POLITECNICO DI TORINO

**Master's Degree in
in Electronics Engineering**

Master's Degree Thesis

Spiking Neural Networks for Speech Recognition: integration of spiking neurons in a sequence-to-sequence architecture

Supervisors
prof. DI CARLO STEFANO
prof. SAVINO ALESSANDRO
prof. MAROSTICA FILIPPO
prof. CAVIGLIA ALESSIO

Candidate
Vittorio Frangipani

Academic year 2024 - 2025

Summary

Speech recognition is the process of transcribing spoken language into written text. In recent years, there has been increasing interest in bringing speech recognition to edge devices, where model size and energy efficiency become crucial. The aim of this thesis is to investigate the integration of spiking networks within a speech recognition context. Spiking neurons are inspired by the functioning of the human brain and have proven capable of reducing the energy consumption of neural networks with little or no loss in performance. Spiking neurons has a so called membrane potential that is a variable implementing a memory mechanism. Based on the inputs, the potential can increase or decrease. Many types of neurons also implement a decay mechanism that at every iteration discharges the membrane by a factor called the decay factor. When the potential reaches a predefined threshold, the neuron emits a spike with unit value; otherwise, it outputs 0. The most common type of spike neuron is the Leaky-Integrate-and-Fire (LIF), the behavior of which is inspired by the RC circuit.

The spikes neurons behavior is intrinsically sequential, which is beneficial for modeling phenomena with temporal correlations, such as speech, but also introduces challenges during training, which cannot be fully parallelized. Furthermore, the presence of spikes complicates training because backpropagation cannot be directly applied due to the non-differentiability of the spike function. Spiking neurons require careful study of the encoding and decoding strategies used to convert information between continuous and binary domains. Many studies compare Artificial Neural Network (ANN) and Spiking Neural Network (SNN) approaches to assess their impact, and more recent works propose fully spiking architectures, demonstrating the growing interest in the field.

Our work integrates different types of spiking layers into a sequence-to-sequence architecture (seq2seq) for speech recognition, in order to evaluate their impact. Seq2seq architectures consist of two main components, plus a searching algorithm used for decoding: an encoder that extracts contextual information and produces a context vector, and a decoder that, based on previous outputs and the context generated by the encoder, produces a prediction that is then interpreted by the searcher.

The chosen architecture includes an encoder with two convolutional blocks followed by four LSTM layers (Long Short-Term Memory, a particular type of recurrent layer capable of maintaining temporal dependencies over long sequences) and two fully connected blocks. The decoder is an autoregressive decoder with an attention mechanism and a Recurrent Neural Network (RNN) layer. The network is trained using a combination of two loss functions: cross entropy as the primary loss, and the Connectioni Temporal Classification (CTC), used only during the first epochs to accelerate training. Following previous works, we focused on the encoder, particularly on its interface with the data and on the ANN–SNN transition.

Our first experiment replaced the LSTM layers with Spiking LSTM (SLSTM) layers, which follow a similar mechanism but integrate a threshold-based spike-generation process. To speed up training and reduce model size, we inserted a linear layer between the SLSTM layers and the convolutional block, and reduced the number of layers from four to two. For data encoding, we adopted firing-rate coding. The idea behind this encoding is that a higher value of inputs corresponds to higher firing activity. Following this idea, we can transform inputs in train of spike, passing them to the neurons, and read the elaboration by counting the number of spikes produced by the layer within a temporal window. We choose as temporal window 10 steps, meaning we evaluate the firing rate every 10 spikes. The outputs of the ANN convolutional layers were converted into spikes through duplication coding, which repeats each value for the number of iterations in the decoding window.

Training was performed using surrogate gradients, which replace the non differentiable spike activation function during backpropagation with a differentiable approximation. After we experimented with spiking convolution in order to further unify the data types processed by the network. Spiking convolution is a broad area; our approach was to alternate standard 2D convolution layers with LIF layers. To keep the network computationally efficient, parallelism was preserved within the convolutional operation; afterwards, the samples were grouped across iterations and processed sequentially by the LIF layers. As pooling strategy, we replaced the max pooling of the reference ANN network with pooling based on the maximum firing rate of the LIF neurons. The encoding technique used allows the first LIF layer to integrate continuous outputs from the convolutional layer, eliminating the need for additional encoding components and improving efficiency.

After we explored training strategies, focusing mainly on the interaction of hybrid networks with the CTC loss and experimenting with different numbers of SLSTM layers.

As performance metrics, we used WER and CER (Word Error Rate and Character Error Rate), which measure network accuracy, and the percentage of insertions (insertion of non-existent tokens), deletions (removal of existing tokens), and substitutions (replacement of true tokens with incorrect ones), which classify error types. We also conducted an analysis of the firing rates of the different layers and the model sizes measured in terms of number of parameters. To account for the lower convergence velocity, we increase the number of training epochs from 15 used for the baseline model to 30 for spiking architectures.

The results show that the best-performing model is SCNN with 2 SLSTM layers (the SCNN2 model), achieving a WER of 18.3%. Although the WER is higher of 8% than the one of the baseline model (WER 10,6%), the SCNN2 is for time smaller than the original model. SCNN2 was also trained for 40 epochs to evaluate its improvement margin. Another interesting observation is that adding two more SLSTM layers, reaching four (SCNN4) as in the original architecture, yields nearly identical results, slightly worse (WER 19.66%), which does not justify using a heavier and slower architecture in our case.

Regarding CTC, we observe that increasing the number of CTC epochs leads to degraded performance, although completely removing CTC resulted in ineffective training.

The SCNN2 model exhibits a generally lower firing rate, particularly in the convolutional part, which is the most energy-intensive, leading to improved energy efficiency. This further demonstrates the superiority of the architecture also from an energy-efficiency standpoint.

List of Tables

List of Figures

Contents

Chapter 1

Speech recognition

Speech recognition is the task of transcribing spoken language from audio recordings. Over the years, the methods used to perform this task have evolved significantly. Early approaches relied on multiple interconnected components and complex structures, such as Hidden Markov Model (HMM) and Gaussian Mixture Model (GMM). Later, these systems became progressively simpler and more efficient: GMM were replaced by deep neural networks, and the focus shifted from recognizing *senones* (subunits of phonemes) to recognizing characters or subwords, thus simplifying the overall process.

In this chapter, we first present the classical Automatic Speech Recognition (ASR) architecture to provide the reader with an overview of the evolution of the field. Some traditional components, such as language models or Mel-Frequency Cepstral Coefficients (MFCC) features, are still in use today, while others have been replaced. We then introduce modern methods based on end-to-end neural networks, outlining the current state of the art and describing the models adopted in our project. Finally, we discuss audio pre-processing, sequence-to-sequence models, and provide theoretical insights into training and decoding procedures.

1.1 ASR Classic — Basic ideas of traditional ASR

The basic goal of ASR is to find the most likely word sequence $\hat{W}$ given an observed acoustic signal X:

$$\hat{W} = \arg\max_{W} P(W \mid X).$$

Here, $\hat{W}$ represents the transcribed sequence of words, while X is its corresponding acoustic representation. Applying Bayes' theorem, we can rewrite this expression as:

$$\hat{W} = \arg\max_{W} P(X \mid W) \cdot P(W),$$

where the term $P(X)$ is ignored since it is constant for all hypotheses.

The first term, $P(X \mid W)$, is called the *Acoustic Model*, while the second term, $P(W)$, is the *Language Model*. The acoustic model represents how likely the observed acoustic sequence corresponds to a given word sequence, whereas the language model defines how probable the word sequence is in the target language. For instance, the sentence "The cat is walking on the floor" is much more likely than "The floor is walking on the cat."

The standard way of obtaining the representation X is to divide the audio signal into small overlapping frames and compute for each frame the *Mel-Frequency Cepstral Coefficients* (MFCCs). We then need a model capable of mapping this temporal sequence of features to a sequence of words. The classical ASR system introduces several key components:

- **HMM (Hidden Markov Model)**: A Hidden Markov Model is a probabilistic structure that models relationships among a sequence of hidden states. These states are not directly observable but are linked to observable events through an *emission probability*. Transitions between states are governed by *transition probabilities*, which describe how likely a state evolves into another. In ASR, the hidden states correspond to acoustic units from the pronunciation dictionary, called *senones*, while the observable events correspond to MFCC vectors. Senones are sub-units of phonemes, typically represented by two or three states depending on the implementation.

- **Lexicon**: The lexicon is a mapping table that decomposes each word in the vocabulary into its constituent phonemes and then into senones. It serves as the link between the sequence of recognized acoustic states and the final words.

- **GMM (Gaussian Mixture Model)**: Gaussian Mixture Models are statistical models used to represent the emission probabilities of the HMM. Each senone is modeled as a mixture of Gaussian distributions in the MFCC feature space. These models are learned during training and provide the likelihood of observing a particular acoustic vector given a specific state.

1.1.1 Decoding

To obtain the final transcription, the system must find the most probable sequence of hidden states that maximizes the likelihood of the observed acoustic sequence X. This decoding problem is efficiently solved using the *Viterbi algorithm*. At each time step t, the algorithm evaluates the most likely state given the observation and the transition probabilities from all possible previous states at time $t-1$. For each state, only the most probable path is retained through the use of *backpointers*, which avoid recomputing all paths from the beginning. Once the end of the sequence is reached, the most likely path is selected, and the complete state sequence is retrieved by following the stored backpointers.

In the literature, several attempts have been made to replace GMM with Deep Neural Network (DNN). However, most recent approaches have shifted toward fully *end-to-end* architectures. [1]

1.2 End-to-End Methods

The new frontier of speech recognition is the end to end method. It consists of using only a DNN for the recognition task, eliminating GMM, HMM and lexicon. For this reason also the units to recognize are different: senons are replaced by tokens. Tokens are subwords in which the dataset is divided. This process is called tokenizzation and it is the only sub task not covered by the DNN (excluding the pre elaboration of the audio signals). Furthermore, we don't have in this structures a match between audio frame and tokens, so we don't need to align text and audio. It is a great advantage to build datasets. The networks' input are still frequency representation of the audio. Models built with this approach has outperformed the state of the art of hybrid and classical speech recognition at least on the most used datasets[9], so in our project we used this approach

1.3 Tokenization

Before analyzing how networks are composed and how data are elaborated, we have to talk about tokenizzation, that is the segmentation of the dataset in subwords called *tokens*. The software used is called tokenizers. In our project, we used SentencePiece. In addition to subwords, tokens also include some special symbols: *blank, bos, eos, ukn* called simply *special tokens*. We will discuss the blank character in the section dedicated to Connectionist Temporal Classification (CTC) loss function, for now let's focus on the other 3:

- *Bos* stays for *Beginning Of Sentence*, it is the token sent to the decoder when it has to generate the first character. This aspect will be discussed better in the Sequence-to-Sequence (seq2seq) section.

- *Eos* stays for *End Of Sentence*, when the token is generated, the decoder stops. This aspect will be discussed better in the seq2seq section.

- *Ukn* instead stays for *Unknown.* Some networks are trained also to handle some unknown word and reserves a special character for this. It is possible training its recognition using a *coverage factor* of the tokenizer less than one. The coverage factor sets the percentage of the dataset's words that will be covered by the segmentation.

Algorithm

In our project you can choose between 3 kinds of segmentation algorithm:

- *BPE (binary pair encoding)* starts decomposing the data set in characters. These are the first possible tokens. Then it chooses the most frequent adjacent tokens' pair, merges it, and repeats this process with the new corpus until the number of tokens matches the required one.
- The *Unigram* algorithm selects the most likely segmentation. For a given vocabulary, it evaluates all valid segmentations and maximizes the product of the token probabilities under an independence assumption. The segmentation with the highest product is chosen. The algorithm iteratively prunes tokens by estimating the impact of removing each token on the corpus, removes those with minimal impact, and re-estimates token probabilities until the vocabulary reaches the target size. This is the most reliable system and for this reason is what is used by default in our project.
- The *Char* option uses characters as tokens, so this option does not use any algorithm.

1.3.1 Token encoding

Tokens are written units, so to allow the network to learn, we should encode them in a numerical way.

One-hot Encoding

Each token is represented as a vector of all zeros except for one position containing one. So for a vocabulary of m words and using vectors of length N,if we assign to each one an integer number $y_i \in \{1,\dots,N\}, \quad i = 1,\dots,m$ we can represent our dataset as a matrix $Y \in \{0,1\}^{m\times N}$ such that:

$$Y_{i,j} = \delta_{j,y_i}$$

This method has the problem that requires to have a vector dimension at least of the vocabulary size.

Embedding

To reduce the size of encode and improve the performance, we can relay on embedding. It works via look at table. At the beginning, every token is mapped into an integer. This integer is used to link tokens to the table's vectors containing weights that will be learned via training. In this way, we not only reduce the size of the encoding, but also we can catch correlations and similarities among tokens.

1.4 ASR pre-elaboration

1.4.1 Audio Elaboration

Audio samples cannot be sent directly to the network. First, there is a phase of normalization in which all the data are resampled at the same sample rate. Then they are segmented into smaller overlapping pieces and translated in the frequency domain. The segmentation is called *windowing.*

Mel features

The translation in frequency is not straightforward. We chose to use *Mel's coefficient.* Our hearing is not linear, we are better to perceive the differences among lower frequencies: for example, is easier for us to catch the difference between 400 Hz and 500 Hz than the difference between 10 000 Hz and 10 100 Hz. The mel scale rescales the frequency domain in a more linear way for our hearing. The formula is:

$$\text{mel}(f) = 2595 \log_{10}\left(1 + \frac{f}{700}\right)$$

This transformation is used and supported by literature because it gives the network a representation similar to what our brain receives.

Mel coefficients are obtained using triangular filters centered on frequencies bands equally long in the mel domain. These bands overlap each other. To get N filters, $N + 2$ points are needed, including frequency range bounds. If we enumerate each point from 0 to $N+1$ then every band goes from the point $p-1$ to $p+1$ for $p \epsilon [1, N]$ To get the mel coefficients, first of all, the audio is divided in pieces via windowing. Then the FTT of the chunk is evaluated, the result is sent to the Mel's filters. The number of features obtained per chunk is equal to the number of filters used.

Some other methods of elaboration are:

- **MFCC**: Mel-frequency cepstral coefficients (MFCCs) are based on mel coefficients but are uncorrelated since adjacent mel coefficients, derived from overlapping frequency bands, are strongly correlated. To obtain the MFCCs, the logarithm is applied to the mel coefficients, followed by the *Discrete Cosine Transform* (DCT), defined by:

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n + \frac{1}{2}\right)k\right], \quad k = 0, 1, \ldots, N-1$$

The DCT is used to decorrelate the coefficients.

- **Cochleogram Overview**: The cochleogram is a spectrogram inspired by the human cochlea and is based on the gammatone filter:

  $$h(t; f_c) = a\, t^{n-1} e^{-2\pi b\, t} \cos\bigl(2\pi f_c t + \phi\bigr), \quad t \geq 0$$

  This filter has shown promising results, sometimes outperforming other spectral analysis methods.

1.4.2 Sound Augmentation

To improve the network flexibility and the amount of data we used some sound augmentation technique. Sound augmentation consists in adding to the dataset some modified version of his own data. Here there show the technique we used:

- **Speed perturbation**: Resample the data a little slower/faster. This modifies the frequency spectrum.
- **Time Dropout**: Some audio parts are replaced with 0 to teach the network to handle signals with missing parts.
- **Frequency Dropout**: Some spectral parts are replaced with 0 to teach the network to handle signals with missing components.
- **Additive noise**: Noise is added to the signal to mimic environmental corruption.

1.5 Architectures

Once the frequency features are extracted, the data are soon elaborated by a DNN architecture. An architecture is a way to organize the network by dividing it into parts and assigning them subtasks. Every architecture can have a peculiar loss function projected for the network or for one of his subpart . During the years many architectures have been proposed, but in this thesis we deepen 2.

1.5.1 Sequence-to-Sequence Models

This is the architecture that we used in our implementation, it is showed in figure 1.1. It is one of the first, simpler but efficient architectures used in speech recognition. His semplicity has been the main reason why we decided to use it. It is composed by 2 parts.

The first part is the *encoder*: its role is to condense the information from the frequency features in a significant representation, called context vector. Often the encoder is the larger and more important part of the network.

The second part is the *decoder*: it generates the final output based on the context vector and the previous output. This feedback behavior is defined as *Autoregressive*. The decoder at every character has to focus on a different part of the context vector as we do when we are listening a discussion. To do it, the decoder implements an attention mechanism.

Attention Mechanisms

An attention mechanism can be described as a function that maps a *query* and a set of *key–value* pairs to an output, where the query, keys, values, and output are all vectors. In this section, we discuss the *Scaled Dot-Product Attention* method introduced in *“Attention Is All You Need”* [16], which is also implemented in the *SpeechBrain* toolkit.

The input consists of queries and keys of dimension d_k, and values of dimension d_v. The mechanism computes the dot product between each query and all keys, divides each result by $\sqrt{d_k}$ to maintain stable gradients, and then applies a softmax function to obtain the attention weights over the values. Finally, the weighted sum of the values produces the output context vector. Formally, the operation is defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V$$

Here, Q represents the *queries*, K the *keys*, and V the *values* (also known as the *context vectors*).

