POLITECNICO DI TORINO

Master degree course in Ingegneria Informatica
(Computer Engineering)

Master Degree Thesis

Energy-Efficient Quality Adaptation for Recurrent Neural Networks

Supervisors
prof. Massimo Poncino
doc. Daniele Jahier Pagliari

Candidate
Francesco Daghero
matricola: 251161

Academic Year 2018-2019

Contents

List of Tables

List of Figures

Abstract

Recurrent Neural Networks (RNNs) are the deep learning models which are considered the state-of-the-art for sequence modeling and machine translation tasks. One of the most used RNN architectures is the Encoder-Decoder network, which consists of two networks, one encoding the input sequence to a fixed length representation and a second one which then decodes it to produce a new sequence. These models, which lead to high accuracy results and are able to handle sequences of different length, come with increased computational complexity and memory requirements compared to other machine learning models. This complexity, in turn, translates into high energy consumption. This additional overhead makes the execution of such models impossible on embedded devices or edge nodes, which are not equipped with hardware powerful enough to sustain the heavy computations required. However, researchers have shown that executing deep learning inference on such devices, if made possible, would provide several benefits in terms of responsiveness, total energy consumption and security.
In order to make the inference more energy efficient on edge nodes, this work proposes an algorithm that improves the energy efficiency of the Encoder-Decoder RNNs and proves its effectiveness by measuring the real energy saving on an embedded device. In particular, this work implements a dynamic Beam Search algorithm, which varies the Beam Width (BW) according to the input processed and the translation complexity. The value for this parameter, which is directly tied to the complexity of the network, will be determined by using a new discrimination criterium which yields on the two datasets tested a reduction of the execution time above 20%. This time reduction also affects the energy consumption, reducing it significantly while keeping the same translation quality.

Chapter 1

Introduction

Machine Learning has become increasingly present in every aspect of society during the last years, performing tasks such as object recognition, image recognition and speech recognition with accuracy close or superior to that achieved by humans. Deep Learning, a subset of Machine Learning, uses a general-purpose learning procedure, which employs layers of features not designed by humans but learned from data. On the contrary, previous techniques required careful engineering and a deep domain knowledge, making the result specific for a single task [12]. Recurrent Neural Networks (RNNs) are deep learning models used for tasks like Neural Machine Translation and Image Captioning [14]. Differently from normal feed-forward Neural Networks, that only produce outputs using the information extracted from current inputs, RNNs have an internal memory, which allows them to process sequences of data. Furthermore, they are able to handle variable length inputs and outputs, which yields improved results in sequence modelling tasks. Deep Learning development and widespread adoption were mainly favoured by the increased availability of computing power, which makes it possible to train these models in a relatively short time [24], despite the increasing complexity of neural network architectures needed to obtain higher accuracies. However, this increasing complexity and hardware requirements of the models caused them to become too resource hungry to be run on low-power embedded devices, where the hardware capabilities are limited and there is no connection to the power grid.
The increasing popularity of edge-nodes, such as mobile phones and IoT sensors, leads to trying to bring Machine Learning models directly on those low power devices. While the training phase of the models should still be

performed on the cloud using high performing clusters of computers, decentralizing the inference phase and performing it directly on the device can provide significant benefits [24]. On the one hand, it can decrease the overall system latency (as data doesn't have to be sent to the cloud for processing) thus improving the responsiveness of the system. Moreover, as computation is typically more efficient than wireless transmission, the energy consumption on the edge node may also decrease. This trend has led to the design of new more energy efficient Deep Learning architectures and to the optimization of the inference phase to lower the complexity or the necessary hardware requirements [25].

In literature various approaches have been proposed in order to optimize deep learning models, such as hardware accelerators, which are specifically tailored to perform the computations required by a neural network [24]. This approach though has been, at the moment, explored mainly for models like Convolutional Neural Networks, whose operations can be easily parallelised. Various hardware solutions are present in literature while in the case of RNNs, where sequentiality plays an important role, very few optimized hardware solutions have been proposed [2]. Another approach to obtain more energy efficient networks is reducing the complexity of the models by using the *approximate computing* paradigm, which lowers the energy required by the network but results in a lower accuracy [24]. Various approximate computing have been proposed in literature, such as the binarization of the network, thus avoiding floating point operations which are very expensive to be performed on embedded devices [8]. Another possibility to reduce the complexity of the network is modifying the architecture of the model by dropping layers or neurons, thus reducing the number of computations required [24].

These approaches can be applied either *statically* or *dynamically* to the network. In the *static* approach the configuration of the network is kept unchanged during the whole execution [25]. This leads to suboptimal solutions since the network will tend either to under-approximate inputs that are harder to process, thus lowering significantly the accuracy, or over-approximate easier inputs, thus wasting energy. The *dynamic* approach on the contrary applies a different degree of approximation (e.g. binarization or quantization of the weights), depending on the current input characteristics and is based on the idea that not all the inputs are equally difficult to process for a network. This adaptation of the network configuration during runtime allows to achieve better accuracy-energy trade off than a static approach and it may even reach higher accuracies [20].

Pagliari et al. [9] propose a more energy efficient Encoder-Decoder network,

the state of the art RNN architecture for tasks like Neural Machine Translation. They introduce a dynamic Beam Search algorithm, changing its Beam Width (BW) depending on the difficulty of the input during the inference phase. The BW parameter influences heavily the network, in fact, an higher value improves significantly the translation accuracy but at the same time comes with an increased number of computations and network complexity. This work implements the dynamic Beam Search on a real embedded device, proposing a novel way to tune the BW depending on the input complexity and proving empirically its effectiveness by measuring the energy saved during the execution of the network with this approach. A reduction of the execution time of more than 20% was obtained on both the datasets used in this work. Furthermore, it was proved that the power consumption of the Encoder-Decoder architecture doesn't change over the whole inference phase, due to its continuously repeated decoding operations. Thus, a reduction in the execution time corresponds to a comparable energy saving.

Chapter 2

Background

2.1 Overview

Deep neural networks (DNNs) have recently become the focus of many works in the machine learning field [25] thanks to the increasing availability of larger datasets, increasing computing power and easy to access and open-source frameworks [24]. Their ability to discover intricate structures in high-dimensional data [12] makes them ideal in many fields, like speech and image recognition, where not only they are superior to the other methods but manage to exceed human accuracy [24] in some cases. Furthermore, an additional perk of these methods is that their higher precision comes with still relatively short periods of time to train.
Before the advent of the DNNs the machine learning techniques were limited by the need to process natural data in a raw form, that made careful engineering and high domain expertise necessary to obtain a suitable internal representation which a classifier could then use [12]. This long and laborious work, which had to be repeated for every problem in a domain, can be substituted by a representation learning method. This set of methods is able to be fed directly with raw data and to discover automatically the representation needed for detection or classification.
Deep learning uses representation-learning methods with multiple levels of representations, obtained by composing non-linear modules that transform the representation at one level. These data representation transformations are called layers and by stacking many of them, even very complex functions can be learned. Furthermore these layers are created through a general learning procedure instead of being designed by human engineers [12].
Deep learning, as the majority of other machine learning algorithms, is more

commonly used with one of the two main methods to detect and extract features: *supervised learning*. In this case the network is fed with data labelled with the respective class, so that the network will try to extract the characterizing features and will be able to recognize the same features when unlabelled data is provided. In order to do so, the data is fed to the network multiple times at training time and the output obtained is compared with the expected output. A loss function calculates how far the output is from the expected ones and the parameters of the network are modified in order to reduce this "distance" in a process called back-propagation.
Other forms of "learning" are *unsupervised learning* which, starting from unlabelled data, tries to group those with similar characteristics and *reinforcement learning*, where the so-called "intelligent-agent" interacts with the environment and calculates the cost of its actions.
There are various different architectures of Neural Networks (NNs), each one born in order to address a specific set of problems:

- *Feed-forward neural networks*: the quintessential deep learning models, made of function approximation machines that are designed to achieve statistical generalization [5]. They are made of three types of layers: the Input layer, the Output layer and a set of Hidden layers. This architecture is mainly used for classification.

- *Convolutional neural networks*: feed-forward networks using one or more layers performing a mathematical operation called convolution instead of a normal matrix multiplication. They are now the dominant approach for almost all image recognition and detection tasks [12].

- *Recurrent neural networks*: neural networks specialized in processing sequences of values even with variable length. They possess a "memory state" of what was previously analysed, so that the output value can be chosen according to the current and the previous inputs.

The main architectures will be analyzed in more detail in the following sections, first an overview on the common elements composing the neural networks will be given.

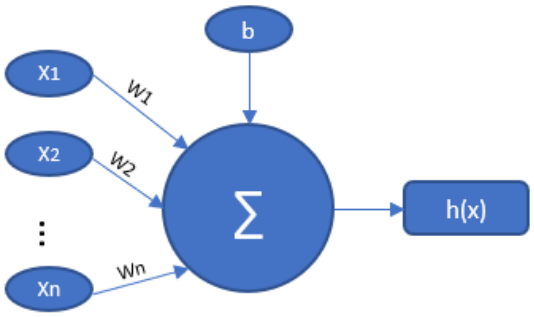


Figure 2.1: The neuron

2.2 Neuron

The artificial neural networks are composed of *neurons*, which are generic computational units that perform a weighted sum of their inputs (and eventually a bias). Then, in order to avoid having a simple linear algebra operation, a non-linear activation function is applied on the result of the weighted sum. Figure 2.1 shows a graphical representation of a neuron whose formula is:

$$y_i = h(\sum_{i=1}^{n} w_i x_i + b) \tag{2.1}$$

where w_i are the weights applied to the respective input x_i, b is the bias and h the non linear activation function which is applied on the output of the weighted sum.
The weights w_i are updated during the training phase, while the bias is used to shift the activation function on the x plane in order to increase the learning flexibility. If all the outputs of the layers are connected to the input of the next one, the network is called *fully connected*.
Among the possible activation functions for the neurons, the following are the most commonly used [12]:

- **Step function**

$$h(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases} \tag{2.2}$$

A binary function (2.2) that manages only the cases in which the neuron *fires* (so it has an output) or not. This is very limiting if we want to understand how confident about the classification the neuron was. Furthermore, an additional issue is the impossibility to derive the function for $x = 0$ during the training time.

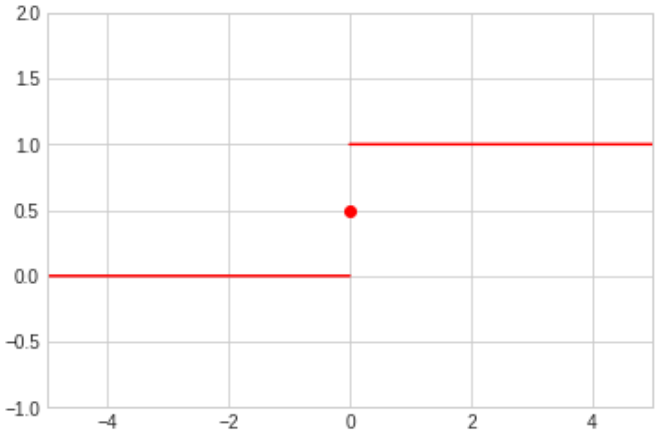


Figure 2.2: The step function plot

- **Logistic function**

$$h(x) = \frac{1}{1 + \exp(-x)} \tag{2.3}$$

The logistic or sigmoid function (2.3) is one of the most conventional activation functions used in the last years [12]. It allows to obtain an output in the interval (0,1), but a network using the logistic sigmoid may have its backpropagated gradient vanish or explode quickly [27].

- **Hyperbolic tangent function**

$$h(x) = \tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} \tag{2.4}$$

The hyperbolic tangent function (tanh) has very similar characteristics to the logistic one but its gradient is much more stable. Even if it is still a saturated function, the saturation value of the hyperbolic tangent is much lower than the one of the logistic function.

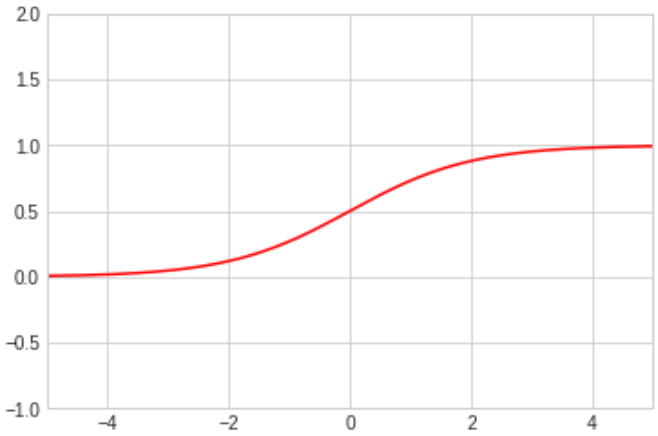


Figure 2.3: The logistic function plot

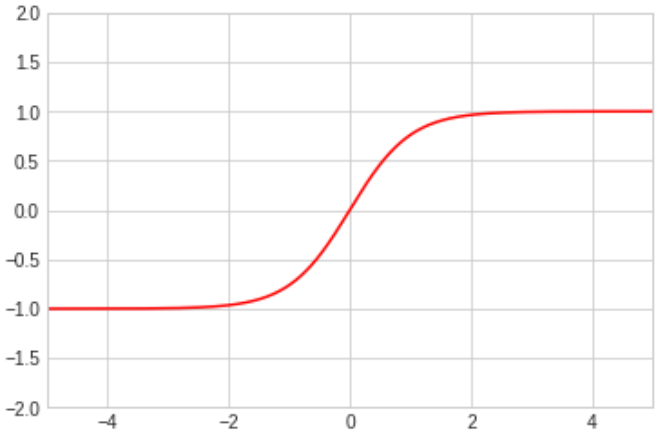


Figure 2.4: The hyperbolic tangent function plot

- **Rectified Linear Unit**

$$h(x) = \max(0, x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} \tag{2.5}$$

The rectified linear unit (ReLU) is an activation function that doesn't saturate even when the input is very large. This solves the vanishing gradient problem of the previous functions and allows a much faster training of the network which uses it in the hidden layers. For these reasons, the ReLU has become in the last years one of the most used activation functions. The ReLU has one major problem though: the "dying ReLU". Some parts of the network will never fire and so the

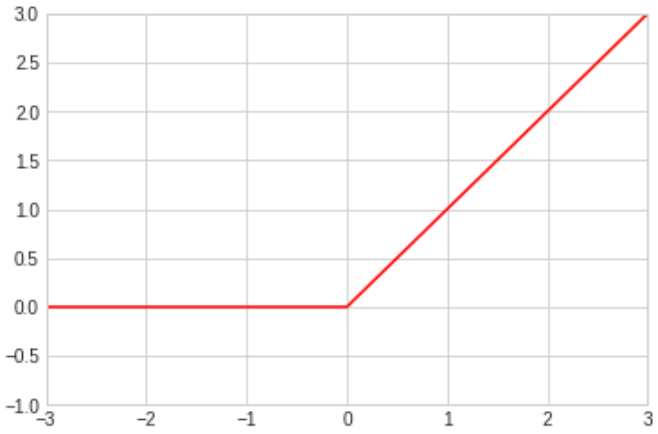


Figure 2.5: The rectified linear unit plot

weights of the neurons will never be updated. This issue can be solved by changing the function to a Leaky ReLU (2.6), where the output of the neuron for negative values is a small number and not zero, so that learning can be performed even on parts that rarely "fire".
The following becomes then the equation of the Leaky ReLU:

$$h(x) = \max(\alpha x, x) \qquad \alpha \in (0,1) \tag{2.6}$$

- **Exponential linear unit**

$$h(x) = \begin{cases} \alpha(\exp(x) - 1) & x \leq 0 \\ x & x > 0 \end{cases} \tag{2.7}$$

The exponential linear unit (ELU) has, like ReLU, been proposed very recently and it was designed mainly to solve the vanishing gradient problem. It still has negative values as output like batch normalization but with lower complexity. The ELU has the advantage over the ReLU to ensure a noise-robust deactivation state, with a lower forward propagated variation and information [3].

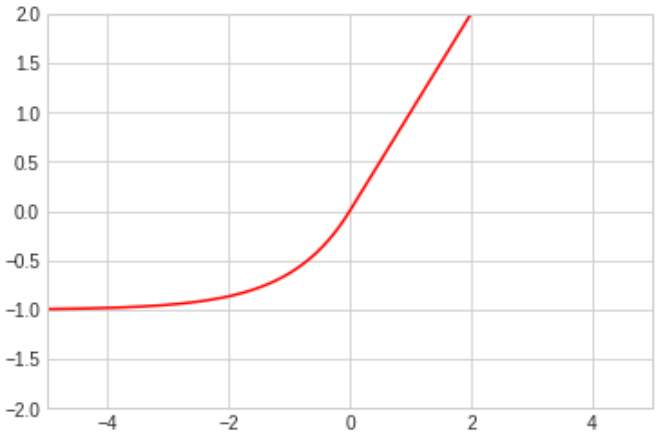


Figure 2.6: The exponential linear unit plot with $\alpha = 1$

2.3 Layers

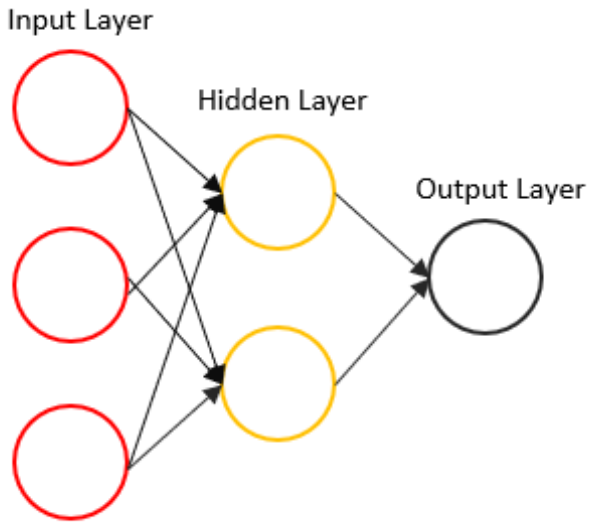


Figure 2.7: An overview of a fully-connected network and its layers.

Layers are aggregations of neurons and are mainly divided into three categories:

- *input layer*: receiving the input data in a suitable format to be processed by the network. Each input element represents a variable and is fed to one of the neurons composing the layer.

- *output layer*: providing the result of the neural network classification or regression. In case of classification the number of neurons composing this layer depends on the number of classes to predict, each node will give the confidence that the output belongs to that class. In this layer the activation function to be used depends strictly on the problem to be solved, the most common ones are the softmax [14] and the logsoftmax.

- *hidden layer*: the layers that are present in the middle and whose output is not directly the class to predict. Each one of them applies a function to the input data, allowing to detect relations not visible during the previous steps and so, to extract new features. These layers perform the heaviest calculations of the network, requiring the majority of the computing power during the execution.

The number of neurons composing an hidden layer has an impact on the performance of the network, in fact, if they're too few it will cause *underfitting*, in which case the model won't be able to recognise patterns. In the opposite case, so if the number of neurons is too high, the model will *overfit*, becoming unable to generalize (causing high accuracies on training data and low ones on test data).
The number of neurons present in a layer gives the model *width*, while the number of layers is the *depth*. In case of multiple hidden layers the network can be called *deep*. The majority of the networks nowadays is a Deep Neural Network since the average number of layers of the models has changed in the last years, going from one hidden layer to a range from five to more than a thousand [24].

