Politecnico di Torino

Faculty of Engineering

Master of Science in
Computer Engineering - Embedded Systems

Energy-Efficient Quality Adaptation for Recurrent Neural Networks

Advisors:
Prof. **Massimo Poncino**

Co-Advisors:
Doc. **Daniele Jahier Pagliari**

Candidate:
Panini Francesco
Matr:242547

22 October 2018

Contents

List of Figures

List of Tables

Abstract

Recurrent Neural Networks - RNNs are state-of-the-art models able to deliver very high accuracy in sequence modeling and machine translation tasks. In particular the Encoder-Decoder architecture excels in sequence-to-sequence tasks in which input and output sequences may not have the same length. These networks work in two stages, at first the input sequence is encoded in a fixed length representation, which is then decoded in order to produce a new target sequence. Due to the abundance of the network parameters, performing inference using these models requires a high computing power and results in large energy consumption, typically unsustainable for an embedded device. While executing the inference on edge nodes is beneficial in terms of latency and responsiveness of the system, generally such nodes do not have the hardware resources needed to sustain the heavy computations involved.

To this end, this work proposes an algorithm to improve the energy efficiency of Encoder-Decoder RNNs. In particular a novel dynamic Beam Search algorithm is introduced, in which the Beam Width - BW is varied according to the evolution of a translation. This method is able to dynamically adapt the Beam Width parameter, i.e. one of the parameter that mostly dominates the inference complexity, according to the currently processed input and the corresponding network's confidence.

Results on two different machine translation models underline that the proposed methodology is able to reduce the average BW by up to 33%, thus significantly reducing the inference execution time and energy consumption, while maintaining the same translation performance.

Chapter 1

Introduction

Machine learning technology is increasingly present and used in many aspects of modern society and everyday life. Tasks like classification, speech recognition and image recognition leverage machine learning to increase the quality of results provided and reduce the effort required by developers [3]. In particular, Recurrent Neural Networks - RNN are now able to deliver state-of-the-art accuracy in sequence modelling and other language tasks such as Neural Machine Translation, Image Captioning and Question Answering [4]. Unlike traditional feed-forward Deep Neural Networks, where there is no notion of correlation between consecutive inputs, RNN are able to both store (remember) information from earlier inputs and handle variable length inputs and outputs thus improving the quality of produced results.
One reason for the growing popularity of machine learning based applications is the increasing availability of computing power. Multicore CPUs and clusters of GPUs have been necessary to sustain the rising computational complexity of these models and to train them in a relatively short period of time [5] [6]. To an increase of computing power demand corresponds an increase of energy consumption and resources needed, both factors that are limited in low-power embedded hardware devices.
Due to the growing demand of intelligent end-nodes, such as IoT sensors or mobile devices, many efforts have been made to implement Neural Networks directly on those devices. While it is better to perform inference *locally*, near acquisition sensors, allowing a sizable reduction of latency and an increased responsiveness of the system [7], the training phase can conveniently take place on *cloud* using high-end cluster computers, where computing power and energy consumption are not an issue [8] [7]. As a result, the power efficient design for Neural Network architecture and the development of optimized inference methods are acquiring a key role for the development of new applications [6] [9].

In literature, many works have studied this problem proposing different solutions. One of those is the implementation of dedicated hardware accelerators for the inference phase. Although these accelerators are able to achieve very good performances (especially for Convolutional Neural Networks - CNNs), nowadays there are very few designs optimized for RNN [9] [10]. Other popular solutions adopted in low-power designs leverage on the approximate computing paradigm, thus treading reduced quality of produced results with lower complexity. "Complexity" can then acquire different meanings according to the context. In relation to network quantization, it is possible to reduce the complexity of computations by reducing the bit-width of the weights with only a small loss in accuracy [6], or in the context of model size, it is possible to lower the complexity by limiting the number of nodes or layers.
In particular, many works propose *static* solutions, in which the configuration of the network is kept constant throughout the whole execution, while only few offer a *dynamic* tradeoff, in which complexity and energy are modulated at runtime [5] [11]. Having a static configuration (e.g. fixed bit-width) can result in a suboptimal solution, since the network can either overapproximate, producing poor results or underapproximate, resulting in a waste of energy. On the contrary, being able to dynamically adapt the network level of precision basing the choice on the input characteristics provides a more effective approach. This is true whenever inputs are not all equally difficult to process, thus they can be treated with different degree of approximation, resulting in a more convenient quality-energy tradeoff [5] [11].
In this work, a similar approach has been taken, but applied for the first time to RNN models, with a specific focus on encoder-decoder networks for Neural Machine Translation - NMT[1]. A new method has been proposed to dynamically tune the Beam Width - BW parameter depending on the currently processed input. Increasing the BW is beneficial for the translation accuracy, but also has a dramatic impact on the computational complexity of the inference task at runtime. Thanks to the proposed method for evaluating the translation confidence and the dispersion of the scores produced after each iteration, the input-dependent tuning approach described in this thesis can be effectively used to dynamically change the Beam Width according to the state of the translation process. This allows to reach comparable or even better results with respect to an execution of the model with fixed beam width, but with a lower beam width on average. Considering a single-threaded software implementation of the considered RNNs, which is the common scenario for an embedded

[1] Application of Natural Language Processing in which the model learn to read the input sentence in one human language and emit the translated sentence in another human language.

device, obtained results translate into a 25% reduction of the total execution time required for inference, which translates into significant energy savings.

Chapter 2

Background

2.1 Overview

Thanks to their ability to address large-scale problems and the large availability of big dataset, Deep Neural Networks (DNNs) have gained increased attention from the machine learning and data analysis communities [5]. Since the extensive development of DNN applied to many task such as image classification or speech recognition, the number of DNN based applications have exploded.
In the past, pattern-recognition systems were limited in their ability to process raw data to extract valuable information, since they required careful engineering and a broad domain expertise to design an effective feature extractor [3]. Current machine learning techniques and in particular Neural Network models, leverage *representation learning*. This set of techniques allow machines to automatically "learn" (or discover) the representations needed for classification or detection. So called deep models are composed of several of these data representation transformations (*layers*) stacked together. each layer is able to transform representation coming from lower level layers into a more abstract representation needed from the following higher level layers, allowing the creation of a more complex classification function. [3]
The founding pillar of these models is the concept that the feature layers composing the network are not designed by humans, but rather, they are learned from the input dataset, after a training phase.

During the years, different Neural Networks models have been introduced to solve specific problems. Each model has its own field of application, in the following the most common architectures are presented as well as their fields of application:

- Feed-Forward Neural Network: is the most commonly used architecture. It is composed of: one input layer, one output layer and one/more hidden layers (if more than one hidden layer, it is called Deep neural network). Data flows from the input to the output without feedback. During this path, the network computes a series of transformations, such that each subsequent layer holds a more abstract representation for the input. This architecture is mainly used for classification problems.
- Convolutional Neural Network: is a specialization of Feed-Forward Neural Network, mainly used for image processing. It inherits similar structure and working principle from Feed-Forward, but adds some specialized layers, such as convolutional and pooling.
- Recurrent Neural Network: is the most commonly used architecture for analyzing sequential data. It has a feedback connection, allowing the network to maintain a "memory state", i.e. a representation of the part of the sequence it has already analyzed. It enables the network to take decision based both on current input and previous history.

In the following sections, a more detailed explanation of each architecture is provided: section 2.2 will give more details on Fully Connected Neural Network, section 2.3 will explain Convolutional Neural Network and finally, section 2.4 will deepen on Recurrent Neural Network.

2.2 Fully Connected Neural Network

Fully Connected Neural Network or Feed-Forward Neural Network is the simplest neural network based classificator. Its goal is to learn a representation function f, which will then be used for classification tasks. Given a classifier $y = f(x, \theta)$, the network performs the mapping of an input $\mathbf{x}$ to an output category $\mathbf{y}$, through the network parameters θ [12].
This architecture is called Feed-Forward because data flow from input to output and fully connected because each *neuron* is connected to all neurons outputs from the previous layer. A neuron is a generic computational unit performing a weighted sum over all its input and applying a non-linear squashing function (also called the neuron activation function) to produce a single output [13]. Figure below shows the structure of a neuron:

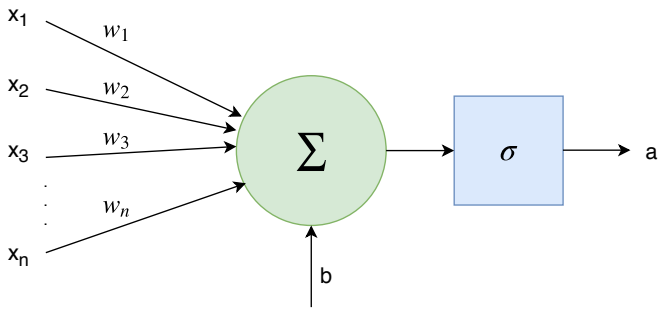


Figure 2.1: Neuron structure

Figure 2.1 shows how an input vector $\mathbf{x}$ is first scaled by the weight matrix $\mathbf{w}$, summed, added to a bias and then passed through a non-linear squashing function. This function is needed to implement non-linear classification function and also to keep the neuron output value within a defined range. Each neuron activation (i.e. output) value indicates the presence or absence of a particular feature. Commonly used activation functions are listed below:

- Sigmoid σ, in the early stages of Neural Network development, was the most common and widely used activation function. $\sigma(z) \in (0, 1)$

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

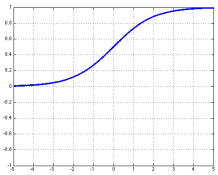


Figure 2.2: Sigmoid activation function and graph respectively

- Tanh is an good alternative to Sigmoid since it has been found that it converges faster [13]. $Tanh(z) \in (-1, 1)$

$$\tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)} = 2\sigma(2z) - 1$$

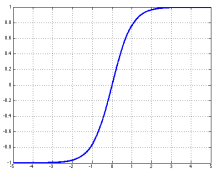


Figure 2.3: Tanh activation function and graph respectively

- ReLU - Rectified Linear Unit is an activation function which does not saturate, even for large value of the input. It is mostly used in Convolutional NN.

$$\text{rect}(z) = \max(z, 0)$$

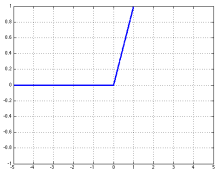


Figure 2.4: ReLU activation function and graph respectively

The activation function for the output layer, instead, may changes according to the task. As an example, for a multiclass classification problem, with K possible alternative classes, the most common activation function is the *softmax* [14]. This function takes the output of all neurons in the last hidden units and return K real values, where each entry is in range (0,1] and all entries sum up to 1. This output represent the output probability distribution of the input $\mathbf{x}$ to belong to one of the K output classes.
Many neurons grouped together create a layer and each layer implements a different representation function [12]. Many of these layers are then stacked up, having the

data flowing out form one layer being the inputs for the next one. Upon receiving a new input, the network performs a feed forward computation, data change representation while moving from one layer to another, increasing the number of features detected, until the last layer is reached, which performs the final classification. The model can be associated to a direct acyclic graph - DAG, showing how data flow and how functions are connected together into a chain [12]. The length of the chain, and so the number of functions, gives the **depth** of the model. First and last layers are called **input** and **output** layer, while layers in the middle are called **hidden** layers. The dimension of each layer, and so the number of neurons it is made of, gives the **width** of the model.

The most common form of training is *supervised learning* [3]. At first a large dataset of what the network has to be trained on is collected. In general, the more data are provided to the network, the more accurate will be the classification. The dataset must also be provided with the label corresponding to each datum. During the training phase, the network is shown an input and it produces a vector of scores (K scores, one for each category). The highest score defines the class the input belongs to, according to the network guess. The desirable output is the class corresponding to the input label, but it is unlikely to happen in the early stages of training. In order to be effectively trained, the network needs an optimization objective, a *cost function* $\mathcal{L}$, that measure the error done while guessing. Most neural networks are trained using maximum likelihood (described also as the cross-entropy between the model output distribution and the training data) as cost function [12]. This means that this cost function penalizes the differences between the model output $\hat{\mathbf{y}}$ and target $\mathbf{y}$. Learning is then accomplished by iteratively updating the weights and biases to minimize this loss function [14]. Due to the non linearity of the classification function, the loss function becomes non-convex. This means that to effectively train the network gradient-based optimizers are needed [12]. In order to calculate the gradient with respect to each parameter involved in the forward pass, **backpropagation** is used. This technique allows to apply the chain rule to calculate the loss function gradient, starting from the output layer till the input layer, backpropagating the error. Then, weights and biases are adjusted using gradient descent[1]. This method explores the cost function in order to find a local minimum,

[1]In calculus, the gradient of a function is a multi-variable generalization of the derivative. It is a vector that can be used to identify the fastest way to increase the function, following the shortest path. Computing the *negative* gradient with respect to all weights, the network can find the best set of values for its parameter (w and b) in order to minimize the cost function.

but does not guarantee to find the optimum solution.
As an example, the steps to calculate the gradient for very simple FeedForward NN using backpropagating are explained. The network considered is composed of only four layers, each one having a single neuron, as shown in Figure 2.5. At first, an input is fed in and propagated through the network in order to produce an activation a^{L-i} for each neuron in the hidden layers, and a prediction $\hat{\mathbf{y}}$ at output layer (forward pass).

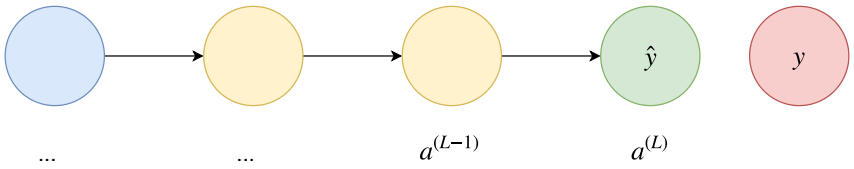


Figure 2.5: Simple network for backpropagation example

The output of this network depends on the value of three weights and three biases, and so the cost function is also a function of these values, $\mathcal{L}(w_1, b_1, w_2, b_2, w_3, b_3)$. Given the label for the activation of the output layer $a^{(L)}$ (which also correspond to the network final guess $\hat{\mathbf{y}}$), activations belonging to previous layers will be labeled with $a^{(L-i)}$. It is important to recall that each neuron activation corresponds to the application of a non-linear squashing function (e.g. sigmoid) to the weighted sum of all previous layer neuron outputs, in this case only one:

$$z_j^{(L)} = \sum_i (w_{ji} x_i) + b_i$$
$$z^{(L)} = w^L a^{L-1} + b^{(L)}$$

$$a^L = \sigma(z^{(L)})$$

Given the expected output value y (the label corresponding to the input example), the cost function will be $\mathcal{L}(y, \hat{y})$.

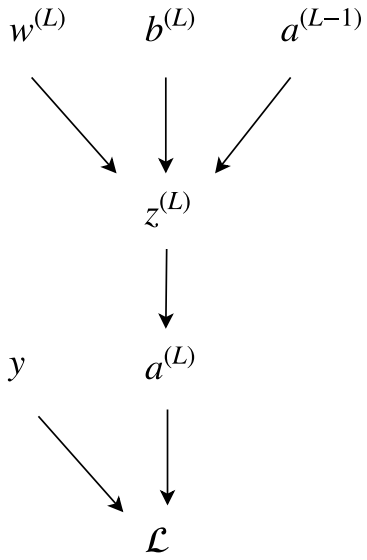


Figure 2.6: Backpropagation contribution

Focusing on just the connection between the last two neurons, the "sensitivity" of the cost function with respect to a small change in the weights $w^{(L)}$ can be expressed as follows:

$$\frac{\partial \mathcal{L}}{\partial w^{(L)}} = \frac{\partial \mathcal{L}}{\partial a^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial z^{(L)}}{\partial w^{(L)}}$$

This is the derivative of the cost function with respect to $w^{(L)}$ only and for a specific training example, since the full cost function involves averaging together all those costs across all training examples. The same procedure is then applied to calculate the sensitivity of $\mathcal{L}$ with respect $b^{(L)}$ and $a^{(L-1)}$, and then iterated back to previous layers. This is the chain rule applied, where the gradient with respect to a layer output is used to compute the gradient with respect to the network parameters for previous layers.

In practice, neural networks are usually trained with Stochastic Gradient Descent - SGD using minibatches [12]. Instead of analyzing the cost function over the entire training set (as in the standard Gradient Descent), only a small subset of it is used for each iteration. For each minibatch (set of few inputs), the network computes the

output error, then compute the average gradient related to those inputs, then adjust the weights accordingly [3]. This process will not give the actual gradient of the cost function (which depends on all training data), but rather a good approximation of it and also results in a significant computation speedup. With unit batch size, the update equation for stochastic gradient descent results:

$$w \leftarrow w + \eta \bigtriangledown_w \mathcal{L}_i$$

where η is the learning rate[2] and $\bigtriangledown_w \mathcal{L}_i$ is the gradient of the cost function with respect to parameter **w**.

[2]It defines how fast the network changes its weights, and it can be visualized as the "length of the leap" while descending the cost function. Big learning rates correspond to big adjustments in the parameters, which may resolve in overshooting and missing the minimum. Conversely, small learning rates correspond to small adjustments in the parameters and a very slow training.

2.3 Convolutional Neural Network

Convolutional Neural Networs are a specialization of Feed-Forward Neural Networks, which have reached state-of-the-art performances in image processing tasks [5]. They inherit similar structure and working principle from Feed-Forward, but add some specialized layers. Even though the literature offers many possible layers, the *convolutional, pooling* and *normalization* are the most commonly used layers [7]:

- The convolutional layer performs a set of matrix multiplications between the input (input feature map), and a set of kernels, also called *filters*, designed to identify specific features. Given an input image of size $m \times m$, a filter is defined as a $n \times n$ set of weights (where $m > n$). Each filter is lined up to a patch of the input image and then, one by one, the pixels are processed. The *filtering* operation is a multiply and accumulation operation between the patch pixels and the filter weights, the final result is then divided by the number of pixels in the filter. Then, the *convolution* operation is performed by repeating this process for all possible positions, sliding each time the filter to a new patch in the image. The result is an output feature map containing an higher level abstraction of the input and a summary of where features are in the image.
- In the pooling layer, a window size of $l \times l$ (where $m > l$) and a stride of p pixels are defined. Max poling (the most used technique of pooling) is performed by striding the window across the filtered image and taking the maximum value inside the window. This operation is meant to shrink down the feature map and to make it less sensible to spatial position, such that the network is still able to detect the same features, but also in different parts of the image [12].
- The normalization layer is composed of ReLU activation functions. This function is used to limit the computation complexity by removing useless values (all negative values) from the feature map.

Figure 2.7 shows typical network, made of an alternation of the convolutional layer, followed by the normalization and a pooling layers, stacked and repeated. The last stage of the network, instead, consists of one/many fully connected layers to perform the actual classification, given the features extracted before.