Teacher forcing

During the training, is evident that the token predicted is not always the correct one. The wrong prediction does not allow to train the network to correctly identify its successor. To overcome this difficulty in the training phase, we pass at the decoder, as previous token, not the predicted one but the correct one. This is called *teaching forcing.*

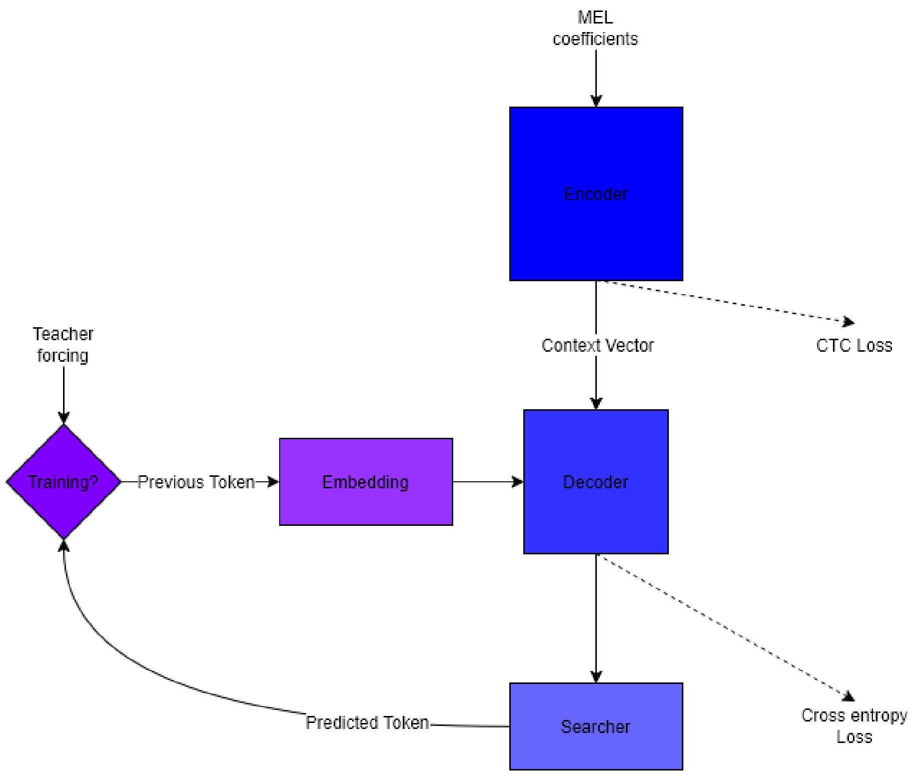


Figure 1.1: A seq2seq network. During training the previous token is taken by theacher forcing, in inference from the searcher.

1.5.2 CTC Loss

One of the main problems in speech recognition, as already said, was the alignment of audio to text. The CTC loss function solved it. At each frame is assigned a token, but using 2 basic ideas:

- Exist a special character that has no a written meaning called *blank*
- More adjacent and identical tokens encode for that token only one time

The blank character encodes the idea that the previous/next frame contains the same character as the analyzed one. This allow to have diverse length of output and time stamp.

The CTC is often used to create another type of network called CTC network, basically composed by an encoder trained using this function. In seq2seq models, it is used to accelerate the convergence of network training.

The idea is to consider all possible token sequences that can give the correct output. For each decoder step, for each valid sequence, evaluate the probability to be predicted, then sum all the probabilities to get the loss function.

Mathematical definition

$$\mathcal{L}' = \mathcal{L} \cup \{\varnothing\},$$

is the extended vocabulary considering the blank. Now consider

$$p_t(k) = P(\pi_t = k \mid X), \quad k \in \mathcal{L}',\ t = 1, \ldots, T$$

where $p_t(k)$ is the probability distribution to get a valid k sequence at time stamp t and $\pi_t = k$ are all the sequences valid at the step t. We can also define B as the collapse map:

$$B : (\mathcal{L}')^T \to \mathcal{L}^{\leq T}, \quad B([\varnothing, a, a\varnothing, b]) = [a, b]$$

Then we can formalize the problem in this way:

$$P(z \mid X) = \sum_{\pi \in B(z)} \prod_{t=1}^{T} p_t(\pi_t),$$

We define the CTC loss as:

$$\mathcal{L}_{\text{CTC}}(X, z) = -\log P(z \mid X)$$

1.5.3 Cross entropy loss

The cross entropy loss is the most used function to train seq2seq models. The idea is to evaluate the difference between the probability distribution of the output and the correct values. In order to understand, we have to introduce 2 concepts:

Entropy is the amount of information encoded in the distribution P:

$$H(P) = -\sum_x P(x)\, \log P(x)$$

In paralel we can define another measurement called cross entropy that evaluates the mean difference of information needed to correctly identify an event from P when we take it from Q

$$H(P, Q) = -\sum_x P(x)\, \log Q(x)$$

In the cross entropy we want to minimize this difference (ideally to 0). In particular in discrete form we get:

$$\mathcal{L}_{\text{CE}} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} p_{ij}\,\log q_{ij}$$

with N number of token, C number of component of the vectors used in the decoding, p_{ij} component of the correct token vector, q_{ij} component of the predicted token vector.

1.5.4 Transducer Models

Transducer models are an evolution of the seq2seq models and they are composed of three parts:

- *Encoder*: transforms the input features into a meaningful representation.
- *Prediction network*: takes the previous output and generates a representation of it.
- *Joint network*: takes as input the outputs of the previous layers to generate the final output. Its last function is a softmax, which produces an output suitable for classification.

The Transducer architecture has long represented the state of the art in automatic speech recognition. For example, ContextNet, which employs a fully convolutional encoder within the RNN-T (Recurrent Neural Network Transducer) framework, achieved a Word Error Rate (WER) of 1.9% on the LibriSpeech test-clean set and 4.1% on the test-other set [3].

1.6 Commonly Used Layers

1.6.1 Convolutional Layers

The convolutional layers are widely used in most machine learning applications due to their capability to condense information and to reduce the amount of data. In relation to the structure exposed before, they are often used in decoders. A layer takes more elements of input at the same time and evaluates a convolution using a filter called *kernel.* The kernel can be a vector or a matrix . The kernel slides then on the next sequence of input, the amount of element it slides on is called *stride.* When the window arrived on the edge there are 2 possibilities:

The kernel stops and the output is smaller than the input.

At the end of the data some zeros are added to get an output of the same size of the input, this action is called *padding*.

Sometimes, for each layer, there are more kernels to extract more representation from an input. In this way the number of dimensions of the output grows. The dimensions are called *channels*. When we have more channels on the input, everyone contributes to the output of every channel.

Pooling

Often we also want to use convolutional layers to reduce the number of inputs for the next layers. To do it, after the elaboration we select the best representation, or use a mean form of the output. This is called *pooling*.

The output of the convolutional layer is divided into boxes composed of a number *pooling size* elements of the same channel. Based on the operation applied, we have: *mean pooling* if we do the mean of all the elements in the box; *max pooling* if we take only the higher values of the box. As the kernel, the pooling size can have one or more dimensions.

1.6.2 Recurrent Layers

Recurrent layers are composed by cells having a variable depending on the previous input, called *hidden state*. In this way they can implement a memory structure.

Elman RNN

All the next cells are recurrent, but from this point, when we write about Recurrent Neural Network (RNN) layers or RNN cells, we talk about Elman's cell, that is the implemented type of cell in the RNN layer of pytorch. The function to evaluate the hidden stat is:

$$h_t = \tanh\big(x_t W_{ih}^\top + b_{ih} + h_{t-1} W_{hh}^\top + b_{hh}\big)$$

where *tanh* can be replaced with RELU and $b_{ih} b_{hh}$ are biases. The output of the cell is h_t.

LSTM

Long-Short Term Memory (LSTM) represents a more complex type of recurrent cell. They were introduced to mitigate the *vanishing gradient* problem that occurs in Elman's RNNs. An LSTM cell, as showed in the scheme 1.2 is composed of three main gates:

- *Forget gate*: decides whether the previous information should be discarded. It takes as input the previous cell state c_{t-1}, then applies the sigmoid function to the concatenation of the previous hidden state and the current input. If the result is close to zero, the corresponding information in the memory is practically not forwarded. This mechanism is implemented through the Hadamard product between the sigmoid output and c_{t-1}.

- *Input gate*: determines which parts of the new input should contribute to the updated cell state c_t. It takes as input the concatenation of the previous hidden state and the current input, applies a sigmoid function to decide what to remember, and a tanh function to process the candidate values. The resulting values are combined with the output of the forget gate to update the cell state.

- *Output gate*: decides which parts of the cell state will be emitted as output. It applies a tanh activation to the current cell state and multiplies it (element-wise) by a sigmoid gate computed from the concatenation of the previous hidden state and the current input. The result is the new hidden state h_t.

The equations governing the LSTM cell are the following:

$$
\begin{aligned}
i_t &= \sigma(W_{ii}\, x_t + b_{ii} + W_{hi}\, h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}\, x_t + b_{if} + W_{hf}\, h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}\, x_t + b_{ig} + W_{hg}\, h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}\, x_t + b_{io} + W_{ho}\, h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

where $\odot$ denotes the Hadamard (element-wise) product, and σ is the sigmoid function defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

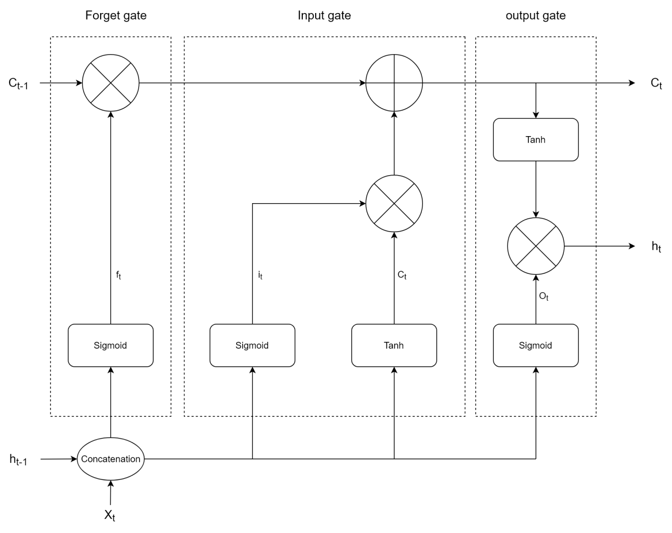


Figure 1.2: LSTM cell

1.7 Final decoding

The output of the decoder is not directly the final result. A *searcher* module is introduced to integrate external components and correct possible imbalances in the network. It evaluates a score based on the decoder predictions and other factors, and the token with the highest score is selected as output.

Here are some components that can be added to the scoring function: *Language model*, which provides the probability of a word sequence (similar to an autocorrect system); *length normalization*, a rescaling term to prevent longer hypotheses from being over-favored; *maximum attention shift*, which limits how much the attention peak can move between decoding steps to avoid unstable hypotheses; and *coverage penalty*, which penalizes attention distributions that leave gaps ("holes") in the context vector. Its formula is given by:

$$\text{Penalty}_t = \sum_{i=1}^{|X|} \big(\max(\text{covi}(t), \tau) - \tau\big)$$

$$\text{covi}(t) = \sum_{u=1}^{t} a_{u,i}$$

$$s_{\text{cov}}(t) = -\frac{\text{Penalty}_t}{t}$$

When we chose at every step always the most likely hypothesis we apply a Greedy approach.

1.7.1 Beam searcher

To increase the performance, at each generation of a token diverse hypothesis are keep. When all arrive to generate the eos toeken (or arrive to the maximum length allowed by the programmer), the most likely phrase is chosen. The number of hypothesis keep every iteration is called *beam size.* The total score is:

$$S(Y \mid X) = lp(Y) \log P_{\text{dec}}(Y \mid X) + w_{LM} \log P_{LM}(Y) - w_{\text{cov}} s_{\text{cov}}$$

1.8 Benchmarks

1.8.1 Evaluation Metrics

We have talked about how networks built for speech recognition work and how they are trained, the last step consists in evaluating their performance. The main metrics used in speech recognition are:

- *WER, Word Error Rate*: the number of words not recognized. Is evaluated:

  $$\text{WER} = \frac{S + D + I}{N}$$

  where S is the number of Substitutions (words not correct), D number of Deletions (missing words), I number of Insertions (added words) and N is the amount of words in the correct sentence. It is considered the most important one.

- *WRR, Word Recognition Rate*: number of correct words. It is evaluated using

  $$\text{WRR} = 1 - \text{WER}$$

- *CER, Character Error Rate*: number of wrong tokens recognized. Is evaluated:

  $$\text{WER} = \frac{S + D + I}{S + D + C}$$

  where S is the number of Substitutions (wrong tokens), D the number of Deletions (missing tokens), I the number of Insertions (added tokens) and C the amount of tokens in the correct sentence.

Chapter 2

Spiking Neural Network Background

In this chapter, we will provide the basic idea behind spike neuron and spike neural network. We will talk about the idea, the basic mathematic models and the training methods. Then we will show some example of spiking in speech recognition and similar tasks,we will deepen the aspects that we used in our project. Then we will close talking about hybrid architectures.

2.1 Overview

A spike neuron is a spike inspired by biological neurons. Biological neurons have a membrane controlling the entering of ions and in this way the electric potential of the cell. When a certain potential is achievied, the membrane discharges emitting a spike. The spike arrives to other neurons via paths called synapsis. The arrive of a spike can excite or inhibit the destination. The emitter is defined as *pre synaptic neuron*, the receiver is called *post synaptic neuron*.

An easy and similar mathematic model that we have to describe a neuron is the RC circuit: the membrane is the capacitor, the tendency of the neuron to lower is potential in absence of stimuli can be described considering in parallel a resistance, the synapsis can be modeled as resistor that transform the pre synaptic spikes in current. This model is the base of the Leaky Integrate-and-Fire (LIF) neuron that will deepen after. There are other models similar to biological behavior, but are more complex and less used because the benefits usually do not justify the required power increment during the training.

2.2 Leaky Integrate-and-Fire (LIF) Neurons

The simplest and most widely used model is the LIF neuron. This neuron model originates from the RC circuit analogy 2.1. Let us derive its characteristic equation.

We know that:

$$I_{\text{in}}(t) = I_R + I_C$$

where I_R is the current through the resistor and I_C is the current through the capacitor. According to Ohm's law:

$$I_R(t) = \frac{U_{\text{mem}}(t)}{R}$$

and from the definitions of capacitance and current we obtain:

$$Q = CU_{\text{mem}}(t)$$

$$\frac{dQ}{dt} = I_C(t) = C\frac{dU_{\text{mem}}(t)}{dt}$$

Substituting these quantities into the equation for I_{in} we obtain:

$$I_{\text{in}}(t) = \frac{U_{\text{mem}}(t)}{R} + C\frac{dU_{\text{mem}}(t)}{dt}$$

Rewriting it to highlight the most important components:

$$\implies RC\frac{dU_{\text{mem}}(t)}{dt} = -U_{\text{mem}}(t) + RI_{\text{in}}(t)$$

$$\tau\frac{dU(t)}{dt} = -U(t) + RI_{\text{in}}(t)$$

where $U(t)$ is the *membrane potential*, R is the resistance, and τ is the *time constant*, defined as $\tau = RC$.

In this way, we have obtained the dynamic equation of the spiking neuron using the RC model. However, working with derivatives in the discrete domain is difficult. To solve this, we use the *Euler method* to approximate the differential equation:

$$\tau\frac{U(t+\Delta t) - U(t)}{\Delta t} = -U(t) + RI_{\text{in}}(t)$$

which can also be written as:

$$U(t+\Delta t) = U(t) + \frac{\Delta t}{\tau}\big(-U(t) + RI_{\text{in}}(t)\big) \tag{1}$$

assuming $\Delta t \ll \tau$.

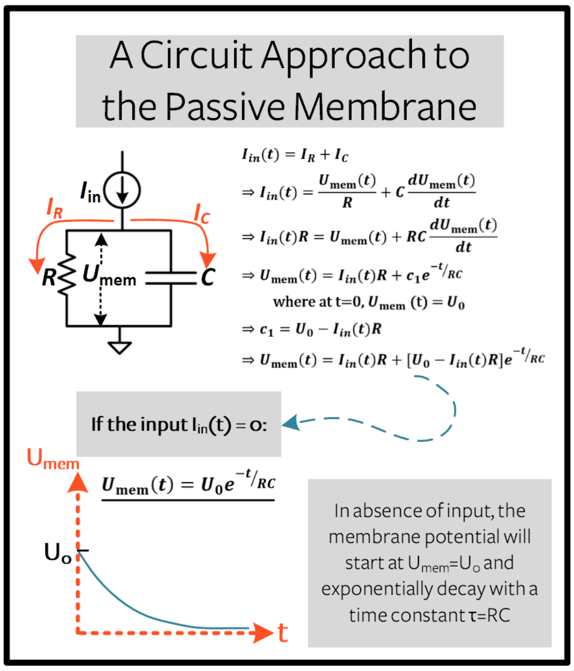

A Circuit Approach to the Passive Membrane
I_{in}
I_R
I_C
R
U_{mem}
C
$I_{in}(t) = I_R + I_C$
$\Rightarrow I_{in}(t) = \frac{U_{mem}(t)}{R} + C\frac{dU_{mem}(t)}{dt}$
$\Rightarrow I_{in}(t)R = U_{mem}(t) + RC\frac{dU_{mem}(t)}{dt}$
$\Rightarrow U_{mem}(t) = I_{in}(t)R + c_1 e^{-t/RC}$
where at t=0, $U_{mem}(t) = U_0$
$\Rightarrow c_1 = U_0 - I_{in}(t)R$
$\Rightarrow U_{mem}(t) = I_{in}(t)R + [U_0 - I_{in}(t)R]e^{-t/RC}$
If the input $I_{in}(t) = 0$:
U_{mem}
$U_{mem}(t) = U_0 e^{-t/RC}$
U_0
t
In absence of input, the membrane potential will start at $U_{mem}=U_0$ and exponentially decay with a time constant τ=RC

Figure 2.1: RC model of a LIF neuron, figure taken by [15]

This form can already be used in practice, and the model is known as the *Lapicque neuron*. However, it has many parameters, making it computationally difficult to handle and even harder to train. It would therefore be useful to simplify it further.

