.
RNN-T Monnotonic Gradient calculation of loss with respect to logits.	Given logits as neural network output, the system efficiently calculates a gradient of loss with respect to the logits.	A tensor of values in the shape of logit tensor.
Loss functions integration with open source speech processing kit.	Given speech fragments and labels the system should train the model using MWER loss.	Model capable of recognizing speech.

Non-Functional Requirements

In addition to the delivering the functional requirements, we aim to maintain quality of the solution. The key requirements determine the project quality are captured with the following points.

Table 2: *Non-Functional Requirements*

Requirements Area	No.	Description
Usability	1	Easily accessible TensorFlow interface in Python language.
Reliability	2	Calculations are expected to be deterministic and reconstructable.
Reliability	3	Calculations are expected to numerically stable.
Reliability	4	Computation of analytical gradients match the results obtained using numerical methods.
Performance	5	System is expected to use parallel programming paradigms and properly utilise GPUs capabilities.

Division of work

Table (3) summarizes the work division between the team members.

Table 3: *Division of Work*

Name	Work done
Jakub Frąc	Implementation of MWER Loss.
Jeremi Kurdek	Implementation of Monotonic RNN-T Loss.

Thesis organization

The body of the work is organized as follows. First, in Chapter (1) the reader is presented with a brief theoretical introduction to the covered topics. The next Chapter (2) covers the design and functioning of the provided implementation. Then, in Chapter (3) we discuss the means we took

to verify the quality of the solution, we introduce domain-specific testing methods and describe what tests were applied to each module. Next, in Chapter(4) we summarize the experiments we run in order to assess the capabilities of the provided solution and discuss the results. Finally, we offer concluding remarks in Chapter (5).

1. Theoretical background

In this chapter we summarize the underlying theoretical foundations behind the solution. The MWER Loss is build upon the RNN-T architecture, closely interacting with it and utilizing a modified version of the original RNN-T Loss (Monotonic RNN-T).

1.1. Baseline RNN-T

RNN-T architecture [5] [4] was introduced as an end-to-end, probabilistic sequence transduction, system able to transform any input sequence into any finite, discrete output sequence.

Architecture

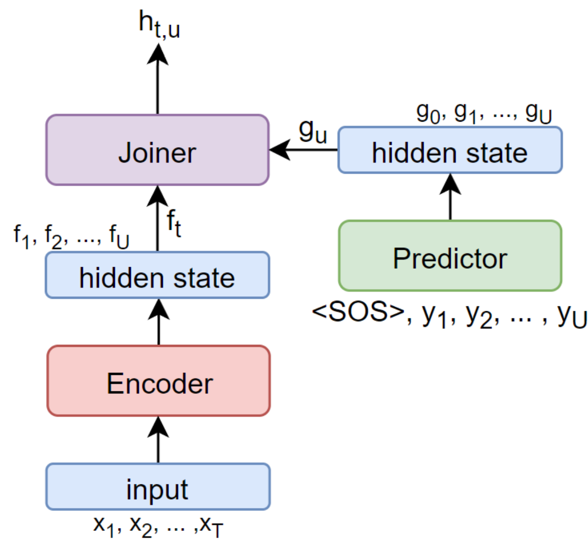


Figure 2: *RNN-T architecture*

The RNN-T architecture consists of an encoder, a prediction (autoregressive decoder) network and a joint network. Intuitively encoder is responsible for understanding the audio, based on the current input, it tries to predict what the next word in the sequence would be. On the other hand, predictor tries to predict the next word in the sequence using the the previously predicted words,

that is it tries to tell which word is going to occur based on the context. Joiner is a network that combines the output of the encoder with the output of the predictor, applies softmax and returns a probability distribution over all labels for each encoder timestep and output symbol.

More precisely, let x be a length T input sequence belonging to the set X^* of all sequences over some input space X. Let y be a length U output sequence belonging to the set Y^* of all sequences over some output space Y. Then let us define the extended output space $\bar{Y} = Y \cup \varnothing$, where $\varnothing$ denotes the blank symbol. The blank symbol can be interpreted as outputting 'nothing', its role is to denote different alignments between the input and output sequence. Let $\mathbf{B} : \bar{Y}^* \to Y^*$ be a function removing blanks from the alignments in $\bar{Y}^*$. Then, $\mathbf{B}^{-1}(y)$ denotes the set of all possible alignments (blank paddings) of sequence $y \in Y^*$. RNN-T loss defines a conditional probability distribution $P(a \in \bar{Y}|x)$, which can be collapsed to a distribution over Y^*:

$$P(y \in Y^*|x) = \sum_{a \in \mathbf{B}^{-1}(y)} P(a|x) \tag{1.1}$$

The encoder receives acoustic vector x and transforms it into a sequence of hidden states f_t where t is the time index. The autoregressive prediction network takes the previous prediction y_{u-1} and produces a hidden representation g_u. The joiner network takes each combination of encoder output f_t and prediction network output h_u and computes output logits $h_{t,u}$. The final posterior for each output token k – $P(k|t,u)$ – is obtained after applying the softmax operation.

1.1.1. Training

The RNN-T loss function can be defined as the negative log posterior of label sequence y given input vector x.

$$L_{RNNT} = -lnP(y|x) \tag{1.2}$$

where $P(y|x)$ is defined as in (1.1)

To minimize L_{RNNT} an efficient forward-backward algorithm was proposed:

$$\frac{\partial L_{RNNT}}{\partial P(k|t,u)} = -\frac{\alpha(t,u)}{P(y|x)} \begin{cases} \beta(t,u+1) & \text{if } k = y_{u+1} \\ \beta(t+1,u) & \text{if } k = \varnothing \\ 0 & \text{otherwise} \end{cases} \tag{1.3}$$

To simplify notation, define

$$y(t, u) = P(y_{u+1}|t, u) \tag{1.4}$$

$$\varnothing(t, u) = P(\varnothing|t, u) \tag{1.5}$$

That is $y(t, u)$ denotes the probability of outputting the next element of the label at time step t and decoder step u. On the other hand, $\varnothing(t, u)$ is the probability of outputting a blank symbol at time step t and decoder step u.

The core of the forward-backward algorithm is defined by two variables: α and β. The forward variable $\alpha(t, u)$ is defined as the probability of outputting $y_{[1:u]}$ during $f_{[1:t]}$. The forward variables for all $1 \leq t \leq T$ and $0 \leq u \leq U$ can be calculated recursively using

$$\alpha(t, u) = \alpha(t - 1, u) \cdot \varnothing(t - 1, u) + \alpha(t, u - 1) \cdot y(t, u - 1) \tag{1.6}$$

with initial condition $\alpha(1, 0) = 1$. The total output sequence probability is equal to the forward variable at the terminal node:

$$\Pr(y|x) = \alpha(T, U) \cdot \varnothing(T, U) \tag{1.7}$$

The backward variable $\beta(t, u)$ is defined as the probability of outputting $y_{[u+1:U]}$ during $f_{[t:T]}$. Then

$$\beta(t, u) = \beta(t + 1, u) \cdot \varnothing(t, u) + \beta(t, u + 1) \cdot y(t, u) \tag{1.8}$$

with initial condition $\beta(T, U) = \varnothing(T, U)$.

From the definition of the forward and backward variables it follows that their product $\alpha(t, u) \cdot \beta(t, u)$ at any point (t, u) in the output lattice is equal to the probability of emitting the complete output sequence if y_u is emitted during transcription step t.