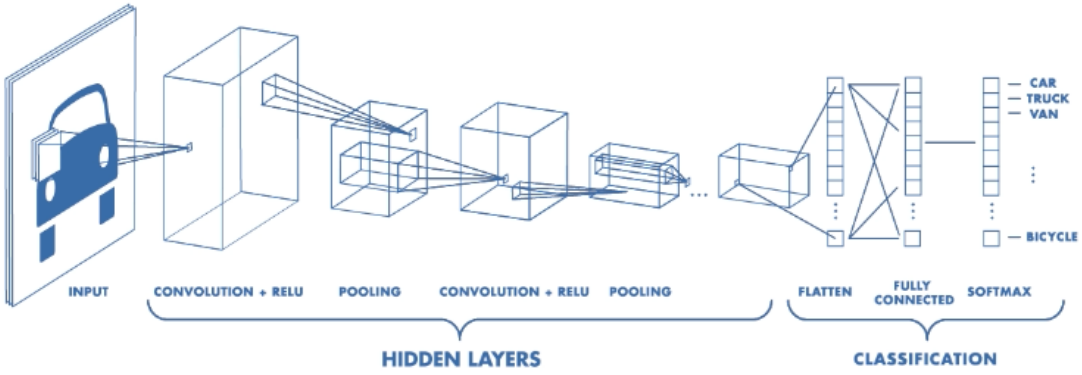


Figure 2.7: Architecture of CNN, source: https://.it.mathworks.com/videos/introduction-to-deep-learning-what-are-convolutional-neural-networks

It is visible how, starting from the input image (on the left), filters are applied followed by the normalization (not sown in the figure) and pooling layers. In the process, the images size is reduced after each layer, finally being flattened to be used in the last fully connected layer(s).

2.4 Recurrent Neural Network

2.4.1 Sequence Definition

In the context of Neural Networks, and in particular during the training phase, datasets play a fundamental role. The quantity, but also the quality, of the input data used for training these models will define, at inference time, their accuracy. After all the samples have been collected, it is common practice to divide them in three different dataset, i.e. training, validation and test sets, each of those has different characteristics and usages. Each set represents a collection of data points, generally an (input, target) vector pair. During the execution of the network, it is not important which one of these pair is fed first into the network, since elements in the set are independent from each others. However, this is not the case for RNNs, where inputs and outputs are elements of a sequence, and thus intrinsically correlated to each other. So, while working with a specific sequence, it is important to respect the order in which data appear, either when samples are fed in or when results are read out.
An input sequence is generally denoted with $(x_1, ..., x_T)$, where each data point $x_{(t)}$ is a value, generally a vector, i.e. word embedding, within the sequence at a particular instant in time. Similarly, the target sequence can be denoted $(y_1, ..., y_T)$. Even though RNNs are not bounded to work only with time-based sequences, their most common applications involve sequences with implicit or explicit temporal dependence.

Different sequence-related task could be:

- Seq. Prediction: involves predicting the next item in the sequence, given the previous data points. Examples of seq. prediction problems are: weather forecast, stock marcket prediction and product recommendation.
- Seq. Classification: involves predicting the target class label once the full sequence has been observed. Examples of seq. classification problems are: DNA sequencing and anomaly detection.
- Seq. Generation: involves producing a new sequence that mimics the characteristics of the training sequences. Examples of seq. generation problems are: text generation and music generation.
- Seq. to Seq. Prediction: involves predicting an entire output sequnce upon

having observed the entire input sequence. Examples of seq2seq problems are: neural machine translation and text summarization.

2.4.2 Vanilla RNN

The founding pillar of sequence modelling tasks is *language modelling*. This problem can be summarized as to compute the probability of occurrence of a number of words in a particular sentence [1]. Given a sentence $(w_1, w_2, ..., w_T)$, of length T, its probability is denoted as $P(w_1, w_2, ..., w_T)$ and can be expressed as:

$$P(w_1, w_2, ..., w_T) = \prod_{i=1}^{i=T} P(w_i | w_1, ..., w_{i-1}) \tag{2.1}$$

Equation 2.1 is especially useful in translation systems, where having a way to estimate the relative likelihood of different phrases is essential to generate a good quality translation. Language models are generative, meaning that, once they have been trained, they can be used to generate a sequence of data by feeding the previous model output back as new input. Equation 2.2 below shows this concept:

$$x^t = f(x^{t-1}) \tag{2.2}$$

where the sequence element x^t, at current time step t, is predicted by the mapping function f on the base of the element x^{t-1}, at previous time step *t-1*. In the context of RNNs, f represents the neural network which predicts the next element in the sequence based on the current element of the sequence.

In this section the Vanilla RNN will be introduced as it is the simplest RNN architecture. It produces an output (next value in the predicted sequence) at each time step and have a single recurrent connections between hidden units. In this architecture input and output sequence have the same length. More advanced and powerful architectures will be presented later in the discussion, in subsection 2.4.6 subsection 2.4.7.

Graph Unfolding

To describe in a more formal and precise manner the set of operations the network is performing and its evolution through time, it is necessary to introduce the **computational graph notation**. Each network can be visualized with its

corresponding *computational graph*, in which nodes represent network parameters, operations are performed on these parameters and finally arrows represent the interconnections between them. Graphs for standard architecture such as FeedForward Neural Networks or CNNs, generally are directed acyclic computational graph, in which data flows from input to output passing through different types of operations. That is not the case for RNN. As anticipated before, RNN are used to represent sequence of events correlated by time and so, they need to keep (memorize) information coming from previous states.
To express the intrinsic dynamism of this kind of system, each network state $s^{(t)}$ can be expressed as a function of time, in particular:

$$h^{(t)} = f(h^{(t-1)}, \theta) \tag{2.3}$$

where θ represents state parameters. Equation 2.3 is recurrent because the definition of the state **h** at time **t** depends to the same definition, but at time (t-1) [14]. Now, it is possible to unfold the recurrent computation by explicitly writing the Equation 2.3 applying repeatedly the definition, yielding an expression without recurrence. This, in turn, can be represented with a traditional acyclic computational graph with a repetitive structure, corresponding to a chain of events and shared parameters across states.
This behavior is illustrated in the Figure 2.8 below. On the left, a classical representation of the folded network, in which time dependence is introduced in the system by the inclusion of the edge looping on the network hidden state. On the right, instead, the same network is represented as unfolded computational graph, where each node is now associated with one particular time instance.

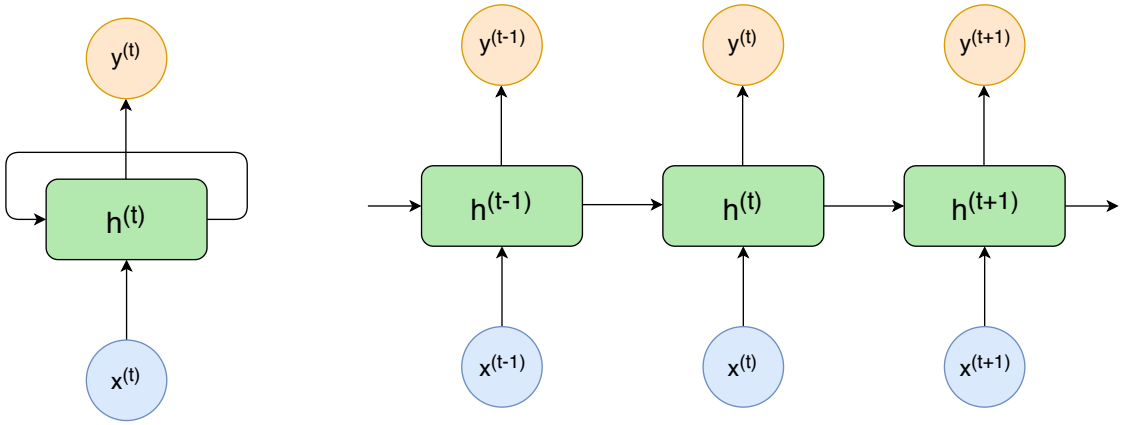


Figure 2.8: Recurrent Computational Graph

It is important to notice that the arrows connecting the states represent the function **f** in Equation 2.3 which maps the state t to the state at $t + 1$, and same parameters θ are used for all time steps.
Finally, the unfolded graph also helps to better visualize the set of computations performed by the network and the paths on which information flow forward (during output prediction and the update of the hidden stage) and backward (during back-propagation) [12].

Hidden Units

Figure 2.8 in the previous section introduces the structure of a basic RNN, where each rectangular box refers to a, so called, **hidden state** of the network at a specific time step. Each of these states refer to a set of neurons, each of which performs linear matrix multiplications on input data, followed by the application of a non-linear function σ. At each time step t, neurons will receive as input data the current data point $x^{(t)}$ and previous hidden state $h^{(t-1)}$ and produce an output prediction $\hat{y}$ for the next value in the sequence as well as a new hidden state $h^{(t)}$ (Equation 2.5, Equation 2.7) [14].

$$a^{(t)} = W^{hx}x^{(t)} + W^{hh}h^{(t-1)} + b_h \tag{2.4}$$

$$h^{(t)} = \sigma(a^{(t)}) \tag{2.5}$$

$$o^{(t)} = W^{yh}h^{(t)} + b_y \tag{2.6}$$

$$\hat{y}^{(t)} = softmax(o^{(t)}) \tag{2.7}$$

A detailed description of all parameters is presented below:

- $x_1, ..., x_t, ..., x_T$: is the word vector corresponding to a sequence of T words
- $h^{(t)}$: is the hidden state output feature for the time step t
- W^{hx}: is the weight matrix used to condition the input word x_t
- W^{hh}: is the weight matrix used to condition the the previous hidden state
- W^{yh}: is the weight matrix used to compute the output prediction

- b_*: is the biasing term
- σ: is the non-linear function (e.g.tanh())
- *softmax*: is needed to compute the output probability distribution over the vocabulary

2.4.3 Training

Training a RNN generally means learning how to use the hidden state h^t, as a lossy summary of important aspects of the input sequence seen till this time, for predicting the next element in the sequence based on the input sequence past history. This summary is intrinsically "lossy" since it has to map an arbitrary-length sequence $(x_1, ..., x_t, ..., x_T)$ to a fixed length vector h^t.
Given the Graph Unfolding property, explained in section 2.4.2, the network can now be treated as a standard FeedForward network with one layer per time step and with shared weights matrices (propagated from one time step to the next). So, it can be trained by simply applying **recursively** the back-propagation algorithm, already explained in section 2.2. Backpropagation applied to the unrolled version of the network is called **back-propagation through time - BPTT**.

During the forward pass, the sequence of steps involved are presented below:

- input x_t is fed into the network
- compute the hidden state Equation 2.5
- compute the output prediction $o^{(t)}$ Equation 2.6
- compute the softmax on $o^{(t)}$ to obtain a normalized probability distribution
- compute loss function $\mathcal{L}^{(t)}$

Given input sequence $\mathbf{x}$ with corresponding expected outputs $\mathbf{y}$, the total loss function can be computed as the accumulative sum of all losses for each time step. If $\mathcal{L}^{(t)}$ is the negative log-likelihood[3] of $y^{(t)}$ given $x_1, ..., x_\tau$, then:

$$\begin{aligned} &\mathcal{L}(\{x_1, ..., x_\tau\}, \{y_1, ..., y_\tau\}) \\ &= \sum_t \mathcal{L}^{(t)} \\ &= -\sum_t \log p_{model}(y^{(t)} | \{x_1, ..., x_\tau\}) \end{aligned} \tag{2.8}$$

where $p_{model}(y^{(t)} | \{x_1, ..., x_\tau\})$ is given by comparing model predicted outputs $\hat{y}^{(t)}$ and expected outputs $y^{(t)}$.

[3]The cross-entropy between the output model distribution and the training data [12]. Log-likelihood is more convenient with respect the classical likelihood, since working in the log space helps reducing precision problems. These problems arise when multiplying words probability, which can be very small numbers, the results can rapidly vanish to zero [15]

When using predictive log-likelihood as training function, as in Equation 2.8, the network has to maximize the conditional distribution of the next sequence element $y^{(t)}$ given the previous inputs and also the outputs of previous time steps[4]

$$\log\left(y^{(t)}|x_1, ..., x_t, y_1, ..., y_{t-1}\right) \tag{2.9}$$

Equation 2.3 shows that, in principles, RNN are very efficient in parametrizing long-term relations due to the recurrent application of the advancing function f over the same parameters θ. In fact, information about the context already seen are propagated through time by continuously updating the hidden state $h^{(t)}$, which acts as a back-up memory of previous states. The downside of this architecture is that computing the gradient for the loss function (in Equation 2.8) to update model parameters is a very expensive operation. This computation involves performing back-propagation through the whole unfolded graph [12]. Moreover, due to the sequentiality of the system, this computation cannot be parallelized, and it must be computed step by step.

It is important to notice, that even if the mathematical formulation of the hidden state could be theoretically applied recursively as many times as are needed for an arbitrary length sequence, empirical proves have shown that RNN are more likely to correctly predict short sequences. This problem is known as **vanishing and/or exploding gradient** [1]. This phenomenon depends on the magnitude of the weights matrix in the recurrent edge W^{hh} and on the used activation function [14]. If $|W^{hh}| > 1$, the gradient values will grow extremely fast causing overflow, this is the so called *Gradient Explosion* and it can be easily recognized at train time. On the other side, if $|W^{hh}| < 1$, the gradient will tend to zero, this is the so called *Gradient Vanishing*. This behavior makes training more difficult, and drastically reduces the quality of the parameter learned..
The techniques commonly used to deal with these problems are listed below:

- **gradient clipping:** this is a simple heuristic solution, introduced at first by Thomas Mikolov, and it is commonly used to prevent gradient explosion. Whenever the gradient is going to reach a certain threshold, it is set back to a smaller value.
- **TBPTT:** in truncated BPTT there is a fixed amount of possible steps through which the error can be propagated. This solution aims to mitigate the explod-

[4] either expected outputs (if Teacher forcing technique is used), or predicted outputs

ing gradient problem, but it also reduces at prior the ability of the network to learn long-range temporal dependences.

- **Matrix initialization:** this solution may not solve the vanishing problem, but it tryes to prevent it. Instead of randomly initialize the hidden weights matrix, it is set as the identity matrix.
- **ReLU:** instead of using the sigmoid activation function, it is preferred to use the ReLU, whose derivative is either 0 or 1. This allows the gradient to propagate back through many time steps without being attenuated.
- **LSTM/GRU:** the LSTM or GRU architectures described in the sections below, are designed and optimized to learn long-range temporal dependences without the vanishing gradient problem.

2.4.4 LSTM

In the following, it is presented the **Long Short Term Memory - LSTM** structure, which is an extension of the basic hidden unit. This architecture provides a more efficient way to learn long-term dependencies, and in addition, it also presents a good way to overcome (or mitigate) the vanishing gradient problem presented in subsection 2.4.3.
In addition to the ordinary hidden state $h^{(t)}$, LSTMs also include a **memory cell** c. This internal memory plays a fundamental role in extending the capability of the network in learning long-term dependencies. Thanks to its self-connected recurrent edge with a weights matrix fixed at one, it allows the gradient to pass unconditionally across many time steps without suffering neither of vanishing nor exploding. In addition, c, being continuously updated during the evolution of the network, allows to preserve valuable information that may not be strictly needed in the current time step, but that can be useful for future predictions. This method is more efficient with respect to previous RNNs architectures, i.e. Vanilla RNNs, in which long-term dependencies were leaned in the form of plain weights matrices.
LSTM architecture is comped of special "modules" designed to allows information to be gated-in when need to enrich learned context, gated-out to contribute at the generation of a new output prediction, or remain untouched in the intermediate period when the gate is closed. So, according to the values of these gates, the information can pass through, be completely blocked or be attenuated. Each gate performs an affine transformation (linear matrix multiplication), followed by a logistic function (e.g. sigmoid) to squash the output values in the [0,1] interval, and finally an element-wise multiplication. The output values of the sigmoid layer defines how much of each component can pass through: 0 means "closed gate", while conversely 1 means "full gate open", and values in the middle define the level of attenuation for that parameter. The sigmoid followed by the element-wise multiplication results in the "gating effect" [15].

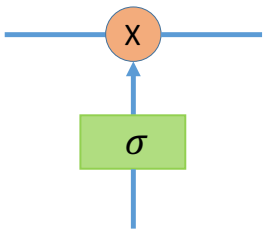


Figure 2.9: Gate

Figure 2.10 below shows the basic blocks composing an LSTM cell

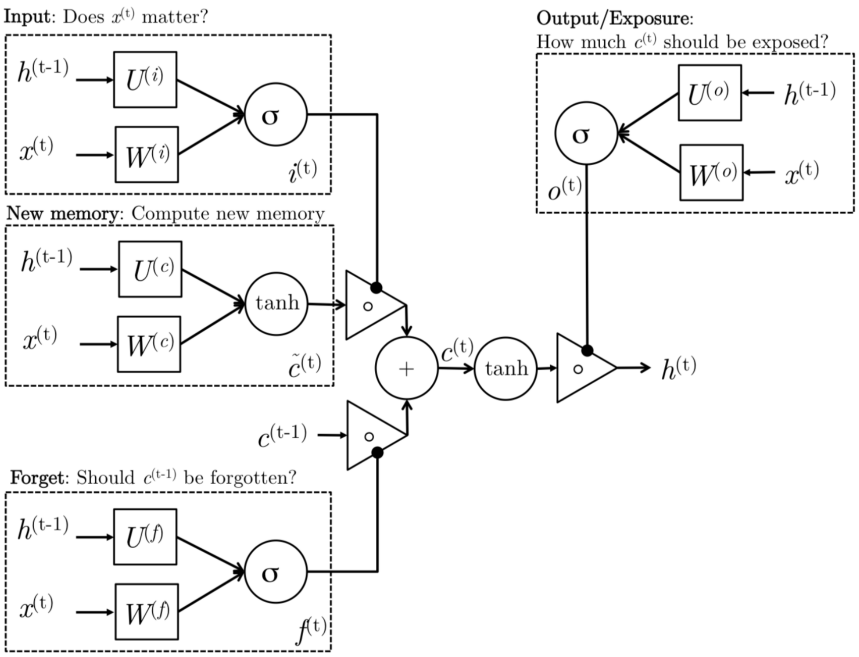


Figure 2.10: LSTM architecture, picture taken from [1]

A detailed description of all nodes is presented:

- Input gate: $i^{(t)} = \sigma(W^{(i)}x^{(t)} + U^{(i)}h^{(t-1)})$
- Forget gate: $f^{(t)} = \sigma(W^{(f)}x^{(t)} + U^{(f)}h^{(t-1)})$
- Out gate: $o^{(t)} = \sigma(W^{(o)}x^{(t)} + U^{(o)}h^{(t-1)})$
- New memory: $\tilde{c}^{(t)} = tanh(W^{(c)}x^{(t)} + U^{(c)}h^{(t-1)})$
- Final memory: $c^{(t)} = f^{(t)} \circ \tilde{c}^{(t-1)} + i^{(t)} \circ \tilde{c}^{(t)}$
- Hidden state: $h^{(t)} = o^{(t)} \circ tanh(c^{(t)})$

Input gate: It takes input data $x^{(t)}$, previous hidden layer state $h^{(t-1)}$ and determines if the input word is important or not for the current time step. This information is then used for gating the new memory $\tilde{c}^{(t)}$.
Forget gate: It looks at the current input word $x^{(t)}$ and previous hidden state $h^{(t-1)}$ and asses if the past memory is useful to update the new memory. This gate allows the cell to forgot information which are not useful anymore.
Out gate: It separates the internal memory $c^{(t)}$ from the hidden state. Since the

memory unit contains information that could not be necessary for updating the hidden state at the current time step, this gate decides which parts of $c^{(t)}$ have to be exposed/passed to $h^{(t)}$.