Assuming $I_{\text{in}}(t) = 0\ A$:

$$U(t + \Delta t) = (1 - \frac{\Delta t}{\tau})U(t)$$

We also know that if the input is zero, the decay of the membrane potential can be expressed as:

$$U(t + \Delta t) = \beta U(t)$$

where β is the *membrane decay rate*, given by:

$$\beta = (1 - \frac{\Delta t}{\tau})$$

Given this simplification, we can proceed further by assuming $\Delta t = 1$ (since we are working in the discrete domain) and $R = 1$ to further reduce the number of hyperparameters in the network:

$$\beta = (1 - \frac{1}{C}) \implies (1 - \beta)I_{\text{in}} = \frac{1}{\tau}I_{\text{in}}$$

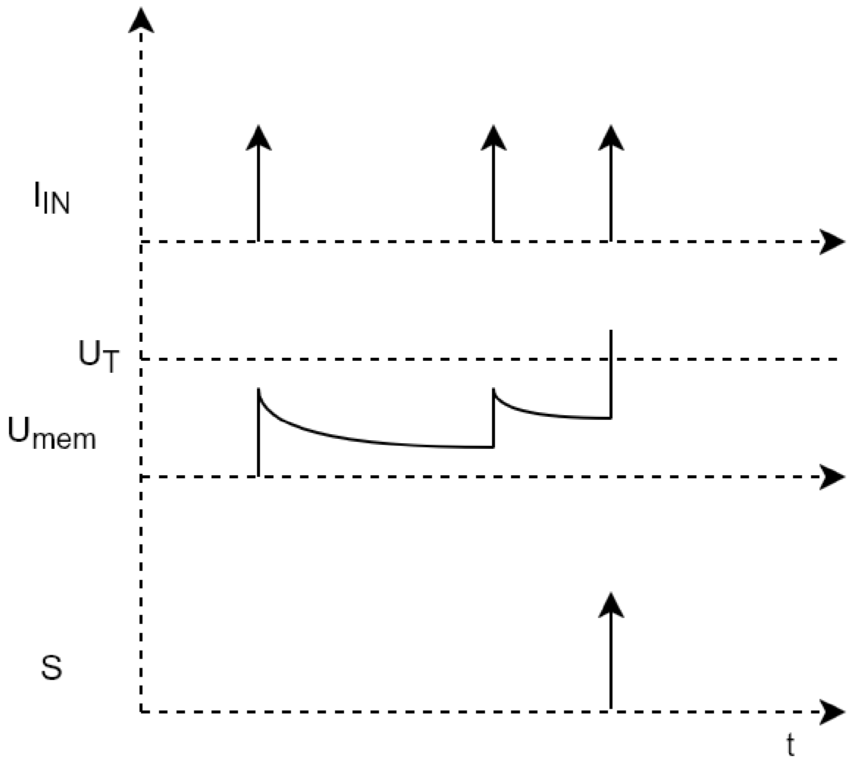


Figure 2.2: Potential of the membrane and spikes function of the current I_{IN}

Substituting everything into Equation (1), we obtain:

$$U[t+1] = \beta U[t] + (1-\beta) I_{\text{in}}[t+1]$$

This formula is very simple and practical. To make it useful in a neural network context, we only need to identify which term represents the input and which is trainable. Referring back to the RC model, the input value is the current $I_{\text{in}}[t+1]$, which we can denote as $X[t+1]$.

In a typical neural network, the learnable parameter is the input weight. In this case, the term $(1-\beta)$ corresponds to that weight. Abstracting from the physical model, we thus have:

$$U[t+1] = \beta U[t] + WX[t+1]$$

where W is the *learnable weight*, decoupled from the behavior of β. The resulting model is very simple: it has only two hyperparameters (β and the threshold) and a single learnable parameter (W). The final functioning is showed in 2.2.

2.2.1 Reset Mechanism

We know that when the membrane potential reaches a certain threshold, the neuron spikes. But what happens after that? The way in which the membrane potential is managed after a spike is referred to as the *reset mechanism*:

$$S[t] = \begin{cases} 1, & \text{if } \; U[t] > U_{\text{thr}} \\ 0, & \text{otherwise} \end{cases}$$

The *subtractive reset* mechanism is based on the idea of subtracting the threshold potential from the membrane potential, preserving any residual potential induced by the spike. This approach maintains a higher information content since the surplus is not completely discarded:

$$U[t+1] = \underbrace{\beta U[t]}_{\text{decay}} + \underbrace{WX[t+1]}_{\text{input}} - \underbrace{S[t]U_{\text{thr}}}_{\text{reset}}$$

A variant of this method only subtracts part of the potential, scaled by β. The equation becomes:

$$U[t+1] = \underbrace{\beta U[t]}_{\text{decay}} + \underbrace{WX[t+1]}_{\text{input}} - \underbrace{\beta S[t]U_{\text{thr}}}_{\text{soft reset}}$$

This *soft reset* approach has been shown to perform better in many cases [12].

Another reset mechanism, the *zero reset*, sets the membrane potential to zero after a spike, regardless of any surplus potential. This method tends to reduce the total number of spikes, thus improving energy efficiency, but at the risk of losing information.

2.3 Encoding

Information in the brain can be represented through spikes, making it essential to understand how this encoding process occurs. For optimization reasons, information is typically encoded in a *sparse* manner, improving energy efficiency. Moreover, neural processing is inherently *event-driven*, meaning that signals are generated and recorded only when significant changes occur.

It is well established that the brain employs at least three main forms of spike-based encoding: *rate coding*, which represents information through spike frequency; *latency coding*, which encodes information through spike timing; and *differential coding*, which transmits spikes only in response to variations in the input signal.

Rate coding appears to conflict with the principle of energetic optimization, as it requires a relatively high number of spikes. Furthermore, it cannot fully explain the brain's ability to process certain stimuli extremely rapidly. In order to obtain a reliable estimate of spike frequency, a sufficiently large temporal window is required—the larger the window, the better the estimation, but the slower the reaction time. Empirical evidence suggests that rate coding accounts for the activity of 15% subset of neurons in the primary visual cortex. Nevertheless, it remains a fundamental and biologically verified mechanism for neural communication.

Latency coding, on the other hand, represents information through the timing of spike emissions. Although this approach is more sensitive to noise, it enhances sparsity and can lead to faster response times. In such schemes, an earlier spike conveys a stronger or more significant signal than a later one.

A further class of methods includes **differential codings**, in which spikes are generated only when there is a change relative to a previous signal. In these cases, spikes can be bipolar $(-1, 1)$, enabling the system to encode not only the change but also its direction. This mechanism is considered biologically plausible and is reminiscent of the processing that occurs in certain sensory organs, such as the retina.

To represent continuous values over time, one approach is to **extend** each value along the temporal dimension, effectively replicating its information across multiple time steps. This method, known as *duplication*, is straightforward to implement.

Alternatively, stochastic encoding techniques can be employed, such as those introducing Gaussian-distributed variability in the spike generation process. The **Gaussian encoding** adds an additional Artificial Neural Network (ANN) layer operating in parallel with the existing one. One branch outputs the mean value of the spikes, while the other provides the natural logarithm of the standard deviation. Using these parameters, a noise variable is introduced to generate a spike train following a Gaussian distribution. Although this method is more complex, it often yields better performance in more challenging tasks, as it introduces controlled variability in the spike representation.

Another method is **Poisson encoding**. When encoding an analog number, the residual reconstruction error due to the discrete number of spikes generated within a given time interval decreases proportionally to $1/N_{\text{spikes}}$. However, because Poisson-distributed spike trains introduce intrinsic variability, the error decreases only as $1/\sqrt{N_{\text{spikes}}}$. This method does not always perform well; for example, in [6], a comparison among these three encoding strategies concluded that Poisson encoding was not particularly advantageous.

Beyond rate-based representations, several **temporal encoding** approaches have been proposed, which convey information through the precise timing of spikes rather than their overall rate. One example is the *filter-based encoding*, which applies specific temporal filters to transform analog inputs into spike sequences. Although effective in certain contexts, this approach can be computationally demanding and sensitive to the statistical properties of the input data. Filters often require adaptation to match the characteristics of the processed signals—for instance, those suited for audio data may not be optimal for visual features.

A well-known example is the **BSA (Ben's Spike Algorithm)** encoding method, which uses a filter whose frequency response closely resembles that of the human auditory system [11].

Temporal encoding methods are particularly promising from an energy-efficiency perspective, as they can substantially reduce the average firing rate. Among them, one of the most widely adopted is the **Time To First Spike (TTFS)** coding, which represents the input value as the delay between the start of a cycle and the emission of the first spike. A simple implementation maps the input amplitude to a spike time inversely proportional to the input magnitude. Another common approach is the **Sinspike encoding**, where the spike delay is determined according to:

$$t_i = (1 - \sin(X_i \cdot \tfrac{\pi}{2})) \cdot T_{\max}$$

This method is particularly suited to temporal signals, such as speech or auditory data, as the sinusoidal transformation naturally reflects their dynamic behavior. In both TTFS and Sinspike encoding, each timestamp corresponds to a spike train, preserving the temporal structure of the original data.

In [18], the **TTFS** approach is compared with alternative methods such as **Send on Delta (SOD)** and **LIF**. The **LIF** approach employs a Leaky Integrate-and-Fire neuron to integrate inputs and generate a spike train. The **SOD** algorithm detects continuous values over a defined sampling period and generates discrete spikes whenever the magnitude of an increase or decrease exceeds a given threshold.

Both SOD and LIF process the entire input sequence to generate spike trains, rather than operating frame by frame. The study also highlights the distinction between **Spectrogram** and **Cochleagram** representations, showing that the latter provides a more effective and biologically inspired encoding for spiking systems. Furthermore, the authors propose a signal reconstruction step using a Finite Impulse Response (FIR) filter before passing the data to a standard neural network.

2.4 Training

We know that the spike output function is defined as:

$$S[t] = \Theta(U[t] - U_{\text{thr}}) \tag{1}$$

where Θ is the *Heaviside step function.* This poses a problem because the Heaviside function is *non-differentiable.*

In a classical neural network, training is performed via *backpropagation*, where gradients are computed from the loss function to update the parameters in the direction that minimizes the loss. However, if a function is non-differentiable or has a derivative equal to zero everywhere else backpropagation cannot work properly. The derivative equal to zero leads to the so-called *dead neuron problem*, where neurons stop learning because their gradient is zero.

To overcome these issues, various techniques have been developed that allow training without traditional backpropagation. Here, we focus on the *Surrogate Gradient* approach, which enables the use of backpropagation even with spiking neurons.

2.4.1 Surrogate Gradient

The *surrogate gradient* method is based on using two different functions for the *forward pass* (loss computation) and the *backward pass* (gradient computation). This allows us to avoid the problems described above.

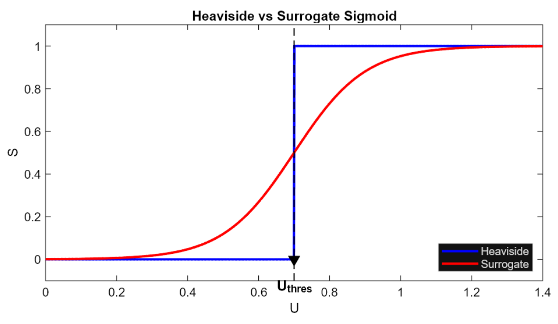
One commonly used surrogate function is the *sigmoid function.* Referring to our earlier notation, and considering $U_{OD} = U - U_{\text{thr}}$, we can define:

$$\tilde{S} = \frac{U_{OD}}{1 + k|U_{OD}|}$$

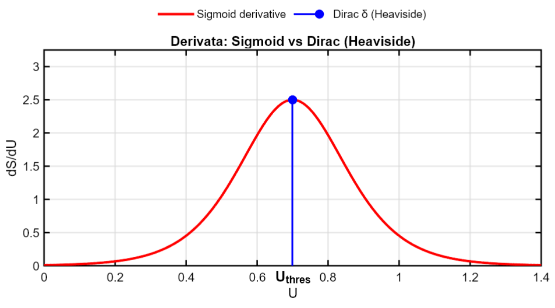
whose derivative is:

$$\frac{\partial \tilde{S}}{\partial U} = \frac{1}{(k|U_{OD}| + 1)^2}$$

Other functions that can serve as surrogate gradients include, for example, the *arctangent* function.



(a) Surrogate sigmoid vs Heaviside



(b) Surrogate derivative vs Heaviside derivative

Figure 2.3: Comparison between surrogate activation and its derivative with respect to the Heaviside function

2.5 Spiking in ASR

The study of spiking neural networks for *speech recognition* is relatively recent; however, several promising results have already been achieved, particularly in the context of **edge devices**. In the following section, we review some of the most relevant studies in this field.

In [17], an acoustic model based on a Spiking Neural Network (SNN) is proposed. The model builds upon the framework described in Section 1.1 of this thesis and is **not** an end-to-end network. The results show that the performance is comparable or slightly worse than traditional approaches, with the performance drop attributed to the discretization introduced by the spikes. The training is performed in a *tandem* manner, meaning that the weights are shared with an equivalent Artificial Neural Network (ANN): the SNN is used during the forward pass, while the ANN is used during the backward pass. The encoding scheme operates frame-by-frame and uses a ReLU activation to generate a spike train of N elements from each frame. Decoding is performed through the membrane potential.

In [8], several network architectures composed of different types of spiking and non-spiking neurons are compared. The study highlights that, although networks built with traditional LIF neurons are generally less performant than classical artificial neurons such as Long-Short Term Memory (LSTM), introducing a structure that enhances their recurrent dynamics allows the performance to approach that of ANN-based models, while retaining the energy efficiency benefits of spike-based computation. Training is carried out via backpropagation using the *surrogate gradient* technique. The input signal is encoded simply by integrating continuous-valued data as input current.

In [2], a Sequence-to-Sequence (seq2seq) network with four LSTM layers is progressively modified by replacing the LSTM layers with LIF layers. The resulting hybrid networks are then compared with equivalent models where the spiking layers are replaced by standard Recurrent Neural Network (RNN) layers. In this case as well, the data are integrated directly at the input of the spiking layers. This study demonstrates that **surrogate gradient-trained SNN** are compatible with large, end-to-end, sequence-to-sequence modern architectures. This addresses the primary research question of whether SNN can scale to more advanced tasks and deeper networks. Moreover, across all tasks, the spiking layers which contain four times fewer trainable parameters were able to replace LSTM with only minor performance degradation. Although retaining a single LSTM layer still significantly helped reduce the error rate, this result shows that the information processed within neural networks can be efficiently reduced to **sparse and binary events** without substantially compromising their encoding capabilities.

Considering smaller-scale studies on simpler tasks such as *command recognition*, it can be seen that **spiking convolutional networks** are also being explored. In [7], the inputs are first integrated by an LIF neuron to generate a spike train, followed by a 2D convolution applied over time and frequency dimensions. The convolution output is then passed through another LIF neuron to maintain the spiking format. This network is also trained using backpropagation.

In [19], the potential benefits of integrating spikes into **Transformer architectures** for speech recognition and other speech tasks are investigated, achieving both higher accuracy and improved energy efficiency compared to their ANN counterparts. The training is done using backpropagation. Although this line of research has not been explored in depth in our work, it undoubtedly represents a highly active and promising research direction.

Overall, research on **speech recognition** within the Automatic Speech Recognition (ASR) domain is very broad, but there is a clear shift towards **end-to-end networks trained with backpropagation**. Our work fits precisely within this line of investigation.

2.6 Spiking LSTM (SLSTM)

The operation of the **Spiking Long-Short Term Memory (SLSTM)** cell is analogous to that of a conventional **LSTM** cell, with the difference that the output is *spike-based*, generated through a thresholding mechanism. The short-term memory is reset according to the chosen strategy, while the long-term memory (the *cell state*) is replaced by **synaptic memory**.

In the `snnTorch` implementation, it is also possible to define the threshold as a trainable parameter, allowing the model to optimize it during the training process.

2.7 Spiking CNN (SCNN)

Convolutions play a particularly important role in *speech recognition*, and they remain fundamental in the spiking domain as well. We have already seen an example of this; however, several different methods have been proposed to implement **Spiking CNNs**. Below, we review some relevant studies addressing these approaches:

In [14], a spiking network is used in which the convolutional layers receive inputs encoded through Poisson spikes. The convolutional filters are connected via inhibitory weights that maintain activation sparsity. At the end of the convolutional layers, leaky lif neurons with a zero-reset mechanism are used. The convolution result is decoded through the firing rate. From a mathematical point of view, convolution based on spikes and firing rates has been shown to be equivalent to a traditional convolution. Pooling is performed by selecting the neuron with the highest number of spikes (maximum activity).