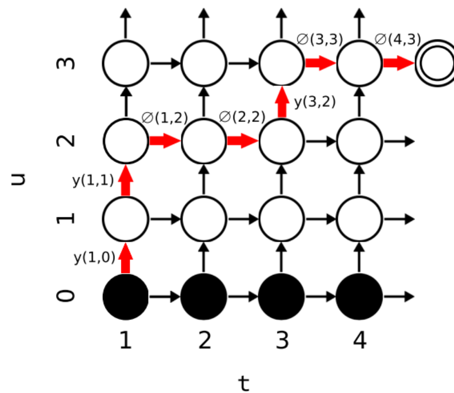


Figure 3: *Output probability lattice*

Formula (1.3) allows us then to compute the gradients with respect to the softmax probabilities. In order to obtain gradients with respect to activations another step is necessary:

$$\frac{\partial L}{\partial h(k,t,u)} = \sum_{k' \in \bar{Y}} \frac{\partial L}{\Pr(k'|t,u)} \frac{\partial \Pr(k'|t,u)}{\partial h(k,t,u)} \tag{1.9}$$

This formula is not optimal for concurrent implementation using GPU threads. For this goal a transformed version has been proposed [8]:

$$\frac{\partial L}{\partial h(k,t,u)} = \frac{\alpha(t,u) \cdot \Pr(k|t,u)}{\Pr(y^*|x)} \cdot \beta(t,u) - \frac{\alpha(t,u) \cdot \Pr(k|t,u)}{\Pr(y^*|x)} \begin{cases} \beta(t,u+1), & k = y_{u+1}, u < U \\ \beta(t+1,u), & k = \varnothing, t < T \\ 1, & k = \varnothing, t = T, u = U \\ 0, & \text{otherwise} \end{cases} \tag{1.10}$$

1.2. Monotonic RNN-T

1.2.1. Loss and training

The computation of α and β variables in the above formula is done using diagonals, that is instead of computing values for single elements on the lattice, we are computing the value of the whole diagonal at once. This results with $T + U$ diagonal computations. Recent paper by Tripathi et. al. proposed a more efficient way of computing the forward and backward variable, based on the assumption that $T >= U$, that is the input sequence must not be shorter than the target sequence. The assumption can found true empirically, as speech is sampled with high frequency and a single output token usually consists of more than one letter. This allows to change the formulas used to compute the forward and backward variables as following:

$$\alpha(t,u) = \alpha(t-1,u) \cdot \varnothing(t-1,u) + \alpha(t-1,u-1) \cdot y(t-1,u-1) \tag{1.11}$$

$$\beta(t,u) = \beta(t+1,u) \cdot \varnothing(t,u) + \beta(t+1,u+1) \cdot y(t,u) \tag{1.12}$$

That is instead of stepping up in the lattice to produce a label element we are making a step on the diagonal and producing both blank token and label element. This modification is demonstrated by Figure (4).

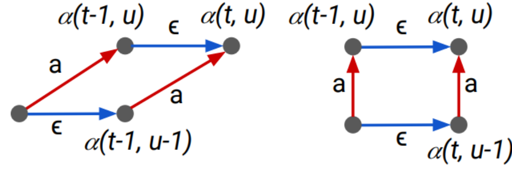


Figure 4: *Monotonic RNN-T step example*

This change also leads to the modification of the formulas used to compute the gradients with respect to the activations. The gradient part related to the probability of outputting the next item of the label has to account for the diagonal step. The modified formula is captured by Equation (1.13).

$$\frac{\partial L_{MRNNT}}{\partial h(k,t,u)} = \frac{\alpha(t,u)\cdot \Pr(k|t,u)}{\Pr(y^*|x)} \cdot \beta(t,u) - \frac{\alpha(t,u)\cdot \Pr(k|t,u)}{\Pr(y^*|x)} \begin{cases} \beta(t+1,u+1), & k = y_{u+1}, u < U \\ \beta(t+1,u), & k = \varnothing, t < T \\ 1, & k = \varnothing, t = T, u = U \\ 0, & \text{otherwise} \end{cases} \tag{1.13}$$

1.2.2. Inference

Since MWER training relies heavily on inference speed (N-best list generation) - similar to Monotonic RNN-T - there has been developed monotonic decoding strategy [12]. For each time-frame it can either expand hypotheses by feeding previous labels to decoder network and then computing the joint network probabilities, or it can terminate hypotheses by taking a blank label transition and move hypothesis to beam for next frame.

Algorithm 1 Monotonic inference algorithm

Require:

t_{max} is a size of encoder output

$encoderOut$ is a list of size t_{max} of encoder outputs

max_{hyps} is the number of maximum hypotheses to keep in beam

$hyps \leftarrow blankList$

$t \leftarrow 0$

while $t < t_{max}$ **do**

  $hidden \leftarrow Decoder(hyps)$

  $tokens_{new} \leftarrow Joint(encoderOut_t, hidden)$

  $hyps_{new} \leftarrow JoinCurrentTokensWithPreviousHyps(hyps, tokens_{new})$

  $hyps \leftarrow GetTopK(hyps_{new}, max_{hyps})$

  $t \leftarrow t + 1$

end while

Algorithm 1 explains monotonic inference (beam search). Below there is presented an explanation of the used functions:

- $Decoder()$ is a call to autoregressive language model. It receives already inferred part of hypothesis as input and returns decoder hidden state, which later on will be used to infer the next token in sentence.

- $Joint()$ is a call to joint network. It receives encoder and decoder hidden states and returns a tensor of probabilities for each token.

- $JoinCurrentTokensWithPreviousHyps()$ is a function that concatenates hypothesis and next token tensors, thus creating new hypotheses.

- $GetTopK()$ is a function that receives list of tokens and an integer K as input and returns top k most probable hypotheses.

It can be observed, that instead of passing scalars as algorithm input, the batches of data can be passed, which results in speedup of decoding time.

1.3. MWER loss function

In order to perform discriminative training between hypotheses we use expected value of Word Error Rate as a loss function.

$$L_{MWER} = E[R(y, y^r)] = \sum_{y_i \in Y} P(y_i|x) R(y_i, y^r) \tag{1.14}$$

The hypothesis space Y can also be approximated by top N most probable hypotheses inferred from probability distribution $P(k|t, u)$.

$$L_{MWER}(x, y^r) = \sum_{y_n \in nbest(x)} \hat{P}(y_n|x) R(y_n, y^r) \tag{1.15}$$

Where

- $nbest(x)$ are hypotheses generated using N-Best lists algorithm on output probability space $P(k|t, u)$ given by joiner model.
- $\hat{P}(y_n|x) = \frac{P(y_n|x)}{\sum_{y'_i \in nbest(x)} P(y'_i|x)}$ is a normalized probability of hypothesis y_i over sum of probabilities of all of the hypotheses in $nbest(x)$.
- $R(y_i, y^r)$ is a risk function, in our case Word Error Rate function between hypothesis y_i and label sequence y^r.