New memory: It takes input data $x^{(t)}$, previous hidden layer state $h^{(t-1)}$ and generates a new vector summarizing the new input word and the context observed till the current time step.

Final memory: According to what input gate and forget gate decide is important, it collects information coming from both input and past memory.

Hidden state: It is the value which will be used in the downstream calculations (e.g. word probabilities calculation). It is derived by the value of the newly updated final memory $c^{(t)}$, scaled in the [-1,1] interval by the tanh function, and modulated by the output gate.

2.4.5 GRU

A simpler variant of LSTM, but still very effective in remembering long-term dependencies, is the **Gated Recurrent Unit - GRU** architecture. Figure 2.11 below shows a standard implementation:

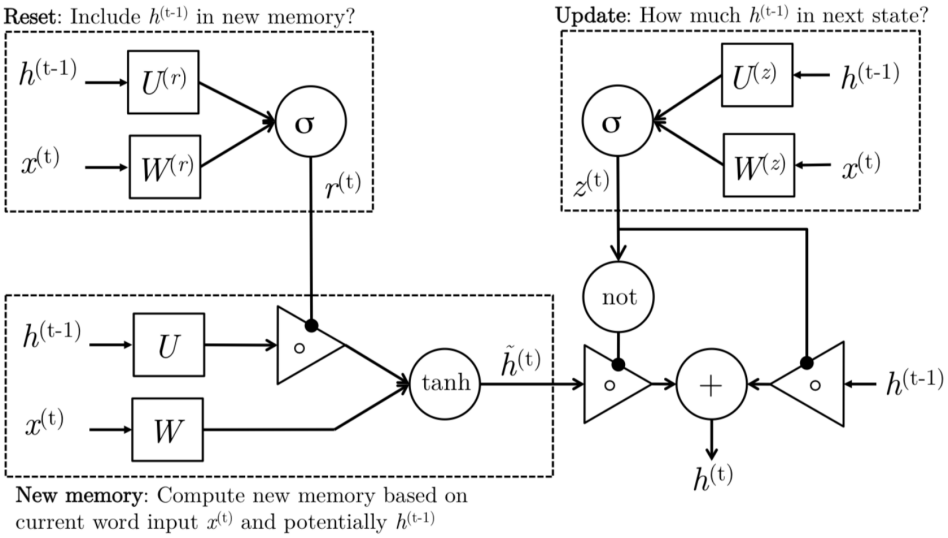


Figure 2.11: GRU architecture, picture taken from [1]

A detailed description of all nodes is presented below:

- Update gate: $z^{(t)} = \sigma(W^{(z)}x^{(t)} + U^{(z)}h^{(t-1)})$
- Reset gate: $r^{(t)} = \sigma(W^{(r)}x^{(t)} + U^{(r)}h^{(t-1)}$

- New memory: $\tilde{h}^{(t)} = tanh(r^{(t)} \circ U^{(t)} h^{(t-1)} + W^{(t)} x^{(t)})$
- Hidden state: $h^{(t)} = (1 - z^{(t)}) \circ \tilde{h}^{(t)} + z^{(t)} \circ h^{(t-1)})$

Update gate: It is responsible for assessing what should be written in the new hidden state $h^{(t)}$. With $z^{(t)} \approx 1$ the previous hidden state will be copied almost entirely out to $h^{(t)}$, while with $z^{(t)} \approx 0$ mostly the new memory $\tilde{h}^{(t)}$ will be forwarded to the next hidden state.
Reset gate: It asses how the previous hidden state $h^{(t-1)}$ is important with respect to $x^{(t)}$ for the computation of the new memory $\tilde{h}^{(t)}$
New memory: It is the combination of the new input $x^{(t)}$ with the previous hidden state (gated by the reset gate). It represents the summary of the previously learned context.
Hidden state: The hidden state will be finally updated with the previous hidden state and the new generated memory, both gated by the update gate.

2.4.6 Bidirectional RNNs

So far, the proposed RNNs architectures always took information from current and previous inputs $x_1, ..., x_{t-1}, x_t$ and, if present, from the past $\mathbf{y}$ values to predict the next element in the sequence.
However, in many applications, to achieve a better accuracy while producing the output prediction $\hat{y}^{(t)}$, it is preferable to have information about the *whole input sequence*. These tasks involve speech recognition, handwriting recognition and many other sequence-to-sequence learning tasks in which the correct interpretation of the current input depends also on the *next* few elements in the sequence [12].
Bidirectional RNNs were invented to address this need. The architecture includes two independent layers of a traditional RNN, one for the left-to-right propagation (forward in time, starting from the beginning of the sentence) and one for the right-to-left propagation (backward to time, starting from the end of the sentence). At each time step, the predicted output $\hat{y}^{(t)}$ is generated by combining both layers's scores, such that the network can leverage on a summary of both past and future.
It is worth noticing that, to maintain and execute such a structure, the network would require twice as much memory, weights parameters and therefore consumes more power with respect to a regular RNN [1]. The Figure 2.12 below shows a standard bidirectional RNN architecture, while Equation 2.10 describes its mathematical formulation

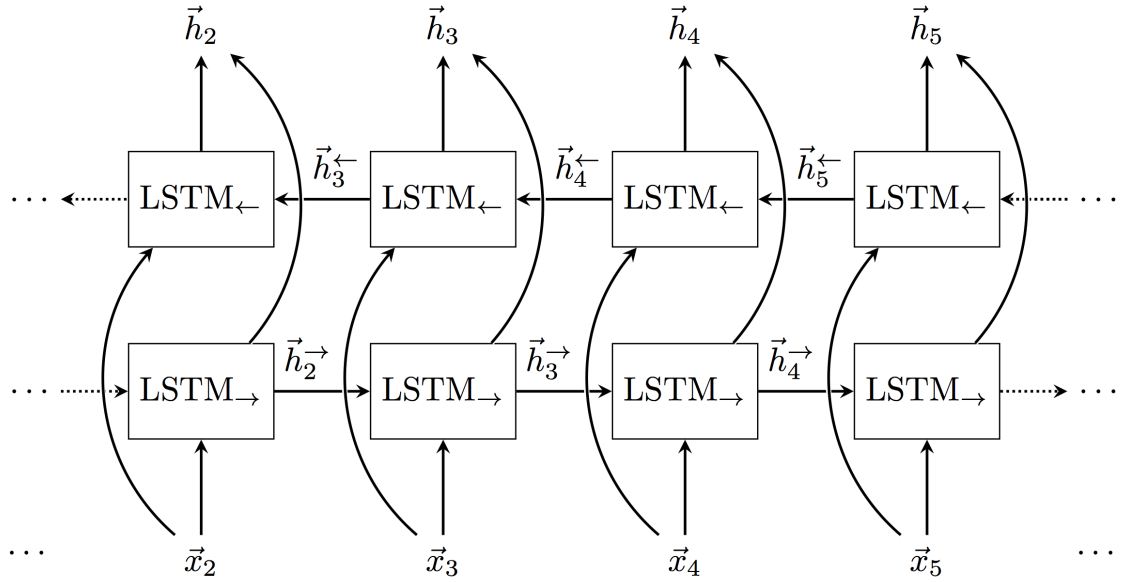


Figure 2.12: Bi-directional RNN architecture

$$\overrightarrow{h^{(t)}} = f(\overrightarrow{W}x^{(t)} + \overrightarrow{V}\,\overrightarrow{h}^{(t-1)} + \overrightarrow{b}) \tag{2.10}$$

$$\overleftarrow{h^{(t)}} = f(\overleftarrow{W}x^{(t)} + \overleftarrow{V}\,\overleftarrow{h}^{(t+1)} + \overleftarrow{b}) \tag{2.11}$$

$$\hat{y}^{(t)} = g(Uh^{(t)} + c) = g(U[\overrightarrow{h^{(t)}}; \overleftarrow{h^{(t)}}] + c) \tag{2.12}$$

The only difference in the equations describing the hidden layers is the *direction* of the propagation through time.
More complex and so more powerful BRNN can be achieved by stacking many bidirectional layers. The output of a lower tier layer will be the input of the next higher tier layer.

2.4.7 Encoder Decoder Architecture

Architectures discussed until now are able to either map a fixed-length input to an output sequence of the same length or, given an input variable-length sequence, can learn how to map it into a fixed-length vector representation. Encoder-Decoder architecture is the state-of-the-art model for solving general sequence-to-sequence problems in which input and output sequences may not have the same length[16]. This approach is very useful in many applications such as speech recognition, question answering and machine translation [12].
The general idea behind the **Encoder-Decoder** architecture is to use two distinct RNNs that act as an encoder and a decoder pair [17]. The **Encoder** reads step by step the input sequence $X = x_1, ..., x_t, ..., x_T$ summarizing it into a fixed-length representation, generally called *context* - *C*. Then the **Decoder**, conditioned with the context vector, is used to produce the output sequence step by step.
Internally, these two networks are generally composed by one or more layers of neurons, each of which can be either implemented as *Vanilla* RNN, *GRU* or *LSTM*. The latter two models, as already explained in subsection 2.4.4, has the capability of efficiently learning long-term dependencies, thus making these models a natural choice for this application [16]. Furthermore, to enhance the capabilities of the Encoder-Decoder architecure, more complex implementations involve stacked Bidirectional RNN and the adoption af an *attention layer*, which will be discussed in the following subsection 2.4.8.
Figure 2.13 below shows a standard implementation of the Encoder-Decoder architecture:

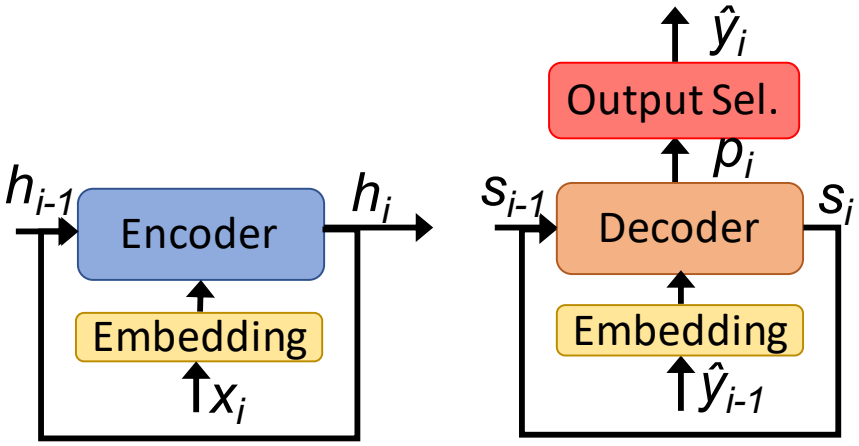


Figure 2.13: Encoder-Decoder RNN architecture

The goal of this model is to learn the conditional distribution $p(y_1, ..., y_{T'}|x_1, ..., x_T)$, where $(x_1, ..., x_T)$ is the input sequence of length T, and $(y_1, ..., y_{T'})$ is the output sequence of length T' (should be noted that T and T' may differ) [16]. The *Embeddings* layers, represented by yellow boxes, are commonly used in Neural Machine Translation in input to both the Encoder and Decoder. These layers allow to create a more compact representation of the input vocabulary (with respect to previously used Bag of Words approach or one-hot encoding) while allowing to extract sematic meanings and contextual similarities (i.e. words that are semantically similar, or that occur commonly nearby in text will also be nearby in the vector space).
To perform the translation of a sentence (commonly referred also as seq2seq mapping), many iterations of both the Encoder and the Decoder are needed. At first the Encoder reads all input words, one symbol at a time, till the end-of-sequence is reached (marked by the special <EOS> character). As the process iterates forward in reading the input sequence, the hidden state is updated accordingly, as a standard RNN:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}) \tag{2.13}$$

The last hidden state of the Encoder is the context summary C, representing a constant-length representation of the whole input sentence.

The Decoder is trained to generate (predict), step by step, an output sequence symbol $\hat{y}^{(t)}$. It is conditioned by its previous hidden state $h^{(t-1)}$, the context summary generated from the Encoder C and the previously generated output $\hat{y}^{(t-1)}$ (or target output $y^{(t-1)}$ at training time) [17]. Hence, the update function for the Decoder hidden state is:

$$h^{(t)} = f(h^{(t-1)}, y^{(t-1)}, C) \tag{2.14}$$

Figure 2.14 shows an example of how this type of RNN really works. On the horizontal axes there is the evolution of the *same* network, unfolded in time, highlighting four consecutive time steps. As the Encoder reads the inputs its hidden state evolves, being updated from h_0 to h_4 where the <EOS> symbol marks the end of the sentence. h_4 becomes the context vector which, is now used to initialize the Decoder, with the first last prediction $y^{(t-1)}$ set as NULL in the first iteration. The Decoder starts producing the output sequence, till another <EOS> is emitted signaling the end of the translated sentence. The *Output Sel.*, represented in dark

orange, is needed to selecting the best output word(s) for the current step, based on the likelihood produced by the Decoder (more details in subsection 2.4.9)

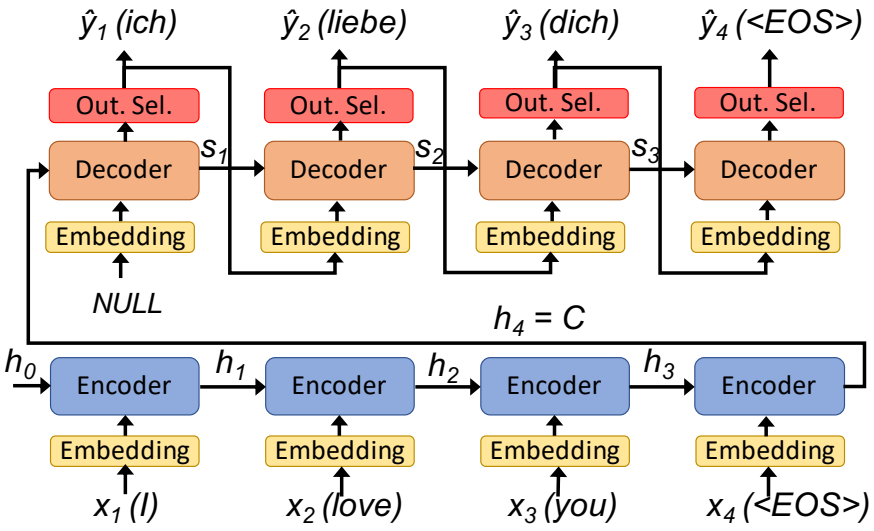


Figure 2.14: Encoder-Decoder RNN architecture

The two components of the RNN Encoder-Decoder architecture represent two different and independent language models, one for the input language and one for the output language. They are thus trained together in order to maximize a cost function, such the one presented in Equation 2.8 [17].
Once the network has been sufficiently trained, it can be used to generate an output sequence, given an new input sequence (never seen during training). At each iteration of the Decoder, it would output a vocabulary-sized vector containing the probability of each symbol to be selected as output prediction $\hat{y}^{(t)}$. The actual method used to sample the new symbol(s) out of this output distribution will be explained in more detail in section subsection 2.4.9

2.4.8 Attention Mechanism

In the context of Neural Machine Translation - NMT, i.e. the main target of this work, the Bidirection Encoder-Decoder LSTM-based RNN architecture is one of the most used model. A NMT system models the conditional probability $p(y|x)$ of translating an input sequence to a new target sequence, using two different language models [2]. One big limitation of the Encoder-Decoder architecture is when the input sequence is too long to be properly summarized by the fixed-length vector C, for which this context summary becomes the information bottleneck. A possible solution can be to use a "big enough" model, trained for "long enough" [12]. Even though that method has been proven to work, it will result is a needlessly large memory usage and long computation time when the network has to process short sentences. In addition, such a network will have many parameters making the training process even longer and less effective. A more efficient approach to deal with very long sequences is to use the **attention mechanism**.

The most popular attention models were proposed by Bahdanau et al., 2015 [18] and Luong et al., 2015 [2]. While they share the same common idea about attention, they differ in how the calculations are performed. In the following, the Luong's methodology is described. Details on Bahdanau method can be found in [18]
The concept of attention allows the models to be trained to learn the *alignment* between the two different language models held in the Encoder and Decoder [2]. The idea is to read the whole sentence, but instead of discarding all the hidden states of the encoder and preserving only the last one (context summary C), the network now creates a dynamic[5] memory of the source information, by saving the Encoder hidden states produced while scanning each word in the input sentence. Then at decoding time, instead of relying only on the hidden state of the decoder, the network produces one word at a time, each time "attending" to a different part of the input sentence. This allows to retrieve semantic information about the actual part of the context under translation [12] [2].
Further classification on attention models can be done based on the approach they use: *Global*, when all the source memory is attended, or *Local*, when only a subpart of it is considered at each time step.

[5] It is said to be dynamic because its size depends on the input sentence. A bigger memory will be created for long sentence with respect to short sentence. This allows to avoid problems of inefficient representation.

Global Attention

The idea in the global attention model is to consider all source memory at each time step [2].