The [4] study describes a multi-layer spiking network that uses integrate-and-fire neurons without leakage (non-leaky IF). The encoding is temporal, allowing only one spike per neuron within a given time window, while inhibitory weights are used to suppress activity of the others neuron across different features to increase their differences. Pooling is performed by taking the first spike occurring within a time interval, following a temporal winner-take-all mechanism.

The model proposed in [10] employs **LIF** neurons without inhibitory connections between filters. Both *temporal* and *firing rate* output encodings are tested. A key aspect of the work is that training is not performed using **STDP**, but rather through a **backpropagation-based** method specifically adapted for spiking neural networks. The study also investigates parameter optimization strategies aimed at improving overall network performance.

Chapter 3

Implementation

In this section, we describe the tools used and the workflow followed, providing the rationale behind our choices. The adopted approach is an ablation process: we started from an already functioning network and progressively modified it based on the results obtained. We then carried out several experimental attempts to evaluate the impact of the introduced changes.

3.1 Software Tools

3.1.1 PyTorch and snnTorch

PyTorch is an open-source machine learning framework widely used for building and training neural networks. Its core data structure is the *tensor*, a multidimensional array that efficiently represents and manipulates numerical data.

SnnTorch, built on top of PyTorch, extends its functionality to support *Spiking Neural Networks*. It provides a set of modules and tools to simulate neuron models that communicate through discrete spikes. The framework supports GPU acceleration via CUDA, enabling efficient large-scale simulations. Furthermore, it includes utilities such as `torch.utils.data`, which facilitate dataset management, preprocessing, and loading during the training process.

3.1.2 BrainSpeech Framework

BrainSpeech is a framework designed to support a wide range of tasks in the field of speech-related artificial intelligence, making it a highly flexible and adaptable tool. The framework provides built-in support for several datasets in particular, *LibriSpeech*, which is the primary dataset used in this work and includes numerous usage examples with extensive documentation.

For these reasons, BrainSpeech represents an excellent foundation for developing new models, as it already includes implementations of training scripts and neural network architectures that can be easily adapted and extended. The entire configuration and management process is handled through `.yaml` files, which makes the framework particularly modular and customizable.

Within these configuration files, a standard audio processing pipeline based on *Mel Coefficients* is already implemented. This pipeline can be easily modified or reconfigured by adjusting the YAML parameters. In addition, the framework includes a built-in, fully configurable script for tokenization based on *SentencePiece*. Through the YAML configuration, users can specify the algorithm to be used, as well as the parameters for special tokens such as `<eos>`, `<bos>`, and `<blank>`, the number of subword units.

3.1.3 Setup

To ensure reproducible results and maintain a clean environment free from unnecessary elements that could cause compatibility issues, we decided to use *Singularity* to create isolated container. Inside the container, a dedicated virtual environment was further created using *Conda*, due to specific compatibility requirements with *SpeechBrain* packages.

The Singularity image files (`.sif`) are compact and read-only by design. This provides the advantage of preventing accidental modifications to the environment, ensuring consistency across experiments. However, this characteristic can also be inconvenient during development: even the absence of a single required dependency may require rebuilding the entire container from the definition file (`.def`), which is used to generate the image. This limitation does not apply to Python packages, which can be installed dynamically within the running container without the need to rebuild it.

To accelerate network training, we used the *High Performance Computing* (HPC) cluster of the Politecnico, which uses *SLURM* as its workload manager. SLURM is a software framework for job scheduling and resource management on HPC systems. To submit a job, an `sbatch` script must be provided, specifying the target partition, the number and type of GPUs and CPU to be reserved, the amount of GPU memory, and the container together with the commands to be executed inside it. It is also possible to specify one or more log files in the script to monitor the progress of the training process and to detect potential runtime errors. Once a job is submitted, an interactive shell can be attached to the main process using the `srun` command, allowing real-time monitoring of GPU utilization and system status.

3.2 Software

3.2.1 Training Settings

In this section, we describe the main training parameters and the configuration of the processing pipeline. All these parameters are defined and can be easily modified within the YAML configuration file. A resume is present at 3.1

The number of *epochs* represents how many times the entire dataset is presented to the network during training. The parameter `ctc_epoch` defines the number of epochs in which the CTC loss contributes to the overall loss function, and its influence is determined by the parameter `ctc_weight`. The *batch size* specifies the number of samples processed in parallel by the GPU. As discussed later, this value differs between the training/validation and testing phases. In particular, during testing, the batch size is typically set to one to better emulate real-world conditions, where the system usually processes a single audio input at a time.

The parameter *sorting* defines the order in which audio samples are processed based on their length. For instance, setting it to `ascending` processes shorter samples first, which reduces the amount of padding required within each batch. Since all samples in a batch must share the same length, shorter sequences are padded to match the longest one before being processed by the network. Alternatively, `descending` or `random` sorting can be used, depending on the desired training dynamics. The *dynamic batching* option allows the batch size not to be fixed in advance; instead, it can increase or decrease automatically depending on the computational load and memory availability, starting from an initial value.

Regarding the audio features, key parameters include the *sample rate*, *n_fft*, and *n_mels*. The sample rate directly affects the number of time steps in the feature representation: a higher sampling rate produces a greater number of timestamps. The parameter `n_fft` specifies the number of frequency coefficients used in the fast Fourier transform, while `n_mels` defines the number of Mel-scale coefficients extracted for each frame.

The *optimizer* is responsible for updating the model's weights according to the chosen learning rate, and the default settings were used in this work. The parameter `num_workers` refers to the number of processes employed by the *dataloader*, the component that transfers data to the GPU. The SpeechBrain framework itself recommends not exceeding the number of available CPU cores for this value, as doing so may lead to process contention and reduced performance.

It is worth recalling that the *validation* phase is used to evaluate the model's performance during training, ensuring that the learning process remains effective and that the model continues to generalize well. The *testing* phase, on the other

Table 3.1: Summary of the parameters described in this section.

Parameter	**Description**
Training parameters	
epochs	Number of times the entire dataset is presented to the network.
`ctc_epoch`	Number of epochs during which the Connectionist Temporal Classification (CTC) loss contributes to the total loss.
`ctc_weight`	Weight assigned to the CTC loss.
batch size	Number of samples processed in parallel; typically 1 during testing.
sorting	Order in which audio samples are processed.
dynamic batching	Allows the batch size to adapt based on memory and load.
Audio processing parameters	
sample rate	Audio sampling frequency.
`n_fft`	Number of frequency coefficients used in the Fast Fourier Transform (FFT).
`n_mels`	Number of Mel-scale coefficients extracted per frame.
Optimization and data loading parameters	
optimizer	Algorithm used to update model weights.
learning rate	Learning rate applied by the optimizer.
`num_workers`	Number of dataloader worker processes.
Experiment management and logging	
log file names	Names of log files generated during training.
seed	Identifier of the training.
dataset configuration	Assignment of dataset subsets (train/valid/test).
`ckpt_interval_minutes`	Time interval between checkpoint saves.
Numeric precision	
`precision`	Numerical precision used during training.

hand, represents the final evaluation stage. In our case, testing can differ not only in the batch size but also in the decoder configuration, which may use different beam sizes or additional parameters to optimize performance.

At the beginning of the YAML file, it is possible to set the output paths, the names of the log files, and most importantly the *seed*. The seed acts as an identifier for each training session and should be changed every time a new experiment is run, in order to avoid overwriting previous results. The dataset configuration section allows the user to specify which subsets of the dataset should be assigned to each of the three main phases: training, validation, and testing. The variable `ckpt_interval_minutes` defines how frequently (in minutes) a training checkpoint is saved. If training is interrupted, it can resume from the last saved checkpoint. The dataset transcription files, typically in `.csv` format, contain the text annotations used during training. In addition, paths for noise augmentation files can also be specified in the configuration.

The parameter `precision` controls the numerical precision used during training. Using 16-bit precision reduces memory consumption, allowing larger batch sizes, but may negatively impact convergence—especially when using surrogate gradient methods—due to the gradient mismatch problem. For this reason, 32-bit precision is used by default.

Model Parameters

This section describes the main model parameters, which may vary slightly depending on the specific experiment, but most of them are shared across all configurations. A resume is present at 3.2. The parameter `activation` defines the activation function used by the artificial neurons and is set to *Leaky ReLU* by default. The variable `cnn_cblocks` specifies the number of convolutional blocks in the network, while `cnn_channels` defines the number of channels in each block. The parameter `dropout` controls the dropout rate applied within layers to prevent overfitting. `SLSTM_neurons` determines how many hidden states per SLSTM layer are presented,`SLSTM_layers` decides the number of SLSTM layers. The parameters `dnn_blocks` and `dnn_neurons` determine the number of fully connected Deep Neural Network (DNN) blocks and the number of neurons per block, respectively. `Learn_threshold` sets if the threshold of spikes neurons' are learned during training, `threshold` set teh initial value of this. The parameter `emb_size` specifies the size of the embedding vector, `dec_neurons` sets the number of RNN neurons in the decoder, and `output_neurons` defines the number of output neurons, which must match the vocabulary size.

The indices `blank_index`, `bos_index`, and `eos_index` are all set to 0, as they are

Table 3.2: Main model parameters description.

Parameter	**Description**
Neural network architecture parameters	
`activation`	Activation function used by artificial neurons; default is Leaky ReLU.
`cnn_cblocks`	Number of convolutional (spiking or not) blocks in the network.
`cnn_channels`	Number of channels per convolutional (spiking or not) block.
`dropout`	Dropout rate applied within layers to reduce overfitting.
`SLSTM_neurons`	Number of hidden states per Spiking Long-Short Term Memory (SLSTM) layer.
`SLSTM_layers`	Number of SLSTM layers in the model.
`dnn_blocks`	Number of DNN blocks.
`dnn_neurons`	Number of neurons per fully connected layer of Character Error Rate (CER) block.
`Learn_threshold`	Flag indicating if spike neuron thresholds are learned.
`threshold`	Initial threshold value for spike neurons.
`emb_size`	Size of the embedding vector.
`dec_neurons`	Number of Recurrent Neural Network (RNN) neurons in the decoder.
`output_neurons`	Number of output neurons; matches vocabulary size.
Decoder indices and ratios	
`blank_index`, `bos_index`, `eos_index`	Indices set to 0, managed internally by the network.
`eos_threshold`	Minimum score for the eos.
`min_decode_ratio`	Minimum ratio between tokens generated and input timestamps.
`max_decode_ratio`	Maximum ratio between tokens generated and input timestamps.
Decoding parameters	
`valid_beam_size`	Beam width used during validation decoding.
`test_beam_size`	Beam width used during testing decoding.
`using_max_attn_shift`	Enables maximum attention shift mechanism.
`max_attn_shift`	Limit of maximum attention shift.
Softmax and coverage settings	
`temperature`	Temperature scaling factor applied before softmax; $T > 1$ flattens, $T < 1$ sharpens distribution.
`coverage_penalty`	Coefficient used for coverage penalty in decoding.

internally managed by the network. The *eos_threshold* is the minimum total score necessary to accept the eos as character. The parameters `min_decode_ratio` and `max_decode_ratio` determine the minimum and maximum ratio between the number of tokens generated by the decoder and the number of input timestamps. The parameters `valid_beam_size` and `test_beam_size` set the beam width for the decoder during validation and testing, respectively, as they may differ between phases. The parameter `using_max_attn_shift` enables the maximum attention shift mechanism, whose limit is specified by `max_attn_shift`. The parameter `temperature` controls the temperature scaling applied before the final softmax layer: the logits are divided by T, where $T > 1$ makes the distribution flatter (encouraging exploration), while $T < 1$ sharpens it (encouraging exploitation). Finally, the parameter `coverage_penalty` sets the coefficient t used for the coverage penalty term in the decoding process.

3.2.2 Starting point

3.2.3 Baseline Network: BrainSpeech Seq2Seq Model

We started from an existing speech recognition network and modified it for our purposes. Specifically, we used the *BrainSpeech* Sequence-to-Sequence (seq2seq) model with the same structure shown in Figure 1.1.

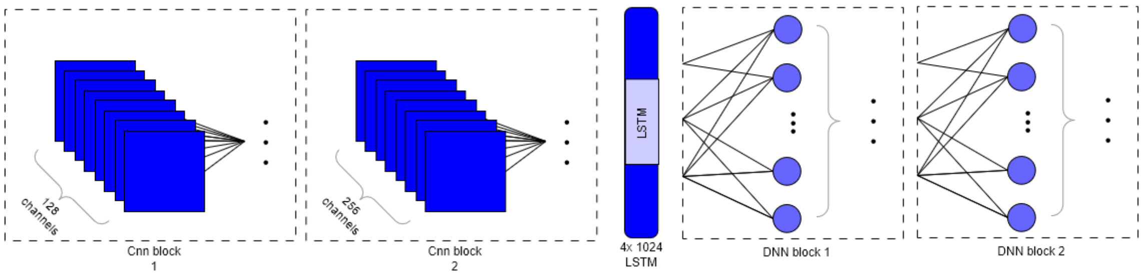


Figure 3.1: Original Encoder structure.

The network consists of a convolutional encoder and an attentional RNN decoder with a single layer. During inference, it employs a beam search mechanism. The main idea behind the encoder is to condense spectral and temporal information through convolutional processing and to model sequential dependencies through DNN and Long-Short Term Memory (LSTM) layers, which together produce the *context vector*. This context vector is then passed to the autoregressive decoder, which employs an attention-based mechanism and a beam search strategy with a beam size of 80.

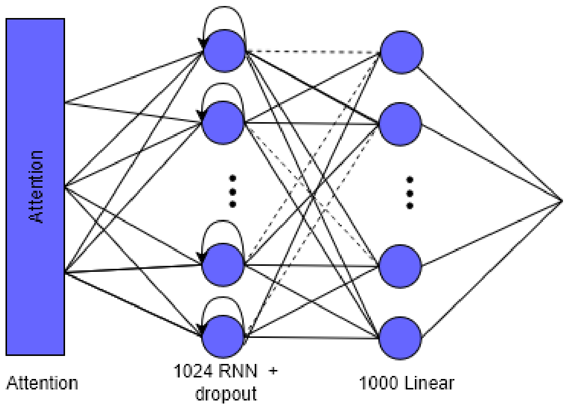


Figure 3.2: Decoder structure.

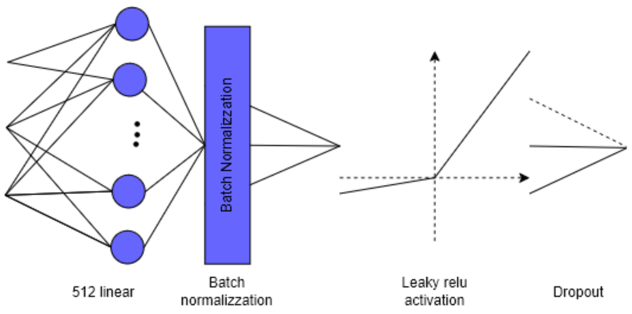


Figure 3.3: DNN block structure.

The Encoder 3.1 operates on Mel-spectrogram inputs arranged as $[Batch, Time, Mel]$. It begins with two convolutional blocks 3.4 that extract representations from the input features.

Each convolutional block consists of two 2D convolutional layers acting on MEL's and time, each followed by layer normalization and a ReLU activation function to ensure stable and non-linear transformations. Max-pooling layers are used to gradually reduce spectral resolution, emphasizing the most salient acoustic patterns. Dropout regularization is applied after each block to prevent overfitting.

The high-level features extracted by the convolutional layers are then passed through a stack of fourLSTM layers, each with a hidden size of 1024. This recurrent component captures long-term temporal dependencies and contextual relationships

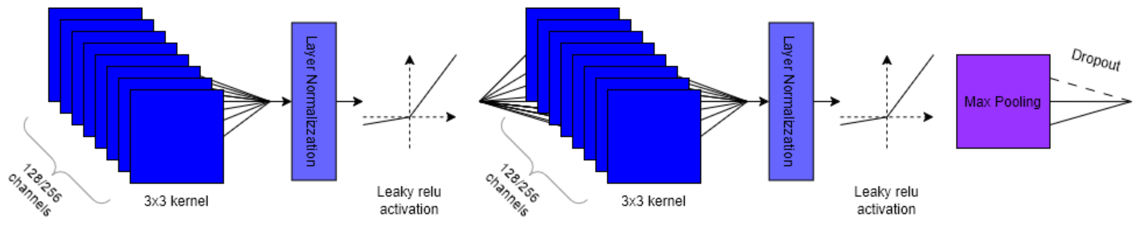


Figure 3.4: Original convolutional block.

across time steps. Afterward, two fully connected blocks 3.3 (DNN blocks) refine the encoded representations. Each of these blocks consists of a linear transformation with 512 neurons, followed by batch normalization, ReLU activation, and dropout.

The Decoder 3.2 receives two inputs: the encoded representation $[Batch, Time, DNN_neurons]$ and the previously generated token. The decoding process starts with an embedding layer (embedding dimension 128, vocabulary size 1000) that maps discrete tokens into a continuous vector space. An attention mechanism is then applied to dynamically weight the encoded features, allowing the model to focus on the most relevant temporal regions during decoding. The attended representations are processed through an RNN layer (in particular are used GRU neurons, that are an optimized version of LSTM, from the option is also possible to set other kind of RNN) followed by dropout. Finally, a linear output layer with 1000 neurons maps the decoder's hidden states to the vocabulary space, producing a probability distribution over possible output tokens.