For a ease of implementation and guaranteed numerical stability the Equation (1.15) can also be rewritten as:

$$L_{MWER}(x, y^r) = \sum_{y_n \in nbest(x)} softmax(logP(y_n|x)) R(y_n, y^r) \tag{1.16}$$

1.3.1. Training

The output probability distribution (output of joint network) generated by model, for a given sample, will be denoted as $P_{y_i}(k|t, u)$ (which means probability of token k, given t acoustic encoder step and u predictor step) for output sentence y_i inferred using the algorithm described in 1.2. The probability of inferring a sequence y_i will be denoted by $P(y_i|x)$. Then we know that:

$$\frac{\partial L_{MWER}}{\partial logP(y_i|x)} = \hat{P}(y_i|x)(R(y_i, y^r) - \hat{R}) \tag{1.17}$$

Where

- $\hat{P}(y_i|x) = \frac{P(y_i|x)}{\sum_{y'_i \in nbest(x)} P(y'_i|x)}$ is a normalized probability of hypothesis y_i over sum of probabilities of all of the hypotheses in $nbest(x)$.
- y^r is a sample label.
- $\hat{R} = \sum_{y'_i \in nbest(x)} \hat{P}(y'_i|x) R(y'_i, y^r)$ is the expected number of word errors within the N-best list.

From definition of monotonic RNN-T loss we know that:

$$\frac{\partial logP(y_i|x)}{\partial P_{y_i}(k|t,u)} = \frac{\alpha(t,u)}{P_{y_i}(y|x)} \begin{cases} \beta(t+1, u+1) & \text{if } k = y_{u+1} \\ \beta(t+1, u) & \text{if } k = \varnothing \\ 0 & \text{otherwise} \end{cases} \tag{1.18}$$

Our goal is to find the derivative of loss with respect to joint network output. This allows us to return gradient with respect to the outer-most layer in RNN-T model, which could further be propagated to deeper layers with backpropagation algorithm. Therefore:

$$\frac{\partial L_{MWER}}{\partial P_{y_i}(k|t,u)} = \frac{\partial L_{MWER}}{\partial logP(y_i|x)} \frac{\partial logP(y_i|x)}{\partial P_{y_i}(k|t,u)} \tag{1.19}$$

Which is the product of (respectively) 1.17 and 1.18. Derivation of the above equations can be found in [7] and [5].

As in [13] we believe that without RNN-T loss regularization the N-Best generation (beam search) slows down significantly which reduced the training efficiency substantially. Therefore, the regularized MWER loss function is adopted:

$$\hat{L}_{MWER} = L_{MWER} + \lambda L_{MRNNT} \tag{1.20}$$

where λ is the regularization factor which we set to 1.0 throughout all of our experiments, as in [13].

2. Architecture

The solution is split into three logical modules, MWER Loss, Beam Search and Monotonic RNN-T Loss. The modules and their inter-dependencies are visualized on Figure (5).

The MWER Loss module interacts directly with the TensorFlowASR Library, during training time a Deep Learning model defined as a part of the said library calls the MWER Loss module and begins the process of MWER Loss Calculation. In order to compute the loss and gradients, MWER Loss module utilizes two other modules: Beam Search and Monotonic RNN-T Loss.

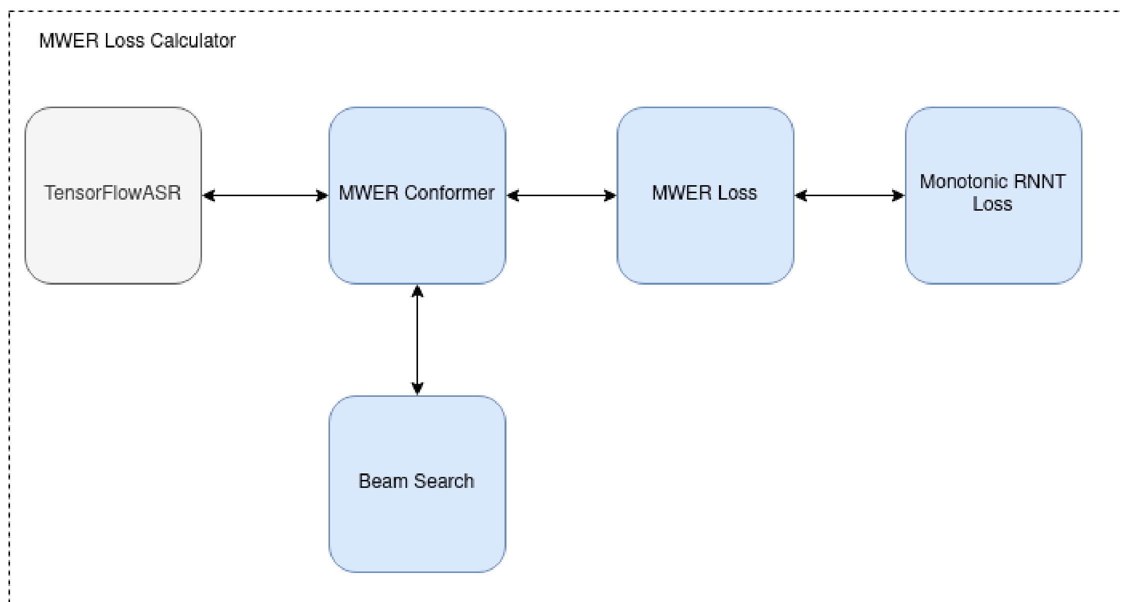


Figure 5: *Architecture overview*

The Beam Search module allows for generation of N-best hypothesis list, whereas the Montonic RNN-T Loss module is responsible for calculation of Monotonic RNN-T loss and gradients with the arguments provided by the trained model via the MWER Loss module.

The MWER Loss module utilizes then the Monotonic RNN-T Loss and the N-best list in order to compute the MWER loss and gradients. The computed loss and gradients are then returned to the model contained in the TensorFlowASR library for training.

2.1. TensorFlowASR

The interface allowing for interacting with our solution is provided by an open-source library TensorFlowASR [9]. The library offers several end-to-end machine learning solutions in the field of ASR. Specifically it includes fully functional RNN-T model as well as a modified version of the basic RNN-T model called 'Conformer' [6]. Conformer can be thought as an RNN-T with additional convolution and attention layers. We have chosen to build our solution on the top of the Conformer model for its usage of attention layers. It is important to note; however, that the design of the model remains the same as RNN-T and is captured by Figure (2). Namely the Conformer model utilizes the same building blocks: Encoder, Predictor and Joiner as the original RNN-T.

2.2. Module Description

2.2.1. Monotonic RNN-T

The Monotonic RNN-T module is responsible for computation of Monotonic RNN-T Loss and gradient of the loss with respect to the input logits.

MonotonicRNNTLoss class

The class serves as an entry point to the module, that is it offers a public interface the other modules can call. The class itself inherits from tf.keras.losses.Loss class, which is a base Loss class contained in high-level TensorFlow API called Keras. This allows for injecting the custom loss into any model utilizing Keras API with ease. The class overrides one method of the base class, namely the *call* method which given two inputs *y_true* and *y_pred* (model's output and the expected label) initiates the loss computation. The *y_true* and *y_pred* parameters are in fact dictionaries containing fields necessary for loss computation. Those fields include:

- *logits* - the output of the RNN-T model as described in Section (1).
- *labels* - the correct transductions of the input audio sequences.
- *labels_len* - the length of the corresponding labels for each element of the processed batch.
- *inputs_len* - the length of the corresponding input for each element of the processed batch.

The call method extracts logits and labels from the input data, and begin the loss and gradients computation logic. The internal computations are illustrated by the state diagram presented on

Figure (6).