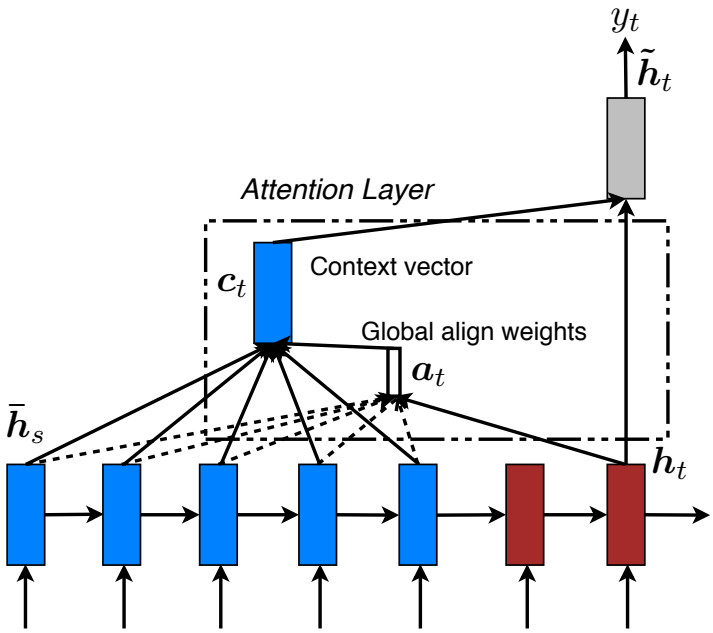


Figure 2.15: Global attention, picture taken from [2]

The attention computation happens for each Decoder step, performing the following sequence of operations:

1. The current target hidden state $h_{(t)}$ (of the Decoder) is compared to the source memory[6] (which comprises the list of the Encoder source hidden states $\overline{h}_{(s)}$) to derive the variable-length alignment vector a_t, or similarly called attention weights. Its size matches the source memory size. This alignment vector tells how much the network should focus on a particular source word for the current time step in order to predict the next word in the output sequence [15]

2. Given the attention weights, the network calculates a **context vector** c_t as a weighted sum over the whole source memory

3. The **attention vector** $\tilde{h}_t$ is generated by combining the context vector with the target hidden state

[6]For each word in the input sequence, a new Eecoder hidden state is calculated and then appended in the source memory

4. The output probability distribution is finally produced by feeding the attention vector through the softmax layer

Equation 2.15 shows how the attention weights vector is generated, as mentioned in Step 1. As this formulations is similar to the softmax, this vector is composed of K values (where K is the source memory size), where each entry is in range (0,1] and all entries sum up to 1

$$\begin{aligned} a_t(S) &= align(h_{(t)}, \overline{h}_{(s)}) \\ &= \frac{exp(score(h_{(t)}, \overline{h}_{(s)}))}{\sum_{s'} exp(score(h_{(t)}, \overline{h}_{(s)}))} \end{aligned} \tag{2.15}$$

Equation 2.16 shows the different scoring functions used to evaluate which part of the source memory the Decoder has to attended in order to generate the next output word. The last formulation was introduced by Bahdanau et al., 2015 and it was proven that even thought it is more computationally complex, it is slightly more accurate with respect to the multiplicative one proposed by Luong et al., 2015 [19].

$$score(h_{(t)}, \overline{h}_{(s)}) = \begin{cases} h_{(t)}^T \overline{h}_{(s)}, & dot \\ h_{(t)}^T W_a \overline{h}_{(s)}, & Luong's att(multiplicative) \\ v_a^T tanh(W_1 h_{(t)} + W_2 \overline{h}_{(s)}), & Bahdanau's att(additive) \end{cases} \tag{2.16}$$

Equation 2.17 shows the weighted sum used to create the context vector, as mentioned in Step 2. Here the previously generated alignment vector is used to scale the source memory.

$$c_t = \sum_s a_t(S) \overline{h}_{(s)} \tag{2.17}$$

Equation 2.18 shows how the current Decoder hidden state $h_{(t)}$ is fist concatenated with the context vector, then scaled by the weights matrix and finally squashed by the tanh function, in order to produce the attention vector. As detailed in Step 3, this is used to combine information coming from the Encoder source memory and the currently processed Decoder input.

$$\tilde{h}_t = tanh(W_c[c_t; h_t]) \tag{2.18}$$

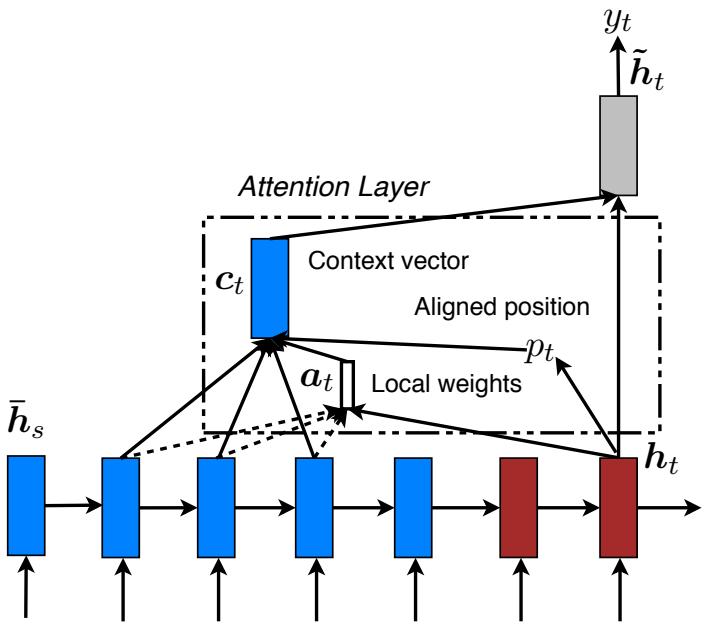


Figure 2.16: Local attention, picture taken from [2]

Equation 2.19 shows the predicted distribution computed by softmaxing the attention vector scaled by its weights matrix.

$$p(y_t|y_{<t}, x) = softmax(W_s\tilde{h}_t) \tag{2.19}$$

Local Attention

The deficiency of global approach is that the network has to pay attention to all input words, which is an expensive computation, and it can potentially decrease the strength of the attention when the input sequence is very long [2]. To address this flaw, the local attention mechanism proposes to only focus on a small window of the source memory at each time step.
The attention computation involves the following steps:

- For each target word, at each time step, the model generates an alignment position p_t
- The context vector c_t is now generate as a weighted sum over the selected window of the source memory [c_t - D, c_t + D], where D is now a parameter of the network and has to be selected

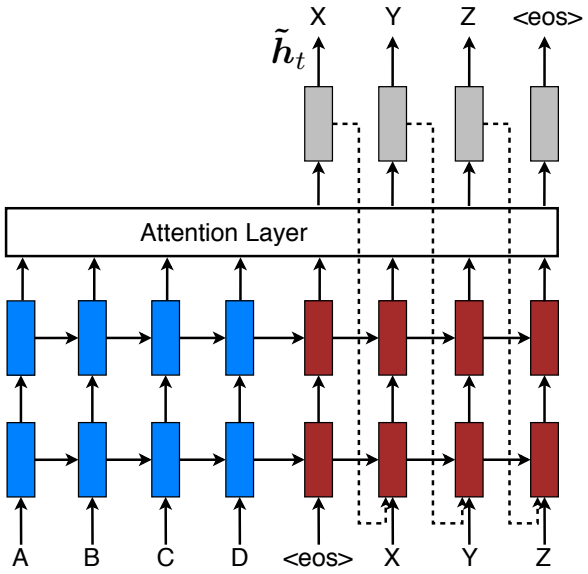


Figure 2.17: Input Feeding approach, picture taken from [2]

- Other steps remain unchanged with respect to the global attention

The alignment function can be expressed in different forms as:
Monotonic: the model sets $p_t = t$, assuming that the source and the target are monotonically aligned. This may not be true in many case, and depends on the language models used in the network [2].
Predictive: the model use an alignment function to predict the position p_t for the current time step. This function, and its parameters, have to be learned during training [2].

Input Feed

The attention models proposed till now take decision on the next word independently from the previously translated words and previous model alignments. This solution is suboptimal, since the model loses track of the evolution of the sentence [2]. To address this drawback, Luong et al., 2015 in their paper proposed the *input feeding* approach in which the attention vector $\tilde{h}_t$ of previous time step and the current input are concatenated and used to calculate the next alignment decision [2].

2.4.9 Word Sampling

In the previous sections, it has been explained how the model is able to generate an output probability distribution $p(y|x)$ of each world (over the output vocabulary) being the next word in the sequence, but it has not been elaborated on how it actually generates the final sentence. The decoding algorithm, and in particular the *Output Sel.* part in Figure 2.13, is in charge of *sampling* the probability distribution in order to generate the most likely sequence of words.
Finding the most likely translation out of all possible sentences is a NP-complete[7] problem, which involves searching through all possible words combinations to identify the sequence which maximizes its overall likelihood. In order to reduce the complexity, **heuristic** search algorithms are used. These methods do not return the best solution, but rather a "good enough" (approximated) sentence with a reasonable usage of resources and execution time.
The most used sampling algorithms are Greedy Search and Beam Search.

Greedy Search

It is the problem of generating only the 1-best result. At each time step the model, based on the output probability distribution, will pick in a "greedy way" the word with the highest probability and use it as next element of the sentence [15]. The appealing characteristic of this approach is that it is very fast and it doesn't introduce any overhead in the computation. The drawback, however, is that the produced sentence is suboptimal [15]. In fact, it can be proven mathematically that picking the most probable word at each time step does not result automatically in the sentence with the *overall* highest likelihood.

Beam Search

While greedy search will pick the most likely word and move on, beam search instead can consider multiple alternatives at the same time. This algorithm is characterized by a parameter called **Beam Width - BW**, which corresponds to the number of considered sentences during each decoding step.
The algorithm first starts by expanding from the root node, generally the start-of-sentence token (received after the encoder emitted the EOS). The model generates an output probability distribution for each word in the vocabulary and then it only keeps the best-BW, while pruning the other with lower probability. At the next it-

[7] The search problem is exponential to the length of the output sequence and so, it is impossible to explore the whole space

eration, the decoder will start expanding from each of the previously selected words (corresponding to the continuation of the first step hypotheses), temporary creating $BW \times vocab\_words$ probabilities for all possible partial sentences. Only the best-BW will be taken, while the others are pruned, such that the number of live hypotheses is kept constant throughout the whole translation. The process then repeats until all beams have reached the <EOS> symbol (or maximum allowed length), when the translation is stopped [15]. The final output sentence is selected as the one with the **overall** highest likelihood.

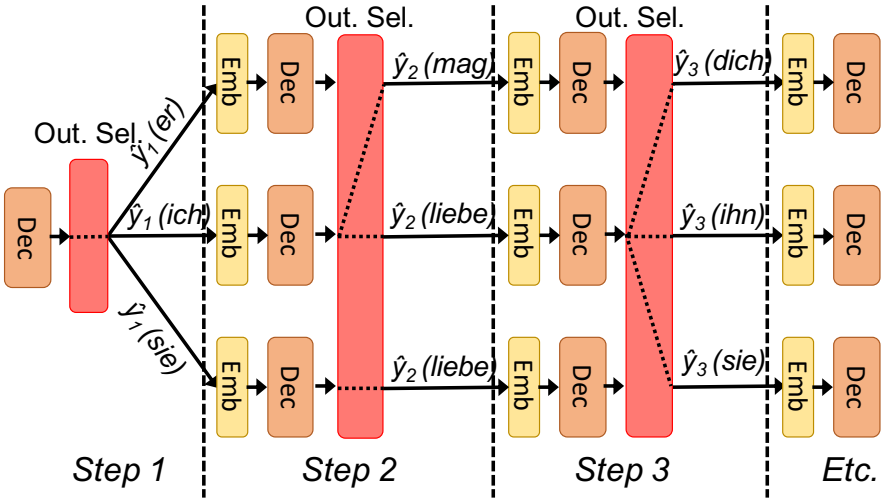


Figure 2.18: Beam Search wiyh a beam size of 5

Figure 2.18 above shows the execution of the beam search algorithm with a BW of 3. Each layer in the figure correspond to an iteration of the decoding, where only the 3 "partial sentences" with the highest probability are kept. In the first iteration, identified with Step 1, the Decoder (only conditioned with the context vector received from the Encoder) will generate the probability distribution for all the words to be the first in the sentence. In the example "ich" and "sie" were selected, and from those, the second expansion is performed. In Step 2, $3 \times vocab\_words$ alternative sentences are generated, but, only the 3 best partial sentences are kept alive. In the example "ich mag", "ich liebe" and "sie liebe" were the hypotheses with the greatest cumulative likelihood. The process then continues until all beams have reached the <EOS> character. It is worth to notice that the Decoder and Embeddings layers are the same already introduced in subsection 2.4.7, executed BW times per iteration.

It has been proven that a well tuned beam search results in a better quality translation with respect to greedy search. Furthermore, it is important to notice that when Beam Search is used, is necessary to keep in memory as many copies of the network as the Beam Width in order to evaluate different partial sentences concurrently, resulting in a computationally heavy task for the hardware hosting the network.

2.5 Metrics of evaluation

Evaluating the quality of the text generated by a RNN during the translation process, from one natural language to another, is a partially unsolved research problem. In the following, an explanation of the most commonly used metrics is provided.

2.5.1 BLEU

The **Bilingual Evaluation Understudy - BLEU** is a commonly used measure to evaluate machine-translated text. This algorithm offers an cheap and inexpensive metric to perform this task [20].
The principle behind this metric is to quantify the similarity between the output produced by the network and the golden reference generated by a professional human translator (it could be either a single sentence or a set of reference translations).
BLEU uses a modified version of the *precision*[8] measure. This method works by counting the number of translated words (unigrams) or aggregations of words (n-grams) from the network output which appear in the reference translation sentence. This comparison is performed regardless of the order in which words occur, and then it is normalized by the number of total words in the generated sentence [20]. Moreover, the precision counting methodology is modified to reward candidate translations which also match the right number of words occurrences in the reference text, penalizing sentences with an abundance of matching n-grams. Since precision can be highly influenced by too short translation candidates, the BLEU score includes a multiplicative *brevity penalty* factor B. For the B factor calculation, let c be the length of the candidate translation and r the length of the reference sentence:

$$B = \begin{cases} 1 & \text{if } c > r \\ e^{1-r/c} & \text{if } c \leq r \end{cases}$$

Then the BLEU can be computed as:

$$BLEU = B \cdot exp(\frac{1}{N} \sum_{n=1}^{N} \log p_n)$$

where p_n is the modified n-gram precision.
BLEU output is in the interval [0,1], 0 meaning perfect mismatch, while 1 refers

[8] In the context of pattern recognition and binary classification, the precision corresponds to the number of the true positive (or relevant) elements among all selected elements.

to the perfect match (unreachable). It is intended to score at sentence level, but a modified version can also be used to score blocks of multiple sentences.

2.5.2 PPT

The **Perplexity** can be considered as the measure of how many equally probable words can follow a given word, and so it can be thought a sort of "confusion" of the model while predicting the next word in the sentence. This estimation allows to measure how the language model, learned during the training phase, is capable to predict a given dataset and how the vocabulary has been compressed. Mathematically, for a training set $W = (w_1, w_2, ..., w_N)$ (where N is the vocabulary size), it can be expressed as:

$$PP(W) = P(w_1, w_2, ..., w_N)^{\frac{1}{N}} = 2^{\mathcal{L}}$$

where $\mathcal{L}$ is the negative log probability of the cross entropy error function (defined in section 2.2 and subsection 2.4.3 as the cost function). An inefficient language model would have a Perplexity equal to the vocabulary size N, meaning that each word in the vocabulary is equally probable to be the next word in the sentence. This is not the case in real language, where words have not the same probability to appear close to each other due to grammar rules and sentence meanings. So, a well trained language model should have a low Perplexity.
However, Perplexity is not a definite way to estimate the quality of a language model, since it closely depends on its training set, but it can be equally useful as a metric for comparing language models.

2.5.3 ROUGE

The **Recall-Oriented Understudy for Gisting Evaluation - ROUGE** is another metric for evaluating machine translated texts and automatic summarizations of texts. It works, similarly to the BLEU score, by comparing the machine-generated text (either translation or summary) and the human-generated golden reference. The most commonly used ROUGE evaluation metrics are:

- ROUGE-N: which measures the overlap of N-grams between the system and reference text
- ROUGE-L: which is based on the longest common suqsequences, it is used to evaluate the sentence similarity

- ROUGE-S: which measures the overlap of words pairs that can have a maximum number of "gaps" between them

Instead of simply evaluating "how much of the reference text has been captured during the process" (which can result in a poor and wrong estimation of the performances of the model), the ROUGE is able to evaluate "how much of the system text is relevant/needed with respect to the reference text". This is useful to cope with the characteristic of bad machine-generated texts to be very long, capturing useless words from the reference.

Chapter 3

Related Works

Deep Neural Networks are currently broadly used to support many Artificial Intelligence applications such as computer vision, speech recognition and robotics. While they are able to provide state-of-the-art accuracy in those applications, it comes at the cost of very high computational complexity and energy required by the system. This problem becomes even more present when these applications are executed to embedded hardware devices, i.e. edge nodes for IoT or mobile devices, where the hardware resources and the power budget are limited. As a natural consequence, the development of energy efficient DNN architectures able to guarantee state-of-the-art results without effecting the quality is critical to the future development of new DNN based applications [7].

One step toward energy efficient Neural Networks has been accomplished by the introduction of dedicated hardware accelerators [21] [22] [7] on both FPGA [23] and ASICS [5]. Different approaches have been proposed, targeting different aspects, both at circuital level and architectural level in order to optimize the most energy consuming operations. Although these designs are able to deliver very good performances in term of accuracy and energy efficiency, the vast majority of those focus on DNN or CNN architectures, while very few have been proposed to improve the efficiency of RNNs [9] [10].