2.4 Neural Network training

To be able to predict the correct values Neural Networks have to undergo a learning or training phase, in which the weights are updated according to the features extracted and the patterns found in the data are fed to the network. In order to penalise the network when a wrong prediction is made in the case of supervised learning, a cost or *loss* function $\mathcal{L}$ is chosen and will be used at each step. The most common approach to optimize this function is the maximum likelihood estimation (MLE) (2.8): during the learning the networks weights w that minimise the error E between the known training values y_i and the predicted training values $f_w(x_i)$ are sought. For instance, if the loss function used is the Mean Squared Error between the known and

predicted output (a typical choice for regression problems), then:

$$E(w) = \sum_{i=1}^{N} (y_i - f_w(x_i))^2 \tag{2.8}$$

The training phase can be split in two main parts, during the first one the network is fed with an input x and will output, after that the vector has propagated in all the layers of the network, a predicted value $f_w(x)$. This will continue until a scalar cost $\mathcal{L}(\theta)$ is produced [5]. In the second phase the weights of the network have to be updated according to the loss obtained in the previous step; this process usually happens through the optimization process called *gradient descent*. The partial derivative with respect to the weight of the loss is calculated and the weight is modified by a small multiple of that value [24]. This gradient (2.9) indicates the change required to lower the loss. This two-step process is repeated iteratively until a certain stopping criterion is met, e.g. a given value of the loss is obtained.
The gradient calculation, while analytically simple, may become very expensive to compute numerically, slowing down the whole training phase. To avoid this, the **backpropagation**, a process derived from the chain rule for derivatives, is used to calculate the gradient because of it being simple and computationally inexpensive. It consists in passing the gradient back to the network and it's computed at each layer, starting from the output one. Once these gradients have been computed, the derivative of the loss is multiplied by α, the *learning rate*, and used to update the weights. Equation (2.9) shows the formula used to update the network weights. The learning rate can be seen as the height of the jump while descending a function; if this value is too big it will miss the minimum (overshooting) while if it is too small it will get stuck in local minima.

$$w_{ij}^{t+1} = w_{ij}^{t} - \alpha\left(\frac{\partial \mathcal{L}}{\partial w_{ij}}\right) \tag{2.9}$$

This backpropagation technique, while being very fast, doesn't avoid the possibility of getting stuck in a local minimum, but with a sufficiently large dataset the chance of it happening is very low [12]. Furthermore this technique requires the network to store the intermediate outputs, in order to be preserved for backwards computations, increasing the storage requirement [24].
Once that the learning phase has concluded, the network can be used to predict values from data whose class is unknown. This process of **inference**

consists only of the forward pass of the network, since the labels are unknown and the loss cannot be calculated any more, so the weights are unchanged during this phase.

2.5 Network architectures

2.5.1 Feed-forward neural networks

Feed-forward networks are the simplest neural networks, which ultimate goal is to approximate some function f^*. They define a mapping $y - f(x;\theta)$ and learn the values of the parameter θ resulting in the best function approximation. The models are called feed-forward because there is no feedback connection feeding the outputs at various steps back to the network.
The name *network* derives instead from the fact that they are typically represented by the composition of many different functions, in fact the model is associated with a directed acyclic graph describing how the functions are composed together [5].

2.5.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a specialized type of neural networks for processing data with grid-like topology [5], that due to their outstanding results in the last years, have become the state of the art for almost all the image recognition and detection tasks [12]. Their huge success has been made possible thanks to the introduction of GPUs for computations (which allow greater parallelization than CPUs and so a huge increase in computational power), thanks to the new normalization technique called *dropout* and a technique to generate new data from the existing ones [12].
The name of this network derives from the *convolution*, a mathematical operation on two functions of a real-valued argument [5]. This operation is implemented by a type of hidden layers called **convolutional layers**, whose formula paired with a ReLU is (2.10). In the equation $h_k^{(n-1)}$ is called input feature map, w_{kj}^n is the kernel and h_j^n is the output feature map. The output can be seen as an abstraction of the input, where only the most meaningful features are left.

$$h_j^n = max(0, \sum_{k=1}^{K} h_k^{n-1} * w_k j^n) \tag{2.10}$$

This operation performed by these layers can be seen as a set of matrix multiplications (as shown in Figure 2.8). Convolution brings several improvements to a machine learning model: it lowers the memory usage, the number of computations and allows the network to manage inputs of different sizes. CNNs use a kernel (or filter) smaller than the input data, which allows to detect small features and reduces the number of parameters to be stored. Furthermore the weights of the network are shared, so they are reused for more than one neuron, lowering even more the memory usage.

Another addition of the CNNs is the **pooling layer**, which applies a sliding

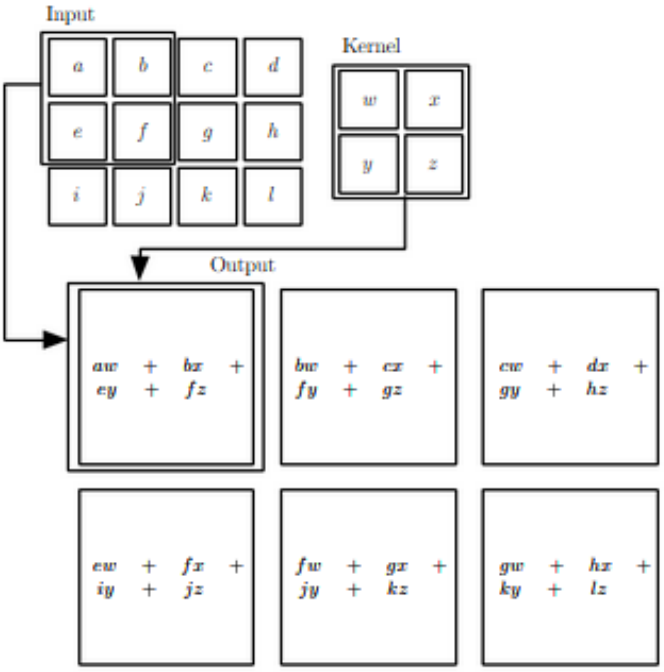


Figure 2.8: An example of convolution applied by a layer on the input [5]

window on the input and outputs only summary statistics of it. Two of the most common pooling types are the *max* and the *average* pooling. These layers can output a matrix of any size and can be used to manage inputs of different dimensions. Furthermore pooling reduces the memory required by the network since allows to discard features impacting less the prediction. The **batch normalization layer** is used in order to stabilize the distribution of the inputs by working on their mean and variance. This technique makes the gradients used in the training phase more predictive, improving the accuracy and lowering the time required for training [23]. The **dropout**

layer is used to avoid overfitting the network during the training time while it is not used during the inference. It allows to switch off some neurons in order to avoid co-adaptation, increasing the network generalization capability.

Figure 2.9 shows an example of CNN, where the final layers are fully-

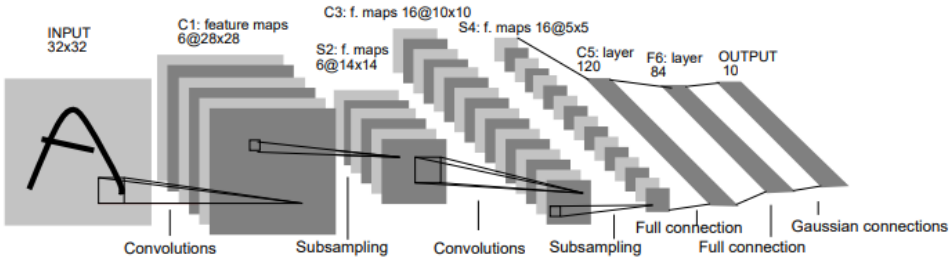


Figure 2.9: The CNN called LeNet-5 [11]

connected in order to predict the classes.

2.5.3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a type of neural network specialised in processing sequential data. They are able to process sequences of values $x^{(1)}, \ldots, x^{\tau}$, that can be longer than for other non-specialised networks but that can also change in length [5]. While the input sequence is still analysed one item at a time like for feed-forward networks, the hidden units keep a "memory" vector containing all the information about the past items of the sequence. The time-stamp index t of the sequence of elements $x^{(t)}$ is not always related to time but just the position in the sequence that, as long as it fully observed before being provided to the network, can even go backward in the time of the real word or be completely unrelated to it. The main fields of application for the RNNs are:

- **Image Recognition and characterization**: if RNNs are paired with CNNs, they can be used to first recognise the image fed to the network and then give it a description.
- **Machine Translation**: handling the translation between languages with different topologies and idioms.
- **Language modelling**: in this case RNNs are used in order to find the next word in a sentence.

In order to define the graphs of the RNNs and their structure, first the computational graphs need to be able to represent cycles, which are needed to show the influence of the current variable on itself in a future time step.

Graph Unfolding

This technique is used in order to display a recursive or recurrent computation by using a *computational graph*, which is a way to formalise the structure of a set of computations. This notation represents network parameters as nodes and as arrows the interconnections among the computations done on the nodes. The computational graph of a normal feed-forward network or of a convolutional neural network, is a simple directed acyclic graph. The unfolding technique is usually used on a chain of events and results in having shared parameters through the whole DNN structure.
For example given the computational graph of a recurrent neural network hidden units, since the definition of h at time t refers back to a definition at a previous time step, it can be said that the graph is recurrent.

$$h^{(t)} - f(h^{(t-1)}, x^t; \theta) \tag{2.11}$$

In equation (2.11), h represents the state and will be used by the network output layer as a summary of the important features of the past sequence of inputs, θ the state parameters, x the input and f the mapping function. In the case of RNNs, f represents the neural network used to perform the prediction of the next element based on the current value (shown as the arrows in the unfolded graph in Figure 2.10). The summary is defined as *lossy* since it maps a variable length sequence to a fixed length one h^t [5]. In order to unfold (2.11), each variable at every time step will be represented by a node of the unfolded graph, which will have size equal to the sequence length and many repeated parts.
The equation of the state at the step t can then be written as:

$$h^{(t)} - g^{(t)}(x^{(t)}, x^{(t-1)}, \cdots, x^{(1)}) \tag{2.12}$$

The function $g^{(t)}$, by using the previous inputs, produces the current state and the unfolded graph allows its factorization by repeated application of f. This factorization allows the network to manage different sequence lengths as input. Furthermore it allows to use the same function f with the same parameters for all time steps [5]. It is then possible to learn a single model that can manage all the time steps and sequence lengths instead of a different

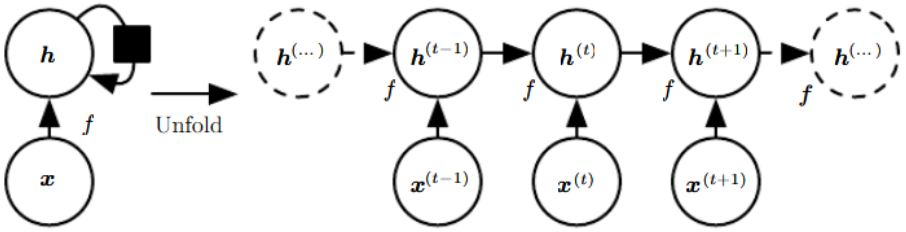


Figure 2.10: The recurrent neural network with no outputs described in (2.11). On the left the recurrent graph and on the right its unfolded computational graph [5].

one for each time step. The main advantage of the recurrent graph is its capability to summarize what the network is doing, while the unfolded graph is explicit and allows to visualise immediately the path of the information and its flow forward and backward in time.

2.5.4 Recurrent Neural Networks Design Patterns

Due to the success of RNNs in many fields, multiple architectures were designed over time, each one focused on performing a specific task. The most important and common patterns [5] can be divided in three classes differentiating themselves for their output and their loop back connections.
They are:

- Recurrent networks producing an output at every time step and that have recurrent connections among hidden units.
- Recurrent networks producing an output at every time step, with recurrent connections from the output to the hidden units only of that time step. This architecture is less powerful than the previous one but it may be trained faster since each time step can be trained independently from the others.
- Recurrent networks that have connections among hidden units and that once read a whole sequence produce a single output. They can be used to summarize a sequence and produce a fixed-length representation of it.

In the following sections the training of a RNN of the first type will be seen in more detail, since the architecture used in this work belongs to this category.

2.5.5 Recurrent Neural Networks Training

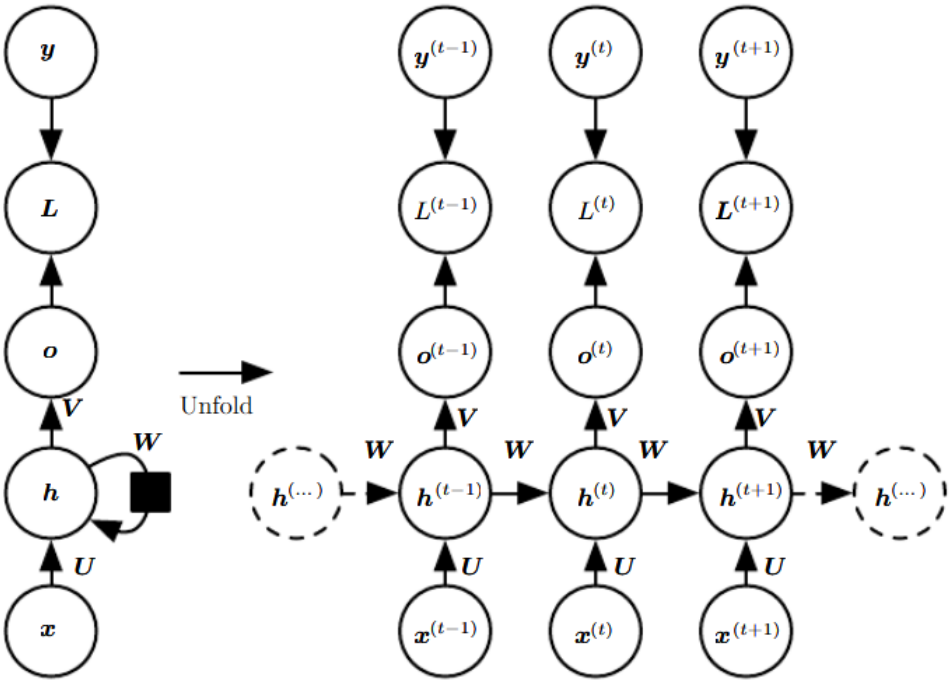


Figure 2.11: An unfolded recurrent neural network [5].

The training of RNNs has long been considered an harder problem than for standard feedforward networks due to the need to learn long-range dependencies. This causes the hidden layers the need to have a "memory" of the previous inputs, increasing the complexity of the networks architectures. Furthermore problems like *vanishing* and *exploding* gradients occur even more frequently than with standard DNNs when propagating back the errors across multiple time steps [14].
Due to this additional level of complexity now an overview of the learning phase in a recurrent neural network will be given.
The training phase of a RNN can be divided, as for the feed-forward networks, in two main parts: the forward propagation and backward propagation. Considering the network depicted in Figure 2.11 and assuming a discrete output, as in the word prediction case, we will describe the output o as the unnormalized log probabilities of each value that the discrete variable can assume. We will then suppose to apply a softmax operation to obtain a vector of normalised probabilities.

The **feed-forward** phase starts at the initial state $\mathbf{h}^0$ and then at each time step t from 1 to τ a set of update equations is applied:

$$\mathbf{a}^{(t)} = \mathbf{b} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)} \tag{2.13}$$

$$\mathbf{h}^{(t)} = \sigma(\mathbf{a}^{(t)}) \tag{2.14}$$

$$\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)} \tag{2.15}$$

$$\hat{\mathbf{y}}^{(t)} = softmax(\mathbf{o}^{(t)}) \tag{2.16}$$

In these equations $\mathbf{b}$ and $\mathbf{c}$ are the bias vectors, while $\mathbf{U}$, $\mathbf{V}$, $\mathbf{W}$ are are the weight matrices respectively for the connections between input to hidden state, hidden to output state and hidden to hidden state. Finally σ is the non-linear activation function of the layer [5].
The equation (2.17) shows how the total loss is derived given a series of $\mathbf{x}$ values together with a series of $\mathbf{y}$ values.

$$\begin{aligned} &\mathcal{L}(\{\mathbf{x}^1, \cdots, \mathbf{x}^\tau\}, \{\mathbf{y}^1, \cdots, \mathbf{y}^\tau\}) \\ &= \sum_{t=1}^{\tau} \mathcal{L}^{(t)} \\ &= -\sum_{t=1}^{\tau} \log p_{model}(y^{(t)} | \{\mathbf{x}^1, \cdots, \mathbf{x}^t\}) \end{aligned} \tag{2.17}$$

The total loss is then the sum of all the losses over all the time steps, where $p_{model}(y^{(t)}|\{\mathbf{x}^1, \cdots, \mathbf{x}^t\})$ is obtained from the vector of output $\hat{\mathbf{y}}^{(t)}$ by taking the value for $y^{(t)}$. This whole propagation is sequential and then it can't be parallelised. To compute the following time step, it is in fact necessary to compute the previous first. Furthermore all these states computed in the forward pass must be stored in order to be used during the next training phase, increasing the memory required with respect to standard neural networks.
The second part of the training is the computation of the gradient and the backward propagation. In this case the backpropagation technique has to be adapted to work with sequences and takes the name of **back-propagation through time** (BPTT). For each node $\mathbf{N}$, the gradient $\nabla_N L$ is computed in a recursive way based on the gradient of the nodes following it in the graph. The recursion starts at the node preceding the final one:

$$\frac{\partial L}{\partial L^{(t)}} = 1 \tag{2.18}$$

Assuming to use the same parameters used for the equation (2.13) and that the loss is the negative log-likelihood of the true label $y^{(t)}$, the gradient can

then be calculated for each of the previous steps. For example the gradient $\nabla_{\mathbf{o}^{(t)}} L$ calculated at the step t for the output is:

$$(\nabla_{\mathbf{o}^{(t)}} L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - \mathbf{1}_{i=y^{(t)}} \tag{2.19}$$

This derivation repeated for every time step can cause the gradient either to *vanish* or to *explode*. In the first case it means it will approach to zero exponentially fast, making the network unable to properly "learn" long term dependencies because of too small weight changes [7].
The opposite problem is when the gradient *explodes*, so it grows exponentially fast. This causes the network to be unable to learn correctly as well, by making the whole model unstable; the model loss will in fact vary greatly between updates and may go to NaN value.
Either of this two problems can happen during the back-propagation time depending on the weights of the current recurrent edge and on the activation function σ chosen [14]. To avoid the vanishing or the explosion of the gradients, the following techniques can be used:

- **Global search methods**: like simulated annealing, that do not depend on gradients but that are effective only for short sequences [7].
- **Weight matrix initialisation**: the weight matrix is initialised as an identity matrix instead of using random values.
- **Truncated backpropagation through time (TBPTT)**: a solution to the exploding gradient, that consists in setting a maximum number of steps for the error propagation. This approach makes the network unable to learn longer term dependencies.
- **Long Short-Term Memory(LSTM) networks**
- **Gated Recurrent Units (GRU) networks**

The LSTM and the GRU networks will be described in the following section, which will give an overview of the most common RNN architectures.