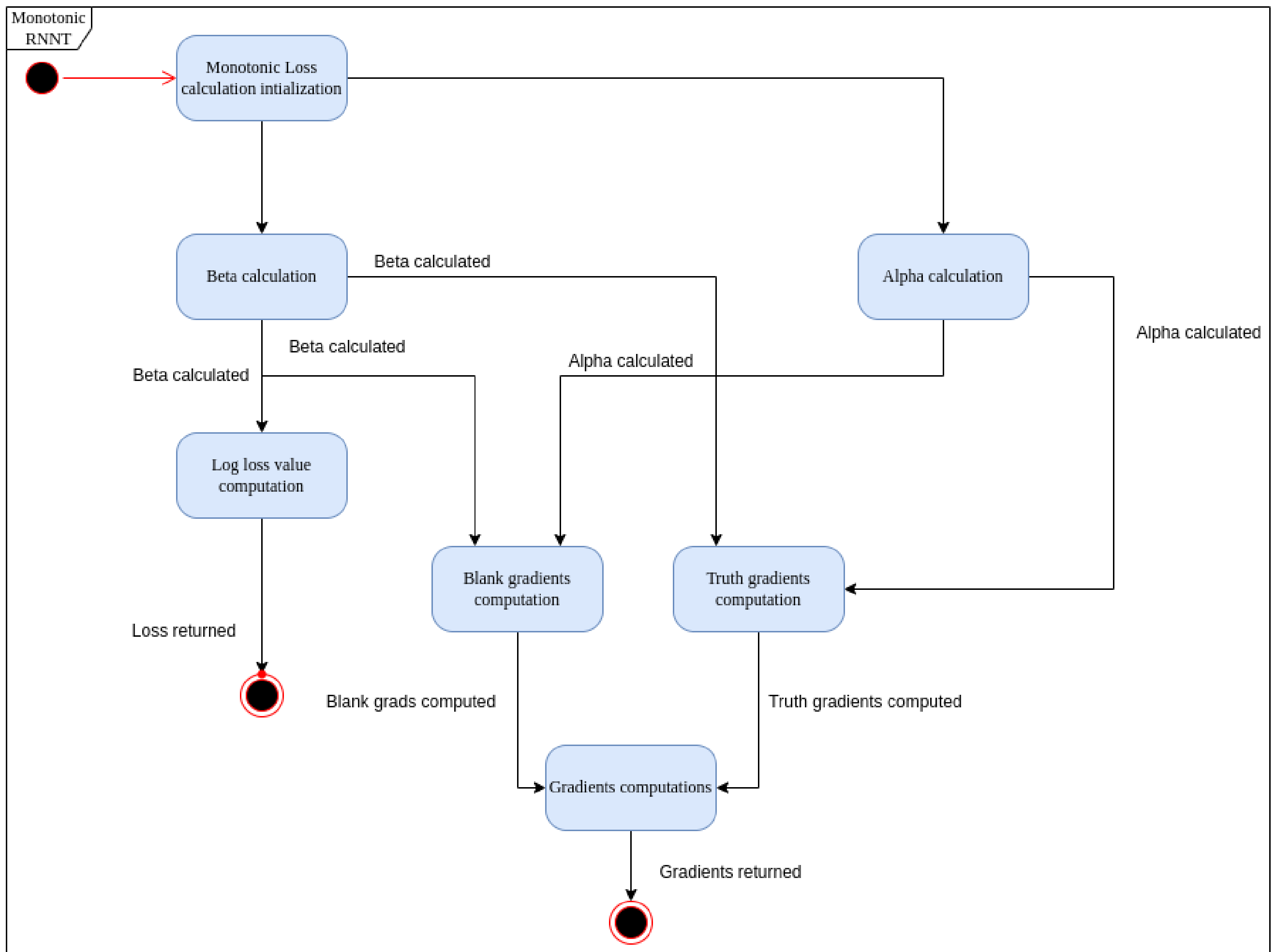


Figure 6: *Monotonic RNN-T module state diagram*

2.2.2. BeamSearch class

BeamSearch class implements monotonic inference algorithm described in Subsection (1.2.2). Its initialization arguments are:

- *vocabulary_size* - A size of vocabulary.
- *predict_net* - An instance of prediction network model, which is implemented by TensorFlowASR library.
- *joint_net* - An instance of joint network model, which is implemented by TensorFlowASR library.
- *blank_token* - An index of blank token.
- *beam_size* - A size of beam.

The class does not change its state throughout computation. There is a *call* method which serves as class instance's interface. The *call* method receives following parameters:

- *encoded* - A tensor of encoder outputs.
- *encoded_length* - A tensor of lengths of encoder outputs.
- *parallel_iterations* - A parameter passed signifying number of parallel computations. Increasing it might improve inference time with a cost of more GPU memory usage.
- *return_topk* - A boolean - when True method returns top K (where K is beam size) hypotheses instead of just one.

The class returns a pair of tensors (*predictions*, *probabilities*) - predictions in terms of token ids and their respective probabilities. The internal computations are illustrated by the state diagram presented on Figure (7).

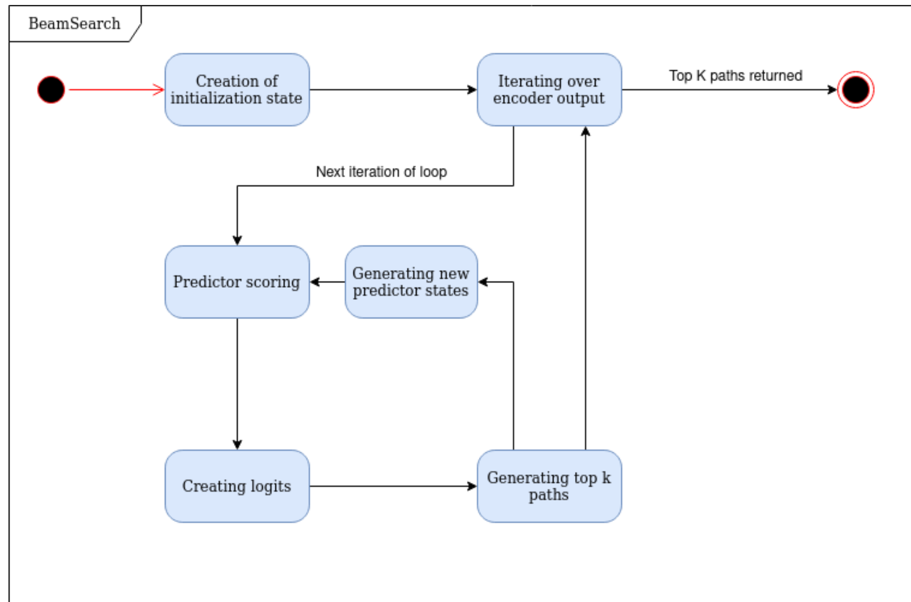


Figure 7: *Beam search module state diagram*

2.2.3. MWERLoss class

MWERLoss class is responsible for calculating MWERLoss value and its respective gradient as described in Section (1.3). Its initialization arguments are:

- *global_batch_size* - A batch size.
- *blank* - An index of blank token.

The class does not change its state throughout computation. There is a *call* method which serves as class instance's interface. The *call* method receives following parameters:

- *prediction* - A dictionary of predictions in terms of logits containing the following fields:
  - *logits* - A tensor of logits.
  - *logits_length* - A tensor of prediction lengths.
- *hypothesis* - A dictionary of tokenized predictions containing the following fields:
  - *labels* - Tokenized hypotheses.
  - *labels_length* - Lengths of hypotheses.
- *text_labels* - A dictionary of hypothesis in terms of text containing the following fields:
  - *text_hypotheses* - Tensor of hypotheses in text format.
  - *text_labels* - Tensor of original sample labels in text format.