At the algorithm level, previous works showed the successful application of *approximate computing* techniques to further reduce the complexity of the models, trading off lower quality of results for lower energy consumption [7]. Indeed, approximate computing could be seen as another source of computing efficiency playing alongside conventional techniques, on a different abstraction level. This paradigm can be effectively applied to all applications in which the "correctness" of the produced result does not mean providing a precise numerical answer, but rather presenting "good

enough" results for the purpose of the application itself. Examples of approaches leveraging approximate computing for neural network optimization are:

- reduce the precision of both operands and operations. Examples of this involve switching from floating-point operation to fixed-point operations and bit-width scaling
- reduce the number of total operations and the model size. Examples of this are pruning and model compression techniques

General purpose machines executing DNN models generally uses 32-bit floating point number representations, whereas embedded devices commonly do not have the hardware support to execute these operations. Fortunately, the high precision of floating point is not always necessary, since the higher energy spent while executing floating point operations does not result in more accurate results [6]. Thus, scaling and quantizing weights, biases and possibly inputs, could provide a good solution to both issues. The aspect which is now critical and has to be considered, is to decide how much the bit-width can be scaled down by switching from 32-bit floating point to 16-bit, 8-bit or even lower fixed point representation without affecting too much the classification accuracy. This process is usually performed having a knowledge of the inputs and weights dynamic ranges such that it is possible to define the minimum bit-with needed to encode all possible values. The granularity of the quantization can then be set to be uniform (single quantization setting for all the network) or per-layer (each layer has an ad-hoc quantization setting). Examples of previous works showing the effective execution of these techniques are the [6] and [24]. Moons at al. [6] have demonstrated how the intrinsic error resilience of machine learning algorithms could be exploited to decrease the bit-width during computation through quantization. Hubara at al. [24], pushed this idea further, showing how Binarized Neural Networks, i.e. models whose weights are constraint to take only values 0 or 1, and consequently can be stored with 1 bit of memory, could be effectively trained and how most of the arithmetic operations could be replaced with bit-wise operations, potentially leading to a substantial increase in power-efficiency. In addition to reducing the size of both operands and operations, many works have also studied how to reduce the number of total computations and model size. One solution proposed is to exploit the sparsity of ReLU output activations (many negative values are set to zero) for skipping the read operation of the weights and the following MAC for zero-valued activations [7]. Many works have also demonstrated how redundant weights in the network can be pruned in order to reduce the model size and also

speed up computations for both the forward and backward pass [25] [7].
Most of the studies presented above implement *static* energy-vs-quality tradeoff, in which the level of approximation is decided at design time and then kept constant throughout the entire execution. Only recently, *dynamic* approaches have been proposed, leveragin the observation that, for some applications, not all inputs are equally difficult to process. In such scenario, networks with a fixed level of approximation would either overapproximate where inputs are more difficult, producing poor results or underapproximate when inputs are easier, resulting in a waste of energy. Park et al. [11] in their work suggest a "Big/Little" implementation, where the network is duplicated, but with different size and complexity. When the network receives a new input, at first the "Little" (least complex) network is evaluated. According to the confidence of the classification produced, the result is committed as it is, or the "Big" network is triggered to provide a more accurate classification. The advantage in terms of reduced energy consumption comes when the Little network is executed most of the times. However, the major limitation of this approach is that the model embeds two complete networks, meaning that both size and training time are substantially increased. To cope with these problems, Tann et al. [5] propose a similar method in which there is only one Big network, which is fully or partially activated according to the different input constraints (i.e. delay and allowed power consumption). Finally, JahierPagliari at al. [8] in their work propose another methodology for adaptively reconfigure the network precision at runtime. In contrast to Park et al. they do not duplicate the network, but rather they use different quantization configurations for the same model, selecting the most appropriate one for the currently processed input, thus without introducing any hardware overhead. Works [11], [5] and [8] are based on the idea that inputs are not equally difficult to process, and so, based on the level of confidence during the classification and on the input constraints, the network could be reconfigured at runtime to match the input characteristics.

In literature there many studies exploring different architectures to improve the performances and accuracy of popular RNNs models. Many of those do not take into account the increment of complexity and computational power required to obtain such results. Only a small fraction focusses on energy efficient optimizations for RNNs architectures. Joachim Ott at al. [26] in their work propose to limit the numerical precision of the network weights and biases (another application of approximate computing introduced before) in order to reduce the computational

needs, achieving good results in specific dataset. Moreover, in [27] and [28] they both brought up the possibility to modify the beam search algorithm by making it dynamically adjustable in size.
Even though their approach is similar to the methodology proposed in this work, they differ in the discriminating criterion used during the adaptation of the beam width. Moreover, the presented methodology, as also the ones presented in [27] and [28], belongs to the *dynamic* approaches previously introduced, operating at the algorithmic level, such that no hardware overhead is introduced.
This thesis proposes a methodology which has been successfully tested on a single core CPU (to better evaluate the behavior of the modified architecture on embedded devices), but it can also be applied in the presence of dedicated hardware accelerators, as the ones presented in [9] [10]. In fact, the presented approach reduces the number of total Decoder calls, thus providing improvements either if the Decoder is implemented on CPU or on specific hardwares, e.g. FPGA.

Chapter 4

Dynamic Beam Search

4.1 Motivation

Many machine learning powered applications are supported by cloud based datacenters equipped with high-end clusters of GPUs. The application sends a query, servers elaborate it and send back the result, resulting in a very low effort for the host. However, this solution is inefficient, as a significant amount of energy and time are spent in the transmission of data to/from the end-node from/to the cloud. The general goal of this work is thus to propose a methodology for reducing the complexity of the inference process, such that RNNs could be implemented directly on low-power embedded devices, with limited hardware resources and energy budget.

At the beginning, a network characterization process was necessary. This operation was meant to identify the bottlenecks and the most demanding computations, such that it could be possible to focus on them directly.
The characterization phase was carried out using the profiling tools made available by the used frameworks (see subsection 5.1.2 for more details about frameworks), on pre-trained models (subsection 5.1.1 explain the choice to use pre-trained models instead of training new ones). In the list below are reported all models tested during this phase:

- Tensorflow - NMT Tutorial [34]
- Tensorflow - seq2seq [35]
- PyTorch - OpenNMT (DE-EN and DE-EN models) [36]

As explained in the guide [37], in order to profile Tensorflow based networks, the

timeline module can be used. By adding a specific run option, during the tf.Session(), it was possible to perform the profiling of the network full execution, and to save all information about operations and relative execution times in a *timeline.json* file. This file can then be opened with a Chrome browser, by going to the page "chrome://tracing" and loading it. On this webpage it is now possible navigate trough the execution history, analyzing the order in which different operations are performed, with relative execution times, and how information are passed between different modules.
In order to profile PyTorch based networks, the *profilehooks* package can be used. It provides a collection of decorators for profiling/timing/tracing individual functions. It is only needed to add @*profile* before the function of interest, and the results (name of function, number of calls and execution time) will be printed when the program exits.
Figure 4.1 and Figure 4.2 below show the results for the Tensorflow based models after the timeline files have been properly parsed, highlighting the most important parts in both the Encoder and Decoder process.

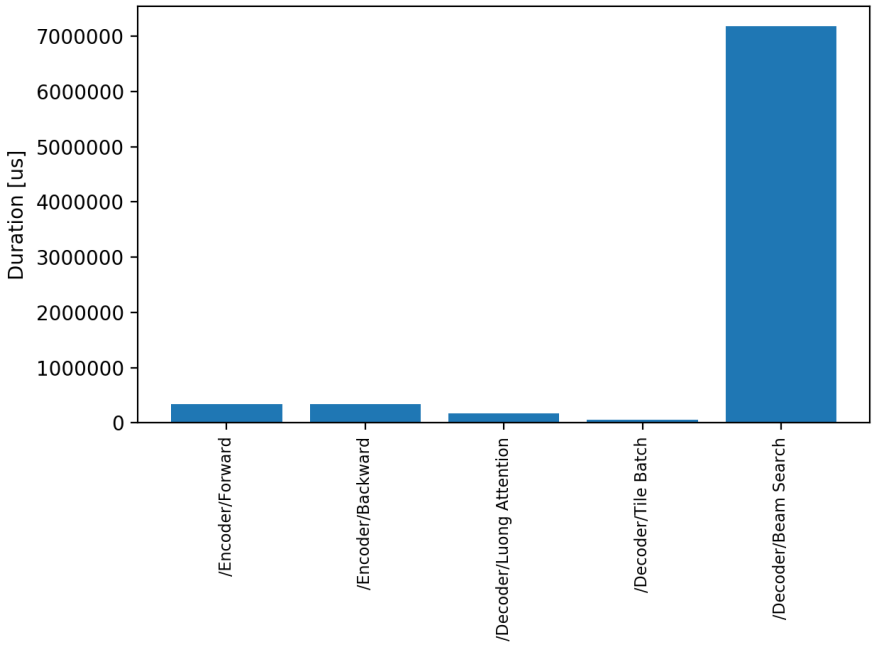


Figure 4.1: Nmt executions times

It is clear that the Encoding takes only a small fraction of the whole execution time, which is mostly due to the Decoding and in particular to the Beam Search.
To understand the growth trend of complexity of the beam search algorithm, many

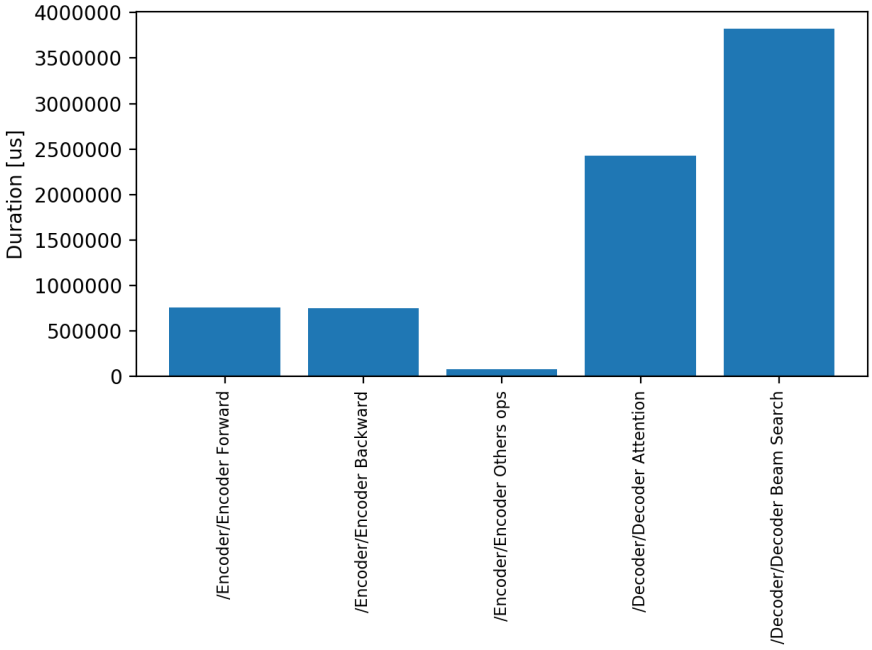


Figure 4.2: Seq2seq executions times

different tests on the models have been performed, each time modifying the beam width and evaluating both execution time and the gain or loss in the quality of results. Figure 4.3, Figure 4.4, Figure 4.5 and Figure 4.6 shows the collected results concerning the execution times.

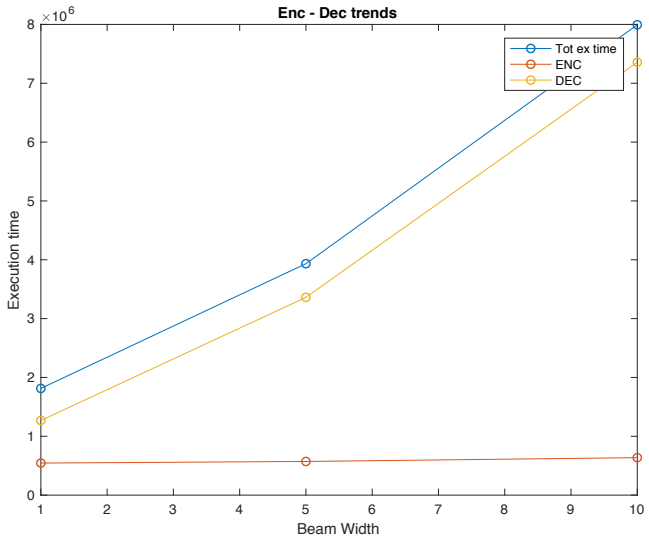


Figure 4.3: Nmt Beam Search complexity trend

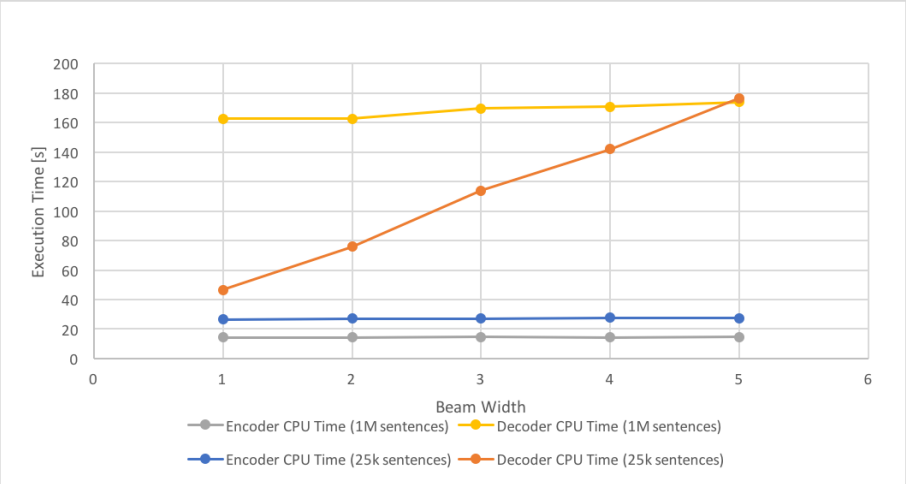


Figure 4.4: OpenNMT Beam Search complexity trend

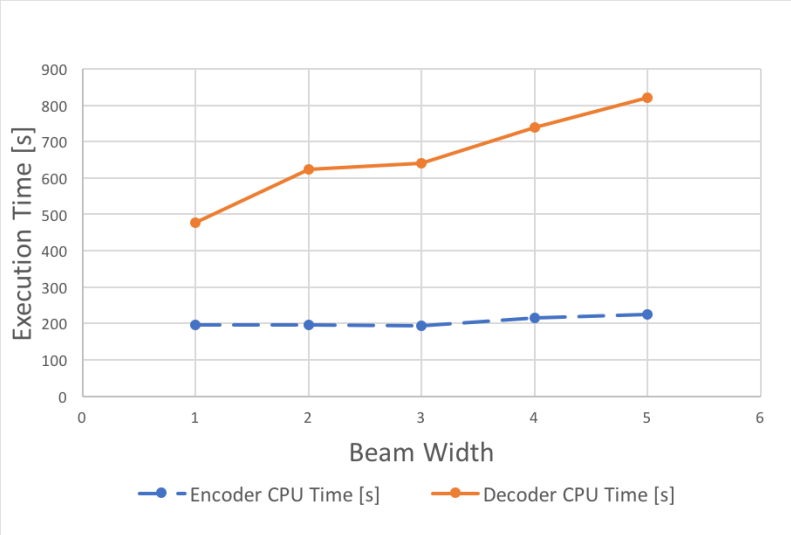


Figure 4.5: OpenNMT Beam Search complexity trend DE-EN model

All results proposed refer to the execution of the models on single-thread CPU. This aims to emulate how a low-power embedded hardware, generally equipped with single-thread CPU, would handle the computations and how complexity scales changing different network parameters, Beam Width in particular. It is clear that, for all tested beam widths, the Encoding process takes only a small fraction of the whole execution time, which is mostly due to the Decoding. Furthermore, increasing the beam width has a dramatic impact on the total number of operations, resulting in higher execution time and energy consumption. As it could be expected, the Encoder execution time is independent from the beam width, remaining almost

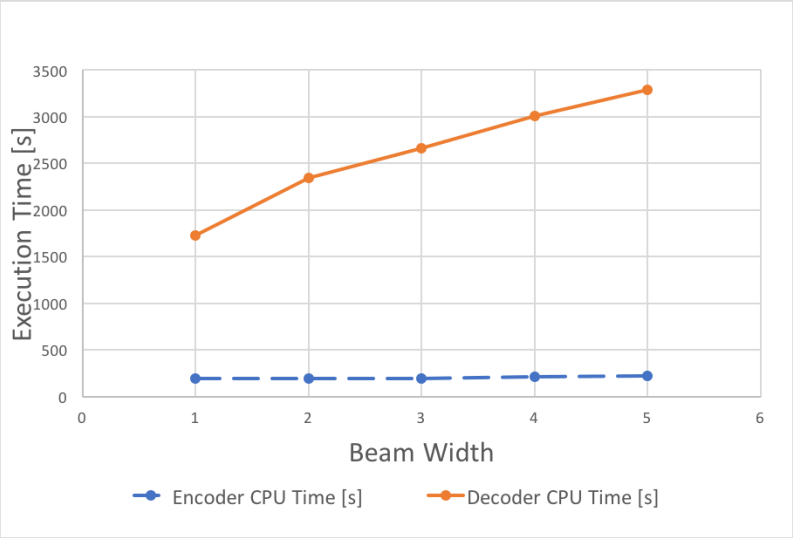


Figure 4.6: OpenNMT Beam Search complexity trend EN-DE model

constant during all tests, while the Decoder execution time grows approximately linearly with the beam width. In Figure 4.3, Figure 4.4 Figure 4.5 and Figure 4.5 this trend is evident, and in Figure 4.4 it can also be observed the impact of different corpus lengths. In fact, passing from 25K lines to 1M lines, the Enc phase increases slightly, while the Dec phase varies considerably for the BW = 1 and also its sensibility to the BW is reduced.

Since each step of the network execution involves multiple calls of the same operations, the CPU power consumption can be expected to remain almost constant during the translation process. Therefore, one way to reduce the energy consumption required by the model is to reduce its *execution time*. In light of proposed results, it has been chosen to focus on the beam search process, since the BW parameter has a strong impact on the decoding phase and on the overall execution time. In fact, having a BW of x is equivalent of performing the decoding process x times. While in a GPU based application this process can be carried out concurrently, on a single thread CPU it has to be performed sequentially, so increasing the execution time. Typical values of BW are in range 3 to > 5, depending on the level of accuracy required and the processing power available.
Exploring the whole searching space is a NP-complete problem. Exact algorithm, such as the Depth-First Search and Breadth-First Search are extremely inefficient since they traverse the entire search tree in order to find the best solution. A considerable improvements for these algorithm is to insert some heuristic in the traversing

process [27]. The Beam Search algorithm is an optimization of the Breadth-First Search leveraging on this concept.
As anticipated before in subsection 2.4.9, the Beam Search algorithm is able to explore the search space by considering multiple alternatives at the same time, finally selecting the sentence with overall highest likelihood. The Beam holds BW hypothesis (where BW is the beam width) while pruning the others with lower probability. In standard implementations of the Enc-Dec RNNs, the Beam Width parameter is chosen statically, before executing the network, in order to guarantee a good level of translation even for the most complex sentences. This imposes at prior the complexity of the model without taking in account the characteristics of the inputs. Hence, there can be cases where sentences with poor likelihood are not pruned simply because there are other sentences with even lower scores. Or viceversa, where sentences with high likelihood are discarded because competing with other sentences with relatively higher scores [27].
For this reason, this works proposes novel dynamic Beam Search algorithm, where the beam width BW is not kept constant throughout the search process, but rather varied accordingly to the evolution of the translation. This allows the network, at each step, to self tune the effort required to produce a good translation. The *confidence* of the network will be assessed after each iteration of the Decoder and then used as an indicator to adjust the width.

4.2 Objective

All policies proposed leverage on the idea of "translation confidence". This work tries to estimate the confidence of the network, after each Decoder step, by analyzing the scores it outputs. As anticipated before in subsection 2.4.7, the decoder outputs represent the probability of partial sentences, and thus their *distribution* can provide good indications about the difficulty of a translation step. This method is based on the intuitive concept that not all inputs, sentences or part of them, are equally difficult to translate. For easier inputs, a small BW could be sufficient, while for more difficult ones a larger BW is needed to obtain a good result. The network is said to be confident whenever scores are "enough spread", and, conversely, it is said to be non-confident whenever scores are "close" to each others. Scores really close indicates similar words (e.g. verbs with different tense, or synonyms), among which the network strive to pick the best candidates. The estimation of the amount of dispersion, and so the confidence of the network, is a metric that this work aims to

evaluate.