3.2.4 Hybrid architectures and decoding

The process of progressively modifying a neural architecture to evaluate the contribution of each component is known as *ablation*. This approach is particularly useful for understanding how artificial neural networks (ANNs) and spiking neural networks Spiking Neural Network (SNN) can interact within a hybrid model, and how each modification affects the final performance.

In hybrid architectures, two main training strategies are typically employed. The first is **separate training**, in which the spiking and non-spiking layers are trained independently in different stages. The second is **simultaneous training**, where both types of layers are optimized together within the same learning process. Experimental evidence suggests that, for complex tasks, separate training generally leads to inferior performance, whereas simultaneous training allows a more coherent adaptation between the two components. [6]

Once the training strategy has been defined, it is also necessary to determine how to decode the spikes in order to extract meaningful information from them. In the case of rate-based encoding, one of the simplest rate-based decoding approaches is the *count-rate* method, where information is represented by the firing frequency within a defined temporal window.

We adopted the count-rate method for decoding and decided to train all the parts of the network at the same time. Our initial idea was to modify the decoder, as it constitutes the smaller portion of the network. However, we abandoned this approach to avoid altering the attention mechanism, the searcher module, or the loss function interface. In particular, the searcher is responsible for managing the memory reset in the RNN-based decoder, and it would require significant adaptation to handle spiking behavior correctly.

3.2.5 SLSTM

Following the approach proposed in [2], we instead decided to modify the **encoder**. The simplest way to achieve this was to replace the LSTM layers with **SLSTM** units. This choice allowed us to maintain compatibility with both the existing data processing pipeline and the CTC interface, minimizing structural changes to the rest of the system. For simplicity, we employed a firing-rate encoding scheme. To further facilitate the integration of the SLSTM module, a linear projection layer was inserted between the last convolutional block and the SLSTM layers 3.5 and the number of layers SLSTM was decreased from for to two to speed up the training. In order not to change the structure of the network, we encode every timestamp in a spiking way, to do it we used duplication as referred to [6] leaving more complicated methods at the further implementations. We chose 10 steps because it is one of the smaller values we found in the literature. For the thresholds, we initialized them based on some simulations using random values, and set them learnable. This approach was justified by [13] , that demonstrates how learnable thresholds can improve the performance of the network.

For this operation, we started from the *brainspeech* code of the CRDNN encoder. To add the linear layer, the following command was used:

Listing 3.1: Adding the linear layer to the CRDNN encoder

```
self.append(sb.nnet.linear.Linear, layer_name="linear_pre_spike",
            n_neurons=rnn_neurons, combine_dims=True)
```

Here, `sb.nnet.linear.Linear` is a custom linear layer from *SpeechBrain* that performs channel flattening simply by using `combine_dims=True`. The keyword `self` refers to the object `CRDNN_spike_encoder`, which is our modified encoder. This object is, in fact, a subclass of `sb.nnet.containers.Sequential`, a custom SpeechBrain class designed to instantiate layers sequentially within the encoder. This is particularly important because inserting a custom module inside such a

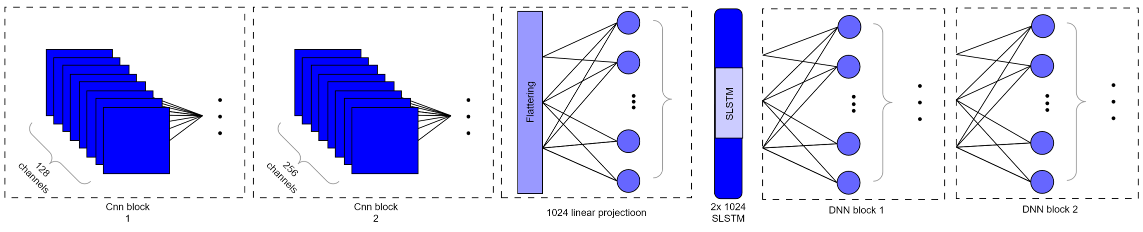


Figure 3.5: Encoder strcutre with SLSTM

structure requires the use of a wrapper belonging to the same class as the encodery. In our case, the wrapper is:

Listing 3.2: Wrapper class for SLSTM instantiation

```
class SLSTMCell(sb.nnet.containers.Sequential):
    def __init__(self, input_shape, hidden_size, num_layers=1):
        super().__init__(input_shape=input_shape)
        self.append(SLSTM, hidden_size=hidden_size,
                    input_size=input_shape[-1], num_layers=
                       num_layers)
```

This class does not define any additional behavior; it only serves to instantiate the `SLSTM` object, which is our actual module.

To make the network as modular as possible, the SLSTM layers were allocated using `nn.ModuleList`, a PyTorch module that allows the sequential and, most importantly, *parametric* allocation of multiple modules. This means it is possible to decide dynamically how many layers to include within the same block.

First, there is a memory management component. The SLSTMs handle one spike at a time, and the membrane potentials are initialized only at the beginning of the time stamp sequence; after that, the software must keep track of their updates. As we know, each neuron has two variables, `hx` and `cx`, which represent the short-term and long-term memory of the cell, respectively. From the outside, these values are passed as `None` so that the function knows it must initialize them.

After the memory phase, an empty list called `spikes_sum_list` is instantiated. The input has shape $[B, \text{stamp}, \text{feature}]$. Since duplication was chosen at the beginning of each stamp, a variable `spikes_sum` is initialized to 0, serving to accumulate the spikes generated at each step so their mean can later be computed.

The stamp is kept fixed for `number_of_steps` iterations, and at each iteration, the spikes from the last layer are added to the previous sum. After all steps are completed, the mean is computed by dividing `spikes_sum` by `number_of_steps`, and the result is appended to `spikes_sum_list`.

To reduce the number of elements in the network, decoding is performed directly at the end of the forward pass. This state initializes the list that stores the output of the last layer at each stamp.

Finally, the list is converted into a tensor using `torch.stack`. The results are discussed in the proper section. In this way we get a reduced performance. Our interpretation was related to the idea that the encoding technique was not the best one, and also that the number of steps was really small. Given the long time of training we decide to work especially on the first part.

Having obtained some preliminary results, it was necessary to decide how to proceed. At first, an attempt was made to implement a new encoding technique based on Gaussian encoding.

3.2.6 Gaussian Encoding

The class `GaussianSpikes(sb.nnet.containers.Sequential)` provides an implementation of the **Gaussian encoding** mechanism using two parallel linear layers at the output of the convolutional block, effectively replacing the previously inserted linear layer after flattening the output channels. As previously explained, one branch outputs the mean spike values, while the other, combined with a random variable, generates spike values following a Gaussian distribution.

Additionally, the **SLSTM** cell was adapted to handle different spike types. Instead of using a double `for`-loop (one for time steps and another for spikes), the implementation was simplified by merging the spike dimension with the temporal one through a `view` reshape operation before processing. The spikes from the last layer are stored in a `spikes_list`, and a final tensor is created using `torch.stack`. The temporal dimension of this tensor is then divided into groups of `number_of_steps` via reshaping, and the mean is computed over these groups to obtain the **firing rate**.

However, due to long training times and the limited expected improvement, the focus was shifted toward adding additional spiking layers. The most natural next step would have been to replace the DNN blocks, but this would have required modifying the interface with the **CTC** loss. Since a detailed description of how spiking layers interact with this loss function was not available, the effort was instead directed toward introducing spikes into the convolutional layers.

3.2.7 SCNN

One of the main challenges in implementing convolutional layers for spiking networks is determining how to properly adapt the convolutional operation itself.

The implemented structure includes a **Leaky Integrate-and-Fire (LIF)** layer after each convolutional layer, followed by a max-pooling operation applied to the

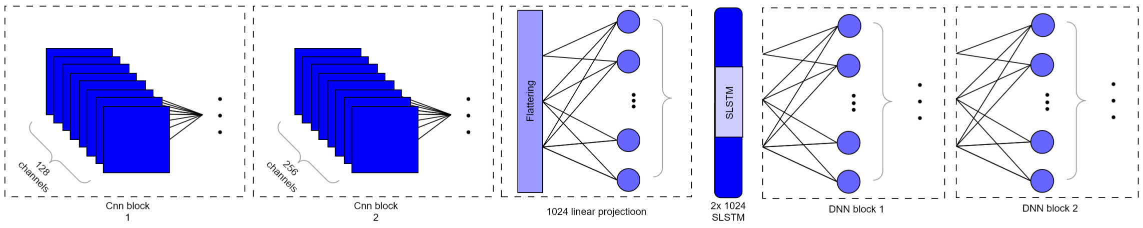


Figure 3.6: Encoder structure with Spiking Convolutional Neural Network (SCNN)

spike frequency, as showed in figure 3.7. As mentioned, each convolutional block is thus paired with a LIF neuron layer.

To define the SCNN group and the encoder final class were rewritten to bypass the `speechbrain` sequential library, providing greater implementation flexibility. For consistency with the base architecture the `sb.nnet.CNN` layers were used instead of standard `nn.Module` layers, even though the only difference lies in the ordering of the channels. Using these modules also facilitates the identification of potential implementation errors, distinguishing them from issues arising from slightly different layer definitions.

Initially, the plan was to maintain the same network structure and apply the encoding techniques explored in previous experiments. The first attempt involved **data duplication**, such that the input entered the convolutional layers already duplicated. However, this approach dramatically increased training time, as discussed in the Results section.

Consequently, the duplication method was discarded. Following the approach suggested in the literature [7], spike encoding was instead delegated to the first spiking layer after the convolution. The encoding technique adopted is the **Convolution + LIF** method. In this approach, non-spiking data are fed directly into the convolutional layers, and it is the subsequent **LIF** layers that convert these activations into spikes. This strategy has the advantage of avoiding an increase in the amount of data to be processed, thereby reducing training time. Another benefit is that both spiking and non-spiking data are handled in the same manner, eliminating the need to implement an explicit conversion interface.

A **padding** strategy was also introduced to ensure that the number of timestamps was always a multiple of the number of processing steps. Padding is handled directly within the decoder, where the number of timestamps is checked: if it does not align, zeros are appended at the end of the tensor. This results in an additional padding stage beyond the standard batch padding. Nevertheless, this solution simplifies the implementation and preserves modularity, since modifying `number_of_steps` automatically updates the padding without further adjustments.

After the padding step, the input tensors undergo an `unsqueeze` operation to

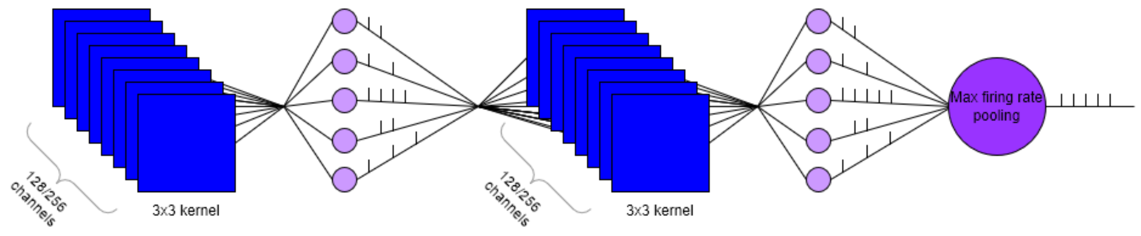


Figure 3.7: SCNN block structure

match the dimensional requirements of convolutional layers, which explicitly expect an additional input dimension. At the output of the convolutional blocks, the procedure follows the same structure used in previous network versions: the feature maps are flattened, passed through a linear projection, followed by the **SLSTM** layer, and finally through the **DNN** blocks. The latter components were not modified and still rely on the `speechbrain` sequential framework.

Regarding the **forward pass** of the SCNN block, the input is first passed through the initial convolutional layer. After each convolution, and before entering the LIF layers, the temporal dimension is divided into groups of `number_of_steps` using a reshape operation. Each group is then processed by the LIF neurons, whose internal states are reinitialized for every batch of spikes, ensuring that each neuron only processes `number_of_steps` spikes at a time. The resulting outputs are stored step by step in a list, converted into tensors, and concatenated for subsequent layer processing.

An additional aspect concerns the management of the **membrane potential** in the LIF neurons. In this implementation, the membrane potential is initialized at the beginning of each group and managed internally by the `SNNLeaky` class through the use of the `None` mechanism.

Pooling

Another issue concerned **pooling**. The simplest way to perform pooling was through temporal encoding; however, this would have required modifying the entire network structure and entering a considerably more complex design space especially since the introduction of SCNN already represented a significant step forward. Other pooling strategies found in the literature were based either on the membrane potential or directly on the firing rate. The latter method was selected, as it appeared the most logical choice: information was already encoded in the firing rate, and, additionally, the use cases reported in related works were more consistent with the present setup.

The `pooling1d` function performs a one-dimensional pooling operation on the spike tensor based on the neurons' firing rate. The input tensor contains spike activity with dimensions $[B, Stamp, F, T, C]$, where B is the batch size, $Stamp$ represents *timestamps*, F corresponds to Mel-frequency coefficients or the already pooled version, T is the temporal dimension within each step, and C denotes the number of channels.

First, the function computes the mean firing rate along the spikes axis, producing a rate map of shape $[B, Stamp, F, C]$. F is then divided into contiguous, non-overlapping groups according to the specified `pooling_size`. Within each group, the element with the highest average firing rate is selected using a `max` operation, and its corresponding index within the original tensor is recorded.

Using these indices, the function retrieves the spike trains associated with the most active neurons from the original spike tensor. The gathered spikes are then rearranged using a permutation operation to maintain a consistent output format. In this way, only the most active spike patterns across Mel coefficients are propagated to the subsequent layers.

We obtained a total improvement, reducing the Word Error Rate (WER) to 18%. It is clear that the discretization introduced by the spikes causes a slight performance loss. Nevertheless, the comparison between activation types shows that the **LIF** model improves performance compared to **ReLU**, demonstrating its ability to capture temporal correlations effectively.

The final architeture has this strctures:

- **ENCODER (Input: [Batch, Time, Mel])**
  - **Convolutional block 1:**
    - Layer Conv2D (128 canali)
    - LIF
    - Layer Conv2D (128 canali)
    - LIF
    - Max_rate Pooling
  - **Convolutional block 2:**
    - Layer Conv2D (256 canali)
    - LIF
    - Layer Conv2D (256 canali)
    - LIF
    - Max_rate Pooling
  - **SLSTM:**
    - Linear projection
    - 2 Layers SLSTM (hidden size 1024)
  - **DNN Block 1:**
    - Linear Layer (512 neurons)
    - Batch Normalization
    - ReLU Activation
    - Dropout
  - **DNN Block 2:**

* Linear Layer (512 neurons)
* Batch Normalization
* ReLU Activation
* Dropout

Chapter 4

Experimental Results and Analysis

For all networks containing spike-based layers, the number of training epochs was doubled to compensate for the slower training speed. Conversely, the number of Connectionist Temporal Classification (CTC) epochs was kept unchanged for all architectures except the last one, where several configurations were explored and more extensive tuning was performed.

The aspects considered in this analysis are the following: accuracy measured through Character Error Rate (CER) and Word Error Rate (WER); network size expressed as the number of parameters; types of errors (Insertion, Deletion, Substitution) expressed as a percentage of the total errors, with the goal of evaluating alignment issues specifically, a higher percentage of substitutions generally indicates better alignment between samples; and the firing rate of the various spiking layers, which provides insight into the internal behavior of the architectures. The parameters common to all the models are represented in table 4.1.

Table 4.1: Common model parameters used across all configurations.

Parameter	**Value**
Neural Network Architecture	
activation	LeakyReLU
cnn_cblocks	2
cnn_channels	128, 256
dropout	0.15
SLSTM_neurons	512
dnn_blocks	2
dnn_neurons	512
Learn_threshold	True
threshold	0.1

Parameter	**Value**
emb_size	128
dec_neurons	1024
output_neurons	1000
Decoding Parameters	
min_decode_ratio	0
max_decode_ratio	1
eos_threshold	1.5
valid_beam_size	80
test_beam_size	80
using_max_attn_shift	True
max_attn_shift	240
temperature	1.25
coverage_penalty	1.5
Audio Processing Parameters	
sample rate	16000
n_fft	400
n_mels	40
Training and Optimization Parameters	
sorting	Ascendent
dynamic batching	False
optimizer	Adadelta
learning rate	1
precision	fp32

4.1 Dataset and vocabulary

Several datasets are available in the literature, but we selected LibriSpeech as our benchmark. The main reasons are its relevance and flexibility. LibriSpeech is in fact one of the most widely used English speech datasets. It contains approximately 1000 hours of English speech, properly segmented and aligned, originating from read audiobooks from the LibriVox project. This makes it suitable even for traditional architectures in which alignment is not automatically handled by the network.

The LibriSpeech training set is divided into three subsets: 100 h clean, 360 h clean wuth good quality recording less accents and more ideal condition, and 500h with worst quality reigistration, different accenta nd more realistic situaotion of use, providing flexibility in both dataset type and scale. This allows experiments to be conducted progressively, with increasing levels of difficulty. The dataset also includes two validation sets (dev-clean and dev-other) and two test sets (test-clean and test-other), containing respectively clean and noisy audio, making it an excellent foundation for future work as well. To reduce training time and enable a faster workflow, we trained on the 100 h clean subset, validated on dev-clean, and tested on test-clean.