It returns a tensor of loss values. If called under tf.GradientTape a gradient can also be calculated with respect to the loss. The internal computations are illustrated by the state diagram presented on Figure (9).

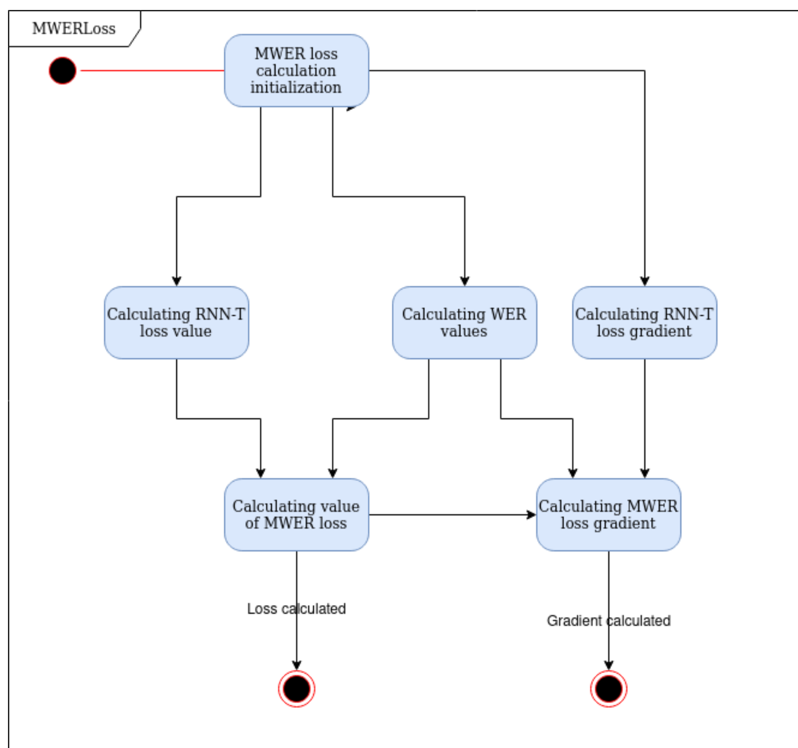


Figure 8: *MWER loss module state diagram*

2.2.4. MWERConformer class

A wrapper class created on top of Conformer class (implemented by TensorFlowASR library). The class utilizes polymorphism to overwrite *train_step* and *test_step* methods used for training. Both of these methods implement the following procedure:

1. Create encoder hidden state on the input sample .
2. Create predictor hidden state on the input label.
3. Merge encoder and predictor hidden states with a use of joint network.
4. Calculate the RNN-T loss value with the output logits as an input the monotonic RNN-T loss class instance.
5. Generate hypotheses using the Beam search class with the encoder hidden state as input.
6. Create predictor hidden state on the generated hypotheses.
7. Merge encoder and predictor hidden states (of hypotheses) with a use of joint network.
8. Calculate MWER loss value with the output logits (of hypotheses) as an input to the MWER loss class instance.
9. Interpolate values of RNN-T loss and MWER-loss.

In addition the *train_step* function also calculates the gradient of the interpolated loss and updates the models weights with a use of the optimizer. Its initialization arguments are:

- *optimizer* - An optimizer for updating the model weights.
- *global_batch_size* - A value of batch size.
- *blank* - An index of blank token.
- *run_eagerly* - A boolean - when True the model is run in eager mode instead of graph mode.

In addition to these parameters, the class also receives standard parameters as the Conformer class in the original TensorFlowASR library. The internal computations are illustrated by the state diagram presented on Figure (9).

Figure 9: *MWER Conformer module state diagram*

3. Tests

The chapter describes the means we took to verify the quality of the delivered solution against the functional and nonfunctional requirements defined in the Introduction. First we present the correctness assurance test methodology, then we shift to performance topic.

3.1. Testing methods introduction

Assessing quality of a custom loss and gradient functions implementations is a non trivial task. Below we describe the method we employed to test our solution along with explanation.

Manual Loss Calculation

The goal of this test is to test whether the loss implementation returns a correct value given some input. In order to verify that, we have precalculated the loss value on a small example by hand. The test passes if the value obtained by the implementation matches the value obtained using manual computation.

Finite Difference Gradient

The goal of this test method is to assure that the gradient of the loss concretion with respect to the logits is calculated correctly. This is a numerical differentiation method is based on the definition of derivative. The derivative of a function f at a point x is defined by the limit.

$$\frac{dy}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{3.1}$$

We can observe that the target $\frac{dy}{dx}$ derivative at some point x may be estimated by choosing sufficiently small constant h and calculating the below formula:

$$\frac{dy}{dx} \approx \frac{f(x+h) - f(x)}{h} \tag{3.2}$$

The method can be easily adapted for numerically estimating the gradient of function $f : \mathbb{R}^n \to \mathbb{R}$. In order to obtain the numerical gradient, we simply need to use the formula 3.2 for

each input variable.

We have incorporated finite difference gradient based tests to verify the correctness of the solution, the test passes when the numerical gradient matches the gradient obtained by programs computation.

Automatic differentiation gradient computation

In order to test the gradient we employ one more method, that is we compare the gradient obtained by auto-differentiation with the gradient obtained using our solution.

Automatic differentiation refers to a general way of taking a program which computes a value, and automatically constructing a procedure for computing derivatives of that value. The TensorFlow library comes with a reverse-mode automatic differentiation. That is the program generates a graph a function we want to differentiate and then computes the derivative in reverse fashion using the chain rule.

Although the procedure is much slower than implementation of analytic gradient calculation, we may use it to check whether the gradient that our implementation computes is accurate.

In order to test our solution, we simply need to compare the automatic differentiation gradient and analytical gradient, the test passes should the gradients be equal.

Monotonic inference tests

Since monotonic inference requires a predictor and joint networks to properly work, there have been developed "stub" classes that act like both of these models. Predictor class is deterministic and the outputs are designed to match the inputs in a tree-like fashion - given some prefix path the stub predictor network has predetermined output token for each of the paths. The stub joint network simply adds the probabilities from synthetic encoder output and predictor network. Because of the nature of said classes, it is easy to design inputs, simulate the expected behaviour of algorithm by hand, predict proper output and compare it with the output of the actual monotonic inference class.

3.2. Module test summary

The tests used to verify the quality of implementations may be found in the 'tests/mwer' directory. Below, the reader may find a description of what test methods were used to test different modules of the solution.

Table 4: *Test summary*

Module name	Tests applied
MWER Loss	• Automatic differentiation. • Finite Difference Gradient.
Monotonic RNN-T Loss	• Finite Difference Gradient. • Manual Loss Computation. • Property Check[1].
Beam Search	• Monotonic inference tests.

1. In order to test the implementation of the Monotonic RNN-T Loss we utilize one of its properties. That is, the final value of the forward variable α should be equal to the final value of the backward variable β. To assured the quality of implementation, we have turned this property check into unit test.

4. Experiments

4.1. Environment

Each experiment was run in the same environment. Below there are specific setups in terms of hardware and software.