The final objective is to propose a discriminating policy able to take produced scores as input and identify the best beam width to continue the beam search process, without sacrificing the translation performances. This aims to reduce the amount of computations required by the Decoder while producing better or comparable results with respect to a standard Decoder implementing beam search with a fixed width. Figure 4.7 below shows an example of the proposed Dynamic Beam Search algorithm, in which the beam width is modified dynamically. In this example in particular, the second iteration of the Decoder may have produced an output with much higher score than the others, so the BW is reduced to 1 (high confidence). Conversely, in the next iteration, the produced scores are more similar and close to each others, so the BW is increased to 3 in order to cope with a more difficult translation (low confidence).

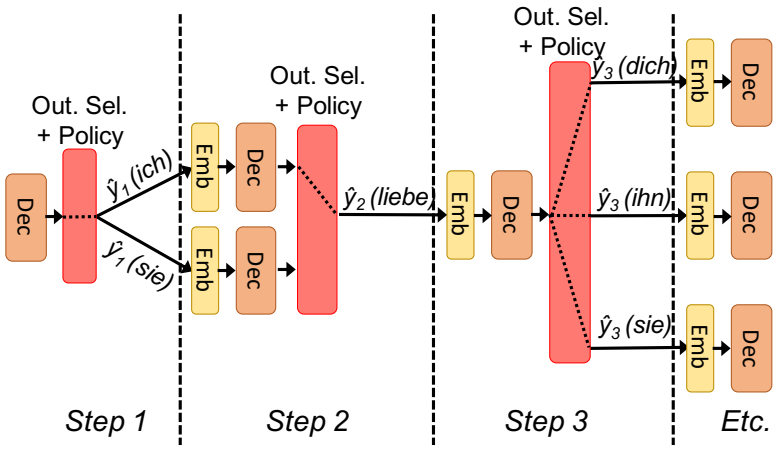


Figure 4.7: Dynamic Beam Search example

4.3 Inference In The Adopted Framework

Before presenting the actual introduced variations of the original architecture, it is worth summarizing how inference is implemented in the selected neural network software framework, i.e. the OpenNMT project in its PyTorch implementation (section 5.2 for details).

Inference is the process to produce a prediction based on pre-learned parameters, given a certain input. It is composed of a single forward pass of the network, without any backpropagation or updating of the weights matrices. The network receives an

input, elaborates it and outputs a prediction (e.g. a translation). This process is composed of many steps, and they change according to the network implementation and task. In the following, the inference process for an Encoder-Decoder RNN will be summarized, with respect to the target network implementation explained in section 5.2. Moreover, the explanation will only consider one sentence at a time, but the network is capable of analyzing batch_size sentences concurrently, executing the same steps in parallel for all the inputs. This because, in single-thread CPU based embedded devices, performing this process with multiple sentences concurrently ($batch\_size > 1$) is not convenient, since it requires a parallelism which is not provided by the CPU (while GPU does) and it also requires much more memory. At the beginning, after a initialization phase for the internal variables, the Encoder is executed. It will read step-by-step the input sentence, till the <EOS> token is reached. As the Encoder progresses forward reading the sequence, the hidden state will be updated and then used to create the memory of the source information. Its last hidden state is the context summary C of the whole input sequence and it will be used as initialization vector for the Decoder. The Decoder of the architecture under test, implements global attention, input feeding and fixed width beam search. Upon receiving the <EOS> token from the Encoder, it starts emitting one target word at a time. For each iteration, at first, the previously predicted output $\hat{y}^{(t-1)}$ and the current input are concatenated to make the model aware of the previous alignment (input feeding). Then, the new attention vector is calculated (as weighted sum over the whole source memory, as explained in global attention subsection 2.4.8) and used to produce the output probability distribution (whose size matches the output vocabulary size) used to select the next words. The beams are then advanced and the BW most probable alternatives are kept for the next iteration. The Dec process continues till all the BW beams has reached either the <EOS> token or the maximum sentence length. At this point, the sentence with the overall highest likelihood will be committed as output.

The Encoder process is straightforward and it is defined in the "run ENC" box. When it has completed, it produces as output its final hidden state, named here as enc_state and the memory of all Enc states, named here memory_bank. The enc_state is then used to initialize the Dec. Now, the decoding process starts. It is implement as a loop, checking that neither the maximum sentence length is reached, nor the beam has finished, before continuing in the computation. Thus, the Dec is advanced by one step (defined in the "run one DEC step" box), generating as output

its hidden state, the output distribution after the attention layer and the attention weights (named here dec_states, dec_out and beam_att respectively). Scores for each word in the vocabulary are now available and are reshaped to be used in the following steps. Finally the beam search algorithm is executed. Since the network is virtually able to work with batch_size sentences at a time, each one having its own beam, each beam is then treated separately in a sequential manner. The beam process is composed of two phases, the *advance* and the *update*. In the advance phase, the best-BW scores are selected and the partial sentences with the highest likelihood are kept for the next iteration, while the others are pruned. Instead of saving the actual words, the network identifies in the vocabulary the indexes corresponding to the previously selected words and append them to a list identifying the sentence. The beam width BW is a inference parameter, set when the network is run and kept constant throughout the whole process (e.g. BW = 5 means that each sentence in the batch would have a beam advancing 5 alternative sentences at a time). Then, the update phase starts and the Dec hidden states related to the new BW hypothesis which have been selected are copied into the old dec_states. This allows the network to start the next iteration from a valid state, corresponding to the sentence the beam has chosen to carry on.
Once the Decoder process has terminated, the alternative with highest score will be selected and the beam would be unrolled backward to retrieve the actual sentence, reconstructing it from the word indexes saved at each iteration.

The implementation proposed by this work follows closely the flow mentioned before, enhancing it with the introduction of the discriminating policy and the dynamic adjustment of the beam width.

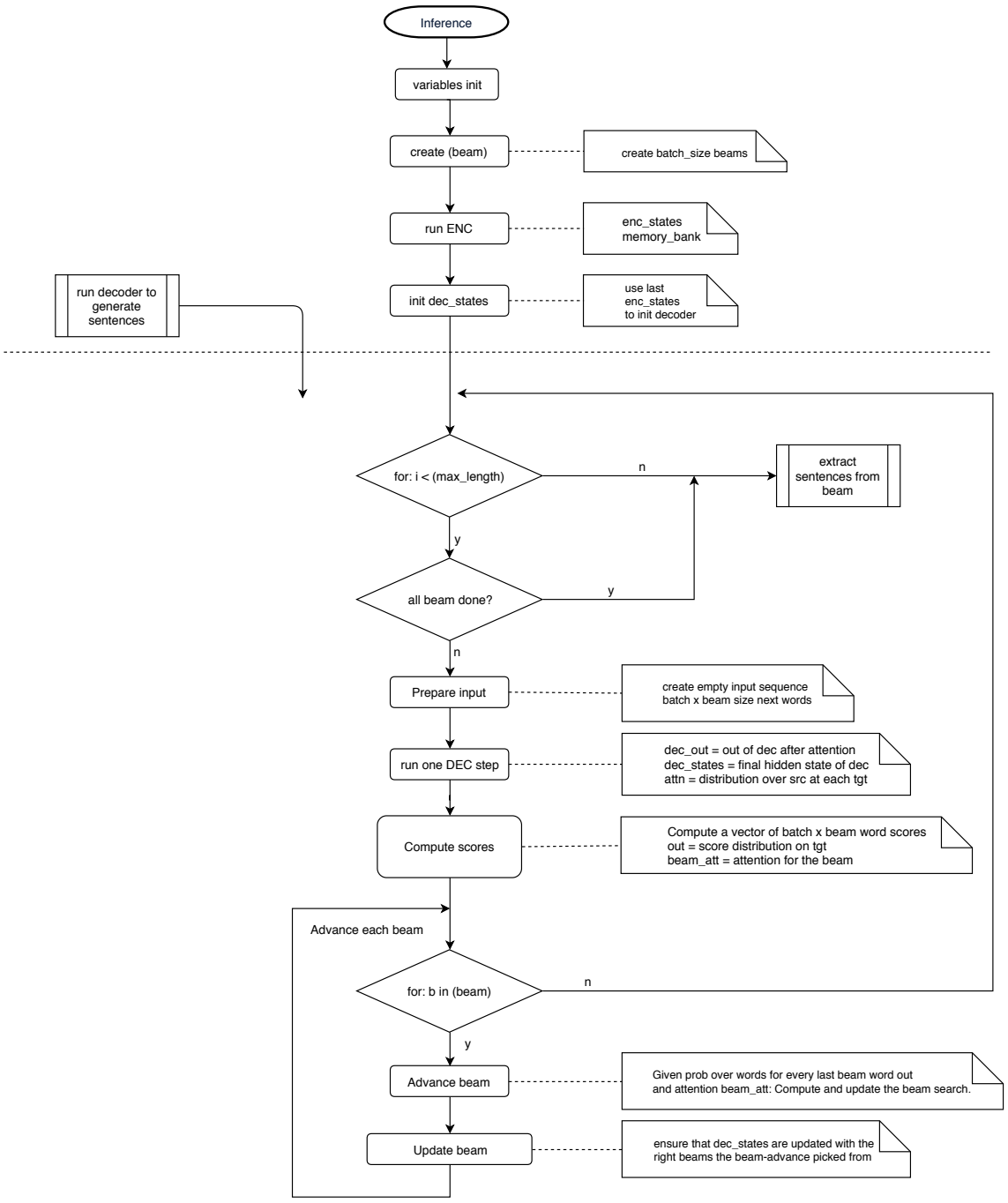


Figure 4.8: Flow chart of Inference phase

4.4 Proposed Policies

In order to explore the optimization space, many different policies have been tested in this work. It is important to notice that the policy is called at each iteration of the Decoder, **before** the execution of the beam search algorithm. Also, a batch size of one is considered, meaning that the beam width is changed dynamically considering only one sentence at a time. With a bigger batch size, the network will have different beams, each one related to a different sentence in the batch, whose beam width will be changed independently, resulting in a very complex task to be handled. Moreover, as explained, a batch_size of 1 is the most common scenario in an embedded device.
In the following, a brief explanation of the idea behind each policy will be provided, as well as its effectiveness compared with a baseline result (see section 5.3 for results).

Generally speaking, all hard-coded parameters (e.g. thresholds) in the proposed policies can now be treated as hyperparameters of the network. Proposed hyperparameters were chosen after experimentation on the validation set of the used dataset. If these policies will be applied in different context and on different networks, it could be necessary to tune these parameters to achieve similar results.

4.4.1 Random

This policy uses a completely random approach. This does not guaranteed to be optimal or logical, but rather easy to implement. Most importantly, this policy can be used as a baseline for comparison for smarter approaches, in order to prove that their rationale is indeed providing better results than a trivial random approach. This policy continuously switch the next beam width between 1, greedy decoding, and 3, point at which BLEU becomes less sensitive with the beam width (as it can be seen in te baseline results in section 5.3).

4.4.2 Standard Deviation

This policy tries to adapt the beam width according to the statistics of the top-k scores produced in the current iteration.
It has been chosen to take into account only the best-5 scores (out of all scores, as many as the vocabulary size) since network instructions suggest 5 as best beam width, and also it has been proven that BLEU does not increase substantially with

higher beam width (see section 5.3). Moreover, statistics computed upon the full scores will be less indicative with respect to statistics computed only on the best next word candidates. After the best-5 scores have been selected, the policy calculates the standard deviation and the mean for the current iteration and then it changes the beam width accordingly. Different implementations of this policy have been proposed, also considering the distribution either as Normal or Uniform[1]. It has been chosen to use the standard deviation since is a statistical measure to evaluate the dispersion of a dataset. In the following, the variants are reported:

- **Threshold-based**: At each iteration, the beam width is modified starting from the previous one. If the std. deviation is greater of a certain threshold, the beam width is reduced by one, otherwise it is increased by one. The std. deviation is calculated as for a Normal distribution, and the threshold is decided empirically. To identify good threshold candidates, a histogram showing possible std. dev. values with relative occurrence rates has been plotted. A script then has been used to identify the highest std. dev. value to cover a certain amount of occurrences (e.g. for the iwslt14 dataset, with a std of 1.23 the 40% of values are covered).
- **Mean ± Std. Dev.**: At each iteration the beam width is set to be 5 (minimum confidence and higher effort for the next iteration). Then, it will be decreased by one for each score which lays out of the boundaries defined by $mean \pm std.dev.$, and increased by one for each score laying inside these boundaries. This policy tries to evaluate how scores are spread around the expected value (the scores mean). The boundaries have been changed by using different fractions of the std. dev., i.e. 1/2, 1/3. The std. dev. is calculated as for a Normal distribution and Uniform.

4.4.3 Mutual Distance

This policy modifies the beam width based on the mutual distances between best-5 scores. For each iteration, the distances between consecutive scores are calculated as well as the mean distance. Starting from an initial beam width of 5, it will be decreased by one for each length greater then a threshold. It has been chosen to use the distance as a measure of the dispersion of the best-5 scores because the more the points are distant to each other, the more the dataset is spread out, meaning that the network is confident. Different thresholds have been tested:

[1]Normal and Uniform distributions differ in the computation of the standard deviation.

- **Mean**: The threshold is the mean of the distances (also fractions of the mean has been tested). The downside of this comparison is that it does not take into account the actual value of the distances, but rather the relation between the mean and the subject distance, resulting in a "mask" of the actual magnitude. Thus, set of very close scores, similarly separated, could result in the same behavior of a set having scores very spread out, but similarly separated too.
- **Real Number**: The threshold is a real number describing "far enough" scores. To identify good threshold candidates, a histogram showing possible distance values with relative occurrence rates has been plotted. A script then has been used to identify the highest length to cover a certain amount of occurrences (e.g. for the iwslt14 dataset, with a length of 0.127 the 40% of length values are covered).
- **Distances Distribution**: This policy is a variant of the Mean ± Std. Dev., introduced in section before. Instead of comparing the scores with their std. deviation, here the policy uses mutual distances and their relative std. dev.

4.4.4 Score Margin Like

This policy is inspired by the Score Margin based approach used in the article [8] and [5]. They assess the classification confidence of a CNNs by using the so called Score Margin[2] - SM, and reconfigure the hardware accordingly. Here, instead, the score margin is computed as the absolute difference between the best-BW scores. At the beginning of each iteration, a minimum beam width is set and the scores of the best-5 are selected. The policy (labeled as SM V3) is implemented as a while loop which will increase the minimum beam width whenever the current beam width is lower than five (upper limit in the worst case scenario) and the SM between the scores under inspection is lower then a threshold. As the while continue executing, it increases by one the beam width and compute again the SM with the next couple of scores (such that the next iteration will compare the second ond third scores, ecc), otherwise the current beam width is returned. This approach starts with a minimum tolerable beam width, under which the network would perform poorly, and increases it only when it is necessary, meaning that the network is not confident and needs a bigger beam width to continue the translation process.
Other variants of this policy have been tested, modifying the comparison window (e.g. comparing the first highest score with lower scores each time the minimum

[2]Defined as the absolute difference between the the first and second highest output values.

beam width is increased, labeled as SM V2), or introducing and hard threshold in the first comparison to identify when the network is very confident about the first two scores (labeled as SM V1).

4.4.5 Std Mapping Function

This policy is a variant of the policy proposed in subsection 4.4.2 because it also uses the standard deviation as indication of the network confidence for the currently processed input. This policy has proven to be the most effective, compared to the previously proposed policies, so it has been decided to explore it in more details.
A line equation is used to formalize the relationship between the std. deviation and the future beam width. The std. deviation is used as the independent variable for this line equation, which identifies min and max beam width and all integer points in the middle. The equation will return a point corresponding to the next beam width (rounded at closest integer) according to the current std. deviation, calculated for the best-5 scores. Large values of std. dev. mean that the scores are very spread out, so the network is considered confident and it will return a small value for next beam width. Conversely, small values of std. dev. mean the network is non-confident, so it will return an higher value for the next beam width. The std. dev. is calculated as for a Normal distribution, and the line equation is defined also empirically. Moreover, as anticipated in subsection 4.4.2, during the inference process of the original network with fixed BW, the values of the std. dev. as well as their relative occurrences have been collected and analyzed since they provide a good starting point for exploring the parameters space.

The tunable parameters of this policy are :

- The two points that define the mapping function: each of these points corresponds to a pair of values in the std. deviation-BW plane. The first point (0,maxBW) gives the worst case beam width when the network is extremely non-confident with the produced scores. Zero or near-zero std. deviation value means that the scores are excessively close, requiring high value for the BW in order to produce good results. The second point (maxSTD, minBW) gives the minimum beam width when the network is really confident. The maxSTD value defines the threshold for which the network is allowed to use the minimum beam width, while still performing properly. Trying different configurations of these points allowed us to experiment different behaviors of the policy, resulting in different degrees of approximation and BLEU - Avg_beam_width results.

Reasonable values of the BW could be 1-2 for the minBW and 2-3-4 for the maxBW according to the quality of results needed and the expected model complexity.

- Rounding technique: the beam width returned by the mapping function is generally a float value, but the network requires an integer value as next beam width. Both rounding and truncation have been tested as rounding techniques, resulting in less aggressive and more aggressive (respectively) policy.

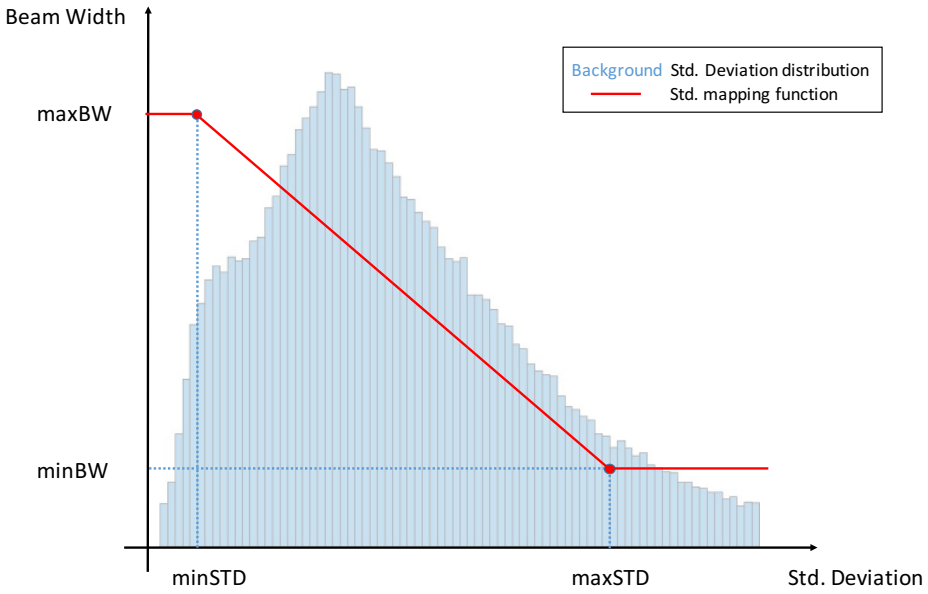


Figure 4.9: Beam Width selection policy

The Figure 4.9 above shows the working principle of this policy. The vertical axes of the graph reports the Beam Width (which may vary between maxBW and 0), while the horizontal axes reports the standard deviation values and, on the background, it is presented the rates of occurrences of the std. dev. values. In the minSTD and maxSTD interval, the BW is reduced linearly as the std. dev. increases, while it is saturated for values lower than minSTD and greater than maxSTD.