The vocabulary was built using *SentencePiece* with the Unigram algorithm, producing 1000 subword units. By default, BrainSpeech supports both 1000 and 5000 subword vocabularies; we selected the smaller one to minimize memory usage and model size. The system also supports the integration of an external language model into the beam search scoring function; however, since our goal was to design a more embedded-oriented system, we deliberately excluded the language model to maintain compactness. For an overview of all the results in tables 4.11 4.12 we resume them all.

4.2 Baseline

CTC	Epoch	WER	CER	Parameters	I	D	S
5	15	10.6%	4.24%	119.9M	12.5%	7.89%	79.61%

Table 4.2: Reference performance metrics for the baseline model.

The baseline model was trained for 15 epochs. During the first 5 epochs the CTC loss was included with a weight of 0.5. This hybrid objective stabilizes alignment learning in the early training stages and improves convergence. As shown by the insertion and deletion statistics, the baseline exhibits the best alignment among all the architectures evaluated. Our baseline achieves a CER of **4.24%** and a WER of **10.61%** on the *test-clean* dataset. To account for the slower convergence observed in spiking architectures, the number of training epochs for subsequent Spiking Neural Network (SNN)-based models was doubled from 15 to 30. We additionally introduced a linear layer after the convolutional block in order to interface more effectively with the SLSTM units, reduce the parameter count, and compensate for the reduced number of recurrent layers. Across experiments, the training settings were kept unchanged, except for minor adjustments to batch size or runtime parameters required by specific architectures.

An interesting observation is that the first five epochs are consistently the fastest

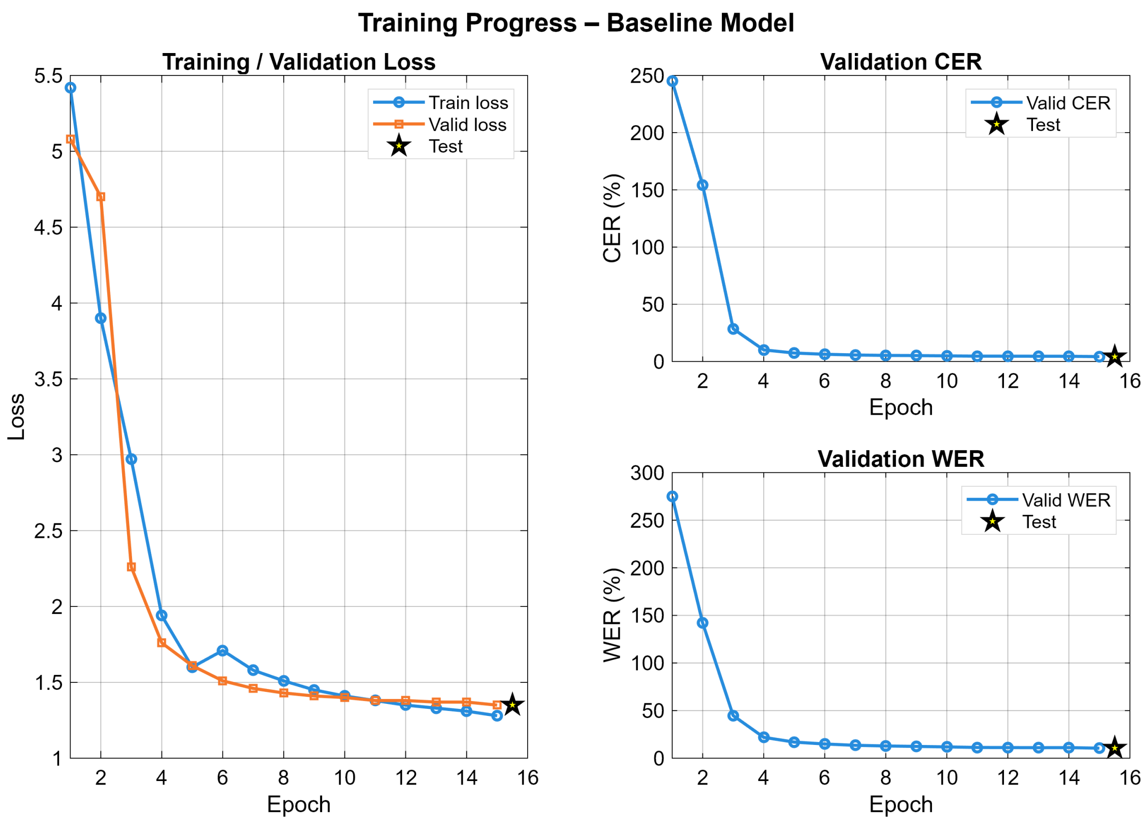


Figure 4.1: Training distribution for the baseline model.

and most effective in terms of performance gain; afterwards, improvements tend to flatten out. Although this behavior is typical in conventional ANNs, as we will show later, it does not necessarily hold for spiking models, where the interaction with the CTC loss may lead to different learning dynamics.

4.3 SLSTM-Only Architecture

Table 4.3: Performance metrics for the Spiking Long-Short Term Memory (SLSTM)-only model.

CTC	Epoch	WER	CER	Parameters	I	D	S
5	30	22.16%	12.35%	50.6M	25.78%	10.41%	63.81%

Table 4.4: Mean firing rate per layer for the SLSTM-only model.

	SLSTM0	SLSTM1
Mean firing rate	34.97%	32.65%

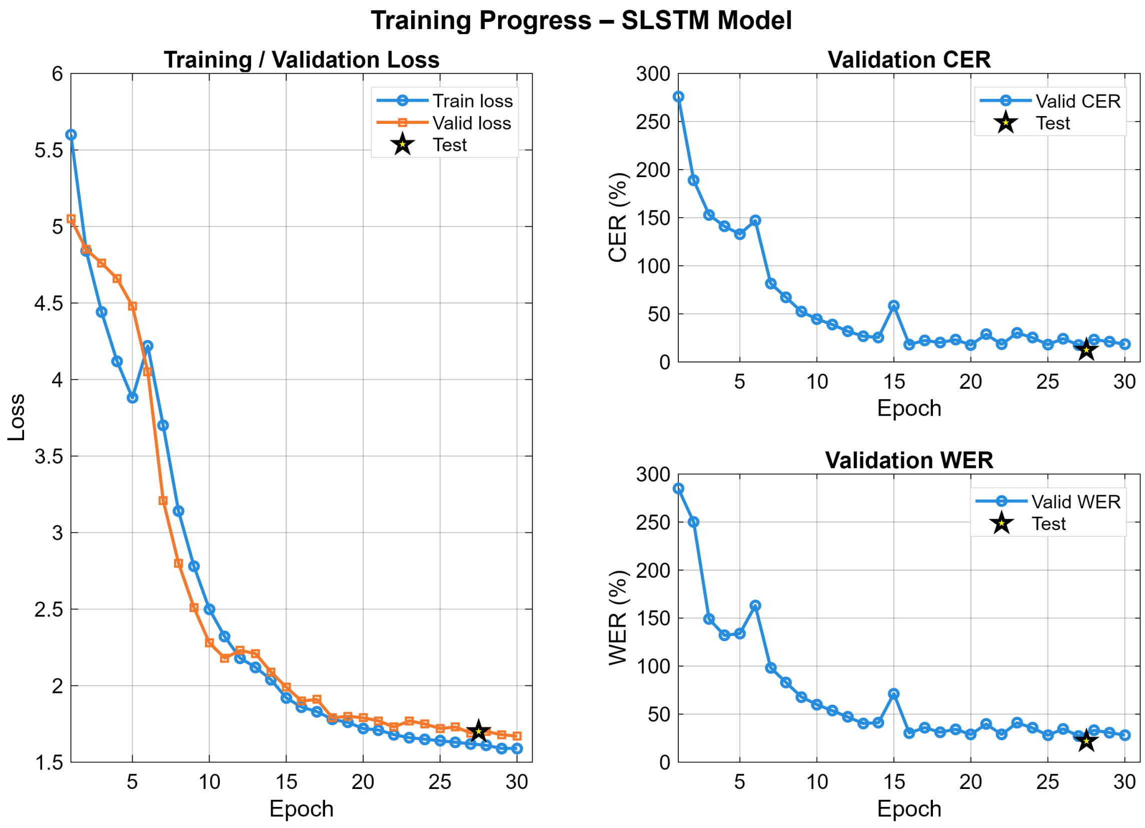


Figure 4.2: Training results for the SLSTM-only model.

The SLSTM-only model was trained using four 44 GB GPUs with a batch size of 6. The thresholds were initialized at 0.1 and set as trainable parameters to avoid information propagation issues and to improve performance. The number_of_steps is 10. A performance degradation was observed, attributable to the reduction in the number of layers, discretization effects, and the simple duplication-based encoding. Compared to the original Artificial Neural Network (ANN) baseline, alignment becomes noticeably worse: substitutions decrease from approximately 80% to 63%, indicating poorer temporal alignment.This is likely related to the discretization introduced by the firing rates. Since only 11 discrete values are available, the Deep Neural Network (DNN) blocks are probably unable to provide the decoder with a sufficiently expressive representation to effectively distinguish between speech segments and silence.

Even with the additional linear layer, the model size is reduced by more than half. The firing rate lies within the typical range for informative spiking activity (30-60%). As we will discuss later, and as previously reported in the literature, this range should not be interpreted as a universal rule, as it depends strongly on architectural and encoding choices. Nonetheless, it remains a useful indicator that information is being propagated through the network.

In this model, the SLSTM layers adopt the *None* reset policy [1]. Under this policy, once a neuron emits a spike, no reset is applied, allowing unrestricted spiking. This increases the risk of uncontrolled firing, information degradation, and excessive energy consumption.

Training dynamics are mostly linear, with occasional negative performance spikes. These fluctuations arise from the discrete nature of spiking activity: even small perturbations can produce large relative increases in firing rate. Another consequence of this discreteness is that validation loss continues to decrease while performance often does not follow proportionally. As we will show later, using spiking signals earlier in the computation pipeline helps reduce this mismatch between loss and behavioral performance.

This architecture is also the only one employing explicit encoding through duplication. As a result, it processes the largest number of samples, increasing the input representation volume without providing clear benefits in terms of representational quality. Duplication not only makes this model the slowest to train but also the slowest at inference time, with the highest memory and energy consumption.

4.4 SCNN Architectures

The Spiking Convolutional Neural Network (SCNN) model analyzed corresponds to the architecture previously described and achieved the best performance among all spiking configurations. To understand the uneven firing-rate distribution observed across layers, it is important to recall the implications of the *None* reset policy.

In this context, the behavior becomes advantageous: given the network depth and the presence of many zero-valued convolutional outputs, information does not degrade and can instead be reinforced by layers that exhibit higher firing rates.

When four SLSTM layers are used, this effect becomes more pronounced. Although a higher firing rate could in principle enrich information flow, it does not result in improved performance; on the contrary, performance slightly worsens. Increasing the number of SLSTM layers leads to more homogeneous firing rates across layers, reducing the risk of imbalances, similarly to what was observed in the SLSTM-only architecture. Moreover, we can also observe that SLSTM layers generally exhibit higher firing rates than Leaky Integrate-and-Fire (LIF) layers, which instead implement a soft reset mechanism. This reset lowers the average firing rate of each neuron and provides greater energy savings, particularly in the layers that are more energy-demanding.

We also increased the number of CTC epochs, but performance deteriorated, while error distributions remained essentially unchanged. Substitutions consistently stabilized around 70%, suggesting that neither the number of SLSTM layers nor the CTC duration substantially affects alignment quality.

Setting the number of CTC epochs to zero caused the training process to stagnate entirely. This indicates that, despite appearing non-essential or even destabilizing when inspecting training curves the CTC loss is in practice necessary for the model to learn meaningful alignments. Another noteworthy observation is that, except for the epochs where CTC is active, training is considerably more linear, particularly toward the final epochs, moreover, the relationship between validation loss and performance becomes more stable, with fewer oscillations. This supports the idea that uniformity in data representation (i.e., using spikes throughout the network) helps the training process. In contrast, the CTC loss remains poorly correlated with performance.

All 2-layer SLSTM architectures were trained on a single 82 GB GPU with batch size 36; the 4-layer variants were also trained on a single 82 GB GPU but with batch size 30. The thresholds were initialized at 0.1 and set as trainable parameters to avoid information propagation issues and to improve performance for both kind of layers. The number_of_steps for all architectures is 10. Compared to the SLSTM-only configuration, the SCNN architectures exhibit improved alignment, suggesting that spiking convolutional encode temporal structure more effectively. Alignment quality still does not match the baseline ANN, but model size especially for the 2-layer version is reduced to roughly one quarter of the original.

4.4.1 SCNN (5 CTC, 4 SLSTM Layers)

The results obtained with the 4-layer SLSTM version are very similar to those of the 2-layer variant trained with the same number of CTC epochs: both WER and CER differ by less than 1%. Training tends to be more stable, and firing rates are higher yet still within the informative range. In particular, firing rates are very stable, showing similar values across same layer types. This architecture therefore represents a reasonable candidate for follow-up studies; however, at present, the performance improvement does not justify the increased number of SLSTM layers, which results in longer training time and a larger model. As seen earlier, adding more spiking recurrent layers worsens the proportionality between performance and CTC behavior, with the worst mismatch occurring when the number of CTC epochs is increased.

CTC	Epoch	WER	CER	Parameters	I	D	S
5	30	19.66%	10.11%	50.6M	18.45%	11.17%	70.37%

Table 4.5: Performance metrics of the SCNN model with 4 SLSTM layers.

	LIF0	LIF1	LIF2	LIF3
Mean firing rate	20.85%	15.93%	18.07%	14.45%

	SLSTM0	SLSTM1	SLSTM2	SLSTM3
Mean firing rate	31.32%	27.60%	20.31%	48.05%

Table 4.6: Mean firing rates for the SCNN model with 4 SLSTM layers.

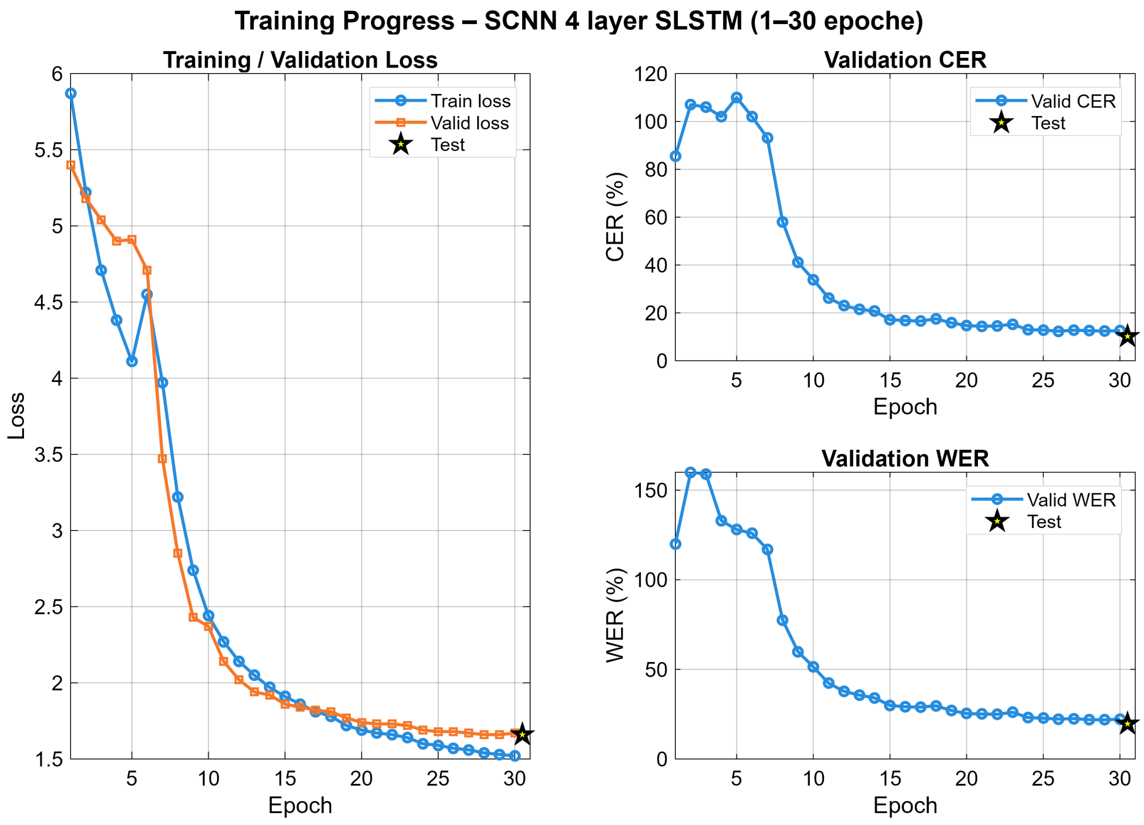


Figure 4.3: Training curve for the SCNN model with 4 SLSTM layers.

4.4.2 SCNN (5 CTC, 2 SLSTM Layers)

This architecture turned out to be the best-performing one, as it is both the smallest and the most accurate. In the following sections, we attempt to train it for 10 additional epochs to assess its potential for further improvement. Its main limitation lies in the imbalance of firing rates within the architecture: in particular, the LIF3 layer shows a notably low firing rate, which may compromise information transmission to the SLSTM layers. Even in this case although less severely than in the other architectures a misalignment between performance and loss can be observed during the CTC epochs.