4.1.1. Software

Since our experimental cloud had certain limitations, the software versions are older than the ones required in the project. Below there is a list of major libraries and their versions:

- TensorFlow 2.3.0
- Python 3.7
- CUDA 10.1
- cuDNN 7.6

Even though the libraries are a bit older than the ones required in the project, no major changes in code were required. Hence the logic of the code stayed the same and the experiments are valid.

4.1.2. Hardware

Each experiment was conducted on the following hardware setup:

- 8x Nvidia Tesla V100 32GB
- 64x CPU
- 480GB RAM memory

4.2. Setup

4.2.1. Data

For training a LibriSpeech [1] dataset has been used. LibriSpeech is a corpus of approximately 1000 hours of 16kHz read English speech, prepared by Vassil Panayotov with the assistance of Daniel Povey. The data is derived from read audiobooks, and has been carefully segmented and aligned. For training 360 hours of clean speech was used. As for development and tests both datasets have similar length of around 5 hours of clean speech.

4.2.2. Methodology

At first we consider random (non-trained) Conformer model. It is trained with librispeech-360-clean (360 hours of clean speech) per epoch with monotonic RNN-T loss for around 100 epochs. Then after each epoch, the validation loss on dataset dev-clean is calculated, and only models with best results in terms of validation loss are kept. After the standard training, the pretrained model is used for MWER aftertraining session (also around 100 epochs) with librispeech-360-clean. Similar to standard training – after each epoch there is calculated validation loss on dataset dev-clean and only models with best results in terms of validation loss are kept.

There is a single training session for standard training with monotonic RNN-T loss, and 4 training sessions for MWER aftertraining - each having different beam size parameter. Eventually we end up with 5 models (1 standard, 4 aftertrained) and each of them is used separately for inference on test-clean dataset. With the results of each model we can calculate mean Word Error Rate for the dataset and compare the models using this metric.

4.2.3. Technical setup

Text tokenizer

As for text tokenizer we used SentencePiece model. SentencePiece [3] is an unsupervised text tokenizer and detokenizer mainly for Neural Network-based text generation systems where the vocabulary size is predetermined prior to the neural model training. For our training we used a model trained on LibriSpeech text corpus with vocabulary size of 512.

Batch size vs beam size

Due to high memory complexity of the solution, we were forced to balance between size of minibatches and beam size for training, since both of these parameters highly affect the

GPU memory usage. For the standard RNN-T training we used batch size of 4. For MWER aftertraining:

- For beam size 1, batch size 3 was used.
- For beam size 2, batch size 1 was used.
- for beam size 3, batch size 1 was used.
- for beam size 4, batch size 1 was used.

There have been problems with beam size 4 setup, since the biggest samples would not fit into the memory. To address this issue we considered sample lengths and removed the samples with $Z_{score} > 2$. Z_{score} is the number of standard deviations by which the value of a raw score is above or below the mean value of what is being observed or measured. Fortunately there were only 18 of such samples, which contributed only around 0.018% of the whole dataset, meaning no significant amount of information was lost.

4.3. Training and validation loss

4.3.1. Monotonic RNN-T

Figure (10) shows the values of validation (blue) and training (orange) losses over 100 epochs of training. We can conclude that model successfully converged on the given dataset and no underfitting can be observed.

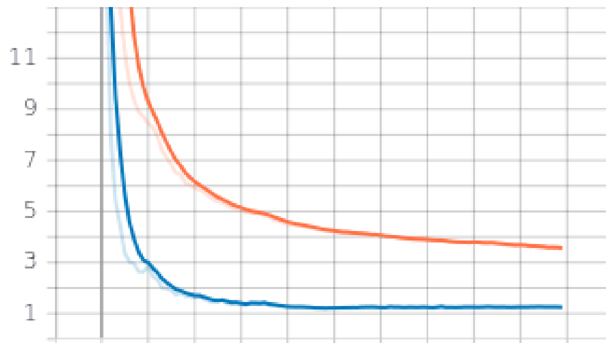


Figure 10: *Validation (blue) and training (orange) loss value graph on Monotonic RNN-T training*

Training time

The figure 10 represents how training and validation loss changed over 100 epochs of training. The time to perform this training was around 50 hours. Since the overfitting started around

50th epoch, we can see this time could possibly be cut in half down to 24 hours.

4.3.2. MWER

On the graphs presented on Figures (11) to (14) we can see how training and validation losses changed over the course of 100 training epochs. It can be observed that no model underfitted the data and all of the models started to overfit in the second half of the training.

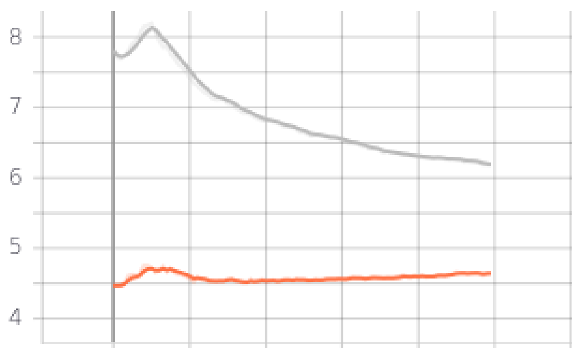


Figure 11: *Validation (orange) and training (grey) loss value graph on MWER training (beam width 1)*

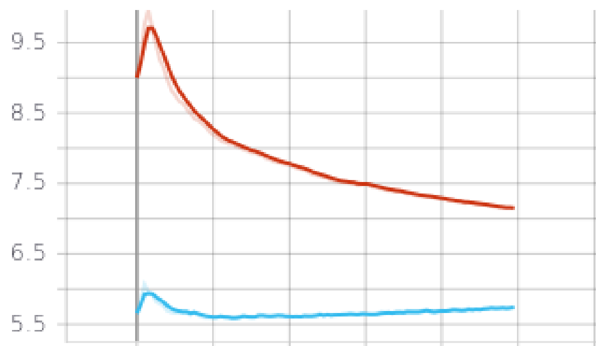


Figure 12: *Validation (blue) and training (red) loss value graph on MWER training (beam width 2)*

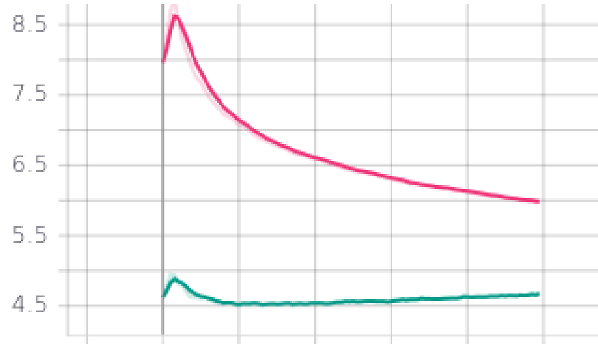


Figure 13: *Validation (green) and training (pink) loss value graph on MWER training (beam width 3)*

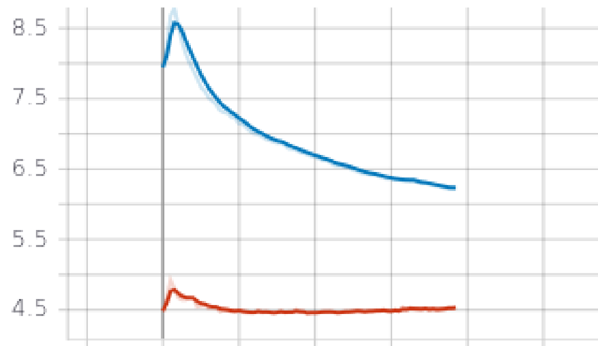


Figure 14: *Validation (orange) and training (blue) loss value graph on MWER training (beam width 4)*

It is visible that validation loss increased during first few epochs of training, but then eventually converged to the smaller value than it was in the beginning. The only exception is the scenario with beam size equal to 1, where the best validation loss was achieved during first few epochs of training.