2.5.6 Recurrent Neural Network architectures

LSTM

This architecture extends the basic hidden unit and allows to improve the learning of long-term dependencies while managing the problem of the vanishing gradient described previously. The weight of the self-loop that is

present in the RNN cell graph (shown in Figure 2.12) is no more fixed but will be now dependant on the context. This "gating" (controlled by another hidden unit) allows to dynamically change the scale of integration based on the input sequence.
This approach has been seen successful in various fields: speech recognition, machine translation, handwriting recognition [5]... The LSTM cells have

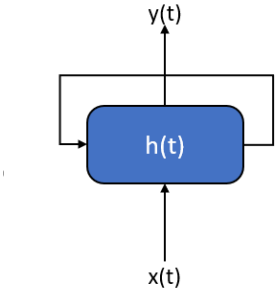


Figure 2.12: The recurrent cell of a RNN.

the same input and the same output of the normal recurrent neural networks but internally, a set of additional parameters and gating units are used in order to control the information flow. Furthermore they possess an internal recurrence working alongside the external one of the RNNs. Figure 2.13 shows the inner structure of a LSTM cell. First the input feature is computed with a normal neuron and, if the input gate allows it, its value is stored in the state, which is the "memory" of the cell. The state possesses a self-loop connected to the forget gate, which will handle the weights. Another important element is the rectangle on the self-loop which instead indicates a delay of one time step. Finally there is the output of the cell, which is controlled by the output gate, that can even decide to shut it down completely depending on the weights. All the gating units have a sigmoid non-linearity while the input can have any squashing non-linearity.
The main part of the LSTM is the state unit s_i^t, controlled by a forget gate unit $f_i^{(t)}$ (where i is the cell and t is the time unit) which sets the weight in the interval [0,1] with a sigmoid activation function.

$$f_i^{(t)} = \sigma(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}) \tag{2.20}$$

In the state unit function (2.20) the following variables are present: $\mathbf{x}^{(t)}$ is the input vector at the current time step, $\mathbf{h}^{(t)}$ is the current hidden layer

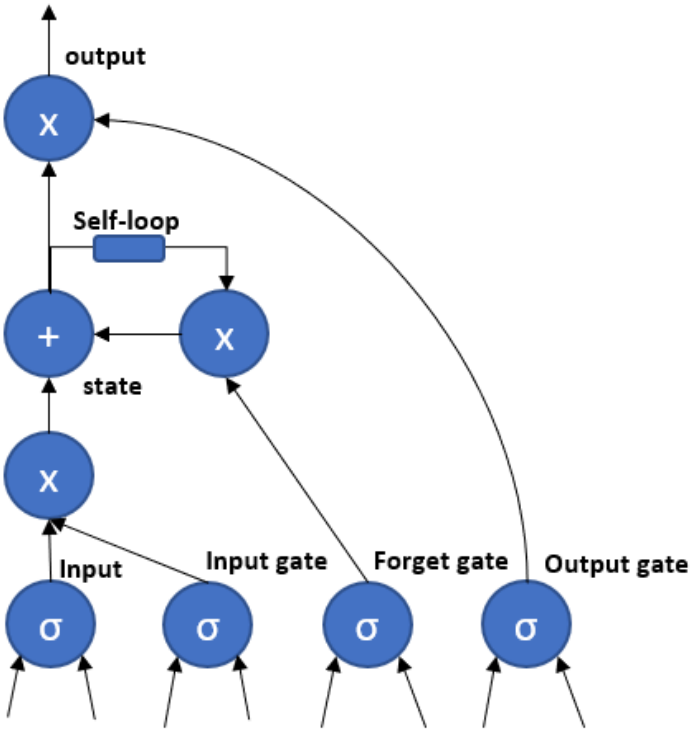


Figure 2.13: The LSTM cell.

vector, containing the outputs of all the LSTM cells. $\mathbf{b}^f$,$\mathbf{W}^f$ and $\mathbf{U}^f$ are respectively the biases, recurrent weights and input weights for the forget gates.
The state of the cell and its biases and weights are then updated using equation (2.21):

$$s_i^{(t)} = f_i^t s_i^{(t-1)} + g_i^t \sigma(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}) \tag{2.21}$$

where $\mathbf{b}$,$\mathbf{W}$ and $\mathbf{U}$ are the biases, the recurrent and the input weights in the LSTM cell. Similarly for the external input gate $g_i^{(t)}$ (2.22), a sigmoidal activation function is used in order to obtain values between 0 and 1.

$$g_i^{(t)} = \sigma(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}) \tag{2.22}$$

Finally the output of the LSTM cell $h_i^{(t)}$ can be changed by the output gate $q_i^{(t)}$, which uses as well a sigmoidal activation function for gating (2.23).

$$\begin{aligned} h_i^{(t)} &= tanh(s_i^{(t)}) q_i^{(t)} \\ q_i^{(t)} &= \sigma(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)}) \end{aligned} \tag{2.23}$$

GRU

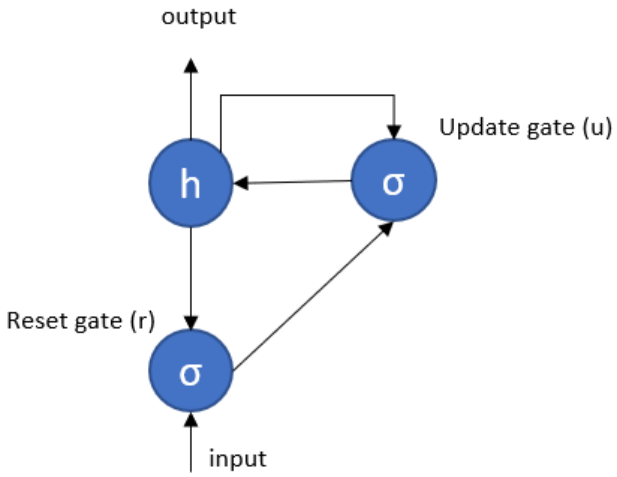


Figure 2.14: The GRU cell.

GRUs or gated recurrent units are a simplified version of the LSTM cells. They still give the possibility to control the time scale and enhance the long term dependencies learning, but with a less complex gating system. In fact instead of multiple gating units controlling the forgetting part of the cell and the update, only one is used.
The new update equation is the following:

$$h_i^{(t)} = u_i^{(t-1)} h_i^{(t-1)} + (1 - u_i^{(t-1)}) \sigma(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} r_j^{(t-1)} h_j^{(t-1)}) \tag{2.24}$$

The functions **u** and **r** represent respectively the update gate and the reset

gate, whose value is defined as:

$$u_i^{(t)} = \sigma(b_i^u + \sum_j U_{i,j}^u x_j^{(t)} + \sum_j W_{i,j}^u h_j^{(t)}) \tag{2.25}$$

$$r_i^{(t)} = \sigma(b_i^r + \sum_j U_{i,j}^r x_j^{(t)} + \sum_j W_{i,j}^r h_j^{(t)}) \tag{2.26}$$

The two gates can act independently from each other and "switch off" different parts of the state vector. In particular the update gates act as conditional leaky integrators which are able to gate any dimension. They can either copy (one of the sigmoid extrema) or decide to ignore (the opposite sigmoid extrema) by substituting the input with the target state value to which the leaky integrator tries to converge. The reset gate adds an additional non-linearity between states by deciding the parts of the state that are used to compute the successive target state.

Bidirectional networks

This network architecture is able to capture not only information from the past and the present input but also the ones from future inputs. $\mathbf{y}^t$ depends in fact from the whole input sequence $\mathbf{x}$, enhancing the results obtained for many applications like handwriting recognition and sequence-to-sequence learning [5].
This architecture is based on two RNNs; one moving forward through time and an other going backward (starting from the end of the sequence). Figure 2.15 shows the structure of the bidirectional network, where $\mathbf{h}$ denotes the forward time propagation while $\mathbf{g}$ the backward one. Then, the output $\mathbf{o}$ will depend on past and future with more sensitivity around the current time step t. The same idea can be extended to images, which are two dimensional, by increasing the number of RNNs to four. While more expensive than convolutional networks, bidirectional networks can allow long range interactions between features in the same feature map [5].

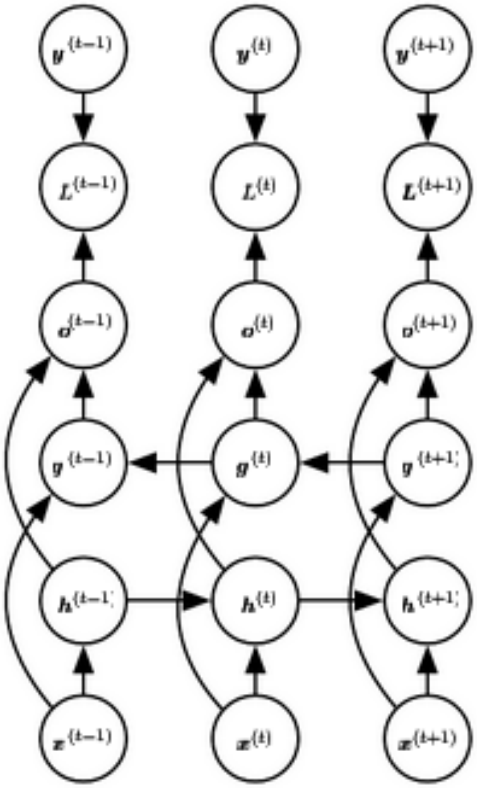


Figure 2.15: The bidirectional RNN computations [5]

Recursive neural networks

Recursive neural networks are a generalization of the recurrent neural network, using a deep tree structured computational graph instead of the classic chain-like one. They have been used mainly in processing data structures as inputs, natural language processing and computer vision [5]. The advantage of this architecture is the possibility to reduce the length τ of a sequence to $O(\log(\tau))$, helping in the process of learning long-term dependencies.
A particular focus has to be put on how to structure the tree since it has a considerable impact on the learning, but this choice is not necessarily dependent on the input data (even balanced binary trees can be chosen to represent the structures). The "best" data tree structure can sometimes be inferred from external sources (like a parser for natural language processing), but ideally should be discovered directly by the learner [5].

Encoder and Decoder

The architectures described previously were able to manage input sequences of fixed or variable length and give as output a sequence respectively of the same length or of a fixed length. This architecture instead is able to map an input sequence to an output sequence of different length, which is ideal for many applications like speech recognition, question answering and machine translation.
The input given to the RNN is often called "context" (C) and to represent it, a vector or a sequence of vectors that summarize the input $\mathbf{X} = (\mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(n_x)})$ can be used. In the field of Neural Machine Translation, often *embedding* layers are used in order to represent efficiently sentences and feed them to the network. This kind of layers substitutes the previously used one-hot encoding or bag-of-words technique, which had very high memory cost, with a more compact representation that manages to capture the semantic meaning and similarity in the context (in this case the words will be mapped in closer vector spaces).

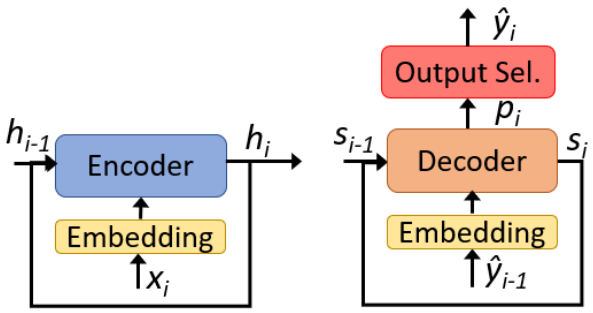


Figure 2.16: An high level overview of an encoder-decoder RNN, both the encoder and the decoder are composed by an embedding layer and multiple hidden layers [9].

The encoder-decoder architecture consists of two distinct recurrent neural networks (as shown in Figure 2.16), each one composed by multiple layers (usually using LSTM or GRU) and with completely different tasks. The *encoder*, also called *reader*, reads the whole input sequence and by processing it, it emits the context C, which is usually a simple function of its final hidden state [5].
The *decoder* task is to predict the final sequence $\hat{\mathbf{Y}} = \hat{y}^{<1>}, \cdots, \hat{y}^{<T'>}$ by

taking as input the context C from the encoder. The update function of the decoder becomes then:

$$h^{(t)} = f(h_{(t-1)}, \hat{y}_{(t-1)}, C) \tag{2.27}$$

The update function will depend not only on the context but also on the input and the hidden state at the previous time step. Figure 2.17 shows an

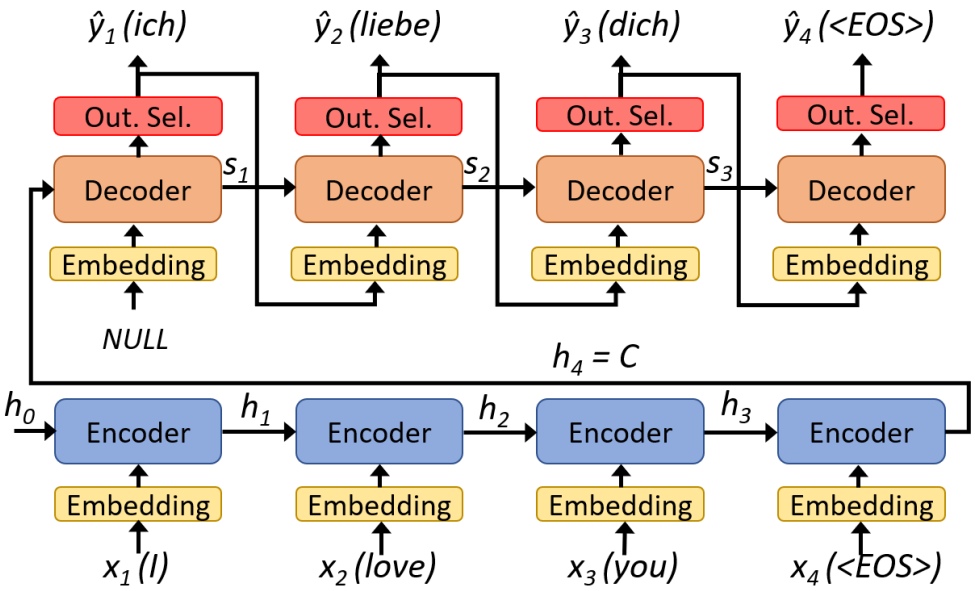


Figure 2.17: The unfolded graph of an encoder-decoder architecture during the translation [9]

example of an unfolded network during the training phase; where a sentence is given as input to the encoder, one word per step. The encoder will keep reading words and update the hidden state until an *EOS* token is received, then it will output the context C.
The decoder receives as input the word translated in the previous step (*NULL* in the first one) and the updated state. The red **Out. Sel.** rectangle purpose is to choose the most likely (based on the Decoder predictions) word at a given time step (it will be described in detail in section 2.5.6).
The two networks are usually trained jointly in order to maximize the average of $\log P(\hat{y}_{(1)}, \cdots, \hat{y}_{(n)} | x_{(1)}, \cdots, x_{(n)})$ and once that the training phase has been performed, the network is able to output sequences given as input sequences not seen during the learning.

Attention

A fixed-size representation of a very long sentence that manages all the semantic details is difficult to achieve without training large RNNs for a considerable amount of time [5]. To overcome this problem without an excessive use of memory or too long training times, the *attention* technique was introduced. Sentences are read fully once in order to understand the context, then the words are translated one per time while focusing on different parts of the input sequence. An additional memory is then necessary to keep track of the attention and it is derived from the hidden state h_t, the context vector c_t and the previous output $\hat{y}_{(t-1)}$ [15] by concatenating the two in the following way:

$$\tilde{\mathbf{h}}_{(t)} = \tanh(\mathbf{W}_c[\mathbf{c}_t; \mathbf{h}_t]) \tag{2.28}$$

This attention memory is then fed to a softmax layer in order to obtain the predictive distribution of the next word, the formula is:

$$p(y_t, y_{<t}, x) = softmax(\mathbf{W}_s, \tilde{\mathbf{h}}_{(t)}) \tag{2.29}$$

There are multiple ways to compute the attention, which change name depending on which part of the input sentence is considered to compute the context vector c_t. They can be divided in two main categories:

- *global attention*: all the hidden states of the encoder are considered when deriving the context vector c_t. $\mathbf{a}_t$ represents a variable-length alignment vector with same size as the number of time steps on the source side, which is obtained by comparing the current target hidden state ($\mathbf{h}_t$) with each source hidden state $\bar{h}_s$:

$$\mathbf{a}_t(s) = align(\mathbf{h}_t, \bar{\mathbf{h}}_s) = \frac{\exp\left(score(\mathbf{h}_t, \bar{\mathbf{h}}_s)\right)}{\sum_{s'} \exp\left(score(\mathbf{h}_t, \bar{\mathbf{h}}_{s'})\right)} \tag{2.30}$$

  In equation (2.30) the function *score* is content-based and has three different alternatives:

$$score(\mathbf{h}_t, \bar{\mathbf{h}}_s) = \begin{cases} \mathbf{h}_t^T \bar{\mathbf{h}}_s & \text{dot} \\ \mathbf{h}_t^T \mathbf{W}_a \bar{\mathbf{h}}_s & \text{general} \\ \mathbf{v}_a^T \tanh \mathbf{W}_a[\mathbf{h}_t; \mathbf{h}_s] & \text{concat} \end{cases} \tag{2.31}$$

- *local attention*: this approach solves the global attention issues in translating long sentences or whole chapters, which would require an impractical memory and computational overhead since for each target word

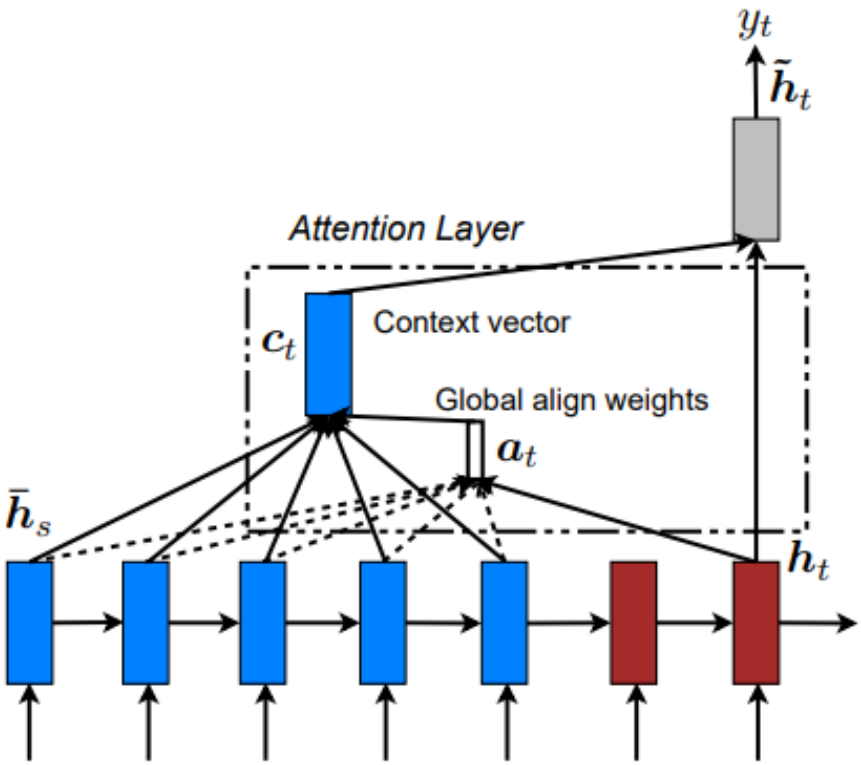


Figure 2.18: An overview of how the global attention mechanism works [15]

all the input words would have to be considered [15]. By considering only a small subset of the input sequence, computations are made less expensive without loosing the differentiability.
The model first generates p_t, an aligned position for each word at time t. Then c_t is obtained from the weighted average over the source of hidden states in the window $[p_t - D, p_t + D]$, with D empirically selected [15]. While the dimension of the alignment vector $\mathbf{a}_t$ changed according to the input for the global attention, its size becomes fixed (R^{2D+1}) thanks to the window.
There are two possible variants of the model : the *monotonic* or the *predictive* alignment. The first sets $p_t = t$ and assumes source and output sequences to be aligned monotonically. The alignment of the vector $\mathbf{a}_t$ is described by (2.31). The other model variant instead predicts aligned positions with the following computation [15]:

$$p_t = S * sigmoid(\mathbf{v}_p^T \tanh(\mathbf{W}_p \mathbf{h}_t)) \tag{2.32}$$

In this formula $\mathbf{W}_p$ and $\mathbf{v}_p$ are used to predict positions and have to be learned by the model, while S is the sentence source length. Furthermore since a sigmoid function is used, p_t lies in the interval $[0, S]$.
The alignment weights are defined by the following formula, placing a Gaussian distribution with centre in p_t (this favours the points near the centre) :

$$\mathbf{a}_t(s) = align(\mathbf{h}_t, \bar{\mathbf{h}}_s) \exp\left(-\frac{(s - p_t)^2}{2\sigma^2}\right) \tag{2.33}$$

The align function is the one previously defined (2.30) with a standard deviation equal to $\sigma = \frac{D}{2}$. In this equation, s is an integer value in the interval centred at p_t.