4.4.6 Practical modifications to the adopted software framework

Among all features and files offered by the OpenNMT toolkit (see section 5.2), the contribution of this work focuses on a subset of those. In the following, a brief explanation of the most important files, with their functionalities will be provided,

as well as the changes introduced.

translator.py: It is the core of the translation process. According to the input commands provided while executing the network, it loads the desired model parameters, build the selected network, prepare input data and finally instantiate one object of the Translator class. This object will then handle all the elements involved and carry out the translation.
The contributions on this part are:

- The memory of the Enc states has been made dynamic in size, matching the beam width currently used in the iteration.
- Implementation of the dynamic beam width selection policy before the execution of the beam search algorithm. According to the produced scores, it defines the best new beam width to be used in the coming beam search iteration.
- The *beam.advance* function has been modified to accept as input the new beam width and to return a tuple containing the newly generated decoder hidden states.
- The Dec has been made adaptive to the new beam width. This allows to advance only the number of beams decided by the policy. Also, the Dec hidden states, scores and backpointers will now adjust their size accordingly.
- For each beam advanced, the new Dec hidden states produced (related to the beams that have been selected to be carried on) are saved in the *new_batch_state* variable. When all beams have been advanced successfully, their states are updated by the function *dec_states.from_list*. The old dec_states and Dec hidden states are replaced by the previously saved states, such that the network is able to start the next iteration from a valid state.

```
1 # (c) Advance each beam.
2 new_batch_state = []
3 for j, b in enumerate(beam):
4     # advance each Beam for each sentence in the input_batch
5     b.advance(out[:, j],
6                 beam_attn.data[:, j, :memory_lengths[j]], ...
      new_beam_size)
7     # save the new_decoder_states computed with the new beam size
8     new_batch_state.append(dec_states.beam_update(j, ...
      b.get_current_origin(), beam_size))
```

```
# update the decoder_states
dec_states.from_list(new_batch_state, beam_size, new_beam_size)

# assign new beam size after all beams in batch have been advanced
beam_size = new_beam_size
# save the currently used beam_size for statistics
beam_size_counter += beam_size
```

Listing 4.1: Beam Advance part - in translator.py

- The network will output the *mean beam width* used during the translation of each sentence. This will be used to estimate the mean beam size used during the whole process.
- The network is now able to receive as input parameter the policy to be tested, such that the experimentation process is easier and faster.

decoder.py: It defines the class and functions related to the attention-based decoder. This is the decoder used in the model to perform the DE-EN translation. The contributions on this part are:

- Modified the *beam_update* function. Now it returns a tuple containing the newly generated Dec hidden states. These states will be used in translator.py, as mentioned above.
- Implementation of *from_list* function, specific for RNNDecoderState Dec states.

```
# internal dec_states are: self.hidden[0], self.hidden[1] and ...
    self.input_feed
def from_list(self, new_batch_state, beam_size, new_beam_size):

    size =self.hidden[0].size()
    batch_size = size[1] // beam_size
    # create a the new dec_state, just a matrix filled with zeros ...
    and already with right dimensions
    self.hidden = (torch.zeros(size[0], batch_size * new_beam_size ...
    , size[2], dtype=self.hidden[0].dtype, ...
    device=self.hidden[0].device),
        torch.zeros(size[0], batch_size * new_beam_size , size[2], ...
    dtype=self.hidden[0].dtype, device=self.hidden[0].device))
    # create a state view as done in the function beam_update, ...
    this can be seen as a "pointer" to the  new dec_state ...
    ("aliasing" ion python)
```

```
10      h0view = self.hidden[0].view(size[0], new_beam_size, ...
        batch_size, size[2])
11      h1view = self.hidden[1].view(size[0], new_beam_size, ...
        batch_size, size[2])
12
13      size = self.input_feed.size()
14      batch_size = size[1] // beam_size
15      self.input_feed = torch.zeros(size[0], batch_size * ...
        new_beam_size , size[2], dtype=self.input_feed.dtype, ...
        device=self.input_feed.device)
16      infview = self.input_feed.view(size[0], new_beam_size, ...
        batch_size, size[2])
17
18
19      for i, new_state in enumerate(new_batch_state):
20          h0, h1, inf = new_state
21          # copy data related for the sentence we are currently ...
        analyzing [:,:,i]
22          # doing the copy_ on the state view will also update the ...
        pointed dec_state
23          h0view[:,:,i].data.copy_(h0)
24          h1view[:,:,i].data.copy_(h1)
25          infview[:,:,i].data.copy_(inf)
```

Listing 4.2: from_list function impmementation - in decoder.py

transformer.py: It defines the class and functions related to the implementation of the "Attention is all you need" decoder. This is the decoder used in the model to perform the EN-DE translation. The contribution on this part is the implementation of *from_list* function, specific for TransformerDecoderState Dec states.

```
1  # internal dec_states are: self.previous_input, ...
       self.previous_layer_inputs and self.src
2  def from_list(self, new_batch_state, beam_size, new_beam_size):
3
4      size = self.previous_input.size()
5      batch_size = size[1] // beam_size
6      # create a the new dec_state, just a matrix filled with zeros and ...
       already with right dimensions
7      self.previous_input = torch.zeros(size[0], batch_size * ...
       new_beam_size , size[2], dtype=self.previous_input.dtype, ...
       device=self.previous_input.device)
8      # create a state view as done in the function beam_update, this can ...
       be seen as a "pointer" to the  new dec_state ("aliasing" ion python)
```

```
9       piview = self.previous_input.view(size[0], new_beam_size, ...
        batch_size, size[2])
10
11      size = self.previous_layer_inputs.size()
12      batch_size = size[1] // beam_size
13      self.previous_layer_inputs = torch.zeros(size[0], batch_size * ...
        new_beam_size , size[2], size[3], ...
        dtype=self.previous_layer_inputs.dtype, ...
        device=self.previous_layer_inputs.device)
14      plview = self.previous_layer_inputs.view(size[0], new_beam_size, ...
        batch_size, size[2], size[3])
15
16      size = self.src.size()
17      batch_size = size[1] // beam_size
18      self.src = torch.zeros(size[0], batch_size * new_beam_size , ...
        size[2], dtype=self.src.dtype, device=self.src.device)
19      srview = self.src.view(size[0], new_beam_size, batch_size, size[2])
20
21
22      for i, new_state in enumerate(new_batch_state):
23          prev_in, prev_lay_in, src = new_state
24          # copy data related for the sentence we are currently analyzing ...
        [:,:,i]
25          # doing the copy_ on the state view will also update the ...
        pointed dec_state
26          piview[:,:,i].data.copy_(prev_in)
27          plview[:,:,i].data.copy_(prev_lay_in)
```

Listing 4.3: from_list function impmementation - in transformer.py

beam.py: It defines the beam object and all functions related to the beam search process. It handles all beams related to each beam object, back pointers and scores. It also defines the scoring system used to re-rank all scores after each iteration using a penalty function. It has been modified to be able to dynamically change the beam width according to the network needs.

One of the greatest challenge found during this phase was to understand how to make Encoder and Decoder internal states adaptive in size with the new beam width (defined at each iteration) and understand how to properly carry on correct and updated dec_states during the beam advance process.
The parts of the network that have been modified and adapted were selected through a reverse engineering process, since the network lacks of both documentations and comments in the code. For the second problem instead, once understood the method

behind the *dec_states.beam_update* function (used to update the beam after the advance phase), the same methodology has been applied to *dec_states.from_list* proposed function. It exploits Python Aliasing addressing mode to access and update the dec_states in a more convenient way.

Chapter 5

Experimental Results

5.1 Framework Selection

The Machine Learning paradigm, with everything it involves, is evolving rapidly. This means that architectures, models, learning algorithms, ecc, are changing constantly based on new trends in the scientific community. Given that, building and successfully training these networks is becoming an increasingly difficult task.
Nowadays, ML Frameworks have become the standard way to develop ML architectures, rather than writing the models from scratch. These frameworks provide a higher abstraction layer, with libraries and tools designed to allow the programmer to build his network faster, easier and without worrying about the underling algorithms. The key features provided are:

- Easy way of building complex computational graph[1]
- Automatic way to compute gradient on computational graph
- Efficient run on both CPUs and GPUs

In the following sections it will be presented what drove the framework selection (subsection 5.1.1) and a side-by-side comparison of pre-selected frameworks which led to the final choice subsection 5.1.2.

5.1.1 Objectives

The main objectives that drove the frameworks selections were:

[1] As already explained in section 2.4.2, each network can be represented by a computational graph

- Easy availability of material as: official documentation, tutorials, active supporting community
- Availability of pre-trained models. That because it would be desirable to prove that the proposed methodology could be applied to ready-to-use models without the need of retraining them. This is a key characteristic because it allows to tune the effort required by the network according to the energy budget available (e.g. local vs cloud or current battery level) without each time retraining the network. Moreover, training a RNNs models would have required weeks or even a month, so it has been decided to focus on optimizing already trained models, rather than creating a new one.
- Possibility to profile the network. That because in a context of optimization, it is important to understand which are the most computationally demanding parts and how the introduced modifications impact on performances
- Possibility to easily modify the network

5.1.2 Framework Comparison

For the sake of this work, two frameworks have been evaluated: TensorFlow and PyTorch. This is because they are the most popular, widely used, well supported and documented currently available. In the following, their main differences are listed and compared.

TensorFlow is based on Theano and has been developed by Google, whereas PyTorch is based on Torch and has been developed by Facebook. Even though they both use Python, their internal cores are based on C++ and CUDA (for GPU execution). Both are designed and optimized to execute efficiently on CPUs and GPUs.

Both frameworks use a computational graph to define the network, but they differ in the way they build it. TensorFlow uses *static graph*, while PyTorch uses *dynamic graph*. In TensorFlow the computation is usually divided in two stages. At first the computational graph is *statically* defined, then computations are ran as many time as it is needed by repeatedly feeding data into the graph. The graph is made accessible by *tf.Placeholders*, which provide input or output nodes where data can be either fed-in or read-out at runtime. The computational graph is then defined by specifying the sequence of operations to be performed on placeholders, which are

now treated as symbolic variables. Still no computation is performed. Finally, once the graph is ready, a *tf.Session* is opened, data are fed-in as well as the outputs of interest. TensorFlow now autonomously performs only the needed computations to produce the output it has been asked for.
In PyTorch instead, the graph can be defined and then modified at runtime, new nodes can be added and executed when needed, without the demand of placeholders or a Session. This is particularly helpful while using variable length inputs, or dynamic network architectures that can benefit from this dynamic approach (e.g. in RNNs).

Since PyTorch is deeply integrated with Python, all Python debuggers as pdb, ipdb can be used to inspect the execution of the network step by step. This is not the case for TensorFlow, for which only tfdbg is available. It is important to notice that debugging in TensorFlow is a difficult operation due to the separation of graph definition and graph execution, as already explained above.
When it comes to model and data visualization, TensorFlow has a native built-in tool called TensorBoard, while PyTorch has nothing comparable. TensorBoard allows to plot the model graph (node's computation, their connections and information flowing through the network), and to visualize variables with relative distributions and histograms. This is very useful to better understand what changes in the network after a modification of some hyperparameters.

TensorFlow is more mature for production models and scalability[2]. It was built to be production ready, whereas, PyTorch is easier to learn and lighter to work with, and hence, it is more indicated for research and building rapid prototypes.

Differences	**TensorFlow**	**PyTorch**
Programming Language	Python/C++	Python/C++
Optimized for	GPU/CPU	GPU/CPU
Graph Definition	Static	Dynamic
Data Visualization	TensorBoard	—
Documentation & Community	Very Broad	Growing
Intention	Production	Research

Table 5.1: Framework comparison

[2]Built-in instruction to automatically parallelize a model on a distributed hardware.

At the beginning, it has been decided to start working with TensorFlow due to the abundance of pre-trained models, better tutorial/documentation and a more active community. But, as the work of customizing the network was proceeding, it has been noticed that TensorFlow was not the right choice. This is due to its way to statically define the computational graph, which did not allow the runtime customizations of the model that are necessary in order to modify the width of the beam at runtime. Therefore, it has been finally decided to implement the proposed strategy in PyTorch.

5.2 Experimental Setup

5.2.1 Models

OpenNMT [29] is an open source project (MIT) for Neural Machine Translation and Sequence Modelling launched in December 2016. It currently has three main implementations in the most popular frameworks: Lua, PyTorch and TensorFlow. Currently all are maintained and in active development. They implements features of recent research topics, offered in a compact toolkit with a simple general purpose interface. The OpenNMT project aims to offer complete library for training and deploying neural machine translation models with the goal of supporting NMT research, prioritizing efficiency and modularity [29].
Among the pre-trained models made available by OpenNMT, the two considered for this work are: English-to-German, and the German-to-English, hereafter referred as EN-DE and DE-EN respectively.
The English-to-German model was trained using the WMT15 dataset, and it is composed of 6 layes of 512 LSTM each, implementing the Transformer[3] as basic units and using BPE[4]. The German-to-English model, instead, was trained using the IWSLT14 dataset, and it implements 2 layes of 500 LSTM each, word embedding of size 500, bidirectional encoder, attention and input-feed decoder. Both model use Beam Search in the last stage to sample words.

5.2.2 Computing Platform

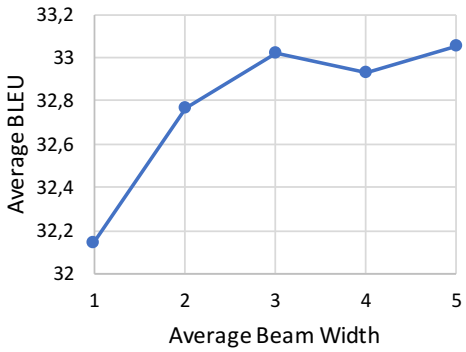
In order to quickly evaluate the effectiveness of the policies, all experiments needed to identify the best sets of parameters have been performed on a desktop

[3] Architecture proposed in "Attention is all you Need", by Google Brain
[4] Byte Pair Encoding, a simple data compression algorithm in which common pair of consecutive bytes are replaced with a new byte which does not occur in the data.

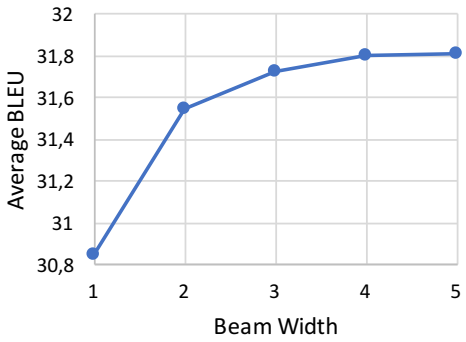
workstation equipped with a NVIDIA Titan XP GPU , whereas the measurements of execution time have been performed on a laptop equipped with an Intel Core i7 CPU and 32GB of RAM. Even though the reported execution times would not be representative of an actual implementation on a real embedded device, the presented trend could be a valuable approximation. This because, as mentioned before, the network has been forced to work with only one sentence at a time ($batch\_size = 1$) and it has been executed on a single-thread CPU, which is the common scenario for an embedded device.
All tests have been performed on the validation set of the dataset used during the training phase and the BLEU, PPL and ROUGE scores have been used as the metrics to evaluate the time (or complexity) versus quality tradeoff. This is because the policy can now be considered as an additional network hyperparameter.

5.3 Results

Before assessing the effectiveness of the proposed policies, it is necessary to identify a baseline measure of the performances of the OpenNMT network without any modifications. The methodology applied consists of running many translations of the same dataset with different beam widths and register each time the BLEU, PPT and ROUGE scores obtained. It has been decided to test beam width form 1 to 5. Figure 5.1, Figure 5.2 and Figure 5.3 below show the baseline results for the scores considered. For the first pair of graphs, on the horizontal axes the Average Beam Width is presented. Since the model is executed with fixed BW, it corresponds to the actual selected BW for the execution. The other two pairs, instead, present on the horizontal axes the Normalized execution time, i.e. the execution time of the network divided by the one of the original network with BW = 1.
As expected, as the beam width is increased, the BLEU score also increases (larger is better), the PPL decreases (smaller is better) and the ROUGE increases as well (larger is better). This shows how the beam search can effectively improve the translation quality with respect to using only greedy search (BW = 1). Being able to evaluate concurrently different candidate sentences allows to correct inaccuracies committed during the process, resulting in a better output. It is important to notice that, even if the minimum and maximum values (going from BW = 1 to BW = 5) related to all scores change of a small amount, this correspond to sizable differences in the translation quality. Furthermore, to an increase of the beam width, corresponds an increase of the execution time required to perform the computations. In

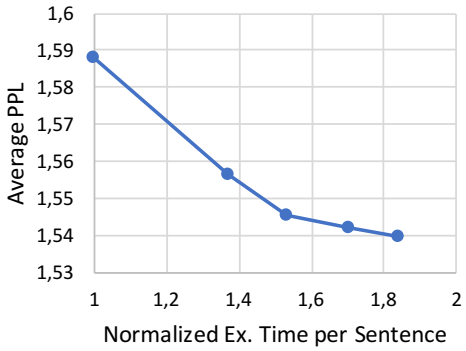


(a) EN-DE

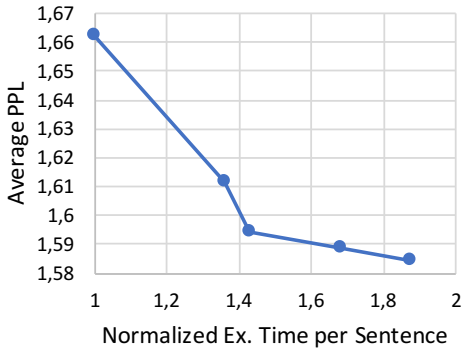


(b) DE-EN

Figure 5.1: Baseline BLEU

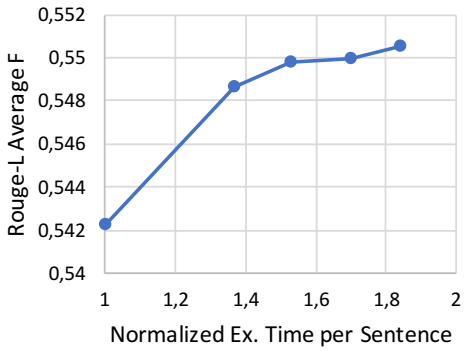


(a) EN-DE

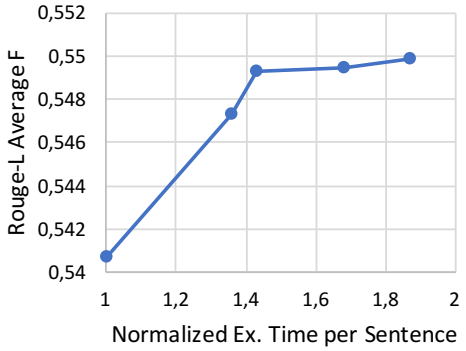


(b) DE-EN

Figure 5.2: Baseline PPL



(a) EN-DE



(b) DE-EN

Figure 5.3: Baseline ROUGE

both models, the greatest improvement of BLEU is between beam width 1 and 2, which causes an increment of 36% and 37% of execution time respectively. After this initial boost, the BLEU becomes less sensitive to the beam width, increasing with a slower trend and eventually flattening. This means that to gain marginal improvements on the quality of the translation, a great effort is required to the network which is reflected in higher complexity of the model.