Training time

Figures (11) to (14) represent how training and validation loss changed over 100 epochs of training. For beam sizes 2, 3 and 4 the training took around 500 hours to finish. It was not the case for the setup with beam size 1, that could efficiently utilize higher batch size of 3, hence reducing the training time down to 290 hours.

From the graphs it can also be observed that all of the training sessions started to overfit around 40th epoch, hence the actual training time could be reduced to 200 hours for beam sizes 2, 3, 4 and 116 hours for beam size 1.

It is safe to conclude that our solution was around 10 times slower in worst case scenario and 6 times slower in the best scenario. If the memory usage was reduced, or the available GPU

memory was increased, we believe the time could be reduced even further. We conclude that from the fact, that setup with beam size 1 used a batch size of 3, which reduced the training time significantly.

4.4. Word Error Rate score

Each of the models performed an inference with beam widths of 1, 2 and 4 on the test set. The results of word error rate metric are presented in the table below. The best results are highlighted in **bold**.

Table 5: *WER results on the test dataset*

Beam size	1	2	4
Monotonic RNN-T	14.75%	14.54%	14.68%
MWER (beam width 1)	**13.81%**	**13.67%**	**13.64%**
MWER (beam width 2)	15.17%	14.88%	14.96%
MWER (beam width 3)	14.79%	14.53%	14.75%
MWER (beam width 4)	14.51%	14.28%	14.49%

Table 6: *Relative WER improvement compared to baseline model*

Beam size	1	2	4
MWER (beam width 1)	**6.4%**	**6.0%**	**7.1%**
MWER (beam width 2)	-2.8%	-2.3%	-1.9%
MWER (beam width 3)	-0.3%	0.1%	-0.5%
MWER (beam width 4)	1.6%	1.8%	1.3%

From the Table (5) we can conclude that MWER training with beam width 1 performed the best, improving the result by around 1 percentage point. However, in this setup the batch size was equal to 3. Batch size can affect the way the loss function is optimized and because of that it is not perfectly clear whether small beam size would always result in the best result.

Except for that model, the only one that performed better was the one trained with beam width 4, but it is hard to deduce whether this small improvement is statistically significant.

Expected results

Standard RNN-T training rewards model for high probability of the label sentence, with no regards to sentences with smaller probabilities. Because of that the next-most-probable sentence might result in high WER. We expected MWER aftertraining to force model to increase probabilities of sentences with smaller WER values, hence smoothing the distribution.

We believe that smoother distribution would allow us to infer better result with increased beam size. Hence increasing inference beam size should result in increase of accuracy. Surprisingly, it can be observed that increasing beam width during inference does not necessarily improve the overall result in terms of WER metric. In [7] we can see that increasing beam size during inference would result in increase of accuracy, instead of decreasing it, making this result even more unexpected.

Comparison with other works

From the Table (6) it can be deduced that, overall, we achieved between $\sim 6\% - 7\%$ relative improvement. These are quite similar results compared to [7] and [10] where the relative improvements (without additional procedures) were respectively 3.6% and 7.4%.

It is worth mentioning though, that these articles claimed to have trained the models on orders of magnitude larger datasets than our - respectively $\sim 23,000$ and $\sim 21,000$ hours. Therefore models trained on this much data posses much smaller variance compared to our model, slightly undermining the basis for comparison.

4.5. Examples

In addition to reporting the WER score, we may look at the actual transcriptions generated during inference in comparison to the ground truth labels. The below Table (7) lists a few examples selected from the inference sentences. The inference was run on the best performing model, that is the one trained with batch size 3 and beam width 1. The errors are marked with **bold**.

Table 7: *Transduction examples*

Ground truths	Best model transductions
unc knocked at the door of the house and a chubby pleasant faced woman dressed all in blue opened it and greeted the visitors with a smile	**a cannot** the door of the house and a chubby pleasant faced woman dressed all **him** blue opened it and greeted the visitors with a smile
algebra medicine botany have each their slang	algebra medicine **bartony** have each **there** slang
the good natured audience in pity to fallen majesty showed for once greater deference to the king than to the minister and sung the psalm which the former had called for	the good **natitureri ordin** in pity **for an** majesty showed for **one scratter** deference to the king than to the minister and **some dis** which the former had called for

As indicated from analysis of the WER scores, the model gets the vast majority of words in a sentences correctly and fails on only a few. We can observe that the model struggles with similarly sounding words as well as has a hard time deciphering less frequently occurring phrases.

It should be noted; however, that because of the method employed for dataset generation some of the sentences (e.g. first ground truth sentence in Table (7) have artifacts. This, combined with the fact the LibriSpeech corpus contains sentences coming from literary English, makes the feat of transducing the audio sequences all the more impressive.

5. Conclusion

The goal of the thesis was to provide an open source implementation of MWER Training based on Monotonic RNN-T Loss. We implemented the modified training procedure and integrated it with the TensorFlowASR Library. Furthermore, we tested the quality of the solution using unit tests and numerical methods. The solution was also evaluated experimentally, we run experiments with several configurations and compared the results to the baseline model in order to demonstrate the effects of the implemented training procedure.

All in all, we consider the project as a success. We delivered an open-source implementation of MWER training whose quality is assured by implemented tests. The experiments indicated a slight improvement in WER score after applying the MWER aftertraining procedure. The obtained results are consistent with the results reported by the research community.

Nonetheless, there are some areas of our work that could use improvement.

First, in order to fully evaluate the solution more experiments would be necessary. Ideally, we would train the model on larger datasets. The state of the art models for audio transduction are usually trained on datasets containing tens of thousands of speech hours ([7] [10] [13]), whereas we used only 360 hours of speech. This would allow for judging the solution with more certainty.

Second, we believe that the solution could be improved by identifying and removing the bottlenecks, which would allow us to perform model aftertraining with larger beam widths and batch sizes.

Finally, we could remove the TensorFlowASR library and create our own model in order to retain a full control of the training process.

Acknowledgments

We would like to show our appreciation to Huawei for allowing us to run our training and experiments on their computing cloud, especially to Sergey Bankievich Ph.D. for allowing us to take this project. We would also like to personally thank Huawei RnD employee Alexey Tochin Ph.D. for meritorical support and our thesis supervisor dr hab. inż. Agnieszka Jastrzębska for taking care of our project.

Bibliography

[1] Librispeech asr corpus. `https://www.openslr.org/12/`.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[3] Google. Sentencepiece - an unsupervised text tokenizer and detokenizer. `https://github.com/google/sentencepiece`.

[4] Alex Graves. Sequence transduction with recurrent neural networks, 2012.

[5] Alex Graves, Abdel rahman Mohamed, and Geoffrey E. Hinton. Speech recognition with deep recurrent neural networks. *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649, 2013.

[6] Anmol Gulati, Chung-Cheng Chiu, James Qin, Jiahui Yu, Niki Parmar, Ruoming Pang, Shibo Wang, Wei Han, Yonghui Wu, Yu Zhang, and Zhengdong Zhang, editors. *Conformer: Convolution-augmented Transformer for Speech Recognition*, 2020.