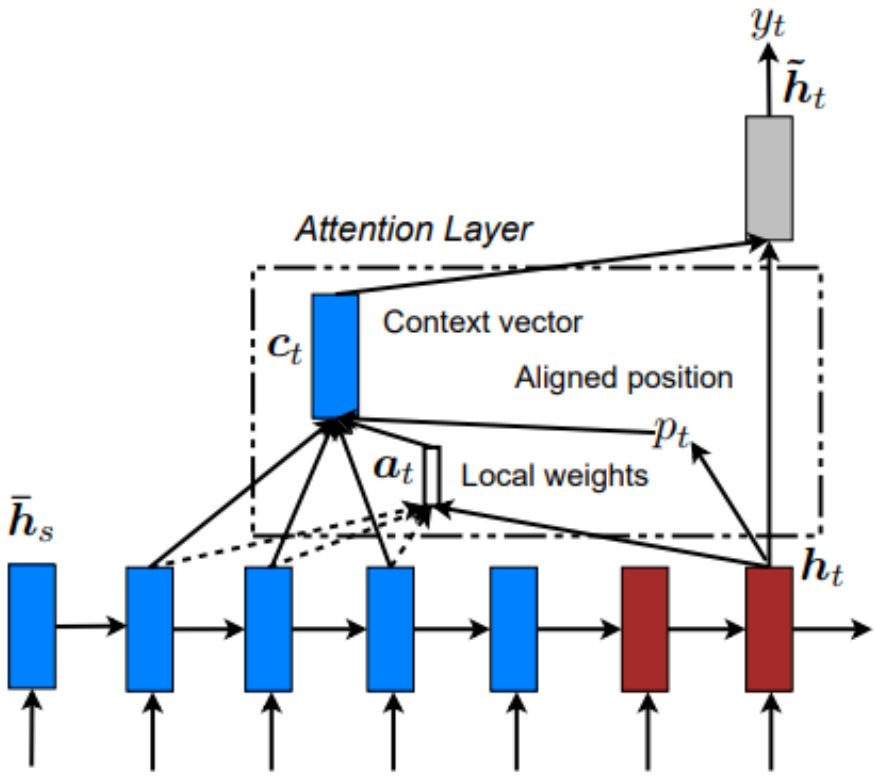


Figure 2.19: An overview of how the local attention mechanism works [15]

Both the *local* and the *global* alignment decisions made during translation should be, as proposed by [15], made jointly by taking into account past alignments as well. This requires additional information to be passed at each step of the network as input, so the attentional vectors $\tilde{h}_t$ are concatenated

with the next time step inputs, as shown in Figure 2.20. This approach takes the name of *input feeding*.

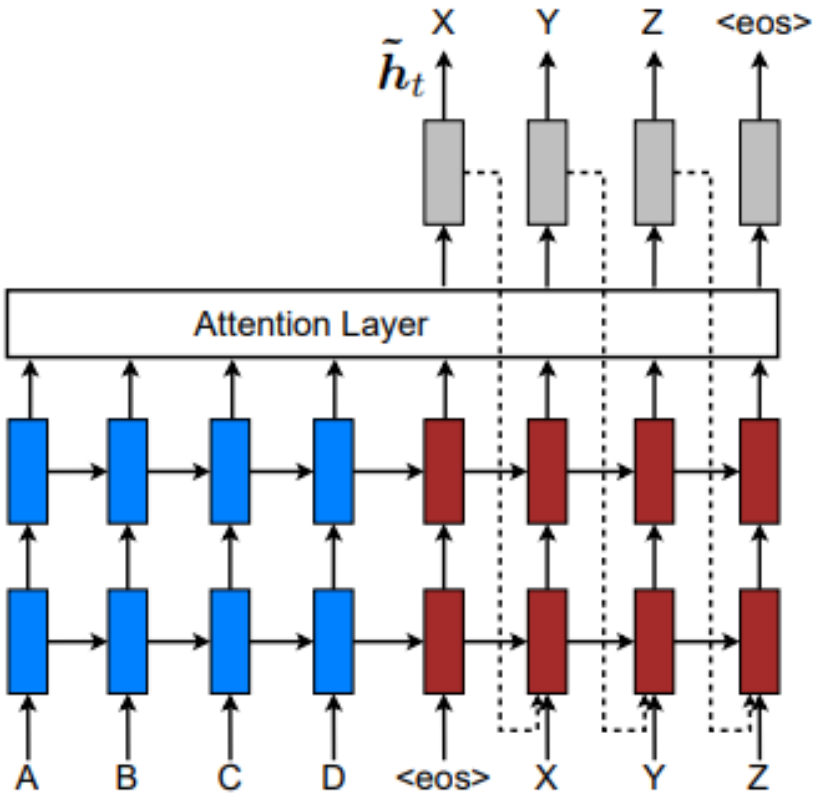


Figure 2.20: The input-feeding approach [15]

Output selection

It has been explained in the previous sections how the model generates an output probability distribution $p(y|x)$ of each word (in the source vocabulary) being the next word in the sentence. In order to get a full translated sequence of words from this probabilities, the words are sampled at each step by the layer *Out. Sel.* seen in Figure 2.17 in the decoder network. Furthermore since the complexity of generating all the possible sentences and getting the one maximizing its likelihood is NP-complete (the search problem is exponential to the length of the output sequence), heuristic search algorithms must be used. This allows the network to pick not the best solution but a

good enough approximation with a feasible complexity.
There are three possible approaches to be considered, each one having different use cases:

- **Random Sampling**: used when from a single input, multiple outputs have to be extracted (e.g. dialogue systems where to avoid monotonicity in answers multiple outputs can be used). Variable values are sampled at each time step, slowly conditioning on an increasing part of the context. In the encoder-decoder architecture this consists of calculating p_t according to the previously sampled inputs. This technique is called **ancestral sampling**.
  Finally to calculate the probability of the sampled sentence it is sufficient to multiply (or in case of log probabilities sum) the probabilities of each sampled word during the sampling process.

- **Greedy Search**: used to generate the 1-best result by calculating p_t at every time step and selecting the word with highest probability [18]. This algorithm, while fast and computationally not expensive, doesn't guarantee to find the sentence with highest probability.

- **Beam Search**: this approach is very similar to the greedy search but instead of selecting the 1-best word at each step, the **BW** best are selected, where BW is a parameter called *beam width*.
  As shown in Figure 2.21, starting from the root node, the model will calculate the output probability distribution of each word in the vocabulary, but keep only the BW-best and prune the others. At the following iteration the model will consider only the previously kept hypothesis and start the translation from there. After that, the model will output a vector of dimension $BW * vocabulary\_size$. Finally it will again prune the possible translations and keep only the BW-best (which can possibly come from the same decoder branch if they have the highest cumulative likelihood). This number of hypothesis is kept constant during the whole translation process, until an *EOS* is received or the maximum sentence length has been reached.
  An issue with this approach is that the beam search tends to prefer shorter sentences since for each word a new probability is multiplied by the cumulative likelihood, reducing it. This *length bias* has a significant impact when a large BW is chosen but can be addressed in many ways. One of the most common is dividing the final cumulative likelihood of the sentence by its length, *normalizing* it. In this way only the sentences

with the highest probability per word are chosen [18].
Finally while the *beam search* leads to better result than a *greedy search*, it increases significantly the memory usage of the network, that needs to keep in memory BW copies of the network (each one taking care of an hypothesis).

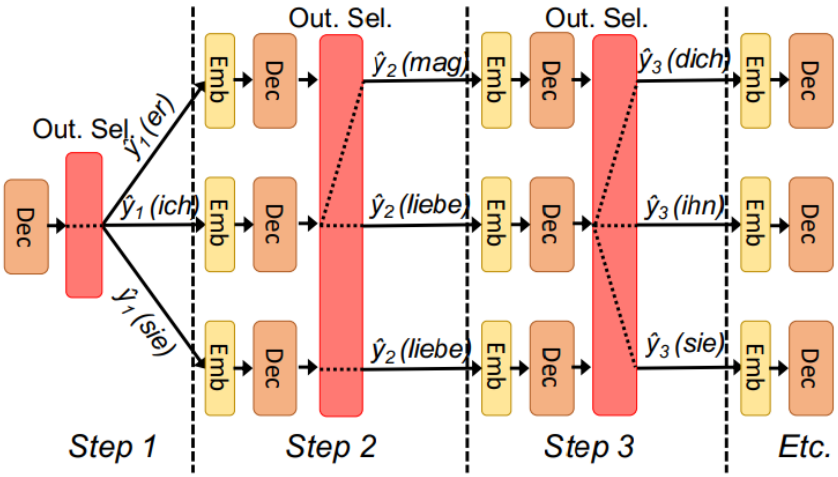


Figure 2.21: An example of the beam search algorithm with a beam width of 3. Image from [9]

2.6 Translation metrics

Evaluating the quality of the translation performed by a model is expensive and time consuming due to the need of professional translators. Furthermore their work can take long time to finish and cannot be reused in other contexts [21]. This high time cost is necessary since a human translator has to consider and evaluate many aspects of the translated text: *adequacy, fidelity* and *fluency*. This slow evaluations are infeasible in a fast changing model, which needs to be checked on a daily basis, to check that its results are reliable. To address this problem multiple automated metrics have been introduced in the last years, bringing a fast and reliable way to compute the quality of a translation or of a summary without the need of a human translator. Not only these metrics take negligible time to be computed, but they are also language independent and so usable for any model.
The following are the most commonly used metrics which have been used in this work to evaluate the quality of the translations made by the embedded

device.

2.6.1 BLEU

BLEU, which stands for **Bilingual Evaluation Understudy**, is a measure used to evaluate in a fast, automated and inexpensive way the quality of a translation [21]. This metric evaluates how close the machine translated text is to one produced by a professional human, quantifying mathematically its closeness/distance.
BLEU uses a modified version of the *precision* metric, whose formula is:

$$Precision = \frac{tp}{tp + fp} \tag{2.34}$$

In the previous equation *tp* are the true positive classifications while *fp* represents the false positive classifications. To calculate the precision in a machine translation context, it is enough to count the number of translation words (also called *unigram*) which are present in a reference translation and then divide the value by the number of words in the candidate translation. This approach doesn't perform well though, since the model can generate sentences composed of many "reasonable" words, which aren't actually good translations (but their score would be high). This problem can be solved by avoiding to count the same word in the candidate translation more times than the number of times the same word appears in the reference translation. The approach just described takes the name of *modified unigram precision* and can be used, with small modifications, even for n-gram tasks. Furthermore while the basic unit of the BLEU measurement is the sentence, using the n-grams approach with some slight changes is possible to obtain the measurement even in the common case of one source sentence but multiple target sentences.
Precision favours shorter sentences, so BLEU contains a brevity penalty **BP** which is obtained in the following way:

$$BP = \begin{cases} 1 & c > r \\ e^{(1-r/c)} & c \leq r \end{cases} \tag{2.35}$$

where c is the length of the candidate translation and r the length of the reference sentence. This penalty can be then multiplied by the original formula to obtain an updated and length independent BLEU score:

$$BLEU = BP * \exp(\frac{1}{N} \sum_{n=1}^{N} \log p_n) \tag{2.36}$$

where p_n is the n-gram precision.
BLEU output is in the interval [0,1], with a value of 0 indicating a perfect mismatch and a value of 1 a perfect and ideal translation [21]. This score can be also measured as percentage which results in obtaining values in the interval [0,100].

2.6.2 ROUGE

ROUGE, which stands for *Recall-Oriented Understudy for Gisting Evaluation*, is a metric used to evaluate the quality of translated and summarised texts [13]. As for the BLEU, this metric compares the candidate text with a golden reference, translated by a human professional translator.
There are multiple ROUGE measures:

- **ROUGE-N**: an n-gram recall between a candidate translation and its reference. It can be used with multiple reference translation
- **ROUGE-S**: metric based on the co-occurrence of a pair of any word in the sentence (also called skip-bigrams, since an arbitrary number of gaps is allowed between the two words). This metric measures the overlap of skip-bigrams between candidate and reference translation.
- **ROUGE-L**: metric based on the longest common subsequence (LCS). It uses structural sentence level similarity in order to calculate the overlap between the candidate translation and the reference one.

2.6.3 Perplexity

The Perplexity (**PPT**) measures the number of words that can follow a given word, indicating how hard is for a model to tell which is the next word in the sentence. It indicates how well the vocabulary was compressed and the ability to translate of a model, which were learned during the training phase. Given a training set $W = (w_1, \cdots, w_N)$, where N is the size, the perplexity can be calculated in the following way:

$$PP(W) = P(w_1, \cdots, w_N)^{\frac{1}{N}} = 2^{\mathcal{L}} \tag{2.37}$$

where $\mathcal{L}$ is the negative log probability of the cross entropy function, which is normally used as cost function in RNNs. This measure highest value is N , that means that all the words in the model vocabulary are equally probable. This is very unlikely to happen for languages since grammar rules constraint

the position of the words. Then good translation models should have low Perplexity after training.
This metric though depends on its training set, making it weak in evaluating the quality of a translation model in a definite way. However it can be effectively used in order to compare multiple language models.

Chapter 3

Related Works

Deep neural networks (DNNs) are currently the state of the art for many applications in very different fields. The ability of these machine learning models to reach high accuracies, sometimes superior to the human level, requires a great computational and energy effort [24]. This cost becomes infeasible for edge computing, where the computations are performed not by a high performance hardware but by an embedded device, with limited resources. Since edge computing has become more and more used in recent years, due to the need for privacy and fast response times, more energy efficient solutions (without lowering in a significant way the accuracy) for DNNs have been developed. This problem has been approached in several ways, with changes performed at the hardware level and software level.
Specific hardware accelerators have been introduced in order to enhance the computational speed of fully connected and convolutional layers, whose calculations can be easily performed in parallel since they are mainly composed of multiply-and-accumulate (MAC) operations [24]. In these accelerators, modifications are introduced not only in the processing logic, but also on the memory hierarchy, bringing the compute into memory instead of the opposite.
Techniques at the software/algorithm level that can further decrease the energy-cost of DNNs were often proposed even if in this case, a degradation in the accuracy is also present. Approaches that trade-off complexity for accuracy are instances of the *approximate computing* paradigm and can be divided mainly in two categories:

- reducing the size of operands and operations by passing from floating point operations to fixed point ones, reducing bitwidth, weight sharing...

- reducing the number of operations and/or model size thanks to technique such as pruning and compression...

Reducing the floating point (fp) precision is one of the first steps to be considered when working with embedded devices. In fact the hardware to support 32-bit operations, which are commonly used by DNN models, may be not present on embedded devices. Even if present, the hardware support is less performing and efficient than the one of GPUs or cloud servers.

3.1 CNN Optimization

The majority of the works in literature related to improving the energy vs accuracy trade off of DNNs models focus on the image classification task performed by CNNs.
Moons et al. [17] show in their work that even by reducing the fp precision, the final accuracy isn't lowered significantly. Conversely, quantization of weights can lower the energy consumption and the computational complexity.
The "weakness" of this approach is that the choice of the floating point precision can't be decided a priori but often requires a knowledge of the dataset. In fact the accuracy may be unchanged even when the 8-bit fp are used in some cases or may deteriorate immediately when using 16-bit operations. This problem can be partially addressed by choosing the type of quantization to apply, in fact it can be either *uniform*, so the same applied to all the network, or *ad hoc*, changing depending on the layer [17].
Hubara et al. [8] replace the floating point weights with binarized ones (their value can be either 0 or 1), reducing considerably the amount of memory required to store them. This change yields an increase in the power-efficiency ratio of the network, since it allows to drop arithmetic operations in favour of bitwise ones. However, for complex classification tasks, binarization significantly affects accuracy.
The other approach considered in other works is the reduction of the number of operations or the simplification of the model. **Sze et al.** [24] propose in their work the *weight sharing* approach which consists of reusing the same set of weights for all the outputs, thus lowering the memory requirements of the network. Another approach is based on the ReLU activation, which by setting all the negative values to 0, leads to sparse output activation functions and allows to skip read operations (and successive MAC) for zero-valued activations [24]. All the previous works considered only a *static* approach to

optimizing the network: the energy-accuracy trade off is decided at design time and kept constant during the usage of the network.
The *dynamic approach* instead has been recently proposed and is based on the fact that for a network not all the inputs are equally difficult to process. For example the classification of images where the subject is very small or blurry will result in a much more complex task to perform for CNNs. Due to this reason networks which were tuned at design time but stay fixed during their execution will waste energy in case of very simple inputs and perform poorly on hard inputs.
Park et al. [22] propose in their work to use two DNNs, a *big* and a *little* one, instead of a single one. At runtime then the "little" network, which is the simplest and less computationally expensive of the two, is used first in order to get a prediction. The output of the first network is then used to decide the next step to perform: in case of a simple input the simpler DNN will perform good enough and that output will become the final one. Instead if the confidence, which is the measure of the likelihood of the classification performed by the network being correct, is not sufficiently high, then the "big" network will be activated and perform the classification. This approach works due to the fact that inputs are frequently "easy" enough to achieve accurate scores with only the "little" network. Therefore, energy can be saved with a slight decrease in final accuracy. This method though requires longer time to train the models and furthermore increases the memory required since two different DNNs, each with different weights, have to be stored in the system that executes the classification. This limitation can prevent the deployment of the "Big/Little" approach on embedded systems for IoT with limited memory space available.
Tann et al. [25] enhance the previous "Big/Little" architecture by proposing only a big network whose parts are selectively activated at runtime depending on the input. This approach requires a new training algorithm which performs the learning incrementally but allows to reduce the memory required to store the network while still performing similarly.
JahierPagliari at al. [20] propose another method to change the operations' precision of the network during run-time, which depends on the input and exploits the error resilience of deep learning. This approach allows a more energy efficient network with an acceptable accuracy, furthermore it doesn't require models of increased size, retraining of existing networks or any particular hardware to support it.