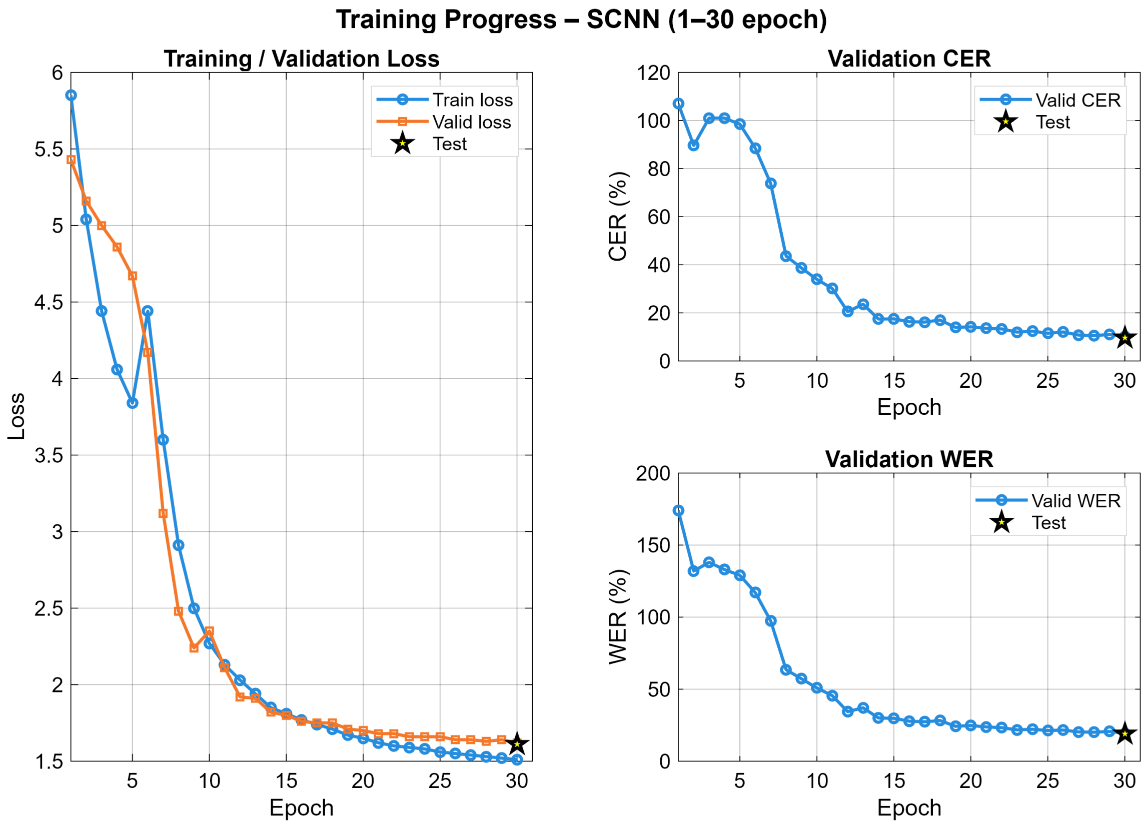


Figure 4.4: Training curve for the 2-layer SCNN architecture.

	LIF0	LIF1	LIF2	LIF3	SLSTM0	SLSTM1
Mean firing rate	12.66%	11.78%	11.83%	8.72%	21.01%	46.19%

Table 4.7: Mean firing rates for the 2-layer SCNN architecture.

CTC	Epoch	WER	CER	Parameters	I	D	S
5	30	18.95%	9.61%	33.8M	16.58%	13.94%	69.48%

Figure 4.5: Performance metrics for the 2-layer SCNN architecture.

4.4.3 SCNN (10 CTC, 2 SLSTM Layers)

Increasing the number of CTC epochs worsens performance without improving alignment. Although the loss continues to decrease, performance remains uncorrelated with it and tends to improve only once CTC supervision has ended. Even then, results do not match those of the model trained with 5 CTC epochs. Firing-rate distributions remain fundamentally similar to the 5-epoch version, with slightly higher activity in the convolutional layers. This again supports the observation that firing rate is primarily determined by the architecture rather than by training dynamics at least when using learnable thresholds.

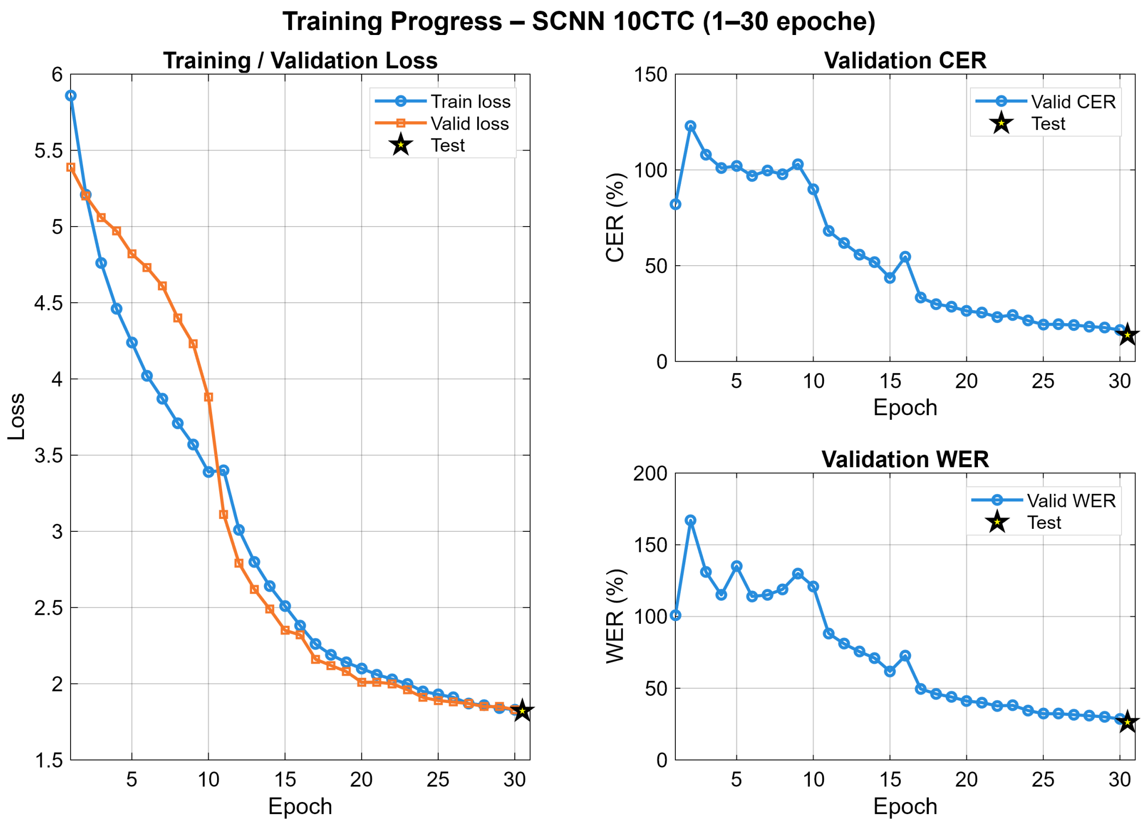


Figure 4.6: Training curve for the SCNN with 2 SLSTM layers and 10 CTC steps.

CTC	Epoch	WER	CER	Parameters	I	D	S
10	30	26.15%	13.88%	33.8M	17.04%	12.12%	70.84%

Table 4.8: Performance metrics for the SCNN with 2 SLSTM layers and 10 CTC steps.

	LIF0	LIF1	LIF2	LIF3	SLSTM0	SLSTM1
Mean firing rate	25.73%	18.13%	15.43%	9.35%	9.27%	42.85%

Figure 4.7: Mean firing rates for the SCNN with 2 SLSTM layers and 10 CTC steps.

4.4.4 SCNN (5 CTC, 2 SLSTM Layers, 40 Epochs)

To further improve performance, we increased the total number of training epochs from 30 to 40 for the 2-layer architecture. This confirms that 30 epochs do not represent the maximum useful training duration, although they provide a solid basis for comparison. Improvements are present but remain modest: WER decreases by less than 1%, while CER and alignment (as seen from the substitution rate) improve by just over 1%. Firing rates increase slightly, suggesting that the network still suffers from low activity levels in some layers. The rise is most noticeable in the early and late layers, indicating where certain imbalances emerge. In particular, the LIF3 layer increases its firing rate by only 0.2%.

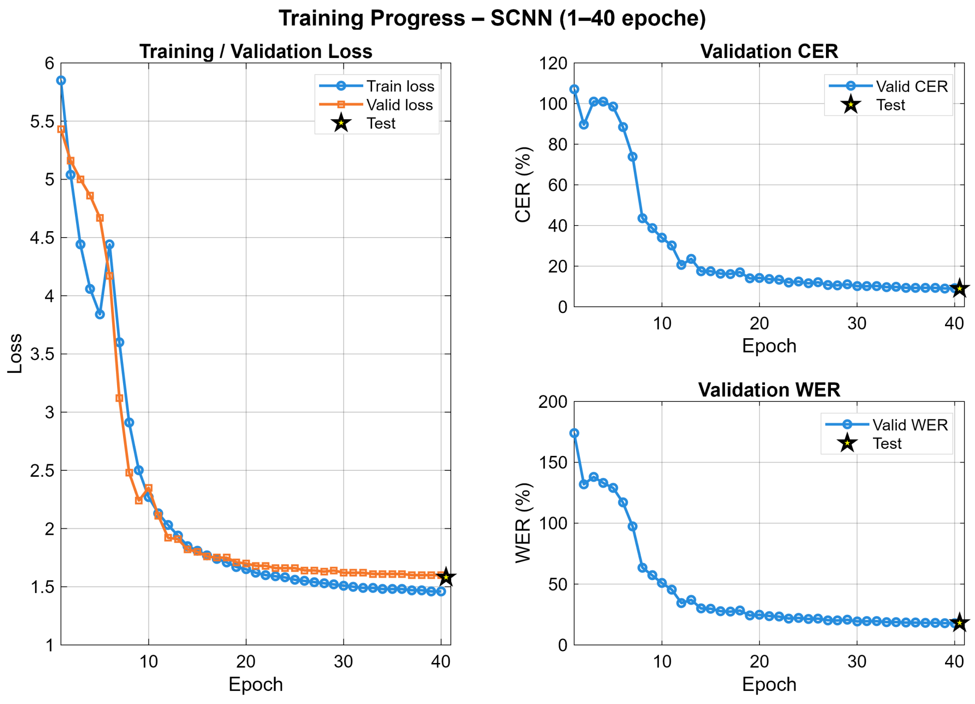


Figure 4.8: Training curve for the SCNN model trained for 40 epochs.

CTC	Epoch	WER	CER	Parameters	I	D	S
5	40	18.03%	8.88%	33.8M	15.73%	13.03%	71.25%

Table 4.9: Performance metrics for the SCNN model trained for 40 epochs.

	LIF0	LIF1	LIF2	LIF3	SLSTM0	SLSTM1
Mean firing rate	13.29%	12.27%	12.17%	8.92%	21.46%	46.12%

Table 4.10: Mean firing rates for the SCNN model trained for 40 epochs.

Overall, the 2-layer SCNN architecture provides the best: it is the smallest model, achieves the highest accuracy, and maintains lower firing rates implying reduced energy consumption particularly in the convolutional stages, which are typically the most energy consuming components.

Model	CTC	Epochs	WER	CER	Params	I	D	S
Baseline	5	15	10.6%	4.24%	119.9M	12.5%	7.89%	79.61%
SLSTM	5	30	22.16%	12.35%	50.6M	25.78%	10.41%	63.81%
SCNN4	5	30	19.66%	10.11%	50.6M	18.45%	11.17%	70.37%
SCNN2	5	30	18.95%	9.61%	33.8M	16.58%	13.94%	69.48%
SCNN2	10	30	26.15%	13.88%	33.8M	17.04%	12.12%	70.84%
SCNN2	5	40	18.03%	8.88%	33.8M	15.73%	13.03%	71.25%

Table 4.11: Comparison of training, performance, and error rates across analyzed models.

Model	LIF0	LIF1	LIF2	LIF3	SL0	SL1	SL2	SL3
Baseline	-	-	-	-	-	-	-	-
SLSTM	-	-	-	-	34.97%	32.65%	-	-
SCNN4	20.85%	15.93%	18.07%	14.45%	31.32%	27.60%	20.31%	48.05%
SCNN2	12.66%	11.78%	11.83%	8.72%	21.01%	46.19%	-	-
SCNN2	25.73%	18.13%	15.43%	9.35%	9.27%	42.85%	-	-
SCNN2	13.29%	12.27%	12.17%	8.92%	21.46%	46.12%	-	-

Table 4.12: Comparison of the mean firing rates per layer across different models.

4.5 CNN–LIF Implementations and Timing Analysis

To understand which parameters influence the training speed, we conducted a study on how different implementations of the SCNN architecture affect performance. Since the decoder remains identical across all variants, our analysis focuses exclusively on the encoder. For the implementations in which only the convolutional block is showed, we report the results assuming that the subsequent layers follow in sequential way.

4.5.1 Implementations with Duplication

All implementations use two convolutional layers per timestep followed by a LIF activation, differing in how operations are organized. Notation: B=batch size, S=sequence length, T=number of timesteps, F=features, C=channels.

Implementation 1

The first implementation adopts an encoding with duplication. Each spike of every spike train is passed through the entire convolutional block, except for the pooling stage. All outputs are collected until the whole spike train has been processed. At that point, pooling over the firing rate can be performed. This implementation is also the closest to the original one: each spiking convolution processes spikes belonging to different timestamps.

The spiking convolutional block therefore has the following timing behaviour:

$$T_{BSCNN} = T \cdot 2(t_{conv} + t_{LIF}) + t_{pool}.$$

In addition, the SLSTM block must be considered. Since we have T spikes per timestamp, the number of data processed by each SLSTM layer is $S \cdot T$, which leads to:

$$T_{BSLSTM} = 2ST\, t_{SLSTM}.$$

The total encoder time is therefore:

$$T_{ENC} = 4T(t_{conv} + t_{LIF}) + 2t_{pool} + 2ST\, t_{SLSTM} + T_{lin} + 2T_{DNN}.$$

It is straightforward to see that the dominant term is the one proportional to S, as $S \gg$ the other components.

SCNN block (pseudocode)

```
sp = []
for t in range(T):
    conv1 = conv(x[:, :, t, :, :])
    sp1 = LIF(conv1)
    conv2 = conv(sp1)
    sp2 = LIF(conv2)
    sp.append(sp2)
sp = pool(sp)
return sp
```

Implementation 2

Implementation 2 aggregates all spikes along the temporal dimension before performing the convolution. After the convolution, a reshape is applied to recover the temporal structure into groups of `number_of_steps`. An empty list is allocated, and each group of steps is processed in parallel by a LIF neuron. Each spike is stored in the list, which is then converted back into a tensor via `stack`. If another convolutional layer follows, spikes are re-aggregated along the temporal dimension; otherwise, they are directly passed to the pooling function. The rest of the encoder is identical to the previous case. Reshape operations can be considered negligible in time, especially when applied to contiguous data.

The timing of the SCNN block is thus:

$$T_{BSCNN} = 2T\,t_{LIF} + 2t_{conv} + t_{pool}.$$

The encoder runtime becomes:

$$T_{ENC} = 4T\,t_{LIF} + 4t_{conv} + 2t_{pool} + 2ST\,t_{SLSTM} + T_{lin} + T_{DNN}.$$

In this way, the convolution has been parallelised, and its runtime no longer depends on T, which is a significant advantage; however, it does not solve the SLSTM bottleneck.

As we will show, this implementation is the fastest among those involving convolution, even though it does not preserve the original network structure.

SCNN block (pseudocode)

```
x = x.reshape(B, S*T, F, C)
conv1 = conv(x)
conv1 = conv1.reshape(B, S, T, F, C)

sp1_list = []
for t in range(T):
    sp1_list.append(LIF(conv1[:, :, t, :, :]))

sp1_list = sp1_list.reshape(B, S*T, F, C)
#seciond layer ...
sp = pool(sp2_list)
return sp
```

Implementation 3 (No Pooling, No Linear Layer)

The third implementation evaluates whether processing spikes one at a time, with minimal aggregation, can offer any advantage. The only aggregation required is that inherent to the SLSTM layers.

We start from duplicated data along a dimension independent of time. Each spike is processed by convolution following the style of Implementation 1, meaning each spike corresponds to a different timestamp.

Skipping the pooling layer allows both convolutional blocks to be combined into a single timing expression:

$$T_{Bmod} = 4T(t_{conv} + t_{LIF}).$$

To maintain temporal consistency, the output for the entire spike train still needs to be collected. Thus, the total encoder time is:

$$T_{ENC} = 4T(t_{conv} + t_{LIF}) + 2TS\, t_{SLSTM} + 2T_{DNN}.$$

This implementation is likely the least efficient, as the convolution time still depends on T, and the only speedup compared to Implementation 1 results from the removal of some layers. Again, the SLSTM layer remains the bottleneck.