[7] Jinxi Guo, Gautam Tiwari, Jasha Droppo, Maarten Van Segbroeck, Che-Wei Huang, Andreas Stolcke, and Roland Maas. Efficient Minimum Word Error Rate Training of RNN-Transducer for End-to-End Speech Recognition. In *Proc. Interspeech 2020*, pages 2807–2811, 2020.

[8] Mingkun Huang. Notes on sequence transduction with recurrent neural networks. `https://github.com/HawkAaron/warp-transducer`.

[9] Huy Nguyen Le. Tensorflowasr: Almost state-of-the-art automatic speech recognition in tensorflow 2. `https://github.com/TensorSpeech/TensorFlowASR`.

[10] Rohit Prabhavalkar, Tara N. Sainath, Yonghui Wu, Patrick Nguyen, Z. Chen, Chung-Cheng Chiu, and Anjuli Kannan. Minimum word error rate training for attention-based sequence-to-sequence models. *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4839–4843, 2018.

[11] Tencent. Pika: a lightweight speech processing toolkit based on pytorch and (py)kaldi. https://github.com/tencent-ailab/pika.

[12] Anshuman Tripathi, Han Lu, Hasim Sak, and Hagen Soltau. Monotonic recurrent neural network transducer and decoding strategies. In *2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, pages 944–948, 2019.

[13] Chao Weng, Chengzhu Yu, Jia Cui, Chunlei Zhang, and Dong Yu. Minimum Bayes Risk Training of RNN-Transducer for End-to-End Speech Recognition. In *Proc. Interspeech 2020*, pages 966–970, 2020.

List of symbols and abbreviations

ASR	Automatic Speech Recognition
MWER	Minimum Word Error Rate
RNN-T	Recurrent Neural Network Transducer
WER	Word Error Rate
L_{RNNT}	RNN-T Loss
L_{MRNNT}	Monotonic RNN-T Loss
L_{MWER}	RNN-T MWER Loss
$\varnothing$	Blank Symbol
x	input vector
y	label
t	timestep
u	predictor network step
k	vocabulary element

List of Figures

1 Different alignments of the word 'cat' . . . 13

2 RNN-T architecture . . . 18
3 Output probability lattice . . . 20
4 Monotonic RNN-T step example . . . 22

5 Architecture overview . . . 26
6 Monotonic RNN-T module state diagram . . . 28
7 Beam search module state diagram . . . 29
8 MWER loss module state diagram . . . 30
9 MWER Conformer module state diagram . . . 32

10 Validation (blue) and training (orange) loss value graph on Monotonic RNN-T training . . . 38
11 Validation (orange) and training (grey) loss value graph on MWER training (beam width 1) . . . 39
12 Validation (blue) and training (red) loss value graph on MWER training (beam width 2) . . . 39
13 Validation (green) and training (pink) loss value graph on MWER training (beam width 3) . . . 40
14 Validation (orange) and training (blue) loss value graph on MWER training (beam width 4) . . . 40

List of tables

1 Functional Requirements 15

2 Non-Functional Requirements 16

3 Division of Work 16

4 Test summary 35

5 WER results on the test dataset 41

6 Relative WER improvement compared to baseline model 41

7 Transduction examples 43

List of appendices

1. User Manual

A. User Manual

This appendix describes how to install and run the solution. The user is assumed to have access to a workstation with GPU. The solution was tested exclusively on Linux (Ubuntu), henceforth we assume the user's workstation runs this operating system.

A.1. Installation instructions

The solution may be used directly from the shipped source code or accessed via docker container. Below we present instructions for both of the methods.

A.1.1. Manual installation

1. Install Python.
2. Add current directory to PYTHONPATH

```
export PYTHONPATH="${PYTHONPATH}:${PWD}"
```

3. Install requirements.

```
pip install -r requirements-pre.txt && pip install -r requirements.txt
```

4. Download LibriSpeech test dataset and extract it.

```
wget https://www.openslr.org/resources/12/test-clean.tar.gz

tar -xzvf test-clean.tar.gz
```

5. Create transcriptions for the downloaded dataset.

```
python ./scripts/create_librispeech_trans.py \
-d ./LibriSpeech/test-clean ./LibriSpeech/test_transcriptions/test.tsv
```

6. Change paths in the examples/conformer/config.yml to match directory layout. Specifically edit dataset paths to match locations of transcripts.

A.1.2. Docker installation

The solution may be accessed using a docker environment. The steps listed above in Manual installation instructions are enclosed by a docker container simplifying the installation process.

1. Install Docker.
2. Install docker compose.
3. Install nvidia-docker.
4. Run docker container.

```
docker compose run tensorflow_asr
```

A.1.3. Training

NOTE: Training will fail on majority of consumer tier GPUs due to lack of memory.

First, the training and evaluation data needs to downloaded. This can be achieved by utilizng the following commands.

```
wget https://www.openslr.org/resources/12/dev-clean.tar.gz
tar -xzvf dev-clean.tar.gz

wget https://www.openslr.org/resources/12/train-clean-360.tar.gz
tar -xzvf train-clean-360.tar.gz
```

Then, transcriptions for the training and eval data should be created:

```
python ./scripts/create_librispeech_trans.py \
-d. /LibriSpeech/train-clean-360/data/LibriSpeech/test_transcriptions/train.tsv
python ./scripts/create_librispeech_trans.py  \
-d ./LibriSpeech/dev-clean/data/LibriSpeech/test_transcriptions/dev.tsv
```

In order to start training, under the path *examples/conformer/train.py*, there's a script starting the training process. If user wishes to train with MWER training procedure, in **config.yml** under *model_config* there's a boolean *mwer_training* which, if set to True, starts the MWER training procedure. Otherwise it starts standard training procedure with regular RNN-T loss. *train.py* receives specific arguments for training. Most important are:

- *–config* a path to model **config.yml** file.
- *–subwords* a flag whether to use subwords as text tokenizer.
- *–bs* batch size.
- *–devices* which GPU devices are supposed to be used.

The rest of arguments are described in **train.py** file.

Example of such command (that works under default setup) would be:

```
python examples/conformer/train.py \
--config examples/conformer/config.yml --sentence_piece --devices 0
```

A.1.4. Test

In order to start testing, under the path *examples/conformer/test.py*, there's a script starting the inference process. *test.py* receives specific arguments for training. Most important are:

- *–saved* a path to saved model.
- *–config* a path to model **config.yml** file.
- *–subwords* a flag whether to use subwords as text tokenizer.
- *–bs* batch size.
- *–device* which GPU devices are supposed to be used.
- *–output* path to output transcriptions.

Example of such command (that works under default setup) would be:

```
python ./examples/conformer/test.py --config ./examples/conformer/config.yml \
--saved predefined_checkpoints/weights.hdf5 \
--sentence_piece \
--output test_result.tsv \
--bs 1
```

A.1.5. Demonstration

To run a demonstration on the actual .flac file, from the root directory one needs to run the command:

```
python examples/demonstration/conformer.py \
--config ./examples/conformer/config.yml \
--saved predefined_checkpoints/weights.hdf5 \
--sentence_piece \
--subwords ./vocabularies/librispeech/spm_512 \
--beam_width 1 \
examples/demonstration/wavs/1089-134691-0000.flac
```

This script demonstrates the usage of the model on real world data.