3.2 RNN Optimization

RNNs literature mainly regards the creation of new architectures which yield improved accuracy by approaching the task to fulfil in a different way. However these networks often improve their result at the cost of an increased number of computations and higher complexity, which makes their execution critical to perform on embedded devices, especially since vanilla RNNs already have an high computational cost.
Much less work is present in literature about optimizing the energy-efficiency aspect of RNNs with respect to CNNs.
The aim of many of these works, as for this thesis, it is to optimize RNNs in order to be able to perform the inference phase on embedded devices, whose computational power and memory are constrained. The training phase, which is the most computationally expensive, is still performed on high performing hardware and then the model is deployed on the devices.
Chang et al. [2] propose to implement LSTM cells on specialized hardware since current CPU and GPU either offer a too limited parallelism or are limited by the sequentiality of inputs of the RNNs. This allows greater speed and increased energy efficiency with respect to a normal ARM CPU. Aside of hardware accelerators, which are often too costly to be integrated in chips for small IoT devices, other changes at the architectural level were introduced. For example **Amo et al.** [1] introduce a new GRU cell architecture to be used in embedded devices called eGRU. This new cell drops some of the gates characterizing it in order to reduce its dimension, which leads to significant increases in speed while keeping the accuracy loss low.
Many works in literature tried also to apply quantization and binarization to the network weights, as other works did with considerable results for CNNs and normal DNNs. **Ott et al.** [19] work shows high degradation of accuracies when weight binarization is introduced in RNNs, while other approaches like stochastic and deterministic ternarization yield performances close to the baseline RNNs but only for simple datasets..
He at al. [6] manage to achieve better performance than [19], it is shown that 4-bit precision RNNs can achieve accuracy similar to the one of 32-bit precision by designing LSTM and GRU cells specifically. Furthermore a method to obtain quantized weights distributed in a balanced way over the available parameter space is introduced, leading to increased accuracies in predictions since the parameter space usage is improved.
Another approach to optimize RNNs is changing at the algorithm level the network, in order to reduce the number of computations and the memory

size of the network. This is the approach followed in this thesis.
Due to the success of the encoder-decoder architecture in tasks like translating and summarizing textual data, *dynamic* approaches have been developed in order to optimize these models. **Mejia-Lavalle et al.** [16] introduce the possibility to make the beam search size (BW) dynamic by proposing two different approaches which prune the less used states, allowing a faster execution and achieving the same performance as the fixed size beam search.
Freitag et al. [4] apply the dynamic beam search approach on the encoder-decoder architecture, considering that the drawback of this heuristic is that it may continue to work with hypothesis which are far less probable than the current top ones, thus slowing the whole decoding process and wasting memory, or drop some hypothesis close to the current top ones, since the BW was exceeded. This problem can be avoided by making the BW parameter change according to the current top hypothesis or by pruning the hypothesis which are less likely than the top hypothesis by a given fixed threshold. Another approach is to avoid taking more than a given number (which was decided beforehand) of hypothesis from the same "beam", even at the cost of pruning more likely words, in order to avoid dropping hypothesis which could result in better results in the successive steps because of a single more likely one. All these beam width discrimination policies are *input* dependant and allow to achieve notable speed ups in the decoding phase, saving time and memory required. A drawback of this approach is the required knowledge of the input in order to choose the best parameters for the "policies", which adds additional complexity to the model.
JahierPagliari at al. [9] work regarding dynamic beam search proposes different policies which, based on the output of the decoder decide the beam width required on the following beam search. This value doesn't depend on the most likely word or hypothesis like for [4] but on the top-n likely words. This approach is furthermore tailored for embedded hardware and tested on single core CPU, so the maximum beam width is constrained to a lower value. This thesis is an extension of the previously mentioned work [9] and its purpose is the introduction of new beam width policies less dependant on the input parameters (so requiring no knowledge on the input text) and the optimization of the already found ones while working on a true embedded device, with limited hardware resources and optimized libraries for performing calculations. Furthermore another goal of the thesis is the measurement of the actual energy consumption to run a recurrent network in order to confirm that these dynamic approaches can actually reduce it.

All the previously mentioned works on the beam search [9] [4] [16], including this work, obtain speed-ups in the network execution (and consequently reduce its energy consumption) due to the lower number of decoder steps performed, independently from the usage of an hardware accelerator as the one proposed in [2]. Therefore, the two approaches are orthogonal and could be combined to obtain even better results.

Chapter 4

Dynamic Beam Search

4.1 Motivation

Due to the computational complexity of machine learning solutions, that are now part of an increasing amount of applications these models are currently evaluated in cloud based data centres. These centres are composed of clusters of high performing GPUs, ready to run the machine learning models necessary for the application. They wait in fact for a simple query from a host, receive the necessary information to run and send back the result to the querying application.
This approach results in a very low effort for the host that is able to use high complexity machine learning models on low power devices connected to the network. However, this method presents some defects. It is in fact very time and energy inefficient, since the required data has to be sent from the *edge* device to the cloud and vice versa. Another "weakness" of the cloud based computations is the need to send the data through the network to an unknown server, which may modify the data received and return a biased result, which the host has no way to check for any manipulation. Furthermore sending this information to the network and awaiting for the response may lead to high latencies, which for some specific tasks are not acceptable.
The security, latency and energy problems mentioned above have led to the development of an alternative approach, known as edge computing. Part of the computations, generally the prediction part concerning the RNNs, is moved to the very devices that would have previously needed to perform the query, avoiding to transmit the data on the network and to wait for a response. Due to the low computational power of these devices though, optimizing the machine learning model is necessary and this is the general goal of

the thesis. More specifically, the goal will be the optimization of an encoder-decoder RNN so that the energy efficiency of the network is improved and the chosen models, which are computationally and time expensive to use, can be run on a low ARM CPU.
The RNNs pre-trained models that were chosen for this work, that will be described in detail later, have been first profiled by using the python module *profilehooks* while being executed on the embedded device. This module allows to track the number of times a function was called and its partial execution time, reporting the results at the end of the execution of the python script.
Additionally even if the ARM CPU of the device used in this thesis can use up to four cores, using its full computational power to perform the predictions was causing the process to freeze, so it was constrained to use only one core for all the beam sizes tested. This can be useful to obtain a network optimized even for embedded devices with lower computational power than the one used in this thesis, such as those equipped with a single-core CPU.
The framework that was used in order to perform this profiling is OpenNMT, specifically in its version based on PyTorch. The framework in the chosen version offers many pre-trained models ready for prediction. Two of them, whose common task is natural language translation, but with different architectures and different input languages were chosen and deployed on the embedded device. The whole architecture of the framework and the details about the models will be described in the following chapter.
The main parts of the two networks, which are the encoding and the decoding phases have been tracked during the execution and plots of the time spent by both tasks have been generated. Each network has been executed five times on the same dataset, increasing each time the beam width from one to five. The interval $[2-5]$ represents in fact a good trade off between the final score of the network and the execution time, while a beam width of one represent a Greedy Search. The results of profiling the first network, which is less computational heavy of the two, are shown in Figure 4.1, from which it's clearly depicted that the time of execution of the encoder is almost unchanged if the beam width is increased. Moreover, it impacts in a minimal way the total execution time. Instead the decoder execution time is the most time consuming part of the inference and increases almost linearly together with the beam width. The second network is instead composed by more layers than the other and introduces an attention mechanism, which makes it more computationally expensive. However, Figure 4.2 shows a similar behaviour as before. The decoding part increases together with the beam width while

the encoding time remains unchanged and minimal with respect to the total execution time.

Behaviours and time differ when a GPU is used, since the decoder calls can

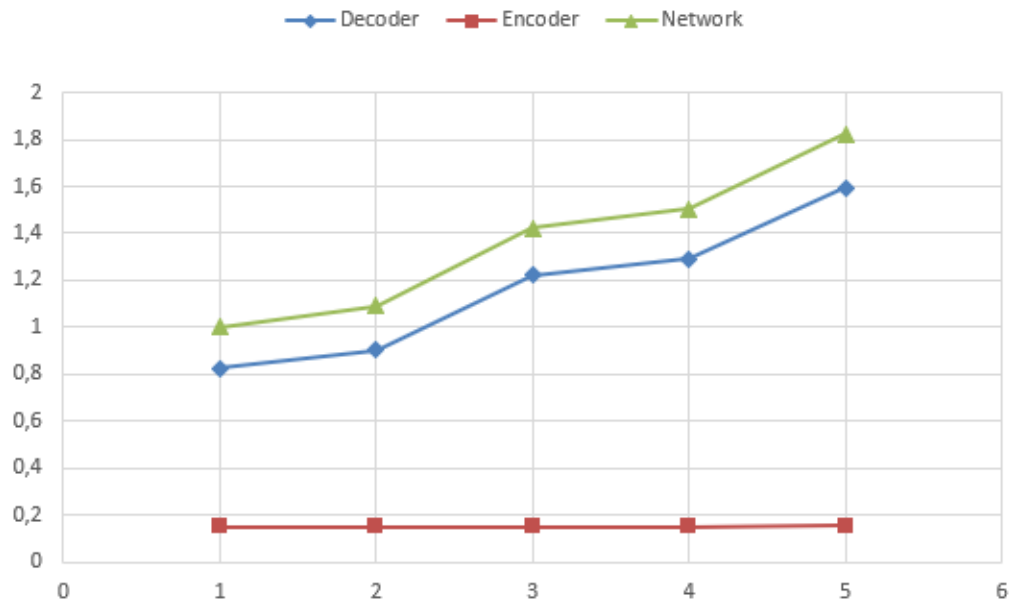


Figure 4.1

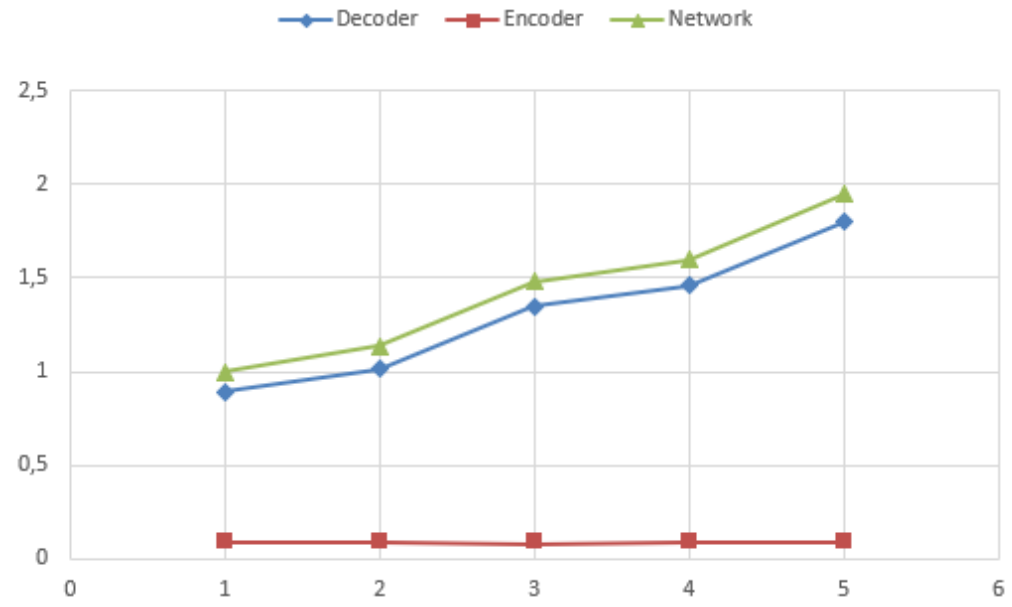


Figure 4.2

happen in parallel. In this case the increase in time due to a higher beam width is less noticeable, however GPUs are rarely present on embedded devices. On the contrary the single core CPU present on those devices has to perform all the decoding calls sequentially.
By limiting the number of cores of the CPU that can be utilised during the execution of both networks and considering that when the beam width chosen changes, the only difference is the number of decoder calls but the performed operations are the same, the CPU power consumption should not change during the translation. Then, since the power is constant during the whole execution, it is enough to lower the execution time in order to reduce the energy consumption of the model.
Due to the major role of the decoding process in the total execution time of the network and since the number of decoding steps performed depends directly on the beam width chosen, acting on the Beam Search can lead to a more energy efficient model.
Simply lowering the beam width may cause high losses of accuracy, since hypothesis which may result in more likely translations in the following steps may be dropped too soon. In fact a beam width equal to one is a Greedy Search, whose result may not be optimal. Instead, increasing the beam width too much makes the network too memory and computationally heavy for an embedded device, while not granting always noticeable increases in accuracy.
Beam search normally uses a static beam width that is chosen before starting the execution and is not influenced by the input being processed. A given input may be either very easy to translate (and thus be fine with a low value) or hard to translate (and then benefit from a higher value). The aim of this work is to change this static approach into a dynamic one, with a network that decides according to the input at each step the best value for the beam width. This choice happens not depending on the best hypothesis but on the whole probability distribution that the decoder outputs. Based on these probabilities it is in fact possible to understand how uncertain the network is about a possible translation of the current word and act accordingly. The network will in fact dynamically adapt the beam width according to its uncertainty and perform the beam search with an updated beam width. This will allow to use a small beam width for inputs which are considered easy to translate and in such a way to save energy, but keep a larger number of hypothesis for hard translations such that the final accuracy doesn't decrease.

4.2 Objective

All the policies to determine the best beam width to be picked for the following beam search step are based on the idea that, given the output of the decoding step (which is a vector storing the probabilities that the next word in the sentence is the word i in the source vocabulary), it is possible to understand how "confident" the network is. In fact a network which is unsure about the translation will output very similar probability values for multiple words in the dictionary. In this case, if the beam width is smaller than the number of possible words with very close probability, some possibly good hypothesis will be pruned and lower the final translation accuracy. Some examples of very close words where the network may be "confused" are synonyms and verbs tenses.
In the opposite case, when the network is confident, the probability distribution will be "spread", with one out of the many words with a much higher probability than the others. In this case having a beam width too high will cause the network to keep unlikely translations that will slow down the whole execution.
To estimate the degree of uncertainty of a network various measures can be used, some were introduced in [9] and have been brought to an embedded device in this work, others were introduced in this thesis. The final goal of this work is finding the best performing metrics which don't require heavy computations that would slow down the network and which allow not to decrease the final precision of the translation. Furthermore an ideal metric should perform equally good on any dataset given as input for the translation, so that it depends only on the input and not on any prior knowledge.
An example of a network implementing the dynamic beam width is shown in Figure 4.3, while translating a sentence. In the example, supposing to start with a beam width of two, it is then reduced in *step 2* to one, since the probability distribution resulted to be very spread with the word "liebe" much more likely than the others. In this case the network avoids to perform an additional and useless decoding, saving energy and time. In *step 3*, the network confidence is lower than before and then the beam width is increased to three, to avoid pruning words that may result in good translations.

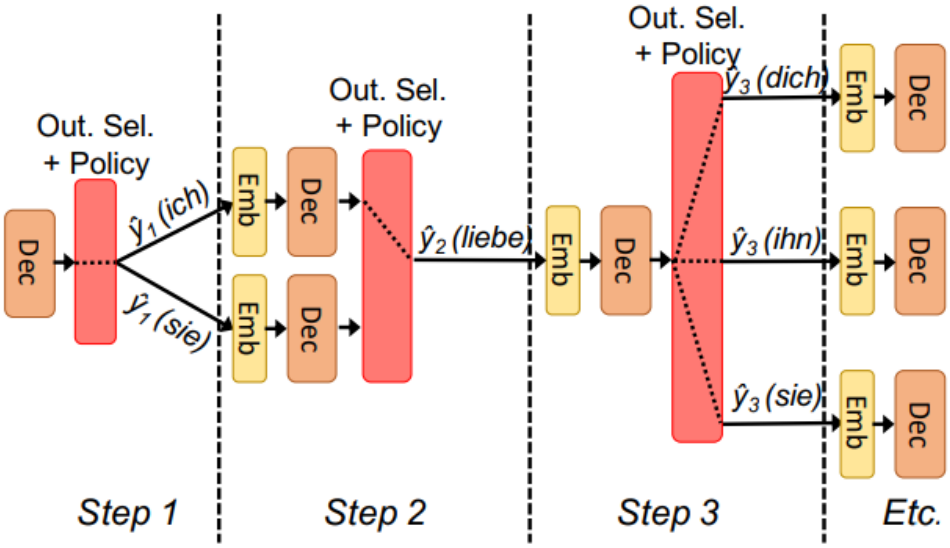


Figure 4.3: Dynamic beam search, figure taken from [9]

4.3 Inference in OpenNMT

In this section an overview of how the framework OpenNMT performs its inference phase is given, while more details about the models and the framework in general are given in the next chapter. The *inference* phase, where the translations are performed by the models, is the part that was modified in this work and which was performed on the embedded device. Given an input text, OpenNMT splits it in sentences and translates it sentence by sentence. To do this the network uses the weights it learned during the training phase, performing in this case only the forward propagation and not the backward one. OpenNMT is based on PyTorch and offers an encoder-decoder RNN architecture which can handle different networks. In fact it is possible to load the weights and the structure (layers, attention mechanism ...) of a network, so that the framework can use it directly for the inference.
The network will receive as input a source text that has to be translated, the beam width to be used in the beam search, the model whose weights and structure need to be replicated and the *batch size*. This last parameter indicates how many sentences have to be processed by the network in a single chunk. An higher batch size lowers the execution time, but comes with a decrease in the responsitivity (the model needs to receive *batch size*

sentences before starting to translate) and an increased memory and computational load. Servers equipped with GPUs can join inputs coming from various sources and exploit the advantages of a high batch_size without lowering significantly the responsitivity. On the contrary an embedded device is rarely equipped with a GPU, often instead just with a single core CPU, which can't handle parallelization. Furthermore embedded devices often handle a single source of input, thus making the process too unresponsive if *batch size* inputs need to be received before starting the inference. In practice then, due to these hardware and inputs limitations, the models on embedded devices don't use a *batch size* > 1. Additional parameters that have been used in this work are the *target file* and the *shard size*. The first one is the golden reference to calculate metrics while the second one splits the input file to be translated in smaller parts so that a file too big is not entirely loaded into memory.
After that the model has been built by the framework with the layers and all the weights learned during training time, a loop with one iteration per batch (so per sentence) starts. As first step the encoder reads the whole sentence, one word at time, until the <EOS> token is found, updating for each word parsed the hidden state which contains the compressed information on the input sentence. This memory, which at its last step is called C (*context summary*) and is stored as a vector, is then used to initialize the decoder. There are different types of decoders ready to use in OpenNMT, the one used in this work implements the global attention mechanism, input feeding and fixed width beam search.
The network, after instantiating both the decoder and the beam search class, will keep looping over the words contained in C until the decoder predicts a <EOS>. In this way the network is able to handle an input sentence with different length with respect to the output one. The decoder, at each step of the loop, is fed with the context C (weighted with the attention vector), the previously decoded word and hidden state and finally the output of the current cell. The decoder will then, as explained previously, calculate the global attention vector as weighted sum of the source memory values and use it in order to calculate the log probabilities of each word in its vocabulary being the correct translation. This vector will then be then processed by the beam search algorithm, which will extract the most likely translations (as many as the beam width). This process will be repeated, with each step aside from the first one having to handle beam width hypothesis that have to be processed by the decoder and then by the beam search.

In OpenNMT, the encoder receives as input the whole sentence and will output the context as *enc state* and the memory of all the encoder states, named *memory bank*. It will then initialize the decoder with those states, repeating them already beam width times, so that each beam will be initialized. Finally the Beam Search class is initialised with the memory bank and the beam width chosen.
After the initialisations of all the classes the main decoding loop can start, setting the number of iterations as the maximum length acceptable for a sentence. At each iteration the function *_decode_and_generate* will receive as input the previous output of the decoder, which is stored inside the Beam Search class, and output the probabilities of each word as *log_probs* and the attention vector *attn*.
The output of the decoder is then fed to the beam search class, which possesses two main functions: *advance* and *update_finished*. The first one is called at each iteration and performs the beam search on the current vector of log probabilities, whose size is the same as the source vocabulary of the network multiplied for the beam width, since each hypothesis has been decoded and has its own attention and possible words. The chosen hypothesis are then appended in a tensor with size *beam width x decoding_step* (where the decoding step is current step in the decoding loop) and the *memory bank* is updated. After the beam *advance*, the results are checked to understand if any sentence has been completely translated, in that case the function *update_finished* is called. This function will remove the beam that has finished and store it after applying a penalty for its length, then it will check if also the other hypothesis kept with the beam search have finished. In that case, it will tell to the main loop to finish otherwise it will let the translation continue.
After repeating the loop described above enough times to have all the hypothesis completed, the one with the largest likelihood is retrieved and transformed from a list of indexes of words in the source vocabulary to a full text sentence.
This inference loop is the main focus of this thesis. The loop in fact was modified in order to allow the network to accept a changing beam width as input and to determine the best beam width to use at each iteration. Determining the best beam width to set at the current time step, without slowing the decoding process in a sensitive way, is the main focus of this thesis.