Figure 5.4 below shows the different policies applied to DE-EN model with the IWSLT14 dataset, in which each point in the graph represents the best parameters setting for the tested policies. For the BLEU score, the target is to produce results that come closer to the top left part of the graph, area in which for a mid value of the BW, the BLEU score is very high. Conversely, for the PPL score, the target is to produce results that come closer to the bottom left part of the graph. It can be seen that the random approach proposed in subsection 4.4.1 does not work at all, resulting in the network performing very poorly. This means that to reach better results, at least comparable with the baseline, more sophisticated policies are required. The most promising results are obtained with the Score Margin V3 (blue circle), Mutual distance based (green circle), Normal $Mean \pm Std$ (diamond black) and finally the Std. Mapping Function (black circle).

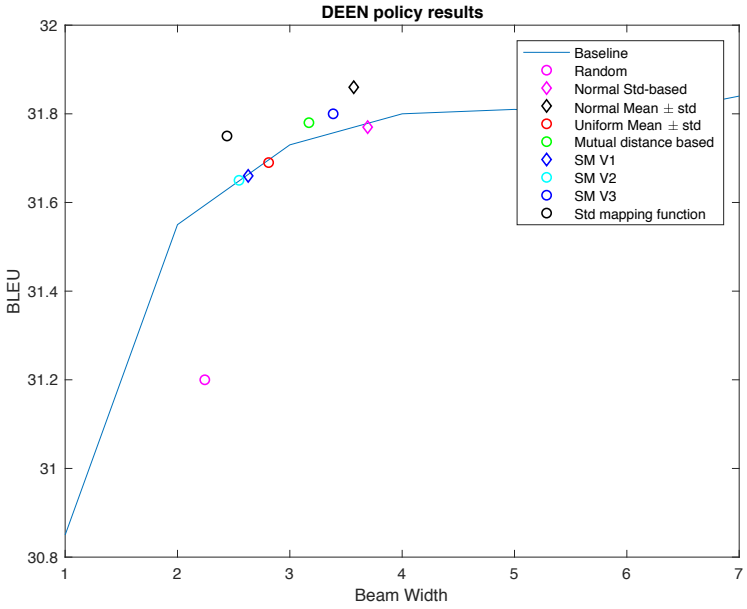


Figure 5.4: DE-EN policy comparison

Figure 5.5 below shows the different policies applied to the EN-DE model with the WMT15 dataset. It can be seen that, also here, random approach does not work at all. The most promising results are obtained with the Score Margin V2 (cyan circle), Mutual distance based (green circle), and the Std. Mapping Function (in black circle).

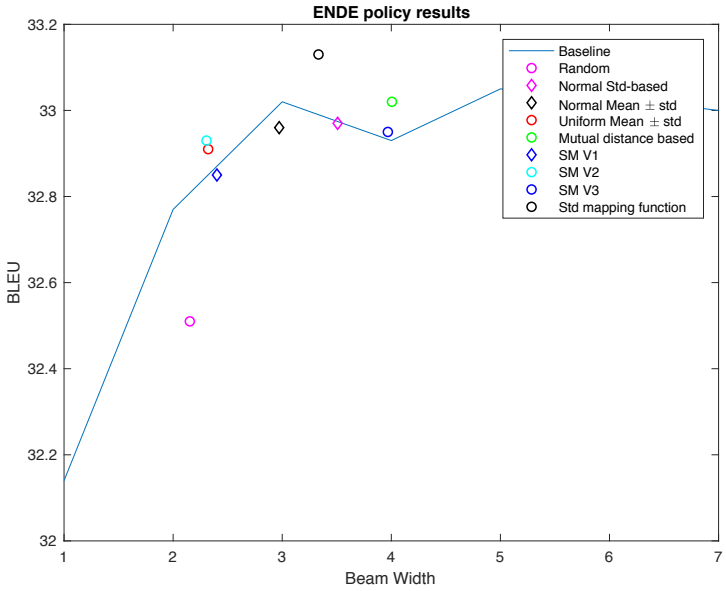
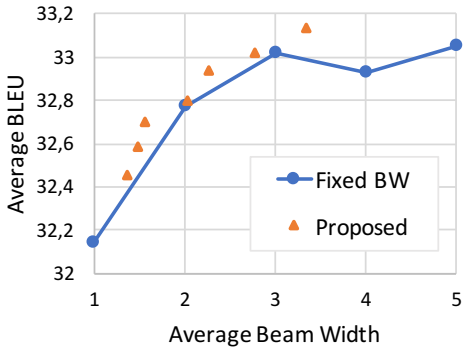
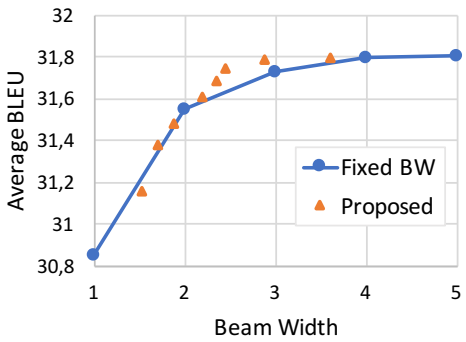


Figure 5.5: DE-EN policy comparison

By varying the parameters of the different policies, it has been discovered that the Std. Mapping Function was the one that provided positive results (i.e. better than the baseline) more consistently. This is shown in Figure 5.6, Figure 5.7 and Figure 5.8. The orange triangles represent some of the Pareto points obtained during the experiments. This time, the Average Beam Width on the horizontal axes refers to the result of the application of the mapping policy.
The policy has allowed to reach comparable or better accuracy with, on average, smaller beam width. In the DE-EN model, with BW of 2.347 and 2.869, it has been possible to reach comparable results with the execution having a fixed BW of 4 and 5. In the EN-DE model, with BW of 3.33, it allowed to reach the highest BLEU score ever recorded during the tests, while still performing consistently better with respect to the model executing with fixed beam width. The differences in the results of the two models is due to the fact that the EN-DE model implements a different architecture (Transformer) with respect the DE-EN model, which clearly benefits more of this policy.

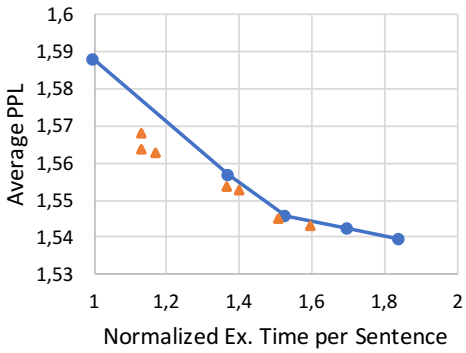


(a) EN-DE

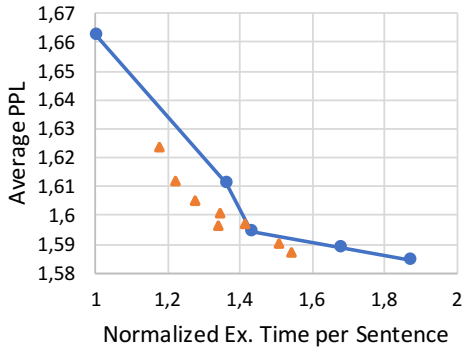


(b) DE-EN

Figure 5.6: BLEU with proposed policy

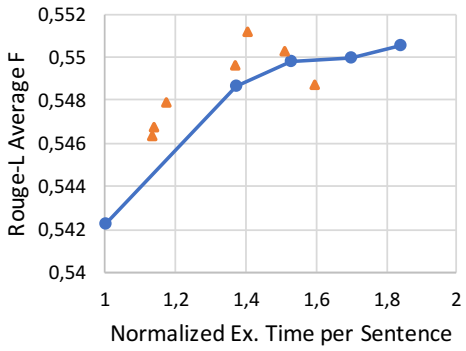


(a) EN-DE

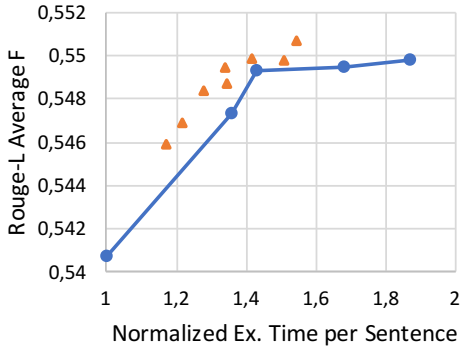


(b) DE-EN

Figure 5.7: PPL with proposed policy



(a) EN-DE



(b) DE-EN

Figure 5.8: ROUGE with proposed policy

Comparable results are also obtained for PPL and ROUGE metrics, were the proposed Std mapping policy is able to consistently outperform the models with fixed beam width.

In Table 5.3 and Table 5.2 the considered metrics vs execution time results are reported, formalizing the conclusions already discussed with graphs in sections above. In particular, Table 5.3, explicits the parameters used while testing the Std mapping function, with relative points presented in the graph above (orange triangles)

BS Fixed	**EN-DE BLEU**	**Enc Exec Time [Avg]**	**Dec Exec Time [Avg]**	**DE-EN Fixed**	**Enc Exec Time [Avg]**	**Dec Exec Time [Avg]**
1	32.14	1	1	30.85	1	1
2	32.77	1	1.35	31.55	1	1.31
3	33.02	1	1.54	31.73	1	1.34
4	32.93	1.1	1.75	31.80	1.09	1.55
5	33.05	1.14	1.91	31.81	1.11	1.72

Table 5.2: Normalize BLEU and execution time results with **fixed** BS

RNN	**minBW/ maxBW**	**minSTD/ maxSTD**	**Avg. BW**	**Ex. Time**	**BLEU**	**PPL**	**ROUGE**
EN-DE	1/2	0.1/2.2	1.36	1.13	32.45	1.568	0.5463
	1/2	0.1/3.1	1.55	1.17	32.70	1.563	0.5479
	1/3	0.1/1.7	1.48	1.13	32.58	1.564	0.5468
	1/3	0.1/3.1	2.02	1.37	32.80	1.554	0.5496
	2/4	0.1/1.7	2.27	1.40	32.94	1.552	0.5511
	2/5	0.1/1.3	2.77	1.51	33.02	1.545	0.5503
	2/5	0.1/1.7	3.33	1.59	33.13	1.543	0.8457
DE-EN	1/2	0.1/2.2	1.51	1.17	31.16	1.623	0.5459
	1/2	0.1/3.1	1.69	1.22	31.38	1.612	0.5469
	1/3	0.1/2.2	1.88	1.27	31.48	1.605	0.5483
	1/3	0.1/3.1	2.19	1.34	31.61	1.596	0.5494
	2/4	0.1/0.6	2.35	1.34	31.69	1.600	0.5487
	2/4	0.1/1.7	2.86	1.51	31.79	1.590	0.5498
	2/5	0.1/1.7	3.59	1.54	31.80	1.586	0.7707

Table 5.3: Normalize numerical results

Thanks to the introduced method for evaluating the translation confidence and how the scores produced after each iteration are spread, the input-dependent tuning approach described before can be effectively used to dynamically change the beam width according to the state of the translation process. This allows to reach comparable or even better results with respect to an execution of the model with fixed beam width, but with a lower beam width on average. Considering a single-threaded software implementation of the considered RNNs, which is the common scenario for an embedded device, obtained results translate into a 25% reduction of the total

execution time required for inference, while maintaining translation quality. This translates into significant energy savings.
Moreover, beside the execution points defined with fixed beam widths, this approach provides more power-performance working points in which the network can be executed. These points can be varied at runtime by acting on the policy parameters, in response to external conditions (e.g. battery status in a mobile system).

Chapter 6

Conclusions and Future Works

The increase of computing power required by DNNs, and especially RNNs, makes these models almost impossible to be executed on embedded devices, where energy budget and hardware resources are limited. In this work it has been proposed a methodology for reducing the complexity of the inference process by dynamically tuning the beam width depending on the currently processed input. It has been shown that applying the proposed input-dependent dynamic beam width tuning technique it was possible to speedup the translation process achieving comparable or even better accuracy, while reducing the average beam width and model complexity. This work describes different policies, then focuses on the most promising one, which leaves the space to future works for exploring other variants and optimizations. More experimentation should also be performed trying to integrate this approach to approximate computing techniques (e.g. quantization) to further reduce the complexity of RNNs. In addition, the proposed policies could be extended also on GPU based application, allowing a $batch\_size > 1$. This would allow to process many sentences at the same time (taking full advantage of the parallelism offered by the GPUs), while modifying the BW for each one of them, achieving same benefits explained before. Finally, the implementation of the proposed technique on an actual embedded device is also part of the envisioned future work.

Bibliography

[1] M. Milad, M. Rohit, and S. Richard, "Cs 224d: Deep learning for nlp," Spring 2015.

[2] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *arXiv preprint arXiv:1508.04025*, 2015.

[3] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, 2017, pp. 5998–6008.

[5] H. Tann, S. Hashemi, R. Bahar, and S. Reda, "Runtime configurable deep neural networks for energy-accuracy trade-off," in *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 2016, p. 34.

[6] B. Moons, B. De Brabandere, L. Van Gool, and M. Verhelst, "Energy-efficient convnets through approximate computing," in *Applications of Computer Vision (WACV), 2016 IEEE Winter Conference on*. IEEE, 2016, pp. 1–8.

[7] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[8] D. J. Pagliari, E. Macii, and M. Poncino, "Dynamic bit-width reconfiguration for energy-efficient deep learning hardware," in *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM, 2018, p. 47.

[9] F. Silfa, G. Dot, J.-M. Arnau, and A. Gonzalez, "E-pur: An energy-efficient processing unit for recurrent neural networks," *arXiv preprint arXiv:1711.07480*, 2017.

[10] Y. Hao and S. Quigley, "The implementation of a deep recurrent neural network language model on a xilinx fpga," *arXiv preprint arXiv:1710.10296*, 2017.

[11] E. Park, D. Kim, S. Kim, Y.-D. Kim, G. Kim, S. Yoon, and S. Yoo, "Big/little deep neural network for ultra low power inference," in *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*. IEEE Press, 2015, pp. 124–132.

[12] G. Ian, B. Yoshua, and C. Aaron, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[13] M. Milad, M. Rohit, and S. Richard, "Cs 224d: Deep learning for nlp," Spring 2015.

[14] Z. C. Lipton, J. Berkowitz, and C. Elkan, "A critical review of recurrent neural networks for sequence learning," *arXiv preprint arXiv:1506.00019*, 2015.

[15] G. Neubig, "Neural machine translation and sequence-to-sequence models: A tutorial," *arXiv preprint arXiv:1703.01619*, 2017.

[16] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[17] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[18] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[19] D. Britz, A. Goldie, M.-T. Luong, and Q. Le, "Massive exploration of neural machine translation architectures," *arXiv preprint arXiv:1703.03906*, 2017.

[20] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual*

meeting on association for computational linguistics. Association for Computational Linguistics, 2002, pp. 311–318.

[21] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An architecture for ultralow power binary-weight cnn acceleration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 48–60, 2018.

[22] J. Zhu, J. Jiang, X. Chen, and C.-Y. Tsui, "Sparsenn: An energy-efficient neural network accelerator exploiting input and output sparsity," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*. IEEE, 2018, pp. 241–244.

[23] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.

[24] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Advances in neural information processing systems*, 2016, pp. 4107–4115.

[25] E. Watanabe and H. Shimizu, "Algorithm for pruning hidden units in multilayered neural network for binary pattern classification problem," in *Neural Networks, 1993. IJCNN'93-Nagoya. Proceedings of 1993 International Joint Conference on*, vol. 1. IEEE, 1993, pp. 327–330.

[26] J. Ott, Z. Lin, Y. Zhang, S.-C. Liu, and Y. Bengio, "Recurrent neural networks with limited numerical precision," *arXiv preprint arXiv:1608.06902*, 2016.

[27] M. Mejia-Lavalle and C. G. P. Ramos, "Beam search with dynamic pruning for artificial intelligence hard problems," in *2013 International Conference on Mechatronics, Electronics and Automotive Engineering*. IEEE, 2013, pp. 59–64.

[28] M. Freitag and Y. Al-Onaizan, "Beam search strategies for neural machine translation," *arXiv preprint arXiv:1702.01806*, 2017.

[29] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush, "Opennmt: Open-source toolkit for neural machine translation," *arXiv preprint arXiv:1701.02810*, 2017.

[30] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, p. 533, 1986.

[31] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Approximate computing and the quest for computing efficiency," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 120.

[32] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," *arXiv preprint arXiv:1604.03168*, 2016.

[33] C. Olah, "Understanding lstm networks," https://colah.github.io/posts/2015-08-Understanding-LSTMs/, Aug 2017.

[34] "Tensorflow - nmt tutorial," https://github.com/tensorflow/nmt#hands-on--building-an-attention-based-nmt-model.

[35] "Tensorflow - seq2seq," https://github.com/eske/seq2seq/tree/master.

[36] "Pytorch - opennmt," https://github.com/OpenNMT/OpenNMT-py.

[37] "Tensorflow profiling guide," https://towardsdatascience.com/howto-profile-tensorflow-1a49fb18073d.

[38] M. Verhelst and B. Moons, "Embedded deep neural network processing: Algorithmic and processor techniques bring deep learning to iot and edge devices," *IEEE Solid-State Circuits Mag.*, vol. 9, no. 4, pp. 55–65, Fall 2017.

[39] P. Adams, "The title of the work," *The name of the journal*, vol. 4, no. 2, pp. 201–213, 7 1993, an optional note.

[40] P. Babington, *The title of the work*, 3rd ed., ser. 10. The address: The name of the publisher, 7 1993, vol. 4, an optional note.

[41] P. Caxton, "The title of the work," How it was published, The address of the publisher, 7 1993, an optional note.