CNN + SLSTM (pseudocode)

```
for t in range(T):
    conv1 = conv(x[:, :, t, :, :])
    sp1 = LIF(conv1)
    # repeated to obtain sp4
sp4 = sp4.flatten()

for s in range(S):
    sps1 = SLSTM(sp4[:, s, :])
    sps2 = SLSTM(sps1)

# followed by DNN
```

4.5.2 Implementation without Duplication

The implementation without duplication is the fastest. Here, no additional spike dimension is introduced, and no temporal merging is required. Padding is applied to ensure that the number of timestamps is a multiple of `number_of_steps`. A standard convolution is performed, then a reshape divides the temporal axis into groups of `number_of_steps`. Each group is processed in parallel by a LIF neuron. The outputs are collected and reshaped back into a tensor.

The convolutional block runtime is:

$$T_{BSCNN} = 2T\, t_{LIF} + 2t_{conv} + t_{pool},$$

which matches the expression from Implementation 2.

At the end of the convolutional block, no further aggregation is needed, and the number of inputs to the SLSTM layers is simply S. Hence:

$$T_{BSLSTM} = 2S\, t_{SLSTM}.$$

The resulting encoder time is therefore:

$$T_{ENC} = 2S\, t_{SLSTM} + 4T\, t_{LIF} + 4t_{conv} + 2t_{pool} + T_{lin} + T_{DNN}.$$

This provides two advantages: (1) the SLSTM bottleneck remains, but reduced by a factor of `number_of_steps`; (2) the bottleneck no longer depends on the number of spikes unless $T > S$, enabling the exploration of more computationally expensive models. This is the final implementation adopted in our code.

```
x = B, S, F, C
x = padding(x)  # requires S % T == 0
conv1 = conv(x)
conv1 = conv1.reshape(B, S//T, T, F, C)

# LIF per timestep
...
conv2 = conv(sp1_list)
conv2 = conv2.reshape(B, S//T, T, F, C)
...
sp = pool(sp2_list)
return sp
```

4.5.3 Standard Network (No Linear Layer)

We now analyse the encoder time of the classical architecture, neglecting the activation function time. This is naturally the fastest case, since the absence of spiking layers allows maximum parallelisation. Moreover, without duplication, the Long-Short Term Memory (LSTM) bottleneck depends on S rather than $S \cdot T$.

The Convolutional Neural Network (CNN) block runtime is:

$$T_{BCNN} = 2t_{conv} + t_{pool}.$$

Thus, the total encoder time is:

$$T_{ENC} = 4t_{conv} + 2t_{pool} + T_{DNN}.$$

4.5.4 Performance Observations

- Among the duplicated versions, Implementation 2 is the most efficient since convolutions are moved outside the temporal loop and the convolution cost is paid only twice.
- The main bottleneck is the SLSTM: the term $2ST\, t_{SLSTM}$ dominates all other contributions. This also explains why the non-duplicated implementation, with coefficient $2S$, is significantly faster.
- Pooling has a marginal impact on training time compared to SLSTM and convolution layers.

4.6 Testing Code

To evaluate the neural networks, we implemented 2 scripts (one for architecture) based on forward hooks. Hooks are small functions attached to spike layers that accumulate spikes at runtime. The first 100 samples of the test set are evaluated in random order, and the average firing rate is computed for each layer.

At the beginning of the script, several compatibility adjustments are included to ensure that models load correctly despite architectural changes and updated layer naming conventions. The script also prints the full model structure, serving as a useful debugging tool for identifying potential inconsistencies.

Chapter 5

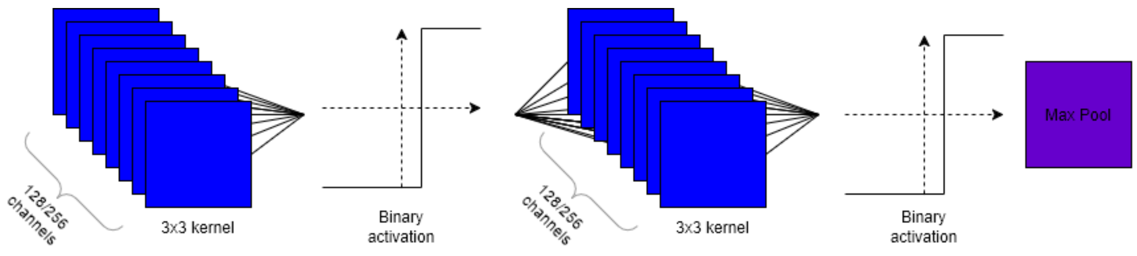
Future Work and Insights

During the thesis work, several training-related challenges and many possible experimentation paths emerged. This led to exploring multiple directions, although most of them could only be evaluated at a preliminary level. Despite this, some of these ideas remain highly relevant and offer excellent opportunities for future work, especially in light of the results obtained. In this chapter, we briefly present the most significant ones, discussing where present the code already implemented and the work carried out, and providing ideas for their further development.

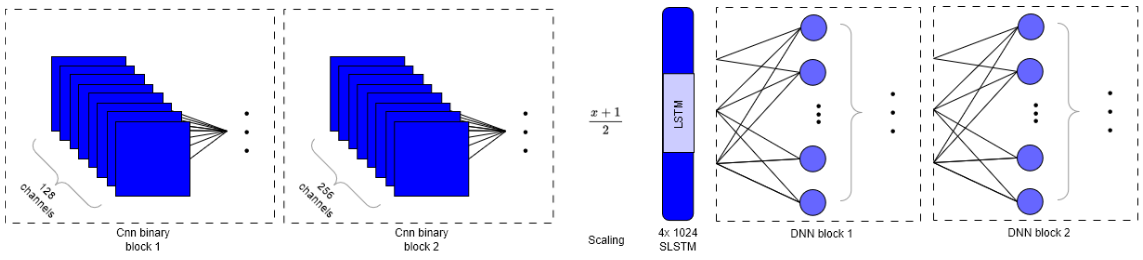
5.1 Binary convolution

During the research on spike-based convolution, a particularly interesting direction emerged involving convolutions on binary networks. Binary networks rely on binary activations and/or weights to reduce memory footprint or improve energy efficiency. Compared to spike convolutions, they offer the advantage of being much faster, as they are fully parallelizable like standard convolutions. The idea arose from the need to represent the data in binary form without relying on an encoding that would reduce training efficiency. Indeed, the use of encoding schemes often requires either an increase in data volume, such as through duplication, or the addition of further sequential components. Moreover, the use of binary layers is frequently associated with energy and memory savings, making their adoption fully consistent with the philosophy of the project.

In our preliminary exploration, we considered networks using binary activations only. Convolutions are performed normally and then binarized through a step activation function as in figure 5.1a 5.1b. From an implementation standpoint, this simply requires replacing ReLU with a Heaviside/sign function with a learnable threshold. During backpropagation, as with spike activations, the step function is replaced with a surrogate gradient.



(a) Convolutional binary block



(b) Convolutional binary encoder

Figure 5.1: Binary convolutional components: (a) block structure and (b) encoder structure.

Binary activations can be either in the range (0, 1) or (-1, 1), with the latter being generally preferred because symmetric ranges have shown better training stability [5]. The advantage for spike-based architectures is that this allows data to be translated into an efficient spike compatible encoding without losing performance or training speed. Before interfacing with spiking layers, activations in (-1, 1) can be easily mapped into (0, 1) by rescaling. This approach also aligns well with goals of energy and memory savings in embedded systems.

In our project, we implemented an architecture composed of a binary-activation block, followed by two Spiking Long-Short Term Memory (SLSTM) layers and two Deep Neural Network (DNN) blocks. The activation function is custom, since PyTorch does not provide a Sign activation with learnable threshold and surrogate gradient. The implementation is divided into two classes: `BinaryActivationWithThreshold`, which implements the forward and backward passes, and `LearnableBinaryActivation`, which defines the actual activation module used in the network. During the forward pass, both the input and the threshold are stored in the context object (`ctx`) used by PyTorch for autograd, and the output is computed. The backward pass implements a Straight-Through Estimator (STE): the derivative is set to 1 when the input minus the threshold lies within {-1, 1}, and 0 otherwise. The gradient for the threshold is computed as the negative sum of the gradients over the relevant dimensions. The second class implments the channel-wise thresholds.

The decoder used in this version of the model was kept identical to the original one. Training proved to be very fast, and promising results appeared already within the first iterations on the training dataset. The project was eventually discontinued as it fell outside the objectives of this thesis. The focus of the thesis is to investigate the use of spiking neurons in speech recognition; therefore, a spike-based convolution was implemented in the network, as it is more closely aligned with this objective. Nevertheless, this direction remains highly interesting for future work, given the preliminary results obtained and the strong technical and conceptual affinity between binary networks and spiking networks.

5.2 Spike-Based Encoder

A very natural research direction is the development of a fully spike-based encoder, replacing the DNN blocks with fully connected Leaky Integrate-and-Fire (LIF) layers. Having a fully spiking encoder would be advantageous in terms of modularity, as it would provide a ready to use, entirely spiking component that could be integrated into different Sequence-to-Sequence (seq2seq) architectures or used to develop fully spiking models. Moreover, it would further increase the network's energy efficiency and compactness. However, this introduces several challenges, such as the interaction between spikes and the Connectionist Temporal Classification (CTC) loss, and the presence of a discretized context vector.

Depending on the design choices, the context vector can either contain spikes forcing the decoder to perform the decoding or it can contain already-decoded values (in our case, the firing rate). This introduces a clear discretization problem, since each entry of the context vector can take at most `number_of_steps+1` distinct values. In our prototype, we experimented with the second approach. We attempted removing the CTC loss from training, as well as increasing the CTC unrolling cycles, but in all cases the training progress was extremely poor and the network failed to learn. This suggests that some Artificial Neural Network (ANN) layers are still fundamental in these hybrid architectures.

This means that the success of a fully spiking encoder requires further investigation into appropriate training methods and the optimal number of simulation steps.Our project includes prototype code where these modifications were implemented, serving as a basis for future developments.

5.3 Transducer and ContextNet Approaches

Parallel investigations were also conducted, including preliminary experiments with Transducer-based architectures starting from ContextNet. As previously mentioned, ContextNet is a high-performance Transducer network, especially considering its simplicity, as it does not rely on attention mechanisms an uncommon characteristic in the field of speech recognition. This makes it an excellent starting point for exploring the integration of spiking networks into a more complex architecture and for extending our work in this direction. Moreover, it does not require training with multiple loss functions, thereby avoiding the limited compatibility between spikes and CTC. In our study, we chose an architecture with fewer components (two in the seq2seq model versus three in the transducer model), so that modifying a single component would have a more significant impact on the behavior of the entire network, making the evaluation of spiking neurons' performance within the system simpler and more meaningful. Furthermore, the presence of an additional training objective allowed us to experiment with one more parameter during training, thereby broadening our research scope.

Bibliography

[1] Daniel Auge et al. “A Survey of Encoding Techniques for Signal Processing in Spiking Neural Networks”. In: *Neural Processing Letters* 53.6 (Dec. 1, 2021), pp. 4693–4710. ISSN: 1573-773X. DOI: 10.1007/s11063-021-10562-2. URL: https://doi.org/10.1007/s11063-021-10562-2 (visited on 04/06/2025).

[2] Alexandre Bittar and Philip N. Garner. *Surrogate Gradient Spiking Neural Networks as Encoders for Large Vocabulary Continuous Speech Recognition.* Feb. 16, 2023. DOI: 10.48550/arXiv.2212.01187. arXiv: 2212.01187[cs]. URL: http://arxiv.org/abs/2212.01187 (visited on 09/17/2025).

[3] Sangchun Ha (Patrick). *upskyy/ContextNet.* original-date: 2021-06-20T09:37:44Z. Apr. 15, 2025. URL: https://github.com/upskyy/ContextNet (visited on 04/22/2025).

[4] Saeed Reza Kheradpisheh et al. “STDP-based spiking deep convolutional neural networks for object recognition”. In: *Neural Networks* 99 (Mar. 2018), pp. 56–67. ISSN: 08936080. DOI: 10.1016/j.neunet.2017.12.005. arXiv: 1611.01421[cs]. URL: http://arxiv.org/abs/1611.01421 (visited on 09/19/2025).

[5] Zechun Liu et al. “ReActNet: Towards Precise Binary Neural Network with Generalized Activation Functions”. In: *Computer Vision – ECCV 2020.* Ed. by Andrea Vedaldi et al. Vol. 12359. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 143–159. ISBN: 978-3-030-58567-9 978-3-030-58568-6. DOI: 10.1007/978-3-030-58568-6_9. URL: https://link.springer.com/10.1007/978-3-030-58568-6_9 (visited on 10/13/2025).

[6] Naoya Muramatsu and Hai-Tao Yu. *Combining Spiking Neural Network and Artificial Neural Network for Enhanced Image Classification.* Feb. 28, 2021. DOI: 10.48550/arXiv.2102.10592. arXiv: 2102.10592[cs]. URL: http://arxiv.org/abs/2102.10592 (visited on 09/11/2025).

[7] Thomas Pellegrini, Romain Zimmer, and Timothée Masquelier. *Low-activity supervised convolutional spiking neural networks applied to speech commands recognition.* Nov. 13, 2020. DOI: `10.48550/arXiv.2011.06846`. arXiv: `2011.06846[cs]`. URL: `http://arxiv.org/abs/2011.06846` (visited on 11/08/2025).

[8] Wachirawit Ponghiran and Kaushik Roy. "Spiking Neural Networks with Improved Inherent Recurrence Dynamics for Sequential Learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 36.7 (June 28, 2022), pp. 8001–8008. ISSN: 2374-3468. DOI: `10.1609/aaai.v36i7.20771`. URL: `https://ojs.aaai.org/index.php/AAAI/article/view/20771` (visited on 11/08/2025).

[9] Rohit Prabhavalkar et al. *End-to-End Speech Recognition: A Survey.* Mar. 3, 2023. DOI: `10.48550/arXiv.2303.03329`. arXiv: `2303.03329[eess]`. URL: `http://arxiv.org/abs/2303.03329` (visited on 10/24/2025).

[10] Bernardete Ribeiro et al. "Convolutional Spiking Neural Networks targeting learning and inference in highly imbalanced datasets". In: *Pattern Recognition Letters* 189 (Mar. 1, 2025), pp. 241–247. ISSN: 0167-8655. DOI: `10.1016/j.patrec.2024.08.002`. URL: `https://www.sciencedirect.com/science/article/pii/S0167865524002344` (visited on 09/20/2025).

[11] B. Schrauwen and J. Van Campenhout. "BSA, a fast and accurate spike train encoding scheme". In: *Proceedings of the International Joint Conference on Neural Networks, 2003.* the International Joint Conference on Neural Networks, 2003. Vol. 4. ISSN: 1098-7576. July 2003, 2825–2830 vol.4. DOI: `10.1109/IJCNN.2003.1224019`. URL: `https://ieeexplore.ieee.org/document/1224019` (visited on 04/28/2025).

[12] *snntorch — snntorch 0.9.4 documentation.* URL: `https://snntorch.readthedocs.io/en/latest/snntorch.html` (visited on 11/17/2025).

[13] Hongze Sun et al. *A Synapse-Threshold Synergistic Learning Approach for Spiking Neural Networks.* Apr. 3, 2023. DOI: `10.48550/arXiv.2206.06129`. arXiv: `2206.06129[cs]`. URL: `http://arxiv.org/abs/2206.06129` (visited on 11/25/2025).

[14] Amirhossein Tavanaei and Anthony S. Maida. *Bio-Inspired Spiking Convolutional Neural Network using Layer-wise Sparse Coding and STDP Learning.* June 24, 2017. DOI: `10.48550/arXiv.1611.03000`. arXiv: `1611.03000[cs]`. URL: `http://arxiv.org/abs/1611.03000` (visited on 09/19/2025).

[15] *Tutorial 2 - The Leaky Integrate-and-Fire Neuron — snntorch 0.9.4 documentation.* URL: `https://snntorch.readthedocs.io/en/latest/tutorials/tutorial_2.html#lapicques-lif-neuron-model` (visited on 11/17/2025).

[16] Ashish Vaswani et al. *Attention Is All You Need.* Aug. 2, 2023. DOI: 10.48550/arXiv.1706.03762. arXiv: 1706.03762[cs]. URL: http://arxiv.org/abs/1706.03762 (visited on 10/27/2025).

[17] Jibin Wu et al. "Deep Spiking Neural Networks for Large Vocabulary Automatic Speech Recognition". In: *Frontiers in Neuroscience* 14 (Mar. 17, 2020). Publisher: Frontiers. ISSN: 1662-453X. DOI: 10.3389/fnins.2020.00199. URL: https://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2020.00199/full (visited on 03/19/2025).

[18] Sidi Yaya Arnaud Yarga, Jean Rouat, and Sean U. N. Wood. "Efficient spike encoding algorithms for neuromorphic speech recognition". In: *Proceedings of the International Conference on Neuromorphic Systems 2022.* July 27, 2022, pp. 1–8. DOI: 10.1145/3546790.3546803. arXiv: 2207.07073[cs]. URL: http://arxiv.org/abs/2207.07073 (visited on 09/15/2025).

[19] Zhaokun Zhou et al. "Spiking Transformer with Experts Mixture". In: *Advances in Neural Information Processing Systems* 37 (Dec. 16, 2024), pp. 10036–10059. URL: https://proceedings.neurips.cc/paper_files/paper/2024/hash/137101016144540ed3191dc2b02f09a5-Abstract-Conference.html (visited on 03/19/2025).