4.4 Beam Width Policies

In order to pick the best beam width from the probability distribution vector of the words being the correct translation, which was calculated by the decoder, many different discrimination metrics were tried. These discriminating functions are called at each iteration of the translation, after the decoding phase and before the beam search *advance*. These functions are able to manage a batch size of one, which is the most commonly used in a low power and memory environment. It would be in fact much more complex to manage each sentence in the batch, all with their own beam size. Furthermore all the policies need to be fast to compute in order to avoid a considerable overhead for each decoder step, which would make it perform worse than the fixed beam width network.
The majority of these policies are based on parameters that can be passed from the command line and which have to be tuned according to the dataset. They can be considered as additional parameters of the network, tuned in this work on the validation dataset given with the pre-trained models.

4.4.1 Random

This policy implements a totally random approach which is not based on the decoder output at the current time step. While very fast to implement and to run, it is not "smart" and will be used together with the next policy to check if policies based on words probabilities are at least able to produce better-than-random results.
The interval in which the beam width can be picked is $[1 - 5]$, so ranging from a Greedy Search to the heaviest Beam Search that will be used in this work.

4.4.2 Alternated

Another policy which implements a random approach and whose output is independent from the decoded probabilities. The beam width value will be alternating between one (a Greedy Search) and three (from where the gain in an increased beam width is lower). This policy will be also used in order to be compared to the baseline (the network with no modifications) and to the other smarter policies.

4.4.3 Standard Deviation

This policy is based on the top-k scores of the decoded probabilities. The k value, which is the same as the maximum beam width for this policy, has been limited to five for various reasons. Using the whole search space would make the statistics computed less indicative than a reduced sample. Furthermore a beam width of five is the value suggested by OpenNMT, in fact it has been shown that increasing it won't lead to substantial increases in score. On the contrary an higher beam width would significantly increase the execution time and the computational load.
After the top-5 log probabilities have been extracted the policy calculates their standard deviation, which represents how dispersed the values are, and mean. These values are used in order to choose the best value for the beam width. Multiple versions of this policy have been implemented:

- **Threshold-based**: this variant keeps a memory of the previous beam width and uses it in order to decide the following one. If the standard deviation is greater than a threshold the beam width will be increased by one, otherwise it will be decreased by one. In this case the std. dev. is calculated as for a Normal distribution, while the threshold is a hyperparameter that has to be set empirically for a given dataset.

- **Mean ± Std. Dev.**: the beam width is initially set to the maximum value, then it is decreased by one for each value outside of the interval with boundaries $mean \pm std.dev.$ and increased by one for each score inside this interval. In this way, the sparsity of the value around the mean is calculated and used to determine the best beam width value. The boundaries used are the same proposed in [9], so 1/2, 1/3 etc. The std. dev. is calculated both as Uniform and Normal.

4.4.4 Mutual Distance

This policy uses, in order to choose the beam width, the distance between the top-5 log probabilities at the current decoding step. The distances between consecutive scores are calculated and a loop over them is started. For each distance greater than a threshold, the beam width is decreased by one. This measurements indicates how far the scores are from each other, so it measures the sparsity of the log probabilities. Different thresholds have been considered:

- **Mean**: the mean of the distances is used as threshold, this approach doesn't consider the actual value of the distance, causing to have the same results when scores are close and similarly separated or far and similarly separated.
- **Real Number**: The threshold is a fixed parameter which has to be tuned on the dataset which is used as input.
- **Distances Distribution**: variation of policy *Mean* $\pm$ *Std. Dev.*, the relative std. dev. of two scores is compared with their mutual distance.

An issue of this policy is that a high beam width is used if the top score is much more likely than the others (which happens for easier translations). In fact unlikely words will have low distances between them, which isn't greater than the chosen threshold, so the policy won't decrease the beam width.

4.4.5 Score Margin

This policy has been proposed in the work [20], where it was used as well to assess the classification confidence. Given the vector of log probabilities produced by the decoder, the top five are extracted and the beam width is set to one. Then a loop over the top values is started and at each step the distance between the probabilities is derived. If this value is smaller than a given threshold, which means that the probabilities are very close and the network is "confused", the beam width is increased by one otherwise the current value of the beam width is used. The maximum beam width allowed is five as in the other policies, while the threshold has to be tuned along the dataset.

4.4.6 Score Margin Variant

This policy is a variation of the *Score Margin* described above. The score margin is in this case computed as the difference between the best scores of the decoder. This policy will set an initial minimal beam width (to avoid using too often a Greedy Search) and will then iterate over the top-5 log probabilities of the words. As long as the beam width is lower than five (the maximum value accepted), it will be increased by one if the score margin between the currently two inspected scores is lower than a threshold, while if it is below the policy will return the current beam width. In this way when the scores are close, which means that the network is undecided on which word

to pick, the beam width is increased and no useful hypothesis are pruned. Some variation of this policy include an additional threshold that can immediately stop the loop if the first and the second score are much more likely than the others and comparing the scores in different order than from the highest to the lowest.

4.4.7 Standard Deviation Mapping

This policy has been taken from [9], where it is presented as the top performing policy on the validation dataset of the models that will be used in this work.
This policy is a variation of the Standard Deviation approach, it is in fact using the standard deviation to understand the network "confusion" for the next word to be translated. In their work, **Pagliari et al.** [9] use a line equation to formalise the relationship between the standard deviation and the future beam width. The standard deviation is the independent variable indicating the integer points inside the interval $[BW_{min}, BW_{max}]$. This equation returns a point in this space, which once rounded to the closest integer, will become the next beam width. The standard deviation used for the calculations of the point is derived from the top-5 scores and when it is very large it indicates very spread out distribution, which means that the network is very confident and the beam width returned will be small. In the opposite case, the standard deviation will be a small value, so with a "confused" network a high beam width will be returned.
Both the line equation and the two main points (shown in Figure 4.4), which handle the maximum beam width and the maximum standard deviation to be considered, that will be used in this work are the same as the ones suggested by **Pagliari et al.** [9], since have been tested already on the same dataset. The rounding techniques used, which play an important role in the final average beam width of the translation, have been taken from the previously mentioned work as well.

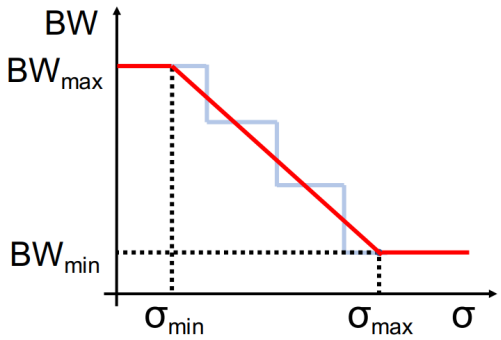


Figure 4.4: Standard Deviation Mapping, figure taken from [9]

4.4.8 Entropy

This policy is based on the Shannon's entropy calculated on all the log probabilities produced by the decoder. The formula is:

$$S = -\sum_{i}^{N} P_i \log P_i \tag{4.1}$$

This metric is also known as Information Entropy and it is a measurement of the amount of information of a source [26]. The higher this value is, the higher the "confusion" of the network is, so an higher beam will be needed. This policy will often use the maximum entropy value, which depends on the vocabulary size of the network and can be calculated as:

$$max\_entropy = \log_b (vocabulary\_size) \tag{4.2}$$

In this work b, the base of the logarithm, will be e instead of 2, which is more common in the computer science word, because the decoder directly outputs the log probabilities required for 4.1, which are already calculated with base e. This value still needs to be multiplied by the beam width of the previous step in order to be the true maximum entropy, but the values can be precomputed and stored in a data structure when the network is instantiated. Furthermore the maximum beam width has been constrained to five.
Various versions of this policy have been developed, the first set, which depend directly on the value of the entropy includes:

- **Interval based**: the interval between $[0, max\_entropy]$ is split in five parts of equal dimensions, then the current step entropy is calculated and the index of the interval in which the entropy is will be the new beam width.

- **Interval with offsets**: since values close to the maximum value of the entropy are almost impossible to see (it would require the whole set of words in the vocabulary to be equally probable), the maximum beam width is rarely used. To avoid this problem two offsets are added as additional parameters, so that the final interval will be [*start_offset*, *max_entropy−end_offset*]. These two offsets shrink the five intervals created allowing to give to the policy a greater sensitivity. In case of an entropy smaller than the *start_offset* a beam width of one is selected, while if the entropy results higher than *max_entropy-end_offset* a beam width of five is returned.

Even with the addition of the offsets, this first set of policies wasn't sensitive enough to small changes. Furthermore the time required to split in equal parts the considered interval slowed down the decoding phase excessively.
The second set of versions drops the idea of the interval in favour of a similar approach as the *Standard Deviation Mapping* policy. Specifically, a line equation is used to create a mapping between entropy and beam width (rounding the result appropriately). The entropy is divided by the *max_entropy* in order to obtain a value in the interval [0,1]. This is the list of different implementations:

- **Entropy linear**: the basic version of the policy described above, it is the fastest "smart" policy to compute. It has been tested with beam widths in the intervals 1-2, 2-4, 1-5.

- **Entropy exponential**: this policy calculates the line equation using the exponential of the current value of the entropy. While this allows the policy to be more sensitive to changes it makes the tuning harder and the calculations slower.

- **Entropy relative**: this version of the policy was developed in order to speed up the calculations of the entropy by removing all the probabilities of the decoder lower than the mean value of the probabilities at that time step. However finding and pruning the probability vector resulted to be a too time consuming task.

- **Entropy absolute**: similar to the previous approach, the probability vector was instead pruned of the candidate words with probability lower than a fixed threshold. This approach is slightly slower than the *Entropy linear*, but it didn't bring considerable improvements, so it was dropped.

In order to understand the time required by the network in order to compute each policy, the most promising ones in terms of accuracy (see below) have been profiled and the results out of 5000 calls are displayed in Figure 4.5. The *Entropy Linear* policy was the fastest to compute even if, differently from the Standard Deviation Mapping policy, the whole probability vector is used, while the *Entropy Relative*, is by far the slowest to be computed.

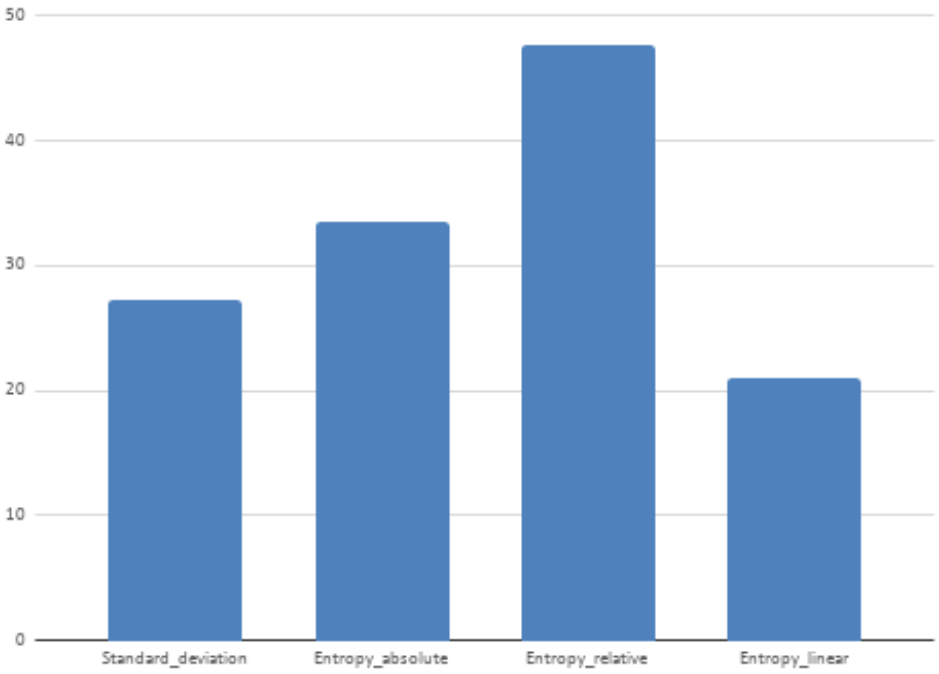


Figure 4.5: Execution times of some of the proposed policies.

4.5 Framework modifications

The framework OpenNMT supports not only translation, but many other tasks like summarization, speech to text and image to text. Only a subset of the files composing it were modified in order to allow the network to use the dynamic beam width approach at inference time. This section details the modifications done to the framework as well as some of the challenges encountered in order to make the encoder and the decoder handle different beam width sizes during the runtime. To optimize the memory usage of these new variables, in order to avoid duplicating large vectors which could fill the

main memory and cause hangs or crashes on the embedded device.
The following is the list of the files which have been modified:

- **opts.py**: the file which parses the parameters passed on the command line. Here all the thresholds and the command to choose the policy have been added.

- **translator.py**: the main file which performs the translation, here the encoding and the decoding phase are performed. The *memory bank*, which holds the states of the encoder, has been made adaptive in size, so that it can handle different beam widths. The *beam policy function* has been added as an additional parameter of the network, it is first chosen among the list of possible options when the network is created, then the function itself is passed as parameter to the function called to translate a batch of sentences. This additional step is performed after the decoding function but before advancing the beams.

- **beam_search.py**: this is the class which, given a vector of probabilities, performs the beam search on it. The main function called after every decoding step is *advance*, which now receives as new argument the new beam width and will handle not only the current results but update the previous ones as well. The tensor where both the results and the various attentions are stored is in fact updated every time the beam width changes.

The whole set of modification has been deduced by a reverse engineering process due to the lack of documentation provided by the framework, which went through several changes during the writing of the thesis. Furthermore, even if the pre-trained models used in this work are the same as in [9], both OpenNMT and the framework PyTorch used to write it, have gone through significant changes which made the previous code impossible to run.
As mentioned before, the main constraint was the lack of memory of the embedded devices, which made it critical to preserve resources at each iteration of the decoding phase. Updating the sizes of the variables which are changing along the beam width, without allocating new variables, has been a major challenge in this work and has required profiling the network numerous times.

Chapter 5

Experimental Results

5.1 Framework selection and installation

Deep learning has become an increasingly popular approach to various tasks such as image recognition, speech recognition and machine translation. This rise in popularity is not due only to the possibility of retrieving big datasets or the increasing computational power available, but also thanks to the availability of open-source frameworks for machine learning. These libraries avoid rewriting every time models, architectures and algorithms from scratch, offering optimized implementations which are on par with the fast changing machine learning world.
These frameworks not only provide an useful abstraction layer over the algorithms but allow to use a high level API, while implementing the computation on a lower level, speeding up the development of new models.
Deep learning frameworks can be divided in two main categories according to the way in which they implement the computational graph of the network, i.e. the underlying programming model that allows to implement inference and training. *Static* frameworks like TensorFlow create the computational graph of the network only once, keeping it unchanged during the rest of the execution. This approach ensures very good performances and is usually the solution adopted for production-ready models. Furthermore *TensorFlow* offers many pre-trained models ready for the inference phase, multiple programming languages APIs, and the version *TensorFlowLite* which is optimized to run models on ARM CPUs.
The other framework category is represented by the the *dynamic* ones, which allow the programmer to modify the computational graph during its execution. *PyTorch*, which belongs to this category, is a framework available in

Python and C++, which has become more and more common in the research world. It is a less mature project than *TensorFlow*, but it's growing rapidly and offers as well pre-trained models, extensive documentation and faster prototyping for new network architectures.
In this work *PyTorch* was used since the dynamic beam width approach requires modifications of the network during runtime, which would not be possible with *TensorFlow*. This choice comes with a drawback though, since the chosen framework doesn't offer any support for ARM CPUs, limiting its optimization to GPUs and Intel CPUs. Furthermore its installation is very memory heavy which made it impossible to install directly on the embedded device. The whole framework had to be recompiled directly on the device with an increased swap partition and with the flags for enabling the usage of the Neon library and the floating point 32-bit operations too.

5.2 Experimental Setup

OpenNMT [10] is an open-source framework for neural machine translation and sequence modelling created by the Harvard NLP group in 2006. The library purpose is to ease the implementations of complex architectures, by offering an easy to use tool kit which can improve the research on NMT. Originally it was built for *Torch*, but then it got implemented in *TensorFlow* and *PyTorch* as well.
Currently the *Torch* version of the framework has been dropped in favour of the other two, which are continuously improved by adding the state of the art implementation for the tasks mentioned before. Both the *PyTorch* and the *TensorFlow* implementations offer pre-trained models. As mentioned previously, this work will use OpenNMT in its *PyTorch* implementation.

5.2.1 Models

OpenNMT offers pre-trained models for three different tasks: translation, summarization and dialog. In this work the translation models were used, which are:

- **German to English (DEEN)**: this model was trained using the IWSLT14 dataset, the network implements 2 layers of 500 LSTM, word embedding of size 500, a bidirectional encoder, input-feeding and attention mechanism.

- **English to German (ENDE)**: this model has been trained using the WMT15 dataset and is composed of 6 layers of 512 LSTM. It uses the "Attention is all you Need" approach proposed by Google Brain and Byte pair encoding (BPE), which is a compression algorithm which substitutes consecutive common pairs of bytes with a new byte not occurring in the data.

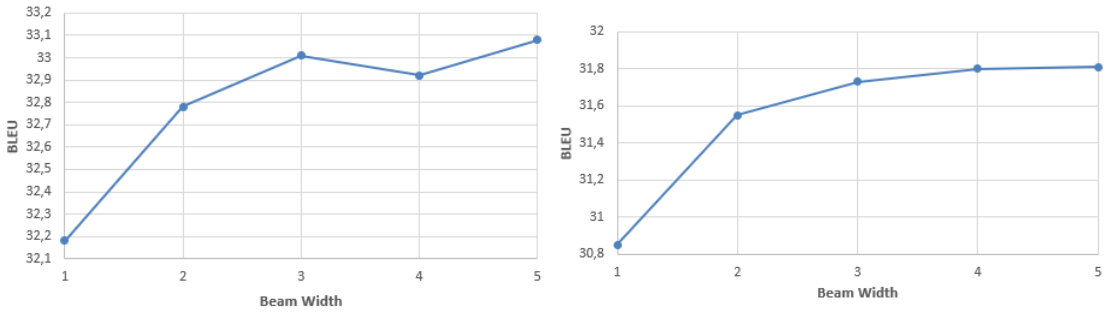
Both networks implement a static Beam Search approach in order to select the output of the decoder which has been modified into a dynamic one for this work.

5.2.2 Inference Platform

The evaluation of the policies proposed in this work has been performed on an embedded device equipped with a ARMv8 Cortex A53 CPU and 1 GB of RAM. Even if the processor is able to use four cores, *PyTorch* has been constrained to use only one during the inference. Using all the cores caused the execution to hang or crash due to the high memory consumption. Furthermore, in this way it was possible to achieve results similar to those that would be obtained with an even less powerful embedded device. Additionally the network has been set to use only one sentence at time (i.e. to use a batch size of 1), as it normally happens in edge computing. Increasing it would in fact lower the responsitivity of the system, which would need to wait to receive *batch_size* sentences before starting the translation.
In order to quickly evaluate the policies proposed during the experimental phase a desktop computer equipped with a Intel Core i5 CPU and 8 GB of RAM has been used. However this approach only works for accuracy vs Beam Width plots, and not for accuracy vs execution time plots. In fact, the time required when performing the inference on an ARM CPU, even when normalized, is different from the one on an Intel CPU (which can count on additional optimizations provided by PyTorch).
As mentioned above all the policies have been tested on the validation dataset provided by OpenNMT together with the model. PPL, BLEU and ROUGE are the metrics that have been chosen to evaluate the time (or complexity) versus score trade off.

5.3 Results

In order to understand the improvement and the effectiveness brought by the proposed policies, a baseline set of measurements of the unmodified network provided by OpenNMT has been performed on the ARM CPU. The inference of each network has been ran once for every beam width in the interval [1,5] using the whole validation dataset. For each execution it was kept track of the BLEU, the PPL and the ROUGE scores and the execution time. Figures 5.1, 5.2, 5.3 and 5.4 show the results obtained with the inferences mentioned before, both for the ENDE and DEEN network. In the Figures 5.2, 5.3 and 5.4 the normalized time which is used as horizontal axis is the total time required by the network to perform the translation of the sentences (excluding the time to allocate classes and load the required modules), divided by the time required by the network with beam width equal to one.

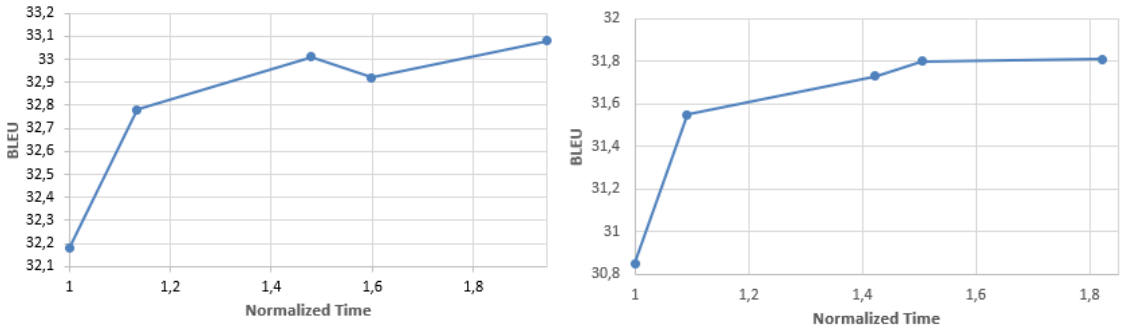


(a) ENDE

(b) DEEN

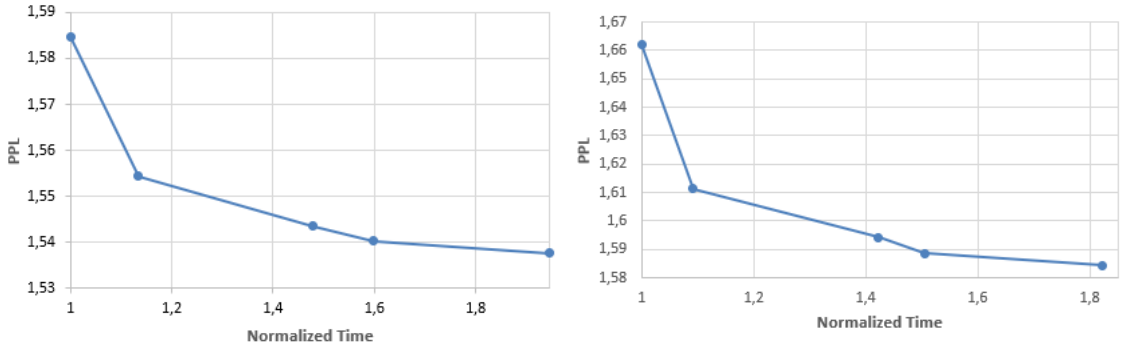
Figure 5.1: BLEU baseline

The graphs show that increasing the Beam Width increases the BLEU and the Rouge scores and decreases the perplexity, which corresponds to an increased quality of the translation. This is due to the fact that the networks, by increasing the beam width, will keep in memory more possible translations instead of just taking the best one at the current time step (which is a Greedy Search, performed when the beam width is 1). While some of the scores differ by a small amount when the beam width increases from 1 to 5, the output and then the translation quality are still significantly affected by these changes. However the increase of the beam width leads to an increasing number of computations necessary to perform the translations, which in turn brings longer inference times.



(a) ENDE (b) DEEN

Figure 5.2: BLEU vs Time baseline



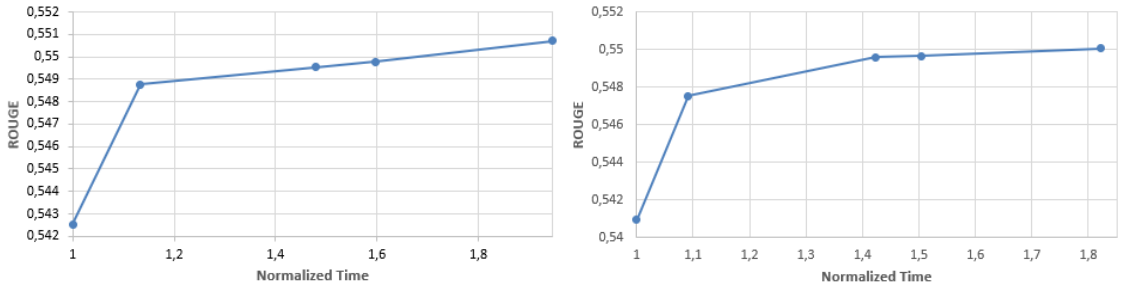
(a) ENDE (b) DEEN

Figure 5.3: Perplexity baseline

The highest increase of BLEU and ROUGE is, for both models, between the translations with Beam Width 1 and 2, while further increasing the beam width doesn't lead to such noticeable improvements. The BLEU after the initial jump doesn't increase significantly (while the inference times keep increasing). In fact, the score almost saturates with Beam Width bigger than 4.

While the most significant increase in BLEU happens between 1 and 2, the normalized execution time increases only by 13% for DEEN and by 9% for ENDE, while it increases respectively by 30.5% and 30.4% between 2 and 3. This behaviour, changes when the inferences are performed on an Intel CPU as shown by Figure 5.5, where the highest gap in execution time is between the beam width 1 and 2.

In both cases though, to gain a minimal increase in the score and so in the



(a) ENDE (b) DEEN

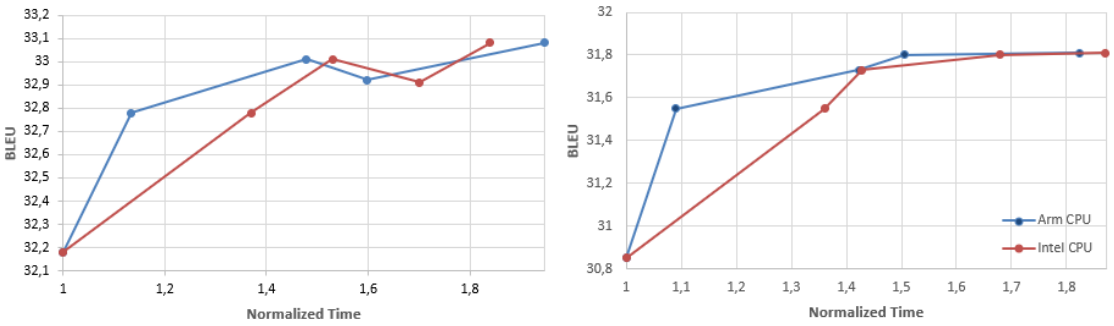
Figure 5.4: Rouge baseline

Network	Beamwidth	PPL	BLEU	ROUGE	Ex_time
DEEN	1	1.662	30,85	0.541	1
	2	1.611	31,55	0.548	1.091
	3	1.594	31,73	0.550	1.423
	4	1.589	31,8	0.550	1.505
	5	1.585	31,81	0.550	1.822
ENDE	1	1.585	32,18	0.543	1
	2	1.554	32,78	0.549	1.134
	3	1.544	33,01	0.550	1.478
	4	1.540	32,92	0.550	1.598
	5	1.538	33,08	0.551	1.946

Table 5.1: Fixed Beam width network results

translation quality, a significant increase of computations and model complexity is required, which together with the low computational power of an embedded device may slow down excessively the inference. Table 5.1 summarizes the results obtained by the various inferences with the original network in a more compact way.

Figure 5.6 shows some of the policies described previously and their BLEU score on the IWSLT14 dataset using the DEEN network, where in this case the horizontal axis represents the average beam width of the whole inference. The policies results should be as close as possible to the top left corner of the plot, which would mean a high BLEU score with a low complexity network. As expected, some policies are below the baseline score which was obtained with a fixed beam width approach, like the Alternated policy and



(a) ENDE (b) DEEN

Figure 5.5: Comparison of the inferences performed on an Intel CPU by [9] with the ones performed on an Arm CPU

the Random policy. These under performing policies are the ones that use an input independent approach while policies with a more sophisticated approach which change the network depending on the current input manage instead to outperform the baseline score while keeping the complexity of the network low. The policies which manage to surpass the baseline score more often and by a noticeable amount are the Entropy based policy and the Standard Deviation Mapping policy, while the Score Margin policy and the Standard Deviation policy weren't so consistent.

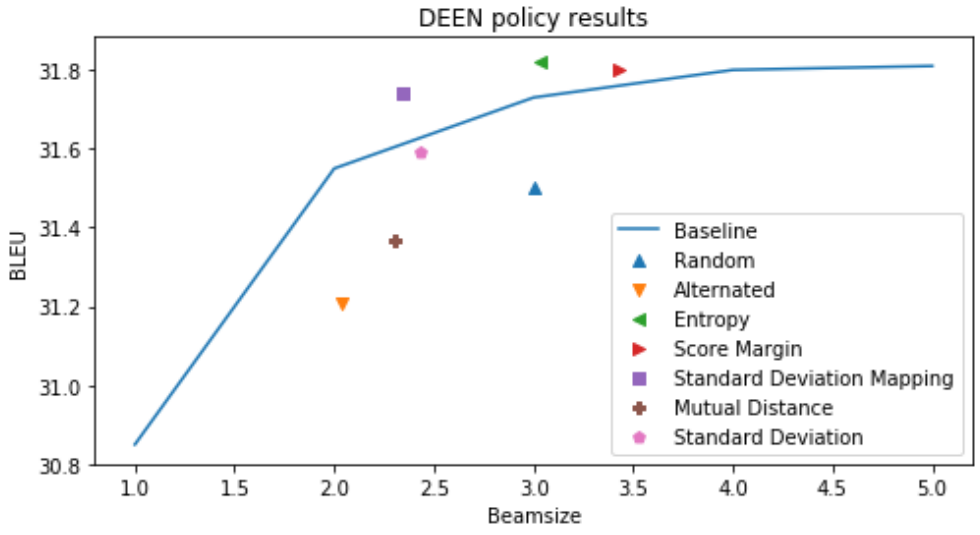


Figure 5.6: DEEN policies results

Figure 5.7 shows the BLEU scores obtained on the WMT15 dataset by the

ENDE network, the best performing policies are still the Entropy and the Standard Deviation Mapping, while the Alternated and Random approach are under performing. Due to the previous results, which allowed to understand which were the best performing policies, the Standard Deviation Mapping and the Entropy policies were explored more in detail, with a more careful fine tuning of the parameters necessary to choose the best beam width at a given time step.

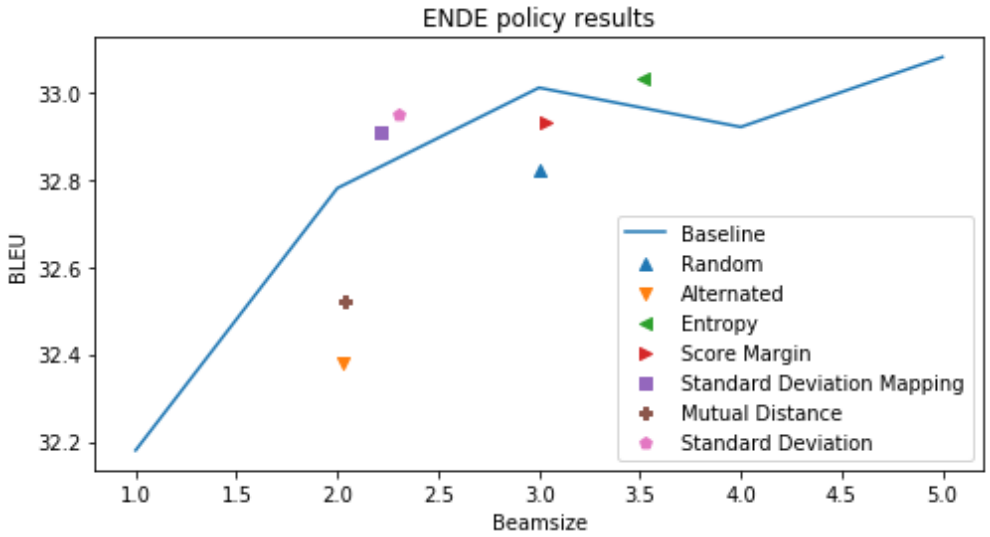
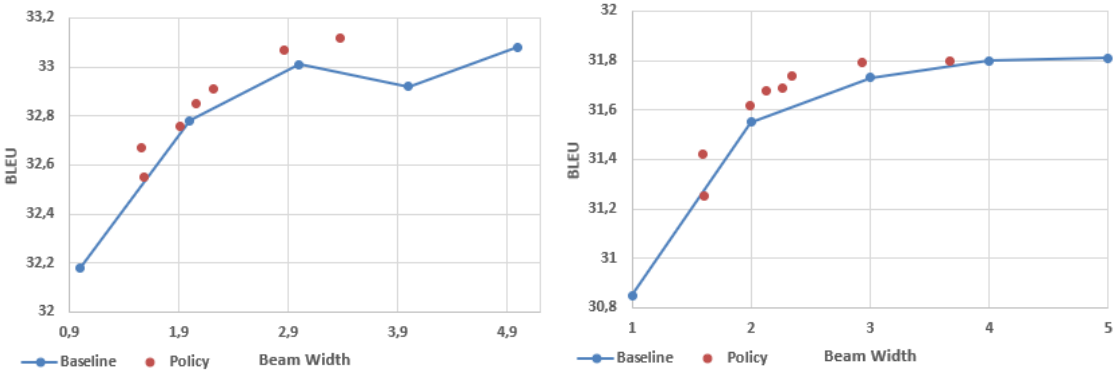


Figure 5.7: ENDE policies results

5.3.1 Standard Deviation Mapping Results

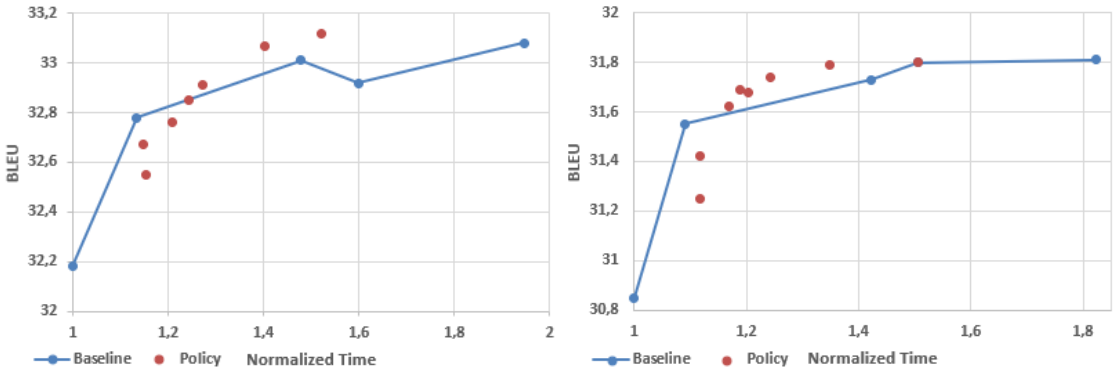
Figures 5.8, 5.9, 5.10, 5.11 show the results of the Standard Deviation Mapping policy obtained using the parameters found in [9], which while tested on the same datasets and with the same models, have been obtained from an embedded device equipped with an Intel CPU instead of an ARM CPU like the one used in this work. Notice that, for average beam width values between 1 and 2, the policy results are below the baseline (Figure 5.9). Because of the steep increase in BLEU with a short time overhead it is in fact difficult to find parameters outperforming the baseline. Regarding the Perplexity (where a better score is in the lower left corner) and the Rouge scores these policies perform better than the baseline except, as mentioned before, for average beam widths between 1 and 2. Table 5.2 summarizes the results obtained; it can be seen that with a beam width between 2 and 3,

results equal or superior to those obtained with a fixed beam width 5 can be obtained in a shorter time both for the DEEN and the ENDE networks.



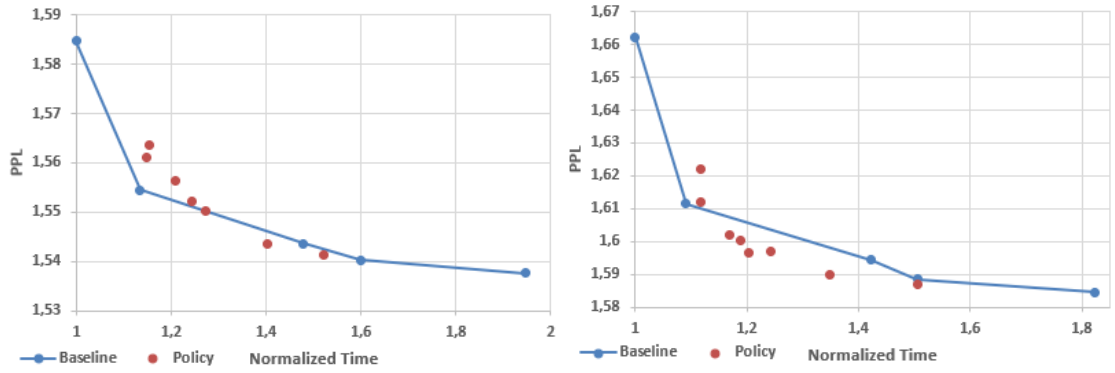
(a) ENDE (b) DEEN

Figure 5.8: BLEU with Standard Deviation Mapping



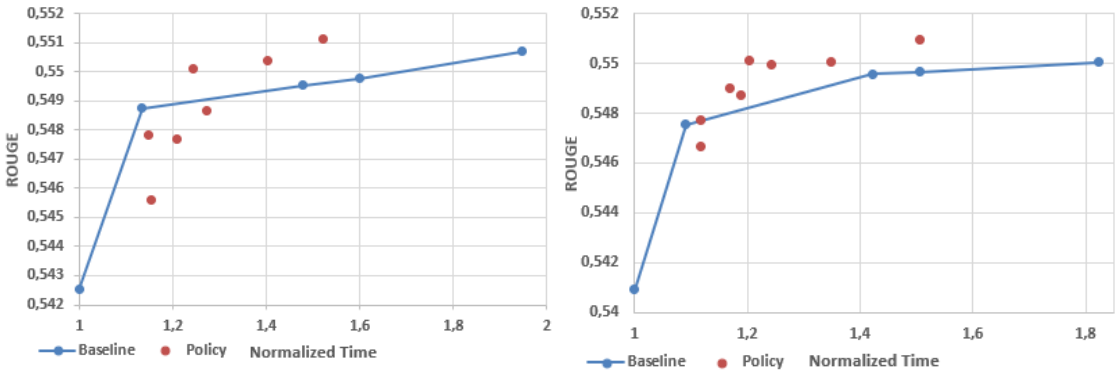
(a) ENDE (b) DEEN

Figure 5.9: BLEU vs Time with Standard Deviation Mapping



(a) ENDE (b) DEEN

Figure 5.10: Perplexity with Standard Deviation Mapping



(a) ENDE (b) DEEN

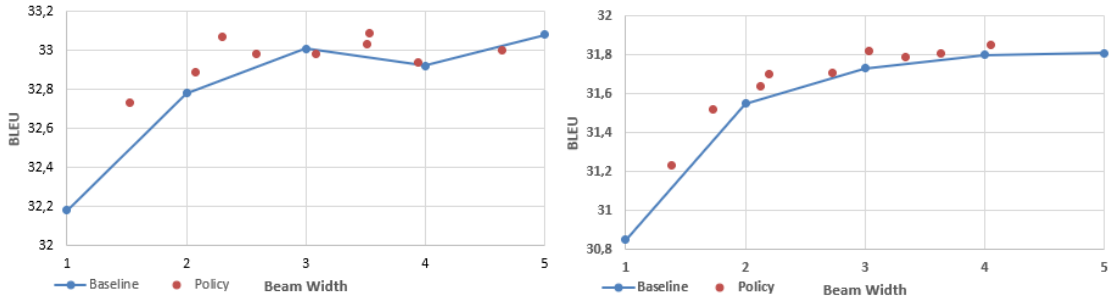
Figure 5.11: Rouge with Standard Deviation Mapping

Network	min/maxBW	min/max STD	Avg BW	PPL	BLEU	ROUGE	Ex_time
DEEN	2/4	0,1/0,6	2.268	1,6004	31,69	0,54875	1.189
	2/5	0,1/1,7	2.936	1,59	31,79	0,55007	1.348
	2/5	0,1/1,7	3.670	1,5867	31,8	0,55098	1.506
	2/4	0,1/1,7	2.342	1,5969	31,74	0,54998	1.244
	1/3	0,1/2,2	1.990	1,6018	31,62	0,549	1.169
	1/3	0,1/3,12	2.129	1,5965	31,68	0,55011	1.203
	1/2	0,1/3,12	1.594	1,6121	31,42	0,54775	1.117
	1/2	0,1/2,2	1.603	1,6222	31,25	0,54664	1.118
ENDE	2/4	1,7	2.218	1,5503	32,91	0,54865	1.272
	2/5	0,1/1,7	3.371	1,5412	33,12	0,55114	1.521
	2/5	0,1/1,3	2.861	1,5434	33,07	0,55041	1.403
	1/3	0,1/1,7	1.906	1,5563	32,76	0,54771	1.208
	1/3	0,1/3,12	2.061	1,5521	32,85	0,55011	1.243
	1/2	0,1/3,12	1.564	1,561	32,67	0,54784	1.148
	1/2	0,1/2,2	1.588	1,5637	32,55	0,54561	1.155

Table 5.2: Standard Deviation Mapping parameters and results

5.3.2 Entropy Results

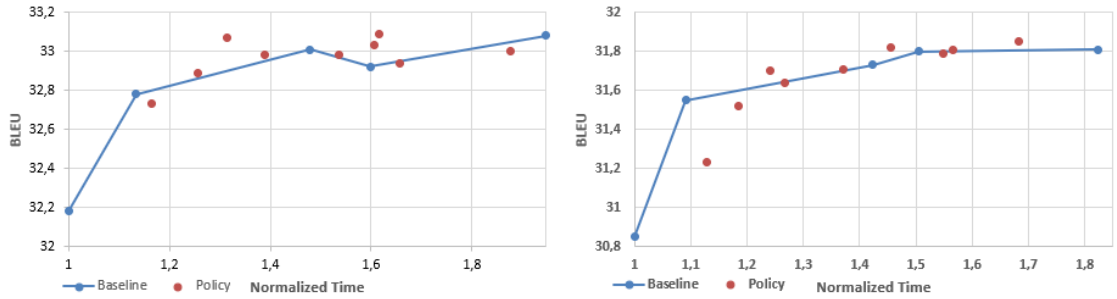
Figures 5.12, 5.13, 5.15, 5.14 show the results obtained with the policy based on Entropy (orange dots) using a linear mapping. Table 5.3 lists the parameters used in order to to obtain each point in the graph. Furthermore the maximum and the minimum Beam Width (*max/min BW* in the table) that the network could use were modified as well together with the line parameters. As for the Standard Deviation Mapping, for beam width between 1 and 2, even if the BLEU increases and surpasses the baseline value, the network becomes slower than the fixed one with beam width 2. This is due to the steep increase of BLEU score between 1 and 2, which doesn't increase significantly the total execution time.
The policies outperform the baseline scores for higher beam widths, for the ENDE network in fact a result almost equal to the one with a fixed beam width of 5 was obtained with an average beam width of 2.29, which leads to a reduction of the execution times by 33% and thus to a considerable energy saving. The same policy but with different parameters applied to the DEEN allows to obtain a BLEU score higher than the one obtained with beam width 5 with an average beam width of 3.03, so with a reduction of the execution time of 20%. This policy manages to obtain better results when the interval of possible beam width is reduced and smaller values are excluded.



(a) ENDE

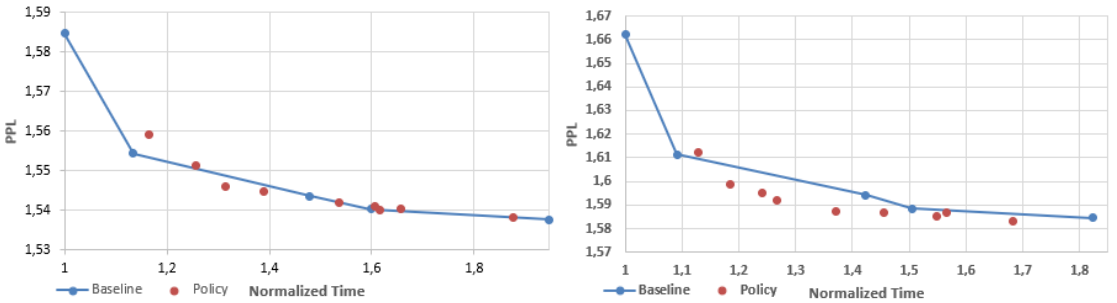
(b) DEEN

Figure 5.12: BLEU with Entropy policy



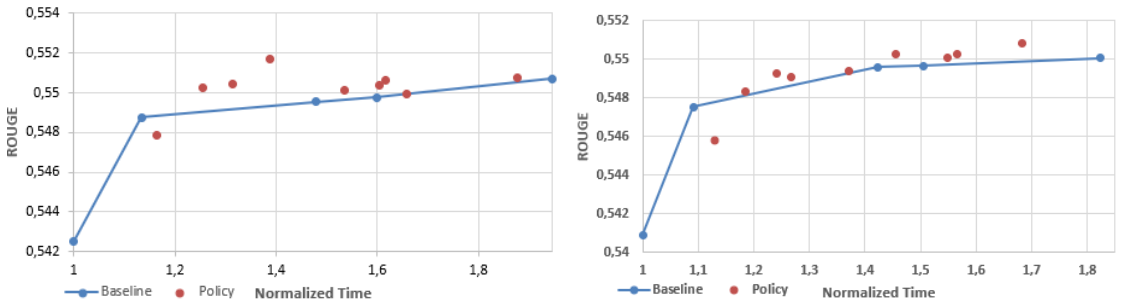
(a) ENDE (b) DEEN

Figure 5.13: BLEU vs Time with Entropy policy



(a) ENDE (b) DEEN

Figure 5.14: Perplexity with Entropy policy



(a) ENDE (b) DEEN

Figure 5.15: Rouge with Entropy policy

Network	min/maxBW	m/q	Avg BW	PPL	BLEU	ROUGE	Ex_time
	1/5	1/0,4	2.729	1,5874	31,71	0,54937	1.371
	1/5	1,5/0,2	2.125	1,5918	31,64	0,54905	1.267
	1/5	2,5/0,5	4.056	1,5829	31,85	0,55085	1.683
	1/5	1/0,2	1.730	1,5988	31,52	0,54834	1.185
DEEN	1/5	1/0,5	3.339	1,5854	31,79	0,55006	1.549
	1/5	1/0,1	1.382	1,6121	31,23	0,54582	1.129
	2/4	0,5/0,5	2.196	1,5953	31,7	0,54927	1.242
	2/4	2/0,5	3.033	1,587	31,82	0,55028	1.455
	2/4	2/0,7	3.634	1,5868	31,81	0,55023	1.566
	1/5	1/0,4	3.081	1,5419	32,98	0,55014	1.536
	1/5	1,5/0,2	2.580	1,5447	32,98	0,55171	1.389
	1/5	2,5/0,5	4.644	1,5383	33	0,55076	1.877
	1/5	1/0,2	2.072	1,5514	32,89	0,55025	1.256
ENDE	1/5	1/0,5	3.528	1,5401	33,09	0,55063	1.616
	1/5	1/0,1	1.523	1,559	32,73	0,54784	1.165
	2/4	0,5/0,5	2.299	1,5459	33,07	0,55047	1.314
	2/4	2/0,5	3.515	1,5411	33,03	0,55041	1.605
	2/4	2/0,7	3.944	1,5405	32,94	0,54996	1.658

Table 5.3: Entropy Policy parameters and results

5.4 Energy Measurements

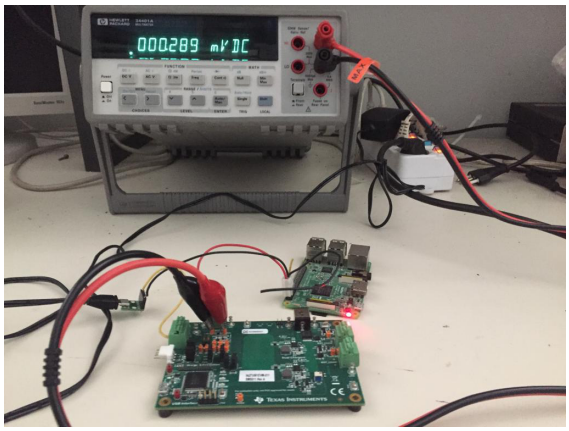


Figure 5.16: The devices used to take the measurements

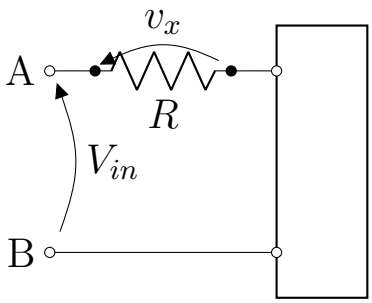


Figure 5.17: High level view of the circuit used for the measurements

In order to prove that the energy consumption of the embedded device during the inference is constant since the same operations are repeated during the whole execution time, measurements with a multimeter were performed. In case of a constant energy consumption, reducing the execution time (as done in this work) would yield comparable energy savings. The energy consumption of both the network was measured while translating 400 randomly selected sentences of the validation datasets due to the excessively long execution time of their full version. For the same reasons, the measurements were performed only for the networks with a fixed beam width of 1,3 and 5, plus the entropy policy.
The measurement, which can be seen in Figure 5.16 , can be summarized by the circuit shown in Figure 5.17. It is composed by the embedded board mounting the ARM CPU, a Digital Multimeter (HP/Agilent-34401A) and

a Battery Manager Evaluation Module (Texas Instruments BQ27Z561EVM-011). The voltage drop V_x was read by the multimeter every second and saved on a file on a computer. Then, after the measurements were completed, the current flowing through the circuit was derived:

$$\mathbf{I} = \frac{\mathbf{V}}{R} = \frac{V_i}{10^{(-3)}} \tag{5.1}$$

Once the current was obtained, since the input voltage of the board (the rectangle in Figure 5.17) is known, the power can be obtained:

$$\mathbf{P} = \mathbf{I} * V_{input} = \mathbf{I} * 5 \tag{5.2}$$

Finally the power was integrated over time in order to obtain the energy:

$$\mathbf{E} = \sum_i P_i \tag{5.3}$$

Figures 5.18 and 5.19 , show the voltage obtained during the inferences respectively for the DEEN and the ENDE network. It can be observed that the voltage remains almost constant during the whole execution time and that it doesn't change depending on the beam width of the network. Table 5.4 shows the energies obtained from the graphs. This proves that by reducing the execution time of the network thanks to the proposed policies there is an actual energy saving proportional to the time saved.

Table 5.4: Entropy Policy results

Network	**Avg BW**	**Energy ENDE (J)**	**Energy DEEN (J)**
Fixed BW	1	14 605.224	4260.878
Fixed BW	3	21 298.098	5561.152
Fixed BW	5	27 867.806	7383.225
Dynamic BW	1,27	16 579.907	4489.028

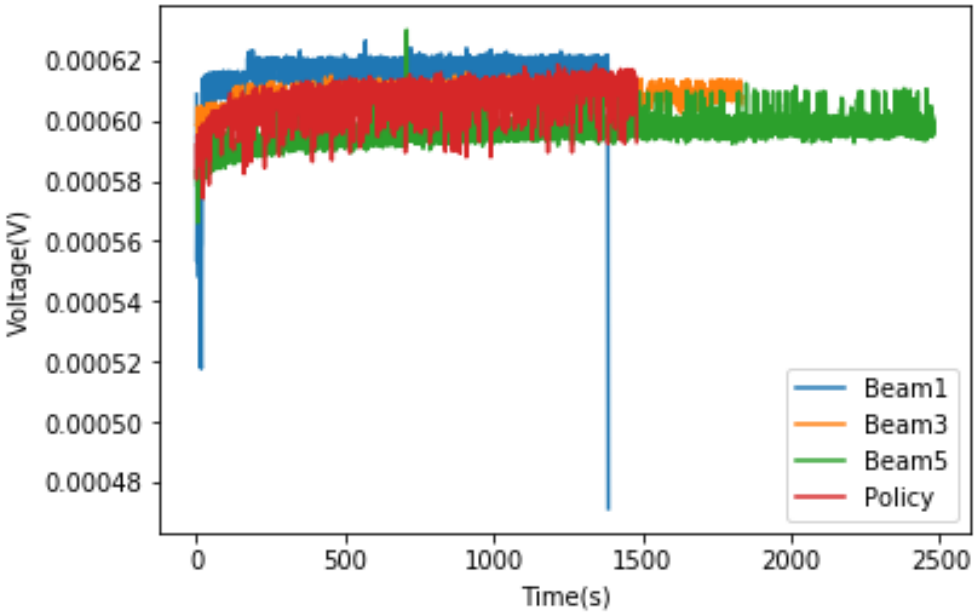


Figure 5.18: DEEN inferences comparison

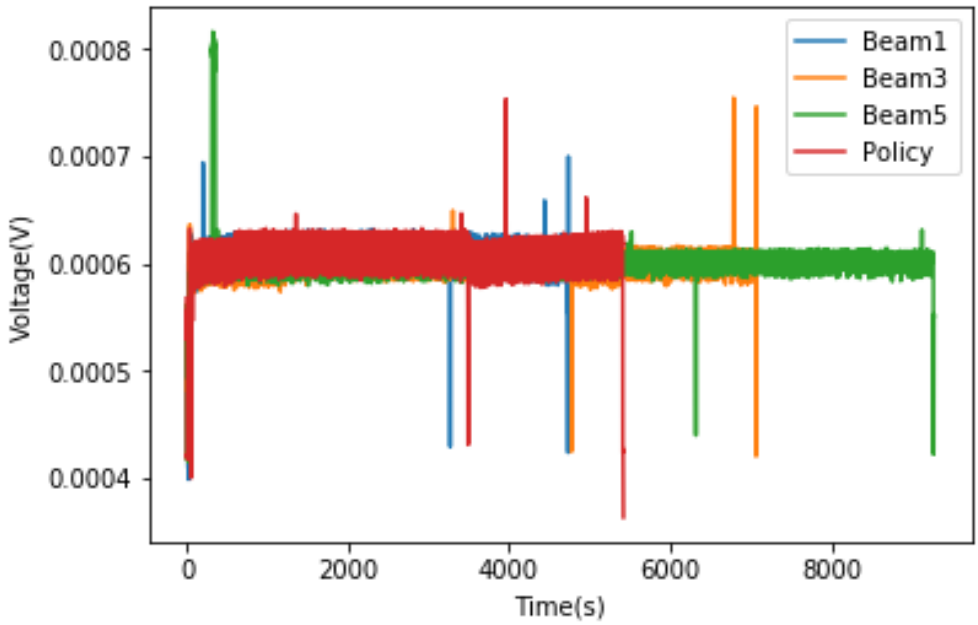


Figure 5.19: ENDE inferences comparison

Chapter 6

Conclusions and Future Works

The high amount of resources needed to execute deep learning models and in particular recurrent neural networks, has made them too complex to be run on embedded devices with low memory and computational power.
This work proposes an optimization of such models, in particular the Encoder-Decoder architecture, that is modified at runtime depending on the complexity of the input that is being processed. This reduces the number of computations required and so the energy consumption of the network. It has been proved that applying the tuning to the network on an embedded device reduces effectively the time required to perform the inference. Furthermore various discrimination criteria to decide which is the best configuration for the network at the current time step, that base their decision on the input, were tested. The two most promising criteria were explored more in detail and by applying them on the inference phase it was possible to reduce the execution time of more than 20%, while yielding higher accuracies than the unmodified networks. Furthermore the whole process power consumption was measured using a multimeter.
The network can be further optimized by applying other approximated computing techniques like the quantization of the weights or the simplification of LSTM cells. In future this approach and the proposed discrimination criteria can be implemented on GPUs, in order to exploit its parallelism and translate multiple sentences, each one with a different dynamic beam width. Furthermore the relationship between the proposed policy using the Shannon's Entropy and the Perplexity used by the network as loss function can be explored more in detail, in order to choose better parameters policies or

even change them at runtime.

Bibliography

[1] J. Amoh and K. Odame. An optimized recurrent unit for ultra-low-power keyword spotting. *arXiv preprint arXiv:1902.05026*, 2019.

[2] A. X. M. Chang, B. Martini, and E. Culurciello. Recurrent neural networks hardware implementation on fpga. *arXiv preprint arXiv:1511.05552*, 2015.

[3] D.-A. Clevert, T. Unterthiner, and S. Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

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

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

[6] Q. He, H. Wen, S. Zhou, Y. Wu, C. Yao, X. Zhou, and Y. Zou. Effective quantization methods for recurrent neural networks. *arXiv preprint arXiv:1611.10176*, 2016.

[7] S. Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

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

[9] D. Jahier Pagliari, F. Panini, E. Macii, and M. Poncino. Dynamic beam width tuning for energy-efficient recurrent neural networks. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, pages 69–74. ACM, 2019.

[10] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. Rush. OpenNMT: Open-source toolkit for neural machine translation. In *Proceedings of ACL 2017, System Demonstrations*, pages 67–72, Vancouver, Canada, July 2017. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/P17-4012.

[11] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86 (11):2278–2324, 1998.

[12] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553): 436, 2006.

[13] C.-Y. Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81, 2004.

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

[16] 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*, pages 59–64. IEEE, 2013.

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

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

[19] 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.

[20] 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*, page 47. ACM, 2018.

[21] 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*, pages 311–318. Association for Computational Linguistics, 2002.

[22] 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*, pages 124–132. IEEE Press, 2015.

[23] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry. How does batch normalization help optimization? In *Advances in Neural Information Processing Systems*, pages 2483–2493, 2018.

[24] 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*, 105(12):2295–2329, 2017.

[25] 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*, page 34. ACM, 2016.

[26] Y. Wu, Y. Zhou, G. Saveriades, S. Agaian, J. P. Noonan, and P. Natarajan. Local shannon entropy measure with statistical tests for image randomness. *Information Sciences*, 222:323–342, 2013.

[27] B. Xu, R. Huang, and M. Li. Revise saturated activation functions. *CoRR*, abs/1602.05980, 2016. URL http://arxiv.org/abs/1602.05980.